

UNIT-I

Unit-1: Introduction to Programming:

Introduction to components of a computer system: disks, primary and secondary memory, processor, operating system, compilers, creating, compiling and executing a program etc., Number systems Introduction to Algorithms: steps to solve logical and numerical problems. Representation of Algorithm, Flowchart/Pseudo code with examples, Program design and structured programming Introduction to C Programming Language: variables (with data types and space requirements), Syntax and Logical Errors in compilation, object and executable code , Operators, expressions and precedence, Expression evaluation, Storage classes (auto, extern, static and register), type conversion, The main method and command line arguments

Bitwise operations: Bitwise AND, OR, XOR and NOT operators

Conditional Branching and Loops: Writing and evaluation of conditionals and consequent branching with if, if-else, switch-case, ternary operator, goto, Iteration with for, while, do-while loops

I/O: Simple input and output with scanf and printf, formatted I/O

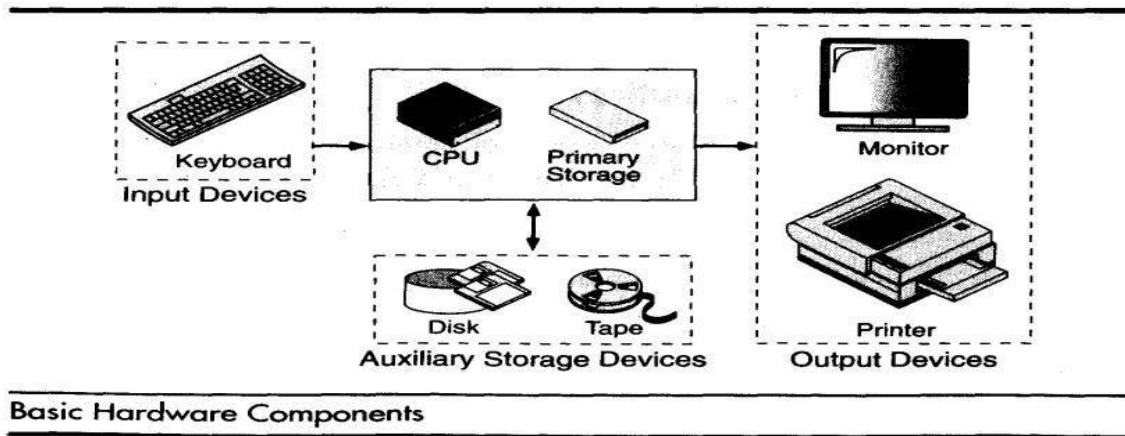
Introduction to Computers:

Computer Systems:

- Computer is a machine made of Electronic devices (switch, capacitor, transistor which known 0's and 1's or on and off) that enable user to enter data (Input), process it (CPU), and store it in a computer memory.
- A computer is a system made of two major components:
- Hardware and software.
- Hardware is the physical equipment which can be touch and feel.
- Software is the collection of programs (instructions) that perform specific task and allow the hardware to do its job.

Computer Hardware

- The hardware component of the computer system consists of five parts:
- Input devices,
- Central processing unit (CPU)
- Primary storage,
- Output devices
- Auxiliary storage devices.



Input device:

Data or instruction are entered into the computer with the help of input devices.

Ex: keyboard, Mouse, Scanner.

Central processing unit (CPU) :

CPU is a computer brain use to perform calculation and other operation.

Output device :

The result given by the computer after processing data is called as output. the output device shows or plays the result after the input has been proceed.

Ex: Monitor, Printer, Speaker are output devices .

Computer Memory:

Computer memory is any physical device capable of storing information temporarily or permanently.

Primary Memory

- Primary memory is computer memory that a computer accesses directly.
- Primary memory is a volatile storage mechanism.
- It may be random access memory (RAM), cache memory or data buses.
- primarily associated with RAM.
- Primary memory is considered faster than secondary memory

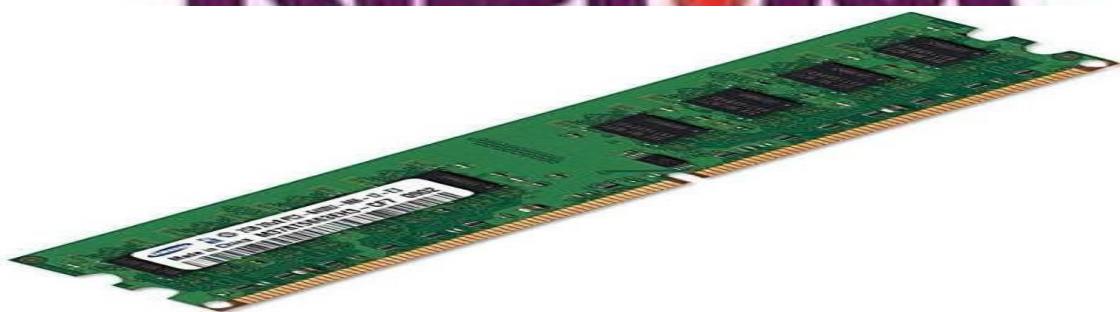


Fig3: Ram

Secondary Storage and Disk

- Secondary storage devices are those devices whose memory is non volatile, meaning, the stored data will be in computer even if the system is turned off.
- Here are a few secondary storage. **Hard Disk, CD RAM, DVD RAM, Pen Drive**
- Secondary storage is also called auxiliary storage.
- Secondary storage is less expensive when compared to primary memory like RAMs.
- The speed of the secondary storage is also lesser than that of primary storage.

Extended Memory Hierarchy



Source: http://www.ts.avnet.com/uk/products_and_solutions/storage/hierarchy.html

Processor:

- A processor is an integrated electronic circuit that performs the calculations that run a computer.
- A processor performs arithmetical, logical, input/output (I/O) and other basic

NRCM



instructions that are passed from an operating system (OS).

Computer Software

- Computer software is divided into two broad categories: system software and application software.

System software:

- System software manages the computer resources. It provides the interface between the hardware and the users.
- System software consists of programs that manage the hardware resources of a computer and perform required information processing tasks.
- These programs are divided into three classes: the operating system, system support, and system development.
- The operating system provides services such as a user interface, file and database access, and interfaces to communication systems such as Internet protocols.
- System support software provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs.
- The language translators convert user programs into machine language for execution, debugging tools to ensure that the programs are error free and computer-assisted software engineering (CASE) systems.

Application software

- Application software, on the other hand, is directly responsible for helping users solve their problems.
- Application software is classified into two classes:
- **General-purpose software**: is purchased from a software developer and can be used for more than one application and application-specific software.
Ex: MS-Office, Adobe Photoshop, Adobe Reader
- **specific purpose software**: use only for specific purpose. Ex: Library software, Banking software, IRTC software.

Computer Languages:

- To write a program for a computer, we must use a computer language. Over the years computer languages have evolved from machine languages to natural languages.

1940's Machine level Languages

1950's Symbolic Languages

1960's High-Level Languages

Machine Languages

- In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language, which is made of streams of 0's and 1's.
- Instructions in machine language must be in streams of 0's and 1's because the internal circuits of a computer are made of switches transistors and other electronic devices that can be in one of two states: off or on. The off state is represented by 0 , the on state is represented by 1.

The only language understood by computer hardware is machine language.

Symbolic Languages:

- In early 1950's Admiral Grace Hopper, A mathematician and naval officer developed the concept of a special computer program that would convert programs into machine language.
- The early programming languages simply mirror to the machine languages using symbols of mnemonics to represent the various machine language instructions because they used symbols, these languages were known as symbolic languages.
- Computer does not understand symbolic language it must be translated to the machine language.
- A special program called assembler translates symbolic code into machine language. Because symbolic languages had to be assembled into machine language they soon became known as assembly languages.
- Symbolic language uses symbols or mnemonics to represent the various, machine language instructions.

High Level Languages:

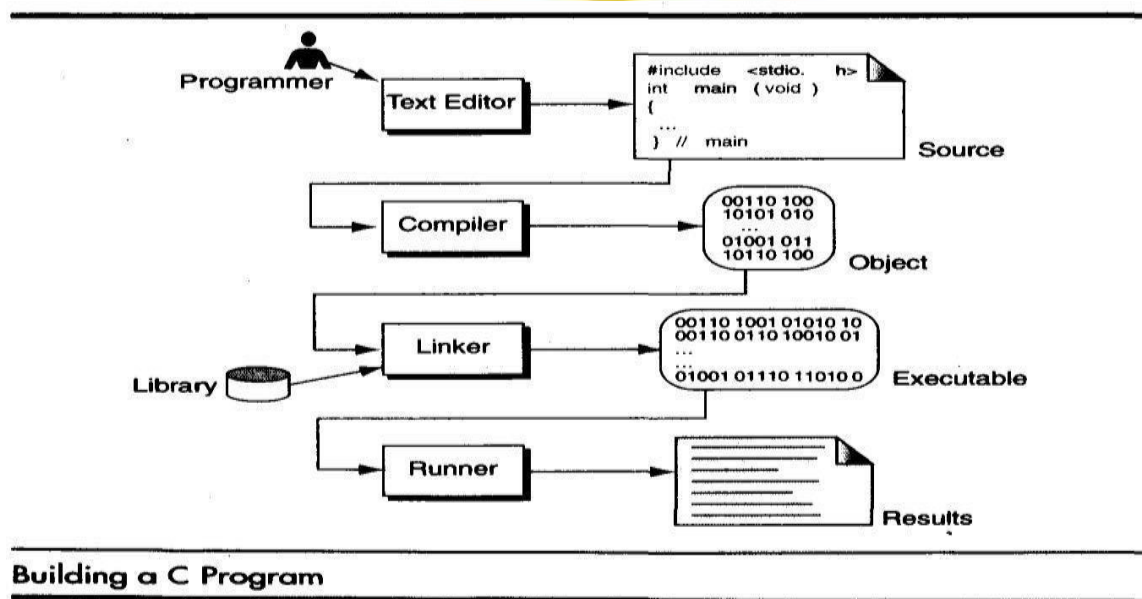
- High level languages are portable to many different computers, allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer.
- High-level languages are designed to relieve the programmer from the details of the assembly language.
- High level languages share one thing with symbolic languages; they must be converted into machine language.
- The process of converting them is known as compilation.

The first widely used high-level languages, **FORTRAN (FORmula TRANslation)** was created by John Backus and an IBM team in 1957;it is still widely used today in scientific and engineering applications. After FORTRAN was **COBOL (Common Business-Oriented Language)**. Admiral Hopper was played a key role in the development of the COBOL Business language. C is a high-level language used for system software and new application code.

Creating and Running Programs:

Computer hardware understands a program only if it is coded in its machine language. It is the job of the programmer to write and test the program. There are four steps in this process:

1. Writing and Editing the program
2. Compiling the program
3. Linking the program with the required library modules
4. Executing the program.



Building a C Program

Writing and Editing Programs

- The software used to write programs is known as a text editor.
- A text editor helps user to enter, change, and store character data.
- The main difference between text processing and program writing is that programs are written using lines of code, while most text processing is done with character and lines.
- After writing a program, we save our file to disk. This file will be input to the compiler; it is known as a source file.

Compiling Programs:

- The code in a source file stored on the disk must be translated into machine language; this is the job of the compiler.
- The 'c' compiler is two separate programs. The preprocessor and the translator.
- The code generated after compilation is called object code.

The preprocessor reads the source code and prepares it for the translator. While preparing the code, it scans for special instructions known as preprocessor commands.

After the preprocessor has prepared the code for compilation, the translator convert the program into machine language and generate the object code that is,not executable because it does not have the required C and other functions included.

Linking Programs:

- A C program is made up of many functions.
- Function can be user defined or predefined
- Predefined function, such as input/output function and mathematical library functions that exist elsewhere and must be attached to our program.
- The linker assembles all of these functions code to object code and then generate final executable program.

Executing Programs:

- Once program has been linked, it is ready for execution.
- To execute a program we use an operating system command, such as run,
- OS use the program called loader to load program from secondary memory into primary memory to execute the program

In a typical program execution, it reads data for processing, either from the user or from a file. After the program processes the data, it prepares the output. At output can be to the user's monitor or to a file. When the program has finished its job, it tells the operating system, which then removes the program from memory.



Interpreter Vs Compiler

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.

It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Errors in C

- Error is an illegal operation performed by the user which results in abnormal working of the program.
- Programming errors often remain undetected until the program is compiled or executed.
- Some of the errors restrict the program from getting compiled or executed. Thus errors should be removed before compiling and executing.

Types of Error

1. Syntax errors:

- Errors that occur when you violate the rules of writing C syntax are known as syntax errors.
- This compiler error indicates something that must be fixed before the code can be compiled.
- All these errors are detected by compiler and thus are known as compile-time errors.
- Most frequent syntax errors are:
Missing Parenthesis ()

Printing the value of variable without declaring it
Missing semicolon like this:

2. Run-time Errors :

- Errors which occur during program execution (run-time) after successful compilation are called run-time errors.
- One of the most common run-time error is division by zero also known as Division error.
- These types of error are hard to find as the compiler doesn't point to the line at which the error occurs.

3.Logical Errors :

- On compilation and execution of a program, desired output is not obtained when certain input values are given.
- These types of errors which provide incorrect output but appears to be error free are called logical errors.
- These are one of the most common errors done by beginners of programming.

Number System:

When we type some letters or words, the computer translates them in numbers as computers can understand only numbers. A computer can understand the positional number system where there are only a few symbols called digits and these symbols represent different values depending on the position they occupy in the number.

The value of each digit in a number can be determined using –

- The digit
- The position of the digit in the number
- The base of the number system (where the base is defined as the total number of digits available in the number system)

Decimal Number System

- The number system that we use in our day-to-day life is the decimal number system.
- Decimal number system has base 10 as it uses 10 digits from 0 to 9.
- In decimal number system, the successive positions to the left of the decimal point represent units, tens, hundreds, thousands, and so on.

Example: the decimal number 1234 consists of the digit 4 in the units position, 3 in the tens position, 2 in the hundreds position, and 1 in the thousands position. Its value can be written as

- $(1 \times 1000) + (2 \times 100) + (3 \times 10) + (4 \times 1)$
- $(1 \times 10^3) + (2 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$
- $1000 + 200 + 30 + 4$
- 1234

Other Number System:

1. Binary Number System: Base 2. Digits used : 0, 1

2. Octal Number System: Base 8. Digits used : 0 to 7

3. Hexa Decimal Number System: Base 16. Digits used: 0 to 9, Letters used : A- F

Representation of Algorithm ,Flowchart /Pseudocode with Examples:

Algorithm:

Algorithm is a finite set of instructions that , if followed accomplishes a particular task. The same problem can be solved with different methods. So, to solve a problem different algorithms, may be accomplished. Algorithm may vary in time, space utilized.

User writes algorithm in his / her own language. So, it cannot be executed on computer. Algorithm should be in sufficient detail that it can be easily translated into any language.

Characteristics or properties of Algorithm:

- 1.Input :** Zero or more quantities are externally supplied.
- 2.Output:** At least one quantity is produced.
- 3.Definiteness:** Each instruction is clear and unambiguous. Ex: Add B or C to A.
- 4. Finiteness:** Algorithm should terminate after finite number of steps.
- 5. Effectiveness:** Every Instruction must be basic.

Advantages of Algorithms:

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

Disadvantages of Algorithms:

- In large algorithms the flow of program control becomes difficult to track.
- Algorithms lack visual representation of programming constructs like flowcharts.
- Understanding the logic becomes relatively difficult.

Examples:

Example1 :Add two numbers.

- Step 1: Start
- Step 2: Read 2 numbers as A and B
- Step 3: Add numbers A and B and store result in C
- Step 4 :Display C
- Step 5: Stop

Example2: Average of 3 numbers.

- Step 1. Start
- Step 2. Read the numbers a , b , c
- Step 3. Compute the sum and divide by 3
- Step 4. Store the result in variable d
- Step 5. Print value of d

Step 6.End

Example3: Average of n inputted numbers.

- Step 1.Start
- Step 2.Read the number n
- Step 3.Initialize i to Zero
- Step 4.Initialize sum to zero
- Step 5.If i is greater than n
- Step 6.Read a
- Step 7.Add a to sum and goto step 5.
- Step 9.Dividesum by n and store the result in avg
- Step 10.Print value ofAvg
- Step 11.End

Flowchart:

A flowchart is a Visual representation of the sequence of steps for solving a problem. A flowchart is a set of symbols that indicate various operations in the program. For every process, there is a corresponding symbol in the flowchart. A pictorial representation of a textual algorithm is done using a flowchart. A flowchart gives a pictorial representation of an algorithm.

- The first flowchart is made by John Von Neumann in 1945.
- It is a symbolic diagram of operations sequence, data flow, control flow and processing logic in information processing.
- The symbols used are simple and easy to learn.
- It is a very helpful tool for programmers and beginners.

Purpose of a Flowchart :

- Provides communication.
- Provides an overview.
- Shows all elements and their relationships.
- Quick method of showing program flow.
- Checks program logic.
- Facilitates coding.
- Provides program revision.
- Provides program documentation.

Advantages of a Flowchart :

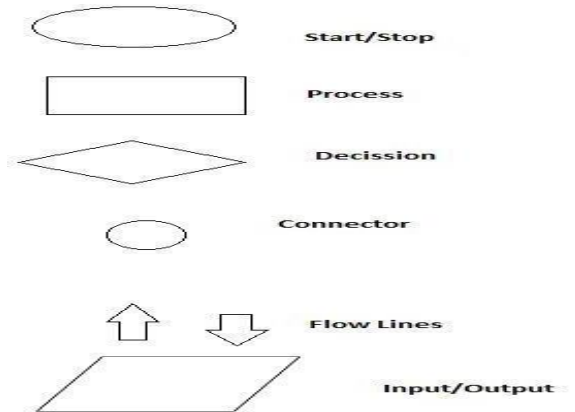
- Flowchart is an important aid in the development of an algorithm itself.
- Easier to understand than a program itself.
- Independent of any particular programming language.
- Proper documentation.
- Proper debugging.
- Easy and clear presentation.

Limitations of a Flowchart :

- Complex logic.
- Drawing is time consuming.
- Difficult to draw and remember.
- Technical detail.

Symbols : Symbols are divided in to the following twoparts.

- I. Auxiliary Symbols.
- II. Primary Symbols.



Prepare an algorithm and flow chart for swapping two numbers

To Swap two integer numbers:

Algorithm :

Using third variable

- Step 1 : Start
- Step 2 : Input num1 , num2
- Step 3 : Temp = num1
- Step 4 : num1 = num2
- Step 5 : num2 = Temp
- Step 6 : Output num1 , num2
- Step 7 : Stop



Flowcharts

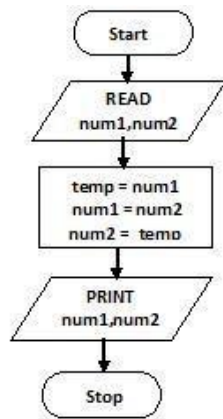


Fig a: With using third variable

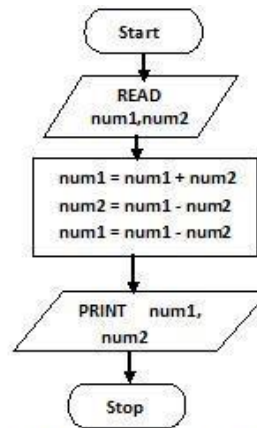


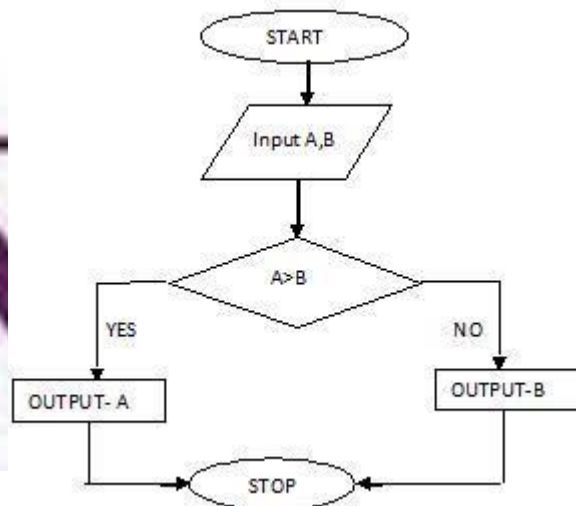
Fig b: Without using third variable

Develop an algorithm and flowchart to find the largest among two numbers.

Ans:Algorithm:

- Step 1: Start
- Step 2: Input A ,B
- Step 3: if $A > B$ then output A
else outputB
- Step 4: Stop

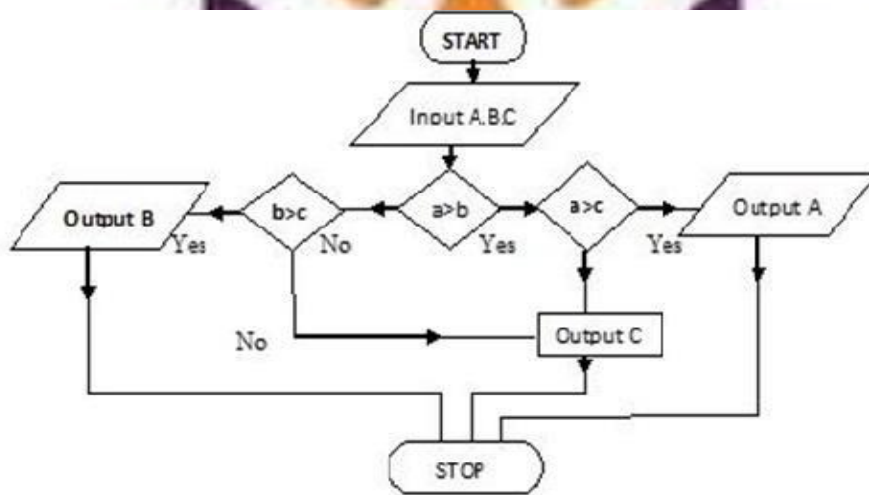
Flowchart



Develop an algorithm and flowchart to find the largest among three numbers. Algorithm:

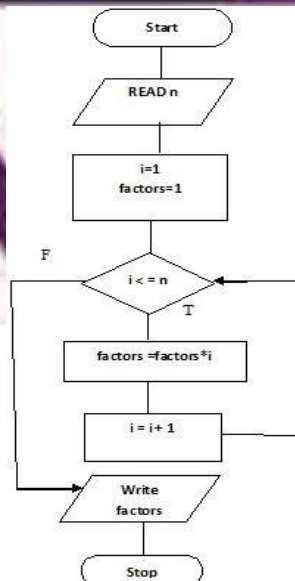
- Step 1 : Start
- Step 2 :Input A, B, C
- Step 3 :If $A > B$ goto step 4 ,otherwise goto step 5.
- Step 4 :if $A > C$ goto step 6,otherwise goto step 8
- Step 5 :if $B > C$ go to step 7,otherwise goto step 8
- Step 6 : print A is largest and goto step9
- Step 7 : print B is largest and goto step9
- Step 8 : print C is largest and goto step9
- Step 9 :Stop

Flowchart:



Simulate an algorithm and flowchart to find factorial of a number

Flowchart:



Algorithm:

Step 1:Start
Step 2:Input n
Step 3:Initialize counter variable, i , to 1 and factors = 1
Step 4:if $i \leq n$ go to step 5 otherwise goto step 7
Step 5:Calculate factors = factors * i
Step 6:Increment counter variable, i, and goto step 4
Step 7:output factors.
Step 8:stop

Pseudocode:

Pseudocode is an informal way of programming description that does not require any strict programming language syntax or underlying technology considerations. It is used for creating an outline or a rough draft of a program. Pseudocode summarizes a program's flow, but excludes underlying details. System designers write pseudocode to ensure that programmers understand a software project's requirements and align code accordingly.

Advantages of pseudocode –

- Pseudocode is understood by the programmers of all types.
- it enables the programmer to concentrate only on the algorithm part of the code development.
- It cannot be compiled into an executable program.

Example of Pseudocode:

Bigger of 2 numbers:

```
Read A,B.  
IF (A > B)  
    THEN Print A    "is bigger";  
    ELSE Print B    "is bigger";  
ENDIF;
```

Example Sum of numbers from 1 to 5:

```
Program : Print sum 1 to 5  
Sum=0;  
A=1;
```

```
While(A!=6)
Do sum=sum+A; Endwhile.
Print sum;
End
```

Creating compiling and executing a program:

Creating and running programs:

It is the job of programmer to write and test the program. The following are four steps for creating and running programs:

A. Writing and Editing the Program.

B. Compiling the Program.

C. Linking the Program with the required library modules.

D. Executing the Program.

A. Writing and Editing Program: The Software to write programs is known as text editor. A text editor helps us enter, change and store character data. Depending on the editor on our system, it could be used to write letters, create reports or write programs.

Example : word processor. The text editor could be generalized word processor, but every compiler comes with associated text editor.

Some of the features of editors are Search :

To locate and replace statements.

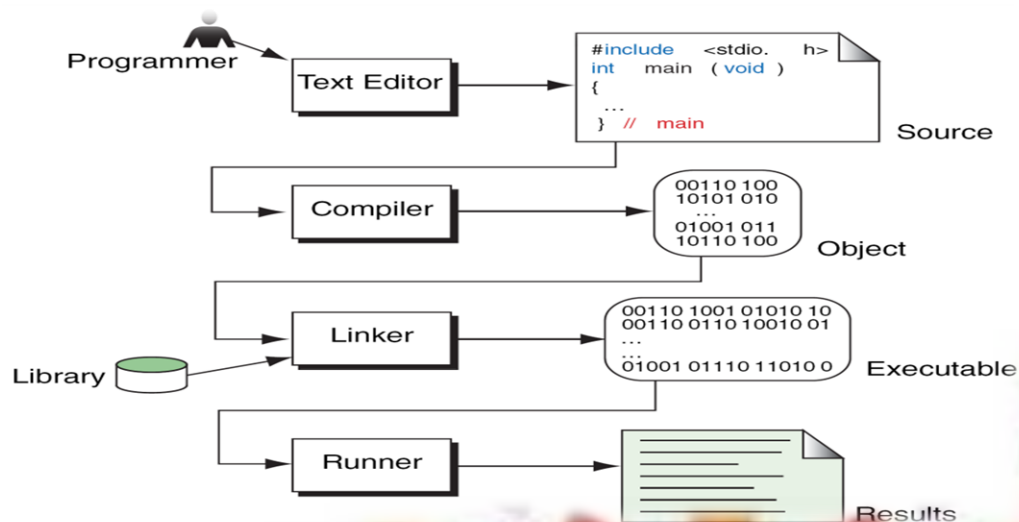
Copy , Paste : To copy and move statements.

Format : To set tabs to align text.

After the program is completed the program is saved in a file to disk. This file will be input to the compiler, it is known as source file. The following figure shows the various steps in building a C – program.

B. Compiling Programs: The code in a source file on the disk must be translated into machine language. This is the job of compiler which translates code in source file stored on disk into machine language. The C compiler is actually two separate programs: the preprocessor and the translator. The preprocessor reads the source code and prepares it for the compiler. It scans special instructions known as preprocessor commands. These commands tell the preprocessor to take for special code libraries, make substitutions in the code. The result of preprocessing is called translation unit. The translator reads the translation unit and writes resulting object module to a file that can be combined with other precompiled units to form the final program. An object module is the code in machine language. This module is not ready for execution because it does not have the required C and other functions included.

Fig: Steps followed to Build a C – program



C. Linking Programs: C programs are made up of many functions.

Example: `printf()` , `cos()`... etc Their codes exist elsewhere , they must be attached to our program. The linker assembles all these functions, ours and the system's, into a final executable program.

D. Executing Programs : Once our program has been linked, it is ready for execution. To execute a program, we use operating system command, such as `run` to load the program into main memory and execute it. Getting program into memory is the function of an Operating System program called loader. Loader locates the executable program and reads it into memory. In a typical program execution, the program reads data for processing, either from user or from file. After the program processes the data, it prepares output. Data output can be to user's monitor or to a file. When program has executed, Operating System removes the program from memory.

Syntax And Logical Errors In Compilation:

In [computer programming](#), a logic error is a [bug](#) in a program that causes it to operate incorrectly, but not to terminate abnormally (or [crash](#)). A logic error produces unintended or undesired output or other behavior, although it may not immediately be recognized as such.

Logic errors occur in both [compiled](#) and [interpreted](#) languages. Unlike a program with a [syntax error](#), a program with a logic error is a valid program in the language, though it does not behave as intended. Often the only clue to the existence of logic errors is the production of wrong solutions, though [static analysis](#) may

sometimes spot them

Object and Executable Codes:

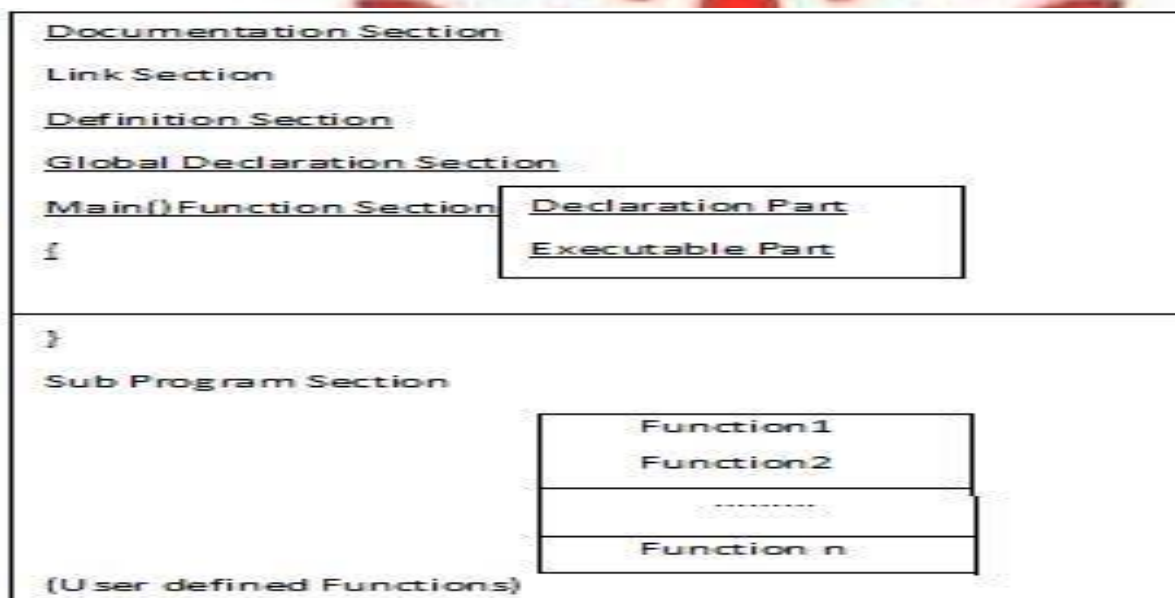
Object Code :

Object code is produced when an interpreter or a compiler translates source code into recognizable and executable machine code. Object code is a set of instruction codes that is understood by a computer at the lowest hardware level. Object code is usually produced by a compiler that reads some higher level computer language source instructions and translates them into equivalent machine language instructions.

Executable Code :

It is software in a form that can be run in the computer. It typically refers to machine language, which is the set of native instructions the computer carries out in hardware. Executable files in the DOS/Windows world use .exe and .com file extensions, while executable files in Unix and Mac do not require specific extensions. They are identified by their file structure.

Structure of C Program:



Documentation section :

This section consists of a set of comment lines giving the name of the program, and other details. which the programmer would like to user later.

Ex:- /*Addition of two numbers */

Link section: Link section provides instructions to the compiler to link functions from the system library.

Ex:- # include<stdio.h>

include<conio.h>

Definition section: Definition section defines all symbolic constants.

Ex:- # define A 10.

Global declaration section: Some of the variables that are used in more than one function throughout the program are called global variables and declared outside of all the functions. This section declares all the user-defined functions.

Main() function section:

Every C program must have one main () function section. This contains two parts.

i) **Declaration part:** This part declares all the variables used in the executable part.

Ex:- int a,b;

ii) **Executable part:** This part contains at least one statement. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. All the statements in the declaration and executable parts end with a semicolon (;).

Sub program section: This section contains all the user-defined functions, that are called in the main () function. User-defined functions are generally placed immediately after the main() function, although they may appear in any order.

Ex:

My first Program

This void means "main"
Returns no value.

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

main function

Library /Header file/ Prototype

```
void main(void)
```

```
{
```

This void means "main" passes no argument.

```
printf("How are you!");
```

Body of the main function.

```
}
```

VARIABLES

It is a data name that may be used to store a data value. It cannot be changed during the execution of a program. A variable may take different values at different times during execution. A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program.

Rules:

- Variable names may consist of letters, digits and under score(_) character.
- First char must be an alphabet or an _
- Length of the variable cannot exceed upto 8 characters, some C compilers can recognized upto 31 characters.
- White space is not allowed.
- Variables name should not be a keyword.
- Uppercase and lower case are significant.

Ex:- mark,sum1,tot_value,delhi (valid)

Prics\$, group one, char (invalid)

Declaration and initialization of variables with examples:

Declaration does two things:

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

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

The syntax for declaring a variable is as follows:

```
data-type v1,v2,.....,vn;
```

v1,v2,...,vn are the names of variables. Variables are separated by commas. A declaration statement must end with a semicolon. For example , valid declarations are:

```
int count;
```

```
int number, total;
```

```
double ratio;
```

The simplest declaration of a variable is shown in the following code fragment:

Example:

```
/*.....ProgramName.....*/  
  
int main()  
{  
//Declarations  
  
float x,y;  
  
int code;  
  
short int count;  
  
long int amount;  
  
double deviation;  
  
unsigned n;  
  
char c;  
  
/*.....Computation.....*/  
}  
  
/*.....Program ends.....*/
```



Initialization of variable :

Initialize a variable in c is to assign it a starting value. Without this we can't get whatever happened to memory at that moment.

C does not initialize variables automatically. So if you do not initialize them properly, you can get unexpected results. Fortunately, C makes it easy to initialize variables when you declare them.

For Example :

```
int x=45;  
  
int month_lengths[] = {23,34,43,56,32,12,24};  
  
struct role = { "Hamlet", 7, FALSE, "Prince of Denmark ", "Kenneth Branagh"};
```

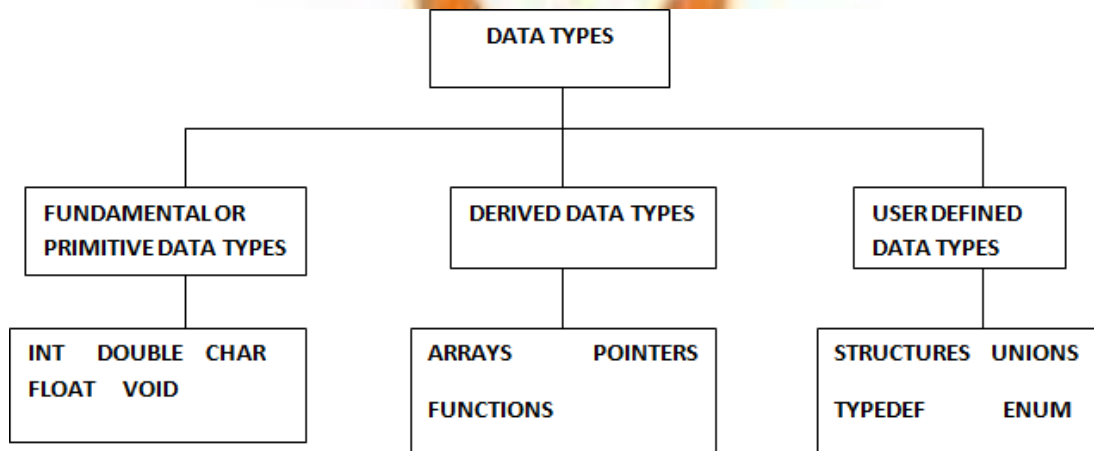
*Note : The initialization of variable is a good process in programming.

Data types :

A **data type** is a classification of the type of data that a variable can hold in computer programming. Data type is the type of the data that are going to access within the program. C supports different data types. Each data type may have pre-defined memory requirement and storage representation.

C supports 4 classes of data types.

1. Primary or (fundamental) data type(int, char, float,double)
2. User-defined data type(typedef,structures,unions)
3. Derived data type(arrays, pointers,)
4. Empty data type(void).



1. Primary or (fundamental) datatype

All C compilers support 4 fundamentals data types, they are

- i) Integer(int)
- ii) Character(char)
- iii) Floating(float)
- iv) Double – precision floatingpoint(double)

i)Integer types:

Integers are whole numbers with a range of values supported by a particular machine. Integers occupy one word of storage and since the word size of the machine vary. If we use 16 bit word length the size of an integer value is -32768 to +32767. In order to control over the range of numbers and storage space, C has 3 classes of integer storage, namely short, long, and unsigned.

DATA TYPE	RANGE	SIZE	FORMAT SPECIFIER

Int	-2^{15} to $2^{15} - 1$ -32768 to +32767	4 bytes (on 32 bit processors)	%d or %i
Signed short int or Short int	-128 to +127	4 bytes	%d or %i
Unsigned short int	0 to 255	4 bytes	%d or %i
Unsigned int	0 to 65,535	4 bytes	%u
Unsigned long int	0 to 4,294,967,295	4 bytes	%lu
Long int or signed long int	-2147483648 to +2147483647	4 bytes	%lu

Character type:-

Single character can be defined as a character (char) type data. Characters are usually stored in 8bits of internal storage. Two classes of char types are there.

signed char, unsigned char.

signed char(or) char 1 byte- -128 to +127%c

unsigned char 1 byte 0 to 255%c

ii) Floating point types:

Floating point (real) numbers are stored in 32 bits, with 6 digits of precision when accuracy provided by a float number is not sufficient.

float 4 bytes $3.4e-38$ to $3.4e+38$ %f

iv). Double precision type:

double data type number uses 64bits giving a precision of 14 digits. These are known as double precision numbers. Double type represents the same data type that float represents, but with a greater precision. To extend the precision further, we may use long double which uses 80bits.

Double 8 bytes $1.7e-308$ to $1.7e+308$ %lf

long double 10 bytes $3.4e-4932$ to $1.1e+4932$ %lf

2. User defined datatypes:

C—supports a feature known as `typedef` that allows users to define an identifier that would represent an existing type.

Syntax: `typedef data-type identifier;` where `data-type` indicates the existing datatype identifier and `identifier` indicates the new name that is given to the data type.

Ex:-`typedef int marks;`

`marks m1, m2, m3;`

`typedef` cannot create a new datatype, it can rename the existing datatype. The main advantage of `typedef` is that we can create meaningful datatype names for increasing the readability of the program. Another user-defined datatype is `enumerated datatype (enum)`

Syntax: `enum identifier {value1, value2, valuen};`

Where `identifier` is user-defined datatype which is used to declare variables that can have one of the values enclosed within the braces. `value1, value2, valuen` all these are known as enumeration constants.

Ex:-`enum identifier v1, v2, vn`

`v1=value3;`
`v2=value1;...`
`.....`

Ex:-`enum day {Monday, Tuesday, Sunday};`
`enum day week-f, week-end`

`Week-f = Monday`

`enum day {Monday...Sunday} week-f, week-end;`

CONSTANTS IN C

Constants is the most fundamental and essential part of the C programming language.

Constants in C are the fixed values that are used in a program, and its value remains the same during the entire execution of the program.

- Constants are also called literals.
- Constants can be any of the [data types](#).
- It is considered best practice to define constants using only *upper-case* names.

Types of C constants:

1. Integer constants

2. Real constants

3.Character constants

4. String constants

1. Integer constants: An integer constant refers to a sequence of digits. There are three types of integers, namely, decimal integer, octal integer and hexadecimal integer.

Examples of Integer Constants:

426 , +786 , -34 (decimal integers)

037, 0345, 0661 (octal integers)

0X2, 0X9F, 0X (hexadecimal integers)

2. Real constants: These quantities are represented by numbers containing fractional parts like 18.234. Such numbers are called real (or floating point) constants.

Examples of Real

Constants: +325.34426.0

-32.67 etc.

The exponential form of representation of real constants is usually used if the value of the constant is either too small or too large. In exponential form of representation the real constant is

represented in two parts. The first part present before 'e' is called Mantissa and the part following 'e' is called Exponent.

For ex. .000342 can be written in Exponential form as $3.42e-4$.

3. Single Character constants: Single character constant contains a single character enclosed within a pair of single quote marks.

For ex. 'A', '5', ' ', ' ' =

Note that the character constant '5' is not same as the number 5. The last constant is a blank space. Character constant has integer values known as ASCII values. For example, the statement



`printf("%d",a);` would print the number 97,the ASCII value of the letter a.

Similarly,the statement `printf("%c",97);` would output the letter `_a'`

Backslash Character Constants: C supports some special backslash character constants

that are used in output functions. Some of the back slash character constants are as follows:

CONSTANT	MEANING
----------	---------

<code>\0'</code>	NULL
<code>\t'</code>	Horizontal tab
<code>\b'</code>	Backspace
<code>\a'</code>	Audible bell
<code>\n'</code>	New line
<code>\v'</code>	Vertical tab
<code>\?'</code>	Question mark
<code>\''</code>	Single Quote
<code>\'''</code>	Double Quote
<code>\\'</code>	Backslash

These character combinations are called escape sequences.

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

Examples are: "HELLO!"

"1979"

"welcome"

".....!"

"5+3"

"A"



Rules for constructing integer constants:

- An integer constant must have at least one digit.
- It must not have a decimal point.
- It can be either positive or negative.
- The allowable range for constants is -32768 to 32767

In fact the range of integer constants depends upon compiler. For ex. 435

Rules for constructing real constants:

- A real constant must have at least one digit
- It must have a decimal point.
- It could be either positive or negative.
- Default sign is positive.

For ex.

+325.34

426.0

In exponential form of representation, the real constants is represented in two parts.

The part appearing before `_e'` is called mantissa where as the part following `_e'` is called exponent.

Range of real constants expressed in exponential form is $-3.4e38$ to $3.4e38$. Ex. $+3.2e-5$

Rules for constructing character constants:

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.
- The maximum length of character constant can be one character.



Ex `A`

Operators :

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 variables.

C operators can be classified into a number of categories, they are

- 1.ArithmeticOperators 2.RelationalOperators 3.LogicalOperators 4.Assignment Operators
- 5.Increment anddecrement operators 6.Conditionaloperators 7.Bitwise Operators
- 8.Specialoperators

1. Arithmetic Operators:C provides all the basic arithmetic operators. These can operate onany built –in data type allowed inC.

The arithmetic operators are listed as follows:

Operator	Meaning	Examples
+	Addition	c=a+5;a=b+c;h=9+6;
-	Subtraction	d=e-f;
*	Multiplication	k=3*g;
/	Division (quotient)	m=u/v;
%	Modulo division (remainder)	l=j%k

NOTE:

1. Integer division truncates any fractionalpart.
2. The modulodivision operation produces the remainder of an integerdivision.
3. Modulus operator is not valid for real and mixed modearithmetic.

Integer Arithmetic:

If both the operands in an arithmetic expression are integers then it is known as integer arithmetic and the result is an integer.

Real Arithmetic:

If both the operands in an arithmetic expression are real operands then it is known as real arithmetic and the result is real.

Mixed mode Arithmetic:

If the operands in an arithmetic expression are of different types then it is known as mixed mode arithmetic and the result is a bigger type.

2. Relational Operators:

Relational operators are used for comparing two quantities, and used for decision making. For example we may compare the age of two persons or the price of two items which can be done with the help of relational operators. An expression containing a relational operator is termed as a relational expression. The value of a relational expression is either one or zero. It is one if the specified relation is true and zero if the relation is false.
 Ex:- $13 < 34$ (true) $23 > 35$ (false)

C supports 6 relational operators

Operator	Meaning	Example	Result
<	is less than	$a < 6$ ($a=7$)	0
<=	is less than or equal to	$x <= y$ ($x=4, y=7$)	1
>	is greater than	$9 > 8$	1
>=	is greater than or equal to	$x + 10 >= y$ ($x=4, y=7$)	1
==	is equal to	$8 == 8$	1
!=	is not equal to	$6 != 5$	1

When arithmetic expressions are used on either side of a relational operator, the arithmetic expression will be evaluated first and then the results compared, that means arithmetic operators have a higher priority over relational operators.

3. Logical Operator:

C has 3 logical operators. The logical operators are used when we want to test more than one condition and make decisions. The operators are as follows:

Operator	Meaning	Example	Result
&&	Logical AND	$4 > 3 \ \&\& \ 5 != 6$	1
	Logical OR	$5 <= 7 \ \ 5 == 5$	1
!	Logical NOT	$!(5 == 4)$	1

The logical operators && and || are used when we test more than one condition and make decisions. Logical expressions combining two or more relational expressions, is also known as compound relational expression. The truth table for Logical AND and Logical OR is as below

OP1	OP2	OP1&&OP2	OP1 OP2
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1



Assignment operator:

These operators are used to assign the result of an expression to a variable. The usual assignment operator is “=”. In addition ,C has a set of shorthand assignment operators of the form:

$v \text{ op} = \text{exp};$

Where v is a variable, exp is an expression and op is a C binary arithmetic operator. The operator $\text{op} =$ is known as the shorthand assignment operator.

Operator	Meaning	Example
=	Assignment	$a=5; \quad x=y;r=s*t;$
+=	Sum equal to	$b+=5$ implies $b=b+5$
-=	Subtract and assign	$c-=55$ implies $c=c-55$
=	Multiply equal to	$b=5$ implies $b=b*5$
/=	Divide equal to	$a/=10$ implies $a=a/10$
%=	Remainder equal to	$d%=7$ implies $d=d\%7$

The assignment statement $v \text{ op} = \text{exp};$ is equivalent to $v = v \text{ op} (\text{exp});$

The use of short hand assignment operators has the following advantages:

- i. What appears on the left- hand side need not be repeated and therefore it becomes easier to write
- ii. The statement is more concise and easier to read
- iii. The statement is more efficient

3. Increment and Decrement operators:

$++$ and $--$ are increment and decrement operators in C. The operator $++$ adds 1 to the operand, while $--$ subtracts 1. Both are unary operators. They exist in postfix and prefix forms.

Operator	Form	Meaning	Examples
$++$	$a++$	Post increment	$i++$
	$++a$	Pre increment	$++\text{sum}$
$--$	$b--$	Post decrement	$j--$
	$--b$	Pre decrement	$--\text{count}$

`++m`; or `m++`; is equal to `m=m+1(m+=1;`)

`--m`; or `m-` - is equal to `m=m-1(m- = 1;`)

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

Consider the following example when `m=5`, then

`y=++m`; the value of `y=6` and `m =6`. Suppose if we write the above statement as `m=5`;

`y= m++`; the value of `y=5` and `m=6`.

A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand. Likewise for decrement operator.

4. Conditional operator:

A ternary operator pair “`? :`” is available in C to construct conditional expressions of the form

`exp1 ?exp2 : exp3` where `exp1`, `exp2` and `exp3` are expressions.

The operator?: works as follows: `exp1` is evaluated first. If it is non-zero (true), then the expression `exp 2` is evaluated and becomes the value of the expression. If `exp1` is false, `exp3` is evaluated and it becomes the value of the expression.

Ex:- `a=10; b=45;`

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

o/p:- Big will be assigned the value of b (45) since `a>b` is false.

5. Bitwise Operators:

C supports a special operator known as bitwise operators for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right to left.

Note: Bitwise operators may not be applied to float or double.

Operator	Meaning	Example
<code>&</code>	Bitwise AND	<code>X&y</code>
<code> </code>	Bitwise OR	<code>x y</code>
<code>^</code>	Bitwise exclusive OR	<code>X^y</code>
<code><<</code>	Shift left	<code>X<<y</code>
<code>>></code>	Shift right	<code>X>>y</code>
<code>~</code>	Bitwise ones complement	<code>~y</code>

6. Special operators:

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

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

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

This statement first assigns the value 10 to x, then assigns 5 to y and finally assigns 15. In for loops: for

```
(n=1, m=10; n<=m; n++, m++);
```

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 variable, a constant or a data type qualifier.

```
m = sizeof (sum);
```

```
n = sizeof (long int);
```

The sizeof operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program

Expressions:

An expression in C is some combination of constants, variables, operators and function calls.

Examples: C= a + b;

tan (angle)

- Most expressions have a value based on their contents.
- A statement in C is just an expression terminated with a semicolon. For

Example:

```
sum = x + y + z;
```

The rules given below are used to evaluate an expression,

1. If an expression has parenthesis, subexpression within the parenthesis is evaluated first and arithmetic expression without parenthesis is evaluated first.
2. The operators of high level precedence are evaluated first.
3. The operators at the same precedence are evaluated from left to right or right to left depending on the associativity of operators.

Expression evaluation:

Expressions are evaluated using an assignment statement of the form:

```
variable = expression;
```


variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted

Ex:- $x = a*b-c;$

$y =b/c*a;$

$z = a-b / c+d;$

Ex:- $x= a-b/3+c*2-1$ when $a=9$, $b=12$, and $c=3$ the expression becomes. $x = 9-$

$12/3 +3*2-1$

Step1: $x = 9-4+3*2-1$

Step2: $x = 9-4+6-1$

Step3: $x = 5+6-1$

Step4: $x = 11-1$

Step5: $x = 10$

Precedence and Associativity of operators:

Operator precedence:

Various relational operators have different priorities or precedence. If an arithmetic expression contains more operators then the execution will be performed according to their properties. The precedence is set for different operators in C.

Type of operator	Operators	Associativity
Unary operators	+, -, !, ++, --, type, sizeof	Right to left
Arithmetic operators	*, /, %, +, -	Left to right
Bit – manipulation operators	<<, >>	Left to right
Relational operators	>, <, >=, <=, ==, !=	Left to right
Logical operators	&&,	Left to right
Conditional operators	?:	Left to right
Assignment operators	=, +=, -=, *=, /=, %=	Right to left

Note:

1. Precedence rules decide the order in which different operators are applied
2. Associativity rule decides the order in which multiple occurrences of the same level operators are applied.

Hierarchy of operators in C :

The higher the position of an operator is, higher is its priority. When an expression contains two operators of equal priority the tie between them is settled using the associativity of the operators.

Associativity can be of two types—Left to Right or Right to Left

Left to Right means, as you go from left to right in an expression which operator among the two is found, execute it first. Right to left means, as you go from right to left in an expression which operator among the two is found, execute it first.

Consider expression $a=3/2*5$ Here there is a tie between operators of same priority, that is between / and *. This tie is settled using the associativity of / and *. But both enjoy Left to Right associativity, which means as you go from left to right / is found so execute it first.

The table clearly shows the associativity and precedence of operators in c.

RANK	OPERATORS	MEANING	ASSOCIATIVITY
1	++	POSTFIX INCREMENT	LEFT TO RIGHT
	--	POSTFIX DECREMENT	
	()	FUNCTION CALL	
	[]	ARRAY SUBSCRIPTING	
	.	STRUCTURE AND UNION MEMBER ACCESS	
	□	STRUCTURE AND UNION MEMBER ACCESS THROUGH POINTER	
2	++	PREFIX INCREMENT	RIGHT TO LEFT
	--	PREFIX DECREMENT	
	+	UNARY PLUS	
	-	UNARY MINUS	
	!	LOGICAL NOT	
	~	BITWISE NOT	
	(TYPE)	TYPE CAST	
	*	INDIRECTION	
	&	ADDRESS OF	
sizeof()	SIZE OF		
3	*	MULTIPLICATION	LEFT TO RIGHT
	/	DIVISION	
	%	MODULO DIVISION	
4	+	ADDITION	LEFT TO RIGHT
	-	SUBTRACTION	
5	<<	BITWISE SHIFT LEFT	LEFT TO RIGHT
	>>	BITWISE SHIFT RIGHT	
6	<	RELATIONAL OPERATORS LESSER	LEFT TO RIGHT
	<=	LESSER THAN OR EQUAL TO	
	>	GREATER THAN	
	>=	GREATER THAN OR EQUAL TO	
7	==	IS EQUAL TO	LEFT TO

	!=	IS NOT EQUAL TO	RIGHT
8	&	BITWISE AND	LEFT TO RIGHT
9	^	BITWISE XOR	LEFT TO RIGHT
10		BITWISE OR	LEFT TO RIGHT
11	&&	LOGICAL AND	LEFT TO RIGHT
12		LOGICAL OR	LEFT TO RIGHT
13	?:	TERANARY	LEFT TO RIGHT
14	=	ASSIGNMENT OPERATOR SIMPLE	RIGHT TO LEFT
	+=	ASSIGNMENT BY SUM	
	-=	ASSIGNMENT BY DIFFERENCE	
	*=	ASSIGNMENT BY PRODUCT	
	/=	ASSIGNMENT BY QUOTIENT	
	%=	ASSIGNMENT BY REMAINDER	
	<<=	ASSIGNMENT BY BITWISE LEFT SHIFT	
	>>=	ASSIGNMENT BY BITWISE RIGHT SHIFT	
	&=	ASSIGNMENT BY BITWISE AND	
	^=	ASSIGNMENT BY BITWISE XOR	
=	ASSIGNMENT BY BITWISE OR		
15	,	COMMA	LEFT TO RIGHT

For Example:

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

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

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

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

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

$i = 2 + 8 - 2 + 0$

$i = 10 - 2 + 0$

$i = 8 + 0$

$i = 8$



The logo for NIRCM (National Institute of Research in Computer Graphics and Multimedia) features a stylized tree with a purple trunk and branches, and a red and white circular emblem in the center of the branches. Below the tree, the letters 'NIRCM' are written in a bold, purple, sans-serif font.

Conditional Branching and Loops:

Writing and Evaluation of conditional statements (examples).

In C, when a logical operation is being evaluated, if the result is known before all sub expressions have been evaluated, then the evaluation stops, or short circuits. The two situations where this can occur is when the first expression of a logical AND operation is FALSE (zero) or the first expression of a logical OR operation is TRUE (non-zero). In both of these cases, the result is already known. For AND, if either of the two expressions is FALSE, the result will be FALSE. For OR, if either of the two expressions is TRUE, then the result will be TRUE.

Example :

If we have two expressions being tested in a logical AND operation:
`expr1 && expr2`

The expressions are evaluated from left to right. If `expr1` is 0 (FALSE), then `expr2` would not be evaluated at all since the overall result is already known to be false.

Truth table for AND (&&)

FALSE=0

TRUE=1

expr1	expr2	Result
0	X (0)	0
0	X (1)	0
1	0	0
1	1	1

`Expr2` is not evaluated in the first two cases since its value is not relevant to the result.

Examples:

```
1.if (5 || ++x)
   {
   printf("%d\n",x);
   }
```

```
2.int    x    =0;
   if(5||2&&++x)
   printf("%d", x);
```

3.(a >= 0) && (b < 10)

4.(a > b) || (b++ / 3)

5. (13 * a) * (b/13 -1)

6 if((a || b || c || d || e || f || g || h || i || j || k)==1)

7.while((x && y)==1)

8. if ((x < 10) && (y > 3))

Conditional Branching:

Conditional statements help you to make decision based on certain conditions. These conditions are specified by a set of conditional statements having *Boolean expressions* which are evaluated to a boolean value true or false.

When we need to execute a block of statements only when a given condition is true then we use variants of if statements.

There are 4 if statements available in C:

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

1. **Simple if statement:** Simple if statement is used to make a decision based on the available choice. It has the following form:

Syntax: if (condition)

```
{
  stmt block;
}
```

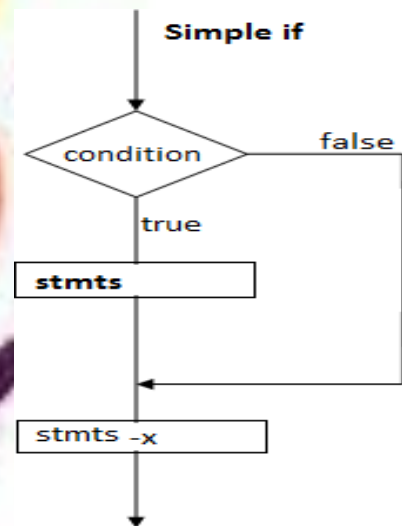
stmt-x;

In this syntax,

- if is the keyword. *<condition>* is a relational expression or logical expression or any expression that returns either true or false. It is important to note that the condition should be enclosed within parentheses '(' and ')'
- The *stmt block* can be a simple statement or a compound statement or a null statement.
- *stmt-x* is any valid C statement.

The flow of control using simple if statement is determined as follows:

Whenever simple if statement is encountered, first the condition is tested. It returns either true or false. If the condition is false, the control transfers directly to stmt-x without considering the stmt block. If the condition is true, the control enters into the stmt block. Once, the end of stmt block is reached, the control transfers to stmt-x.



Example Program for if statement:

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

```

void main()
{
int age;
printf("enter age\n");
scanf("%d",&age);
if(age>=55)
printf("person is retired\n");
}

```

Output: enter age 57

person is retired

2. if—else statement:

if...else statement is used to make a decision based on two choices. It has the following form:

Whenever if...else statement is encountered, first the condition is tested. It returns either true or false. If the condition is true, the control enters into the true stmt block. Once, the end of true stmt block is reached, the control transfers to stmt-x without considering else-body.

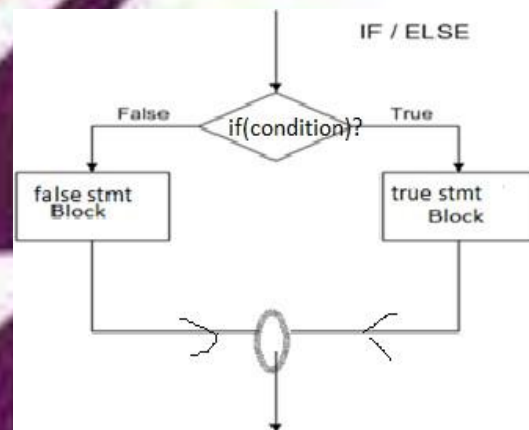
If the condition is false, the control enters into the false stmt block by skipping true stmt block. Once, the end of false stmt block is reached, the control transfers to stmt-x.

Syntax:

```

if(condition)
{
truestmt block;
}
else
{
falsestmt block;
}
stmt-x;

```



In this syntax,

- if and else are the keywords.
- *<condition>* is a relational expression or logical expression or any expression that returns either true or false. It is important to note that the condition should be enclosed within parentheses (and).
- The *truestmt block* and *falsestmt block* are simple statements or compound statements or null statements.
- *stmt-x* is any valid C statement.

The flow of control using if...else statement is determined as follows:

Program for if else statement:

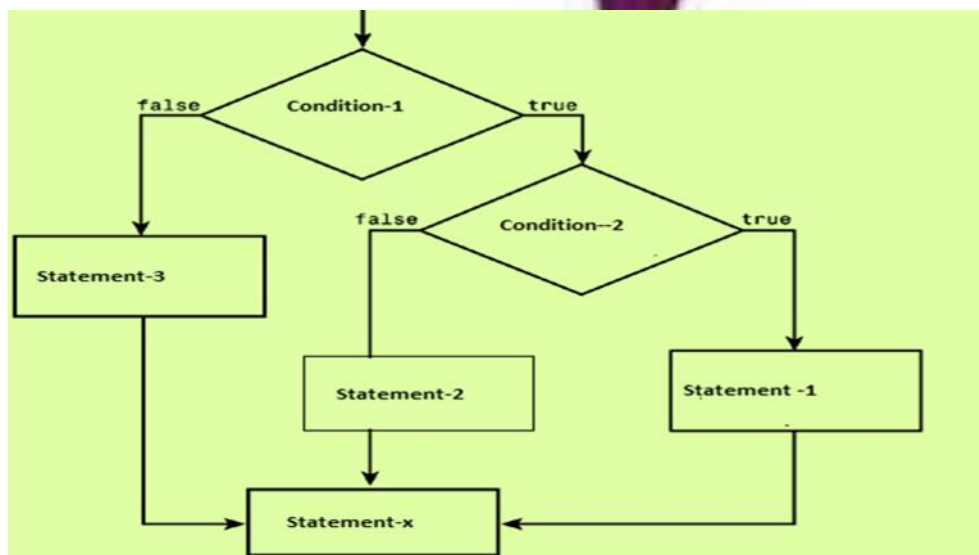
```
#include<stdio.h>
void main()
{
int age;
printf("enter age\n");
scanf("%d",&age);
if(age>=55)
printf("person is retired\n");
else
printf("person is not retired\n");
}
```

Output: enter age 47
person is not retired

Nested if (else) Statements:

- The statement executed as a result of an if statement or else clause could be another if statement
- These are called *nested if statements*
- An *else* clause is matched to the last unmatched if (no matter what the indentation implies)
- Braces can be used to specify the if statement to which an *else* clause belongs

Flow chart:



Example program:

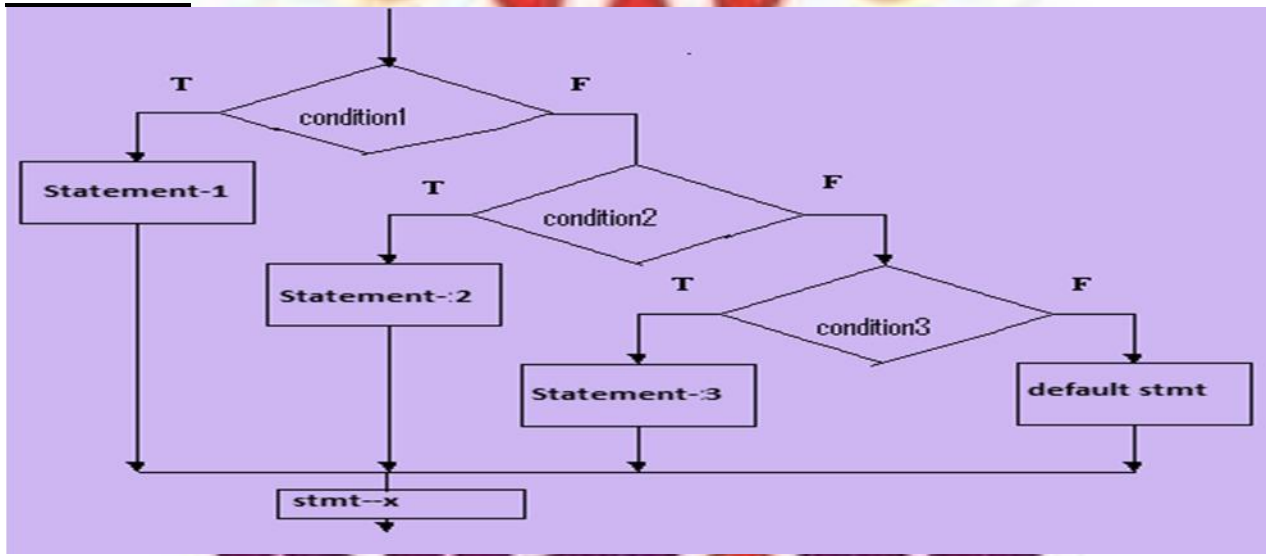
```
void main ( )
int a, b;
```

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

Else-if ladder:

- Else if ladder is one of the conditional control-flow statements.
- It is used to make a decision among multiple choices.

Flow chart:



Example Program:

```
void main()
{
    int m1,m2,m3,avg,tot;
    printf("enter three subject marks");
    scanf("%d%d%d", &m1,&m2,&m3);
    tot=m1+m2+m3;
```



```

avg=tot/3;
if(avg>=75)  {
printf("distinction");  }
else if(avg>=60 && avg<75) {
printf("first class");  }
else if(avg>=50 && avg<60) {
printf("second class");  }
else if (avg<50)  {
printf("fail");
}
}

```

Switch Statement:

switch statement is one of decision-making control-flow statements. Just like else if ladder, it is also used to make a decision among multiple choices. switch statement has the following form:

Syntax:

```

switch(<exp>)
{
case<exp-val-1>: statements block-1;
break;
case<exp-val-2>: statements block-2;
break;
case<exp-val-3>: statements block-3;
break;
case<exp-val-N>: statements block-N;
break;
default: default statements block;
}
Next-statement;

```

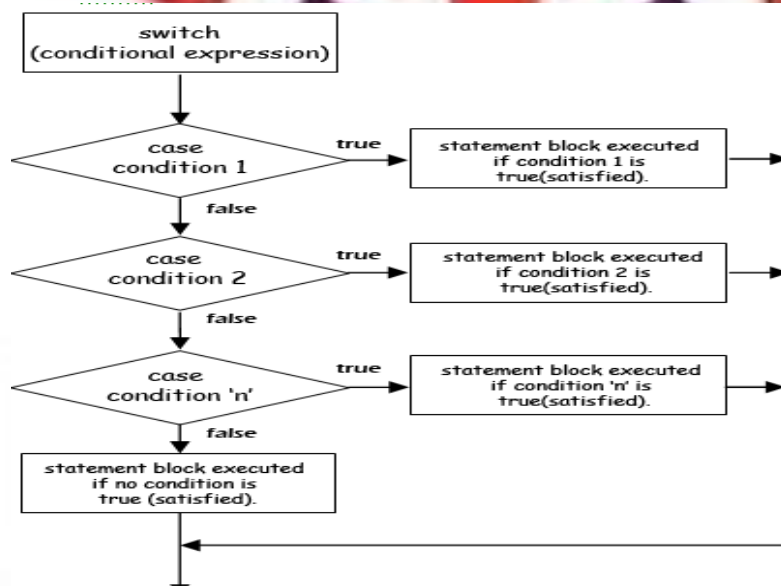
In this syntax,

- switch, case, default and break are keywords.
- *<exp>* is any expression that should give an integer value or character value. In other words, it should never return any floating-point value. It should always be enclosed with in parentheses (and). It should also



be placed after the keyword switch.

- $\langle exp-val-1 \rangle$, $\langle exp-val-2 \rangle$, $\langle exp-val-3 \rangle$ $\langle exp-val-N \rangle$ should always be integer constants or character constants or constant expressions. In other words, variables can never be used as $\langle exp-val \rangle$. There should be a space between the keyword case and $\langle exp-val \rangle$. The keyword case along with its $\langle exp-val \rangle$ is called as a case label. $\langle exp-val \rangle$ should always be unique; no duplications are allowed.
- *statements block-1*, *statements block-2*, *statements block-3*... *statements block-N* and *default statements block* are simple statements, compound statements or null statements. It is important to note that the statements blocks along with their own case labels should be separated with a colon (:)
- The break statement at the end of each statements block is an optional one. It is recommended that break statement always be placed at the end of each statements block. With its absence, all the statements blocks below the matched case label along with statements block of matched case get executed. Usually, the result is unwanted.
- The statement block and break statement can be enclosed within a pair of curly braces { and }.
- The default along with its statements block is an optional one. The break statement can be placed at the end of default statements block. The default statements block can be placed at anywhere in the switch statement. If they are placed at any other place other than at end, it is compulsory to include a break statement at the end of default statements block.
- *Next-statement* is a valid C statement.



Whenever, switch statement is encountered, first the value of $\langle exp \rangle$ gets matched with case values. If suitable match is found, the statements block related to that matched case gets executed. The break statement at the end transfers the control to the *Next-statement*.

If suitable match is not found, the default statements block gets executed and then the control gets transferred to *Next-statement*.

Example Program:

```
#include<stdio.h> void main()
{   inta,b,c,ch;
printf("\nEnter two numbers :");

scanf("%d%d",&a,&b); printf("\nEnter the choice:"); scanf("%d",&ch);
switch(ch)
{
case 1: c=a+b;
break; case 2: c=a-b;
break; case 3: c=a*b;
break; case 4: c=a/b;
break;
default: printf(" enter a valid choice \n");
}
printf("\nThe result is: %d",c);
}
```

Output: enter the choice 1

Enter two numbers 4 2

6

Use of switch statement:

- We can use switch statements alternative for an if...else ladder.
- The switch statement is often faster than nested if...else Ladder.
- Switch statement syntax is well structured and easy to understand.

Iteration and Loops: while, do-while, for loops.

It is the process where a set of instructions or statements is executed repeatedly for a specified number of time or until a condition is satisfied. These statements also alter the control flow of the program. It can also be classified as **control statements** in C Programming Language.

Iteration statements are most commonly known as **loops**. There are

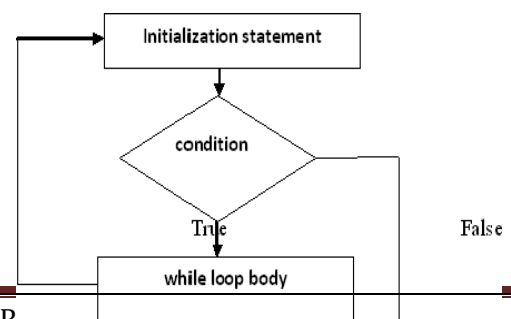
three types of looping statements:

1. **while Loop**
2. **do while loop**
3. **forloop**

while Loop:

The simplest of all the looping structures in c is the while statement. A WHILE loop has one control expression, and executes as long as that expression is true.

The basic format of the while statement is:



Syntax:

```
while(condition)
{
statements;
}
```

Flow chart for while loop

The while is an entry-controlled loop statement. The condition is evaluated and if the condition is true then the statements will be executed. After execution of the statements the condition will be evaluated and if it is true the statements will be executed once again. This process is repeated until the condition becomes false and the control is transferred out of the loop. On exit the program continues with the statement immediately after the body of the loop.

Program to print n natural numbers using using while

```
#include<stdio.h>
void main()
{
inti,n;
printf("enter the range\n");
scanf("%d",&n);
i=1;
while(i<=n)
{
printf("%d ",i);
i=i+1;
}
}
```

Output: enter the range 10 1 2 3 4 5 6

7 8 9 10

Do-while Loop:

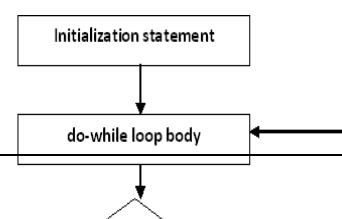
It is one of the looping control statements. It is also called as *exit-controlled looping control statement*. i.e., it tests the condition after executing the do-while loop body.

The main difference between “while” and “do-while” is that in “do-while” statement, the loop body gets executed at least once, though the condition returns the value false for the first time, which is not possible with while statement. In “while” statement, the control enters into the loop body only when the condition returns true.

Syntax:

```
Initialization statement;
```

Flow chart for do while




```
do
{
statement(s);
} while(<condition>);
next statement;
```

In this syntax:

while and **do** are the keywords. <condition> is a relational expression or a compound relational expression or any expression that returns either true or false. initialization statement, statement(s) and next_statement are valid 'c' statements.

The statements within the curly braces are called as do-while loop body. The updating statements should be included within the do-while loop body. There should be a semi-colon (;) at the end of while(<condition>).

Whenever "do-while" statement is encountered, the initialization statement gets executed first. After then, the control enters into do-while loop body and all the statements in that body will be executed. When the end of the body is reached, the condition is tested again with the updated loop counter value.

If the condition returns the value false, the control transfers to next statement without executing do-while loop body. Hence, it states that, the do-while loop body gets executed for the first time, though the condition returns the value false.

Program to print n natural numbers using using do while

```
#include<stdio.h>
voidmain()
{
inti,n;
printf("enter the range\n");
scanf("%d",&n);
i=1;
do
printf("%d ",i);
i=i+1;
}
while(i<=n);
}
```

Output: enter the range 8

1 2 3 4 5 6 7 8

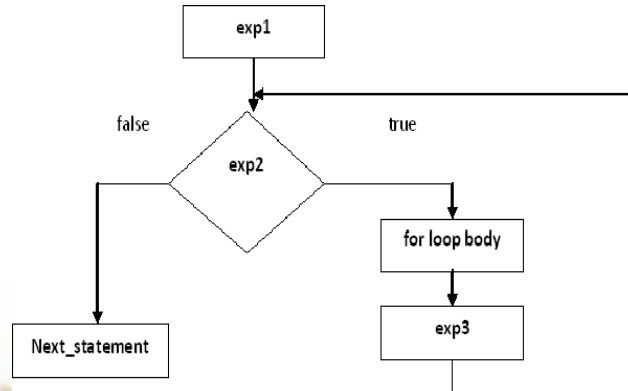
3. for loop:

It is one of the looping control statements. It is also called as *entry-controlled looping control statement*. i.e., it tests the condition before entering into the loop body. The syntax for "for" statement

is as follows:

Syntax:

```
for(exp1;exp2;exp3)
{
for-body;
}
next_statement;
```



In this syntax,

for is a keyword. **exp1 is the initialization statement.** If there is more than one statement, then the statements must be separated with commas. **exp2 is the condition.** It is a relational expression or a compound relational expression or any expression that returns either true or false. **The exp3 is the updating statement.** If there is more than one statement then, they must be separated with commas. exp1, exp2 and exp3 should be separated with two semi-colons. exp1, exp2, exp3, for-body and next_statement are valid 'c' statements. for-body is a simple statement or compound statement or a null statement.

Whenever "for" statement is encountered, first exp1 gets executed. After then, exp2 is tested. If exp2 is true then the body of the loop will be executed otherwise loop will be terminated.

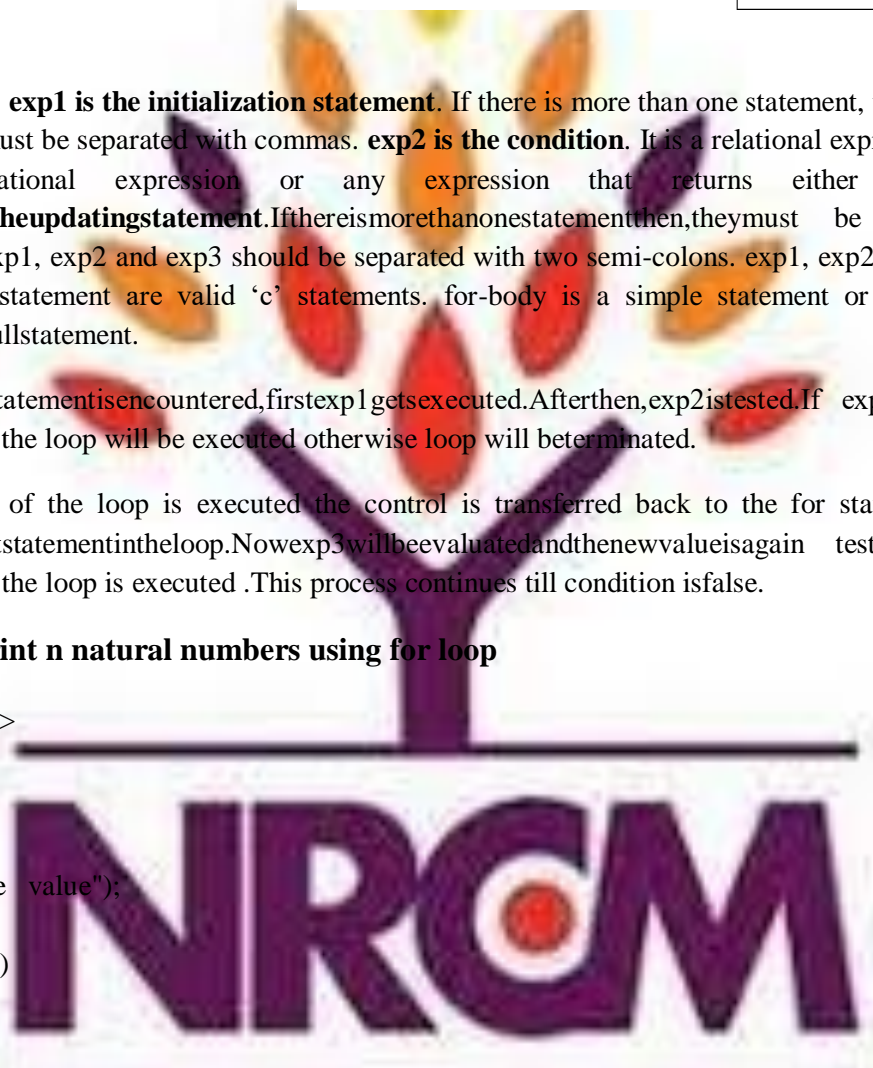
When the body of the loop is executed the control is transferred back to the for statement after evaluating the last statement in the loop. Now exp3 will be evaluated and the new value is again tested. If it satisfies body of the loop is executed. This process continues till condition is false.

Program to print n natural numbers using for loop

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

inti,n;
printf("enter the value");
scanf("%d",&n);
for(i=1;i<=n;i++)
printf("%d\n",i);
}
```

Output: enter the value 5 12345



Nested for Loop:

Nested for loop refers to the process of having one loop inside another loop. We can have multiple loops inside one another. The body of one 'for' loop contains the other and so on. The syntax of a **nested for loop** is as follows (using two for loops):

Syntax:

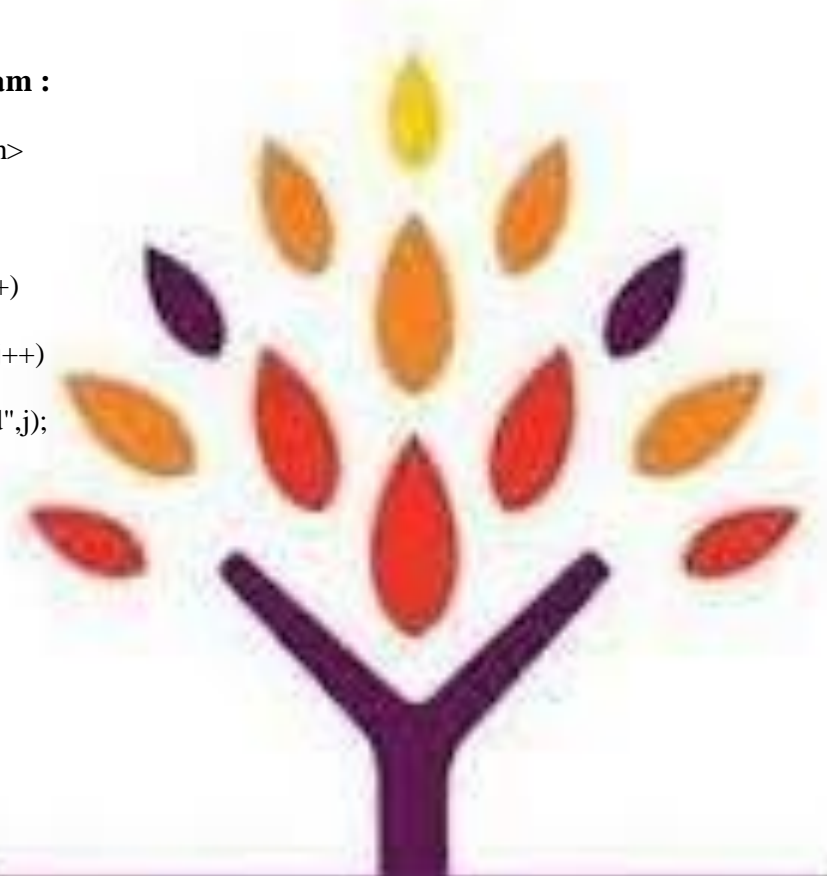
```
for(initialization ; condition; update)
{
    for(initialization ;condition;update)    //using another variable
    {
        body of the inner loop
    }
    body of outer loop    //(might or might not be present)
}
```

Example Program :

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

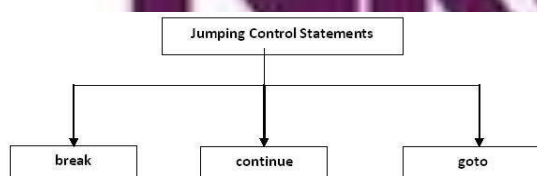
Output:

```
1
12
123
1234
12345
```



Jumping control-flow statements:

Jumping control-flow statements are the control-flow statements that transfer the control to the specified location or out of the loop or to the beginning of the loop. There are 3 jumping control statements:



break statement:

The `break` statement is used within the looping control statements, `switch` statement and nested loops. When it is used with the `for`, `while` or `do-while` statements, the control comes out of the corresponding loop and continues with the next statement.

When it is used in the nested loops or switch statement, the control comes out of that loop/switch statement within which it is used. But, it does not come out of the complete nesting.

Syntax for the —break statement is: **break;**

```
Any loop
{
statement_1;
statement_2;
break;
:
}
next_statement;
```

In this syntax, break is the keyword.

The following representation shows the transfer of control when break statement is used:

Program for break statement:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int i;
for(i=1; i<=10; i++)
{
if(i==6)
break;
printf("%d",i);
}
}
```

Output: 1 2 3 4 5

continue statement:

A continue statement is used within loops to end the execution of the current iteration and proceed to the next iteration. It provides a way of skipping the remaining statements in that iteration after the continue statement. It is important to note that a continue statement should be used only in loop constructs and not in selective control statements.

Syntax for continue statement is:

```
Any loop
{
statement_1;
statement_2;
.....
continue;
```

where continue is the keyword. The following representation shows the transfer of control when continue statement is used:



The image shows the NRCM logo, which consists of a stylized tree with a purple trunk and branches, and colorful leaves (red, orange, yellow, purple). Below the tree, the letters 'NRCM' are written in a large, bold, purple font. A small red circle is positioned between the 'R' and 'C'. A black rectangular box is drawn around the word 'continue;' in the text below the logo.


```
.....
}
next_statement;
```

Program for continue statement:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int i, sum=0;
for(i=1; i<=n; i++)
{
if(n==6)
continue;
printf("%d\n",i);
}
}
```

Output: 1 2 3 4 5 7 8 9 10

Exit :

The C library function **void exit(int status)** terminates the calling process immediately. This function does not return any value.

`void exit(int status)`

```
//Example
#include <stdio.h>
#include <stdlib.h>
int main()
{
printf("Start of the program. \n");
printf("Exiting the program. \n");
exit(0);
printf("End of the program. \n");
return(0);

}
```

Output:
Start of the program....
Exiting the program....



goto statement

The goto statement transfers the control to the specified location unconditionally. There are certain situations where goto statement makes the program simpler. For example, if a deeply nested loop is to be exited earlier, goto may be used for breaking more than one loop at a time. In this case, a break statement will not serve the purpose because it only exits a single loop.

Syntax for goto statement is:

label:

```
{  
    statement_1;  
    statement_2;  
    :  
}  
:
```

goto label;

In this syntax, goto is the keyword and label is any valid identifier and should be ended with a colon (:).

The identifier following goto is a statement label and need not be declared. The name of the statement or label can also be used as a variable name in the same program if it is declared appropriately. The compiler identifies the name as a label if it appears in a goto statement and as a variable if it appears in an expression.

If the block of statements that has label appears before the goto statement, then the control has to move to backward and that goto is called as backward goto. If the block of statements that has label appears after the goto statement, then the control has to move to forward and that goto is called as forward goto.

Program for goto statement:

```
#include<stdio.h>  
void main()  
{  
    printf("Hello");  
    goto x;  
    y:  
    printf("C Language");  
    goto z;  
  
    x:  
    printf("Let us learn");  
    goto y;  
    z:  
    printf("together.");  
}
```

Output: Hello Let us learn C Language together.

Type conversion:

It is a process of converting a variable value or a constant value temporarily from one data type to other data type for the purpose of calculation is known as type conversion.

There are two types of conversions

- i.. Automatic type conversion (or) Implicit conversion.
- ii . Type Casting (or) explicit conversion (or) manual conversion.

Implicit: In this lower data type can be converted into higher data type automatically. The figure below shows the C conversion hierarchy for implicit –type conversion in an expression:

The sequence of rules that are applied while implicit type conversion is as follows: All short and char are automatically converted into int then

1. if one of the operands is long double the other will be converted to long double and the result will be longdouble.
2. else, if one of the operand is double, the other will be converted to double and the result will bedouble.
3. else, if one of the operand is float ,the other will be converted to float and the result will be float.
4. else, if one of the operand is unsigned long int, the other will be converted to unsigned long int and the result will be unsigned longint.
5. else, if one of the operand is long int, the other is unsigned intthen
 - a)if unsigned int can be converted into long int, the unsigned int operand will be converted as such and the result will be longint.
 - b)else both operands will be converted to unsigned long int and the result will be unsigned longint.
6. else if one of the operand is long int ,the other will be converted into long int and the result will be longint.
7. else if one of the operand unsigned int ,the other will be converted into unsigned int and the result will be unsignedint.

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

- i. float to int causes truncation of the fractionalpart
- ii. double to float causes rounding ofdigits
- iii. long int to int causes dropping of the excess higher orderbits

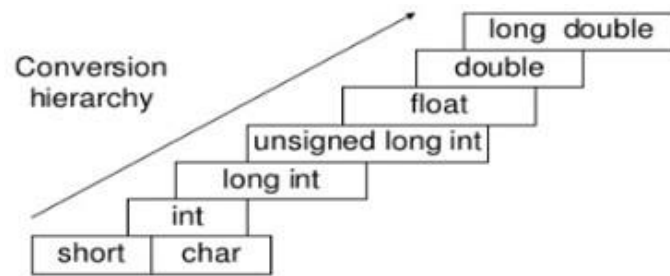


Fig: Conversion hierarchy

Example : Consider the following variables along with their datatypes: int i,x; float f; double d; longint l;

Explicit:

In this type of conversion, the programmer can convert one data type to other datatype explicitly .Such conversions are also known as forced conversions or manual conversions or type casting.

Syntax: (datatype) (expression) Expression can be a constant or a variable

Ex:y = (int) (a+b) y= cos(double(x)) double a = 6.5; double b = 6.5;

int result = (int) (a) + (int) (b); result = 12 instead of 13.

int a=10;

float(a)> 10.00000

Example:

```
//converting fahrenheit to celsius by type casting #include<stdio.h>
```

```
#include<conio.h> void main()
```

```
{
```

```
float c,f;
```

```
clrscr();
```

```
printf("\n enter fahrenheit:");
```

```
scanf("%f",&f);
```

```
c=(float)5/9*(f-32);
```

```
printf("\n c=%f",c);
```

```
getch();
```



}

Managing Input and Output:

Managing input and output operations:

Reading, processing and writing of data are the three essential functions of a computer program. Most program takes some data as input and displays the processed data. We have two methods of providing data to the program variables. One method is to assign values to variables through the assignment statements like $x=5, a=0$ and so on. Another method is to use the input function `scanf`, which can read data from a keyboard. We have used both the methods in programs. For outputting results, we have used extensively the function `printf`, which sends results out to a terminal.

Input – Output functions:

The program takes some I/P- data, process it and gives the O/P. We have two methods for providing data to the program

- (i) Assigning the data to the variables in a program.
- (ii) By using I/P-O/P statements.

C language has 2 types of I/O statements. All these operations are carried out through function calls.

1. Unformatted I/O statements

2. Formatted I/O statements

Unformatted I/O statements:

i/p function	o/p function
1) <code>getchar()</code>	<code>putchar()</code>
2) <code>gets()</code>	<code>puts()</code>
3) <code>getc()</code>	<code>putc()</code>
4) <code>getw()</code>	<code>putw()</code>
5) <code>getch()</code>	<code>putch()</code>
6) <code>getche()</code>	

getchar ():- It reads single character from standard input device. This function don't require any arguments.

Syntax : - `char variable _name = getchar();`

Ex:- `char x;`

`x = getchar();`

putchar ():- This function is used to display one character at a time on the standard output device.

Syntax:- `putchar(variable_name);`

Ex:- `char x;`

`putchar(x);`

program:

```
void main( )
{
    char ch;
    printf("enter a character:");
    ch=getchar( );
    printf("\n Character is:");
    putchar(ch);
}
```

Output:

```
enter a character:a

character isa
```

gets() :- This function is used to read group of characters(string) from the standard I/P device.

Syntax:-gets(string name);

Ex:- gets(s);

puts() :- This function is used to display string to the standard O/P device.

Syntax:-puts(string name);

Ex:- puts(s);

program:

```
void main()
{
    char    s[10];
    puts("enter name");
    gets(s);
    puts("print name:");
    puts(s);
}
```

Output:

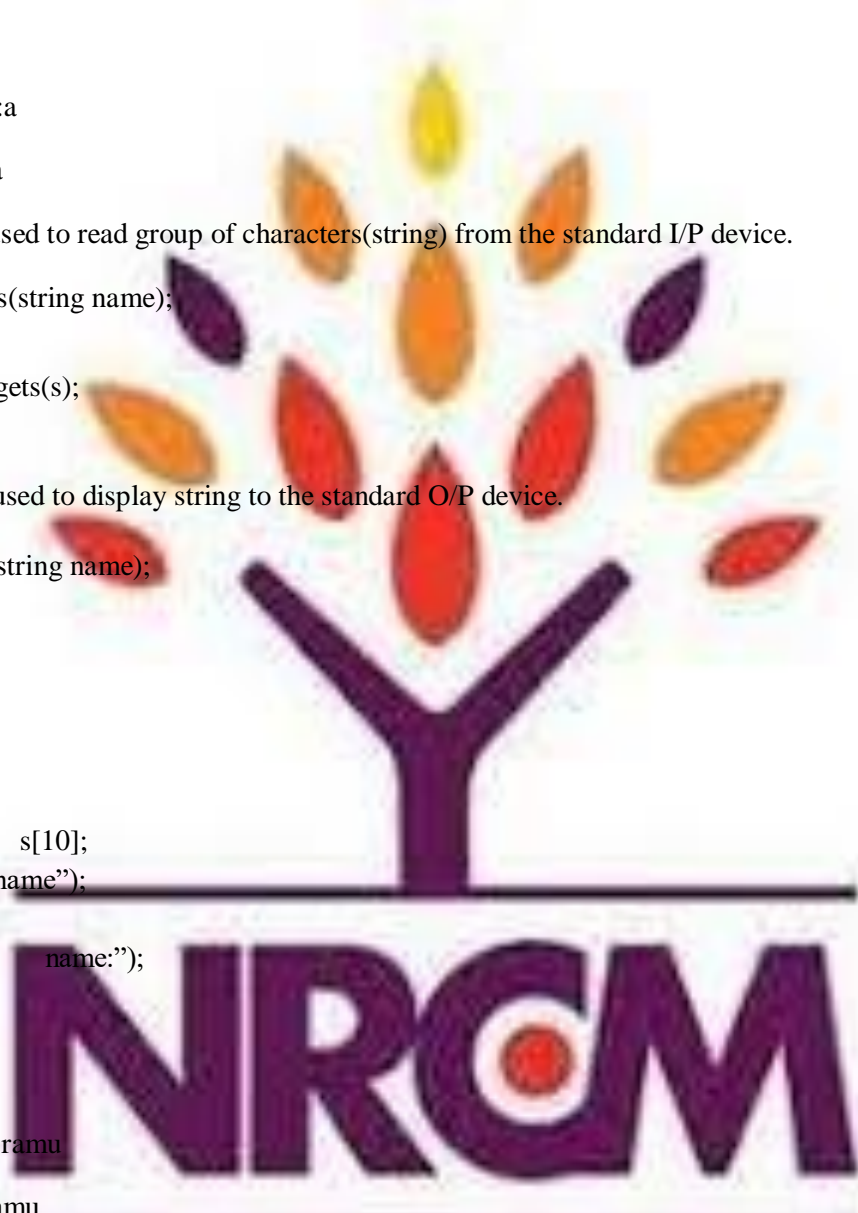
```
enter name ramu

print name : ramu
```

getch() :- This function reads a single character directly from the keyboard without displaying on the screen. This function is used at the end of the program for displaying the output (without pressing (Alt-F5)).

Syntax: char variable_name = getch();

Ex:- char c;



```
c = getch();
```

getche():- This function reads a single character from the keyboard and echoes(displays) it to the current text window.

Syntax:-char variable_name = getche();

Ex:- char c;

```
c = getche();
```

Program:

```
void main()
{
    char ch, c;
    printf("enter char");
    ch = getch();
    printf("%c", ch);
    printf("enter char");
    c = getche();
    printf("%c",c);
}
```

Output :-

```
enter character a
enter character b
b
```



Character test functions:

ctype.h

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

isupper(c)	Is c an upper case letter?
tolower(c)	Convert c to lower case
toupper(c)	Convert c to upper case

program:

```

void main()
{
char a,x;
printf("enter char");
a = getchar();
if (isupper(a))
{
x= tolower(a);
putchar(x);
}
else
putchar(toupper(a));
}
    
```

Output:- enter char A
a

Formatted I/O Functions:

Formatted I/O refers to input and output that has been arranged in a particular format.

Formatted I/P functions ----- scanf(), fscanf()

Formatted O/P functions ----- printf(), fprintf()

scanf() :-scanf() function is used to read information from the standard I/P device.

Syntax:-scanf("controlstring", &variable_names);

Control string (also known as format string) represents the type of data that the user is going to accept and gives the address of variable.

(char-%c , int-%d , float - %f , double-%lf).

Control string and variables are separated by commas. Control string and the variables going to I/P should match with each other.

Ex:-int n;

scanf("%d",&n);

Inputting Integer Numbers:



The field specification for reading an integer number is: %w d

The percentage sign(%) indicates that a conversion specification follows. w is an integer number that specifies the field width of the number to be read and d known as data type character indicates that the number to be read is in integer mode.

Ex:-scanf(“%2d,%5d”,&a,&b);

The following data are entered at the console: 50

31426

Here the value 50 is assigned to a and 31426 to b

Inputting Real Numbers:

The field width of real numbers is not to be specified unlike integers. Therefore scanf reads real numbers using the simple specification %f.

Ex: scanf(“%f %f”,&x,&y);

Suppose the following data are entered as input: 23.45 34.5 The value 23.45 is assigned to x and 34.5 is assigned to y.

Inputting character strings:

The field specification for reading character strings is %ws or %wc

%c may be used to read a single character.

Ex:scanf(“%s”,name1);

Suppose the following data is entered as input:Griet

Griet is assigned to name1.

Formatted Output:

printf(): This function is used to output any combination of data. The outputs are produced in such a way that they are understandable and are in an easy to use form. It is necessary for the programmer to give clarity of the output produced by his program.

Syntax:- printf(“control string”, var1,var2.....);

Control string consists of 3 types of items

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

Programming for problem solving[CS1103ES]

The control string indicates how many arguments follow and what their types are.

The var1, var2 etc..are variables whose values are formatted and printed according to the specifications of the control string.

The arguments should match in number, order and type with the format specifications

O/P of integer number: - Format specification : %wd

Format	O/P
Print("%d", 9876);	9 8 7 6
Printf("%6d",9876);	9 8 7 6
<u>printf</u> ("%d",9876);	9 8
Print("%6d",9876);	9 8 7 6
Print("%06d",9876);	0 0 9 8 7 6

O/P of real number: %w.pf

w---- Indicates minimum number of positions that are to be used for display of the value.

p---- Indicates number of digits to be displayed after the decimal point.

Format y=98.7654

O/P

<u>printf</u> (-%7.4fl,y)	9 8 . 7 6 5 4
<u>printf</u> (-%7.2fl,y)	9 8 . 7 7
<u>printf</u> (-%-7.2fl,y)	9 8 . 7 7
<u>printf</u> (-%-10.2el,y)	9 . 8 8 e + 0 1

O/P of single characters and string:

%wc

%ws

Format

O/P

<u>%s</u>	R a J U R A j e s h R a N i
<u>%18s</u>	R a J U R A J E s h R a n i
<u>%18s</u>	R A j U R a J E S h R a n i



UNIT-II

Arrays: one- and two-dimensional arrays, creating, accessing and manipulating elements of arrays
Strings: Introduction to strings, handling strings as array of characters, basic string functions available in C (strlen, strcat, strcpy, strstr etc.), arrays of strings

Structures: Defining structures, initializing structures, unions, Array of structures

Pointers: Idea of pointers, Defining pointers, Pointers to Arrays and Structures, Use of Pointers in self-referential structures, Enumeration data type.

Arrays

Introduction :

The fundamental data types, namely char, int, float, double and variations of int and double can store only one value at any given time. Therefore they can be used only to handle limited amounts of data. In many applications, we need to handle a large volume of data in terms of reading, processing and

printing. To process such large volume of data C supports a derived data type known as ARRAY that facilitate efficient storing, accessing and manipulation of data items.

Definition :

An array is a fixed-size sequenced collection of elements of the same data type.

Examples where the concept of an array can be used are :

- List of employees in an organization
- List of products and their cost
- Test scores of a class of students.
- List of customers and their telephone numbers.etc..

Since an array provides a convenient structure for representing data, it is classified as one of the data structures in C.

Types of Arrays :

1. One-dimensional arrays
2. Two-dimensional arrays
3. Multi-dimensional arrays.

ONE-DIMENSIONAL ARRAYS:

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

DECLARATION OF ONE-DIMENSIONAL ARRAYS:

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

Syntax: type variable-name[size];

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

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

For e.g.-

```
int number[5];
```

Declares the number to be an array containing 5 elements. Any subscripts 0 to 4 are valid and the computer reserves five storage locations as shown below

	number[0]
	number[1]
	number[2]
	number[3]
	number[4]

Note :

1. any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.
2. The size should be either a numeric constant or a symbolic constant.

INITIALIZATION OF ONE-DIMENSIONAL ARRAYS:

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

1. At Compile time
2. At run time

COMPILE TIME INITIALIZATION :

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

Syntax:

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

the values in the list are separated by commas. For example, the statement `int number[5]={35,40,50,20,25};` will declare the variable number as an array of size 5 and will assign the value to each element as follows

number[0]	35
number[1]	40
number[2]	50
number[3]	20
number[4]	25

If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically if the array type is numeric and NULL if the type is char. For e.g.- `float total[5]={0.0,5.2,-10};`

Will initialize the first three elements to 0.0, 5.2 and -10.0 and the remaining two elements to zero.

If we have more initializers than the declared size, the compiler will produce an error.

`int number[3]={10,20,30,40};` is illegal in C.

The **size** may be omitted in the declaration statement. In such cases, the compiler allocates enough space for all initialized elements.

For e.g.- `int count[]={1,1,1,1};`

Will declare the count array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

Character arrays may be initialized in a similar manner. Thus the statement

`char name[]={‘h’,‘e’,‘l’,‘l’,‘o’,‘\0’};`

Declares the name to be array of five characters, initialized with the string “hello” ending

with the null character. Alternatively we can assign the string literal directly as

```
char name[]="hello";
```

RUN TIME INITIALIZATION

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. For example, consider the following segment of C program

```
for(i=0;i<100;i=i+1)
{
if(i<50)
sum[i]=0.0;
else
sum[i]=1.0;
}
```

The first 50 elements of the array sum are initialized to 0 while the remaining 50 are initialized to 1.0 at run time.

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

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

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

EXAMPLE PROGRAMS:

write a program to evaluate the following expression for $\sum x^2$ i

```
program :void main()
{
int i;
float x[10],total;
printf("enter the 10 real numbers");
for(i=0;i<10;i++)
scanf("%f", &x[i]);
total=0.0;
for(i=0;i<10;i++)
total=total+x[i]*x[i];
for(i=0;i<10;i++)
printf("x[%d]=%f\n",i,x[i]);
printf("\n sum of the squares =%f",total);
}
```

TWO- DIMENSIONAL ARRAYS:

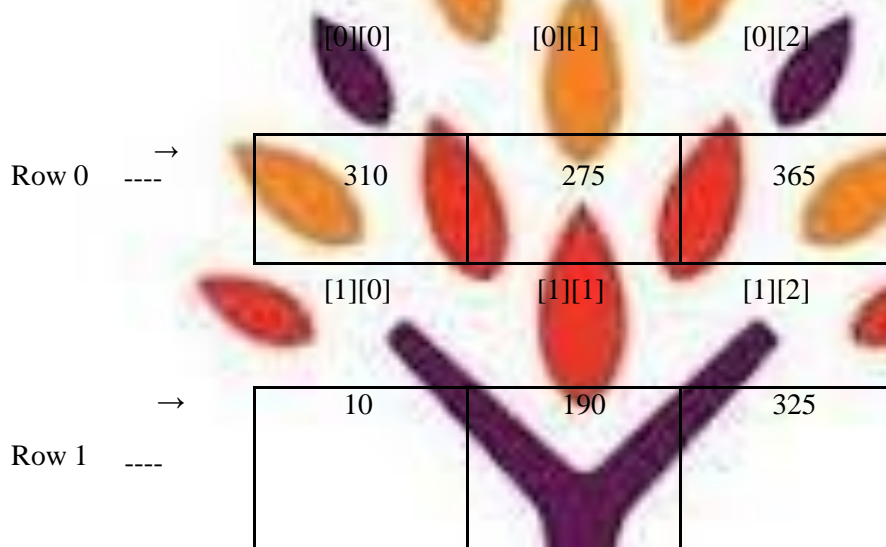
1D array variables stores a list of values only but there could be a situations where a table of values have to be stored which need 2D arrays. For examples

- Periodictable
- Sales information of a company
- Data in spread sheetsetc.

The two dimensional (2D) array in C programming is also known as matrix. A matrix can be represented as a table of rows and columns. A particular value in a matrix can be accessed by using two subscripts such as V_{ij} . here V denotes the entire matrix and V_{ij} refers to the value in the i^{th} row and j^{th} column. An array of arrays is known as 2D array.

Syntax: `type array-name[row_size][column_size];`

2D arrays are stored in memory as shown below:



INITIALIZING 2D ARRAYS:

Like the 1D arrays, 2D arrays can be initialized by following their declaration with a list of initial values enclosed in braces. For example

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

Initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement is equivalently written as `int table[2][3]={{0,0,0},{1,1,1}}`;

by surrounding the elements of each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below: `int`

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

When the array is completely initialized with all values, explicitly, we need not specify the size of the first dimension. That is, the below statement is permitted.

```
int table[][3]= {
    {0,0,0},
```

```
    {1,1,1}
    };
```

If the values are missing in an initialize, they are automatically set to zero. For example `int table[2][3]={ {1,1},{2}}`;

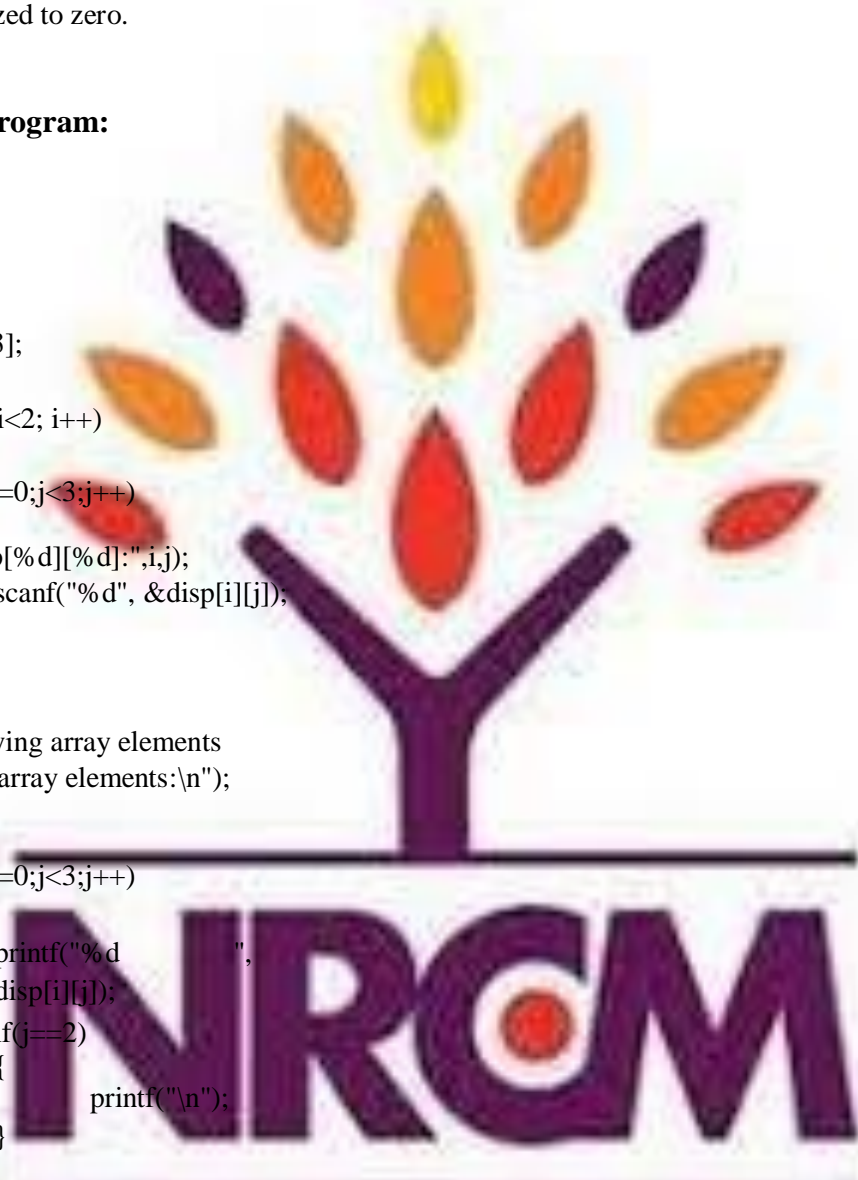
Will initialize the first two elements of the first row to one, the first element of the second row to two, and all other elements to zero. When all the elements are to be initialized to zero, the following short-cut method may be used.

```
int m[3][5]={0},{0},{0};
```

the first element of each row is explicitly initialized to zero while other elements are automatically initialized to zero.

Example program:

```
#include<stdio.>
void main()
{
    int
    disp[2][3];
    int i, j;
    for(i=0; i<2; i++)
    {
        for(j=0; j<3; j++)
        {
            printf("Enter value for disp[%d][%d]:", i, j);
            scanf("%d", &disp[i][j]);
        }
    }
    //Displaying array elements
    printf("Two Dimensional array elements:\n");
    for(i=0; i<2; i++)
    {
        for(j=0; j<3; j++)
        {
            printf("%d\t",
            disp[i][j]);
            if(j==2)
            {
                printf("\n");
            }
        }
    }
}
```



Multidimensional arrays :

C allows arrays of three or more dimensions. The exact limit is determined by the compiler.

Syntax:

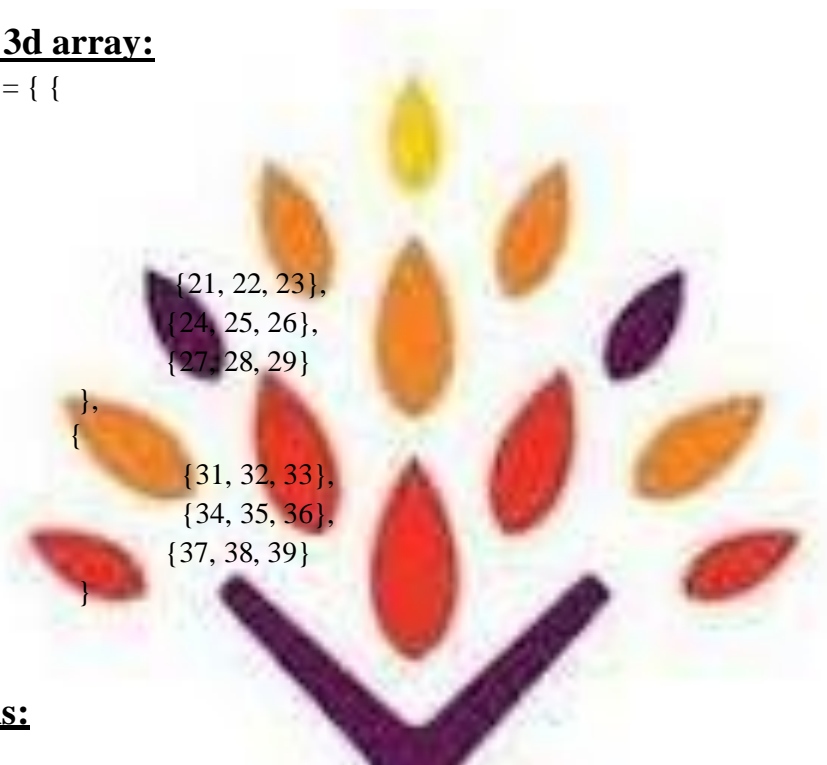
typearray-name[s1][s2][s3]----- [sm];Where si is the size of the ith dimension.

Eg : int a[3][3][3];
int b[5][4][5][3];

A 3D array is essentially an array of arrays of arrays: it's an array or collection of 2D arrays, and a 2D array is an array of 1D array.

Initializing 3d array:

```
int arr[3][3][3]= { {  
  {11, 12, 13},  
  {14, 15, 16},  
  {17, 18, 19}  
}, {  
  {21, 22, 23},  
  {24, 25, 26},  
  {27, 28, 29}  
},  
 {  
  {31, 32, 33},  
  {34, 35, 36},  
  {37, 38, 39}  
}  
};
```



Example Programs:

1. Develop a program to Find the Largest Two Numbers in a given Array and also calculate their average

```
#include <stdio.h>  
#define MAX 4  
void main()  
{  
  int array[MAX], i, largest1, largest2, temp;  
  printf("Enter %d integer numbers \n", MAX);  
  for (i = 0; i < MAX;i++)  
  {  
    scanf("%d", &array[i]);  
  }  
  printf("Input interger are \n");  
  for (i = 0; i < MAX;i++)  
  {  
    printf("%5d", array[i]);  
  }  
  printf("\n");  
  /* assume first element of array is the first  
  larges t*/ largest1 =array[0];  
  /* assume first element of array is the second largest */  
  largest2 = array[1];
```

```

if (largest1 < largest2)
{
    temp = largest1;
    largest1 = largest2;
    largest2 = temp;
}
for (i = 2; i<4; i++)
{
    if (array[i] >= largest1)
    {
        largest2 = largest1;
        largest1 = array[i];
    }
    else if (array[i] > largest2)
    {
        largest2 = array[i];
    }
}
printf("n%d is the first largest \n", largest1);
printf("n%d is the second largest \n", largest2);
printf("nAverage of %d and %d = %d \n", largest1, largest2,(largest1 + largest2) / 2);
}

```

Output:

Enter 4 integer numbers 80

23

79

58

Input integer are

80 23 79 58

80 is the first largest 79

is the secondlargest

Average of 80 and 79 = 79



2.Implement a program to insert an elements of an Array in the requiredposition

```

#include <stdio.h>

int main()
{
    int array[100], position, c, n, value;
    printf("Enter number of elements in array\n");
    scanf("%d", &n);
    printf("Enter %d elements\n", n);
    for (c = 0; c < n; c++)

```

```

scanf("%d", &array[c]);
printf("Enter the location where you wish to
insert anelement\n");
scanf("%d", &position);
printf("Enter the value to insert\n");
scanf("%d",&value);
for (c = n - 1; c >= position - 1; c--)
array[c+1] = array[c];
array[position-1] = value;
printf("Resultant array is\n");
for (c = 0; c <= n;c++)
printf("%3d", array[c]);
return 0;
}

```

Output:

```

Enter the value of the n = 4
Enter the numbers 3
40
100
68
Enter the location where you wish to insert an element 2
Enter the value to insert 35

```

```

3    35    40    100    80

```

3.Develop a program to reverse anarray

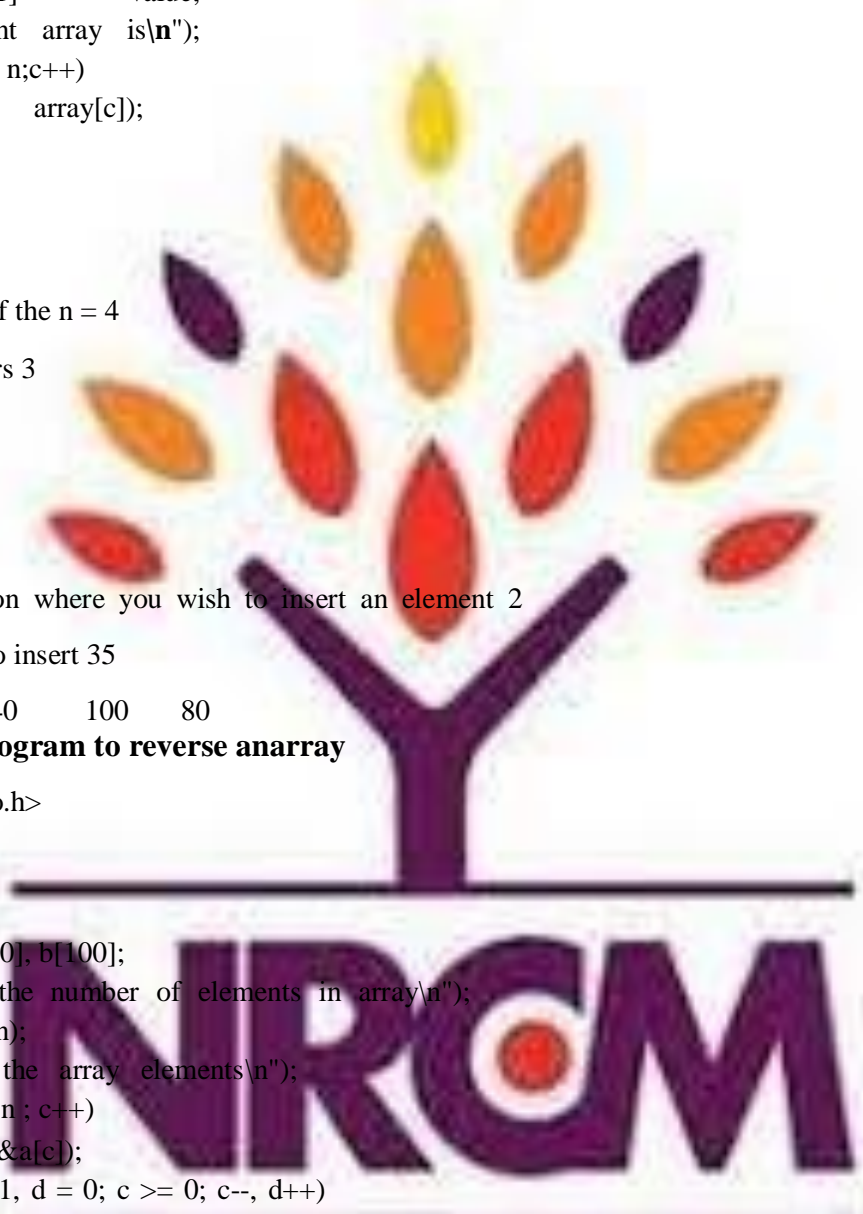
```

#include <stdio.h>

void main()
{
int n, c, d, a[100], b[100];
printf("Enter the number of elements in array\n");
scanf("%d", &n);
printf("Enter the array elements\n");
for (c = 0; c < n; c++)
scanf("%d", &a[c]);
for (c = n - 1, d = 0; c >= 0; c--, d++)
b[d] = a[c];
for (c = 0; c < n; c++)
a[c] = b[c];

printf("Reverse array is\n");
for (c = 0; c < n; c++)
printf("%d\n",a[c]);
}

```



4. Program to add two matrices

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[5][5],b[5][5],c[5][5],r1,r2,c1,c2,i,j;
clrscr();
printf("\n enter r1,c1,r2,c2");
scanf("%d %d %d %d",&r1,&c1,&r2,&c2);
if(r1==r2&&c1==c2)
{
printf("\n matrix addition possible");
for(i=0;i<r1;i++)
{
for(j=0;j<c1;j++)
{
printf("\n enter a[%d][%d]:",i,j);
scanf("%d",&a[i][j]);
}
}
for(i=0;i<r2;i++)
{
for(j=0;j<c2;j++)
{
printf("\n enter b[%d][%d]:",i,j);
scanf("%d",&b[i][j]);
}
}
for(i=0;i<r1;i++)
{
for(j=0;j<c1;j++)
{
c[i][j]=a[i][j]+b[i][j];
}
}
printf("\n sum of matrix c:"); for(i=0;i<r1;i++)
{
printf("\n");
for(j=0;j<c1;j++)
{
printf("\t%d",c[i][j]);}
}
}
else
{
printf("\n matrix multiplication is not possible");
}
getch();
}
```



5. Program to multiply twomatrices

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



```

#include<conio.h>
void main()
{
int a[5][5],b[5][5],c[5][5],r1,r2,c1,c2,i,j,k;
clrscr();
printf("\n enter r1,c1,r2,c2");
scanf("%d %d %d %d",&r1,&c1,&r2,&c2);
if(r1==r2&&c1==c2)
{
printf("\n matrix mutliplication possible");
for(i=0;i<r1;i++)
{
for(j=0;j<c1;j++)
{
printf("\n enter a[%d][%d]:",i,j);
scanf("%d",&a[i][j]);
}
}
for(i=0;i<r2;i++)
{
for(j=0;j<c2;j++)
{
printf("\n enter b[%d][%d]:",i,j);
scanf("%d",&b[i][j]);
}
}
for(i=0;i<r1;i++)
{
for(j=0;j<c2;j++)
{
c[i][j]=0;
for(k=0;k<c1;k++)
{
c[i][j]=c[i][j]+a[i][k]*b[k][j];
}
}
}
printf("\n product of matrix c:");
for(i=0;i<r1;i++)
{
printf("\n");
for(j=0;j<c2;j++)
{
printf("\t%d",c[i][j]);
}
}
}
else
{
printf("\n matrix multiplication is not possible");
}
getch();
}

```



6. Program to display the transpose of a matrix

```
#include<stdio.h>
#include<conio.h>
void main()
{
int row,col,a[5][5],i,j,b[5][5];
clrscr();
printf("\n enter order of matrix r,c:"); scanf("%d
%d",&row,&col); for(i=0;i<row;i++)
{
for(j=0;j<col;j++)
{
printf("\n enter a[%d][%d]:",i,j);
scanf("%d",&a[i][j]);
}
}
for(i=0;i<row;i++)
{
for(j=0;j<col;j++)
{
b[j][i]=a[i][j];
}
}
printf("\n transpose matrix b");
for(i=0;i<col;i++)
{
printf("\n");
for(j=0;j<row;j++)
{
printf("\t %d",b[i][j]);

```



STRINGS

INTRODUCTION:

A string is a sequence of characters that is treated as a single data item. Any group of characters defined between double quotation marks is a string constant.

Example : “ hello world”

“123ase\$”

The common operations performed on character strings include:

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

DECLARING AND INITIALIZING STRING VARIABLES:

C does not support strings as a data type. However, it allows us to represent strings as character arrays. The general form of declaration of a string variable is: **char string_name[size];** the size

determines the number of characters in the string_name.

Example: `char city[10];`
`char name[30];`

when the compiler assigns a character string to a character array, it automatically supplies a null character('\0') at the end of the string. Therefore, the size should be equal to the maximum number of characters in the string plus one.

Like numeric arrays, character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms:

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

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

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

For example: `char string[]={ 'G','O','O','D','\0'}`; defines the array string as a five element array.

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

`char str[10]="GOOD";` is permitted. In this case, the computer creates a character array of size 10, places the value "GOOD" in it, terminates with the null character, and initializes all other elements to NULL.

The storage will look like:

G	O	O	D	\0	\0	\0	\0	\0	\0
---	---	---	---	----	----	----	----	----	----

However, the following declaration is illegal

`char str2[3]="GOOD";` This will result in a compile time error.

Note: we cannot separate the initialization from declaration. That is, `char str[5]; Str="GOOD"` is not allowed.

READING STRINGS FROM TERMINAL:

Using scanf function:

using %s, it automatically terminates the string with '\0' character, so the array size should be large enough to hold the input plus null character.

e.g.- `char name[10];`
`scanf("%s",name);`

Note: in the case of character arrays, the ampersand(&) is not required before the variable name.

Disadvantage: scanf terminates its input on the first white space it finds. e.g.- if enters new york then only new will be read in array name. We can also specify the field width using the form %ws in scanf for reading a specified number of characters from the input string. E.g.- `scanf("%ws",name);`

If w is equal or greater than the number of characters typed, then entire string will be stored in the variable

If w is less, then excess characters will be truncated and left unread.

Example: `scanf("%6s",name)`

griet---reads entire string

gokaraju---reads only gokara

Reading a Line of Text:

%s and %ws can read only strings without whitespaces. That is, they cannot be used for reading a text containing more than one word. However, C supports a format specification known as the **edit set conversion code** %[.] that can be used to read a line containing a variety of characters, including whitespaces.

For example:

```
Char    line[80];
scanf("%[^\n]",line);
printf("%s",line);
```

will read a line of input from the keyboard and display the same on the screen.

Using getchar and gets function:

Function getchar is used to read single character from the terminal. This function repeatedly can be used to read successive single characters from the input and place them into a character array. Thus, an entire line of text can be read, the reading is terminated when the newline character ('\n') is entered and the null character is then inserted at the end of the string.

Syntax: char ch;
ch=getchar();

Example program:

```
#include<stdio.h>
void main()
{
    char line[50],ch;
    int i=0;
    printf("\n press enter at the end of input"); do
    {
        ch=getchar();
        line[i]=ch; i++;
    }while(ch!='\n');
    i=i-1; line[i]='\0';
    printf("%s",line);
}
```

another more convenient method of reading a string of text containing whitespaces is to use the library function **gets** available in the <stdio.h> header file. **syntax:** gets(str);

str is a string variable, gets function reads a characters into str from the keyboard until a newline character is encountered and then appends a null character to the string. unlike scanf, it does not skip whitespaces.

Example: char line[80];
 gets(line);
 printf("%s",line);

the last two statements can be combined as follows printf("%s",gets(line));

WRITING STRINGS TO SCREEN

Using printf function

The format %s can be used to display an array of characters that is terminated by the null character using printf function.

```
printf("%s",line);
```

We can also specify the precision with which the array is displayed. for instance, the specification %10.4 indicates that the *first four* characters are to be printed in a field width of 10 columns. left-justified.

Features of %s specifications:

1. When the field width is less than the length of the string, the entire string is printed.
2. The integer value on the right side of the decimal point specifies the number of characters to be printed.
3. When the number of characters to be printed is specified as zero, nothing is printed.
4. The minus sign in the specification causes the string to be printed left-justified.
5. The specification %.ns prints the first n characters of the string.

Example program:

Write a C program to store the string and display the string under various format specifications.

```
void main()
{
char str[14]="good morning";
printf("%14s",str);
printf("%4s",str);
printf("%14.4s",str);
printf("%-14.4s",str);
printf("%14.0s",str);
printf("%.3s",str);
printf("%s",str);
}
```

Output:

```
good morning
.....good
good.....
goo
good morning
```



Using putchar and puts functions:

putchar function is used to output the values of character variables.

syntax:

```
char ch='a'; putchar(ch);
```

The function putchar requires one parameter. This statement is equivalent to: printf("%c",ch); we can use this function repeatedly to output a string of characters stored in an array using a loop.

Example:

```
char str[13]="good morning";

for(i=0;i<12;i++)

putchar(name[i]);
```

Another and more convenient way of printing string values is to use the function puts declared in the header file <stdio.h>.

syntax:

```
puts(str);
```

where **str** is a string variable containing a string value, this prints the value of the string variable **str** and then moves the cursor to the beginning of the next line on the screen.

ARITHMETIC OPERATIONS ON CHARACTERS:

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into an integer value by the system. For e.g.- x='a';

printf(“%d”,x); will display the number 97 on the screen. x='a'-1;

printf(“%d”,x); will print 96, a=97

character constants can also be used in relational expressions.

Example: ch>='A' && ch<='Z' tests whether the character in the variable ch is an upper-cas letter.

We can convert a character digit to its equivalent integer value using x= character-'0' .-

Example:

```
int x ;
```

```
x= '7'-'0';
```

```
printf(“%d”,x);
```

prints x=7 // ASCII value of 7 = 55

ASCII value of 0=48

The C library supports a function atoi (string) that converts a string of digits into their integer values.

Example: char [10] number =“1988”;

```
year=atoi(number);
```

The function converts the string constant “1988”, to its numeric equivalent 1988.

Present in header file <stdlib.h>.C does not provide operators that work on strings directly like :

```
str1=“ABC”;
```

```
str2=str1; is invalid
```

STRING HANDLING FUNCTIONS

The C library supports a large number of string-handling functions that can be used to carry out many of the string manipulations. following are the most commonly used string-handling functions.

Function	Action
Strcat()	Concatenates two strings
Strcmp()	Compares two strings
Strcpy()	Copies one string over another
Strlen()	Finds the length of a string

Strrev()	Reverse a string
----------	------------------

strcat() Function:

The strcat function joins two strings together. it takes the following form:

strcat(string1, string2);

string1 and string2 are character arrays. when the function strcat is executed, string2 is appended to string1. It does by removing the null character at the end of string1 and placing string2 from there. the string at the string2 remains unchanged. For example consider the following three strings:

str1	V	E	R	Y	\0		
------	---	---	---	---	----	--	--

str2	G	O	\0		
------	---	---	----	--	--

strcat(str1,str2) will result in:

V	E	R	Y		G	O	\0
---	---	---	---	--	---	---	----

We must make sure that the size of str1 to which str2 is appended is large enough to accommodate the final string.strcat function may also append a string constant to a string variable. the following is valid:

strcat(str1,"good");

C permits nesting of strcat functions. for example, the statement *strcat(strcat(str1,str2),str3);* is allowed and concatenates all the three strings together. the resultant string is stored in str1.

strcmp() Function:

The strcmp function compares two strings identified by the arguments and has a value 0 if they are equal. if they are not, it has the numeric difference between the first nonmatching characters in the strings. it takes the form:**strcmp(str1,str2);**str1 and str2 may be string variables or string constants.

Example: strcmp(str1,str2);
 strcmp(str1, "john");
 strcmp("rom", "ram");

strcpy() Function:

The strcpy() function works almost like a string-assignment operator. it takes a form: **strcpy(str1,str2);**and assigns the contents of str2 to str1. str2 may be a character array variable or a string constant.

Example:strcpy(city, "DELHI");

will assign the string "DELHI" to the string variable city. similarly,the statement strcpy(city1,city2) will assign the contents of the string variable city2 to the string variable city1. the size of the array city1 should be large enough to store the contents of city2.

strlen() Function:

This function counts and returns the number of characters in a string. it takes the formn=strlen(string); where n is an integer variable, which receives the value of the length of the string.

Example :

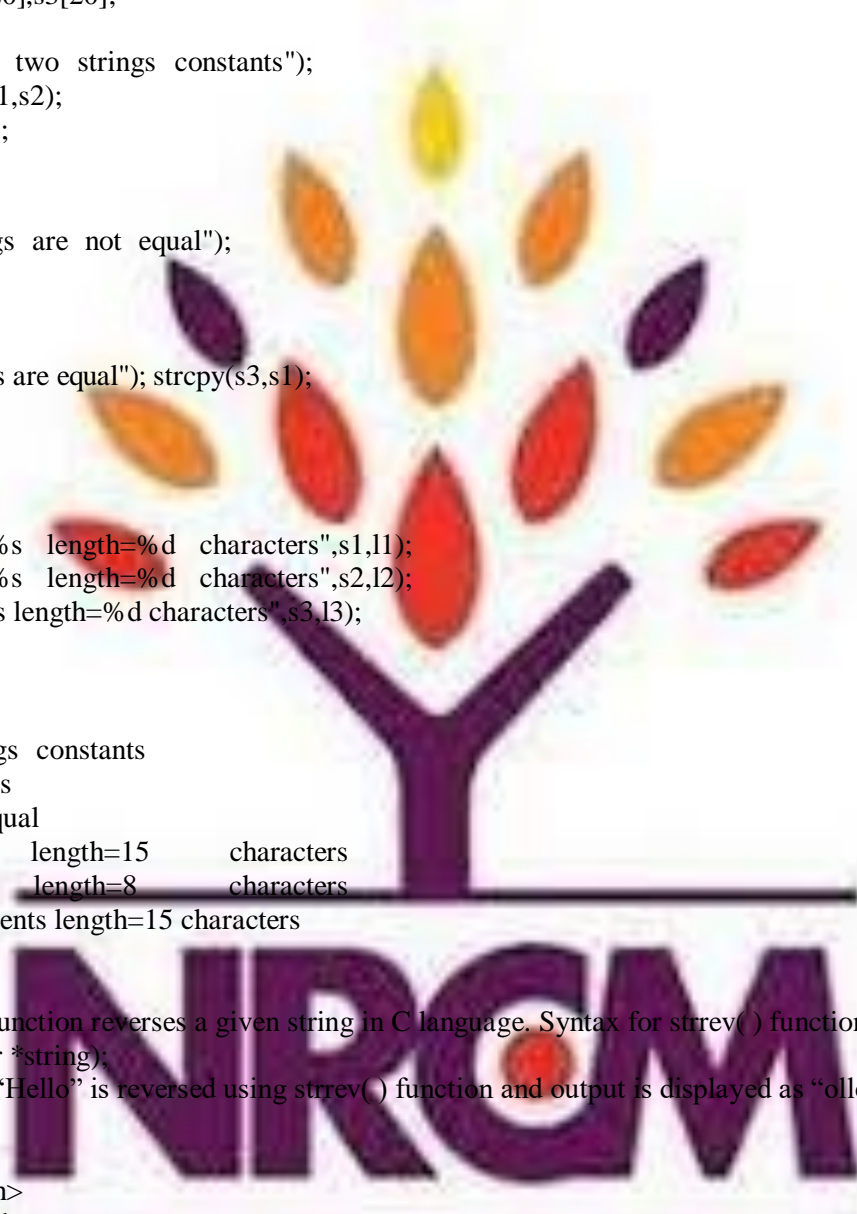
write a program that reads 2 strings s1,s2, and check whether they are equal or not, if they are not , join them together. then copy the contents of s1 to the variable s3. at the end , the program should print the contents of all the 3 variables and their lengths.

```
#include<stdio.h>
void main()
{
char s1[20],s2[20],s3[20];
int x,l1,l2,l3;
printf("\n enter two strings constants");
scanf("%s%s",s1,s2);
x=strcmp(s1,s2);
if(x!=0)
{
printf("\n strings are not equal");
strcat(s1,s2);
}
else
printf("\n strings are equal"); strcpy(s3,s1);
l1=strlen(s1);

l2=strlen(s2);
l3=strlen(s3);
printf("\n s1=%s length=%d characters",s1,l1);
printf("\n s2=%s length=%d characters",s2,l2);
printf("\n s3=%s length=%d characters",s3,l3);
}
```

output:

```
enter two strings constants
welcome students
strings are not equal
s1=welcome      length=15      characters
s2=students     length=8       characters
s3=welcomestudents length=15 characters
```



strrev() function:

strrev() function reverses a given string in C language. Syntax for strrev() function is given below.

```
char *strrev(char *string);
```

example: string "Hello" is reversed using strrev() function and output is displayed as "olleH".

Program:

```
#include <stdio.h>
#include <string.h>
void main()
{
char s[100];
printf("Enter a string to reverse:\n");
gets(s);
strrev(s);
printf("Reverse of the string: %s\n", s);
}
```


output:

Enter a string to reverse: rajesh
Reverse of the string: hsejar

Strings with out string handling function:

string length:

```
#include<stdio.h>
void main()
{
char str[30];
int i,len=0;
clrscr();
printf("\n enter the string");
gets(str);
for(i=0;str[i]!='\0';i++)
len++;
printf("\n the total no.of characters in the given string is:%d",len-1);
getch();
}
```

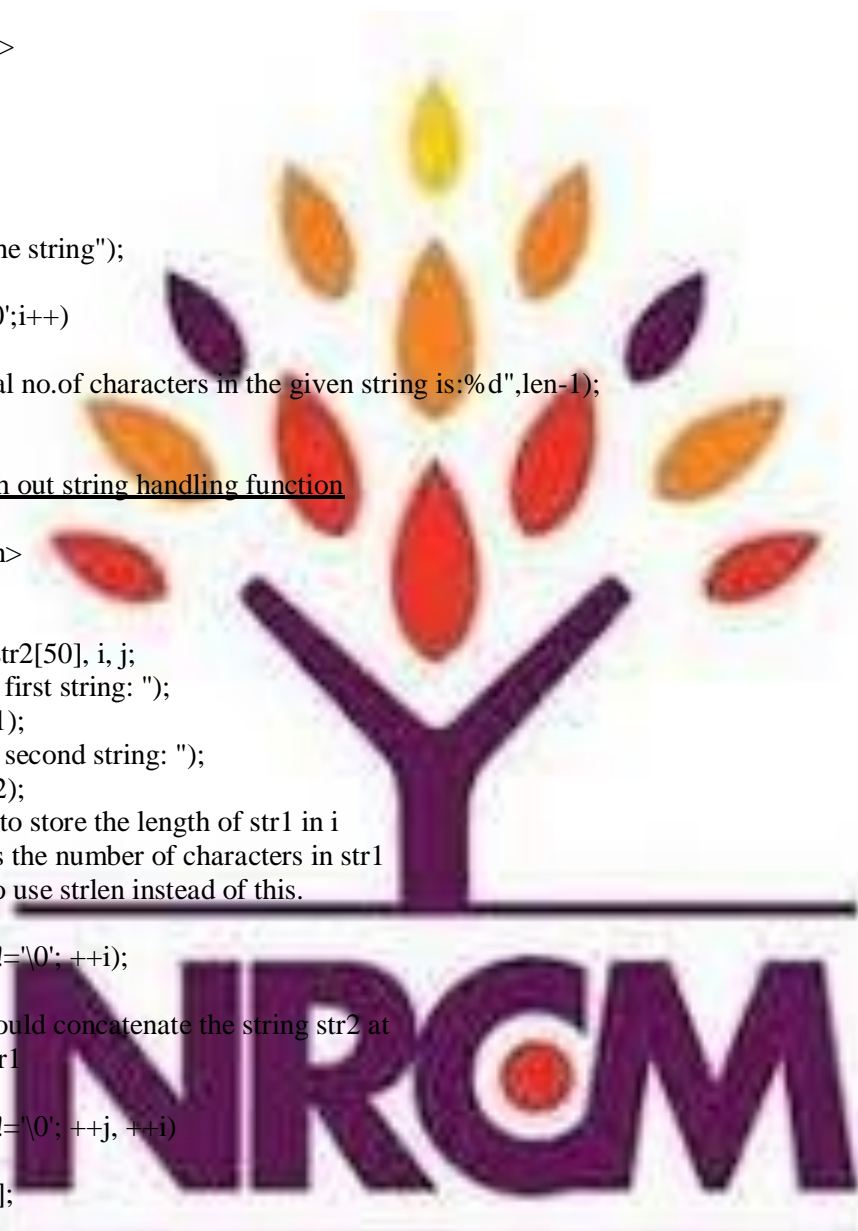
//String concatenation with out string handling function

```
#include <stdio.h>
void main()
{
char str1[50], str2[50], i, j;
printf("\nEnter first string: ");
scanf("%s",str1);
printf("\nEnter second string: ");
scanf("%s",str2);
/* This loop is to store the length of str1 in i
* It just counts the number of characters in str1
* You can also use strlen instead of this.
*/
for(i=0; str1[i]!='\0'; ++i);

/* This loop would concatenate the string str2 at
* the end of str1
*/
for(j=0; str2[j]!='\0'; ++j, ++i)
{
str1[i]=str2[j];
}
// \0 represents end of string
str1[i]='\0';
printf("\nOutput: %s",str1);
}
```

// string copy with out string handling function

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



```

{
char str1[30],str2[30];
int i;
printf("enter the string 1:");
gets(str1);
for(i=0;str1[i]!='\0';i++)
str2[i]=str1[i];
str2[i]='\0';
printf("the string after copy is :");
puts(str2);
getch();
}

```

// string reverse with out string handling function

```

#include<stdio.h> void main()
{
char str1[20],str2[20];
int i,len=1,j=0;
printf("\n enter the string");
scanf("%s",str1);
for(i=0;str1[i]!='\0';i++)
{
len++;
}
printf("the length of the string is:%d",len);
for(i=len-1;i>=0;i--,j++)
str2[j]=str1[i];
str2[j]='\0';
printf("the reverse of a given string is %s",str2); getch();}

```

String comparison with out string handling function

```

#include<stdio.h>
void main()
{
char str1[30],str2[30];
int i,j,flag=0;
printf("enter the strings to be compared:");
scanf("%s %s",str1,str2);
for (i=0,j=0;str1[i]!='\0' || str2[i]!='\0';i++,j++)
{
if(str1[i]!=str2[j])
{
flag=1; break;
}
}
if(flag==0)
printf("the two strings are equal");
else
printf("the two strings are not equal");
getch();
}

```

Array of Strings:



We often use list of character strings, such as a list of the names of students in a class, list of the names of employees in an organization etc. A list of names can be treated as a table of strings and a two-dimensional character array can be used to store the entire list.

for example, a character array city[5][15] may be used to store a list of 5 city names, each of length not more than 15 characters. e.g. -

C	H	A	N	D	I	G	A	R	h	\0
M	A	D	R	A	S	\0				
A	H	M	E	D	A	B	A	D	\0	
H	Y	D	E	R	A	B	A	D	\0	
B	O	M	B	A	Y	\0				

This table can be conveniently stored in a character array city by using the following declaration:

```
char city[ ][15]=
{ "CHANDIGARH", "MADRAS", "AHMEDABAD", "HYDERABAD", "BOMBAY"};
```

Example Programs:

C Program to Display the Characters in Prime Position a given String

```
#include <stdio.h>
#include <string.h>
void main()
{
int i, j, k, count = 0;
char str[50];
printf("enter string\n");
scanf("%s", str);
k = strlen(str);
printf("prime characters in a string are\n");
for (i = 2; i <= k; i++)
{
count = 0;
for (j = 2; j <= k; j++)
{
if (i % j == 0)
{
count++;
}
}
}

if (count == 1)
{
printf("%c\n", str[i - 1]);
}
}
}
```



```

    }
  }
  getch();
}

```

Structures

Definition: A structure is a collection of one or more variables of similar or different data types, grouped together under a single name. By using structures variables, arrays, pointers etc can be grouped together.

Structures can be declared using two methods as follows:

(i) Tagged Structure:

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

The **general form or syntax of tagged structure** definition is as follows,

```

struct TAG
{
Type variable1;
Type variable2;
.....
.....
Type variable-n;
};

```

Ex:- struct student

```

{
    int htno[10];
    char name[20];
    float marks[6];
};

```

Where,

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

(ii) Type-defined structures:-

The structure definition associated with the keyword **typedef** is called type-defined structure. This is the most powerful way of defining the structure.

The **syntax of typedef structure** is

```

typedef struct
{
Type variable1;
Type variable2;
.....
.....
}student;

```

Ex:- typedef struct

```

{
    int htno[10];
    char name[20];
    float marks[6];
}student;

```


Type variable-n;

}Type;

where

- A. typedef is keyword added to the beginning of the definition.
- B. struct is the keyword which tells the compiler that a structure is being defined.
- C. variable1, variable2...are called fields of the structure.
- D. The closing brace must end with type definition name which in turn ends with semicolon.

Variable declaration:

Memory is not reserved for the structure definition since no variables are associated with the structure definition. The members of the structure do not occupy any memory until they are associated with the structure variables.

After defining the structure, variables can be defined as follows:

For first method,

```
struct TAG v1,v2,v3....vn;
```

For second method, which is most powerful is,

```
Type v1,v2,v3,....vn;
```

Alternate way:

```
struct TAG
```

```
{  
Type variable1;
```

```
Type variable2;
```

```
.....
```

```
.....
```

```
Type variable-n; } v1, v2, v3;
```

Ex:

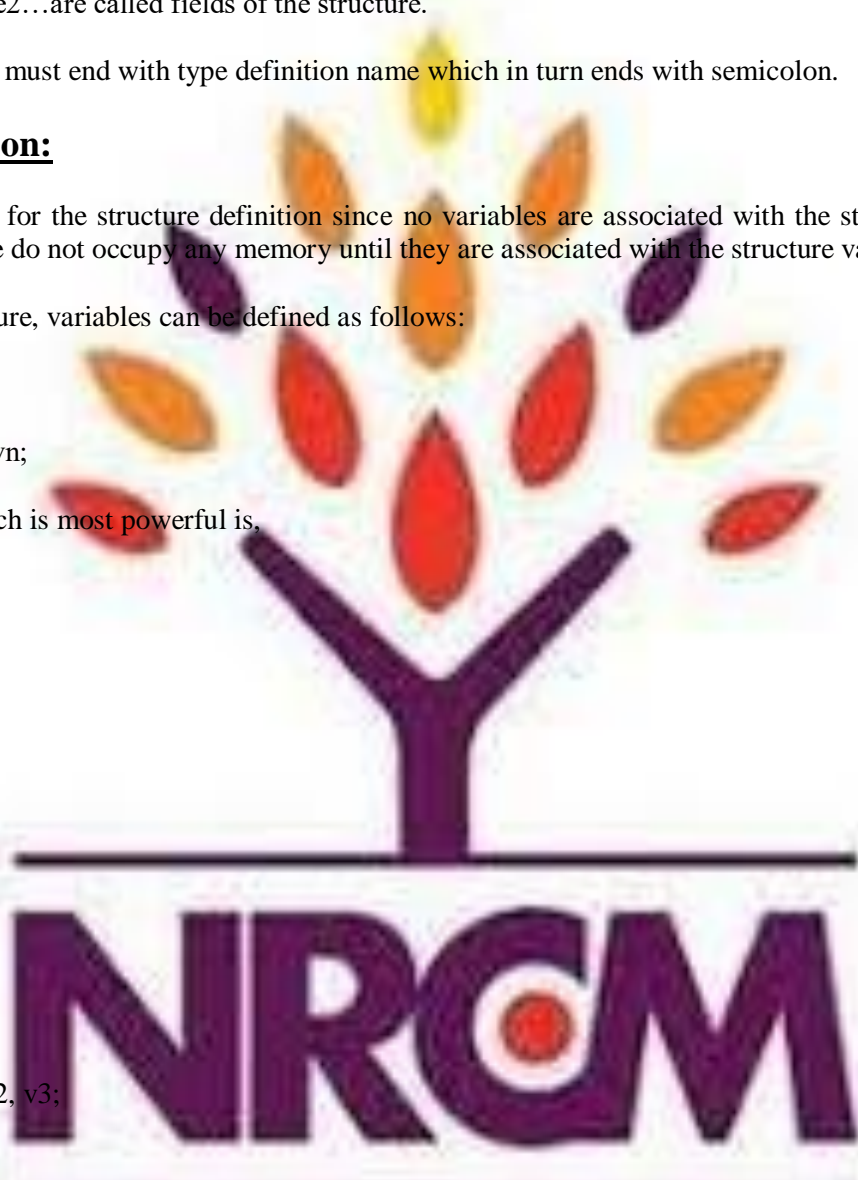
```
struct book
```

```
{
```

```
char name[30];
```

```
int pages;
```

```
float price; }b1,b2,b3;
```



Example for the following scenario:

College contains the following fields: College code (2characters), College Name, year of establishment, number of courses. Each course is associated with course name (String), duration, number of students. (A College can offer 1 to 50 such courses)

(i). Structure definition for college :-

```
struct college
{
    char code[2];
    char college_name[20];
    int year;
    int no_of_courses;
};
```

Variable declaration for structure college :-

```
void main( )
{
    struct college col1,col2,col3;
    ....
}
```

(ii). Structure definition for course :-

```
struct course
{
    char course_name[20];
    float duration;
    int no_of_students;
};
```

Variable declaration for structure course :-

```
void main( )
{
    struct course c1,c2,c3;
    ....
}
```

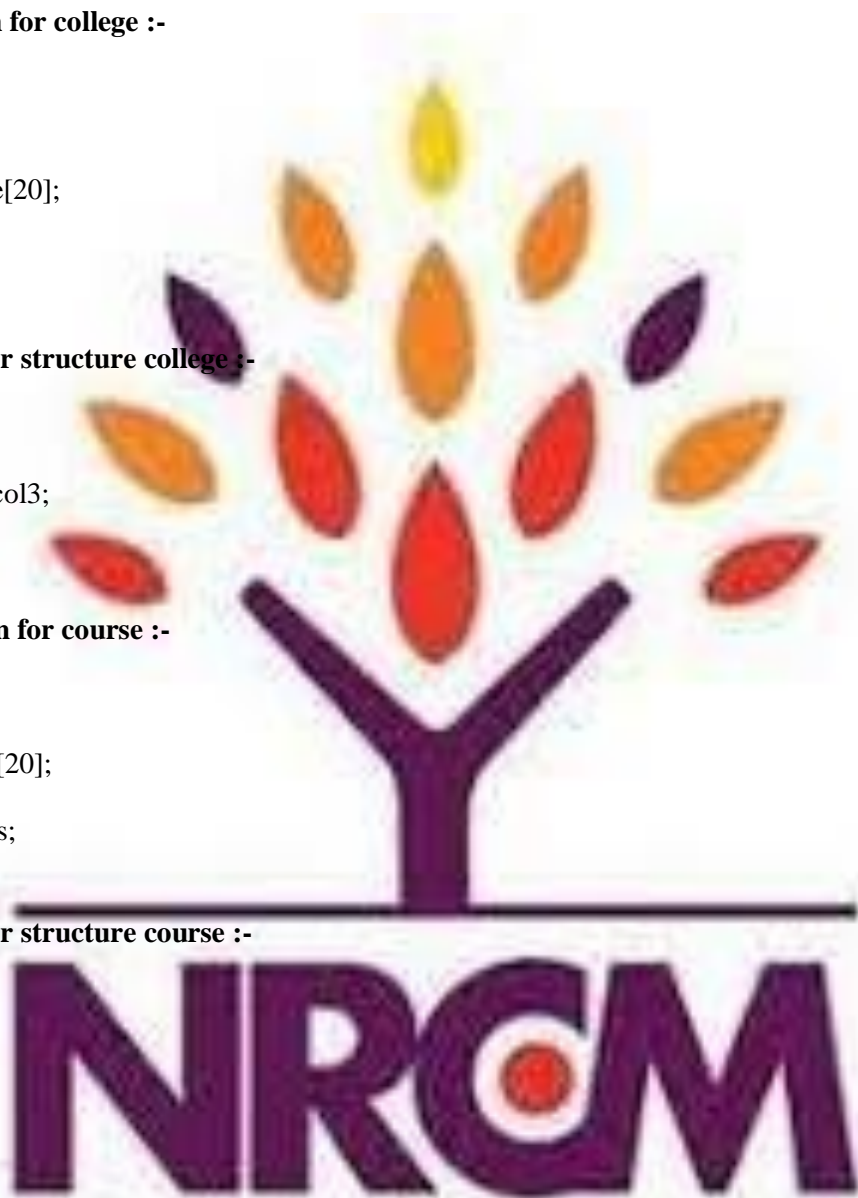
Initialization structures :

The rules for structure initialization are similar to the rules for array initialization. The initializers are enclosed in braces and separated by commas. They must match their corresponding types in the structure definition.

The syntax is shown below:

```
struct tag_name variable = {value1, value2,... value-n};
```

Structure initialization can be done in anyone of the following ways:



(i) Initialization along with Structure definition:-

Consider the structure definition for student with three fields name, roll number and average marks. The initialization of variable can be done as shown below,

```
struct student
{
char name [5];
int roll_number;
float avg;
} s1= {"Ravi", 10, 67.8};
```

The various members of the structure have the following values.

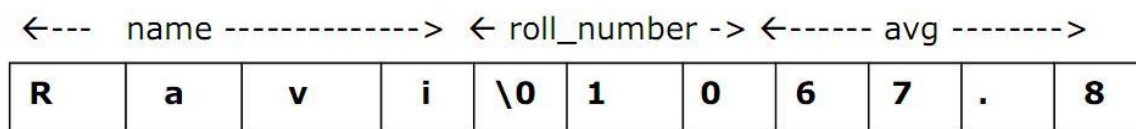


Figure 5.2 Initial Value of S1

(ii) Initialization during Structure declaration:-

Consider the structure definition for student with three fields name, roll number and average marks. The initialization of variable can be done as shown below:

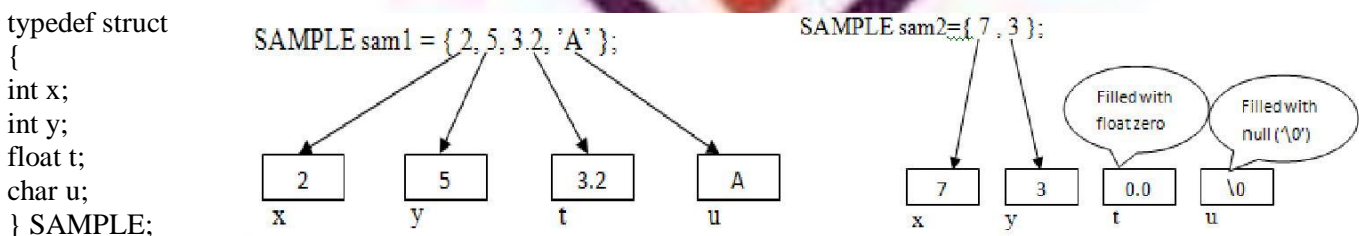


Figure: Initializing Structures

The figure shows two examples of structure in sequence. The first example demonstrates what happens when not all fields are initialized. As we saw with arrays, when one or more initializers are missing, the structure elements will be assigned null values, zero for integers and floating-point numbers, and null ('\0') for characters and strings.

Example: Construct a structure type *personal*, that would contain person name, date of joining and salary. Develop a program to initialize one person data and display the same.

```
struct personal
{
char name[20]; int day;
char month[10]; int year;
float salary;
};
void main( )
{
struct personal person = { "RAMU", 10 JUNE 1998, 20000};
```

```
printf("Output values are:\n");
printf("%s%d%s%d%f",person.name,person.day,person.month,person.year,person.salary );
}
```

Output:- RAMU 10 JUNE 1998 20000

Access the data for structure variables using member operator(.)

We know that variables can be accessed and manipulated using expressions and operators. On the similar lines, the structure members can be accessed and manipulated. The members of a structure can be accessed by using dot(.) operator.

Structures use a **dot (.) operator**(also called **period operator** or **member operator**) to refer its elements. Before dot, there must always be a structure variable. After the dot, there must always be a structure element.

The **syntax** to access the structure members as follows:

structure_variable_name . structure_member_name

Consider the example as shown below,

```
struct student
{
char name [5];
int roll_number;
float avg;
};
struct student s1= {"Ravi", 10, 67.8};
```

The members can be accessed using the variables as shown below,

```
s1.name --> refers the string "ravi"
s1.roll_number --> refers the roll_number 10
s1.avg --> refers avg 67.8
```

Example : Create a structure type *book*, that would contain book name, author, pages and price. Simulate a program to read this data using member operator (.) and display the same.

```
struct book
{
char name[20]; int day;
char month[10]; int year;
float salary;
};

void main()
{
struct book b1;
printf("Input values are:\n");
scanf("%s %s %d %f", b1.title,b1.author,b1.pages,b1.price);
printf("Output values are:\n");
printf("%s\n %s\n %d\n %f\n", b1.title, b1.author,b1.pages,b1.price);
}
```

Output:-

Input values are: C& DATA STRUCTURES KAMTHANE 609 350.00

Output values are:

```
C& DATA STRUCTURES KAMTHANE
609
350.00
```


Array of structures:

An array is a collection of elements of same data type that are stored in contiguous memory locations. A structure is a collection of members of different data types stored in contiguous memory locations. An array of structures is an array in which each element is a structure. This concept is very helpful in representing multiple records of a file, where each record is a collection of dissimilar data items.

As we have an array of integers, we can have an array of structures also. For example, suppose we want to store the information of class of students, consisting of name, roll_number and marks, A better approach would be to use an array of structures.

Array of structures can be declared as follows:**struct tag_name arrayofstructure[size];**

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

```
struct student
{
char name[10];
int rno;
float avg;
};
```

```
struct student s[3];
```

Defines an array called s, which contains three elements. Each element is defined to be of type struct student.

For the student details, array of structures can be initialized as follows,

```
struct student s[3]={{“ABC”,1,56.7},{“xyz”,2,65.8},{“pqr”,3,82.4}};
```

Ex: An array of structures for structure employee can be declared as

```
struct employee emp[5];
```

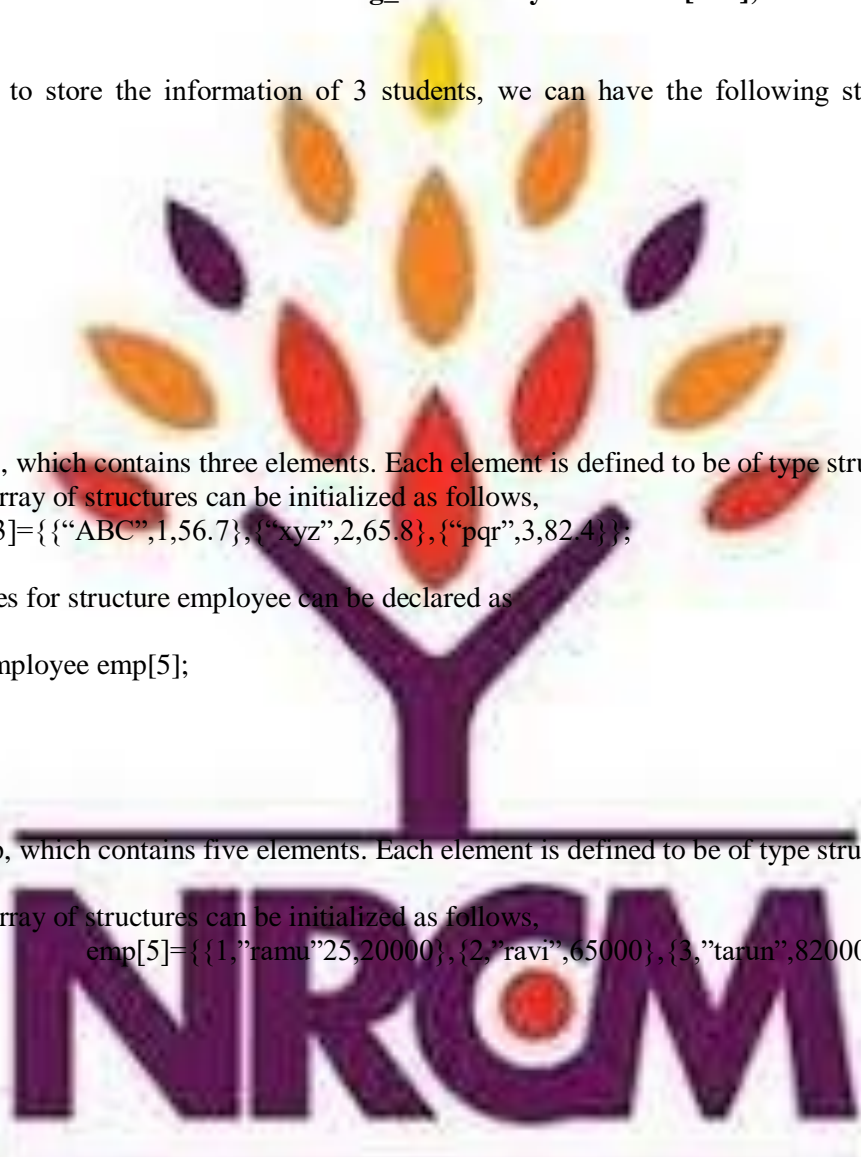
Defines array called emp, which contains five elements. Each element is defined to be of type struct student.

For the student details, array of structures can be initialized as follows,

```
struct employee emp[5]={{1,”ramu”,25,20000},{2,”ravi”,65000},{3,”tarun”,82000},{4,”rupa”,5000},
{5,”deepa”,27000}};
```

Example :

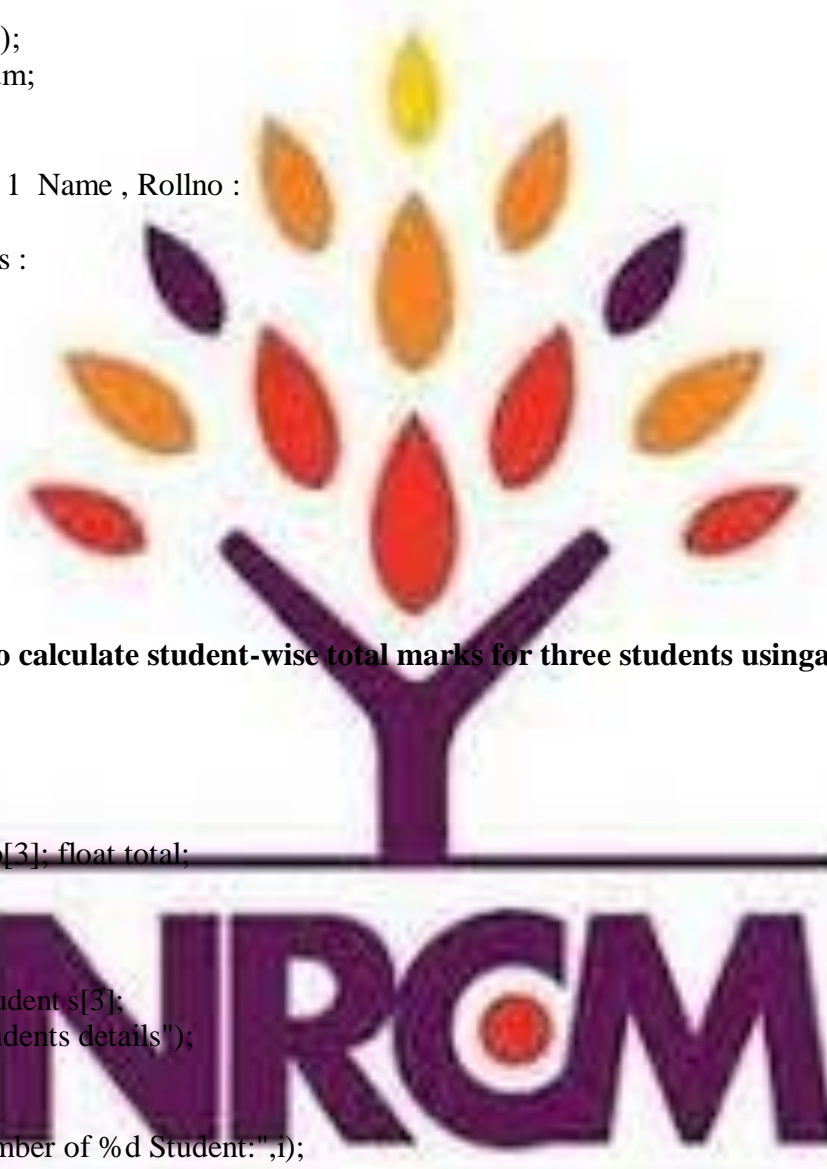
```
struct student
{
char name [20];
int rollno;
int marks [5];
};
void main( )
{
struct student s1;
int i,sum=0;
```



```
printf("Enter details of Student – 1 Name , Rollno:");
scanf("%s %d",s1.name,&s1.rollno);
printf("\n Enter marks of 5 Subjects :");
for(i=0;i<5;i++)
{
scanf("%d",&s1.marks[i]);
sum+=s1.marks[i];
}
Printf("\n Details of students are : \n Name : %s Roll no : %d \n Marks : ", s1.name,s1.rollno);
for(i=0;i<5;i++)
printf("%d ",s1.marks[i]);
printf("\n Total : %d ",sum;
}
```

Output :

```
Enter details of Student – 1 Name , Rollno :
Ravi 201
Enter marks of 5 Subjects :
50 60 70 40 90
Details of students are :
Name : Ravi
Roll no : 201
Marks : 50 60 70 40 90
Total : 310
```



Example :

Simulate a C program to calculate student-wise total marks for three students using array of structures.

```
#include<stdio.h>
struct student
{
char rollno[10];
char name[20]; float sub[3]; float total;
};
void main( )
{
int i, j, total = 0; struct student s[3];
printf("\t\t\t Enter 3 students details");
for(i=0; i<3; i++)
{
printf("\n Enter Roll number of %d Student:",i);
gets(s[i].rollno);
printf(" Enter the name:");
gets(s[i].name);
printf(" Enter 3 subjects marks of each student:");
total=0;
for(j=0; j<3; j++)
{
scanf("%d",&s[i].sub[j]);
total = total + s[i].sub[j];
}
```

```

}
}
printf("\n*****");
printf("\n\t\t Student details:");
printf("\n*****");
for(i=0; i<n; i++)
{
printf ("\n Student %d:",i+1);
printf ("\n Roll number:%s\n name:%s",s[i].rollno,s[i].name);
printf ("\nTotal marks =%f", s[i].total);
}
}

```

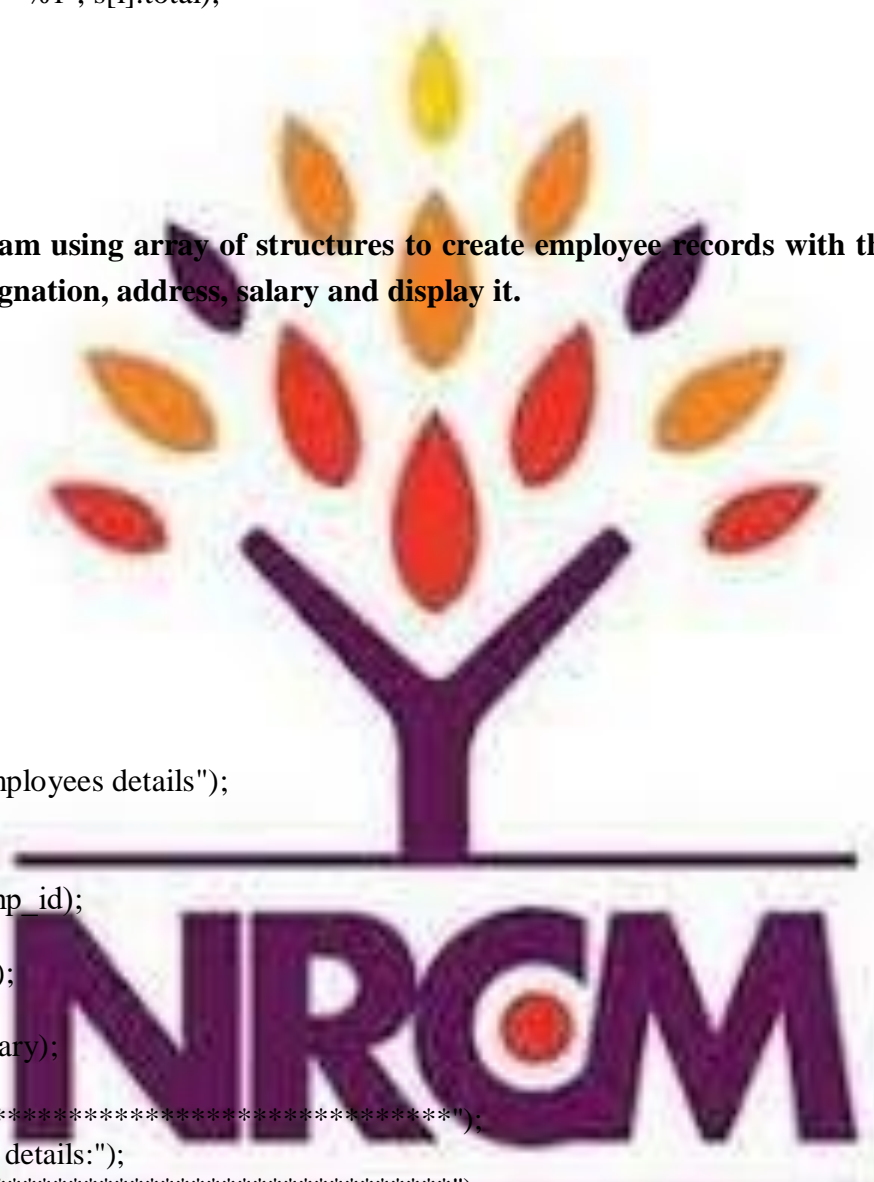
Example:

Develop a C program using array of structures to create employee records with the following fields: emp-id, name, designation, address, salary and display it.

```

#include<stdio.h>
struct employee
{
int emp_id;
char name[20];
char designation[10];
char address[20];
float salary;
}emp[3];
void main( )
{
int i;
printf("\t\t\t Enter 3 employees details");
for(i=0; i<3; i++)
{
scanf("%d",&emp[i].emp_id);
gets(emp[i].name);
gets(emp[i].designation);
gets(emp[i].address);
scanf("%f",&emp[i].salary);
}
printf("\n*****");
printf("\n\t\t Employee details:");
printf("\n*****");
for(i=0; i<3; i++)
{
printf("%d",emp[i].emp_id);
puts(emp[i].name);
puts(emp[i].designation);
puts(emp[i].address);
printf("%f",emp[i].salary);
}
}

```



```
}
```

Arrays within structure:

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

```
struct student
{
char name [5];
int roll_number;
int marks [3];
float avg;
};
```

Then the initialization of the array marks done as follows:

```
struct student s1= {"ravi", 34, {60,70,80}};
```

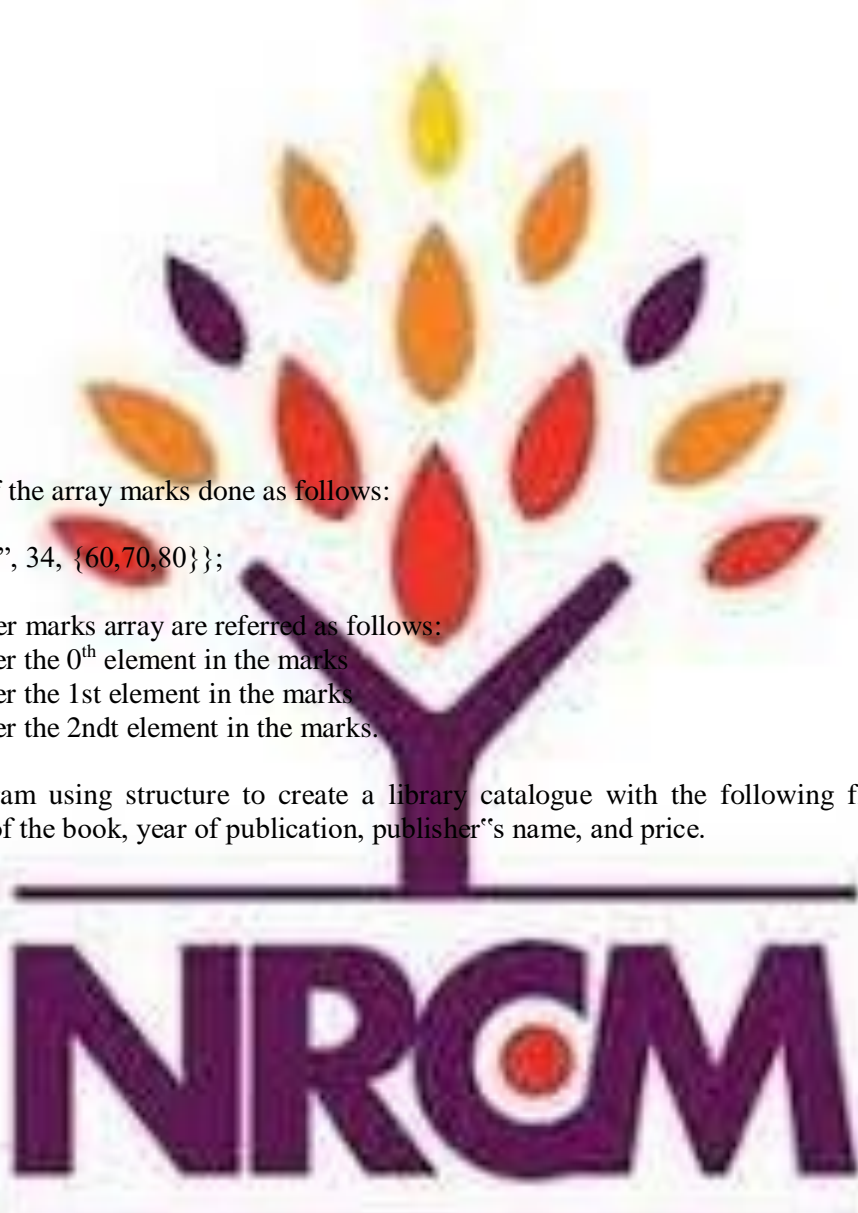
The values of the member marks array are referred as follows:

```
s1.marks [0] --> will refer the 0th element in the marks
s1.marks [1] --> will refer the 1st element in the marks.
s1.marks [2] --> will refer the 2nd element in the marks.
```

Ex: Construct a C program using structure to create a library catalogue with the following fields: Access number, author's name, Title of the book, year of publication, publisher's name, and price.

```
struct library
{
int acc_no;
char author[20];
char title[10];
int year_pub;
char name_pub[20];
float price;
};

void main( )
{
struct library lib;
printf("Input values are:\n");
scanf("%d %s %s %d %s%f", &lib.acc_no, lib.author, lib.title, &lib.year_pub, lib.name_pub, &lib.price);
printf("Output values are:\n");
printf("Access number = %d\n Author name = %s\n Book Title = %s\n Year of publication = %d \n Name of publication = %s\n Price = %f", lib.acc_no, lib.author, lib.title, lib.year_pub, lib.name_pub, lib.price);
}
```



Distinguish between Arrays within Structures and Array of Structures with examples :

Arrays within structure:

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

```
struct student
{
char name [5];
int roll_number;
int marks [3];
float avg;
};
```

Then the initialization of the array marks done as follows:

```
struct student s1= {"ravi", 34, {60,70,80}};
```

The values of the member marks array are referred as follows:

s1.marks [0] --> will refer the 0th element in the marks

s1.marks [1] --> will refer the 1st element in the marks

s1.marks [2] --> will refer the 2nd element in the marks

Array of structures:

An array is a collection of elements of same data type that are stored in contiguous memory locations. A structure is a collection of members of different data types stored in contiguous memory locations. An array of structures is an array in which each element is a structure. This concept is very helpful in representing multiple records of a file, where each record is a collection of dissimilar data items.

Ex:

An array of structures for structure *employee* can be declared as

```
struct employee emp[5];
```

Let's take an example, to store the information of 5 employees, we can have the following structure definition and declaration,

```
struct employee
{
int empid; char name[10]; float salary;
};
struct employee emp[5];
```

Defines array called emp, which contains five elements. Each element is defined to be of type struct student.

For the student details, array of structures can be initialized as follows, struct employee emp[5] = {{1,"ramu",25,20000},

```
{2,"ravi",65000},
{3,"tarun",82000},
{4,"rupa",5000}, {5,"deepa",27000}};
```

Nested Structure (Structure within structure) :

A structure which includes another structure is called nested structure or structure within structure. i.e a structure can be used as a member of another structure. There are two methods for declaration of nested structures.

(i) The syntax for the nesting of the structure is as follows:

```
struct tag_name1
{
type1 member1;
.....
.....
};

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

The syntax for accessing members of a nested structure as follows, outer_structure_variable . inner_structure_variable.member_name

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

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



Example : struct stud_Res

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

In above example, the structure stud_Res consists of stud_subj which itself is a structure with two members. Structure stud_Res is called as 'outer structure' while stud_subj is called as 'inner structure.'

The members which are inside the inner structure can be accessed as follow: result.subj.subjnm

result.subj.marks

Program to demonstrate nested structures.

```
#include <stdio.h>
#include <conio.h>
struct stud_Res
{
    int rno;
    char std[10];
    struct stud_Marks
    {
        char subj_nm[30];
        int subj_mark;
    }marks;
}result;
void main()
{
    printf("\n\t Enter Roll Number :");
    scanf("%d",&result.rno);
    printf("\n\t Enter Standard : ");
    scanf("%s",result.std);
    printf("\n\t Enter Subject Code:");
    scanf("%s",result.marks.subj_nm);
    printf("\n\t Enter Marks: ");
    scanf("%d",&result.marks.subj_mark);
    printf("\n\t Roll Number : %d",result.rno);
    printf("\n\t Standard : %s",result.std);
    printf("\n\t Subject Code : %s",result.marks.subj_nm);
    printf("\n\t Marks : %d",result.marks.subj_mark);
}
```

Output:

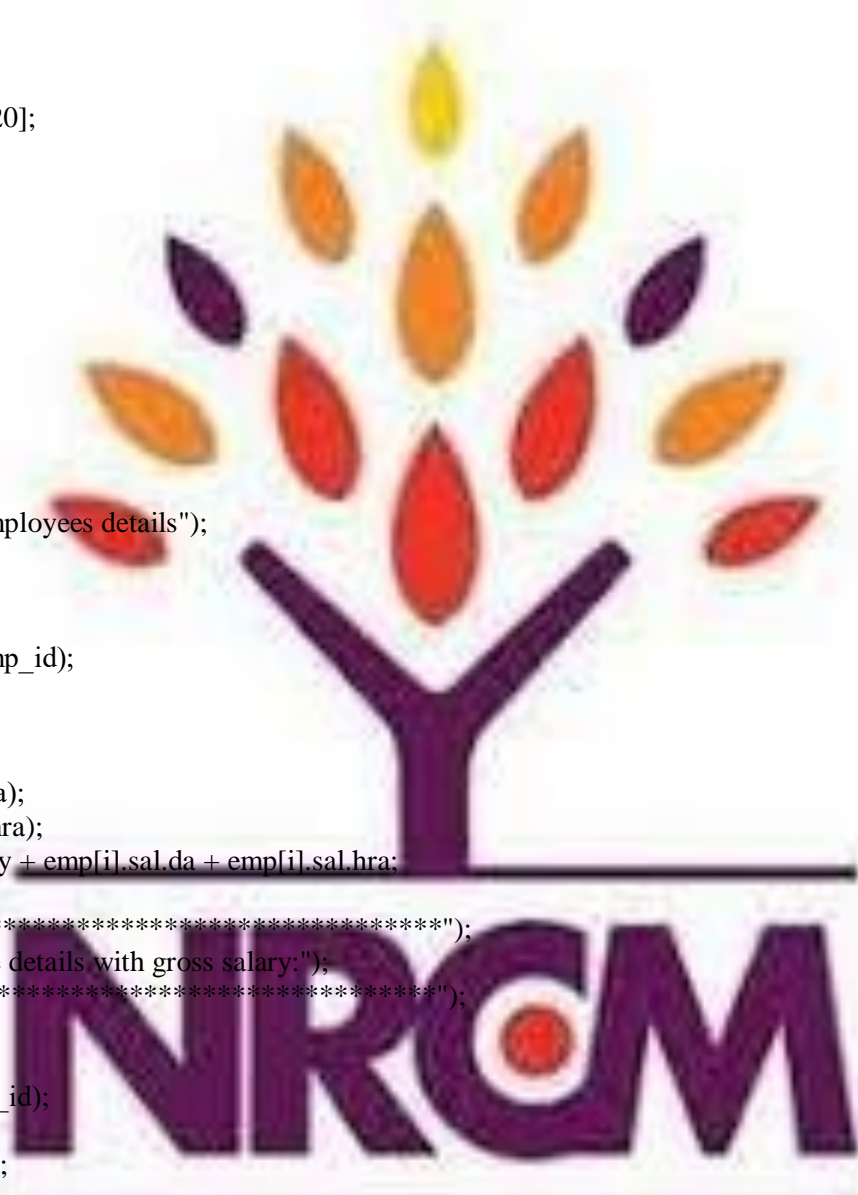
```
Enter roll number : 1
Enter standard :Btech
Enter subject code : GR11002
```



Enter marks : 63
Roll number :1
Standard :Btech
Subject code : GR11002
Marks : 63

Ex: Design a C program using nested structures to read 3 employees details with the following fields; emp-id, name, designation, address, da ,hra and calculate gross salary of each employee.

```
#include<stdio.h>
struct employee
{
int emp_id; char name[20];
char designation[10];
char address[20];
struct salary
{
float da;
float hra;
}sal;
}emp[3];
void main( )
{
int i;
printf("\t\t\t\t\t Enter 3 employees details");
grosssalary = 0;
for(i=0; i<3; i++)
{
scanf("%d",&emp[i].emp_id);
gets(emp[i].name);
gets(emp[i].designation);
gets(emp[i].address);
scanf("%f",&emp[i].sal.da);
scanf("%f",&emp[i].sal.hra);
grosssalary = grosssalary + emp[i].sal.da + emp[i].sal.hra;
}
printf("\n*****");
printf("\n\t\t\t\t Employee details with gross salary:");
printf("\n*****");
for(i=0; i<3; i++)
{
printf("%d",emp[i].emp_id);
puts(emp[i].name);
puts(emp[i].designation);
puts(emp[i].address);
printf("%f",emp[i].sal.da);
printf("%f",emp[i].sal.hra);
printf("%f",grosssalary);
}
}
```



Structures and Functions:

Structures as Function arguments – Passing structures to functions:

Pass a structure member as an argument to a function:

Structures are more useful if we are able to pass them to functions and return them.

By passing individual members of structure

This method is to pass each member of the structure as an actual argument of the function call. The actual arguments are treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.

Program:

```
#include<stdio.h>
void main ( )
{
float arriers (char *s, int n, float m);
typedef struct emp
{
char name[15]; int emp_no;
float salary;
}record;
record e1 = {"smith",2,20000.25};
e1.salary = arriers(e1.name,e1.emp_no,e1.salary);
}

float arriers(char *s, int n, float m)
{
m = m + 2000;
printf("\n%s %d %f ",s, n, m);
return m;
}
}
```

Output

```
smith 2    22000.250000
```

Pass an entire structure as an argument of a function :

Structures are more useful if we are able to pass them to functions and return them.

Passing Whole Structure:

This method involves passing a copy of the entire structure to the called function. Any changes to structure members within the function are not reflected in the original structure. It is therefore, necessary for the function to return the entire structure back to the calling function.

The general format of sending a copy of a structure to the called function is:

```
return_type function_name (structure_variable_name);
```

The called function takes the following form:

```
data_type function_name(struct_typetag_name)
{
.....
.....
return(expression);
}
```

The called function must be declared for its type, appropriate to the data type it is expected to return.

The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same struct type. The return statement is necessary only when the function is returning some data back to the calling function. The expression may be any simple variable or structure variable or an expression using simple variables. When a function returns a structure, it must be assigned to a structure of identical type in the calling function. The called functions must be declared in the calling function appropriately.

Program:

```
#include <stdio.h>
#include <string.h>
struct student
{
    int id;
    char name[20];
    float percentage;
};

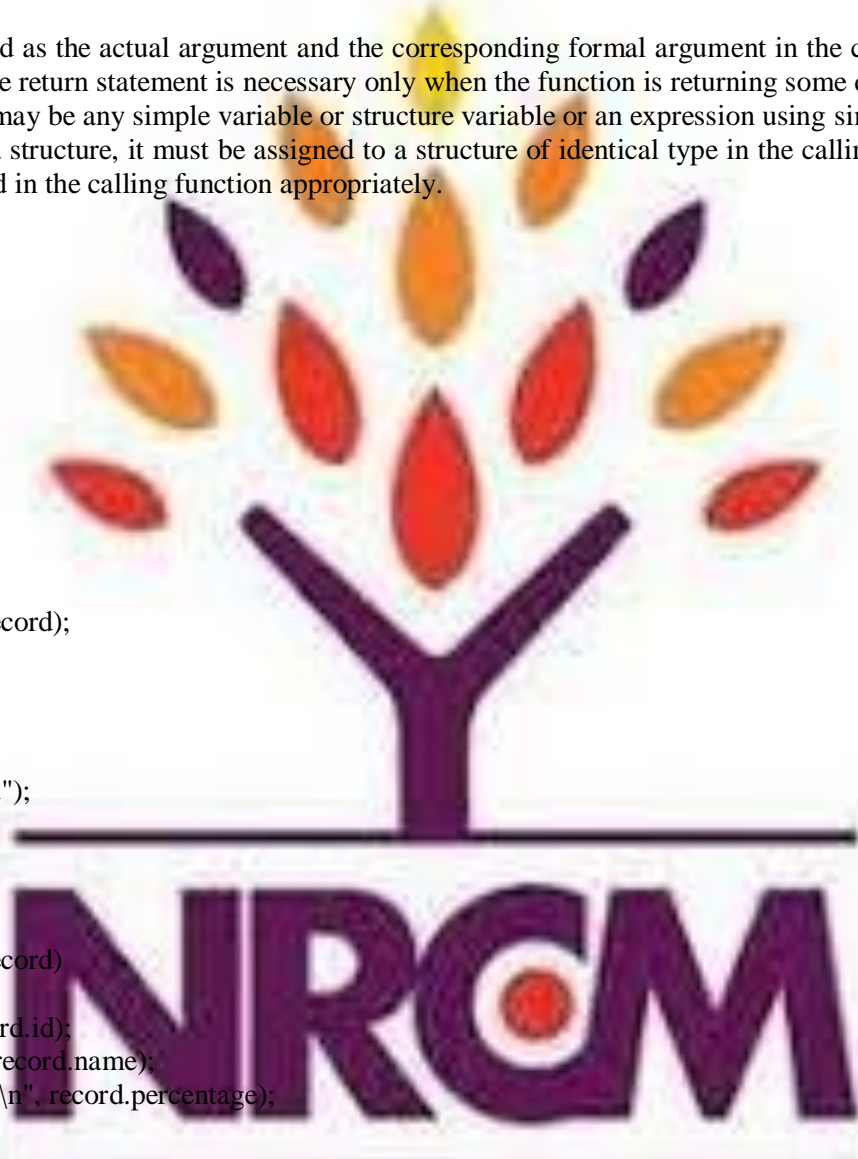
void func(struct student record);
int main()
{
    struct student record;
    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;
    return 0;
}

void func(struct student record)
{
    printf(" Id is: %d \n", record.id);
    printf(" Name is: %s \n", record.name);
    printf(" Percentage is: %f \n", record.percentage);
}
```

Output:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

Function returning a structure:

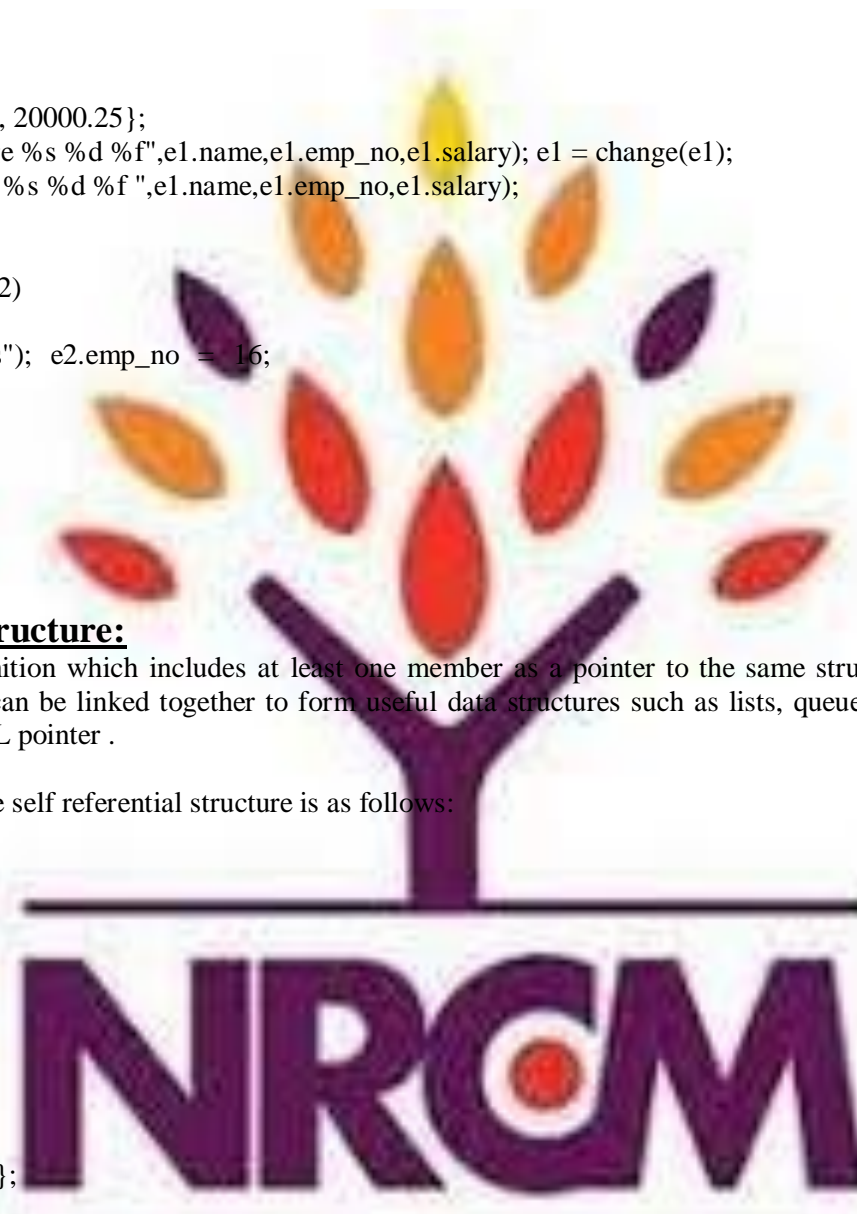


```

#include<stdio.h>
#include<string.h>

typedef struct
{
char name [15];
int emp_no;
float salary;
} record;
void main ( )
{
record change (record);
record e1 = {"Smith", 2, 20000.25};
printf ("\nBefore Change %s %d %f",e1.name,e1.emp_no,e1.salary); e1 = change(e1);
printf ("\nAfter Change %s %d %f ",e1.name,e1.emp_no,e1.salary);
}

record change (record e2)
{
strcpy (e2.name,"Jones"); e2.emp_no = 16;
e2.salary = 9999;
return e2;
}
Output:
Smith 2 20000.25
Jones 16 9999.99
    
```



Self referential structure:

A structure definition which includes at least one member as a pointer to the same structure is known as self-referential structure. It can be linked together to form useful data structures such as lists, queues, stacks and trees. It is terminated with a NULL pointer .

The syntax for using the self referential structure is as follows:

```

struct tag_name
{
Type1 member1;
Type2 member2;
.....
struct tag_name *next; };
    
```

Ex:-

```

struct node
{
int data;
struct node *next; } n1, n2;
    
```

Unions:

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

The general format or syntax of a union definition is as follows,

Syntax:

```
union union_name
{
    <data-type> element 1;
    <data-type> element 2;
    .....
}union_variable;
```

Example:

```
union techno
{
int comp_id; char nm; float sal;
}tch;
```

A union variable can be declared in the same way as structure variable. union tag_name var1, var2...;
A union definition and variable declaration can be done by using any one of the following

<pre>union u { char c; int i; float f; }; union u a;</pre>	<pre>union u { char c; int i; float f; } a;</pre>	<pre>typedef union { char c; int i; float f; } U; U a;</pre>
---	---	---

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

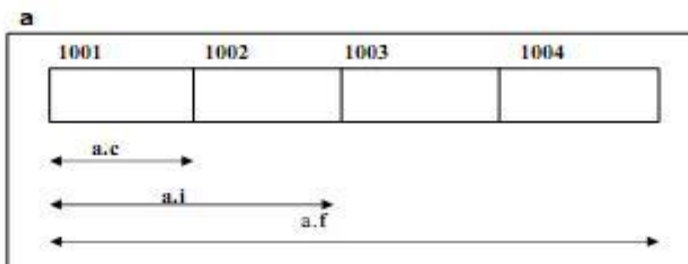


Figure 5.8: Memory Organization Union

In the above declaration, the member **f** requires 4 bytes which is the largest among all the members. Figure 5.8 shows how all the three variables share the same address. The size of the union here is 4 bytes.

A union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supersedes the previous members' value.

Difference between structure and union:-

	Structure	Union
(i) Keyword	Struct	Union
(ii) Definition	A structure is a collection of logically related elements, possibly of different types, having a single name.	A union is a collection of logically related elements, possibly of different types, having a single name, shares single memory location.
(iii) Declaration	<pre>struct tag_name { type1 member1; type1 member2; }; struct tag_name var;</pre>	<pre>union tag_name { type1 member1; type1 member2; }; union tag_name var;</pre>
(iv) Initialization	Same.	Same.
(v) Accessing	Accessed by specifying structure_variable_name.member_Name	Accessed by specifying union_variable_name.member_name
(vi) Memory Allocation	Each member of the structure occupies unique location, stored in contiguous locations.	Memory is allocated by considering the size of the largest member. All the members share the common location
(vii) Size	Size of the structure depends on the type of members, adding size of all the members. sizeof(st_var);	Size is given by the size of the largest member sizeof(un_variable)
(viii) Using Pointers	Structure members can be accessed by using dereferencing operator dot and selection	same as structure.

	operator(->)	
	We can have arrays as a member of structures. All members can be accessed at a time.	We can have array as a member of union. Only one member can be accessed at a time.
	Nesting of structures is possible.	same.
	It is possible structure may contain union as a member.	It is possible union may contain structure as a member



Typedef :

It is a User defined data type. C supports a feature known as —"type definition" that allows users define an identifier that would represents an existing type.

Syntax: typedef data-type identifier;

Where data-type indicates the existing datatype, identifier indicates the new name that is given to the data type.

Ex:- typedef int marks;

marks m1, m2, m3;

Note: typedef cannot create a new data type ,it can rename the existing datatype. The main advantage of typedef is that we can create meaningful datatype names for increasing the readability of the program.

So, how do you actually declare a typedef? All you must do is provide the old type name followed by the type that should represent it throughout the code.

Following is the general syntax for using typedef,

typedef <existing_name><alias_name>

Lets take an example and see how typedef actually works.

```
typedef unsigned long ulong;
```

The above statement define a term ulong for an unsigned long datatype. Now this ulong identifier can be used to define unsigned long type variables.

```
ulong i, j;
```

Application of typedef : Note that in C, typedefs can also be used to remove some of the burden associated with declaring structs

typedef can be used to give a name to user defined data type as well. Lets see its use with structures.

Example:

```
typedef struct
```

```
{  
    type member1;  
    type member2;  
    type member3;  
} type_name;
```

Here type_name represents the sturcture definition associated with it. Now this type_name can be used to declare a variable of this sturcture type.

```
type_name t1, t2;
```

Advantages :

- Typedefs can make your code more clear
- Typedefs can make your code easier to modify.

typedef vs #define :

#define is a C-directive which is also used to define the aliases for various data types similar to typedef but with the following differences –

- typedef is limited to giving symbolic names to types only where as #define can be used to define alias for values as well, q., you can define 1 as ONE etc.
- typedef interpretation is performed by the compiler whereas #define statements are processed by the pre-processor.

Example of typedef :

```
#include <stdio.h>  
#include <string.h>  
typedef struct Books {  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
} Book;
```

```
int main( ) {
```

```
    Book book;
```

The logo for NIRCM (National Institute of Research in Computer Graphics and Multimedia) features the acronym 'NIRCM' in a bold, purple, sans-serif font. A stylized tree with a purple trunk and branches, and leaves in shades of orange, red, and yellow, is positioned behind the letters 'I' and 'R'.

```
strcpy( book.title, "C Programming");
strcpy( book.author, "Nuha Ali");
strcpy( book.subject, "C Programming Tutorial");
book.book_id = 6495407;

printf( "Book title : %s\n", book.title);
printf( "Book author : %s\n", book.author);
printf( "Book subject : %s\n", book.subject);
printf( "Book book_id : %d\n", book.book_id);

return 0;
}
```

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

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
```

POINTERS:

A Pointer is a constant or variable that contains an address that can be used to access data.

Examples:

```
int *ptr;
```

In c programming every variable keeps two types of value Value of variable. Address of variable where it has stored in the memory. Meaning of following simple pointer declaration and definition:

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

Explanation:

About variable —a

Name of variable:a

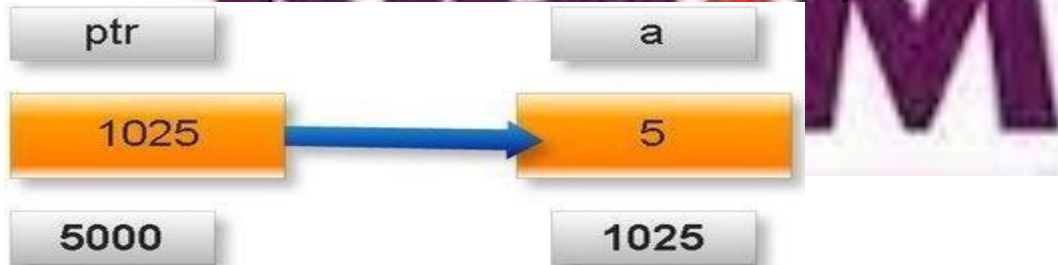
Value of variable which it keeps:5

Address where it has stored in memory: 1025(assume)

About variable —ptr :

Name of variable:ptr

Value of variable which it keeps: 1025 6.Address where it has stored in memory:5000



As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined–

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



```
int main ()
{
int var1;
char var2[10];
printf("Address of var1 variable: %x\n", &var1 );
printf("Address of var2 variable: %x\n", &var2 );
return 0;
}
```

When the above code is compiled and executed, it produces the following result –
Address of var1 variable: bff5a400 Address of var2 variable: bff5a3f6

What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is **type *var-name;**

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –

```
int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, along with a hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Benefits of using pointers are:-

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

How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently.

- We define a pointer variable,
- assign the address of a variable to a pointer and
- finally access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations–

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

```
int main ()
{
int var = 20; /* actual variable declaration */
int *ip; /* pointer variable declaration */
ip = &var; /* store address of var in pointer variable */
printf("Address of var variable: %x\n", &var ); /* address stored in pointer variable */
printf("Address stored in ip variable: %x\n", ip ); /* access the value using the pointer */
```

```
printf("Value of *ip variable: %d\n", *ip );
return 0;
}
```

When the above code is compiled and executed, it produces the following result – Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c Value of *ip variable: 20

Declaration and Initializing a Pointer:

In C, every variable must be declared for its type. Since pointer variable contain addresses that belong to a separate data type, they must be declared as pointers before we use them.

Declaration of a pointer variable:

The declaration of a pointer variable takes the following form: **data type*pt_name;**

This tells the compiler three things about the variable `pt_name`:

The `*` tells that the variable `pt_name` is a pointer variable.

`pt_name` needs a memory location

`pt_name` points to a variable of type `data_type`

Ex: `int*p;` Declares the variable `p` as a pointer variable that points to an integer data type.

Initialization of pointer variables:

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

Ex:

```
int quantity ;
int
```

claration

```
p=&quantity; //initialization
```

We can also combine the initialization with the declaration:

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

Always ensure that a pointer variable points to the corresponding type of data.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step.

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

Example program:

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

Output: The output of the above program would be:

```
Address of a=1444
```

```
Value of a=5
```

The expression `&a` returns the address of the variable `a`, which in this case happens to be 1444

.Hence it is printed out using `%u`, which is a format specified for printing an unsigned integer.

Accessing a variable through its pointer:

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

Ex:

```

Int main()
{
int a = 5 ;
printf ( "\nAddress of a = %u", &a );
printf ( "\nValue of a = %d", a );
printf ( "\nValue of a = %d", *( &a));
return 0;
}

```

Output: The output of the above program would be:
Address of a = 1444 Value of a = 5 Value of a = 5

Example : To Demonstrate Working of Pointers

```

/* Source code to demonstrate, handling of pointers in C program */ #include <stdio.h>
int main()
{
int* pc; int c; c=22;
printf("Address of c:%u\n",&c); printf("Value of c:%d\n\n",c); pc=&c;
printf("Address of pointer pc:%u\n",pc); printf("Content of pointer pc:%d\n\n",*pc); c=11;
printf("Address of pointer pc:%u\n",pc); printf("Content of pointer pc:%d\n\n",*pc);
*pc=2;
printf("Address of c:%u\n",&c); printf("Value of c:%d\n\n",c); return 0;
}

```

Pointer Expressions and Pointer Arithmetic:

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer—
ptr++

After the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If **ptr** points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array –

```

#include <stdio.h>
const int MAX = 3;
int main ()
{
int var[] = { 10, 100, 200};
int i, *ptr;
ptr = var; /* let us have array address in pointer */
for ( i = 0; i < MAX; i++)
{
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
ptr++; /* move to the next location */
}
return 0;
}

```



```
}

```

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

Address of var[0] = bf882b30 Value of var[0] = 10

Address of var[1] = bf882b34 Value of var[1] = 100 Address of var[2] = bf882b38 Value of var[2] = 200

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below –

```
#include <stdio.h> const int MAX = 3; int main ()
{

```

```
int var[] = { 10, 100, 200};
int i, *ptr;
/* let us have array address in pointer */
ptr = &var[MAX-1]; for ( i = MAX; i > 0;i--)
{
printf("Address of var[%d] = %x\n", i-1, ptr ); printf("Value of var[%d] = %d\n", i-1, *ptr );
/* move to the previous location */ ptr--;
}
return 0;
}
```

```
int i, *ptr;
```

```
/* let us have array address in pointer */
```

```
ptr = &var[MAX-1]; for ( i = MAX; i > 0;i--)
```

```
{
printf("Address of var[%d] = %x\n", i-1, ptr ); printf("Value of var[%d] = %d\n", i-1, *ptr );
```

```
/* move to the previous location */ ptr--;
```

```
}
```

```
return 0;
```

```
}
```

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

Address of var[2] = bfedbcd8 Value of var[2] = 200 Address of var[1] = bfedbcd4 Value of var[1] = 100 Address of var[0] = bfedbcd0 Value of var[0] = 10

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example – one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1] –

```
#include <stdio.h> const int MAX = 3; int main ()
{

```

```
int var[] = { 10, 100, 200};
int i, *ptr;
```

```
/* let us have address of the first element in pointer */ ptr = var;
i = 0;
```

```
while ( ptr <= &var[MAX - 1] )
{

```

```
printf("Address of var[%d] = %x\n", i, ptr ); printf("Value of var[%d] = %d\n", i, *ptr );
```

```
/* point to the previous location */ ptr++;
```

```
i++;
```

```
}
```

```
return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result – Address of var[0] = bfdbcb20

Value of var[0] = 10

Address of var[1] = bfdbcb24 Value of var[1] = 100 Address of var[2] = bfdbcb28 Value of var[2] = 200

Pointer Operations:

C language allows us to add integers to pointers and to subtract integers from pointers

Ex: If p1, p2 are two pointer variables then operations such as p1+4, p2 - 2, p1 - p2 can be performed.

Pointers can also be compared using relational operators. Ex: $p1 > p2$, $p1 == p2$, $p1 != p2$ are valid operations.

We should not use pointer constants in division or multiplication. Also, two pointers cannot be added. $p1/p2$, $p1 * p2$, $p1/3$, $p1 + p2$ are invalid operations.

Pointer increments and scale factor:-

Let us assume the address of $p1$ is 1002. After using $p1 = p1 + 1$, the value becomes 1004 but not 1003.

Thus when we increment a pointer, its value is increased by length of data type that points to. This length is called scale factor.

Pointer Expressions:

Like other variables pointer variables can be used in expressions.

If $p1$ and $p2$ are properly declared and initialized pointers, then the following statements are valid:

```
Y = *p1 **p2;
Sum = sum + *p1;
Z = 5 * - *p2 / *p1;
*p2 = *p2 + 10;
*p1 = *p1 + *p2;
*p1 = *p2 - *p1;
```

NOTE: in the third statement there is a blank space between `/'` and `*` because the symbol `/*` is considered as beginning of the comment and therefore the statement fails. if $p1$ and $p2$ are properly declared and initialized pointers then, `'C'` allows adding integers to a pointer variable.

EX:

```
Int a=5, b=10;
```

```
Int *p1, *p2;
```

```
p1 = &a;
```

```
p2 = &b;
```

```
Now, P1 = p1 + 1 = 1000 + 2 = 1002;
```

```
P1 = p1 + 2 = 1000 + (2 * 2) = 1004;
```

```
P1 = p1 + 4 = 1000 + (2 * 4) = 1008;
```

```
P2 = p2 + 2 = 3000 + (2 * 2) = 3004;
```

```
P2 = p2 + 6 = 3000 + (2 * 6) = 3012;
```

Here addition means bytes that pointer data type hold subtracted number of times that is subtracted to the pointer variable.

C program using pointers to determine the length of a character String.

```
/*program to find the length of a char string */
#include <stdio.h>
#include <conio.h>
#include <string.h>
Void main ()
{
Char str [20], *p;
Int l=0;
printf ("enter a string \n");
scanf ("% s", str);
p=str; while(*p!='\0')
{
l++;
p++;
}
printf ("the length of the given string is %d", l); getch ();
}
```

void pointer:

Pointers can also be declared as void type. Void pointers cannot be dereferenced without explicit type conversion. This is because, being void the compiler cannot determine the size of the object that the pointer points to. Though void pointer declaration is possible, void variables declaration is not allowed. Thus, the declaration void P displays an error message. "size of 'p' is unknown or zero" after compilation. C program to declare a void pointer. Assign address of int, float and char variable to the void pointer using type casting method. Display the contents of various variables.

```
#include<stdio.h>
#include<conio.h>
Void main()
{
int p;
float d;
char c;
void *pt;
clrscr();
pt=&p;
*(int*)pt=10;
printf("\n p=%d",p);
pt=&d;
*(float *)pt=3.4;
printf("\n d=%f",d);
pt=&c;
*(char *)pt='s';
printf("\n c=%c",c); getch();
}
```

O/P:

p=10 d=3.4
c=s

In the above example, the statement `*(int*)pt=10` assigns the integer value 10 to pointer `pt`. i.e., to variable 'p'. The declaration `*(int *)` tells the compiler that the value assigned is of integer type. Thus, assignments of float and char type are carried out. The statements `*(int*)pt=10`, `*(float *)pt=3.4`, `*(char *)pt='s'` helps the compiler to exactly determine the size of data type.

Null pointer:

A null pointer is a pointer which points nothing.

Some uses of the null pointer are:

- To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.
- To pass a null pointer to a function argument when we don't want to pass any valid memory address.
- To check for null pointer before accessing any pointer variable. So that, we can perform error handling in pointer related code e.g. dereference pointer variable only if it's not NULL.

Example program:

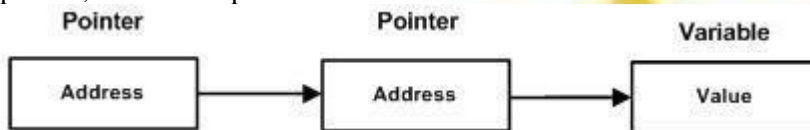
```
#include <stdio.h>
int main() {
```

```
int *p= NULL;//initialize the pointer as null.
printf("The value of pointer is %u",p);
return 0;
}
```

o/p: The value of pointer is 0

Pointers to pointers:

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



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

```
int **var;
```

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

```
#include <stdio.h> int main ()
{
int var; int *ptr;
int **pptr; var = 3000;
/* take the address of var */
ptr = &var;
/* take the address of ptr using address of operator & */
pptr = &ptr;
/* take the value using pptr */
printf("Value of var = %d\n", var );
printf("Value available at *ptr = %d\n", *ptr );
printf("Value available at **pptr = %d\n", **pptr);
return 0;
}
```

Output:

Value of var = 3000

Value available at *ptr = 3000 Value available at **pptr = 3000

Example 2: program to print value of a variable through pointer and pointer to pointer.

```
#include<stdio.h>
#include<conio.h> Void main ()
{
int x=10,*p, **q;
P=&x;
Q=&p;
clrscr();
printf("value of x=%d address of x=%u", x, &x);
printf("through *p value of x=%d address of x=%u",*p, p);
printf("through **q value of x=%d address of x=%u", **q,*q);
getch();
}
```

Output:

Value of x=10 address of x=2000
 Through *p value of x=10 address of x=2000
 Through **q value of x=10 address of x=2000

Pointers to Arrays- Pointers and Arrays :

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

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

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

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

the name x is defined as a constant pointer pointing to the first element, x[0], and therefore the value x is 1000, the location where x[0] is stored. That is

x=&x[0]=1000;

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

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

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

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

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

Eg:- int x[5]; x=1000;

Address of x[3]= base address of x+ (3*scale factor of int)

$$1000+(3*2)$$

$$1000+6 = 1006$$

Ex:- float avg[20]; avg=2000;

Address of avg [6]=2000+(6*scale factor of float) =2000+6*4

=2000+24 =2024.

Ex:- char str [20]; str =2050;

Address of str[10]=2050+(10*1)

=2050+10

=2060.

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

The pointer accessing method is faster than the array indexing.

Accessing elements of an array:-

```

/*Program to access elements of a one dimensional array*/
#include<stdio.h>
void main()
{
int arr[5]={ 10,20,30,40,50}; int p=0;
printf("\n value@ arr[i] is arr[p] | *(arr+p)| *(p+arr) | p[arr] | located @address \n");
for(p=0;p<5;p++)
{
printf("\n value of arr[%d] is:",p);
printf(" %d | ",arr[p]);
printf(" %d | ",*(arr+p));
printf(" %d | ",*(p+arr));
printf(" %d | ",p[arr]);
printf("address of arr[%d]=%u\n",p,(arr+p));
}
}
    
```

output:

```

value@ arr[i] is arr[p] | *(arr+p)| *(p+arr) | p[arr] | located @address

value of arr[0] is: 10 | 10 | 10 | 10 | address of arr[0]=3713876608
value of arr[1] is: 20 | 20 | 20 | 20 | address of arr[1]=3713876612
value of arr[2] is: 30 | 30 | 30 | 30 | address of arr[2]=3713876616
value of arr[3] is: 40 | 40 | 40 | 40 | address of arr[3]=3713876620
value of arr[4] is: 50 | 50 | 50 | 50 | address of arr[4]=3713876624
    
```

Pointers to two dimensional Arrays:

Pointers to two-dimensional arrays:-

Pointer can also be used to manipulate two-dimensional arrays. Just like how x[i] is represented by

*(x+i) or *(p+i), similar, any element in a 2-d array can be represented by the pointer as follows.

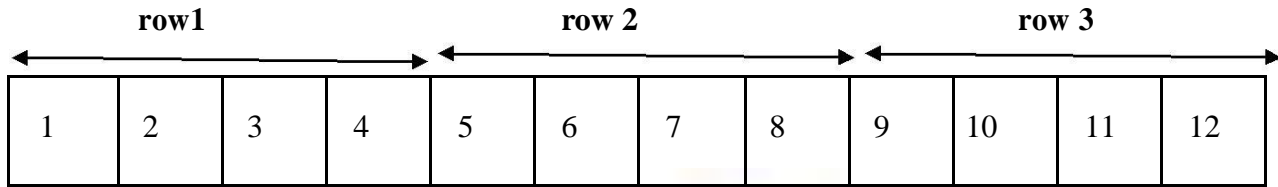
((a+i)+j) or *(*(p+i)+j)

The base address of the array a is &a[0][0] and starting from this address the compiler allocates contiguous space for all the element row – wise .

i.e the first element of the second row is placed immediately after the last element of the first row and so on
 Ex:- suppose we declare an array as follows.

```
int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

The elements of a will be stored as shown below.



Base address = &a[0][0]

If we declare p as int pointer with the initial address &a[0][0] then a[i][j] is equivalent to *(p+4 x i+j). You may notice if we increment „i“ by 1, the p is incremented by 4, the size of each row. Then the element a[2][3] is given by *(p+2 x 4+3) = *(p+11).

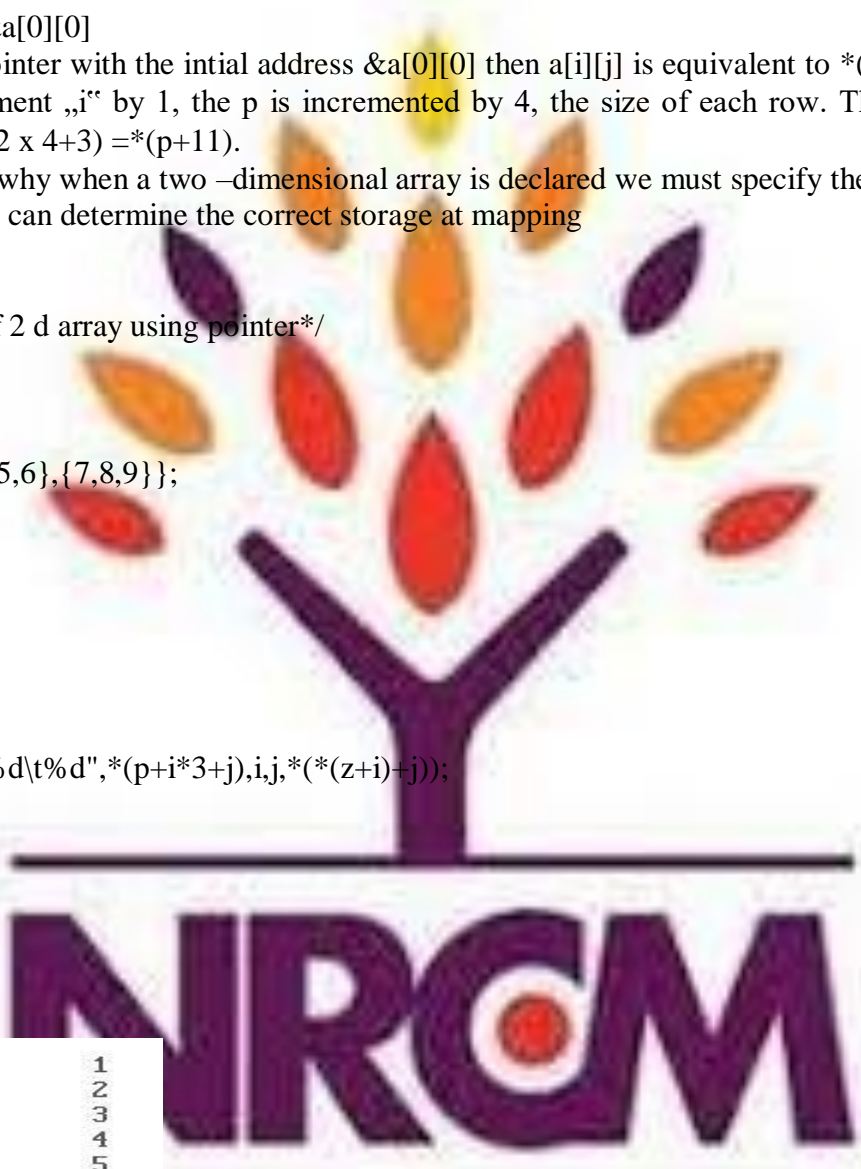
Note: This is the reason why when a two –dimensional array is declared we must specify the size of the each row so that the compiler can determine the correct storage at mapping

Program:

```
/* accessing elements of 2 d array using pointer*/
#include<stdio.h>
void main()
{
int z[3][3]={{1,2,3},{4,5,6},{7,8,9}};
int i, j;
int *p;
p=&z[0][0];
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
printf("\n %d\ti=%d j=%d\t%d",*(p+i*3+j),i,j,*(z+i+j));
}
}
for(i=0;i<9;i++)
printf("\n\t%d",*(p+i));
}
```

output:

```
1      i=0  j=0      1
2      i=0  j=1      2
3      i=0  j=2      3
4      i=1  j=0      4
5      i=1  j=1      5
6      i=1  j=2      6
7      i=2  j=0      7
8      i=2  j=1      8
9      i=2  j=2      9
1
2
3
4
5
6
7
8
9_
```



Pointers to character strings with an example program:

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

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

```
char *name;
name = "delhi";
```

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

This type of declarations is not valid for character string

Like:- char name [20];
name ="delhi" ;//invalid

Program:

```
/* pointers and characters strings*/
//length of string using pointers

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

void main()
{
int length=0; char *name,*cptr;

name="Dennis Ritchie";
cptr=name; //assigning one pointer to another
printf("\n ADDRESS OF POINTERS: name=%u\t cptr=%u",name,cptr);

puts("\n entered string is:");

puts(name);
cptr=name;
while(*cptr!='\0') //is true untill end of str is reached
{
printf("\n\t%c is @ %u",*cptr,cptr);
cptr++; //when while loop ends cptr has the address of '\0' in it
}
length=cptr-name; //end of str minus strat of string gives no of chars in between
//i.e. length of str;

printf("\n Length of string is:%d",length);
}
```

Output:

```

ADDRESS OF POINTERS: name=170  cptr=170
entered string is:
Dennis Ritchie
D  i  s  s  e  170
e  i  s  s  e  171
n  i  s  s  e  172
n  i  s  s  e  173
i  s  s  e  174
s  i  s  s  e  175
R  i  s  s  e  176
i  t  c  h  i  e  177
c  i  s  s  e  178
e  s  s  e  179
    i  s  s  e  180
    i  s  s  e  181
    i  s  s  e  182
    i  s  s  e  183
Length of string is:14_
    
```

Pointer to table of string:-

One important use of pointers is in handling of a table of strings.

Consider the following array string :

```
char name [3][25];
```

Here name containing 3 names, each with a maximum length of 25 characters and total storage requirement for the name table is 75 bytes

We know that rarely the individual string will be of equal length instead of making each row a fixed number of characters we can make it a pointer to a string of varying length.

```
Eg:- char *name[3]={"apple", "banana", "pine apple"};
```

This declaration allocates only 21 bytes sufficient to hold all the character

- To print all the 3 names use the following statement ,
for (i=0; i<=2; i++)
printf("%s/n", name [i]);
- To access the jth character in the ith name we may write,
*(name [i]+j)

Note:- The character arrays with the rows of varying length are called "ragged arrays" and are better handled pointers.

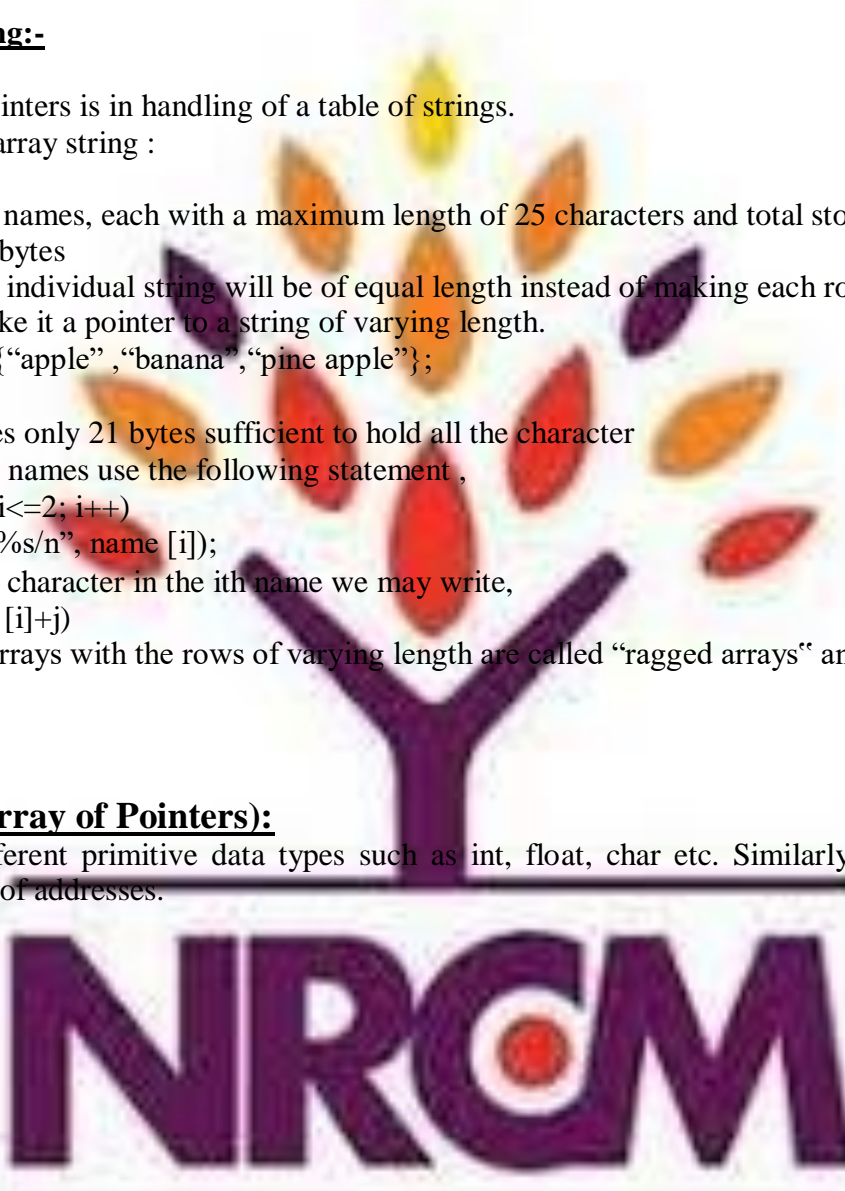
Pointer Arrays (Array of Pointers):

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

Example:

```

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



Output:

Address	Element
4060	10
4062	20
4064	30

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

data_type array_name [exp1] [exp2];

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

data_type *array_name [exp1];

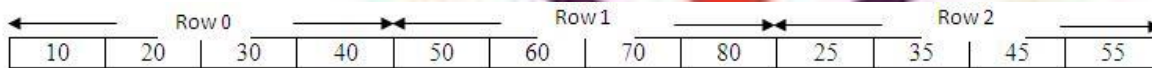
Where,

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

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

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

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



Using array of pointers we can declare p as,

Fig: Two-Dimensional Array

```
int *p[3];
```

Here, p is an array of pointers. p[0] gives the address of the first row, p[1] gives the address of the second row and p[2] gives the address of the third row. Now, p[0]+0 gives the address of the

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

- Address of ith row is given by a[i].
- Address of an item in ith row and jth column is given by, p[i]+j.
- The element in ith row and jth column can be accessed using the indirection operator * by specifying, *(p[i]+j)

Pointers to structures:

We have pointers pointing to int, float, arrays etc., We also have pointers pointing to structures. They are called as structure pointers.

To access the members of a structure using pointers we need to perform The following operations:

- 1 Declare the structure variable
- 2 Declare a pointer to a structure
- 3 Assign address of structure variable to pointer
- 4 Access members of structure using (.) operator or using (->)member selection operator or arrow operator.

Syntax:

```
struct tagname  
  
{  
datatype member1;  
datatype member2;  
...  
};  
struct tagname var;  
struct tagname *ptr;  
ptr=*var;
```

to access the members

(*ptr).member1; or ptr->member1;

The parentheses around *ptr are necessary because the member operator "." has a higher precedence than the operator ",".

Example:

```
struct student  
{  
int rno;  
char name[20];  
};  
struct student s1; struct student *ptr;  
ptr=&s1;  
to access (*ptr).rno; (*ptr).name;  
(or)  
ptr->rno; ptr->name;
```

Ex: Program to read and display student details using pointers to structures

```
struct student  
{
```



```
int HTNO;
char NAME[20];
float AVG;
};
void main()
{
    struct student s1; struct student *ptr; ptr=&s1;
    printf("Enter student details");
    scanf("%d%s%f",&ptr->HTNO,ptr->NAME,&ptr->AVG);
    printf("HTNO=%d",ptr->HTNO); printf("NAME=%s",ptr->NAME);
    printf("AVERAGE MARKS=%f",ptr->AVG);
}
```

Pointers and Functions :

Pointers as Function Arguments.

Pointers can act as function arguments when addresses are passed as actual arguments by the calling function, in the called function the formal arguments must be pointers.

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

Example :-

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

When the function change () is called, the address of the variable x, not its value, is passed into the function change (). Inside change (), the variable p is declared as a pointer and therefore p is the address of the variable x. The statement,

```
*p = *p + 10;
```

Means —add 10 to the value stored at the address p. Since p represents the address of x, the value of x is changed from 20 to 30. Thus the output of the program will be 30, not 20.

Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function.

Pointers as Functions return type:

Functions Returning Pointers:

A function can return an address through a pointer. Functions return indirectly the values using pointers. The return type of function can be a pointer of type int , float ,char ,struct etc.

Example :

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

```
int * smallest(int *a, int *b)
{
if(*a<*b)
return a;
else
return b;
}
```

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

Pointers to function – Function Pointer :

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

A pointer to a function can be declared as follows.

```
type (*fptr);
```

This tells the compiler that fptr is a pointer to a function which returns type value the parentheses around *fptr is necessary.

Because type *gptr(); would declare gptr as a function returning a pointer to type.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer.

Eg: double mul(int,int);


```
double (*p1)();
```

```
p1=mul();
```

It declares p1 as a pointer to a function and mul as a function and then make p1 to point to the function mul.

To call the function mul we now use the pointer p1 with the list of parameters.
i.e (*p1)(x,y); //function call equivalent to mul(x,y);

Program:

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

double mul(int a,int b)
{
double val; val=a*b;
return(val);
}
```

Output:

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

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

The logo for NRCM (National Resource Centre for Micro-Entrepreneurship) is displayed in a stylized purple font. The letters 'N', 'R', and 'M' are large and bold, while the 'C' is smaller and positioned between the 'R' and 'M'. A red circle is integrated into the design, partially overlapping the 'C' and the 'M'.

UNIT-III

Preprocessor: Commonly used Preprocessor commands like include, define, undef, if, ifdef, ifndef Files: Text and Binary files, Creating and Reading and writing text and binary files, Appending data to existing files, Writing and reading structures using binary files, Random access using fseek, ftell andrewind functions.

Preprocessors:

The C Preprocessor is not part of the compiler but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP. All preprocessor commands begin with a pound symbol #. It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column.

Invoked automatically by the C compiler o 1st pass: invokes Cpreprocessor

o 2nd pass: invokes compiler on the resulting C code

Manually invoke C preprocessor gcc -Efoo.c

This section lists Three (3) kinds of preprocessor directives:

1.1 File Inclusion (#include):

1.1 #include: Inserts a particular header from another file

Eg: #include <stdio.h> #include "myheader.h"

These directives tell the CPP to get stdio.h from System Libraries and add the text to the current

source file. The next line tells CPP to get myheader.h from the local directory and add the content to the current source file.

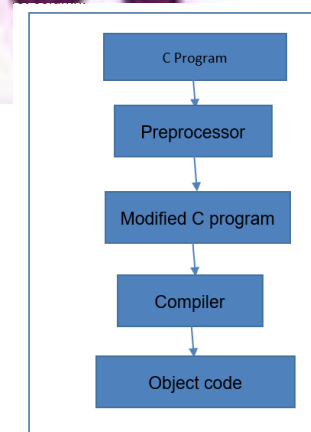


Fig1.1 Execution process Diagram

Conditional compilation (#if, #ifdef, #ifndef, #elif, #else, #endif):

#if:

The #if directive tests an expression. This expression can be of any form used in a C program, with virtually any operators, except that it can include only integer constant values. No variables, or function calls are permitted, nor are floating point, character or string constants. The #if directive is true, if the expression evaluates to true (nonzero). Any undefined preprocessor symbol used in the #if expression is treated as if it has the value 0. Using a symbol that is defined with no value does not work with all preprocessors, and an attempt to do so might result in an error.

Eg:

```
#include <stdio.h>
#define WINDOWS 1
int main()
{
    #if WINDOWS
    printf("Windows");
    #endif
    return 0;
}
```

Here is the output of the executable program: Windows

#ifdef:

Returns true if this macro is defined

Eg:

```
#ifdef DEBUG
/* Your debugging statement enter here */ #endif
```

This tells the CPP to do the process the statements enclosed if DEBUG is defined.

This is useful if you pass the *-DDEBUG* flag to gcc compiler at the time of compilation.

This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

#ifndef: Returns true if this macro is not defined

Eg:

```
#ifndef MESSAGE
#define MESSAGE "You wish!" #endif
```

This tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

Few other

#else: The alternative for #if

#elif: #else an #if in one statement

#endif: Ends preprocessor conditional

Macros (#define, #undef, #error, #pragma):

#define: Substitutes a preprocessor macro

eg: #define MAX_ARRAY_LENGTH 20. This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

#undef: Undefines a preprocessor macro Eg: #undef FILE_SIZE

```
#define FILE_SIZE 42
```

This tells the CPP to undefine existing FILE_SIZE and define it as 42.

#error: Prints error message on stderr

#pragma: Issues special commands to the compiler, using a standardized method

Preprocessors Examples Predefined Macros

ANSI C defines a number of macros. Although each one is available for your use in programming, the predefined macros should not be directly modified.

Macro Description

DATE_____The current date as a character literal in "MMM DD YYYY"format

TIME_____The current time as a character literal in "HH:MM:SS"format

FILE_____This contains the current filename as a stringliteral.

LINE_____This contains the current line number as a decimalconstant.

STDC_____Defined as 1 when the compiler complies with the ANSIstandard.

Let's try the following example:

```
#include <stdio.h>
main()
{
printf("File:%s\n",FILE_____);
printf("Date:%s\n",DATE_____);
printf("Time:%s\n",TIME_____);
printf("Line:%d\n",LINE_____);
printf("ANSI:%d\n", STDC_____);
}
```

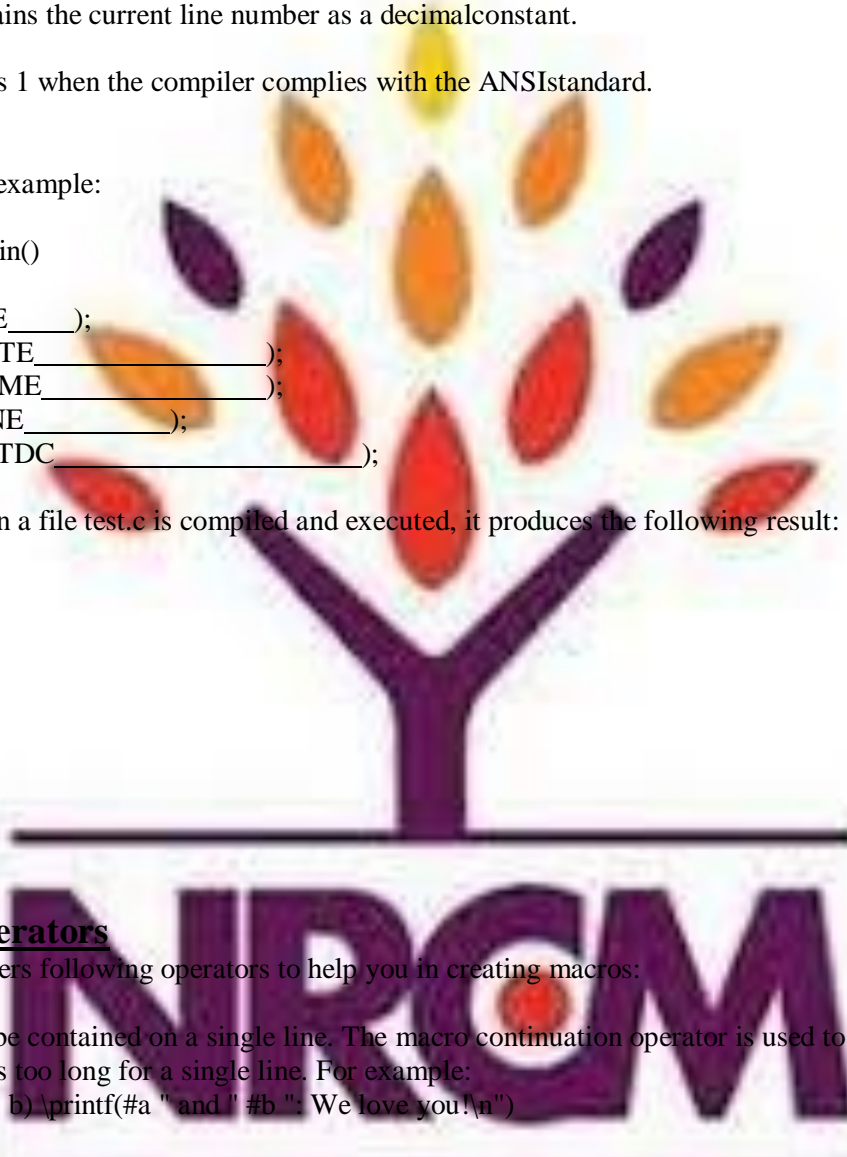
When the above code in a file test.c is compiled and executed, it produces the following result:
File :test.c

Date :Jun 2 2012

Time :03:36:24

Line :8

ANSI :1



Preprocessor Operators

The C preprocessor offers following operators to help you in creating macros:

Macro Continuation (\)

A macro usually must be contained on a single line. The macro continuation operator is used to continue a macro that is too long for a single line. For example:

```
#define message_for(a, b) \printf(##a " and " ##b ": We love you!\n")
```

Stringize #

The stringize or number-sign operator '#', when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro that has a specified argument or parameter list. For example:

```
#include <stdio.h>
#define message_for(a, b) \
printf(##a " and " ##b ": We love you!\n")
int main(void)
{
message_for(Carole, Debra); return 0;
}
```



```
}
```

When the above code is compiled and executed, it produces the following result: Carole and Debra: We love you!

Token Pasting ##

The token-pasting operator ## within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example: #include <stdio.h>

```
#define tokenpaster(n)
printf ("token" #n " = %d", token##n)
int main(void)
{
int token34 = 40;
tokenpaster(34);
return 0;
}
```

When the above code is compiled and executed, it produces the following result: token34 =40

How it happened, because this example results in the following actual output from the preprocessor:

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both stringize and token-pasting. The defined Operator

The preprocessor defined operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true *non-zero*. If the symbol is not defined, the value is false *zero*. The defined operator is specified as follows: #include <stdio.h>

```
#if !defined (MESSAGE) #define MESSAGE "You wish!" #endif
```

```
int main(void)
{
printf("Here is the message: %s\n", MESSAGE); return 0;
}
```

When the above code is compiled and executed, it produces the following result: Here is the message: You wish!

Parameterized Macros:

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows:

```
int square(int x) {
return x * x;
}
```

We can rewrite above code using a macro as follows: #define square(x) ((x) * (x))

Macros with arguments must be defined using the #define directive before they can be used.

The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between and macro name and open parenthesis.

For example:

```
#include <stdio.h>
#define MAX(x,y)
((x) > (y) ? (x) : (y))
int main(void)
{
printf("Max between 20 and 10 is %d\n", MAX(10, 20));
return 0;
}
```

When the above code is compiled and executed, it produces the following result: Max between 20 and 10 is 20

CREATING YOUR OWN HEADER FILES IN C :

1. #include is to incorporate the predefined header files in to our c programs...

```
#include<string.h>
```

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

CAN WE HAVE OUR OWN HEADER FILES???

YES....

1. It allows the programmers to create their own header files ...
2. #include is also used to include / incorporate user created header files into the programs....

Two ways...

1. #include<stdio.h>> the computer will search for the definitions header files in predefined / standard..locations
2. #include"myown.h" ---> the computer will search for the definitions header files in the current folder (own heard files...)

How to write your own header file in C?

As we all know that files with .h extension are called header files in C.

These header files generally contain function declarations which we can be used in our main C program, like for e.g. there is need to include stdio.h in our C program to use function printf() in the program.

So the question arises, **is it possible to create your own header file?**

The answer to the above is yes.

header files are simply files in which you can declare your own functions that you can use in your main program or these can be used while writing large C programs.

NOTE: Header files generally contain definitions of data types, function prototypes and C preprocessor commands.

Below is the short example of creating your own header file and using it accordingly.

Creating myhead.h : Write the below code and then save the file as myhead.h or you can give any name but the extension should be .h indicating its a header file.

```
// It is not recommended to put function definitions
```

```
// in a header file. Ideally there should be only
```

```
// function declarations. Purpose of this code is
```

```
// to only demonstrate working of header files.
```

```
void add(int a, int b)
```

```
{
```

```
    printf("Added value=%d\n", a + b);
```

```
}
```

```
void multiply(int a, int b)
```

```
{
```

```
    printf("Multiplied value=%d\n", a * b);
```

```
}
```

Including the .h file in other program : Now as we need to include stdio.h as #include in order to use printf() function.

We will also need to include the above header file myhead.h as #include"myhead.h".

The "" here are used to instructs the preprocessor to look into the present folder and into the standard folder of all header files if not found in present folder.

So, if you wish to use angular brackets instead of "" to include your header file you can save it in the standard folder of header files otherwise.

If you are using “ ”you need to ensure that the header file you created is saved in the same folder in which you will save the C file using this header file.

Using the created header file:

```
// C program to use the above created header file
#include <stdio.h>
#include "myhead.h"
int main()
{
    add(4, 6);

    /*This calls add function written in myhead.h
    and therefore no compilation error.*/
    multiply(5, 5);

    // Same for the multiply function in myhead.h
    printf("BYE!See you Soon");
    return 0;
}
```

Output:

```
Added value:10
Multiplied value:25
BYE! See you Soon
```

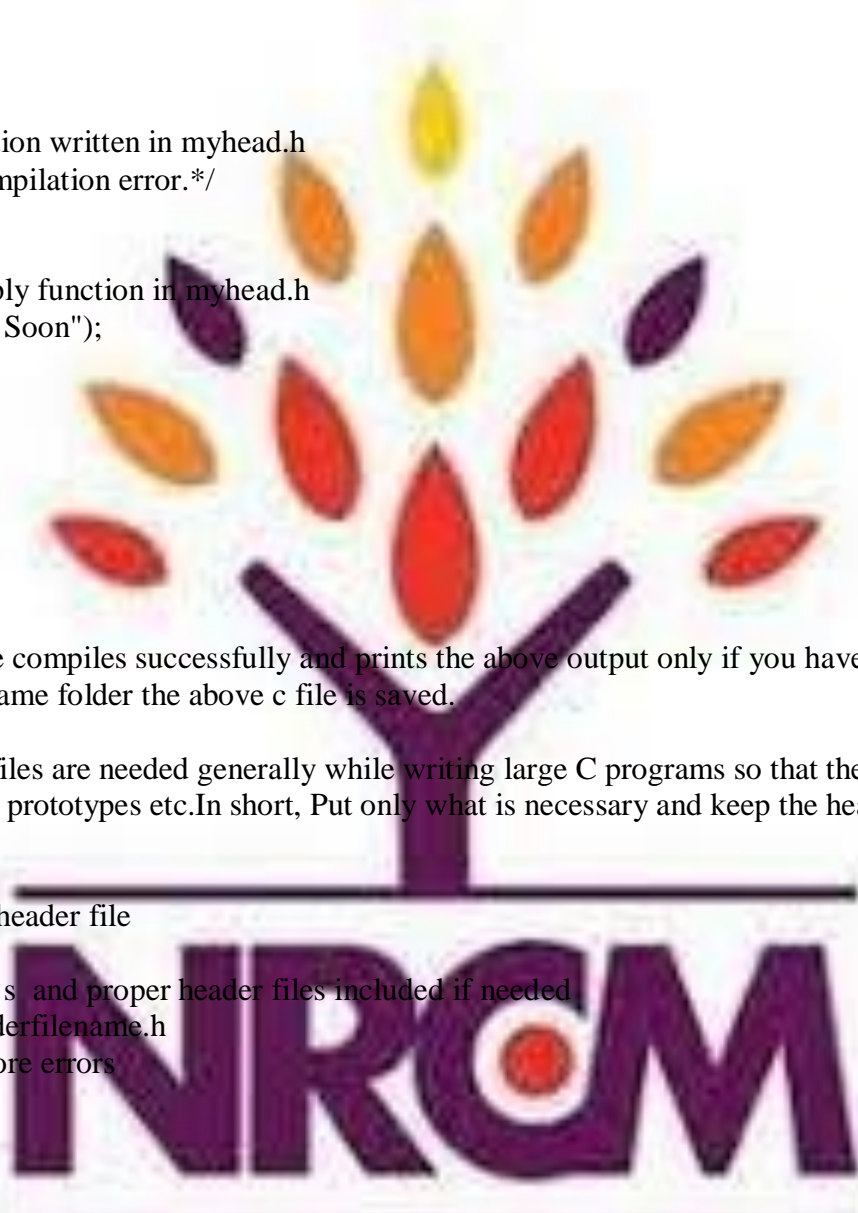
NOTE: The above code compiles successfully and prints the above output only if you have created the header file and saved it in the same folder the above c file is saved.

Important Points:

The creation of header files are needed generally while writing large C programs so that the modules can share the function definitions, prototypes etc.In short, Put only what is necessary and keep the header file concised.

Example Program:

```
//step -1 : creating own header file
// open a new file
// define your function / s and proper header files included if needed
// save the file with headerfilename.h
// compile the file ...ignore errors
// run the file ...
#include<stdio.h>
int sum(int x ,int y)
{
    return(x+y);
}
int findbig(int x,int y)
{
    if(x>y)
    return(x);
    else
    return(y);
}
```




```
}  
void display()  
{  
printf("\n HELLO FROM CSE A C E\n");  
}  
  
// write a new program to include the header file created  
#include<stdio.h>  
#include<conio.h>  
#include"cseace.h"  
int main()  
{  
int x=10,y=20,res;  
clrscr();  
display();  
res=sum(x,y);  
printf("sum = %d",res);  
printf("\n BIGGER OF %d and %d = %d",x,y,findbig(x,y));  
display();  
getch();  
return 0;  
}
```

OUTPUT :

HELLO FROM CSE A C E

SUM=30

BIGGER OF 10 and 20 = 30

HELLO FROM CSE A C E

File : A file is an external collection of related data treated as a unit. A file is a place on a disk where a group of related data is stored and retrieved whenever necessary without destroying data

The primary purpose of a file is to keep record of data. Record is a group of related fields. Field is a group of characters which convey meaning.

Files are stored in auxiliary or secondary storage devices. The two common forms of secondary storage are disks (hard disk, CD and DVD) and tapes.

Each file ends with an end of file (EOF) at a specified byte number, recorded in file structure.

A file must first be opened properly before it can be accessed for reading or writing. When a file is opened an object (buffer) is created and a stream is associated with the object.

It is advantageous to use files in the following circumstances.

When large volume of data are handled by the program and

When the data need to be stored permanently without getting destroyed when program is terminated.

There are two kinds of files depending upon the format in which data is stored:

- 1) Text files
- 2) Binary files

Textfiles:

A text file stores textual information like alphabets, numbers, special symbols, etc.

actually the ASCII code of textual characters its stored in text files. Examples of some text files include c.java,c++sourcecodefilesandfiles with.txt extensions. The text file contains the characters in sequence. The computer process the text files sequentially and in forward direction. One can perform file reading, writing and appending operations. These operations are performed with the help of inbuilt functions of c.

Binary files:

Text mode is inefficient for storing large amount of numerical data because it occupies large space. Only solution to this is to open a file in binary mode, which takes less space than the text mode. These files contain the set of bytes which stores the information in binary form. One main drawback of binary files is data is stored in human unreadable form. Examples of binary files are .exe files, video stream files, image files etc. C language supports binary file operations with the help of various inbuilt functions

Modes of opening files :

To store data in a file three things have to be specified for operating system. They include
FILENAME:

It is a string of characters that make up a valid file name which may contain two parts, a primary name and an optional period with the extension.

Prog1.c

Myfirst.java Data.t

xt store

DATASTRUCTURE:

It is defined as FILE in the library of standard I/O function definitions. Therefore all files are declared as type FILE. FILE is a define data type.

FILE *fp;

PURPOSE:

It defines the reason for which a file is opened and the mode does this job. fp=fopen("filename","mode");

The different modes of opening files are :

"r" (read) mode:

The read mode (r) opens an existing file for reading. When a file is opened in this mode, the file marker or pointer is positioned at the beginning of the file (first character). The file must already exist; if it does not exist a NULL is returned as an error. If we try to write a file opened in read mode, an error occurs.

Syntax: fp=fopen ("filename","r");

"w" (write) mode:

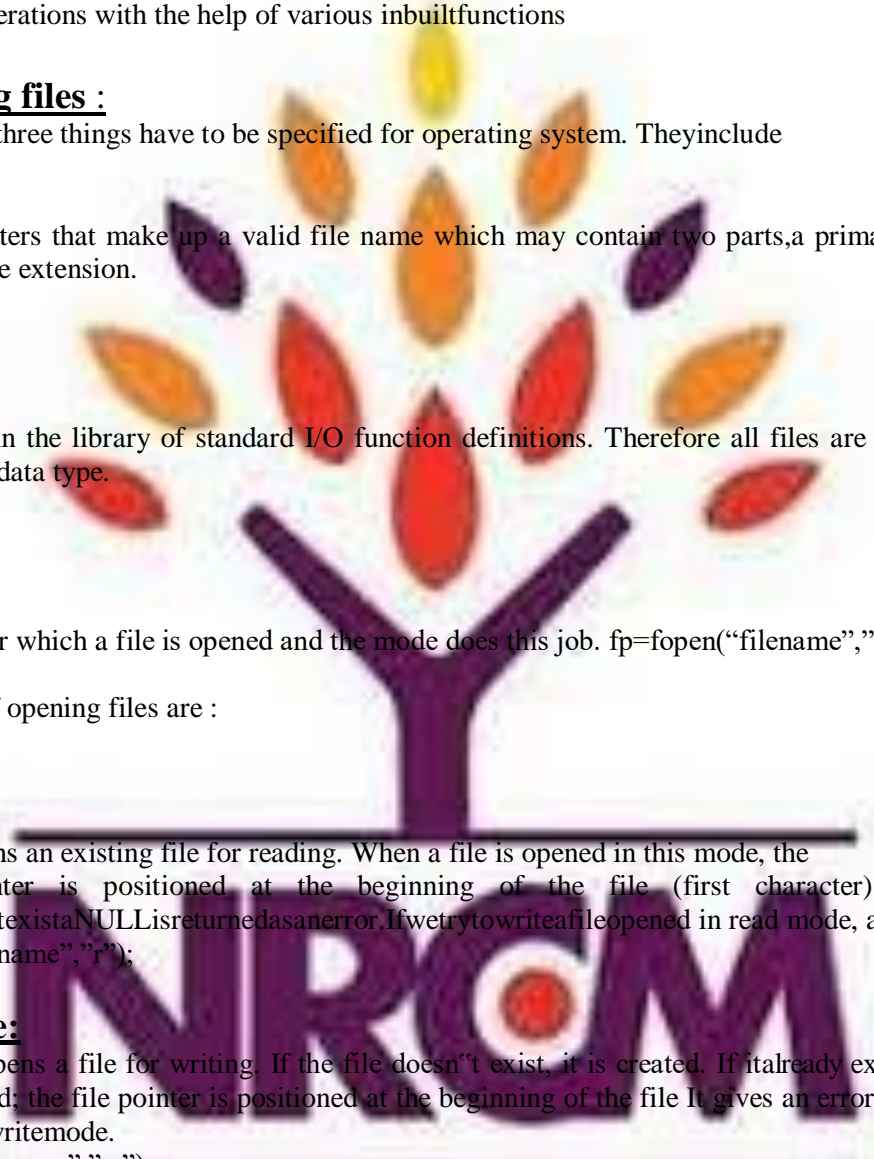
The write mode (w) opens a file for writing. If the file doesn't exist, it is created. If it already exists, it is opened and all its data is erased; the file pointer is positioned at the beginning of the file. It gives an error if we try to read from a file opened in write mode.

Syntax: fp=fopen ("filename","w");

"a" (append) mode:

The append mode (a) opens an existing file for writing instead of creating a new file. However, the writing starts after the last character in the existing file, that is new data is added, or appended, at the end of the file. If the file doesn't exist, new file is created and opened. In this case, the writing will start at the beginning of the file.

Syntax: fp=fopen ("filename","a");



“r+” (read and write) mode:

In this mode file is opened for both reading and writing the data. If a file does not exist then NULL, is returned.
 Syntax: fp=fopen (“filename”,”r+”);

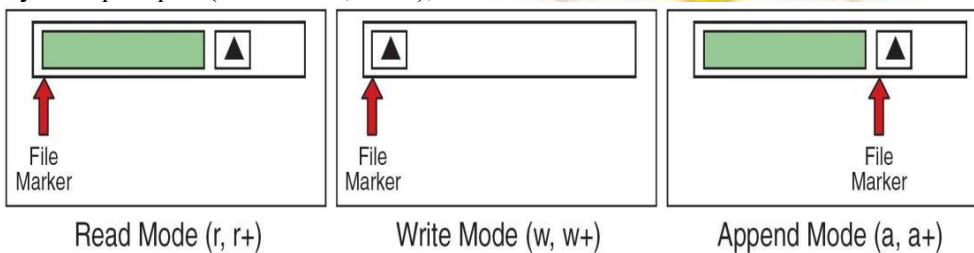
“w+” (read and write) mode:

In this mode file is opened for both writing and reading the data. If a file already exists its contents are erased and if a file does not exist then a new file is created.
 Syntax:fp=fopen (“filename”,”w+”);

“a+” (append and read) mode:

In this mode file is opened for reading the data as well as data can be added at the end.

Syntax:fp=fopen (“filename”, “a+”);



NOTE: To perform operations on binary files the following modes are applicable with an extension b like rb,wb,ab,r+b,w+b,a+b,which has the same menaing but allows to perform operationson binaryfiles.

Basic operations on file:

- Naming afile
- Opening afile
- Reading data fromfile
- Writing data intofile
- Closing afile

In order to perform the basic file operations C supports a number of functions .Some of the Important file handling functions available in the C library are as follows:

FUNCTION NAME	OPERATION
fopen()	Creates a new file for use or Opens an existing file for use
fclose()	Closes a file which has been opened for use
fcloseall()	Closes all files which are opened
getc()/fgetc()	Reads a character from a file
putc()/fputc()	Writes a character to a file
fprintf()	Writes a set of data values to files
fscanf()	Reads a set of data values from files

getw()	Reads an integer from file
putw()	Writes an integer to a file
gets()	Reads a string from a file
puts()	Writes a string to a file
fseek()	Sets the position to a desired point in a file
ftell()	Gives the current position in the file
rewind()	Sets the position to the beginning of the file

Naming and opening a file:

A name is given to the file used to store data. The filename is a string of characters that make up a valid file name for operating system. It contains two parts. A primary name and an optional period with the extension.

Examples: Student.dat, file1.txt, marks.doc, palin.c
 The general format of declaring and opening a file is

```
FILE *fp; //declaration
fp=fopen ("filename",mode"); //statement to open file.
```

Here FILE is a data structure defined for files. fp is a pointer to data type FILE. filename is the name of the file. mode tells the purpose of opening this file.

Reading data from file:

Input functions used are (Input operations on files)

getc(): It is used to read characters from file that has been opened for read operation.

Syntax: c=getc (filepointer);

This statement reads a character from file pointed to by file pointer and assign to c. It returns an end-of-file marker EOF, when end of file has been reached

fscanf();

This function is similar to that of scanf function except that it works on files.

Syntax: fscanf (fp, "control string", list);

Example fscanf(fl, "%s%d", str, &num);

The above statement reads string type data and integer type data from file.

getw(); This function reads an integer from file.

Syntax: getw (file pointer);

fgets(): This function reads a string from a file pointed by a file pointer. It also copies the string to a memory location referred by an array.

Syntax: fgets(string,no of bytes,filepointer);

fread(): This function is used for reading an entire structure block from a given file.

Syntax: fread(&struct_name,sizeof(struct_name),1,filepointer);

Writing data to a file:

To write into a file, following C functions are used

putc(): This function writes a character to a file that has been opened in write mode.

Syntax: putc(c,fp);

This statement writes the character contained in character variable c into a file whose pointer is fp.

fprintf(): This function performs function, similar to that of printf.

Syntax: fprintf(f1,"%s,%d",str,num); **c. putw():** It writes an integer to a file.

Syntax: putw (variable, fp);

d. **fputs():** This function writes a string into a file pointed by a file pointer.

Syntax: fputs(string, filepointer);

e. **fwrite():** This function is used for writing an entire block structure to a given file.

Syntax: fwrite(&struct_name,sizeof(struct_name),1,filepointer);

Closing a file:

A file is closed as soon as all operations on it have been completed. Closing a file ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. Another instance where we have to close a file is to re-open the same file in a different mode. Library function for closing a file is,

fclose(file pointer);

Example: fclose(fp);

Where fp is the file pointer returned by the call to fopen(). fclose() returns 0 on success (or) - 1 on error. Once a file is closed, its file pointer can be reused for another file. Note: fcloseall() can be used to close all the opened files at once.

File I/O functions :

In order to perform the file operations in C we must use the high level I/O functions which are in C standard I/O library. They are

getc() and fgetc() functions:

getc()/fgetc() : It is used to read a character from a file that has been opened in a read mode. It reads a character from the file whose file pointer is fp. The file pointer moves by one character for every operation of getc(). The getc() will return an end-of –marker EOF, when an end of file has been reached.

Syntax:

getc(fp);Ex:

```
char ch; ch=getc(fp);
```

putc()/fputc() -:

It is used to write a character contained in the character variable to the

file associated with the FILE pointer fp. fputc() also returns an end-of –marker EOF, when an end of file has been reached.

Syntax: putc(c,fp); Example: char c; **putc(c,fp);**

Program using fgetc() and fputc():

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

```
{  
FILE *fp; char ch;
```

```
fp=fopen("input1.txt","w");
```

```
printf("\n enter some text hereand press cntrl D or Z to stop:\n"); while((ch=getchar())!=EOF)
```

```
fputc(ch,fp); fclose(fp); fp=fopen("input1.txt","r"); printf("\n The entered text is : \n");
```

```
while((ch=fgetc(fp))!=EOF)
```

```
putchar(ch);
```

```
fclose(fp);
```

```
}
```

fprintf() andfscanf():

In order to handle a group of mixed data simultaneously there are two functions that are fprintf()andfscanf().Thesetwofunctionsareidenticaltoprintfandscanffunctions,exceptthat they work on files. The first argument of these functions is a file pointer which specifies the file to be used.

fprintf(): The general form of fprintf() is

Syntax: fprintf(fp,"control string",list);

where fp is a file pointer associated with a file that has been opened for writing . The control string contains output specifications for the items in the list. .

Example:fprintf(fp,"%s%d",name,age);

fscanf() : It is used to read a number of values from a file.

Syntax: fscanf(fp,"control string",list); Example: fscanf(fp2,"%s%d",item,&quantity);

like scanf , fscanf also returns the number of items that are successfully read. when the end of file is reached it returns the valueEOF.

Program using fscanf() andfprintf():

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

```
> void main()
```

```
{
```

```
int a=22,b; char
```

```
s1[20]="Welocme_to_c",s2[20]; float c=12.34,d;
```

```
FILE *f3;
```

```
f3=fopen("mynew3","w"); fprintf(f3,"%d %s
```

```
%f",a,s1,c); fclose(f3);
```

```
f3=fopen("mynew3","r"); fscanf(f3,"%d %s
```

```
%f",&b,s2,&d);
```

```
printf("\n a=%d \t s1=%s \t c=%f \n b=%d \t s2=%s \td=%f",a,s1,c,b,s2,d); fclose(f3);
}
```

getw() and putw():

The `getw()` and `putw()` are integer oriented functions. They are similar to the `getc()` and `putc()` functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general form of `getw()` and `putw()` are

Syntax: `putw(integer,fp)`; Syntax: `getw(fp)`;

Program using getw() and putw():

```
/*Printing odd numbers in odd file and even numbers in even file*/ #include<stdio.h>
void main()
{
int x,i;
FILE *f1,*fo,*fe;//creating a file pointer f1=fopen("anil.txt","w");//opening a file
printf("\n enter some numbers into file or -1 to stop\n"); for(i=0;i<20;i++)
{
scanf("%d",&x)
; if(x== - 1)break;
putw(x,f1); //writing read number into anil.txt file one at a time
}
fclose(f1); //closing a file opened for writing input

printf("\n OUTPUT DATA\n"); f1=fopen("anil.txt","r");//open anil in read mode to read data
fo=fopen("odd3f","w");
fe=fopen("even3f","w"); while((x=getw(f1))!=EOF)
{
printf("%d\t",x); if(x%2==0) putw(x,fe);
else putw(x,fo);
}
fcloseall(); fo=fopen("odd3f","r");
printf("\n contents of odd file are :\n"); while((x=getw(fo) )!= EOF)
printf(" %d\t",x);
fe=fopen("even3f","r");
printf("\n contents of even file are :\n"); while((x=getw(fe) )!= EOF)
printf("
%d\t",x); fcloseall();
}
```

fputs() and fgets():

fgets(): It is used to read a string from a file pointed by file pointer. It copies the string to a memory location referred by an array.

Syntax: `fgets(string,length,filepointer)`; Example: `fgets(text,50,fp1)`;

fputs(): It is used to write a string to an opened file pointed by file pointer.

Syntax: `fputs(string,filepointer)`; Example: `fputs(text,fp)`; Program using `fgets()` and `fputs()`:

```
#include<stdio.h> voidmain()
{
FILE *fp; charstr[50];
fp=fopen("fputget.txt","r"); printf("\n the read string is :\n"); fgets(str,50,fp);
puts(str); fclose(fp);
}
```

```
fp=fopen("fputget.txt","a+"); printf("\n Enter string : \n"); gets(str);
fputs(str,fp); puts(str); fclose(fp);
}
```

Block or structures read and write:

Large amount of integers or float data require large space on disk in text mode and turns out to be inefficient .For this we prefer binary mode and the functions used are fread() and fwrite():

fwrite(): It is used for writing an entire structure block to a given file in binary mode.

Syntax: **fwrite(&structure variable,sizeof(structure variable),1,filepointer);**

Example: **fwrite(&stud,sizeof(stud),1,fp);**

fread(): It is used for reading an entire structure block from a given file in binary mode.

Syntax: **fread(&structure variable,sizeof(structure variable),1,filepointer);**

Example: **fread(&emp,sizeof(emp),1,fp1);**

Program using fread() and fwrite():

```
#include<stdio.h>
struct player
{
char pname[30]; intage;
int runs; };
void main()
{
struct player p1,p2; FILE *f3;
f3=fopen("player.txt","w");
printf("\n Enter details of player name ,age and runs scored : \n"); fflush(stdin);
scanf("%s %d %d",p1.pname,&p1.age,&p1.runs);
fwrite(&p1,sizeof(p1),1,f3); fclose(f3); f3=fopen("player.txt","r"); fread(&p2,sizeof(p2),1,f3); fflush(stdout);
printf("\nPLAYERNAME:=%s\tAGE:=%d\tRUNS:=%d",p2.pname,p2.age,p2.runs
); fclose(f3);
}
```

Random access to files :

At times we needed to access only a particular part of a file rather than accessing all the data sequentially, which can be achieved with the help of functions **fseek**, **ftell** and **rewind** available in IO library.

ftell():-

ftell takes a file pointer and returns a number of type **long**, that corresponds to the current position. This function is useful in saving the current position of the file, which can later be used in the program.

Syntax: **n=ftell(fp);**

n would give the Relative offset (In bytes) of the current position. This means that already **n** bytes have a been read or written

rewind():-

It takes a file pointer and resets the position to the start of the file.

Syntax: **rewind(fp);**

n=ftell(fp);

would assign 0 to **n** because the file position has been set to start of the file by rewind(). The first byte in the file is numbered 0, second as 1, so on. This function helps in reading the file more than once, without having to close and open the file.

Whenever a file is opened for reading or writing a rewind is done implicitly.

fseek:- fseek function is used to move the file pointer to a desired location within the file.

Syntax: fseek(fileptr,offset,position);

file pointer is a pointer to the file concerned, offset is a number or variable of type long and position is an integer number which takes one of the following values. The offset specifies the number of positions(**Bytes**) to be moved from the location specified by the position which can be positive implies moving forward and negative implies moving backwards.

POSITION VALUE	VALUE CONSTANT	MEANING
0	SEEK_SET	BEGINNING OF FILE
1	SEEK_CUR	CURRENT POSITION
2	SEEK_END	END OF FILE

Example: **fseek(fp,10,0) ;**

fseek(fp,10,SEEK_SET); // file pointer is repositioned in the forward direction 10 bytes. **fseek(fp,-10,SEEK_END);** // reads from backward direction from the end of file.

When the operation is successful fseek returns 0 and when we attempt to move a file beyond boundaries fseek returns -1. Some of the Operations of fseek function are as follows:

STATEMENT	MEANING
fseek(fp,0L,0);	Go to beginning similar to rewind()
fseek(fp,0L,1);	Stay at current position
fseek(fp,0L,2);	Go to the end of file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1) th byte in the file.
fseek(fp,m,1);	Go forward by m bytes
fseek(fp,-m,1);	Go backwards by m bytes from the current position
fseek(fp,-m,2);	Go backwards by m bytes from the end.(positions the file to the m th character from the end)

Program on random access to files:

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



```
FILE *fp;
char ch; fp=fopen("my1.txt","r"); fseek(fp,21,SEEK_SE
T); ch=fgetc(fp);
while(!feof(fp))
{
printf("%c\t",ch); printf("%d\n",ftell(fp
)); ch=fgetc(fp);
}
rewind(fp); printf("%d\n",ftell(fp
)); fclose(fp);
}
```

Error handling in files:

It is possible that an error may occur during I/O operations on a file. Typical error situations include:
Trying to read beyond the end of file mark. Device overflow.
Trying to use a file that has not been opened.

Trying to perform an operation on a file, when the file is opened for another type of operations.
Opening a file with an invalid filename. Attempting to write a write protected file.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output. In C we have two status-inquiry library functions **feof** and **ferror** that can help us detect I/O errors in the files.

a). feof(): The feof() function can be used to test for an end of file condition. It takes a FILE pointer as its only argument and returns a non zero integer value if all of the data from the specified file has been read, and returns zero otherwise. If fp is a pointer to file that has just opened for reading, then the statement
if(feof(fp)) printf("End of data");
would display the message "End of data" on reaching the end of file condition.

ferror(): The ferror() function reports the status of the file indicated. It also takes a file pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise. The statement
if(ferror(fp)!=0)

```
printf("an error has occurred\n");
```

would print an error message if the reading is not successful

fp==null: We know that whenever a file is opened using fopen function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a null pointer. This facility can be used to test whether a file has been opened or not.

Example

```
if(fp==NULL)
```

```
printf("File could not be opened.\n");
```

perror(): It is a standard library function which prints the error messages specified by the compiler. For example:
if(ferror(fp)) perror(filename);

Program for error handling in files:

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

```

*fp; char ch;
fp=fopen("my1.txt","r"); if(fp==NULL) printf("\n file cannotbe
opened"); while(!feof(fp))
{
ch=getc(fp); if(ferror(fp))
perror("problem in file"); else
putchar(ch);
}
fclose(fp);
}
    
```

Introduction to stdin, stdout and stderr (or) Special File Pointers:

When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

There are 3 special FILE *s that are always defined for a program. They are stdin (*standard input*), stdout (*standard output*) and stderr (*standard error*).

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

Standard Input

Standard input is where things come from when you use scanf(). In other words, scanf("%d", &val); is equivalent to the following fscanf():
fscanf(stdin, "%d",&val);

Standard Output

Similarly, *standard output* is exactly where things go when you use printf(). In other words, printf("Value = %d\n", val); is equivalent to the following fprintf():
fprintf(stdout, "Value = %d\n", val);

Remember that standard input is normally associated with the keyboard and standard output with the screen, unless *redirection* is used.

Standard Error

Standard error is where you should display error messages. We've already done that above: `fprintf(stderr, "Can't open input file in.list!\n");`

Standard error is normally associated with the same place as standard output; however, redirecting standard output does not redirect standard error.

For example,

```
% a.out > outfile
```

only redirects stuff going to standard output to the file **outfile**... anything written to standard error goes to the screen.

Simulate a c program to read and display the contents of a file?

Program:

```
#include<stdio. h>
void main()
{
FILE*f1;
char ch;
f1=fopen("data.txt","w");
printf("\n enter some text here and press cntrl D or Z to stop:\n");
while((ch=getchar())!=EOF)
fputc(ch,f1);
fclose(f1);
printf("\n the contents of file are \n:");
f1 = fopen("data.txt","r");
while( ( ch = fgetc(f1) ) != EOF )
putchar(ch);
fclose(f1);
}
```

Develop a c program to copy the contents of one file to another?

Program:

```
#include<stdio.h>
void main()
{
FILE *f1,*f2;
char ch;
f1=fopen("mynew2.txt","w");
printf("\n enter some text here and press cntrl D or Z to stop :\n");
while((ch=getchar())!=EOF)
fputc(ch,f1);
fclose(f1);
f1=fopen("mynew2.txt","r");
f2=fopen("dupmynew2.txt","w");
while((ch=fgetc(f1))!=EOF)
putc(ch,f2);
fcloseall();
printf("\n the copied file contents are :");
f2 = fopen("dupmynew2.txt","r");
while( ( ch = fgetc(f2) ) != EOF )
```



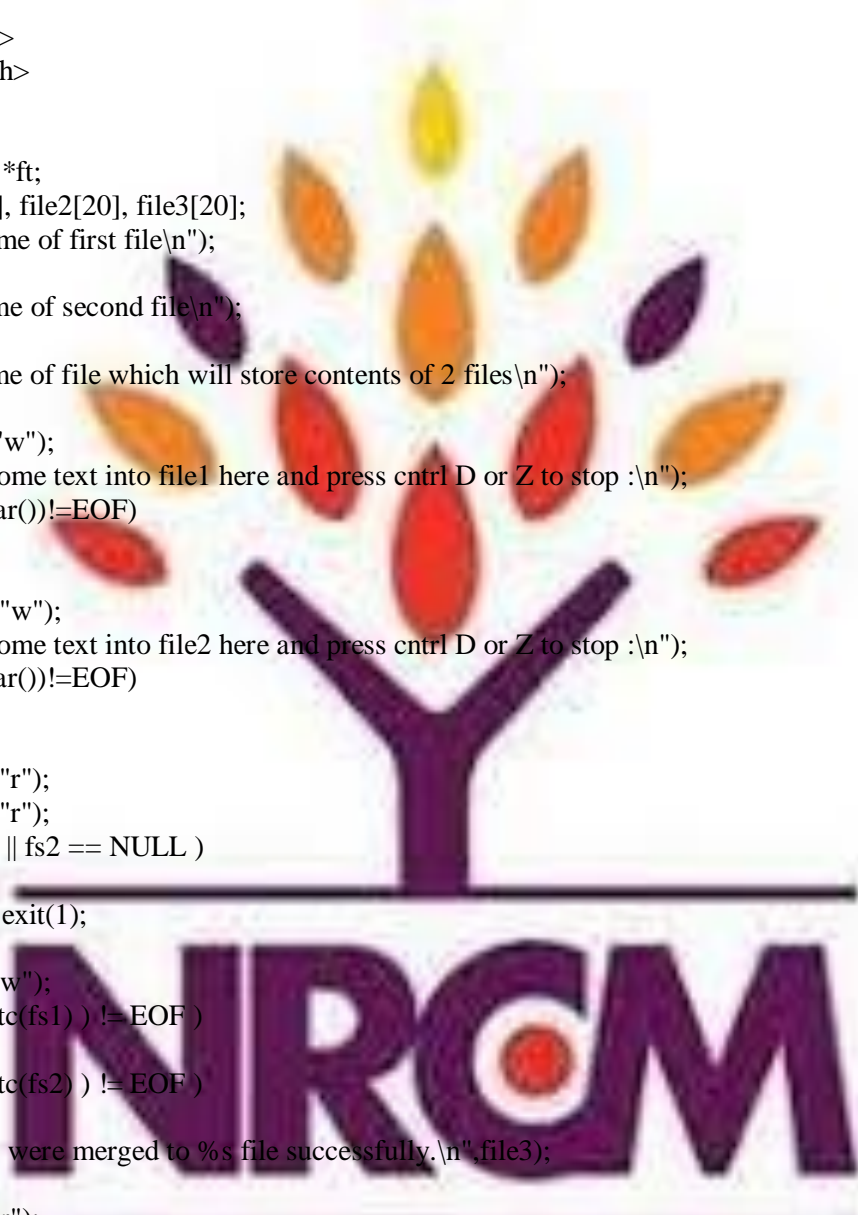
```
putchar(ch);
fclose(f2);
}
```

Write a c program to merge two files into a third file?(Or)

Develop a c program for the following .there are two input files named “first.dat” and “second.dat” .The files are to be merged. That is,copy the contents of first.dat andthen second.dat to a new file named result.dat?

Program:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
FILE *fs1, *fs2, *ft;
char ch, file1[20], file2[20], file3[20];
printf("Enter name of first file\n");
gets(file1);
printf("Enter name of second file\n");
gets(file2);
printf("Enter name of file which will store contents of 2 files\n");
gets(file3);
fs1=fopen(file1,"w");
printf("\n enter some text into file1 here and press cntrl D or Z to stop :\n");
while((ch=getchar())!=EOF)
fputc(ch,fs1);
fclose(fs1);
fs2=fopen(file2,"w");
printf("\n enter some text into file2 here and press cntrl D or Z to stop :\n");
while((ch=getchar())!=EOF)
fputc(ch,fs2);
fclose(fs2);
fs1 =fopen(file1,"r");
fs2= fopen(file2,"r");
if( fs1 == NULL || fs2 == NULL )
{
perror("Error "); exit(1);
}
ft = fopen(file3,"w");
while( ( ch = fgetc(fs1) ) != EOF )
fputc(ch,ft);
while( ( ch = fgetc(fs2) ) != EOF )
fputc(ch,ft);
printf("Two files were merged to %s file successfully.\n",file3);
fcloseall();
ft = fopen(file3,"r");
printf("\n the merged file contents are:");
while( ( ch = fgetc(fs1) ) != EOF )
putchar(ch);
fclose(f t);
return 0;
}
```



Write a C program to append the contents of a file ?

Program:

```
#include<stdio.h>
void main()
{
FILE *fp1;
char ch;
fp1=fopen("sunday.txt","w");
printf("\n Enter some Text into file :\n");
while((ch=getchar())!='.')
fputc(ch,fp1);
fclose(fp1);
fp1=fopen("sunday.txt","a+"); //to append
printf("\n Enter some MORE Text into file:\n");
while((ch=getchar())!='.')
fputc(ch,fp1);
rewind(fp1);
printf("\n The complete Text in file is:\n");
while((ch=fgetc(fp1))!=EOF)
putchar(ch);
fclose(fp1);
}
```

Construct a c program to display the reverse the contents of a file?

Program:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
FILE*fp1;
char ch;
int x;
fp1=fopen("any1.txt","w");
printf("\n Enter some text into file:\n");
while((ch=getchar())!=EOF)
fputc(ch,fp1);
fclose(fp1);
fp1=fopen("any1.txt","r");
if(fp1==NULL)
{
printf("\n cannot open file:");
exit(1);
}
fseek(fp1,- 1L,2);
x=ftell(fp1);
printf("\n Th text in the file in reverse order is : \n");
while(x>=0)
{
ch=fgetc(fp1);
printf("%c",ch);
x--;
fseek(fp1,x,0);
}
fclose(fp1);
}
```



}

Command line arguments :

Command line argument is the parameter supplied to a program when the program is invoked. This parameter may represent a file name the program should process. For example, if we want to execute a program to copy the contents of a file named X_FILE to another one name Y_FILE then we may use a command line like

```
C:> program X_FILE Y_FILE
```

Program is the file name where the executable code of the program is stored. This eliminates the need for the program to request the user to enter the file names during execution. The „main“ function can take two arguments called `argc`, `argv` and information contained in the command line is passed onto the program to these arguments, when „main“ is called up by the system.

The variable `argv` is an argument vector and represents an array of characters pointers that point to the command line arguments.

The `argc` is an argument counter that counts the number of arguments on the command line. The size of this array is equal to the value of `argc`. In order to access the command line arguments, we must declare the „main“ function and its parameters as follows:

```
main(argc,argv) int argc;
char *argv[ ];
```

```
{
.....
}
```

Generally `argv[0]` represents the program name.

Example:- A program to copy the contents of one file into another using command line arguments.

```
#include<stdio.h>
#include<stdlib.h>
void main(int argc,char* argv[]) /* command line arguments */
{
FILE *ft,*fs; /* file pointers declaration*/
int c=0;
if(argc!=3)
Printf(“\n insufficient arguments”);
fs=fopen(argv[1],”r”);
ft=fopen(argv[2],”w”);
if (fs==NULL)
{
printf(“\nsource file opening error”); exit(1) ;
}
if (ft==NULL)
{
printf(“ target file opening error”); exit(1) ;
}
}
```



```

while(!feof(fs))
{
fputc(fgetc(fs),ft);
c++;
}
printf("\n bytes copied from %s file to %s file =%d", argv[1], argv[2], c);
c=fcloseall(); /*closing all files*/
printf("files closed=%d",c);
}

```

Simulate a c program to add two numbers using command line arguments?

Program:

```

#include<stdio.h>
int main(int argc, char *argv[])
{
int x, sum=0;
printf("\n Number of arguments are:%d",argc);
printf("\n The agruments are:");
for(x=0;x<argc;x++)
{
printf("\n agrv[%d]=%s", x,argv[x]);
if(x<2)
continue;
sum=sum+atoi(argv[x]);
}
printf("\n program name:%s",argv[0]);
printf("\n name is:%s",argv[1]);
printf("\n sum is:%d",sum);
return(0);
}

```

Write a C program to write all the members of an array of structures to a file using fwrite(). Read the array from the file and display on the screen.

```

#include <stdio.h>
struct s
{
char name[50];
int height;
};
int main()
{
struct s a[5],b[5];
FILE *fptr;
int i;
fptr=fopen("file.txt","wb");
for(i=0;i<5;++i)
{
fflush(stdin);
printf("Enter name: ");
gets(a[i].name);
printf("Enter height: ");
scanf("%d",&a[i].height);
}

```



```

fwrite(a,sizeof(a),1,fptr);
fclose(fptr);
fptr=fopen("file.txt","rb");
fread(b,sizeof(b),1,fptr);
for(i=0;i<5;++i)
{
printf("Name: %s\nHeight: %d",b[i].name,b[i].height);
}
fclose(fptr);
}

```

Output:

```

Enter name:ramu
Enter height:156
Enter name:sita
Enter height:140
Enter name:ravi
Enter height:172
Enter name:vijay
Enter height:167
Enter name:raju
Enter height:148
Name:ramu
Height:156
Name:sita
Height:140
Name:ravi
Height:172
Name:vijay
Height:167
Name:raju
Height:148

```



Enumertaion Datatype :

Another user-defined datatype is ~~enumerated data type(enum)~~

Syntax: enumidentifier{value1, value2,.....valuen};

Where identifier is user-defined datatype which is used to declare variables that can have one of the values enclosed within the braces. value1 ,value2,.....valuen all these are known as enumeration constants.

Ex:- enum identifier v1, v2,.....vn

v1=value3; v2=value1;.....

Ex:-enum day {Monday,Tuesday..... sunday}; enum day week-f,week-end

Week-f = Monday

-
-
-

(or)

enum day{Monday...Sunday} week-f, week-end;

EXAMPLE:

```
// An example program to demonstrate working of enum in C
// Enum is used to assign names to the integral constants
// enum starts its values from 0 (if not initialized) and every next value is 1 + its previous value
```

```
#include<stdio.h>
enum week{Mon, Tue, Wed, Thur=6, Fri, Sat, Sun};
int main()
{
    enum week day; // variables of type enum and can assign the values
    day = Wed;
    printf("\n Mon=%d",Mon);
    printf("\n Wed= %d",day); // assigning to an object of type enum
    printf("\n Fri=%d",Fri);
    printf("\n Sun=%d",Sun);
    return 0;
}
Mon=0
Wed= 2
Fri=7
Sun=9
```

EXAMPLE:

```
// An example program to demonstrate working of enum in C
// Enum is used to assign names to the integral constants
// enum starts its values from 0 (if not initialized) and every next value is 1 + its previous value
```

```
#include<stdio.h>
enum week{Mon=1, Tue, Wed, Thur, Fri, Sat, Sun};
int main()
{
    enum week day; // variables of type enum and can assign the values
    day = Wed;
    for(day=Mon;day<=Sun;day++) {
        printf("\n Day = %d",day);
    }
    return 0; }
```

```
Day = 1
Day = 2
Day = 3
Day = 4
Day = 5
Day = 6
Day = 7
```



Interesting facts about initialization of enum:

1. Two enum names can have same value. For example, in the following C program both 'Failed' and 'Freezed' have same value 0.

```
#include <stdio.h>
enum State {Working = 1, Failed = 0, Freezed = 0};
int main()
{
    printf("%d, %d, %d", Working, Failed, Freezed);
    return 0;
}
```

Output:

1, 0, 0

2. If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0. For example, in the following C program, sunday gets value 0, monday gets 1, and so on.

```
#include <stdio.h>
enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};
int main()
{
    enum day d = thursday;
    printf("The day number stored in d is %d", d);
    return 0;
}
```

Output:

The day number stored in d is 4

3. We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

```
#include <stdio.h>
enum day {sunday = 1, tuesday = 5, wednesday, thursday = 10, friday, saturday};
int main()
{
    printf("%d %d %d %d %d %d %d", sunday, monday, tuesday, wednesday, thursday, friday, saturday);
    return 0;
}
```

Output:

1 2 5 6 10 11 12

4. All enum constants must be unique in their scope. For example, the following program fails in compilation.

```
enum state {working, failed};
enum result {failed, passed};
int main() { return 0; }
```

Output:

Compile Error: 'failed' has a previous declaration as 'state failed'

Enum vs Macro

We can also use macros to define names constants. For example we can define 'Working' and 'Failed' using following macro.

```
#define Working 0
#define Failed 1
#define Freezed 2
```

There are multiple advantages of using enum over macro when many related named constants have integral values.

a) Enums follow scope rules. b) Enum variables are automatically assigned values. Following is simpler

```
enum state {Working, Failed, Freezed};
```

UNIT-IV

Functions: Designing structured programs, Declaring a function, Signature of a function, Parameters and return type of a function, passing parameters to functions, call by value, Passing arrays to functions, passing pointers to functions, idea of call by reference, Some C standard functions and libraries

Recursion: Simple programs, such as Finding Factorial, Fibonacci series etc., Limitations of Recursive functions

Dynamic memory allocation: Allocating and freeing memory, Allocating memory for arrays of different data types

Functions:

Why Use Functions? (Need for Functions)

The ability to divide a program into abstract, reusable pieces is what makes it possible to write large programs that actually work right.

The following reasons make functions extremely useful and practically essential:

- Enhances readability
- Easy to understand.
- Allows Code reusability
- Structured organization.
- Achieves Modular programming
- Easy identification of bugs
- Data abstraction
- Reduction of code repetition so reduces lines of code.
- Best suited for large programs or complex programs.
- It is used to check logical conditions involving multiple objects and just returns the result.

Designing structured programs:

Structured programming is a programming technique in which a larger program is divided into smaller subprograms to make it easy to understand, easy to implement and makes the code reusable etc.. The structured programming enables code reusability.

Code reusability is a method of writing code once and using it many times. Structured programming also makes the program easy to understand, improves the quality of the program, easy to implement and reduces time.

In C, the structured programming can be designed using **functions** concept.

Definition:

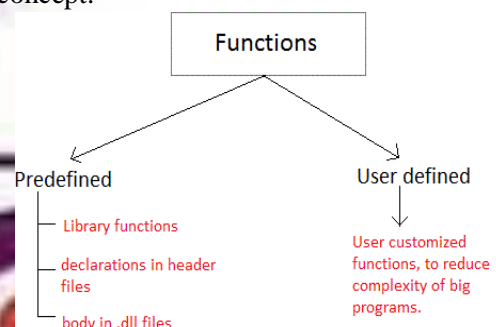
A **function** is a group of statements that together perform a task. Every C program has at least one **function**, which is main(), and all the most trivial programs can define additional functions.

Main Function in C:

In C, the "main" function is treated the same as every function, it has a return type (and in some cases accepts inputs via parameters called **command line arguments**).

In C, program execution starts from the main () function. The only difference is that the main function is "called" by the operating system when the user runs the program.

The main function can in-turn may call other functions. When main calls a function, it passes the execution control to that function. The function returns control to main when a return statement is executed or when end of function is reached.



Types of Functions:

Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C programming.

- **Standard library functions**
- **User defined**

Standard library functions

Library functions:

- The standard library functions are built-in functions in C programming.
- They handle tasks such as mathematical computations, I/O processing, string handling etc.
- These functions are defined in the headerfile.
- When we include the header file, eg: **#include**<string.h> these functions are available for use.
- **For example:** The “**printf()**” is a standard library function to send formatted output to the screen (display output on the screen).
- This function is defined in “**stdio.h**” header file.
 - Implementation details are not known to the user.
 - In order to use these functions programmers should only know the
 - name of the function
 - header file in which it is available.
 - Inputs and their type and order.
 - Return type of the function.

Example Program:

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

void main()
{
    int a,b;
    scanf("%d%d",&a,&b);
    printf("sum =%d",a+b);
    printf("Length of string = %d",strlen("INDIA"));
    printf("Root of 2 = %.4f",sqrt(2));
    printf("5 raised to Power of 3 = %d",pow(5,3));
    printf("To lower of B=%c",tolower('B'));
}
```

Output:

```
5 2
Sum=7
Length of string=5
Root of 2=1.4142
5 raised to Power of 3=125
To lower of B=b
```

Example:

```
// to find length of string

#include<string.h> // string.h is header file, strlen is name of function

int len=strlen(name);

// name is a string type argument and len is int value returned
```

Example of some C header files - library functions:

S No	Header File	Library functions (few)
1	stdio.h	printf(), scanf(), getchar(), putchar(), gets(), puts(), fopen(), fclose(),
2	stdlib.h	exit()
3	string.h	strcat(), strcpy(), strlen(), strcmp()
4	math.h	sqrt(), pow(), sin(), cos(), exp(), log(), abs(),
5	conio.h	clrscr(), getch()
6	ctype.h	isalpha(), isnum(), isalnum(), islower(), isprint(), ispunct(), isxdigit(), tolower(), toupper()

User Defined Functions in C:

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

A. FunctionDeclaration

B. FunctionDefinition

C. FunctionCall

1. Functiondeclaration

The program or a function that calls a function is referred to as the calling program or calling function. The calling program should declare any function that is to be used later in the program this is known as the function declaration or function prototype.

2. FunctionDefinition

Thefunctiondefinitionconsistsofthewholesdescriptionandcodeofafunction. thatwhatthefunctionisdoingandwhataretheinputoutputsforthat.Afunctioniscalled bysimplywritingthenameofthefunctionfollowedbytheargumentlistinsidethe

Ittells



parenthesis. Function definitions have two parts:

Function Header

The first line of code is called function Header.

```
int sum( int x, int y)
```

It has three parts

- (i) The name of the function i.e.sum
- (ii) The parameters of the function enclosed in parenthesis
- (iii) Return value type i.e.int

Function Body

Whatever is written within { } is the body of the function.

3. Function Call

In order to use the function we need to invoke it at a required place in the program.

This is known as the function call.

Signature of a function:

A function's signature includes the function's name and the number, order and type of its formal parameters.

Syntax: return-type function-name (parameters)

```
{  
declarations  
    Statements  
    return value  
}
```

Different Sub parts of Above Syntax :

return-type

1. Return Type is Type of value returned by function
2. Return Type may be "Void" if function is not going to return a value.

function-name

1. It is Unique Name that identifies function.
2. All Variable naming conventions are applicable for declaring valid function name.

parameters

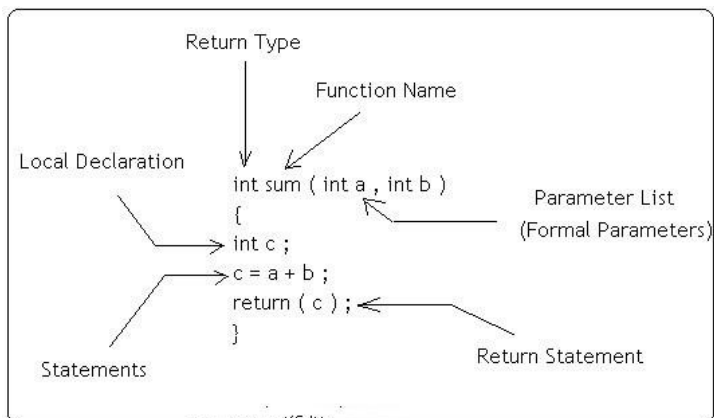
Comma-separated list of types and names of parameters

Parameter injects external values into function on which function is going to operate.

Parameter field is optional.

If no parameter is passed then no need to write this fieldvalue

1. It is value returned by function up on termination.
2. Function will not return a value if return-type isvoid.



- It contain **Executable Code**(Executable statements)
- First Line is called as **Function Header**.
- Function**Header**should be identical to function Prototype with the exception ofsemicolon

Terminology associated with functions:

Parameter:The term *parameter* refers to any declaration within the parentheses following the function name in a function declaration or definition.

Example:`void sum (int a, int b); // a,b are parameters.`

Argument:The term *argument* refers to any expression within the parentheses of a function call.

Example: `sum(num1, num2); //Call Function Sum With Two Parameters`

Calling function: The function which is making a call to any function is called calling function

Example: Calling function main()

Called function: The function which is being executed due to function call is known as called function.

Example: // sum is a called function by main function

```
int sum(num1, num2)
{
return(num1+num2);
}
```

Categories of user defined functions in C:

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

- i) **Functions with no arguments and no return values.**
- ii) **Functions with arguments and no return values.**
- iii) **Functions with arguments and return values.**
- iv) **Functions with no arguments and return values.**

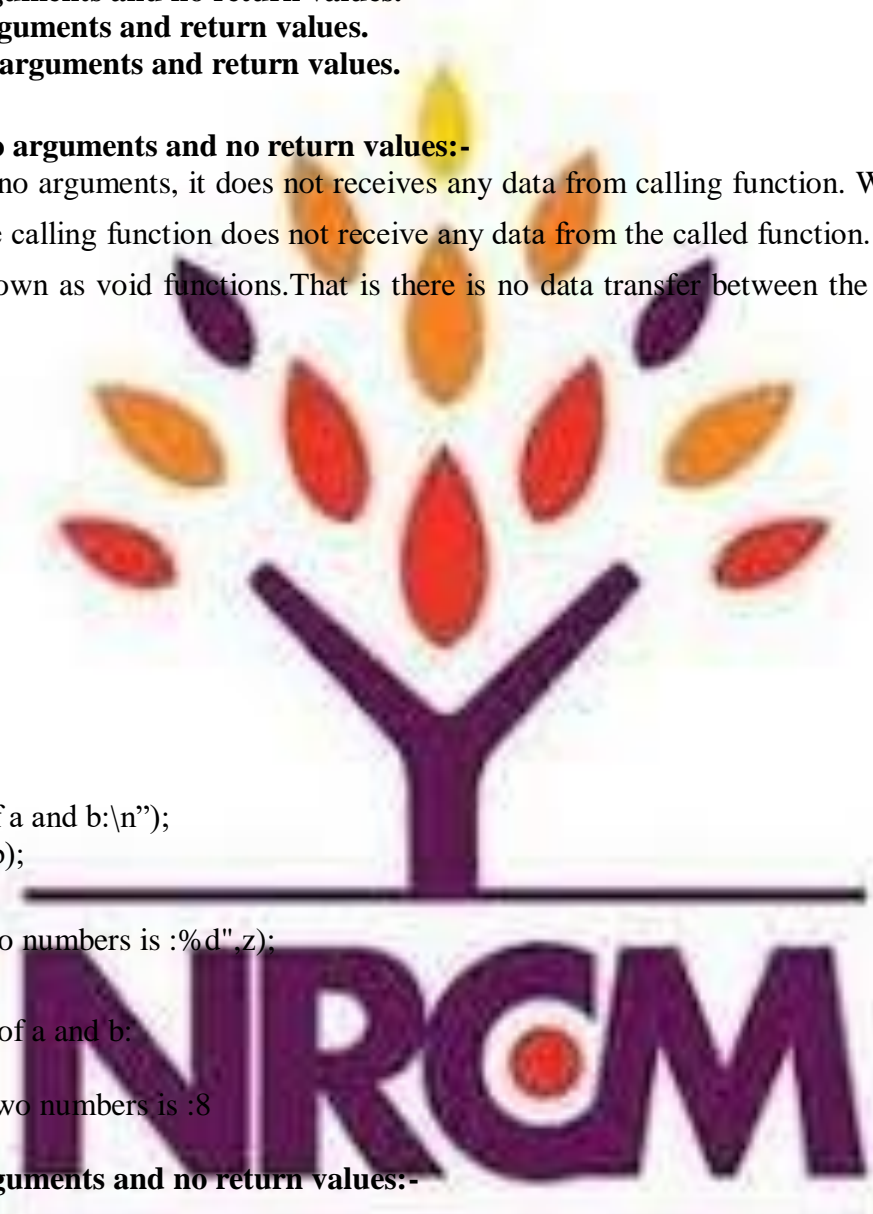
(i) Functions with no arguments and no return values:-

When a function has no arguments, it does not receives any data from calling function. When a function does not return a value, the calling function does not receive any data from the called function. Functions that don't return a value are known as void functions. That is there is no data transfer between the calling function and the called function.

Example

```
#include<stdio.h>
void add();
void main()
{
    add();
}
void add ()
{
    int z,a,b ;
    printf("enter values of a and b:\n");
    scanf("%d%d",&a,&b);
    z=x+y;
    printf ("The sum of two numbers is :%d",z);
}
```

Output : enter values of a and b:
 3 5
 The sum of two numbers is :8



(ii) Functions with arguments and no return values:-

When a function has arguments data is transferred from calling function to called function. The called function receives data from calling function and does not send back any values to calling function. Because it doesn't have return value.

Example

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



```
#include <conio.h>
void add(int,int);
void main()
{
int a, b;
printf("enter values of a and b:\n");
scanf("%d%d",&a,&b);
add(a,b);
}
void add (int x, int y)
{
int z ; z=x+y;
printf ("The sum of two numbers is :%d",z);
}
```

Output : enter values of a and b :
3 5
The sum of two numbers is : 8

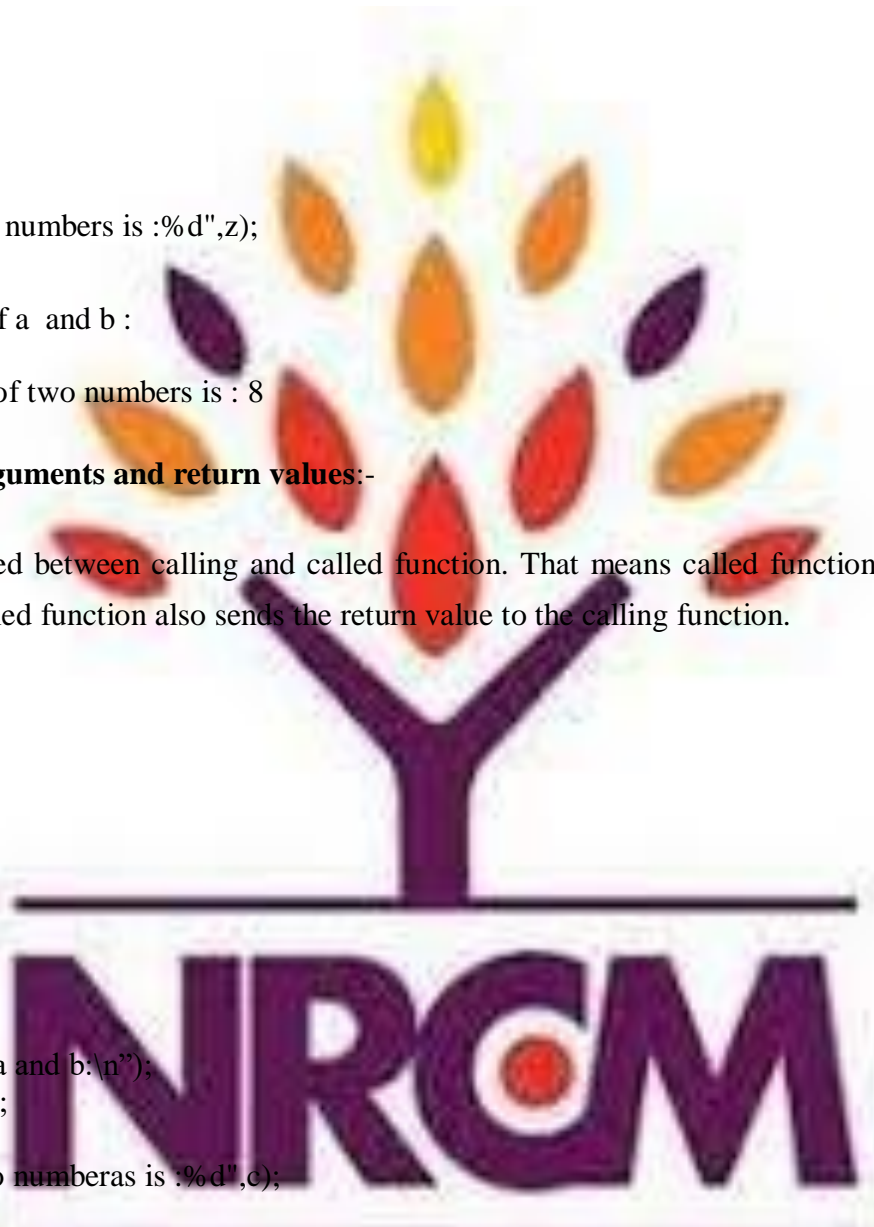
(iii) Functions with arguments and return values:-

In this data is transferred between calling and called function. That means called function receives data from calling function and called function also sends the return value to the calling function.

Example

```
#include<stdio.h>
#include <conio.h>
int add(int, int);
main()
{
int a,b,c;
printf("enter values of a and b:\n");
scanf("%d%d",&a,&b);
c=add(a,b);
printf ("The sum of two numberas is :%d",c);
}
int add (int x, int y)
{
int z; z=x+y;
return z;
}
```

output : enter values of a and b:



3 5

The sum of two numbers is :8

(iv) Function with no arguments and return type:-

When function has no arguments data cannot be transferred to called function. But the called function can send some return value to the calling function.

Example

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

```
int add();
```

```
void main()
```

```
{
```

```
int c;
```

```
c=add();
```

```
printf("The sum of two numbers is :%d",c);
```

```
}
```

```
int add ()
```

```
{
```

```
int x,y,z;
```

```
printf("enter values of a and b:\n");
```

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

```
z=x+y;
```

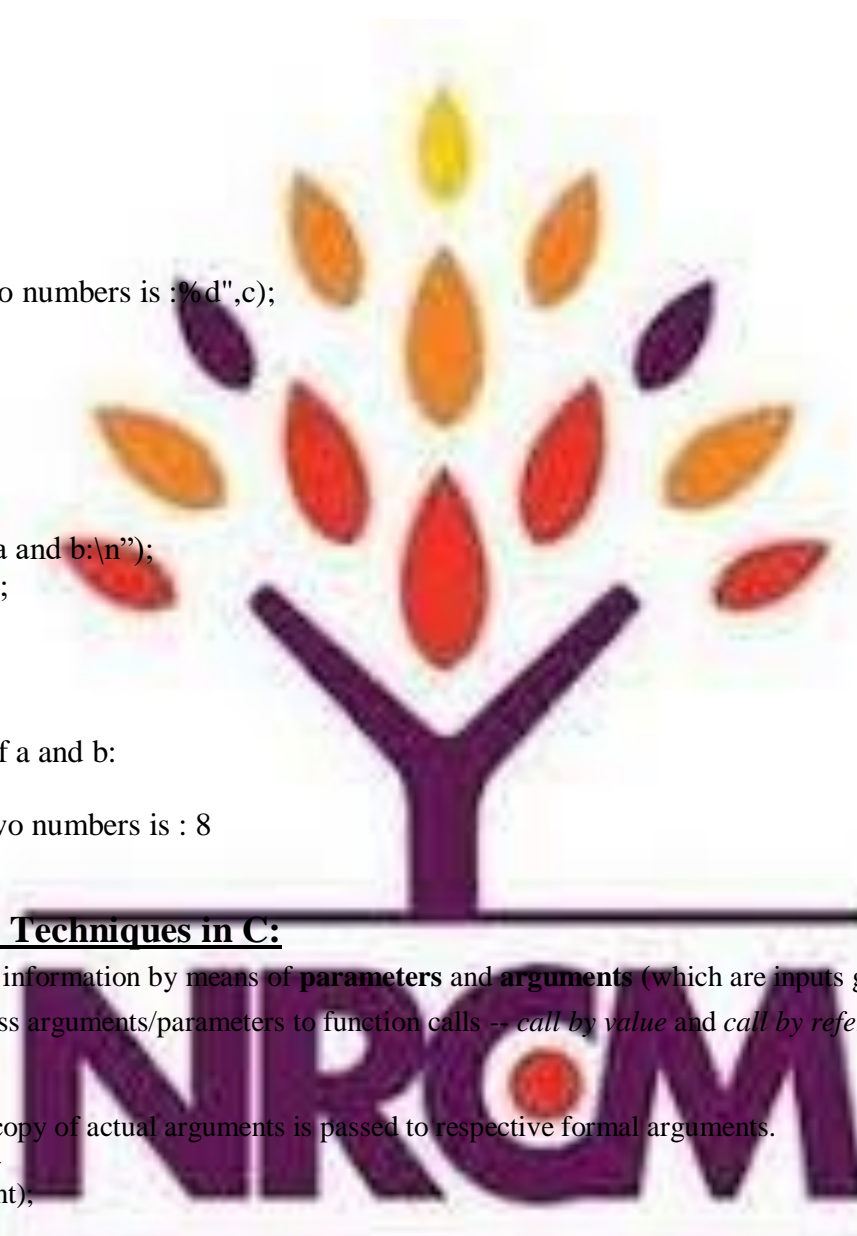
```
return z;
```

```
}
```

Output: enter values of a and b:

3 5

The sum of two numbers is : 8



Parameter Passing Techniques in C:

In C functions exchange information by means of **parameters** and **arguments** (which are inputs given to a function).

There are two ways to pass arguments/parameters to function calls -- *call by value* and *call by reference*.

Call by value:

In call by value a copy of actual arguments is passed to respective formal arguments.

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

```
void swap(int, int);
```

```
int main ()
```

```
{
```

```
int x, y;
```

```
printf("Enter the value of x and y\n");
```

```
scanf("%d%d",&x,&y);
```

```
printf("Before Swapping\nx = %d\ny = %d\n", x, y);
```

```
swap(x, y);
```

```
printf("After Swapping\nx = %d\ny = %d\n", x, y);
```

```
return 0;
```

```

}
void swap(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
    printf("Values of a and b is %d%d\n",a,b);
}

```

Output:

```

Enter the value of x and y 2
3
Before Swapping
x =2
y =3
Values of a and b is 3 2
After Swapping
x =2
y =3

```

Call by reference:

In call by reference the location (address) of actual arguments is passed to formal arguments, hence any change made to formal arguments will also reflect in actual arguments.

```

#include <stdio.h>
void swap(int*, int*);
int main()
{
    int x,y;
    printf("Enter the value of x and y\n");
    scanf("%d%d",&x,&y);
    printf("Before Swapping\nx = %d\ny = %d\n", x, y);
    swap(&x, &y);
    printf("After Swapping\nx = %d\ny = %d\n", x, y); return 0;
}
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

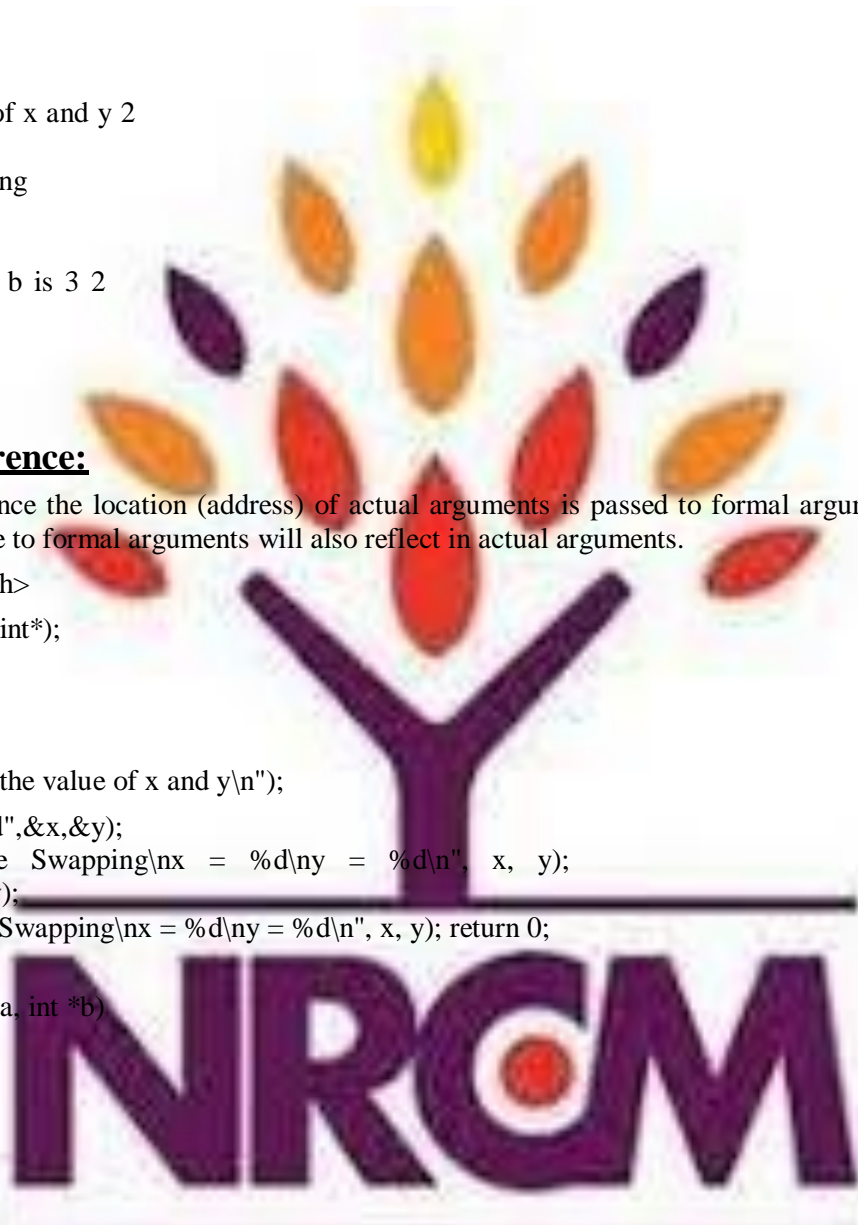
```

Output:

```

Enter the value of x and y 2
3
Before Swapping
x =2
y =3
Values of a and b is 3 2
After Swapping

```



x =3
y =2

Call by Value vs Call by Reference:

S NO	CALL BY VALUE	CALL BY REFERENCE
1	It is also known as pass by value	It is also known as pass by reference.
2	Values are passed as inputs.	Addresses are passed as inputs.
1	The actual arguments are variables, constants or expressions.	The actual arguments are addresses of variables.
4	The formal arguments are also variables.	The formal arguments are pointers.
5	Changes made in the called function are not reflected back in the calling function.	Changes made in the called function are reflected back in the calling function.
6	The calling and the called function maintains separate copies of data.	The calling and the called function maintains or shares a common location.
7	A copy of contents is maintained by calling function	The called function shares common data.
8	Example of function proto type: void Swap(int a, int b);	Example of function proto type: void Swap(int *p, int *q);
9	Example of function call: Swap(a,b);	Example of function call: Swap(&a,&b);

Parameters vs Arguments in C:

The term **parameter** (sometimes called formal **parameter**) is often used to refer to the variable as found in the function definition, while **argument** (sometimes called actual **parameter**) refers to the actual input supplied at function call.

Actual parameters vs Formal parameters:

S NO	Actual parameters	Formal parameters
1	These are the variable found in the function call.	These are the variable found in the function definition or declaration.

2	They can be variables, constants or expressions or another function calls.	They can only be variables
3	Example: <code>func1(12, 23); // constants</code> <code>unc1(a, b); // variables</code> <code>func1(a + b, b + a); //expression</code>	Example: <code>Void func1(int x, int y); //always variables</code>
4	They are the original source of information	They get the values after function call occurs.
5	They are supplied by the caller or programmer.	They are initialized with the values of actual arguments.

Note:

1. Order, number, and type of the actual arguments in the function call must match with formal arguments of the function (except for functions with variable list of arguments).
2. If there is type mismatch between actual and formal arguments then the compiler will try to convert the type of actual arguments to formal arguments if it is legal, otherwise a garbage value will be passed to the formal argument.
3. When actual arguments are less than formal arguments, the garbage value is supplied to the formal arguments.
4. Changes made in the formal argument do not affect the actual arguments.

Passing arrays to functions (Arrays with functions)

To process arrays in a large program, we need to pass them to functions. We can pass arrays in two ways:

1. passing individual elements
2. passing the whole array.

Passing Individual Elements:

One-dimensional Arrays:

We can pass individual elements by either passing their data values or by passing their addresses. We pass data values i.e; individual array elements just like we pass any data value .As long as the array element type matches the function parameter type, it can be passed. The called function cannot tell whether the value it receives comes from an array, a variable or an expression.

Program using call by value

```
void func1( int );
void main()
```

```
{
int a[5]={ 1,2,3,4,5};
func1(a[3]);
}
void func1( int x)
{
printf(“%d”,x+100);
}
```

Two-dimensional Arrays:

The individual elements of a 2-D array can be passed in the same way as the 1-D array. We can pass 2-D array elements either by value or by address.

Ex: Program using call by value

```
void fun1(int );
void main()
{
int a[2][2]={1,2,3,4};
fun1(a[0][1]);
}
void fun1(int x)
{
printf(“%d”,x+10);
}
```

2. Passing whole array (idea of call by reference)

As array name is a constant pointer pointing to first element of array(storing the base address), Passing array name to a function itself indicates, that we are passing address of first element of array .

One-dimensional array(passing array name as base address to a function)

To pass the whole array we simply use the array name as the actual parameter. In the called function, we declare that the corresponding formal parameter is an array. We do not need to specify the number of elements.

Program:

```
void fun1(int x[],int);
void main()
{
int a[5]={ 1,2,3,4,5};
fun1(a,5);
}
void fun1( int x[],int n)
{
int i, sum=0;
```

```
for(i=0;i<n;i++)
sum=sum+x[i];
printf("\nSum of the elements in an array is : %d ",sum);
}
```

Two-dimensional array:

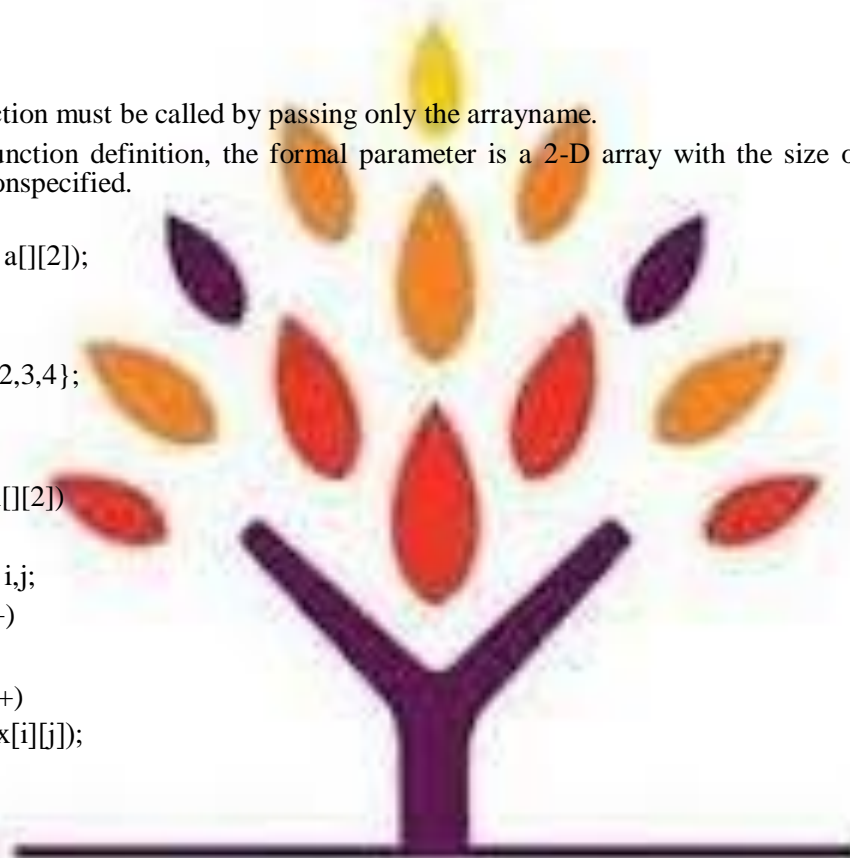
When we pass a 2-D array to a function, we use the array name as the actual parameter. The formal parameter in the called function header, however must indicate that the array has two dimensions.

Rules:

1. The function must be called by passing only the array name.
2. In the function definition, the formal parameter is a 2-D array with the size of the second dimension specified.

Program:

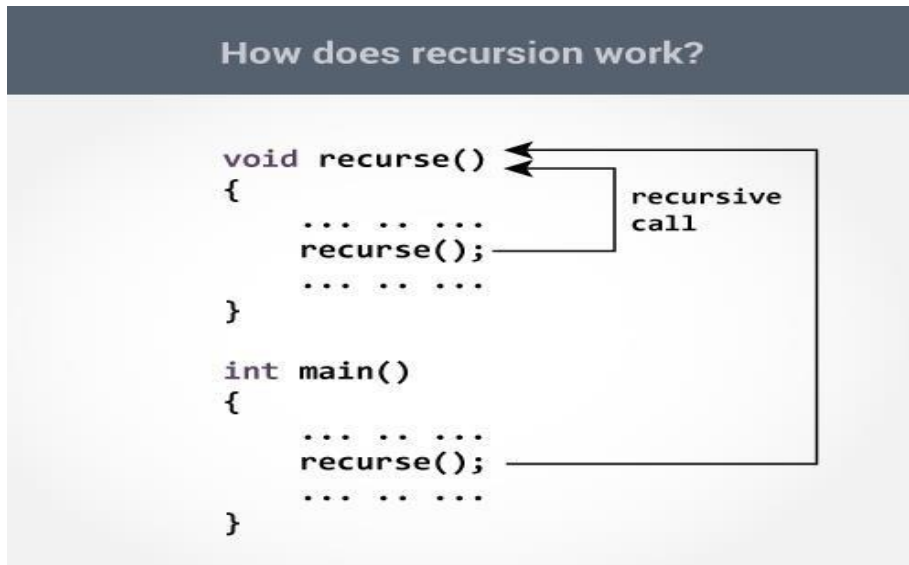
```
void fun1(int a[][2]);
void main()
{
int a[2][2]={1,2,3,4};
fun1(a);
}
void fun1(int x[][2])
{
int i,j;
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
printf("%3d",x[i][j]);
printf("\n");
}
}
```



NRCM

Recursion :

- Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a **recursive call** of the function.
- The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.
- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.



The recursion continues until some condition is met to prevent it.

To prevent infinite recursion, **if...else** statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes immediately after the recursive call inside the function.

The main **advantage** of recursive functions is that we can use them to create clearer and simpler versions of several programs.

Example1: Factorial of a Number Using Recursion

The factorial of a positive number n is given by:
factorial of n (n!)=1*2*3*4.....n

The factorial of a negative number doesn't exist. And the factorial of 0 is 1.

```

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



```
    return 1;  
}
```

Output

Enter a positive integer: 6
Factorial of 6 = 720

Example 2:C program to generate Fibonacci series using recursive functions.

```
#include<stdio.h>  
void fibo_rec(int n, int f1, int f2); void  
main()  
{  
int f1=0,f2=1,n;  
printf("Enter any number to print the fibonacci series: ");  
scanf("%d",&n);  
if(n<=0)  
printf("Enter a valid number n");  
else if(n==1)  
printf("fibonacci series is: %d",f1);  
else if(n==2)
```



```

printf("fibonacci series is: %d %d",f1,f2);
else {
printf("fibonacci series is: %d %d ",f1,f2);
fibonacci_rec(n-2,f1,f2);
printf("\n\n");
}
}
void fibonacci_rec(int n, int f1, int f2)
{
int    f3;
if(n==0)
return;
f3=f1+f2;
printf("%3d    ",f3);
fibonacci_rec(n-1,f2,f3);
}

```

Difference between Recursion and Non recursion(Iteration)

Both recursion and iteration are used for executing some instructions repeatedly until some condition is true. A same problem can be solved with recursion as well as iteration but still there are several differences in their working and performance that I have mentioned below.

	Recursion	Iteration
definition	Recursion refers to a situation where a function calls itself again and again until some base condition is reached	Iteration refers to a situation where some statements are executed again and again using loops until some condition is true
Performance	It is comparatively slower because before each function call the current state of function is stored in stack. After the return statement the previous function state is again restored from stack.	Its execution is faster because it doesn't use stack.
memory	Memory usage is more as stack is used to store the current function state.	Memory usage is less as it doesn't use stack.
Code Size	Size of code is comparatively smaller in recursion.	Iteration makes the code size bigger.

Limitations of recursion:

- Slower than its iterative solution.
- For each step we make a recursive call to a function....

- May cause stack-overflow if the recursion goes too deep to solve the problem.
- Difficult to debug and trace the values with each step of recursion.

Storage classes in C :

Variables in C differ in behavior. The behavior depends on the storage class a variable may assume. From C compilers point of view, a variable name identifies some physical location within the computer where the string of bits representing the variables value is stored. There are four storage classes in C. Storage classes specify the scope of objects .We define the storage class of an object using one of four specifiers.

- (1) Automatic storage class
- (2) Register storage class
- (3) Static storage class
- (4) External storage class

(1) Automatic Storage Class:-

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

Keyword	Auto
Storage	Memory.
Default initial value	An unpredictable value, which is often called a garbage value.
Scope	Local to the block in which the variable is defined.
Life	Till the control remains within the block in which the variable is defined

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

```
void main( )
{
auto int i, j ;
printf ( "\n%d %d", i, j ) ;
}
```

When you run this program you may get different values, since garbage values are unpredictable. So always make it a point that you initialize the automatic variables properly,

otherwise you are likely to get unexpected results. Scope and life of an automatic variable is illustrated in the following program.

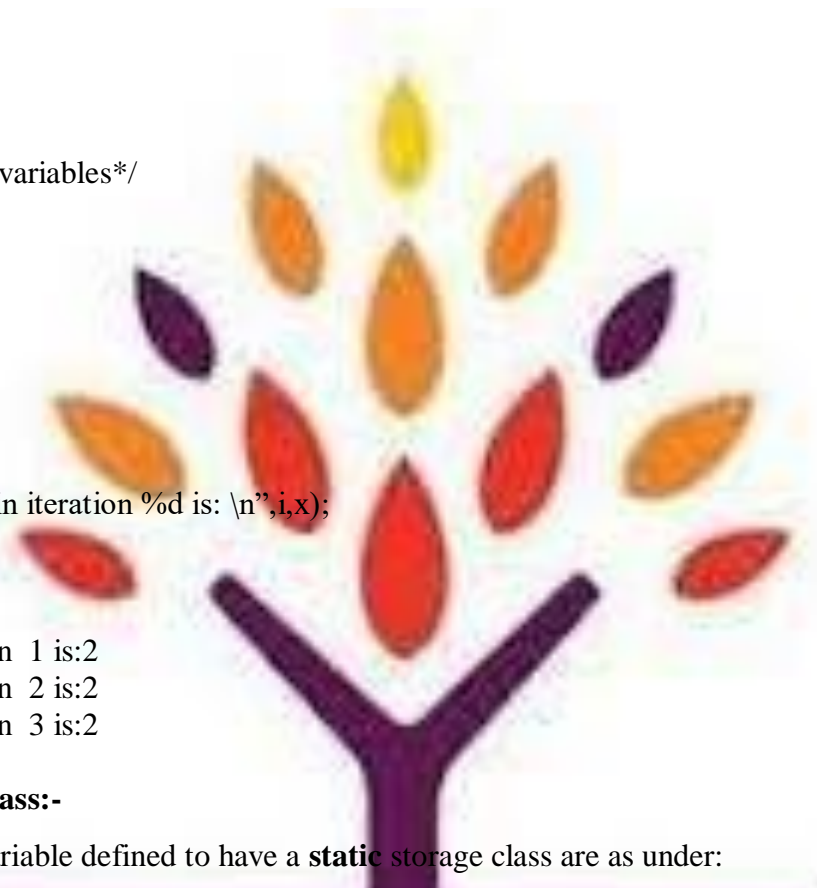
```
void main( )
{
auto int i = 1 ;
```

```
{
  auto int i = 2 ;
  {
    auto int i = 3 ;
    printf ( "\n%d ", i );
  }
  printf ( "%d ", i );
}
printf ( "%d", i );
}
```

```
/*Show the use auto variables*/
#include<stdio.h>
void main()
{
  int i,x;
  for(i=1;i<=3;i++)
  {
    x=1;
    x++;
    printf("Value of x in iteration %d is: \n",i,x);
  }
}
```

Output :

Value of x in iteration 1 is:2
 Value of x in iteration 2 is:2
 Value of x in iteration 3 is:2



(2) Static Storage Class:-

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

Keyword	Static
Storage	Memory.
Default initial value	Zero.
Scope	Local to the block in which the variable is defined.
Life	Value of the variable persists between different function calls.

Programming for problem solving[CS1103ES]

the following program demonstrates the details of static storage class:

```
/*Show the use static variables*/
#include<stdio.h>
void main()
{
    int i;
    static int x;
    for(i=1;i<=3;i++)
    {
        x=1;
        x++;
        printf("Value of x in iteration %d is: \n",i,x);
    }
}
```

Output :

```
Value of x in iteration 1 is:2
Value of x in iteration 2 is:3
Value of x in iteration 3 is:4
```

(3) Extern Storage Class:-

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

Keyword	Extern
Storage	Memory
Default initial value	Zero
Scope	Global
Life	As long as the program execution does not come to end.

External variables differ from those we have already discussed in that their scope is global, not local. External variables are declared outside all functions, yet are available to all functions that care to use them. Here is an example to illustrate this fact.

Ex:

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

4)Register Storage Class:-

The features of a variable defined to be of **register** storage class are as under:

Keyword	Register
Storage	CPU Registers
Default initial value	An unpredictable value, which is often called a garbage value.
Scope	Local to the block in which the variable is defined.
Life	Till the control remains within the block in which the variable is defined.

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

```
void main( )
{
  register int i ;
  for ( i = 1 ; i <= 10 ; i++ )
    printf ( "\n%d", i ) ;
}
```

Scope rules in C.

Scope: Scope defines the visibility of object.it defines where an object can be referenced.Scope of a variable is the part of program in which it can be used

Scope rules

The rules are as under:

1. Use **static** storage class only if you want the value of a variable to persist between different function calls.
2. Use **register** storage class for only those variables that are being used very often in a program. Reason is, there are very few CPU registers at our disposal and many of them might be busy doing something else. Make careful utilization of the scarce resources. A typical application of **register** storage class is loop counters, which get used a number of times in a program.
3. Use **extern** storage class for only those variables that are being used by almost all the functions in the program. This would avoid unnecessary passing of these variables as arguments when making a function call. Declaring all the variables as **extern** would amount to a lot of wastage of memory space because these variables would remain active throughout the life of the program.
4. If we don't have any of the express needs mentioned above, then use the **auto** storage class. In fact most of the times we end up using the **auto** variables, because often it so happens that once we have used the variables in a function we don't mind losing them.

UNIT-V

Algorithms for finding roots of a quadratic equations, finding minimum and maximum numbers of a given set, finding if a number is prime number, etc. Basic searching in an array of elements (linear and binary search techniques), Basic algorithms to sort array of elements (Bubble, Insertion and Selection sort algorithms), Basic concept of order of complexity through the example programs

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Characteristics of an Algorithm:

Not all procedures can be called an algorithm.

An algorithm should have the following characteristics –

- **Unambiguous** – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** – Algorithms must terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.

Solving a quadratic equation:

The formula to find the roots of a quadratic equation is given as follows

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

The discriminant of the quadratic equation is

$$k = (b^2 - 4ac).$$

Depending upon the nature of the discriminant, the roots can be found in different ways.

1. If the **discriminant is positive**, then there are two distinct real roots.
2. If the **discriminant is zero**, then the two roots are equal.
3. If the **discriminant is negative**, then there are two distinct complex roots.

Algorithm to find all the roots of a quadratic equation:

1. Input the value of a, b, c.
2. Calculate $k = b^2 - 4ac$
3. If $(d < 0)$
 Display "Roots are Imaginary, calculator1 = $(-b + i\sqrt{k}) / 2a$ and $r2 = (b + i\sqrt{k}) / 2a$.
 else if $(d = 0)$
 Display "Roots are Equal" and calculate $r1 = r2 = (-b / 2a)$
 else
 Display "Roots are real and calculate $r1 = -b + \sqrt{k} / 2a$ and $r2 = -b - \sqrt{k} / 2a$
4. Print r1 and r2.
5. End the algorithm

Prime number is a number which is exactly divisible by one and itself only.

Algorithm:

- Step 1: start
- Step 2: read m and n
- Step 3: initialize i = m,
- Step 4: if $i \leq n$ goto step 5
- If not goto step 10
- Step 5: initialize j = 1, c=0
- Step 6: if $j \leq i$ do as the follow.
If not goto step 7
- i) if $i \% j == 0$ increment c
- ii) increment j
- iii) goto Step 6
- Step 7: if $c == 2$ print i
- Step 8: increment i
- Step 9: goto step 4
- Step 10: stop

Write a C Program to generate prime number series between m and n, where m and n are integers.

Ans:

Program:

```
#include<stdio.h> void main()
{
int m,n, i, j, count; printf("Prime no.series\n");
printf("Enter m and n values\n"); scanf("%d%d",&m, &n);
printf("The prime numbers between %d to %d\n",m,n); for(i = m; i <= n; i++)
{
count = 0;
for(j = 1; j <=i; j++) if(i % j == 0)
{
count++;
}
```



```
}  
if(count == 2)  
{  
printf("%d\t", i);  
}  
}  
}
```

Input & Output:

Prime no. series Enter m and n Values

2 10

The prime numbers between 2 to 10 2 3 5 7

Algorithm and program to find the minimum and maximum number in a given set:

Problem Description: Given an array A[] of size n, you need to find the maximum and minimum element present in the array. Your algorithm should make the minimum number of comparisons.

For Example:

Input: A[] = { 4, 2, 0, 8, 20, 9, 2 }

Output: Maximum: 20, Minimum: 0

Searching linearly: Increment the loop by 1

We initialize both minimum and maximum element to the first element and then traverse the array, comparing each element and update minimum and maximum whenever necessary.

Pseudo-Code:

```
int[] getMinMax(int A[], int n)  
{  
    int max = A[0]  
    int min = A[0]  
    for ( i = 1 to n-1 )  
    {  
        if ( A[i] > max )  
            max = A[i]  
        else if ( A[i] < min )  
            min = A[i]  
    }  
    // By convention, let ans[0] = maximum and ans[1] = minimum  
    int ans[2] = {max, min};  
    return ans  
}
```

```
}
```

Program:


```
#include <stdio.h>
#define MAX_SIZE 100 // Maximum array size
int main()
{
    int arr[MAX_SIZE];
    int i, max, min, size;
    printf("Enter size of the array: "); /* Input size of the array */
    scanf("%d", &size);
```



```
printf("Enter elements in the array: "); /* Input array elements */  
for(i=0; i<size; i++)  
{
```



```
scanf("%d", &arr[i]);
}
max = arr[0]; /* Assume first element as maximum and minimum */
min = arr[0];
/* Find maximum and minimum in all array elements.*/
for (i=1; i<size; i++)
{
    if(arr[i] > max) /* If current element is greater than max */
    {
        max = arr[i];
    }
    if(arr[i] < min) /* If current element is smaller than min */
    {
        min = arr[i];
    }
}
/* Print maximum and minimum element */
printf("Maximum element = %d\n", max);
printf("Minimum element = %d", min);
return 0;
}
```



SEARCHING:

Searching is a technique of finding an element from the given data list or set of the elements like an array, list, or trees. It is a technique to find out an element in a sorted or unsorted list. For example, consider an array of 10 elements. These data elements are stored in successive memory locations. We need to search an element from the array. In the searching operation, assume a particular element n is to be searched. The element n is compared with all the elements in a list starting from the first element of an array till the last element. In other words, the process of searching is continued till the element is found or list is completely exhausted. When the exact match is found then the search process is terminated. In case, no such element exists in the array, the process of searching should be abandoned.

LINEAR SEARCH:

Linear search technique is also known as sequential search technique. The linear search is a method of searching an element in a list in sequence. In this method, the array is searched for the required element from the beginning of the list/array or from the last element to first element of array and continues until the item is found or the entire list/array has been searched.

Algorithm:

- Step1: set-up a flag to indicate “element not found”
- Step2: Take the first element in the list
- Step3: If the element in the list is equal to the desired element
- Set flag to “element found”
 - Display the message “element found in the list”
 - Go to step6
- Step4: If it is not the end of list,
- Take the next element in the list
 - Go to step3
- Step5: If the flag is “element not found”
- Display the message “element not found”
- Step6: End of the Algorithm

Advantages:

1. It is simple and conventional method of searching data. The linear or sequential name implies that the items are stored in a systematic manner.
2. The elements in the list can be in any order, i.e. The linear search can be applied on sorted or unsorted linear data structure.

Disadvantage:

1. This method is insufficient when large number of elements is present in list.
2. It consumes more time and reduces the retrieval rate of the system.

Time complexity: $O(n)$

Linear Search Example:

Consider-

- We are given the following linear array.
- Element 15 has to be searched in it using Linear Search Algorithm.

92	87	53	10	15	23	67
0	1	2	3	4	5	6

Linear Search Example

Now,

- Linear Search algorithm compares element 15 with all the elements of the array one by one.
- It continues searching until either the element 15 is found or all the elements are searched.

Linear Search Algorithm works in the following steps-

Step-01:

- It compares element 15 with the 1st element 92.
- Since $15 \neq 92$, so required element is not found.
- So, it moves to the next element.

Step-02:

- It compares element 15 with the 2nd element 87.
- Since $15 \neq 87$, so required element is not found.
- So, it moves to the next element.

Step-03:

- It compares element 15 with the 3rd element 53.
- Since $15 \neq 53$, so required element is not found.
- So, it moves to the next element.

Step-04:

- It compares element 15 with the 4th element 10.
- Since $15 \neq 10$, so required element is not found.
- So, it moves to the next element.

Step-05:

- It compares element 15 with the 5th element 15.
- Since $15 = 15$, so required element is found.
- Now, it stops the comparison and returns index 4 at which element 15 is present.

Write a C Program to implement Linear Search Algorithm?

```
#include <stdio.h>
int linear_search(int[],int,int);
int main()
{
    int array[100], search, c, n, position;
    printf("Input number of elements in array\n");
    scanf("%d", &n);
    printf("Input %d numbers\n", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
```

```
printf("Input a number to search\n");
scanf("%d", &search);

position = linear_search(array, n, search);

if (position == -1)
    printf("%d isn't present in the array.\n", search);
else
    printf("%d is present at location %d.\n", search, position+1);

return 0;
}
int linear_search(int a[], int n, int find)
{
    int c;

    for (c = 0 ;c < n ; c++)
    {
        if (a[c] == find)
            return c;
    }
    return -1;
}
```

The time required to search an element using the algorithm depends on the size of the list. In the best case, it's present at the beginning of the list, in the worst-case, element is present at the end. Its time complexity is $O(n)$.

BINARY SEARCH:

Binary search is quicker than the linear search. However, it cannot be applied on unsorted data structure. The binary search is based on the approach **divide-and-conquer**. The binary search starts by testing the data in the middle element of the array. This determines target is whether in the first half or second half. If target is in first half, we do not need to check thesecond half and if it is in second half noneed to check in first half. Similarly werepeat this process until we find target in the list or not found from the list. Here we need 3 variables to identify first, last and middle elements.

To implement binary search method, the elements must be in sorted order. Search is performed as follows:

- The key is compared with item in the middle position of an array
- If the key matches with item, return it and stop
- If the key is less than mid positioned item, then the item to be found must be in first half of array, otherwise it must be in second half of array.
- Repeat the procedure for lower (or upper half) of array until the element is found.

RecursiveAlgorithm: int binary_search(int a[], int beg, int end, int item)

```
{  
  Set beg = 0  
  Set end = n-1  
  while ( (beg <= end) and (a[mid] ≠ item) )  
  {  
    Set mid = (beg + end) / 2  
    if (item < a[mid])  
      Set end = mid - 1  
    Else if(item >a[mid])  
      Set beg = mid + 1  
  }  
  else
```




```
        {  
            Set loc=mid;  
        }  
    }  
    Set loc = -1  
}
```

Explanation

Binary Search Algorithm searches an element by comparing it with the middle most element of the array.

Then, following three cases are possible-

Case-01

If the element being searched is found to be the middle most element, its index is returned.

Case-02

If the element being searched is found to be greater than the middle most element, then its search is further continued in the right sub array of the middle most element.

Case-03

If the element being searched is found to be smaller than the middle most element, then its search is further continued in the left sub array of the middle most element.

This iteration keeps on repeating on the sub arrays until the desired element is found or size of the sub array reduces to zero.

Binary Search Example-

Consider-

- We are given the following sorted linear array.
- Element 15 has to be searched in it using Binary Search Algorithm.

3	10	15	20	35	40	60
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

Binary Search Example

Binary Search Algorithm works in the following steps-

Step-01:

- To begin with, we take beg=0 and end=6.
- We compute location of the middle element as-

$$\text{mid} = (\text{beg} + \text{end}) / 2$$

$$= (0 + 6) / 2$$

= 3

- Here, $a[\text{mid}] = a[3] = 20 \neq 15$ and $\text{beg} < \text{end}$.
- So, we start next iteration.

Step-02:

- Since $a[\text{mid}] = 20 > 15$, so we take $\text{end} = \text{mid} - 1 = 3 - 1 = 2$ whereas beg remains unchanged.
- We compute location of the middle element as-

$$\text{mid} = (\text{beg} + \text{end}) / 2$$

$$= (0 + 2) / 2$$

$$= 1$$

- Here, $a[\text{mid}] = a[1] = 10 \neq 15$ and $\text{beg} < \text{end}$.
- So, we start next iteration.

Step-03:

- Since $a[\text{mid}] = 10 < 15$, so we take $\text{beg} = \text{mid} + 1 = 1 + 1 = 2$ whereas end remains unchanged.
- We compute location of the middle element as-

$$\text{mid} = (\text{beg} + \text{end}) / 2$$

$$= (2 + 2) / 2$$

$$= 2$$

- Here, $a[\text{mid}] = a[2] = 15$ which matches to the element being searched.
- So, our search terminates in success and index 2 is returned.

Binary Search Algorithm Advantages-

The advantages of binary search algorithm are-

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

Binary Search Algorithm Disadvantages-

The disadvantages of binary search algorithm are-

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.(because of its random access nature)

Time Complexity Analysis-

Binary Search time complexity analysis is done below-

- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.

Thus, we have-

Time Complexity of Binary Search Algorithm is $O(\log_2 n)$.

Here, n is the number of elements in the sorted linear array.

This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.



**Cprogramforrecursive
binarysearchto find the given
element within array.**

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

```
int bsearch(int [ ],int, int, int);
```

```
void main( )
```

```
{
```

```
int a[20],pos,n,k,i,lb,ub;
```

```
clrscr( );
```

```
printf("\nEnter the n value:");
```

```
scanf("%d",&n);
```

```
printf("\nEnter elements for an array:");
```

```
for(i=0;i<n;i++)
```

```
scanf("%d",&a[i]);
```

```
printf("\nEnter the key value:");
```

```
scanf("%d",&k);
```

```
lb=0;
```

```
ub=n-1;
```

```
pos=bsearch(a,k,lb,ub);
```

```
if(pos!=-1)
```

```
printf("Search successful, element found at position %d",pos);
```

```
else
```

```
printf("Search unsuccessful, element not found");
```

```
getch( );
```

```
}
```

```
int bsearch(int a[ ], int k, int lb, int ub)
```

```
{
```

```
int mid;
```

```
while(ub>=lb)
```

```
{
```

```
mid=(lb+ub)/2;
```

```
if(k<a[mid])
```

```
ub=mid-1;
```

```
else if(k>a[mid])
```

```
lb=mid+1;
```

```
else if(k==a[mid])
```

```
return(mid);
```




```
return(bsearch(a,k,lb,ub));
}
```

```
return -1;
```

```
}
```

OUTPUT:

```
Enter'n' value      :          6
Enter elements foranarray      :      10  32   25  84   55   78
Enter the element tobesearched :          78
Search successful, Element foundat Position      :      5
```

```
/* recursive program for Linear Search*/
```

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

```
int linear(int [ ],int,int ,int); void main( )
```

```
{
```

```
int a[20],pos=-1,n,k,i; clrscr();
```

```
printf("\nEnter n value:"); scanf("%d",&n);
```

```
printf("\nEnter elements for an array:"); for(i=0;i<n;i++)
```

```
scanf("%d",&a[i]);
```

```
printf("\n Enter the element to be searched:"); scanf("%d",&k);
```

```
pos=linear(a,n,k); if(pos!=-1)
```

```
printf("\n Search successful, Element found at Position %d",pos);
```

```
else
```

```
printf("Search unsuccessful, element not found ");
```

```
}
```

```
int linear(int a[ ],int n,int low, int k)
```

```
{
```

```
if(low>= n)
```

```
return -1;
```

```
else if(k==a[low])
```

```
return low;
```

```
else
```

```
return linear(a,n,low+1,k);
```

```
}
```

```
}
```

```
return -1;
```

```
}
```

Output:-

```
Enter'n' value      :          6
Enter elements foranarray      :      10  32   22  84   55   78
Enter the element tobesearched :          55
Search successful, Element foundat Position      :      4
```

Sorting:

Sorting is the basic operation in computer science. Sorting is the process of arranging data in some given sequence or order (in increasing or decreasing order).

For example you have an array which contain 10 elements as follow;
10, 3 ,6 12, 4, 17, 5, 9

After sorting value must be;

3, 4,5, 6, 9, 10, 12, 17

Above value can be sorted by applying any sorting technique. C language have following technique to sort values;

- BubbleSort
- SelectionSort
- InsertionSort

Bubble Sort in C:

Bubble sort is a simple sorting algorithm in which each element is compared with adjacent element and swapped if their position is incorrect. It is named as bubble sort because same as like bubbles the lighter elements come up and heavier elements settledown.

Both worst case and average case complexity is O

(n^2) . **Example Program for Bubble Sort**

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

```
int main()
{
int a[50],n,i,j,temp;
printf("Enter the size of array: ");
scanf("%d",&n);
printf("Enter the array elements: ");
```

```
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[j]>a[j+1])
{
temp=a[j];
}
}
```

How Bubble Sort Works?

Programming for problem solving[CS1103ES]

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



Bubble sort starts with very first two elements, comparing them to check which one is greater.



In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –



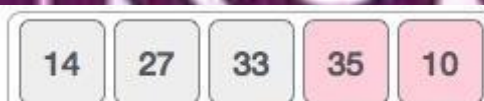
Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to the next two values, 35 and 10.



We know then that 10 is smaller 35. Hence they are not sorted.



We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –



To be precise, we are now showing how an array should look like after each iteration. After the

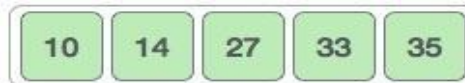
second iteration, it should look like this –



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sort learns that an array is completely sorted.



Algorithm:

```
void bubbleSort( int a[], int n )
{
    for i = 0 to n-1
    {
        for j = 0 to n-1
        {
            /* compare the adjacent elements */
            if list[j] > list[j+1])
            {
                /* swap them */
                temp=list[j]
                list[j]=list[j+1];
                list[j+1]=temp;
            }
        }
    }
}
```

C Program to implement Bubble Sort Algorithm.

The logo for NIRCM (National Institute of Remote and Continuing Medical Education) features the acronym 'NIRCM' in a bold, purple, sans-serif font. The letter 'O' is stylized with a red and white circular graphic inside it. Above the text is a stylized tree with a purple trunk and branches, and several colorful leaves in shades of orange, red, and yellow.

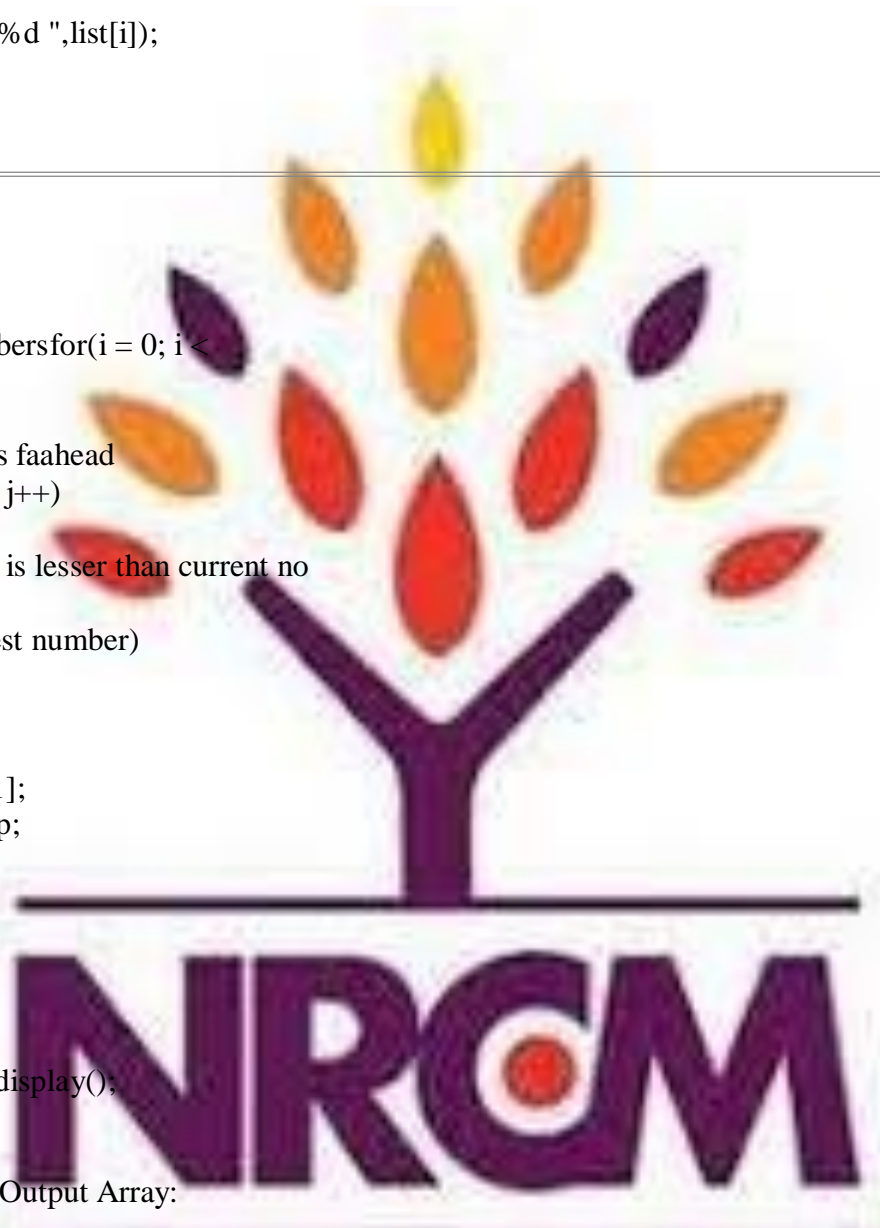
Programming for problem solving[CS1103ES]

```
#include <stdio.h>
#define MAX 10
int list[MAX] = {1,8,4,6,0,3,5,2,7,9};
void display()
{
    int i; printf("[");
    // navigate through all itemsfor(i
    = 0; i < MAX; i++)
    {
        printf("%d ",list[i]);
    }
    printf("]\n");
}
```

```
void bubbleSort()
{
    int temp;
    int i,j;
    // loop through all numbersfor(i = 0; i <
    MAX-1; i++)
    {
        // loop through numbers faahead
        for(j = 0; j < MAX-1-i; j++)
        {
            // check if next number is lesser than current no
            // swap the numbers.
            // (Bubble up the highest number)
            if(list[j] > list[j+1])
            {
                temp = list[j];
                list[j] = list[j+1];
                list[j+1] = temp;
            }
        }
    }
}

void main()
{
    printf("Input Array: ");display();
    printf("\n");

    bubbleSort(); printf("\nOutput Array:
    ");display();
}
```



SelectionSortAlgorithm:

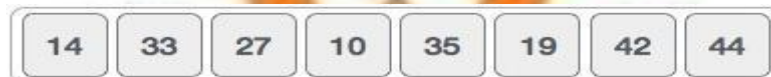
Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.



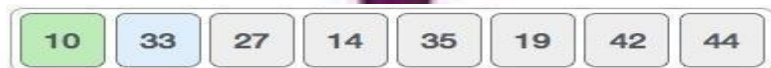
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process –



Selection sort algorithm:

```

void selection sort( int list[], int n)// list : array of items n : size of list
{
    for i = 1 to n - 1
    {
        /* set current element as minimum*/min = i
        /* check the element to be minimum */for j = i+1
        to n
        {
            if (list[j] < list[min])
            {
                min = j;
            }
        }
        /* swap the minimum element with the current element*/temp=
        list[min]
        list[min]=list[i]
        list[i]=temp;
    }
}
    
```




C program to implement selection sort Algorithm:

```
#include <stdio.h>
#define MAX 7
int Array[MAX] = {4,6,3,2,1,9,7};
void display()
{
    int i; printf("[");
    // navigate through all items
    for(i = 0; i < MAX; i++)
    {
        printf("%d ", intArray[i]);
    }
    printf("]\n");
}
void selectionSort()
{
    int indexMin, i, j, temp;
    // loop through all numbers
    for(i = 0; i < MAX-1; i++)
    {
        // set current element as minimum
        indexMin = i;
        // check the element to be minimum
        for(j = i+1; j < MAX; j++)
        {
            if(intArray[j] < intArray[indexMin])
            {
                indexMin = j;
            }
        }
        // swap the numbers
        temp = intArray[indexMin];
        intArray[indexMin] = intArray[i];
        intArray[i] = temp;
    }
}
void main()
{
    printf("Input Array: ");
    display();
    selectionSort();
    printf("Output Array: ");
    display();
}
```



Insertion Sort:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.



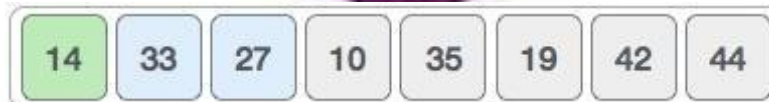
Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



Programming for problem solving[CS1103ES]

These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list

Algorithm:

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        keyarr[i];j = i - 1;

        /* Move elements of arr[0..i-1], that are greater than key, to one position ahead of
        their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```



```
}  
}
```

C Program to implement Insertion Sort Algorithm:

```
// C program for insertion sort
```

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

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

```
/* Function to sort an array using insertion sort*/
```

```
void insertionSort(int arr[], int n)
```

```
{
```

```
int i, key, j;
```

```
for (i = 1; i < n; i++)
```

```
{key = arr[i];j = i - 1;
```

```
/* Move elements of arr[0..i-1], that are greater than key, to one position ahead of  
their current position */
```

```
while (j >= 0 && arr[j] > key)
```

```
{
```

```
arr[j + 1] = arr[j];j = j - 1;
```

```
}
```

```
arr[j + 1] = key;
```

```
}
```

```
}
```

```
// A utility function to print an array of size n
```

```
void printArray(int arr[], int n)
```

```
{
```

```
int i;
```

```
for (i = 0; i < n; i++)
```

```
printf("%d ", arr[i]);
```

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

```
}
```

```
main()
```

```
{
```

```
int arr[] = { 12, 11, 13, 5, 6 };
```

```
int n = sizeof(arr) /
```

```
sizeof(arr[0]);
```

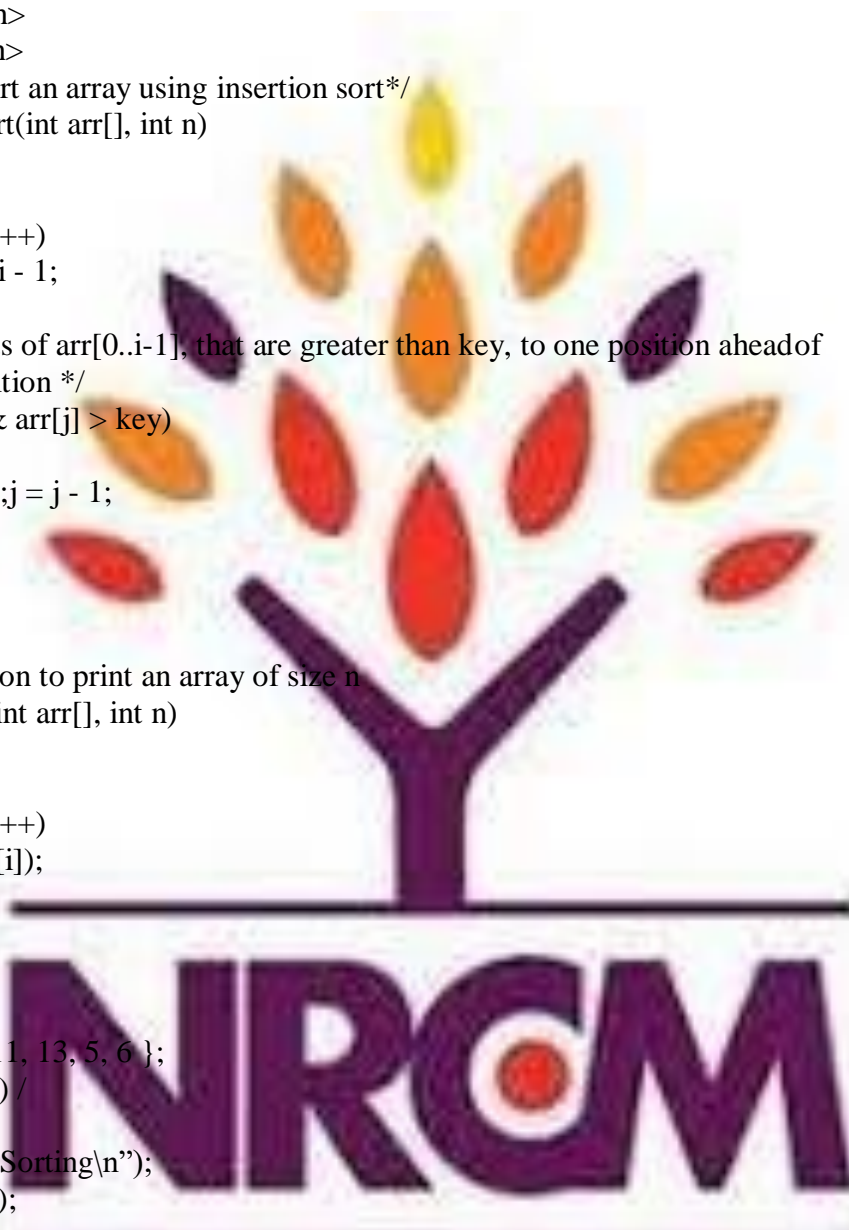
```
printf("\nBefore Sorting\n");
```

```
printArray(arr, n);
```

```
insertionSort(arr,n);
```

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

```
printArray(arr, n);
```



```
return 0;  
}
```

Complexity:

Complexity has no formal definition at all. It just defines the rate of efficiency at which a task is executed. In data structures and algorithms, there are two types of complexities that determine the efficiency of an algorithm. They are:

Space Complexity: Space complexity is the total memory consumed by the program for its execution.

Time Complexity: It is defined as the times in number instruction, in particular, is expected to execute rather than the total time is taken. Since time is a dependent phenomenon, time complexity may vary on some external factors like processor speed, the compiler used, etc.

In computer science, the time complexity of an algorithm is expressed in big O notation. Let's discuss some time complexities.

O(1): This denotes the constant time. $O(1)$ usually means that an algorithm will have constant time regardless of the input size. **Hash Maps** are perfect examples of constant time.

O(log n): This denotes logarithmic time. $O(\log n)$ means to decrease with each instance for the operations. **Binary search trees** are the best examples of logarithmic time.

O(n): This denotes linear time. $O(n)$ means that the performance is directly proportional to the input size. In simple terms, the number of inputs and the time taken to execute those inputs will be proportional or the same. Linear search in **arrays** is the best example of linear time complexity.

O(n²): This denotes quadratic time. $O(n^2)$ means that the performance is directly proportional to the square of the input taken. In simple, the time taken for execution will take square times the input size. **Nested loops** are perfect examples of quadratic time complexity.

Let's move on to the main plan and discuss the time complexities of different sorting algorithms.

Time Complexity of Bubble Sort

Programming for problem solving[CS1103ES]

Bubble sort is a simple sorting algorithm where the elements are sorted by comparing each pair of elements and switching them if an element doesn't follow the desired order of sorting. This process keeps repeating until the required order of an element is reached.

Average case time complexity: **$O(n^2)$**

Worst-case time complexity: **$O(n^2)$**

Best case time complexity: **$O(n)$**

The best case is when the given list of elements is already found sorted. This is why bubble sort is not considered good enough when the input size is quite large.

Time Complexity of Selection Sort

Selection sort works on the fundamental of in-place comparison. In this algorithm, we mainly pick up an element and move on to its correct position. This process is carried out as long as all of them are sorted in the desired order.

Average case time complexity: **$O(n^2)$**

Worst-case time complexity: **$O(n^2)$**

Best case time complexity: **$O(n^2)$**

Selection sort also suffers the same disadvantage as we saw in the bubble sort. It is inefficient to sort large data sets. It is usually preferred because of its simplicity and performance-enhancing in situations where auxiliary memory is limited.

Time Complexity of Insertion Sort

Insertion sort works on the phenomenon by taking inputs and placing them in the correct order or location. Thus, it is based on iterating over the existing elements while taking input and placing them where they are ought to be.

Best case time complexity: **$O(n)$**

Average and worst-case time complexity: **$O(n^2)$**