PROGRAMMING FOR PROBLEM SOLVING

Course Code:ACSB01

I B.Tech II Semester

BY

Mr. P Ravinder
Assistant Professor

Electrical and Electronics Engineering
Institute of Aeronautical Engineering
(Autonomous)
Dundigal, Hyderabad -500043

Running Course Outcomes

CO's	Course outcomes
CO1	Describe the concept of computer system, analyze a given problem, develop an algorithm, fundamental programming constructs, identify data representation formats, describe operators and their precedence, associativity.
CO2	Understanding branching and loop statements.
CO3	Describe the concept of homogeneous derives data types, strings and functions
CO4	Understanding pointers and heterogeneous data types.
CO5	Describe the concept of file system.

MODULE – I
INTRODUCTION

Running Course Learning Outcomes

CLOs	Course Learning Outcomes
CLO1	Describe the concept of file system.
CLO2	Analyze a given problem and develop an algorithm to solve the problem.
CLO3	Describe the fundamental programming constructs and articulate how they are used to develop a program with a desired runtime execution flow.
CLO4	Gain knowledge to identify appropriate C language constructs to write basic programs.

CLOs	Course Learning Outcome
CLO5	Identify the right data representation formats based on the requirements of the problem.
CLO6	Describe the operators, their precedence and associativity while evaluating expressions in program statements.

Computer systems

Definition:

➢A Computer is an electronic device that stores, manipulates and retrieves the data.

➢The following are the objects of computer System

1. User (A person who uses the computer)

2. Hardware

3. Software
4. Output Devices(O/P)
5. KEYBOARD

Computer system components

❖ Hardware:Hardware of a computer system can be referredas anything which we can touchand feel.

➢**Example : Keyboard and Mouse**

❖ The hardware of a computer system can beclassifieds

❖ Input Devices(I/P)-Keyboard, scanner, mouse

❖ Processing Devices(CPU)-ALU,CU,MU

❖ Output Devices(O/P) -Monitor, printer,speakersetc

❖ ALU: It performs the Arithmetic and Logical Operations such as CU: Every Operation such as storing , computing and retrieving the data should be governed by the control unit.

❖MU: The Memory unit is used for storing the data.

Memory Unit

The Memory unit is classified into two types.

They are:

1. Primary Memory
2. Secondary Memory

Primary memory: The following are the types of memories which are treated as primary

ROM: It represents Read Only Memory that stores data and instructions even when the computer is turned off. The Contents in the ROM are modified if they are written . It is used to store the BIOS information.

RAM: It represents Random Access Memory that stores data and instructions when the computer is turned on. The contents in the RAM can be modified any no. of times by instructions. It is used to store the programs under execution.

Secondary Memory: The following are the different kinds of memories Magnetic Storage: The Magnetic Storage devices store information that can be read, erased and rewritten a number of times.

Example: Floppy Disks, Hard Disks, Magnetic Tapes

Software classification:

Software is classified into two categories:
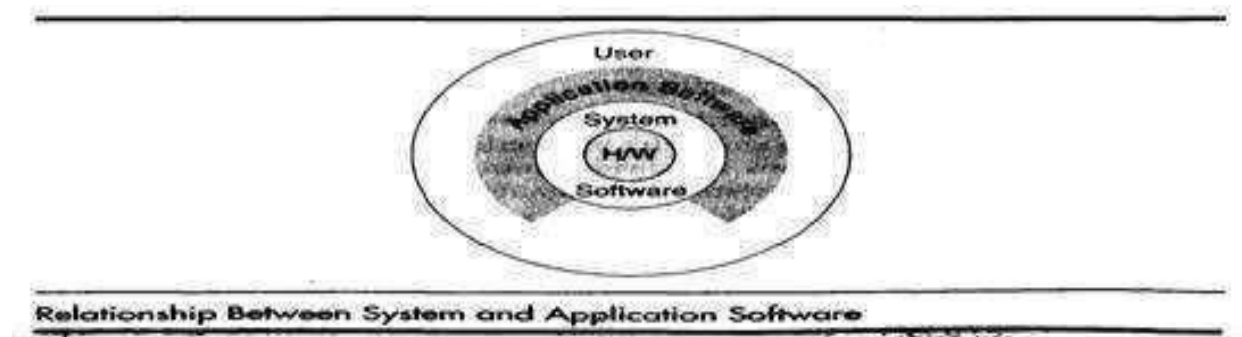
1. System Software

2. Applicationsoftware

➢ **System Software** consists of programs that manage the hardware resources of a computer and perform required information processing tasks. These programs are divided into threeclasses:

➢ **Ex: operating system, System support software , system development software**

Application software

- ➢ **Application software** is broken in to two classes: general-purpose software and application –specific software.

- ➢ **General purpose software** is purchased from a software developer and can be used for more than one application.

- ➢ **Application –specific software** can be used only for its intended purpose. A general ledger system used by accountants and a material requirements planning system used by a manufacturing organization are examples of application-specific software.

➢Relationship between system software and application software

Relationship Between System and Application Software

each circle represents an interface point .The inner core is hard ware. The user is represented by the out layer. To work with the system, the typical user uses some form of application software. The application software in turn interacts with the operating system, which is a part of the system software layer. The system software provides the direct interaction with the hard ware.

1

Computing Environments

1. The following are the different kinds of computing environments available.

2. Personal Computing Environment.

3. Time Sharing Environment.

4. Client/Server Environment.

5. Distributed Computing Environment.

> In 1971, Marcian E. Hoff, working for INTEL combined basic elements the of the central processing unit into the microprocessor. If we are using a personal computer then all the computer hardware components are tied together. This kind of computing is used to satisfy the needs of a single user, who uses the computer for the personal tasks.

Personal Computing Environment
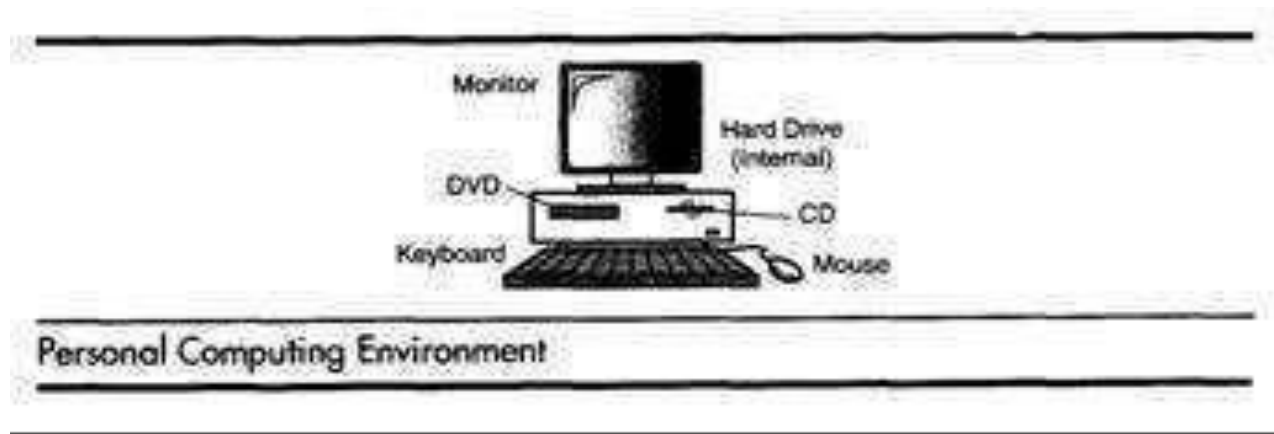
Distributed Computing

➢A distributed computing environment provides a seamless integration of computing functions between different servers and clients. A client not just a requestor for processing the information from the server. The client also has the capability to process information. All the machines Clients/Servers share the processing task.

Distributed Computing

Computer languages
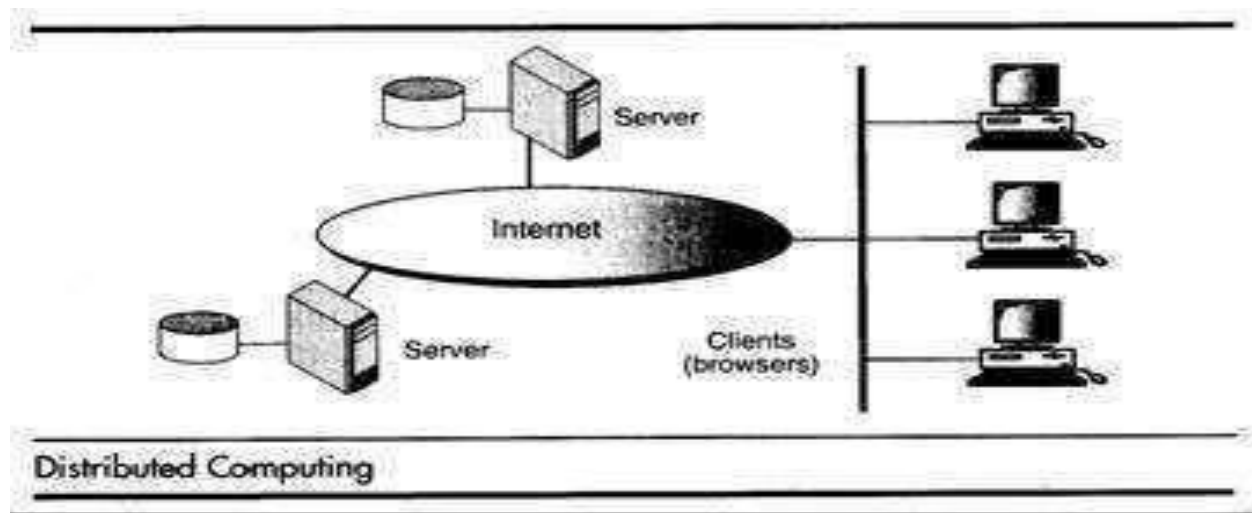
Computer languages

To write a program (tells what to do) for a computer, we must use a computer language. Over the years computer languages have evolved from machine languages to natural languages. The following is the summary of computer languages

1. 1940's   --   Machine Languages

1. 1950's   --   Symbolic Languages

1. 1960's   --   High Level Languages

Machine Language

- 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. The instructions in machine language must be in streams of 0's and 1's.

- This is also referred as binary digits.

- These are so named as the machine can directly understood the programs

Symbolic Languages (or) Assembly

➤ In the 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.

➤ These early programming languages simply mirrored the machine languages using symbols or mnemonics to represent the various language instructions.

➤ These languages were known as symbolic languages. Because a computer does not understand symbolic language it must be translated into the machine language.

High-Level Languages

➢ The symbolic languages greatly improved programming efficiency they still required programmers to concentrate on the hardware that they were using working with symbolic languages was also very tedious because each machine instruction had to be individually coded.

➢ The desire to improve programmer efficiency and to change the focus from the computer to the problems being solved led to the development of High level languages.

➢ High level languages are portable to many different computer allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer.

Language Translators

These are the programs which are used for converting the programs in one language into machine language instructions, so that they can be excuted by the computer.

✓**Compiler**: It is a program which is used to convert the high level language programs into machine language

✓**Assembler**: It is a program which is used to convert the assembly level language programs into machine language

✓**Interpreter**: It is a program, it takes one statement of a high level language program, translates it into machine language instruction.

Creating And Running Programs

➢ The procedure for turning a program written in C into machine Language.

➢ The process is presented in a straightforward, linear fashion but you shuld recognize that these steps are repeated many times during development to correct errors and make improvements to the code.

The following are the four steps in this process

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

Creating And Running Programs
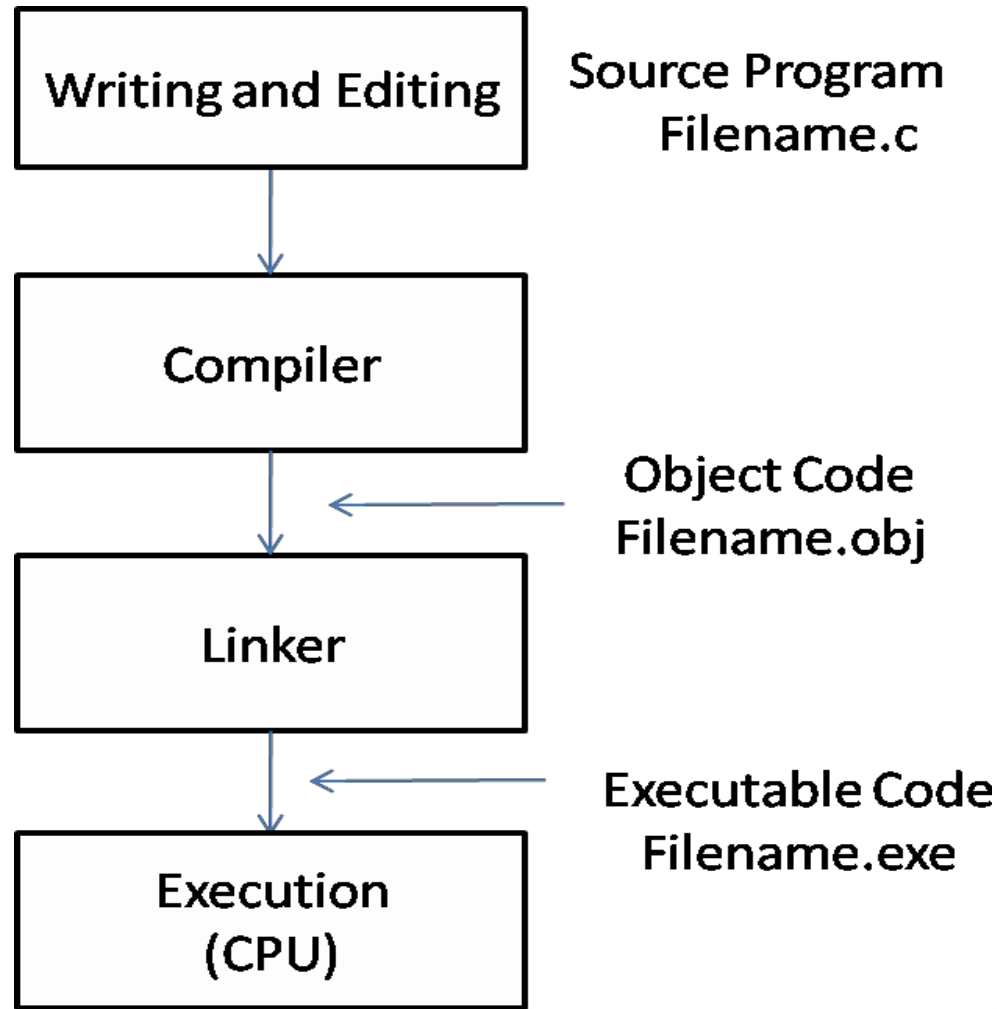
Writing and Editing Programs

1. The software used to write programs is known as a text editor.

2. A text editor helps us enter, change and store character data.

3. Once we write the program in the text editor we save it using a filename stored with an extension of .C.

4. This file is referred as source code 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 Compiler is a computer program that translates the source code written in a high- level language into the corresponding object code of the low-level language.

- This translation process is called *compilation.*

- The entire high level program is converted into the executable machine code file.

Executing Programs

- ➢ To execute a program we use an operating system command, such as run, to load the program into primary memory and execute it.

- ➢ Getting the program into memory is the function of an operating system program known as the loader.

- ➢ It locates the executable program and reads it into memory.

- ➢ When everything is loaded the program takes control and it begins execution.

Algorithm

➢ Precise step-by-step plan for a computational procedure that begins with an input value and yields an output value in a finite number of steps.

➢ It is an effective method which uses a list of well-defined instructions to complete a task, starting from a given initial state to achieve the desired end state.

➢ An algorithm is written in simple English and is not a formal document.

➢ An algorithm must: Be lucid, precise and unambiguous Give the correct solution in all cases

Properties of algorithms

Finiteness:

an algorithm terminates after a finite numbers of steps.

Definiteness:

each step in an algorithm is unambiguous. This means that the action specified by the step cannot be interpreted in multiple ways & can be performed without any confusion.

Input:

An algorithm accepts zero or more inputs.

Output:

It produces at least one output.

Effectiveness:

-It consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time.

Flowchart

- A flowchart is a type of diagram, that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows.

- This diagrammatic representation can give a step- by- step solution to a problem.

- Data is represented in the boxes, and arrows connecting them represent direction of flow of data.

- Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields .
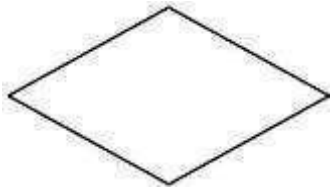
Common Flowchart Symbols

Terminator: Shows the starting   and ending points of a program
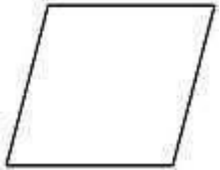
Data Input or output: Allows the user   to input data or to display the results .
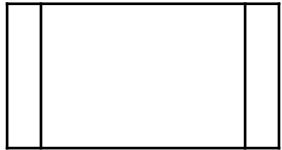
Processing: Indicates an operation performed by the computer, such as a variable Assignment or mathematical operation
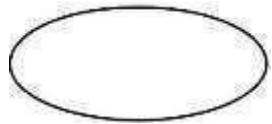
Decision: A diamond has two flow lines going   out. One is labeled as "Yes" Branch and the   other as "No" branch.
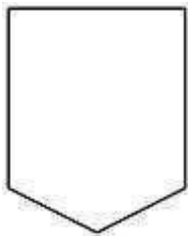
1

Common Flowchart Symbols

Predefined Process. Denotes a group of   previously defined statements.

Connector. Connectors avoid crossing flowlines, Connectors come in pairs, one with a flowline in and the other with a flowline out.

Off Page Connector: Come in Pairs, Extends Flow charts to more than a page

Flowline. Flowlines connect the flowchart symbols and show the sequence of operations during the program execution.

Examples

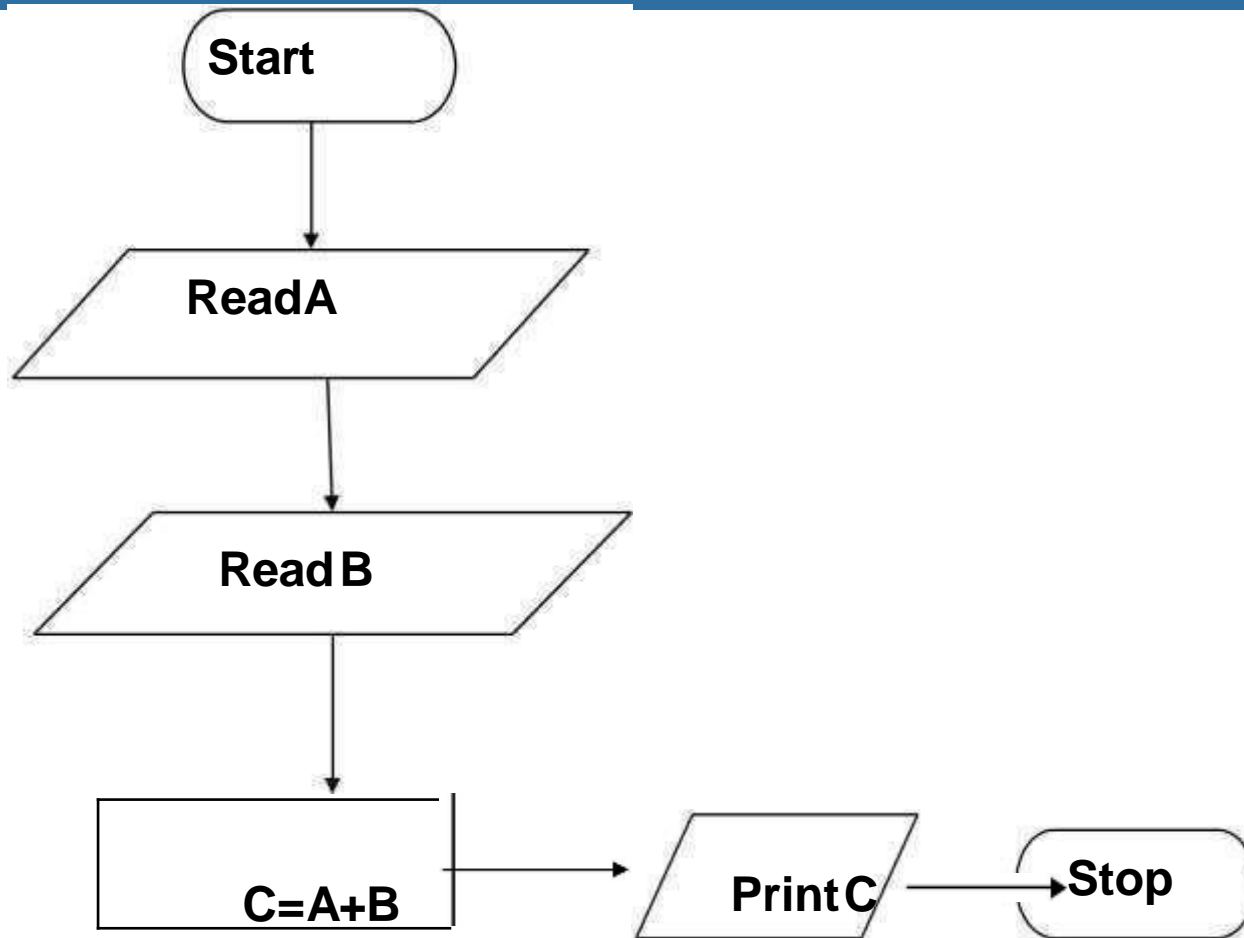Example 1: Finding the sum of two numbers.

Variables:

- A: First Number
- B: Second Number
- C: Sum (A+B)

Algorithm:

- Step 1 – Start
- Step 2 – Input A
- Step 3 – Input B
- Step 4 – Calculate C = A + B
- Step 5 – Output C
- Step 6 – Stop

Flowchart

Example 2

➢ Find the difference and the division of two numbers and display the results.

Variables:

- N1: First number
- N2: Second number
- D : Difference
- V : Division

Algorithm:

* Step 1: Start
 * Step 2: Input N1
 * Step 3: Input N2
 * Step 4: D = N1 − N2
 * Step 5: V = N1/N2
 * Step 6: Output D
 * Step 7: Output V
 * Step 8: Stop

Flowchart
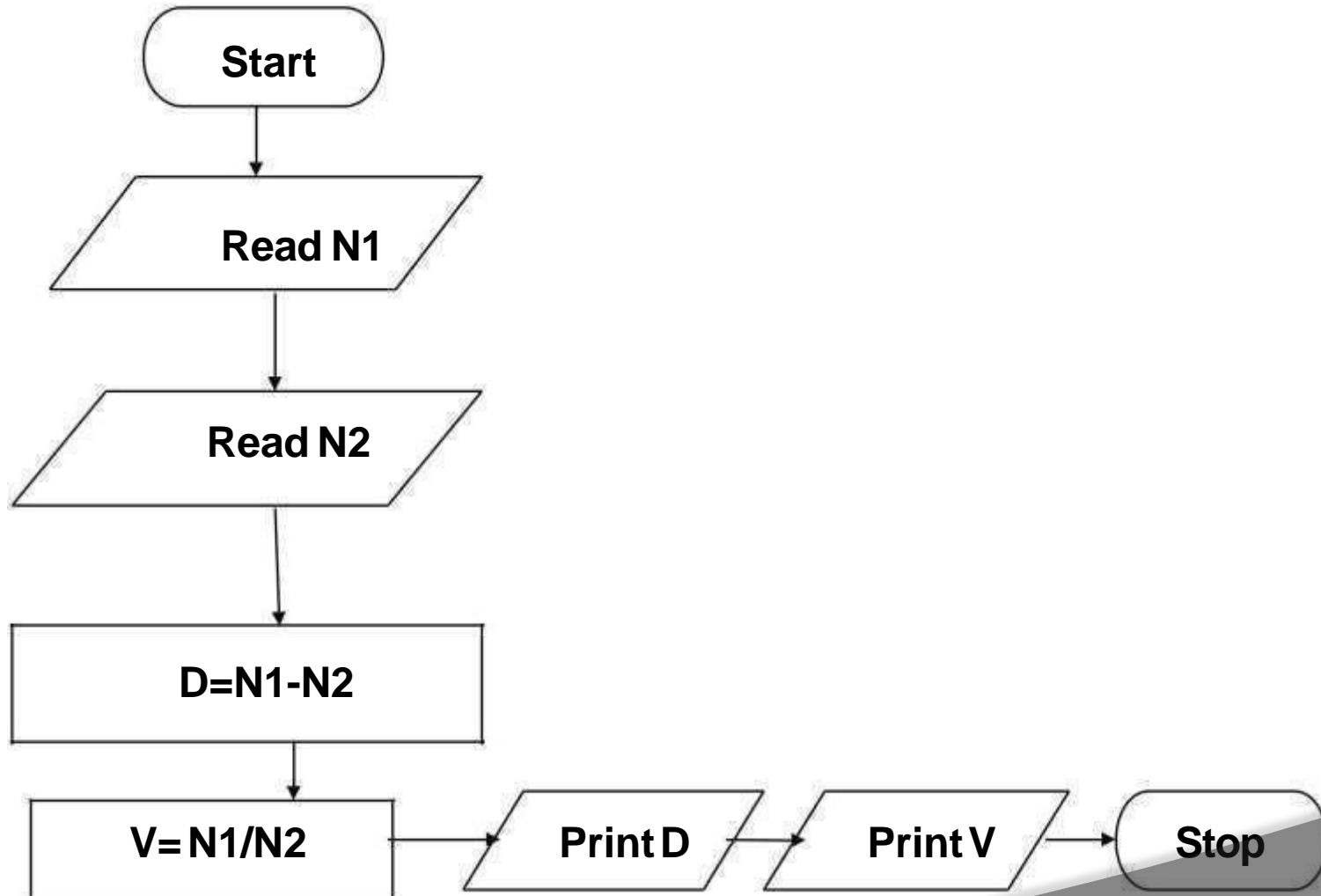
Example 3

➢Work on the algorithm and the flow chart of the problem of calculating the roots of the equation $Ax^2 + Bx + C = 0$

Variables:

A: Coefficient of X^2

B: Coefficient of X

C: Constant term

➢delta: Discriminant of theequation

X1: First root of theequation

X2: Second root of theequation

Algorithm:

Step 1: Start

Step 2: Input A, B and C

Step 3: Calculate delta = $B^2 - 4AC$

Step 4: If delta<0 go to step 6, otherwise go to 5

Step 5: If delta>0 go to step 7, otherwise go to 8

Step 6:Output —Complex roots . Go to step 13

Step 7: Output —real roots . Go to step 9

Step 8:Output —eɟual ꝺoots . Go to step 9

Step 9: Calculate X1=(-b+√delta)/(2*A)

Step 10: Calculate X2=(-b-√delta)/(2*A)

Flowchart

Start

Read A,B,C

Delta=B*B- 4*A*C

Delta < 0

Yes   No

"Complex Roots"

Delta>0

Yes   No

"Real Roots"

"Equal Roots"

$X1=(-b+\sqrt{\text{delta}})/(2*A)$

$X2=(-b-\sqrt{\text{delta}})/(2*A)$

Print X1   Print X2   Stop
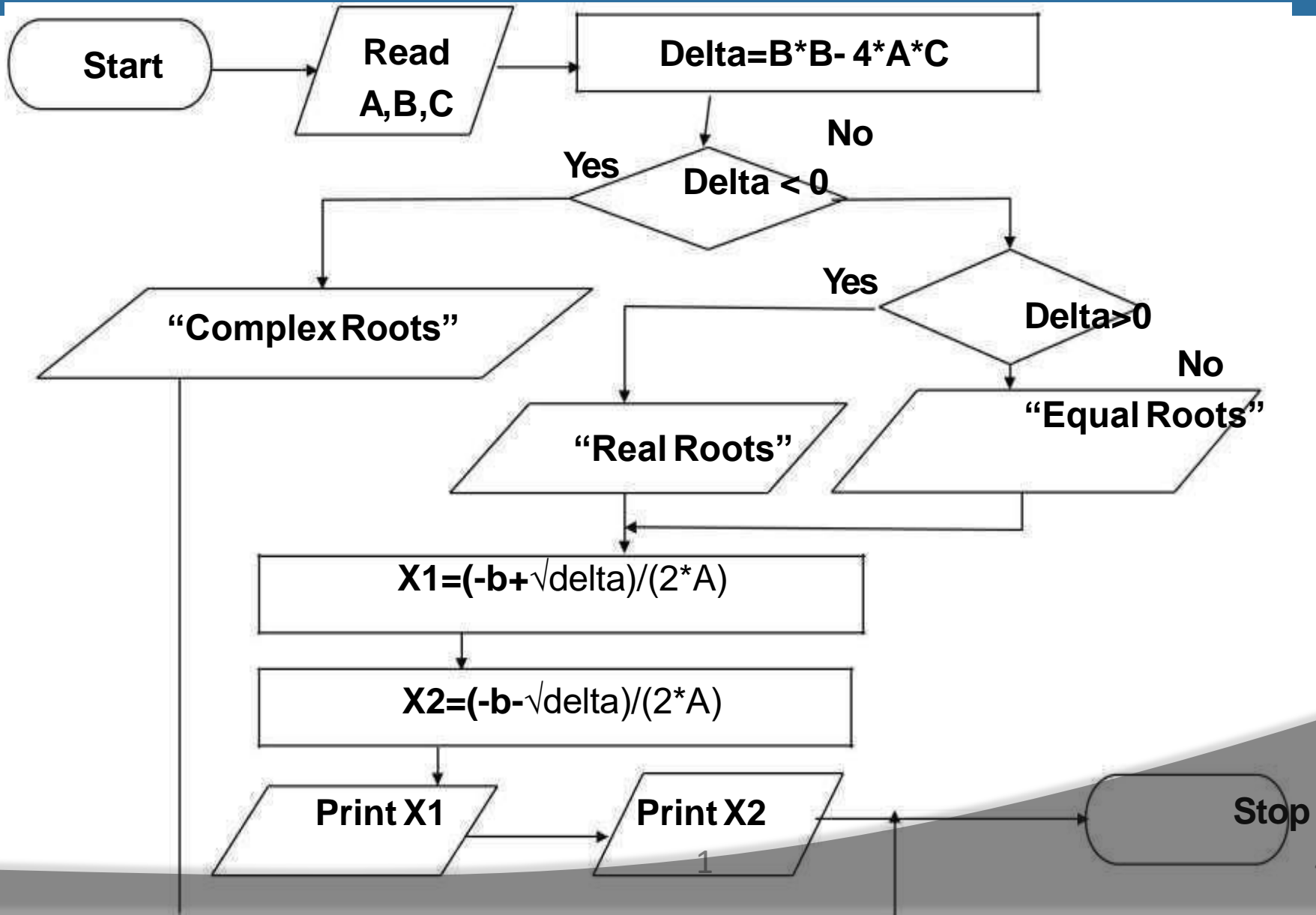
History of C language

History of C language is interesting to know. Here we are going to discuss brief history of c language.

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A.

Dennis Ritchie is known as the **founder of c language**.
It was developed to overcome the problems of previous languages such as B, BCPL etc.
Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL.
Let's see the programming languages that were developed before C language.

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

Structure Of C Program

Preprocessor Directives

Global Declarations

```
int main ( void )
{
        Local Declarations

        Statements

} // main
```

Other functions as required.
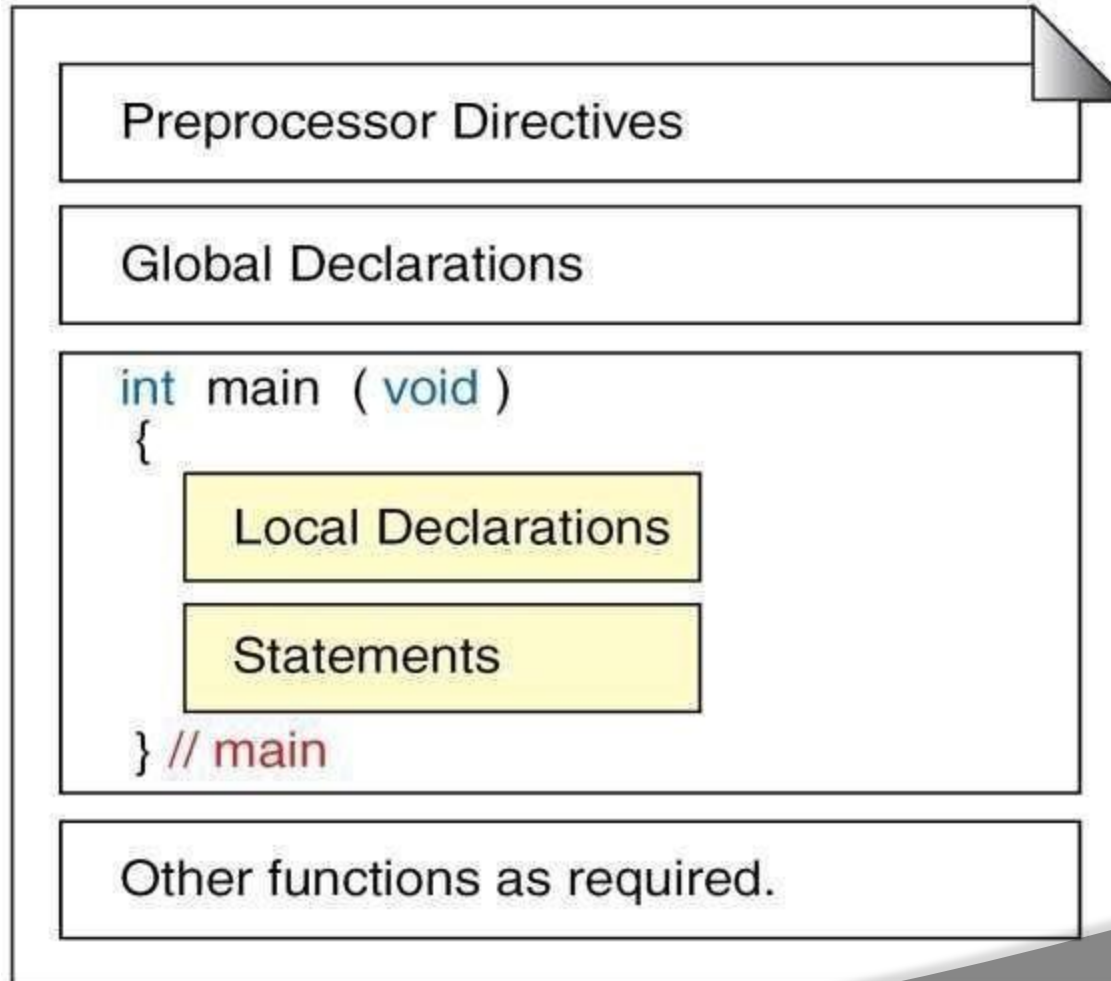
Preprocessor directives:

- Every C program is made of one or more Preprocessor directives or commands.

- They are special instructions to the preprocessor that tell it how to prepare the program for compilation.

- The preprocessor directives are commands that give instructions to the C preprocessor.

- A preprocessor directive begins with a number symbol (#) as its first non-blank character.

- Preprocessor commands can start in any column, but they traditionally start in column 1.

  Ex: #include <stdio.h>

Global Declaration Section:

Contains declarations that are visible to all parts of the program

Declaration section :

It is at the beginning of the function. It describes the data that will be used in the function. Declarations in a function are known as local declarations as they are visible only to the function that contains them.

Statements:

Statements follows the declaration section. It the contains instructions to the computer. Every statement ends with a semicolon.

Comments:

1. Comment about the program should be enclosed within /**/.
2. Any number of comments can be written at any place in the program.
3. Comments in the code helps to understand the code
4. Comments cannot be nested.

main( ):

The executable part of the program begins with the   function always. All statements that belong to main are enclosed in a pair of braces { }.

First C Program

```
#include <stdio.h>
void main ()
{
Printf("Hello World!");
}
```

Process of compiling and running C program

The steps involved in Creating and Running Programs are:

➢ Writing and Editing Programs
➢ Compiling Programs
➢ Linking Programs
➢ Executing Programs

Writing and Editing Programs

➢ To solve a particular problem a Program has to be created as a file using text editor / word processor. This is called source file.

➢ The program has to be written as per the structure and rules defined by the high-level language that is used for writing the program (C, JAVA etc).

Compiling Programs

➢The compiler corresponding to the high-level language will scan the source file, checks the program for the correct grammar (syntax) rules of the language.

➢If the program is syntactically correct, the compiler generates an output file called Object File which will be in a binary format and consists of machine language instructions corresponding to the computer on which the program gets executed.

Linking Programs

➢Linker program combines the Object File with the required library functions to produce another file called — executable file. Object file will be the input to the linker program.

➢The executable file is created on disk. This file has to be put into (loaded) the memory.

Executing Programs:

➢Loader program loads the executable file from disk into the memory and directs the CPU to start execution.

➢The CPU will start execution of the program that is loaded into the memory .

Building a C Program
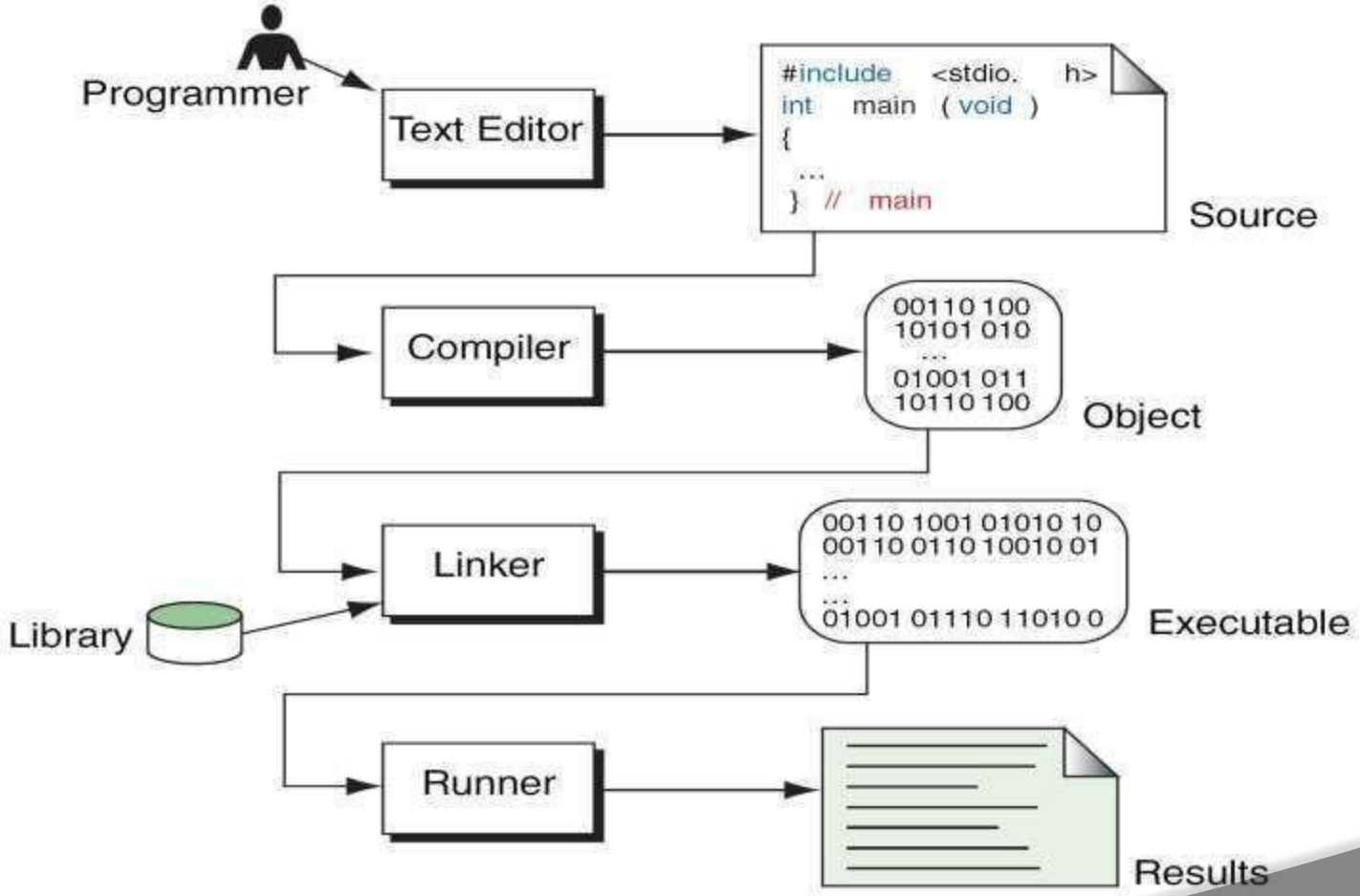
C tokens

➢ C tokens are the basic buildings blocks in C language which are constructed together to write a C program.

➢ Each and every smallest individual unit in a C program is known as C tokens. C tokens are of six types. They are

Keywords	(eg: int, while),
Identifiers	(eg: main, total),
Constants	(eg: 10, 20),
Strings	(eg: —total, —hello, Special Symbols;eg: (), {}),
Operators	(eg: +, /,-,*)

C Keywords

➢ **C keywords** are the words that convey a special meaningto the c compiler.

➢ The keywords cannot be used as variablenames.

➢ The list of C keywords is givenbelow:

- auto
- continue
- enum
- if
- short
- switch
- volatile
- break
- default
- extern
- int
- Signed
- const
- else
- Goto
- void
- Typedef
- While
- case
- do
- float
- long
- sizeof
- union
- char
- double
- for
- register
- static
- Unsigned
- return
- struct

c Identifiers

➢ Identifiers are used as the general terminology for the names of variables, functions and arrays.

➢ There are certain rules that should be followed while naming c identifiers:

➢ They must begin with a letter or underscore (_).

➢ They must consist of only letters, digits, orunderscore. No other special character is allowed.

➢ It should not be a keyword.

➢ It must not contain white space.

➢ It should be up to 31 characters long as only first31 characters are significant.

C Constants

➤A C constant refers to the data items that do not change their value during the program execution. Several types of C constants that are allowed in C are:

Integer Constants

➤Integer constants are whole numbers without any fractional part. It must have at least one digit and may contain either + or – sign. A number with no sign is assumed to be positive.

➤There are three types of integer constants:

Decimal Integer Constants

➢Integer constants consisting of a set of digits, 0 through 9, preceded by an optional – or + sign.

Example of valid decimal integer constants

341, -341, 0, 8972

Octal Integer Constants

➢Integer constants consisting of sequence of digits from the set

0 through 7 starting with 0 is said to be octal integer constants.

Example of valid octal integer constants

010, 0424, 0, 0540

Hexadecimal Integer Constants

➢ Hexadecimal integer constants are integer constants having sequence of digits preceded by 0x or 0X. They may also include alphabets from A to F representing numbers 10 to 15.

➢ Example of valid hexadecimal integer constants 0xD, 0X8d, 0X, 0xbD

➢ It should be noted that, octal and hexadecimal integer constants are rarely used in programming.

Real Constants

The numbers having fractional parts are called real or floating point constants.

These may be represented in one of the two forms called *fractional form* or the *exponent form* and may also have either + or − sign preceding it.

Example of valid real constants in fractional form or decimal notation 0.05, -0.905, 562.05, 0.015

Representing a real constant in exponent form

➢The general format in which a real number may be represented in exponential or scientific form is

mantissa e exponent

➢The mantissa must be either an integer or a real number expressed in decimal notation.

➢The letter e separating the mantissa and the exponent can also be written in uppercase i.e. E And, the exponent must be an integer.

Character Constants

➤A character constant contains one single character enclosed within single quotes. Examples of valid character constants

    a' , Z', 5'

➤It should be noted that character constants have numerical values known as ASCII values, for example, the Value of A' is 65 which is its ASCII value.

Escape Characters/ Escape Sequences

C allows us to have certain non graphic characters in character constants.

Non graphic characters are those characters that cannot be typed directly from keyboard, for example, tabs, carriage return, etc.

These non graphic characters can be represented by using escape sequences represented by a backslash() followed by one or more characters.

NOTE: An escape sequence consumes only one byte of space as it represents a single character.

String Constants

> String constants are sequence of characters enclosed within double quotes. For example,

—hello|
—abc|
—hello911|

- Every sting constant is automatically terminated with a special character **„"** called the**null character** which represents the   end of the string.
- For example, —hello| will represent—hello| in thememory.

•Thus, the size of the string is the total number of characters
plus one for the null character.

Special Symbols

➢ The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.

➢ *+ ;() ,- , ; : * … = #

➢ **Braces{}:** These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.

➢ **Parentheses():** These special symbols are used to indicate function calls and function parameters.

➢ **Brackets[]:** Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.

Variables

- A variable is nothing but a name given to a storage area that our programs can manipulate.

- Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

- The name of a variable can be composed of letters, digits, and the underscore character.

- It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive.

Type	Description
Char	Typically a single octet(one byte). This is an integer type.
Int	The most natural size of integer for the machine.
Float	A single-precision floating point value.
Double	A double-precision floating point value.
void	Represents the absence of type.

Data types

- Data types specify how we enter data into our programs and what type of data we enter.

- C language has some predefined set of data types to handle various kinds of data that we can use in our program.

- These datatypes have different storage capacities.

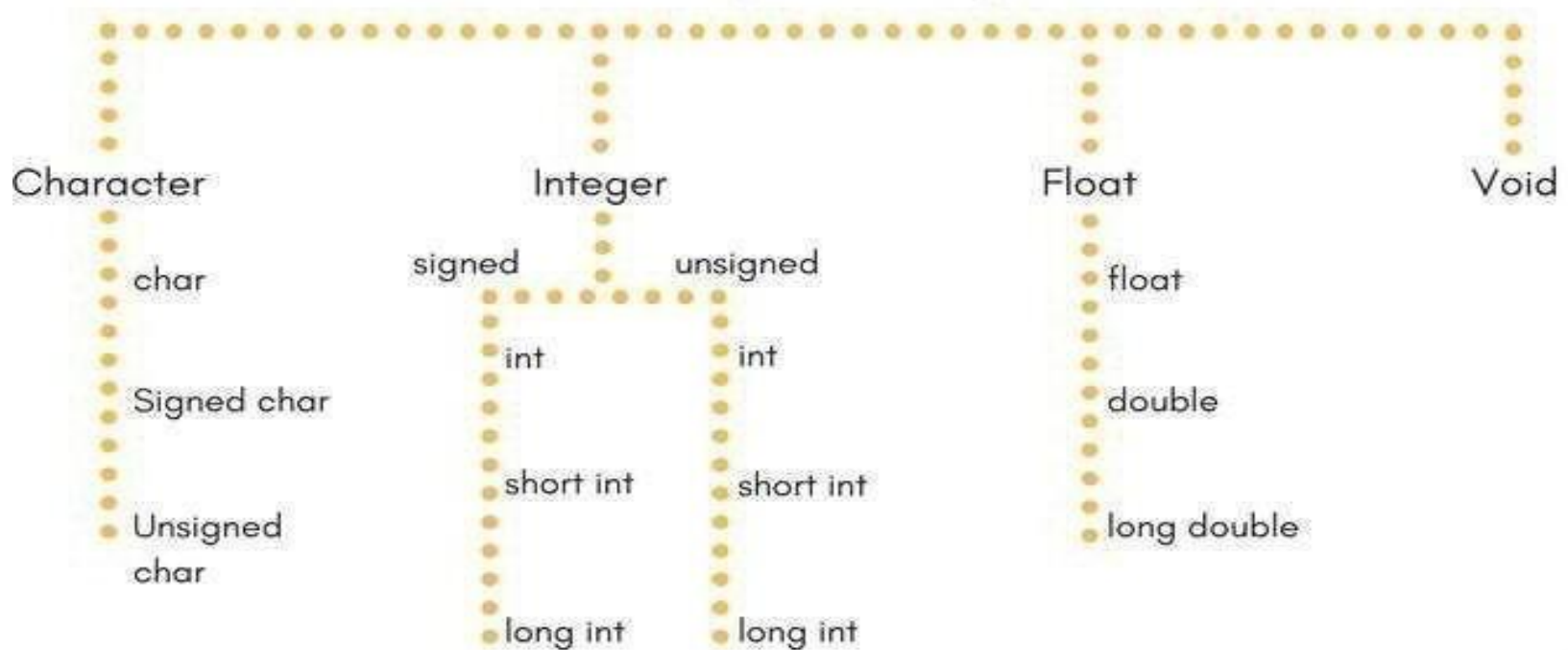C language supports 2 different type of data types:

Primary data types:

➤These are fundamental data types in C namely integer(int), floating point(float), character(char) and void.

Derived data types:

➤Derived data types are nothing but primary data types but a little twisted or grouped together like array, structure, union and pointer. These are discussed in details later.

Primary Data Type

Character — char — Signed char — Unsigned char

Integer — signed (int, short int, long int) / unsigned (int, short int, long int)

Float — float — double — long double

Void

Integer type

Integers are used to store whole numbers.

Size and range of Integer type on 16-bit machine:

Type	Size(bytes)	Range
int or signed int	2	-32,768 to 32767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Floating point type

Floating types are used to store real numbers.

Size and range of Integer type on 16-bit machine

Float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Character type

Character types are used to store characters value.

Size and range of Integer type on 16-bit machine

char or signed char	1	-128 to 127
unsigned char	1	0 to 255

void type

- void type means no value.

- This is usually used to specify the type of functions which returns nothing.

- We will get acquainted to this datatype as we start learning more advanced topics in C language, like functions, pointers etc.

Operators

> C language offers many types of operators. They are,

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bit wise operators
6. Conditional operators (ternary operators)
7. Increment/decrement operators
8. Special operators

Arithmetic Operators

➢C Arithmetic operators are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus in C programs.

S.no	Arithmetic Operators	Operation	Example
1	+	Addition	A+B
2	–	Subtraction	A-B
3	*	multiplication	A*B
4	/	Division	A/B
5	%	Modulus	A%B

Assignment operators

➢ The following table lists the assignment operators supported by the C language –

Operator	Description
=	Simple assignment operator. Assigns values from right side operands to left side operand
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.
<<=	Left shift AND assignment operator.
>>=	Right shift AND assignment operator.

Relational operators

➢ The following table shows all the relational operators supported by C language. Assume variable A holds 10 and variable B holds 20 then –

Operator	Description
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.

Logical operators

➢ Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then –

Operator	Description
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.

1

Bit wise operators

&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand

Conditional operators (ternary operators)

➢ Conditional operators return one value if condition is true and returns another value is condition is false.

➢ This operator is also called as ternary operator.

Syntax : (Condition?true_value:false_value)

➢ In above example, if A is greater than 100, 0 is returned else 1 is returned. This is equal to if else conditional statements.

Increment/decrement operators

➤ Increment operators are used to increase the value of the variable by one and decrement operators are used to decrease the value of the variable by one in C programs.

Syntax:

Increment operator: ++var_name; (or) var_name++;

 decrement operator: --var_name; (or) var_name--;

Special operators

S.no	Operators	Description
1	&	This is used to get the address of the variable. Example : &a will give addressof a.
2	*	This is used as pointer to a variable. Example : * a where, * is pointer to the variablea.
3	Sizeof ()	This gives the size of the variable. Example : size of (char) will give us 1.

Expressions

➢ Arithmetic expression in C is a combination of variables, constants and operators written in a proper syntax.

➢ C can easily handle any complex mathematical expressions but these mathematical expressions have to be written in a proper syntax.

➢ Some examples of mathematical expressions written in proper syntax of C are

➢ **Note**: C does not have any operator for exponentiation.

C Operator Precedence And Associativity

➤ C operators in order of *precedence* (highest to lowest).

➤ Their associativity indicates in what order operators of equal precedence in an expression are applied.

( ), [ ] --- Left to Right

*, /, % ---Left to Right

+, - ----Left to Right

>, < , >=, <= ---Left to Right

MODULE – II
CONTROL STRUCTURES

CLOs	Course Learning Outcome
CLO7	Understand branching statements, loop statements and use them in problem solving.
CLO8	Learn homogenous derived data types and use them to solve statistical problems.
CLO9	Learn homogenous derived data types and use them to solve statistical problems.
CLO10	Understand procedural oriented programming using functions

1

CLOs	Course Learning Outcome
CLO11	Understand how recursion works and write programs using recursion to solve problems.
CLO13	Differentiate call by value and call by reference parameter passing mechanisms.

Running Course Outcomes

| The course will enable the students to:	
CLO 7	Understand branching statements, loop statements and use them in problem solving.

Decision Statements - if Statement

> **Syntax of if statement:**

The statements inside the body of "if" only execute if the given condition returns true. If the condition returns false then the statements inside "if" are skipped

```
if (condition)
{
    //Block of C statements here
    //These statements will only execute if the condition is true
}
```

Flow Diagram of if statement
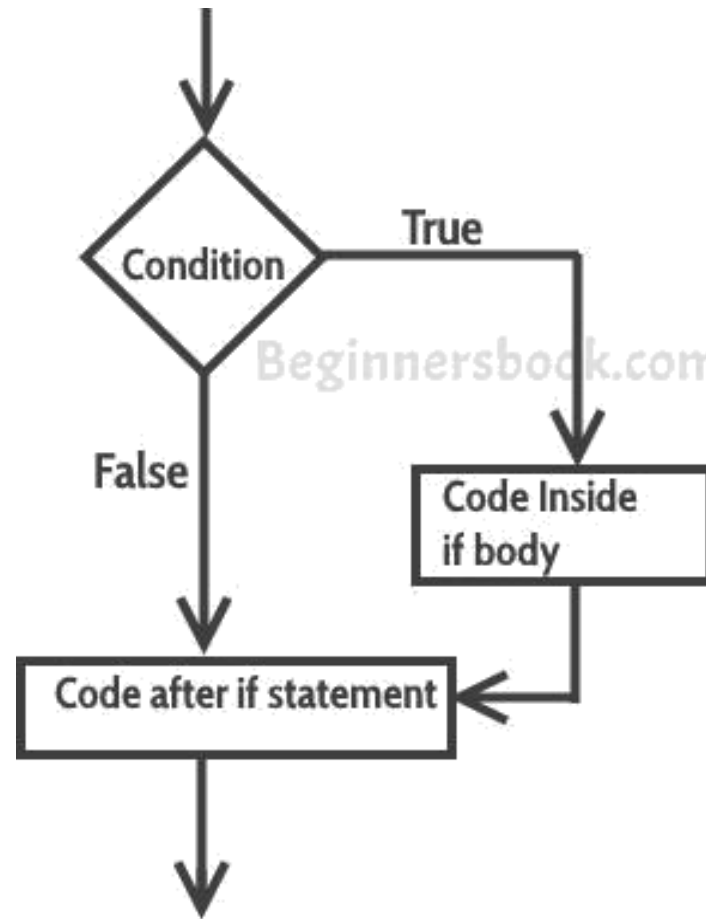
Example of if statement

```c
#include <stdio.h>
int main()
{
int x = 20;
int y = 22;
if (x<y)
{
printf("Variable x is less than y");
}
return 0;
}
```

If else statement

➤ **Syntax of if else statement:**

```
if(condition) {
// Statements inside body of if
}
else {
//Statements inside body of else
}
```

- If condition returns true then the statements inside the body of "if" are executed and the statements inside body of "else" are skipped.

- If condition returns false then the statements inside the body of "if" are skipped and the statements in "else" are executed.

Flow diagram of if else statement
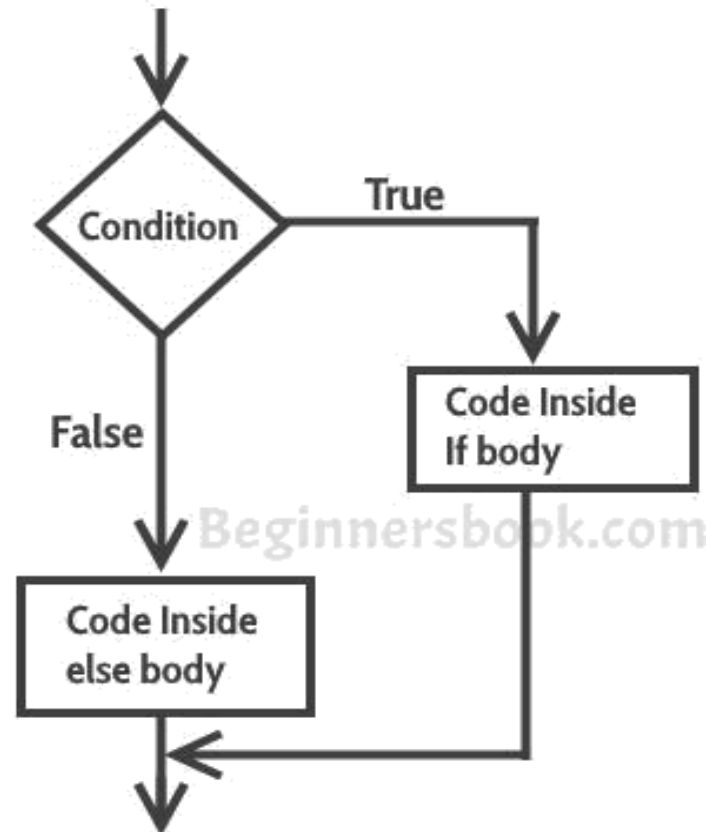
Program for checking eligible

```c
#include <stdio.h>
int main()
{
int age;
printf("Enter your age:");
scanf("%d",&age);
if(age >=18)
printf("You are eligible for voting");
else
 printf("You are not eligible for voting");
return 0;
}
```

Nested If..else statement

Syntax of Nested if else statement:

```
if(condition) {
//Nested if else inside the body of "if"
if(condition2) {
//Statements inside the body of nested "if"
}
else {
//Statements inside the body of nested "else"
}
}
else {
//Statements inside the body of "else"
}
```

Example of nested if..else

```c
#include <stdio.h>
int main()
{
int var1, var2;
printf("Input the value of var1:");
scanf("%d", &var1);
printf("Input the value of var2:");
scanf("%d",&var2);
if (var1 != var2)
{
printf("var1 is not equal to var2\n");
//Nested if else
```

```c
if (var1 > var2)
{
printf("var1 is greater than var2\n");
}
else
{
printf("var2 is greater than var1\n");
}}
else
{
printf("var1 is equal to var2\n");
}
return 0;
}
```

else..if statement

Syntax of else..if statement:

```
if (condition1)
{
//These statements would execute if the condition1 is true
}
else if(condition2)
{
//These statements would execute if the condition2 is true
}
.
.
else
{//These statements would execute if all the conditions returnfalse.}
```

Example of else..if statement

```
#include <stdio.h>  int
main()
{
int var1, var2;
printf("Input the value of var1:");
scanf("%d", &var1);
printf("Input the value of var2:");
scanf("%d",&var2);
if (var1 !=var2)
{
printf("var1 is not equal to var2\n");
}
else if (var1 > var2)
```

1

```c
{
printf("var1 is greater than var2\n");
}
else if (var2 > var1)
{
printf("var2 is greater than var1\n");
}
else
{
printf("var1 is equal to var2\n");
}
return 0;
}
```

Switch statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax

The syntax for a **switch** statement in C programming language is as follows –

```
switch(expression)
 {
case constant-expression :
 statement(s);
 break; /* optional */
case constant-expression :
 statement(s);
 break; /* optional */
/* you can have any number of case statements */
default : /* Optional */
 statement(s);
 }
```

The following rules apply to a **switch** statement −

➢ The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

➢ You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

➢ The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

➢ When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.

➢ When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

➢ Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.

➢ A **switch** statement can have an optional **default** case, which must appear at the end of the switch.

Flow Diagram of switch
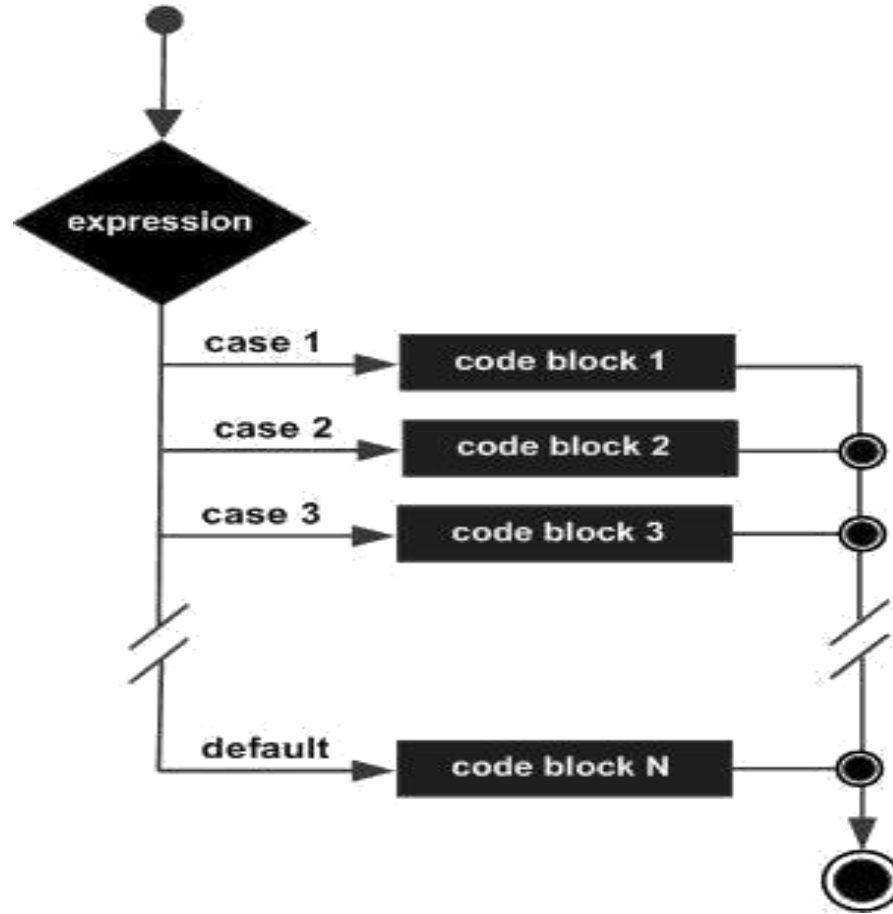
Example

```c
#include <stdio.h>
int main () {
 /* local variable definition */
 char grade = 'B';  switch(grade) {
 case 'A' :
printf("Excellent!\n" );
break;

case 'B' :
case 'C' :
printf("Well done\n" );
```

```c
break;
 case 'D' :
printf("You passed\n" );
break;
case 'F' :
printf("Better try again\n" );
break;

default :
printf("Invalid grade\n" );
}
printf("Your grade is %c\n", grade );
return 0;
}
```

Loop control statements - While loop

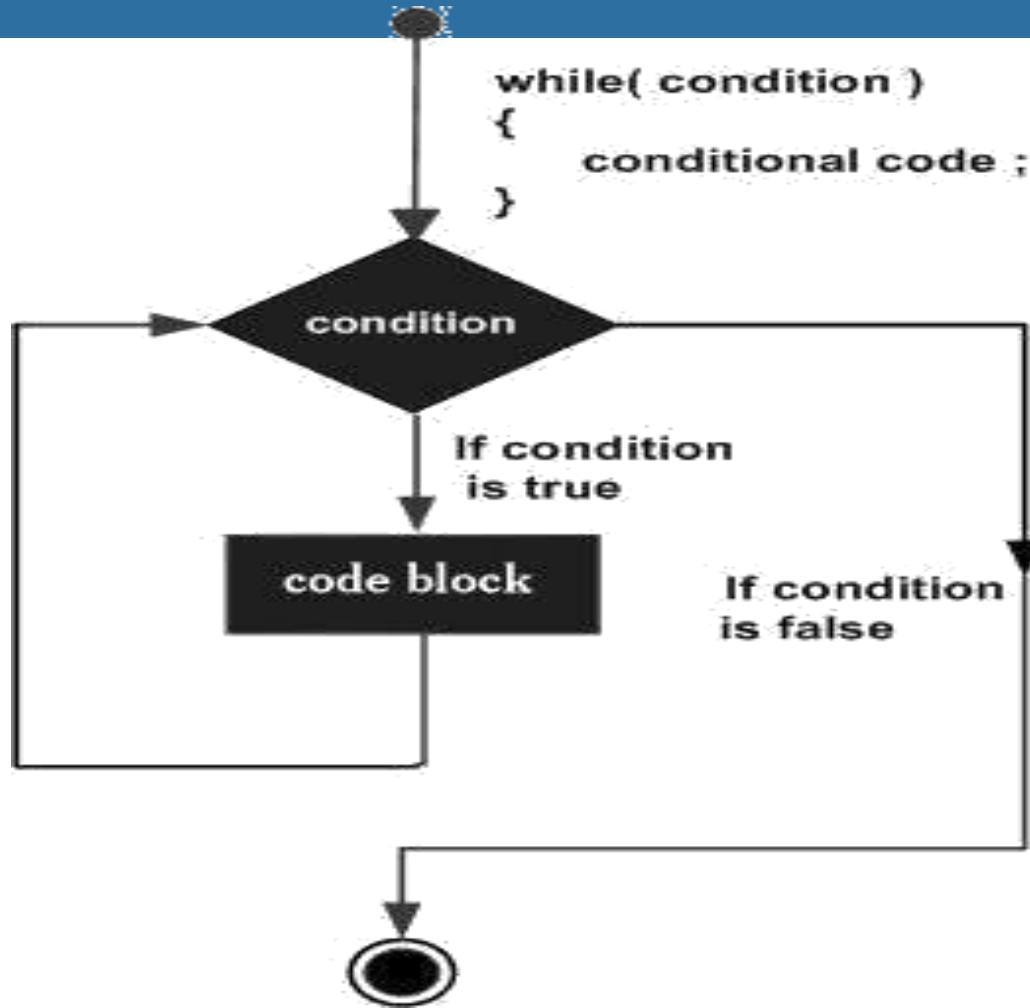A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a **while** loop in C programming language is –

while(condition) {

 statement(s);
 }

➢ Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

➢ When the condition becomes false, the program control passes to the line immediately following the loop.

➢ Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Flow Diagram


```
while( condition )
{
    conditional code ;
}
```

condition

If condition is true

code block

If condition is false

Example

```c
#include <stdio.h>
 int main () {
/* local variable definition */
int a = 10;
/* while loop execution */
while( a < 20 ) {
printf("value of a: %d\n", a);
a++;
}
 return 0;
}
```
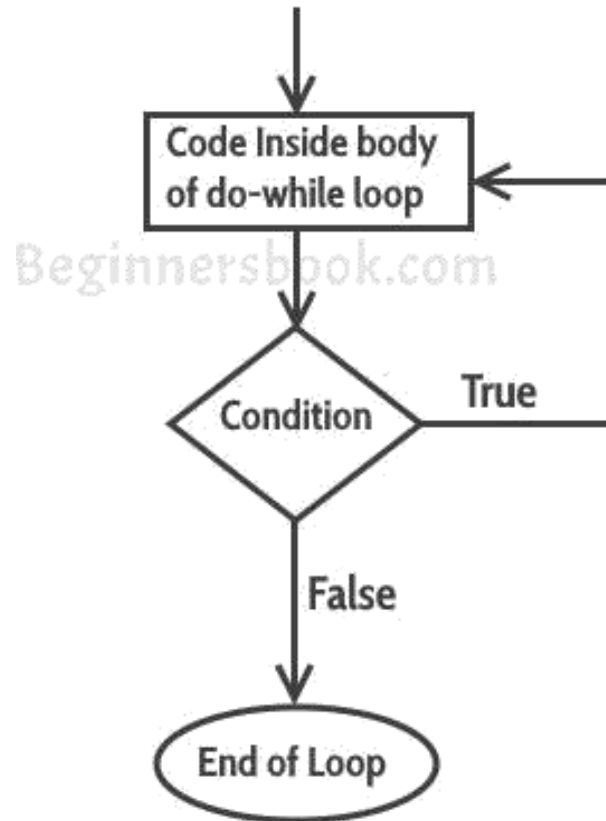
Do-while loop

➢ A do while loop is similar to while loop with one exception that it executes the statements inside the body of do-while before checking the condition.

➢ On the other hand in the while loop, first the condition is checked and then the statements in while loop are executed.

➢ So you can say that if a condition is false at the first place then the do while would run once, however the while loop would not run at all.

Syntax of do-while loop

```
do
{
//Statements

}while(condition test);
```

Flow diagram of do while loop

Example of do while loop

```c
#include <stdio.h>
int main()
{
int j=0;
do
{
printf("Value of variable j is: %d\n", j);
j++;
}while (j<=3);
return 0;
}
```

For loop

- C Language provides us different kind of looping statements such as For loop, while loop and do- while loop.

- In order to do certain actions multiple times, we use loop control statements.

- For loop can be implemented in different verities of using for loop –

- ✓ Single Statement inside For Loop

- ✓ Multiple Statements inside For Loop

- ✓ No Statement inside For Loop

- ✓ Semicolon at the end of For Loop

- ✓ Multiple Initialization Statement inside For

Syntax:

```
for (initialization expr; test expr; update expr)
 {
// body of the loop
// statements we want to execute
 }
```

Steps are repeated till exit condition comes.

➢ **Initialization Expression**: In this expression we have to initialize the loop counter to some value. for example: int i=1;

➢ **Test Expression**: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of loop and go to update expression otherwise we will exit from the for loop. For example: i <= 10;

➢ **Update Expression**: After executing loop body this expression increments/decrements the loop variable by some value. for example: i++;
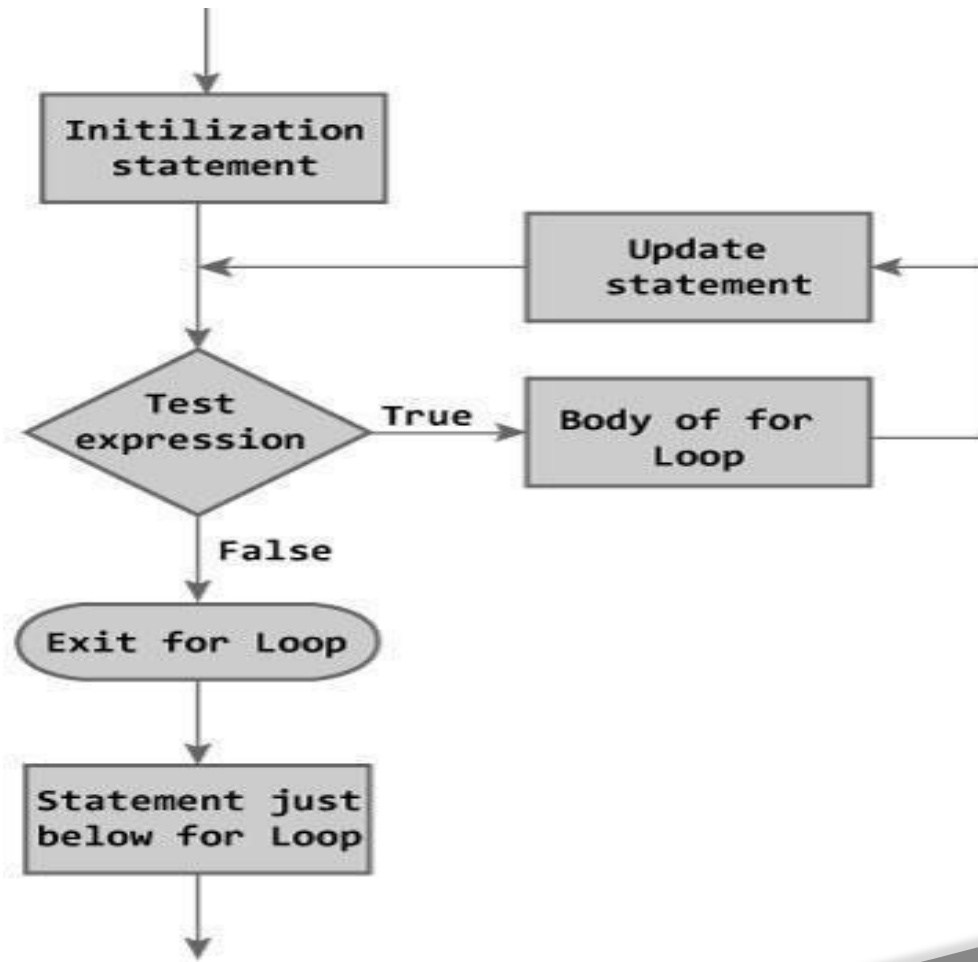
Figure: Flowchart of for Loop

Example: for loop

```c
#include <stdio.h>
int main() {
 int num, count, sum = 0;
printf("Enter a positive integer: ");
scanf("%d", &num);
// for loop terminates when n is less than count
for(count = 1; count <= num; ++count)
{
sum += count;
 }
printf("Sum = %d", sum);
 return 0; }
```

Jump statements-break

➢ Break Statement Simply Terminate Loop and takes control out of the loop.

Break in For Loop :

```
for(initialization ; condition ; incrementation)
{
 Statement1;
Statement2;
break;
 }
```

Break in While Loop :

```
initialization ;
 while(condition)
 {
 Statement1;
Statement2;
incrementation
break;


}
```

Example

```c
#include <stdio.h>
 int main () {
 /* local variable definition */
int a = 10;
 /* while loop execution */
while( a < 20 )
 {
 printf("value of a: %d\n", a);
 a++;
 if( a > 15) {
/* terminate the loop using break statement */ break;
} } return 0; }
```

Continue statement

➤ The **continue** statement in C programming works somewhat like the **break**statement.

➤ Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

➤ For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute.

➤ For the **while** and **do...while**loops, **continue** statement causes the program control to pass to the conditional tests.

Example

```c
#include <stdio.h>
 int main () {
 /* local variable definition */
int a = 10;
/* do loop execution */
do { if( a == 15) {
/* skip the iteration */
a = a + 1;
 continue;
} printf("value of a: %d\n", a);
 a++;
} while( a < 20 ); return 0; }
```

Goto statement

➤ A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

➤ **NOTE** – Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify.

➤ Any program that uses a goto can be rewritten to avoid them.

Example

```c
#include <stdio.h>
int main () {
/* local variable definition */
int a = 10;
/* do loop execution */
 LOOP:
do { if( a == 15) {
 /* skip the iteration */
 a = a + 1;
 goto LOOP;
} printf("value of a: %d\n", a); a++;
 }while( a < 20 ); return 0; }
```

MODULE – III
ARRAYS AND FUNCTIONS

Running Course Learning Outcomes

CLOs	COURSE LEARING OUTCOMES
CLO 8	Learn homogenous derived data types and use them to solve statistical problems.
CLO 9	Identify the right string function to write string programs.
CLO 10	Understand procedural oriented programming using functions.
CLO 11	Understand how recursion works and write programs using recursion to solve problems.
CLO 12	Differentiate call by value and call by reference parameter passing mechanisms.
CLO13	Understand storage classes and preprocessor directives for programming

Array

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

1. Array might be belonging to any of the data types
2. Array size must be a constant value.
3. Always, Contiguous (adjacent) memory locations are used to store array elements in memory.
4. It is a best practice to initialize an array to zero or null while declaring, if we don't assign any values to array.

TYPES OF C ARRAYS:

There are 2 types of C arrays. They are,

1. One dimensional array
2. Multi dimensional array
3. Two dimensional array
4. Three dimensional array
5. four dimensional array etc…

Array declaration, initialization and accessing

Array declaration syntax:

data_type arr_name [arr_size];

Array initialization

data_type arr_name *arr_size+=(value1, value2, value3,….);

Array accessing syntax:

arr_name[index];

ONE DIMENSIONAL ARRAY

Integer array example:

int age [5];

int age[5]={0, 1, 2, 3, 4};

age[0]; /*0 is accessed*/

age[1]; /*1 is accessed*/

age[2]; /*2 is accessed*/

Character array example:

char str[10];

char   str*10+=,'H','a','i'-;

str[0]; /*H is accessed*/

str[1]; /*a is accessed*/

str[2]; /*i is accessed*/

Example Program For One Dimensional Array In C:

```c
#include<stdio.h>
 int main()
 {
int i;
 int arr[5] = {10,20,30,40,50};
 // declaring and Initializing array in C
//To initialize all array elements to 0, use int arr[5]={0};
 /* Above array can be initialized as below also
 arr[0] = 10; to
arr[4] = 50; */
for (i=0;i<5;i++)
{// Accessing each variableprintf("value of arr[%d] is %d \n", i, arr[i]);}
}
```

Two dimensional array

➤ The two dimensional array in C language is represented in the form of rows and columns, also known as matrix. It is also known as *array of arrays* or *list of arrays*.

➤ The two dimensional, three dimensional or other dimensional arrays are also known as *multidimensional* arrays.

Declaration of two dimensional Array in C

We can declare an array in the c language in the following way.

data_type array_name[size1][size2];

A simple example to declare two dimensional array is given below.
int twodimen[4][3];

Here, 4 is the *row* number and 3 is the *column* number.

Two dimensional array example in C

```c
#include<stdio.h>
int main(){
int i=0,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
//traversing 2D array
for(i=0;i<4;i++){
for(j=0;j<3;j++){
printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
}//end of j
}//end of i
return 0;
}
```
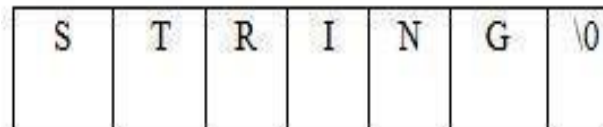
Strings

In C programming, array of characters or collection of characters is called a string. A string always recognized in double quotes. A string is terminated by a null character /0.

For example:

"String"

Here, "String" is a string. When, compiler encounters strings, it appends a null character /0 at the end of string.

S	T	R	I	N	G	\0

Example

```
#include<stdio.h>
#include<conio.h>
#include<string.h> void main ()
{
char str1[20]; char str2[20];
printf("Enter First Name");  scanf("%s"
,&str1); printf("Enter last Name" );
scanf("%s" ,&str2); puts(str1);
puts(str2);
}
```

String Handling Functions in C

These String functions are:

1. strlen().
2. strupr().
3. strlwr().
4. strcmp().
5. strcat().
6. strcpy().
7. strrev().

strlen()

size_t strlen(const char *str);

The function takes a single argument, i.e, the string variable whose length is to be found, and returns the length of the string passed.

The strlen() function is defined in <string.h> header file

Program finding length of a string

```
#include <stdio.h> #include
<string.h> int main()
{
char a*20+="Program"
char b*20+=,'P','r','o','g','r','a','m','\0'-;
char c[20];
printf("Enter string: ");  gets(c);
printf("Length of string a = %d  \n", strlen(a));   printf("Length
of string b = %d \n",  strlen(b));     printf("Length of string c =
%d \n", strlen(c));
return 0;}
```

strupr()

➢ strupr() function converts a given string into uppercase. Syntax for strupr( ) function is given below.

```c
#include<stdio.h>
#include<string.h>
int main()
{
char str[ ] = "Modify This String To Upper";
printf("%s\n",strupr(str));
return  0;
}
```

Output: MODIFY THIS STRING TO UPPER

strlwr()

strlwr( ) function converts a given string into lowercase. Syntax for strlwr( ) function is given below.

```c
#include<stdio.h>
 #include<string.h>
 int main()
{
 char str[ ] = "MODIFY This String To LOwer";
 printf("%s\n",strlwr (str));
 return  0;
}
```

OUTPUT: modify this string to lower

strcmp()

strcmp( ) function in C compares two given strings and returns zero if they are same.

```
#include   <stdio.h>
#include <string.h>
int main( )
{char str1[ ] = "fresh" ;
 char str2[ ] = "refresh" ;
 int i, j, k ;
i = strcmp ( str1, "fresh" ) ;
j = strcmp ( str1, str2 ) ;
 k = strcmp ( str1, "f" ) ;
 printf ( "\n%d %d %d", i, j, k ) ;return 0;}
```

strcat()

strcat( ) function in C language concatenates two given strings. It concatenates source string at the end of destination string.

```
#include <stdio.h>
 #include <string.h>
 int main( )
 {
 char source[ ] = " fresh2refresh" ;
char target[ ]= " C tutorial" ;
 printf ( "\nSource string = %s", source ) ;
 printf ( "\nTarget string = %s", target ) ;
 strcat ( target, source ) ;
 printf ( "\nTarget string after strcat( ) = %s", target ) ;}
```

strcpy()

strcpy( ) function copies contents of one string into another string

```c
#include   <stdio.h>
#include <string.h>
int main( )
{
 char source[ ] = "fresh2refresh" ;
 char target[20]= "" ;
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
strcpy ( target, source ) ;
 printf ( "\ntarget string after strcpy( ) = %s", target ) ;
return 0;
 }
```

strrev()

strrev( ) function reverses a given string in C language

#include<stdio.h>

#include<string.h>

int main()

 {

char name[30] = "Hello";

printf("String before strrev( ) : %s\n",name);

 printf("String after strrev( ) : %s",strrev(name));

 return 0;

}

Arrays of strings

➢ A string is a 1-D array of characters, so an array of strings is a 2-D array of characters.

➢ Just like we can create a 2-D array of int, float etc; we can also create a 2-D array of character or array of strings.

➢ Here is how we can declare a 2-D array of characters.

```
char ch_arr[3][10] = {
                    {'s', 'p', 'i', 'k', 'e', '\0'},
                    {'t', 'o', 'm','\0'},
                    {'j', 'e', 'r', 'r', 'y','\0'}
};
```

➢ It is important to end each 1-D array by the null character otherwise, it's just an array of characters. We can't use them as strings.

➢ Declaring an array of string this way is a tedious and error-prone process that's why C provides a more compact way to it. This above initialization is equivalent to:

```
char ch_arr[3][10] = {
                    "spike",
                    "tom",
                    "jerry"
          };
```

1

The following program demonstrates how to print an array of strings.

```c
#include<stdio.h>
 int main()
{
int i;
 char ch_arr[3][10] = {"spike","tom","jerry"};
 printf("1st way \n\n");
 for(i = 0; i < 3; i++)
{
printf("string = %s \t address = %u\n", ch_arr + i, ch_arr + i);
}
 signal to operating system program ran fine return 0;
}
```

Introduction to functions

➢ 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.

➢ You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

➢ A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

➢ The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

➢ A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

➢ The general form of a function definition in C programming language is as follows –

return_type function_name(parameter list) {
 body of the function

}

   A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function−

➢ **Return Type** – A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

➢ **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.

➢ **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

➢ **Function Body** – The function body contains a collection of statements that define what the function does.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two –

/* function returning the max between two numbers */

int max(int num1, int num2) {

/* local variable declaration */

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

Function Declarations

- A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

- A function declaration has the following parts –

  return_type function_name(parameter list);

- For the above defined function max(), the function declaration is as follows –

  int max(int num1, int num2);

- Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration–

  int max(int, int);

Function prototype

- A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

- A function prototype gives information to the compiler that the function may later be used in the program.

  returnType functionName(type1 argument1, type2 argument2,...);

- In the above example, int addNumbers(int a, int b); is the function prototype which provides following information to the compiler:

  name of the function is addNumbers()

  return type of the function is int

  two arguments of type int are passed to the function

Category of functions:

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

- ✓ Function with no return values, no arguments
- ✓ Functions with arguments, no return values
- ✓ Functions with arguments and return values
- ✓ Functions with no arguments and return values.

- **Function with no return values, no arguments**

- In this category, the function has no arguments. It does not receive any data from the calling function. Similarly, it doesn't return any value. The calling function doesn't receive any data from the called function. So, there is no communication between calling and called functions.

- **Functions with arguments, no return values**

- In this category, function has some arguments . it receives data from the calling function, but it doesn't return a value to the calling function. The calling function doesn't receive any data from the called function. So, it is one way data communication between called and calling functions.

- **Functions with arguments and return values**

- In this category, functions has some arguments and it receives data from the calling function. Simillarly, it returns a value to the calling function. The calling function receives data from the called function. So, it is two-way data communication between calling and called functions.

- **Functions with no arguments and return values.**

- In this category, the functions has no arguments and it doesn't receive any data from the calling function, but it returns a value to the calling function. The calling function receives data from the called function. So, it is one way data communication between calling and called functions.

Inter Function communication

- When a function gets executed in the program, the execution control is transferred from calling function to called function and executes function definition, and finally comes back to the calling function.

- In this process, both calling and called functions haveto

  communicate each other to exchange information.

- The process of exchanging information between calling and called functions is called as inter function communication.

- In C, the inter function communication is classified as follows...

❖ Downward Communication
❖ Upward Communication
❖ Bi-directional Communication

- **Downward Communication**

- In this type of communication, the data is transferred from calling function to called function but not from called function to calling function.

- The function with parameters and without return value are considered under Downward communication.

Example Program for swapping using function

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

int num1, num2 ;
void addition(int, int) ; // function declaration
clrscr() ;
num1 = 10 ;
num2 = 20 ;
printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2);
    addition(num1, num2) ; // calling function
getch() ;
}
void addition(int a, int b) // called function
{printf("SUM = %d", a+b) ;}
```

Upward Communication

- In this type of communication, the data is transferred from called function to calling function but not from calling function to called function.

- The function without parameters and with return value are considered under Upward communication.

Example Program on Upward Communication

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

 int result ;
 int addition() ; // function  declaration
 result = addition() ; // calling function  printf("SUM = %d",
 result) ;

 getch() ;
 }
int addition() // called function
{
 int num1, num2 ;
num1 = 10;num2 = 20;
return (num1+num2) ;}
```

Bi-Directional Communication

- In this type of communication, the data is transferred from called function to calling function and also from callingfunction to called function.

- The function with parameters and with return value are considered under Bi-Directional communication.

Example Program on Bidirectional Communication

```c
#include <stdio.h>
 void main(){
int num1, num2, result ;
 int addition(int, int) ; // function declaration
 num1 = 10 ;
 num2 = 20 ;
 result = addition(num1, num2) ; // calling function
 printf("SUM = %d", result) ;
getch() ;
 }
 int addition(int a, int b) // called function
{
 return (a+b) ;
 }
```

Function Calls

➤ There are two ways that a C function can be called from a program. They are,

1. **Call by value**
2. **Call by reference**

➤ Note:

- **Actual parameter** – This is the argument which is used in function call.
- **Formal parameter** – This is the argument which is used in function definition

Call by Value

➢ In **call by value** method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function.

➢ **The changes made on the formal parameters does not effect the values of actual parameters**. That means, after the execution control comes back to the calling function, the actual parameter values remains same.

➢ For example consider the following program…

```c
#include <stdio.h>
void main(){
int num1, num2 ;
void swap(int,int) ; // function declaration num1 = 10;
num2 = 20 ;

printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2);
swap(num1, num2) ; // calling function
printf("\nAfter swap: num1 = %d\nnum2 = %d", num1, num2);
}
void swap(int a, int b) // called function
{
int temp ;    temp = a ;    a = b ;
b = temp ;
}
```

Call by Reference

- In **Call by Reference** parameter passing method, the memory location address of the actual parameters is copied to formal parameters.

•This address is used to access the memory locations of the actual parameters in called function.

•In this method of parameter passing, the formal parameters must be **pointer** variables.

- **The changes made on the formal parameters effects the values of actual parameters**. For example consider the following program…

```c
#include <stdio.h>

void main(){

int num1, num2 ;

void swap(int *,int *) ; // function declaration num1 = 10;

num2 = 20 ;
    printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2);

swap(&num1, &num2) ; // calling function


printf("\nAfter swap: num1 = %d, num2 = %d", num1, num2);
}
```

*contd...+

```c
void swap(int *a, int *b) // called function
{
    int temp ;
    temp = *a ;
    *a = *b ;
    *b = temp ;
}
```

Parameter Passing Mechanism

In C Programming we have different ways of parameter passing schemes such as Call by Value and Call by Reference.

Function is good programming style in which we can write reusable code that can be called whenever require.

Whenever we call a function then sequence of executable statements gets executed. We can pass some of the information to the function for processing called argument.

Two Ways of Passing Argument to Function in C Language
✓ Call by Reference
✓ Call by Value

```c
#include<stdio.h>
void interchange(int number1,int number2)
{
int temp;
temp = number1;
number1 = number2;
number2 = temp;}

   int main() {
int num1=50,num2=70; interchange(num1,num2);
printf("\nNumber 1 : %d",num1);
printf("\nNumber 2 : %d",num2);
return(0);
}
```
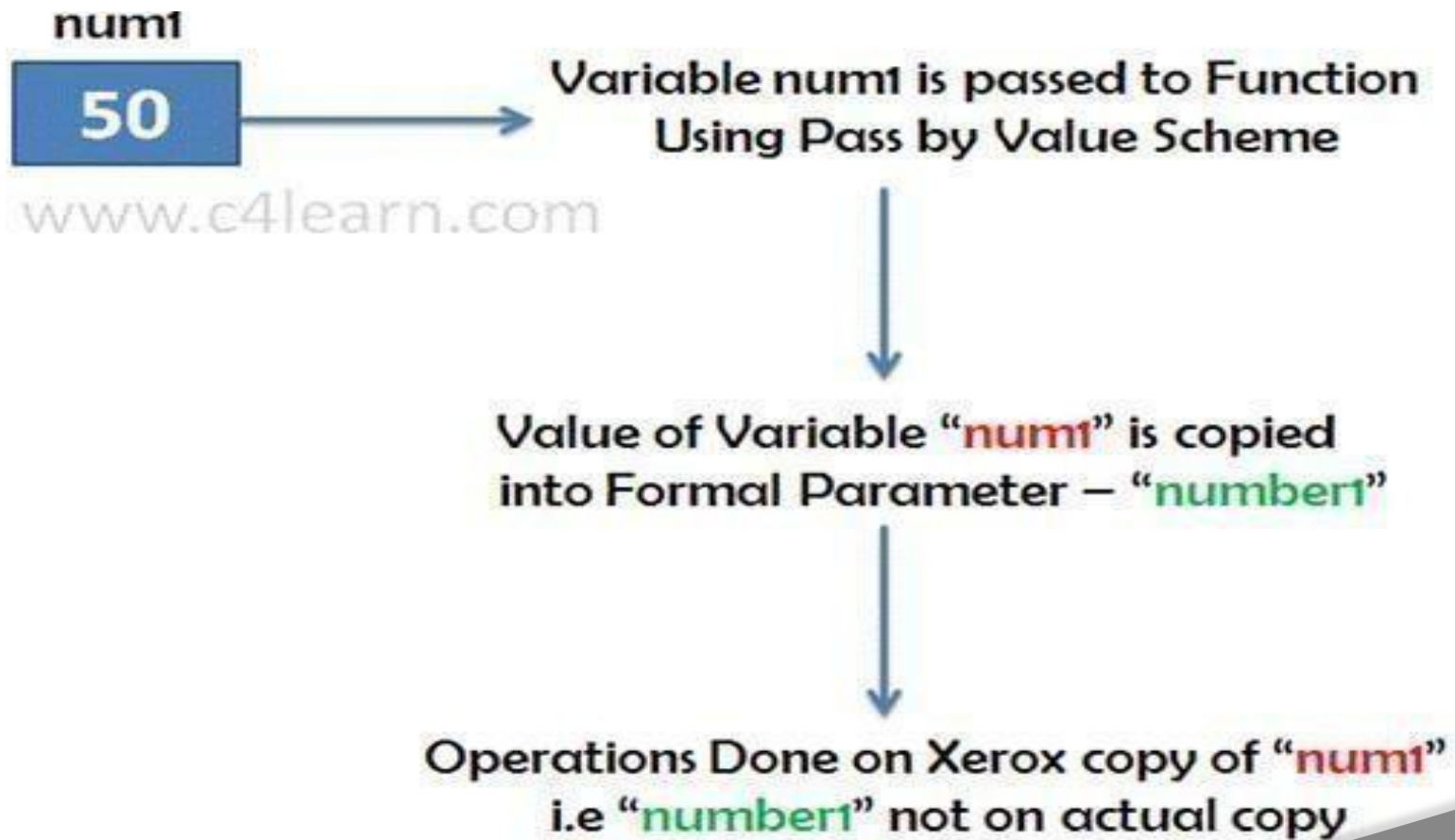
Output :
Number 1 :50
Number 2 :70

While Passing Parameters using call by value , xerox copy of original parameter is created and passed to the called function.

Any update made inside method will not affect the original value of variable in calling function.

In the above example num1 and num2 are the original values and xerox copy of these values is passed to the function and these values are copied into number1,number2 variable of sum functionrespectively.

A. Call by Reference/Pointer/Address :

```c
#include<stdio.h>
void interchange(int *num1,int *num2)
{
int temp;
temp = *num1;
*num1 = *num2;
*num2 = temp;
}
int main() {
int num1=50,num2=70;
interchange(&num1,&num2);
printf("\nNumber 1 : %d",num1); printf("\nNumber 2 : %d",num2);
return(0);
}
```

Output :
Number 1 :70
Number 2 :50

While passing parameter using call by address scheme , we are passing the actual address of the variable to the called function.

Any updates made inside the called function will modify the original copy since we are directly modifying the content of the exact memory location

Summary of Call By Value and Call By Reference :

Point	Call by Value	Call by Reference
Copy	Duplicate Copy of Original Parameter is Passed	Actual Copy of Original Parameter is Passed
Modification	No effect on Original Parameter after modifying parameter in function	Original Parameter gets affected if value of parameter changed inside function

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.

```
void recursion()
{
recursion();/* function calls itself */
}
int main()
{
recursion();
}
```

- 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.

Example: Factorial

```c
#include <stdio.h>
 unsigned long long int factorial(unsigned int i) {
 if(i <= 1) {
 return 1;
}
return i * factorial(i - 1);
 }
int main() {
 int i = 12;
printf("Factorial of %d is %d\n", i, factorial(i));
return 0;
}
```

Passing arrays to function

- Whenever we need to pass a list of elements as argument to any function in C language, it is prefered to do so using an array.

Declaring Function with array as a parameter

- There are two possible ways to do so, one by using call by value and other by using call by reference.

We can either have an array as a parameter.

int sum (int arr[]);

Or,

we can have a pointer in the parameter list, to hold the base address of our array.

int sum (int* ptr);

Returning an Array from a function

- We don't return an array from functions,   rather we return a pointer holding the base address of the array to be returned.

int* sum (int x[])

{

// statements return x ;

}

Passing a single array element to a function

```c
#include<stdio.h>
void giveMeArray(int a);
int main()
{
int myArray[] = { 2, 3, 4 };
giveMeArray(myArray[2]);
return 0;
}

void giveMeArray(int a)
{
printf("%d", a);
}
Output: 4
```
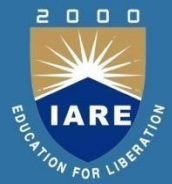
Passing a single array element to function (Call by value)

```c
#include <stdio.h>
void disp( char ch)
{
printf("%c ", ch);
}
int main()
{
char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
for (int x=0; x<10; x++)
{
disp (arr[x]);
}
return 0;
}
OUTPUT: a b c d e f g h i j
```

Passing array to function using call byreference

```c
#include <stdio.h>
void disp( int *num)

{
printf("%d ", *num);
disp(&arr[i]);
}

int main()

{
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9,0};
for (int i=0; i<10; i++)return0;
}
```
OUTPUT:

1 2 3 4 5 6 7 8 9 0

Passing String to a Function

- **Function declaration to accept one dimensional string**

  We know that strings are saved in arrays so, to pass an one dimensional array to a function we will have the following declaration.

  returnType functionName(char str[]);

- Example:

  void displayString(char str[]);

- In the above example we have a function by the name displayString and it takes an argument of type char and the argument is an one dimensional array as we are using the []square brackets.

Passing one dimensional string to a function

• To pass a one dimensional string to a function as an argument we just write the name of the string array variable.

• In the following example we have a string array variable message and it is passed to the displayString function.

```c
#include <stdio.h>
void displayString(char []); int main(void)
{
// variables
char message[] = "Hello World";
// print the string message
displayString(message); return 0;
}
void displayString(char str[])
{
printf("String: %s\n", str);
}
```

Storage Classes

➢ Every Variable in a program has memory associated with it.

➢ Memory Requirement of Variables is different for different types of variables.

➢ In C, Memory is allocated & released at different places

Scope	Region or Part of Program in which Variable is accessible
Extent	Period of time during which memory is associated with variable
Storage Class	Manner in which memory is allocated by the Compiler for Variable **Different Storage Classes**

Storage class of variable Determines following things

- ➢ Where the variable isstored
- ➢ Scope of Variable
- ➢ Default initial value of theVariable
- ➢ Lifetime of variable

Where the variable is stored:

➢ Storage Class determines the location of variable, where it is declared.

➢ Variables declared with auto storage classes are declared inside main memory whereas variables declared with keyword register are stored inside the CPU Register.

Scope of Variable

➤ Scope of Variable tells compile about the visibility of Variable in the block.

➤ Variable may have Block Scope, Local Scope and External Scope.

➤ A scope is the context within a computer program in which a variable name or other identifier is valid and can be used, or within which a declaration has effect.

Default Initial Value of the Variable

➢ Whenever we declare a Variable in C, garbage value is assigned to the variable.

➢ Garbage Value may be considered as initial value of the variable.

➢ C Programming have different storage classes which has different initial values such as Global Variable have Initial Value as 0 while the Local auto variable have default initial garbage value.

Lifetime of variable

➢ Lifetime of the = Time Of variable Declaration - Time of Variable Destruction

➢ Suppose we have declared variable inside main function then variable will be destroyed only when the control comes out of the main .i.e end of the program.

Different Storage Classes:

- ➢ Auto Storage class

- ➢ Static storageclass

- ➢ Extern storageclass

- ➢ Register storageclass

Automatic (Auto) storage class

This is default storage class

All variables declared are of type Auto by default

In order to Explicit declaration of variable use ‚auto' keyword auto

int num1 ; // Explicit Declaration

Storage   :   Memory

Scope   :   Local / Block Scope

Life time   :   Exists as long as Control remains in the block

Default initial Value : Garbage

External (extern) storage class

• Variables of this storage class are —Globalvariables

• Global Variables are declared outside the function and are accessible to all functions in theprogram

• Generally , External variables are declared again in the function using keyword extern In order to Explicit declaration of variable use ‗extern' keyword

extern int num1 ; // ExplicitDeclaration

Storage : Memory

Scope : Global / File Scope

Life time : Exists as long as variable is running Retains value within the function

Default initial Value : Zero

Static Storage Class

➤ The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.

➤ Therefore, making local variables static allows them to maintain their values between function calls.

➤ The static modifier may also be applied to global variables.

➤ When this is done, it causes that variable's scope to be restricted to the
file in which it is declared.

static int count = 5; /* global variable

Register Storage Class

➤ Register keyword is used to define local variable. Local variable are stored in register instead of **RAM**.

➤ As variable is stored in register, the **Maximum size of variable = Maximum Size of Register** unary operator [&] is not associated with it because Value is not stored in RAM instead it is stored in Register.

➤ This is generally used for **faster access**. Common use is —**Counter**—

Syntax

{

register int count;

}

Preprocessor directives

➢ The C preprocessor is a macro processor that is used automatically by the C compiler to transform your program before actual compilation(Proprocessor direcives are executed before compilation.).

➢ It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs.

➢ A macro is a segment of code which is replaced by the valueof macro. Macro is defined by #define directive.

➢ Preprocessing directives are lines in your program that start with #. The # is followed by an identifier that is the directive name. For example, #define is the directive that defines a macro. Whitespace is also allowed before and after the #.

➢ The # and the directive name cannot come from a macro expansion. For example, if foo is defined as a macro expanding to define, that does not make #foo a valid preprocessing directive.

➢ All preprocessor directives starts with hash # symbol.

List of preprocessor directives :

1. #include
2. #define
3. #undef
4. #ifdef
5. #ifndef
6. #if
7. #else
8. #elif
9. #endif
10. #error
11. #pragma

#include

➢ The #include preprocessor directive is used to paste code of given file into current file. It is used include system- defined and user-defined header files. If included file is not found, compiler renders error. It has three variants:

#include <file>

➢ This variant is used for system header files. It searches for a file named file in a list of directories specified by you, then in a standard list of system directories.

#include "file"

Macro's (#define)

➢ Let's start with macro, as we discuss, a macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive.

➢ Syntax

#define token value

#undef

➤ To undefine a macro means to cancel its definition. This is done with the #undef directive. Syntax:

#undef token

➤ define and undefine example

#include <stdio.h>
#define PI 3.1415
 #undef PI
main() { printf("%f",PI);
}

#ifdef

➢ The #ifdef preprocessor directive checks if macro is defined by #define. If yes, it executes the code.

Syntax:

```
#ifdef MACRO
    //code
#endif
```

#ifndef

➢ The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code.

Syntax:

```
#ifndef MACRO
  //code
#endif
```

#if

➤ The #if preprocessor directive evaluates the expressionor condition. If condition is true, it executes thecode.

Syntax:
#if expression

  //code

  #endif

#else

➢ The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax:

```
 #if expression
//if code
 #else
//else code
#endif
```

Syntax with #elif

#if expression
//if code
#elif expression
//elif code
#else
//else code
#endif

#error

> The #error preprocessor directive indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.

C #error example

#include<stdio.h>

#ifndef MATH_H

#error First include then compile
#else
void main()
{ float a; a=sqrt(7); printf("%f",a);
}
#endif

#pragma

➢ The #pragma preprocessor directive is used to provide additional information to the compiler.

➢ The #pragma directive is used by the compiler to offer machine or operating-system feature. Different compilers can provide different usage of #pragma directive.

➢ Syntax:

#pragma token

Example:

#include<stdio.h>
#include<conio.h>
void func() ;
#pragma startup func
#pragma exit func

Example Program on pragma

```
void main()
{
printf("\nI am in main");
getch();
}
void func()
{
printf("\nI am in func");
getch();
}
```

MODULE – IV
STRUCTURES, UNIONS AND POINTERS

CLOs	COURSE LEARNING OUTCOMES
CLO 14	Understand pointers conceptually and apply them in C programs.
CLO 15	Distinguish homogenous and heterogeneous data types and apply them in solving data processing applications.

Need of Structures

- **For example:** You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables name, citNo, salary to store these information separately.

- However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2

- You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

225

- A better approach will be to have a collection of all related information under a single name Person, and use it for every person.

- Now, the code looks much cleaner, readable and efficient as well.

- This collection of all related information under a single name Person is a structure.

Structure Basics

Structure Definition

•Structure is a collection of variables of different types under a single name.

•Keyword struct is used for creating a structure.

Syntax of structure

```
struct structure_name
{
data_type member1;
data_type member2;
.
.
data_type memeber;
};
```

Note: Don't forget the semicolon }; in the endingline.

We can create the structure for a person as mentioned above as:

struct person

{

char name[50];

int citNo;

float salary;

};

This declaration above creates the derived data type struct person.

Structure variable declaration

When a structure is defined, it creates a user-defined type but, no storage or memory is allocated.

For the above structure of a person, variable can be declared as:

```
struct person
{
char name[50];
int citNo;
float salary;
};
```

```
int main()
{
struct person person1, person2, person3[20];
return 0;
}
```

Another way of creating a structure variable is:

```
struct person
{
char name[50];
int citNo;
float salary;
} person1, person2, person3[20];
```

In both cases, two variables person1, person2 and an
array person3 having 20 elements of type **struct person** are created.

Accessing members of a structure

- There are two types of operators used for accessing members of a structure.
- ✓ Member operator(.)
- ✓ Structure pointer operator(->)

- Any member of a structure can be accessed as:

  structure_variable_name.member_name

- Suppose, we want to access salary for variable person2. Then, it can be accessed as:

  person2.salary

1

Example Program on structure

- **Write a C program to add two distances entered by user. Measurement of distance should be in inch and feet. (Note: 12 inches = 1 foot)**

```c
#include <stdio.h>
struct Distance
{    int feet;
float inch;
} dist1, dist2, sum;
int main()
{
printf("1st distance\n");
// Input of feet for structure variable dist1
```

Example Program on structure

```
printf("Enter feet: ");
scanf("%d", &dist1.feet);
// Input of inch for structure variable dist1
printf("Enter inch: ");
scanf("%f", &dist1.inch);
printf("2nd distance\n");
// Input of feet for structure variable dist2
printf("Enter feet: ");
scanf("%d", &dist2.feet);
// Input of feet for structure variabledist2
printf("Enter inch: ");
```

```c
scanf("%f",  &dist2.inch);
sum.feet = dist1.feet + dist2.feet;
sum.inch = dist1.inch +dist2.inch;
if (sum.inch > 12)
{
//If inch is greater than 12, changing it to feet.
  ++sum.feet;
sum.inch = sum.inch - 12;
}   // printing sum of distance dist1 and dist2
    printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);
    return 0;
}
```

Output

1st distance

Enter feet: 12

Enter inch: 7.92

2nd distance

Enter feet: 2

Enter inch: 9.8

Sum of distances = 15'-5.7"

Structure Initialization

When we declare a structure, memory   is not allocated   for un-initialized variable.

Let us discuss very familiar example of structure student , we can initialize structure variable in different ways –

Way1: Declare and Initialize

```
struct student
{
char name[20];
int roll;
float marks;
}std1 = { "Pritesh",67,78.3};
```

➢ In the above code snippet, we have seen that structure is declared and as soon as after declaration we have initialized the structure variable.

std1 = { "Pritesh",67,78.3}

Way2: Declaring and Initializing MultipleVariables

struct student

{

char name[20];

int roll;

float marks;

}

std1 = {"Pritesh",67,78.3};

std2 = {"Don",62,71.3};

std1 = {"Pritesh",67,78.3};

std2 = {"Don",62,71.3};

In this example, we have declared two structure variables in above code. After declaration of variable we have initialized two variable.

struct student

{

int mark1; **int**

mark2;   **int**

mark3;

} sub1={67};

- Though there are three members of structure,only one is initialized.
- Then remaining two members are initialized with Zero.
- If there are variables of other data type then their initial values will be

Data Type	Default value if not initialized
integer	0
float	0.00
char	NULL

```
struct student
{
int  mark1;  int  mark2;
 int mark3;
};
void main()
{
Struct student s1={89,54,65};
- - - --
- - - --
- - --
};
```

> When we declare a structure then memory won't be allocated for the structure. i.e only writing below declaration statement will never allocate memory

struct student

{

int mark1; **int** mark2; **int** mark3;

};

We need to initialize structure variable to allocate some memory to the structure.

struct student s1 = {89,54,65};

Accessing Structure Members

- Array elements are accessed using the Subscript variable , Similarly Structure members are accessed using dot [.] operator.

- (.) is called as "Structure member Operator".

- Use this Operator in between **"Structure name"** & **"member name".**

```c
#include<stdio.h>
struct stud
{
 char name[20];
char fname[10];
};
 struct stud s;
main()
{
 scanf("%s%s",&s.name,&s.fname);
printf("%s%s",s.name,s.fname);
}
 Output:vedha srinivas Vedhasrinivas
```

```c
struct employee
 {
 char name[100]; int age;
float salary;
char department[50];
 } employee_one = {"Jack", 30, 1234.5, "Sales"};
 int age = employee_one.age;
 float salary= employee_one.salary;
 char department= employee_one.department;
```

Accessing array of structure elements
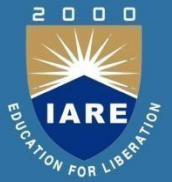
STRUCT STUD

{

 Datatype member1; Datatype member2;

.

.

} struct stud s[50];

Members of structures are accessed through dot operator.

Nested Structures

- A structure can be nested inside another structure. In other words, the members of a structure can be of any other type including structure.

- Here is the syntax to create nested structures.

Example Program

```
structure tagname_1
{
member1; member2; member3;...
Member n; structure tagname_2
{
member_1; member_2; member_3;...
member_n;
}, var1
}var2;
```

- To access the members of the inner structure, we write a variable name of the outer structure, followed by a dot(.) operator, followed by the variable of the inner structure, followed by a dot(.) operator, which is then followed by the name of the member we want to access.

- var2.var1.member_1 - refers to the member_1 of structure tagname_2 var2.var1.member_2 - refers to the member_2 of structure tagname_2

Example

```
struct student
{
struct person
{
char name[20];
 int age;
char dob[10];
} p ;
int rollno;
float marks;
} stu;
```

- It is important to note that structure person doesn't exist on its own. We can't declare structure variable of type struct person anywhere else in the program.

- Instead of defining the structure inside another structure. We could have defined it outside and then declare it's variable inside the structure where we want to use it. For example:

struct person

{

char name[20];

 int age;

   char dob[10];

};

- We can use this structure as a part of a bigger structure.

```
struct student
{
struct person info;
 int rollno;
float marks;
}
```

Initializing nested Structures

- Nested structures can be initialized at the time of declaration. For example:

struct person

{

char name[20]; int age; char dob[10];

};

struct student

{

struct person info; int rollno; float marks[10];

}

struct student student_1 = {{"Adam", 25,1990},101,90};

The following program demonstrates how we can use nested structures

```c
#include<stdio.h>
struct person
{
char name[20]; int age; char dob[10];};
 struct student
{
struct person info; int roll_no; float marks;
};
 int main()
{
struct student s1;printf("Details of student: \n\n");
```

```c
printf("Enter name: "); scanf("%s", s1.info.name);
 printf("Enter age: "); scanf("%d", &s1.info.age);
 printf("Enter dob: "); scanf("%s", s1.info.dob);
printf("Enter roll no: "); scanf("%d", &s1.roll_no);
printf("Enter marks: "); scanf("%f", &s1.marks);
printf("\n*******************************\n\n");
printf("Name: %s\n", s1.info.name);
printf("Age:  %d\n",  s1.info.age);
printf("DOB: %s\n",  s1.info.dob);
printf("Roll no:
%d\n", s1.roll_no);
printf("Marks: %.2f\n", s1.marks);
// signal to operating system program ran fine return 0;
}
```

Array of structures:

Need of array of structures:

•Structure is collection of different data type. An object of structure represents a single record in memory, if we want more than one record of structure type, we have to create an array of structure or object.

•As we know, an array is a collection of similar type, therefore an array can be of structure type.

•Structure is used to store the information of One particular object but if we need to store such 100 objects then Array of Structure is used.

Syntax:

Struct struct-name
{
datatype var1;
datatype var2;

----
datatype varN
};
Struct struct-name obj[size]

1

Initializing Array of Structure:

Alternative 1:

```
struct Book
{
char bname[20];
int pages;
char author[20];
float price;
} b1[3] = {
{"Let us C",700,"YPK",300.00},
{"Wings of Fire",500,"APJ Abdul Kalam",350.00},
{"Complete C",1200,"Herbt Schildt",450.00}
                        };
```

Initializing Array of Structure:

Alternative 2:

```
struct Book
  {
    char bname[20];
    int pages;
    char author[20];
    float price;
};
void main()
{
struct Book b1[3] = {{"Let usC",700,"YPK",300.00},
            {"Wings of Fire",500,"Abdul Kalam",350.00},
            {"Complete C",1200,"Herbt Schildt",450.00}
  };
}
```

Important Points:

Note 1: All Structure Members need not be initialized

#include<stdio.h>

struct Book

{

char bname[20];

int pages;

char author[20];

float price;

}b1[3] = {

{"Book1",700,"YPK"},

{"Book2",500,"AAK",350.00},

{"Book3",120,"HST",450.00}}}

```c
void main()
{
printf("\nBook Name       : %s",b1[0].bname);
printf("\nBook Pages : %d",b1[0].pages);

printf("\nBook Author : %s",b1[0].author);

printf("\nBook Price : %f",b1[0].price);
```

```c
struct Book
{
char bname[20];
int pages;
char author[20];
float price;
}b1[3] =
{
{},
{"Book2",500,"AAK",350.00},
{"Book3",120,"HST",450.00}
                                };
```

Output
- Book Name   :
- Book Pages   : 0 Book Author
- Book Price   : 0.000000

It is clear from above output , Default values for different data types.

Data Type	Default InitializationValue
Integer	0
Float	0.0000
Character	Blank

Structures and functions

Structure can be passed to functions by twomethods

1. Passing by value (passing actual value asargument)

2. Passing by reference (passing address of asargument)

Passing structure by value

•A structure variable can be passed to the function as an argument as a normal variable.

•If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.

Passing structure by reference

•The memory address of a structure variable is passed to function while passing it by reference.

•If structure is passed by reference, changes made to the structure variable inside function definition reflects in the originally passed structure variable.

Structures and pointers

Structures can be created and accessed using pointers. A pointer variable of a structure can be created as below:

```
struct name {
member1;
member2;
.
.
};

int main()
{
struct name *ptr;
}
```

Here, the pointer variable of type **struct name** is created.

Accessing structure's member through pointer

- A structure's member can be accesssed through pointer in two ways:

- Referencing pointer to another address to accessmemory
- Using dynamic memory allocation

1. Referencing pointer to another address to access the memory

Consider an example to access structure's member through pointer.

```c
#include <stdio.h>
typedef struct person
{
int age;
float weight;
};

int main()
{
struct person *personPtr, person1;
personPtr = &person1; // Referencing pointer to memory address of
        person1
printf("Enter integer: ");
scanf("%d",&(*personPtr).age);
```

```
printf("Enter number: ");
scanf("%f",&(*personPtr).weight);
printf("Displaying: ");
printf("%d%f",(*personPtr).age,(*personPtr).weight);
return 0;
}
```

In this example, the pointer variable of type struct person is referenced to the address of person1. Then, only the structure member through pointer can can accessed.

Using -> operator to access structure pointer member

Structure pointer member can also be accessed using -> operator.

(*personPtr).age is same as personPtr->age

(*personPtr).weight is same as personPtr->weight

2. Accessing structure member through pointer using dynamic memory allocation

To access structure member using pointers, memory can be allocated dynamically using malloc() function defined under "stdlib.h" header file

Syntax to use malloc()

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

Example

Example to use structure's member through pointer using

   malloc() function.

#include   <stdio.h>

#include <stdlib.h>

struct person {

int age;
float weight;
char name[30];
};

int main()
{
struct person *ptr;

```c
int i, num;

printf("Enter number of persons:");
scanf("%d", &num);
ptr = (struct person*) malloc(num * sizeof(structperson));
// Above statement allocates the memory for n structureswith
    pointer personPtr pointing to base address */
for(i = 0; i < num; ++i)
{
printf("Enter name, age and weight of the person respectively:\n");
        scanf("%s%d%f", &(ptr+i)->name, &(ptr+i)->age, &(ptr+i)-
    >weight);
}.
```

```c
    printf("Displaying Infromation:\n");
for(i = 0; i < num; ++i)

    printf("%s\t%d\t%.2f\n", (ptr+i)->name, (ptr+i)->age, (ptr+i)-
    >weight);
return 0;
}
```

Output

Enter number of persons: 2

Enter name, age and weight of the person respectively:
Adam

2

3.2

Enter name, age and weight of the personrespectively:
Eve

6

2.3

Displaying Information:
Adam

Eve   6

Structure and Functions

- In C, structure can be passed to functions by two methods:

- ✓ **Passing by value (passing actual value as argument)**
- ✓ **Passing by reference (passing address of an argument)**

- **Passing structure by value**

- ✓ A structure variable can be passed to the function as an argument as a normal variable.
- ✓ If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.

Example 1

C program to create a structure student, containing name and roll and display the information.

```c
#include <stdio.h>

struct student

{

char name[50];

int roll;

};

void display(struct student stu);

// function prototype should be below to the structuredeclaration otherwise compiler shows error
```

```
int main()
{
struct student stud;
printf("Enter student's name: ");
scanf("%s", &stud.name);
printf("Enter roll number:");
scanf("%d", &stud.roll);
display(stud);          // passing structure variable stud as argument
return 0;

}
void display(struct student stu){
printf("Output\nName: %s",stu.name);
printf("\nRoll: %d",stu.roll);

}
```

Example Contd

Enter student's name: Kevin Amla

Enter roll number: 149

Output

Name: Kevin Amla

Roll: 149

Passing structure by reference

The memory address of a structure variable is passed to function while passing it by reference.

If structure is passed by reference, changes made to the structure variable inside function definition reflects in the originally passed structure variable.

Self Referencial Structures

Self referential structures contain a pointer member that points to a structure of the same structure type.

In other words, a self-referential C structure is the one which includes a pointer to an instance of itself.
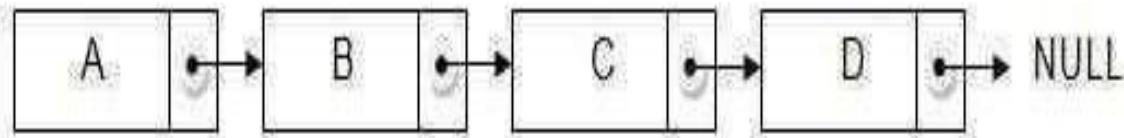

```
struct demo
{
    Data_type member1, member2;
    struct demo *ptr1,*ptr2;
}
```

Unions

- A union, is a collection of variables of different types, just like a structure. However, with unions, you can only store information in one field at any one time

- You can picture a union as like a chunk of memory that is used to store variables of different types.

- Once a new value is assigned to a field, the existing data is wiped over with the new data

•A union can also be viewed as a variable type that can contain many different variables (like a structure), but only actually holds one of them at a time (not like a structure).

•This can save memory if you have a group of data where only one of the types is used at a time.

•The size of a union is equal to the size of it's largest data or element

Union Declaration

union union-type-name

{

type variable-names;

type variable-names;

…

… }[union variable];

Union Intialization

union   student

{

char name[20];

int marks;

- union student s=,"surya",560-;

Members of union are accessed as s.name and s.marks

Sharing of a storage locating by union members

1000   1001   1002   1004

c

m

x
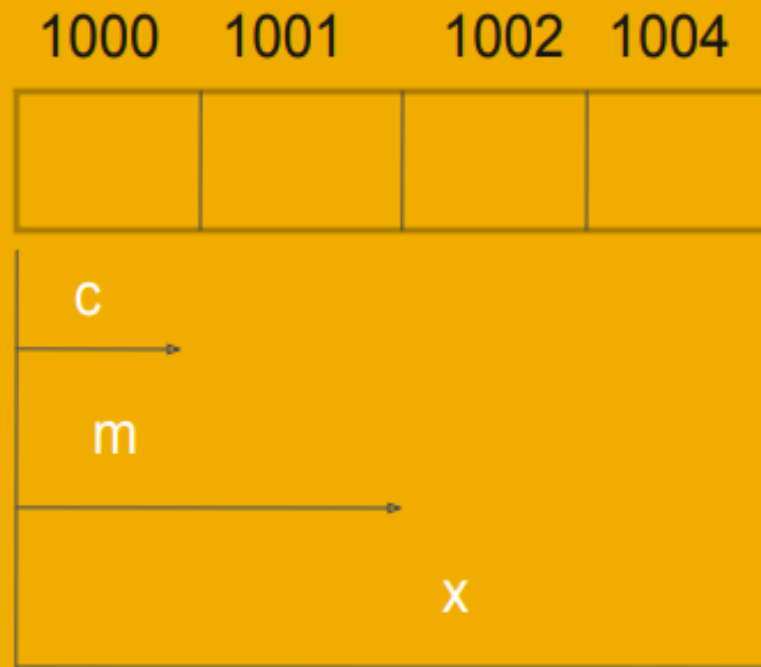
Difference between Structure & Union

Structure	Union
A structure allocates the total size of all elements in it.	A union only allocates as much memory as its largest element (member) requires.
Members inside a structure are always stored in separate memory locations throughout the lifetime and scope of the entire structure.	The union will store one and only one actual value for one element at a time.
Manipulations of one member will not affect the values of any of the others in any way unless they are operated on in code to do so.	If another element is stored before the first is retrieved, the first stored value is lost.
The keyword `struct` is used to declare a structure	The keyword `union` is used to declare an union.
All data members in a structure are active at time	Only on data member is active at a time.

Similarities between Structure & Union

They both can accept the dot (.) operator to address a member from the object name, as struct.member or union.member

They both use brace delimited declarations to create the template for the data object. Both accept tagname and name as well as explicit initialization as options.

They both can have their size correctly determined as maximum size in bytes by use of the sizeof() operator.

Bit Fields

- Bit field
  - Member of a structure whose size (in bits) has been specified
  - Enable better memory utilization
  - Must be defined as int or unsigned
  - Cannot access individual bits
- Defining bit fields
  - Follow unsigned or int member with a colon (:) and an integer constant representing the width of the field
  - Example:
    - struct BitCard {
    - unsigned face : 4;
    - unsigned suit : 2;
    - unsigned color : 1;
  - };

- Unnamed bit field
  - Field used as padding in the structure
  - Nothing may be stored in the bits
    - struct Example {
    - unsigned a : 13;
    - unsigned   : 3;
    - unsigned b : 4;
    - }
  - Unnamed bit field with zero width aligns next bit field to a new storage unit boundary

Example

For example, consider the following declaration of date withoutuse of bit fields.

```c
#include <stdio.h>
// A simple representation of date
struct date
{
unsigned int d;
 unsigned int m;
 unsigned int y;
};
int main()
{
```

printf("Size of date is %d bytes\n", sizeof(structdate));

 struct date dt = {31, 12, 2014};

  printf("Date is %d/%d/%d", dt.d, dt.m,dt.y);

Output:

Size of date is 12 bytes

 Date is 31/12/2014

typedef and enumerators

typedef:

- The C programming language provides a keyword called **typedef**, which you can use to give a type, a newname.

Syntax:

  typedef   data_type   new_name;

- typedef: It is a keyword.
- data_type: It is the name of any existing type or user defined type created using structure/union.
- new_name: alias or new name you want to give to anyexistingtype or user defined type.

Example for typedef:

- typedef unsigned char BYTE;

- After this type definition, the identifier BYTE can be used as an abbreviation type **unsigned char, for example..** For the BYTE b1, b2;

Example program on typedef

```c
#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;
```

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

```c
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;
}
```

Structures with typedef:

- struct student
  {
        int mark [2];
        char name [10];
        float average;
  }

Variable for the above structure can be declared intwoways

1st way :

- struct student record;   /* for normal variable */
  struct student *record;   /* for pointer variable*/

2nd way :

- typedef struct studentstatus;

Struct with typedef:

// Structure using typedef:

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

- {
- status record; record.id=1; strcpy(record.name, "Raju"); record.percentage = 86.5; printf(" Id is: %d \n", record.id);
- printf(" Name is: %s \n", record.name);
- printf(" Percentage is: %f \n", record.percentage);
- return 0;
- }

Enumeration data type:

- An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword enum is used.

Syntax:

   enum flag ,const1, const2,……constN};

- Here, name of the enumeration is *flag*. Constants like *const1*, *const2*,...., *constN* are values of type *flag*. By default, *const1* is 0, *const2* is 1 and so on. You can change default values of enum elements during declaration (if necessary).

   // Changing the default value of enum elements
   enum suit{
   club=0; diamonds=10; hearts=20; spades=3;
   };

```
enum boolean
{
 false;
true;
};
```

enum boolean check;

Here, a variable check is declared which is of type **enum**

boolean.

Example of enumerated type

```c
#include <stdio.h>

enum week{ sunday, monday, tuesday, wednesday, thursday,
friday, saturday};
int main()
{
enum week today;
today=wednesday;
printf("%d day",today+1);
return 0;
}
```
Output 4 day

Pointer Basics

Pointer Definition and syntax

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.

Syntax:

Data_type   *variable_name;

- Asterisk is called as **Indirection Operator.** It isalso called as Value at Address Operator

- It Indicates **Variable declared is of Pointer type.** pointer_name must follow the rules ofidentifier.

Examples of pointer declaration

- int   *ip;   /* pointer to an integer*/

- double *dp;   /* pointer to a double */

- float *fp;   /* pointer to a float */

- char   *ch   /* pointer to a character */

Diff   Between pointer and normal variable:

- int *ptr;   //Here ptr is Integer Pointer Variable

- int ptr;   //Here ptr is Normal Integer Variable

Pointer concept with diagrams

- **int i;**
- **int *j;**
- **j=&i;**

Variable Name ➔	I	j
Value of Variable ➔	3	65524
Address of Location ➔	65524	65522

Pointer Basic Example:

```c
#include <stdio.h>
int main()
{
int *ptr, i; i = 11;
/* address of i    is assigned to ptr */

ptr = &i;
/* show i's value using ptr variable */

printf("Value of i : %d", *ptr);
return 0;
}
```
OUTPUT:
Value of i is 11.

```c
#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;
}
```

- Address of c: 2686784
- Value of c: 22
- Address of pointer pc: 2686784
- Content of pointer pc: 22
- Address of pointer pc: 2686784
- Content of pointer pc: 11
- Address of c: 2686784
- Value of c: 2

pc	c	pc	c	pc	c	pc	c	pc	c
			22		22		11		2
Line 4: int *pc, c;		Line 5: c=22;		Line 8: pc=&c;		Line 11: c=11;		Line 14: *p=2;	
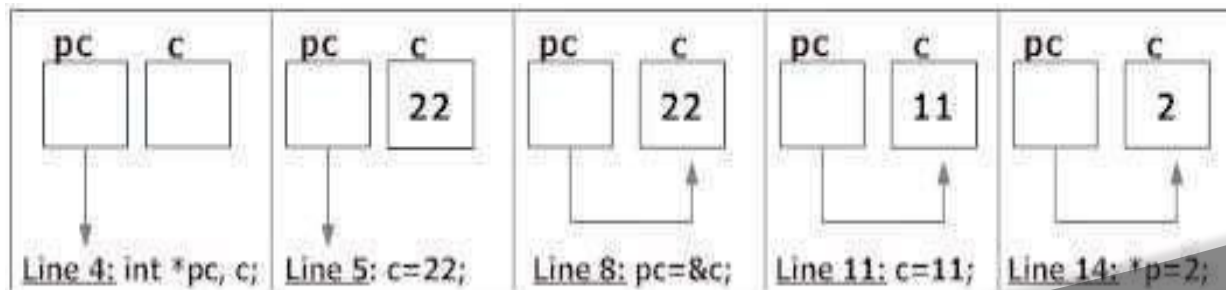
Pointer Arithmetic

- Pointer is a variable that points to a memory location. Memory addresses are numeric value that ranges from zero to maximum memory size in bytes.

- These addresses can be manipulated like simple variables. You can increment, decrement, calculate or compare these addresses manually.

- C language provides a set of operators to perform arithmetic and comparison of memory addresses.

- Pointer arithmetic and comparison in C is supported by following operators -

- Increment and decrement ++ and --

- Addition and Subtraction + and -

- Comparison <, >, <=, >=, ==, !=

Pointer increment and decrement

- Increment operator when used with a pointer variable returns next address pointed by the pointer. The next address returned is the sum of current pointed address and size of pointer data type

- Or in simple terms, incrementing a pointer will cause the pointer to point to a memory location skipping Nbytes from current pointed memory location. Where N is size of pointer data type.

- Similarly, decrement operator returns the previous address pointed by the pointer. The returned address is the difference of current pointed address and size of pointer data type.

Pointer addition and subtraction

- Pointer increment operation increments pointer by one. Causing it to point to a memory location skipping N bytes (where N is size of pointer data type).

- We know that increment operation is equivalent to addition by one. Suppose an integer pointer int * ptr. Now, ptr++ is equivalent to ptr = ptr + 1. Similarly, you can add or subtract any integer value to a pointer.

- Adding K to a pointer causes it to point to a memory location skipping K * N bytes. Where K is a constant integer and N is size of pointer data type.

- Let us revise the above program to print array using pointer

```c
#include <stdio.h>
#define SIZE 5
 int main()
{
    int arr[SIZE] = {10, 20, 30, 40, 50};
    int *ptr;
    int count;
    ptr = &arr[0]; // ptr points to arr[0]
count = 0;
printf("Accessing array elements using pointer \n");
```

```c
while(count < SIZE)
    {
        printf("arr[%d] = %d \n", count, *(ptr + count));
count++;
    }


    return 0;
}
```

Pointer comparison

- In C, you can compare two pointers using relational operator. You can perform six different type of pointer

  comparison <, >, <=, >=, == and !=.

- *Note: Pointer comparison compares two pointer addresses towhich they point to, instead of comparing their values.*

- Pointer comparisons are less used when compared to pointer arithmetic. However, I frequently use pointer comparison when dealing with arrays.

- Pointer comparisons are useful,

If you want to check if two pointer points to same location. For example,

```
int main()
{
    int num = 10;
    int *ptr1 = &num;    // ptr1 points to num
    int *ptr2 = &num;    // ptr2 also points to num
    if(ptr1 == ptr2)
    {
        // Both pointers points to same memory location
        // Do some task
    }
    return 0;}
```

Pointer to pointer

- Pointers are used to store the address of other variables of similar datatype. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable or **Double Pointer**
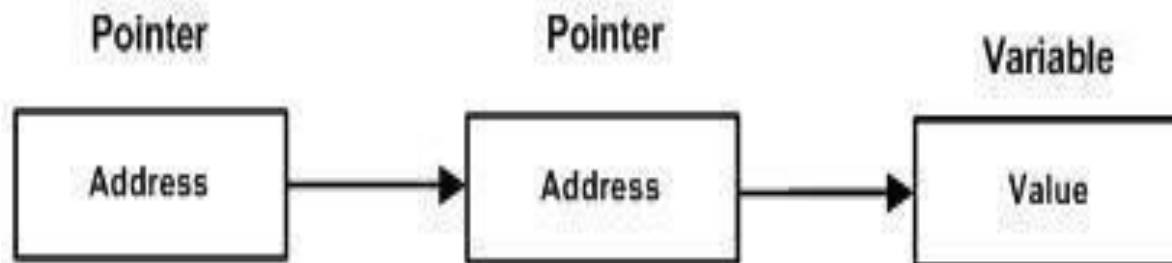
Syntax:

    int **p1;

- Here, we have used two indirection operator(*) which stores and points to the address of a pointer variable i.e, int *. If we want to store the address of this (double pointer) variable p1, then the syntax would become:

- **Simple program to represent Pointer to a Pointer**

```c
#include <stdio.h>
int main()
{
int a = 10;
   int *p1;        //this can store the address of variable a
   int  **p2;

   /*
      this can store the address of pointer variable p1 only.
      It cannot store the address of variable 'a'
    */
   p1 = &a;
   p2 = &p1;
```

```c
printf("Address of a = %u\n", &a);
    printf("Address of p1 = %u\n", &p1);
    printf("Address of p2 = %u\n\n", &p2);
// below print statement will give the address of 'a'
    printf("Value at the address stored by p2 = %u\n", *p2);
    printf("Value at the address stored by p1 = %d\n\n", *p1);
    printf("Value of **p2 = %d\n", **p2); //read this *(*p2)
    /*
        This is not allowed, it will give a compile time error-
        p2 = &a;
        printf("%u", p2);
    */ return 0;}
```

➢While dereferencing a void or Generic pointer, the C compiler does not have any clue about type of value pointed by the void pointer.

➢Hence, dereferencing a void pointer is illegal in C. But, a pointer will become useless if you cannot dereference it back.

➢ To dereference a void pointer you must typecast it to a valid pointer type.

void or Generic pointer

- **void** or **Generic** <u>pointer arithmetic</u> is illegal in C programming, due to the absence of type. However, some compiler supports void pointer arithmetic by assuming it as a char pointer.

- To perform pointer arithmetic on void pointer you must first typecast to other type.

Here is some code using a void pointer:

```c
#include <stdio.h>
#include<string.h> intmain()

{
int       num[3] = {10,20,30};
char name[30] = "Welcome to C World";
int *pint = NULL;
void *pvoid = NULL; int i;
pint = &num;
```

Example for array of pointer

```
char *name[3] = {
    "Adam",
    "chris",
    "Deniel"
};
//Now lets see same array without using pointer
char name[3][20] = {
    "Adam",
    "chris",
    "Deniel"
};
```

1

Array of Pointers

- In the second approach memory wastage is more, hence it is preferred to use pointer in such cases.

- When we say memory wastage, it doesn't means that the strings will start occupying less space, no, characters will take the same space, but when we define array of characters, a contiguos memory space is located equal to the maximum size of the array, which is a wastage, which can be avoided if we use pointers instead

Advantages

- An array of pointers in C: it sets each pointer in one array to point to an integer in another and then prints the values of the integers by dereferencing the pointers (printing the value in memory that the pointers point to).

Using Pointer

1st	⟶ (A)d a m
2nd	⟶ (C)h r i s
3rd	⟶ (D)e n i e l
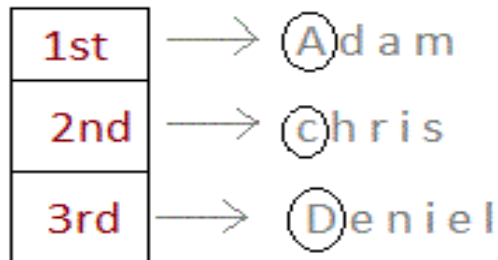
char* name[3]

Only 3 locations for pointers, which will point to the first character of their respective strings.

Without Pointer

A	d	a	m		
c	h	r	i	s	
D	e	n	i	e	l

char name[3][20]

extends till 20 memory locations

Example

```c
#include <stdio.h>

int main()
{
    int i;
    int a[5] = {1, 2, 3, 4, 5};
    int *p = a;         // same as int*p = &a[0]
    for (i = 0; i < 5; i++)
    {
        printf("%d", *p);
        p++;
    }

    return 0;
}
```

Arrays in C

All elements of same type – homogenous

Unlike Java, array size in declaration

```
int array[10];
int b;


array[0]   = 3;

array[9]   = 4;

array[10] = 5;

array[-1] = 6;
```

Compare: C: int array[10];

Java: int[] array = new int[10];

First element (index 0)

Last element (index size - 1)
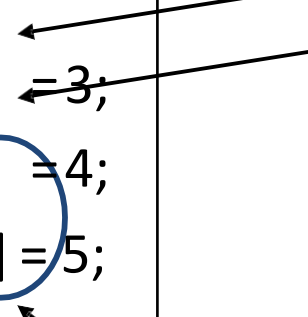
No bounds checking!

Allowed – usually causes no *obvious* error array[10]

may overwrite b

Array Representation

Homogeneous → Each element same size – s bytes

- – An array of m data values is a sequence of m×s bytes
- – Indexing: 0th value at byte s×0, 1st value at byte s×1, ...

m and s are <u>not</u> part of representation

- – Unlike in some other languages
- – s known by compiler – usually irrelevant to programmer
- – m often known by compiler – if not, must be saved by programmer

0x1008	**a[2]**
0x1004	**a[1]**
0x1000	**a[0]**

```
int a[3];
```
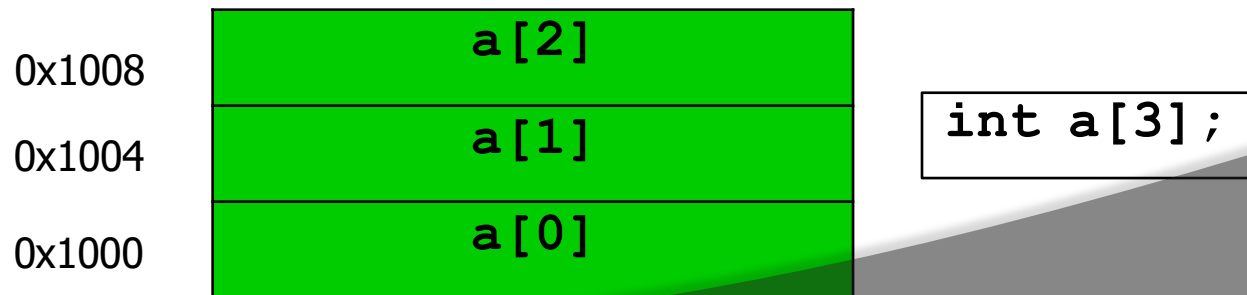
Memory Addresses

Storage cells are typically viewed as being byte-sized

–Usually the smallest addressable unit of memory

- Few machines can directly address bits individually

–Such addresses are sometimes called *byte-addresses*

Memory is often accessed as words

–Usually a word is the largest unit of memory access by a single machine instruction

- CLEAR's word size is 8 bytes (= sizeof(long))

–A *word-address* is simply the byte-address of the word's first byte

Pointers

Special case of bounded-size natural numbers

  – Maximum memory limited by processor word-size

  – 2^{32} bytes = 4GB, 2^{64} bytes = 16exabytes

A pointer is just another kind of value

  – A basic type in C

```
int *ptr;
```

The variable "ptr" stores a pointer to an "int".

Pointer Operations in C

Creation

        & *variable*   Returns variable's memory address

 Dereference

        * *pointer*   Returns contents stored at address

Indirect assignment

        ***pointer = val*   Stores value at address

Of course, still have…

Assignment

        pointer = ptr   Stores pointer in another variable

Using Pointers

```
int i1;
int i2; int *ptr1; int
*ptr2;
i1 = 1;
i2 = 2;
ptr1 = &i1; ptr2 =
ptr1;
*ptr1 = 3;
i2 = *ptr2;
```
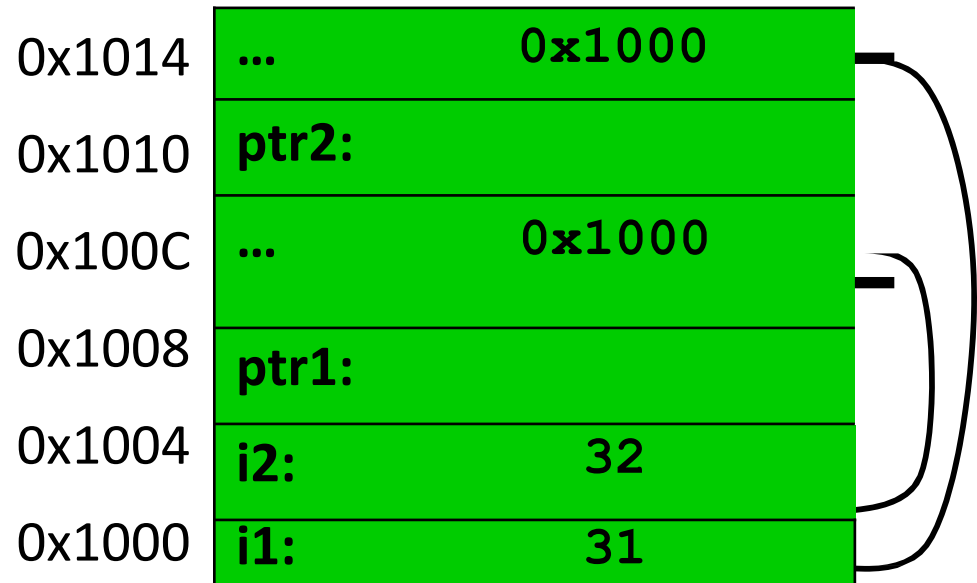
0x1014	...	**0x1000**
0x1010	**ptr2:**	
0x100C	...	**0x1000**
0x1008	**ptr1:**	
0x1004	**i2:**	32
0x1000	**i1:**	31

Using Pointers (cont.)

```
int  int1   = 1036;   /* some data to point to    */
int  int2   = 8;


int *int_ptr1 = &int1;   /* get addresses of data    */
int *int_ptr2 = &int2;



*int_ptr1 = int_ptr2;


*int_ptr1 = int2;
```

What happens?

Type check warning: `int_ptr2` **is not an** `int`

int1 becomes 8

Using Pointers (cont.)

```
int  int1    = 1036;   /* some data to point to    */
int  int2    = 8;


int *int_ptr1 = &int1;   /* get addresses of data    */
int *int_ptr2 = &int2;



int_ptr1 = *int_ptr2;



int_ptr1 = int_ptr2;
```

What happens?

Type check warning: `*int_ptr2` **is not an** `int *`

Changes `int_ptr1` **– doesn't change** `int1`

Arrays and Pointers

Dirty "secret":

Array name ≈ a pointer to the initial (0th) array element

$$a[i] \equiv *(a + i)$$

An array is passed to a function as a pointer

— The array size is lost!

Usually bad style to interchange arrays and pointers

—   Avoid pointer arithmetic!

Passing arrays:
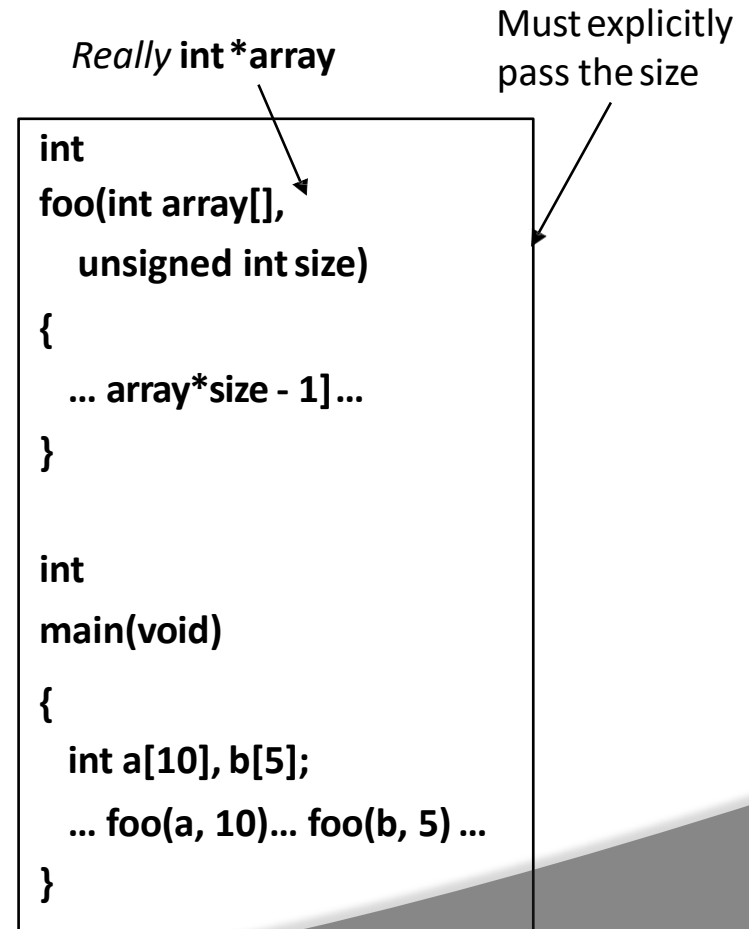
Really **int *array**

Must explicitly pass the size

```
int
foo(int array[],
    unsigned int size)
{
   … array*size - 1] …
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
}
```

```
int  i;
int  array[10];

for (i = 0; i <  10; i++)
{
   array[i] = …;
}
```

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

for (p = array; p < &array[10]; p++)
{
  *p = …;
}
```

These two blocks of code are functionally equivalent

Pointers as function arguments

Pointer preliminaries:

Pointer Definition:

➢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.

Function basics:

➢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.
➢A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

Function Parameter passing methods:
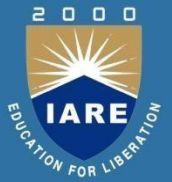
Call by value

➢This method copies the actual value of an   into the formal argument parameter of the function.

➢In this case, changes made to   the parameter inside the   function have no effect on the argument.

Syntax:

Datatype function_name(datatype variable_name);

Call by value example:

```c
#include <stdio.h>
void swap(int i, int j)
{
int t; t=i; i=j;  j=t;
}
void main()
{
int a,b;
a=5;      b=10;
printf("%d %d\n", a, b);
swap(a,b);
printf("%d %d\n", a, b);
}
```

Function Parameter passing methods:
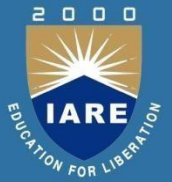
Call by reference:

➤This method copies the address of an argument into the formal parameter.

➤Inside the function, the address is used to access the actual argument used in the call.

➤This means that changes made to the parameter affect the argument.

Syntax:

datatype function_name(datatype *variable_name);

Call by reference Example

```c
#include <stdio.h> void swap(int
*i, int *j)
{
int t; t = *i;
*i = *j;
*j = t;
}
```

```c
void main()
{
int a,b; a=5;  b=10;
printf("%d %d\n",a,b);
swap(&a,&b);  printf("%d
%d\n",a,b);
}
```

Pointers as function arguments:

- ➢ When we pass a pointer as an argument instead of a variable then the address of the variable is passed instead of the value.

- ➢ So any change made by the function using the pointer is permanently made at the address of passed variable.

Functions used in Dynamic Memory allocation

malloc()

Allocates requested size of bytes and returns a pointer first byte of allocated space

Syntax of malloc()

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

Example for passing pointer to function

```c
#include <stdio.h>
void salaryhike(int *var, int b)
{
*var = *var+b;
}
int main()
{
int salary=0, bonus=0;
printf("Enter the employee current salary:");
scanf("%d", &salary);
printf("Enter bonus:");
scanf("%d", &bonus);
salaryhike(&salary, bonus);
printf("Final salary: %d", salary);
return 0;
}
```

Functions of returning pointers

➤ It is also possible for functions to return a function pointer as a value.

➤ This ability increases the flexibility of programs.

➤ In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function.

➤ Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends

Pointer to functions

➢It is possible to declare a pointer pointing to a function which canthen

be used as an argument in another function.

➢A pointer to a function is declared asfollows

type (*pointer-name)(parameter);

➢A function pointer can point to a specific function when it is assigned

the name of that function.

int sum(int, int); int (*s)(int, int); s = sum;

Pointer to functions

Example of   function pointers as returnedvalues

#include <stdio.h>

int* larger(int*, int*);

void main()
{
int a = 15; int b = 92; int *p;
p = larger(&a, &b);
printf("%d is larger",*p);
}

int* larger(int *x, int *y)
{
if(*x > *y) return x;
else
return y;
}

Pointer to functions

➢Either use argument with functions. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function as well.

1. use static local variables inside the function and return them.

2. As static variables have a lifetime until the main() function exits, therefore they will be available throughout the program.

Dynamic memory allocation

➢Dynamic memory management refers to manual memory management.

➢This allows to obtain more memory when required and release it when not necessary.

Functions used in Dynamic Memory allocation

calloc()

Allocates space for an array elements, initializes to zero and then returns a pointer to memory

Syntax of calloc()

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

free()

dellocate the previously allocated space syntax of free()
free(ptr);

realloc()

Change the size of previously allocated space

MODULE – V
FILE HANDLING AND BASICALGORITHMS

CLOs	Course Learning Outcome
CLO16	Explain the concept of file system for handling data storage and apply it for solving problems.
CLO17	Differentiate text files and binary files and write the simple C programs using file handling functions.
CLO18	Apply the concepts to solve real-time applications using the features of C language.
CLO19	Gain knowledge to identify appropriate searching and sorting techniques by calculating time complexity for problem solving.
CLO20	Possess the knowledge and skills for employability and to succeed in national and international level competitive examinations.

Files and Streams

File:

➤A file represents a sequence of bytes on the disk where a group of related data is stored.

➤File is created for permanent storage of data. It is a ready made structure.

Stream:

> In C, the **stream** is a common, logical interface to the various devices that comprise the computer.

> In its most common form, a **stream** is a logical interface to a **file**. As C defines the term "file", it can refer to a disk file, the screen, the keyboard, a port, a file on tape, and so on.

> Although files differ in form and capabilities, all **streams** are the same. The **stream** provides a consistent interface and to the programmer one hardware device will look much like another.

➤ A **stream** is linked to a file using an **open operation**. A **stream** is disassociated from a file using a **close operation**.

➤ The current location, also referred to as the current position, is the location in a file where the next file access will occur.

➤ There are two types of **streams: text** (used with ASCII characters some character translation takes place, may not be one-to-one correspondence between stream and what's in the file) and **binary** (used with any type of data, no character translation, one-to-one between stream and file).

File I/O Streams in C Programming Language

- In C all input and output is done with streams

- Stream is nothing but the sequence of bytes of data

- A sequence of bytes flowing into program is called input stream

- A sequence of bytes flowing out of the program is called output stream

- Use of Stream make I/O machine independent.

Predefined Streams:

stdin	**Standard Input**
stdout	Standard Output
stderr	Standard Error

Standard Input Stream Device:

1. **stdin** stands for (**St**an**d**ard**In**put)

2. Keyboard is **standard input device**.

3. Standard input is data **(Often Text) going into a program**.

4. The program requests data transfers by use of the read operation.

5. Not all programs require input.

Standard Output Stream Device:

1. **stdout** stands for (**St**an**d**ard**Out**put)

2. Screen(Monitor) is **standard output device**.

3. Standard output is data **(Often Text) going out from a program**.

4. The program sends data to output device by using write operation.
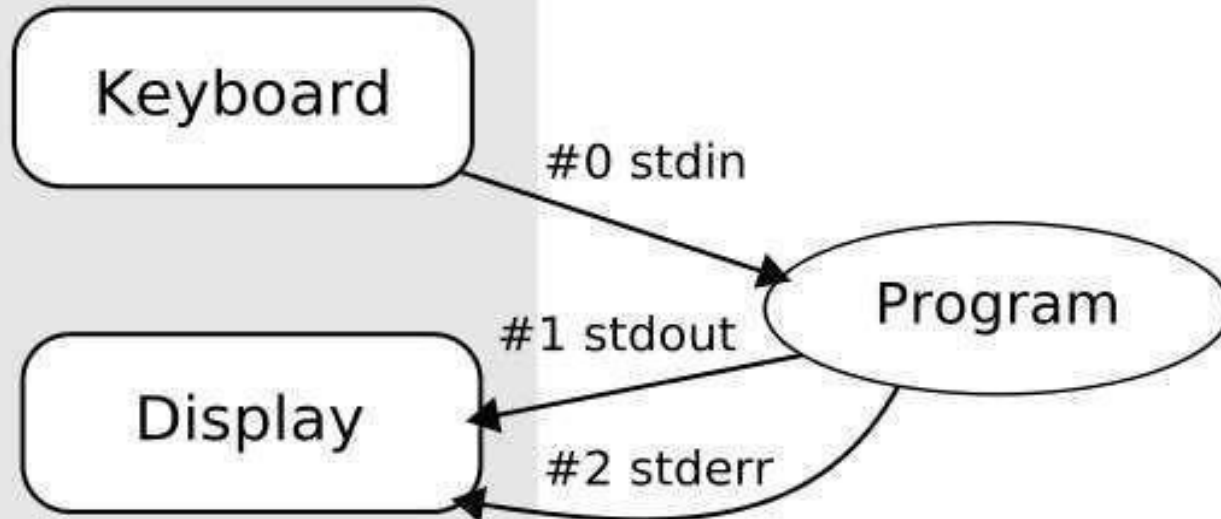
Difference between Standard Input and Standard Output Stream Device:

Point	Std i/p Stream Device	Standard o/p Stream Device
Stands For	Standard Input	Standard Output
Example	Keyboard	Screen/Monitor
Data Flow	Data (Often Text) going into a program	data (Often Text) going out from a program
Operation	Read Operation	Write Operation

Some Important Summary

Point	Input Stream	Output Stream
Standard Device 1	Keyboard	Screen
Standard Device 2	Scanner	Printer
IO Function	scanf and gets	printf and puts
IO Operation	Read	Write
Data	Data goes from stream	data comes into stream

File Operations

➤ In C, you can perform four major operations on the file, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Working with files

Writing a file

When working with files, you need to declare a pointer of type file.

This declaration is needed for communication between the   file and program.

FILE *fp;

Opening a file - for creation and edit

Opening a file is performed using the library function in the **"stdio.h"** header file: fopen().

The syntax for opening a file in standard I/O is:

fp = fopen("fileopen","mode")

For Example:

fopen("newprogram.txt","w");

Opening Modes in Standard I/O

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.

1

Readind and writing a text file

For reading and writing to a text file, we use the functions fprintf() and fscanf().

They are just the file versions of printf() and scanf(). The only difference is that, fprint and fscanf expects a pointer to the structure FILE.

We can also use fgetc function to read the contents of a file.

Syntax:

Ch=fgetc(fp);

Closing a file

The file that is opened should be closed after work is over we   need to create close the file after reading or writing.

Syntax:

Fclose(fp);
Fcloseall();// for multiple files

Files

Programs and data are stored on disk in structures called files

Examples

Turbo C++ - binary file Word 4.0 - binary file lab1.c - text file

lab1.data - text file   term-paper - text file

Text Files

➢ All files are coded as long sequences of bits (0s and 1s)

➢ Some   files   are   coded   as   sequences   values
   of   ASCII   character (referred to as text files)

   – files are organized   as bytes, with each byte being an ASCII
     character

➢ Other files are generally referred to as binaryfiles

File Terms

➢ Buffer - a temporary storage area used to transfer data back and

forth between memory and auxiliary storage devices

➢ Stream   - files are manipulated in C with streams, a stream is a

   mechanism that is connected to a file that allows you to access one

   element at a time

File Pointers

- ➢ Each stream in C is manipulated with the filepointertype

- ➢ FILE *stream

  - FILE is a type containing multipleparts

    - file for stream, current element in file, etc.

  - FILE * is the address where the FILE type is located inmemory

  - FILEs always manipulated as FILE*

fopen Command

Syntax: fopen("*FileName*","*mode*");

- File Name is an appropriate name for a file on the computer youare

  working on, example: "C:\My Files\lab.dat"

- Mode indicates the type of stream:

"r" - file is opened for reading characters

"w" - file is opened for writing characters (existing file deleted)

"a" - file opened for writing characters (appended to the end of

the existing file)

fclose Command

Syntax:

fclose(*FilePointer*)

- The file pointer must be a stream opened using fopen (that remains open)

- fclose returns

  - 0 if the the fclose command is successful

  - special value EOF if the fclose command is unsuccessful

Open/Closing File

```c
int main() {
FILE *stream;
if ((stream = fopen("lab.data","r")
== NULL) {
printf("Unable to open lab.data\n");
return(1);
}
/* Read data from lab.data using FILE *
variable stream */
if (fclose(stream) == EOF) , printf("Error
closing lab.data\n");  return(2);
}
}
```

fprintf Command

Syntax:

fprintf(filep, "Format", ValueList);

- Works similarly to printf, but data sent to file rather than screen

– printf("Format",ValueList) is a shorthand for

fprintf(stdout,"Format",ValueList)

- fprintf returns the number of characters printed or EOF (-1) if an error occurs

- File pointer should be write/append stream

fscanf Command

Syntax:

- fscanf(filep, "Format", AddrList);

➢ Works similarly to scanf, but data received from file rather than keyboard

scanf("Format",AddrList) is a shorthand for

fscanf(stdin,"Format",AddrList)

fscanf returns the number of successful data conversions or EOF if end-of-file reached

- File pointer should be a read stream

fscanf/fprintf Example

```c
if ((ins = fopen("part.data","r")) == NULL)
{
printf("Unable to open part.data\n"); return(-1);
}
if ((outs = fopen("sumpart.data","w")) == NULL)
{
printf("Unable to open sumpart.data\n");
 return(-1);
}
while (fscanf(ins,"%d%d%d%d",&id,&p1,&p2,&p3) == 4)
fprintf(outs,"%3d %3d\n",id,(p1 + p2 + p3));
fclose(ins);
fclose(outs);
```

File opening modes

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

FILE *fptr;

Opening a file is performed using the library function in the **"stdio.h"** header file: fopen().

The syntax for opening a file in standard I/O is:

ptr = fopen("fileopen","mode")

In C Programming we can open file in different modes such as reading mode,writing mode and appending mode depending on purpose of handling file. Following are the different Opening modes of File :

File opening modes

Opening Mode	Purpose	Previous Data
Reading	File will be opened just for reading purpose	Retained
Writing	File will be opened just for writing purpose	Flushed
Appending	File will be opened for appending some thing in file	Retained

File Opening Mode Chart

Mode	Meaning	fopen Returns if FILE -	
		Exists	Not Exists
r	Reading	–	NULL
w	Writing	Over write on Existing	Create New File
a	Append	–	Create New File
r+	Reading + Writing	New data is written at the beginning overwriting existing data	Create New File
w+	Reading + Writing	Over write on Existing	Create New File
a+	Reading + Appending	New data is appended at the end of file	Create New File

Types of file

File Type	Extension
C Source File	.c
Text File	.txt
Data File	.dat

Example to Open a File

```c
#include<stdio.h>
int main()
{
FILE *fp;
char ch;
fp = fopen("INPUT.txt","r")

//Open file in Read mode

fclose(fp);

// Close File after Reading
return(0);
}
```

Different modes:

➤ Reading Mode

        fp = fopen("hello.txt","r");

➤ Writing Mode

        fp = fopen("hello.txt","w");

➤ Append Mode

        fp = fopen("hello.txt","a");

Another Example To Open File:

```c
#include<stdio.h>
void main()
{
FILE *fp; char ch;
fp = fopen("INPUT.txt","r"); // Open file in Read mode

while(1)
{
ch = fgetc(fp); // Read a Character
if(ch == EOF ) // Check for End of File break ;

printf("%c",ch);
}
fclose(fp); // Close File after Reading
}
```

Ways of Detecting End of File

In Text File :

➢Special Character EOF denotes the end of File

•As soon as Character is read, End of the File can be   detected.
•EOF is defined in stdio.h
•Equivalent value of EOF is   -1
•Printing Value of EOF :
•void main()
{
printf("%d", EOF);
}

Ways of Detecting End of File (Contd..)

In Binary File :

➢ feof function is used to detect the end of file
➢ It can be used in text file
➢ feof Returns TRUE if end of file is reached

Syntax :
int feof(FILE *fp);

Way of Writing feof Function :
Way1: with if statement :

```
if( feof(fptr) == 1 ) // as if(1) is TRUE
printf("End of File");
```
Way 2 : with While Loop
•while(!feof(fptr))
{
--- - --}

File reading with fsacnf function

```c
#include <stdio.h> main()
{
FILE *fp;
char buff[255];
fp = fopen("/tmp/test.txt", "r");
fscanf(fp, "%s", buff);
printf("1 : %s\n", buff );
fgets(buff, 255, (FILE*)fp);
printf("2: %s\n", buff );
fgets(buff, 255, (FILE*)fp);
 printf("3: %s\n", buff );  fclose(fp);
}
```

File i/o functions

When working with files, we need to declare a pointer of type file.

This declaration is needed for communication between the file and program.

FILE *fptr;

C provides   functions that helps to perform basic file operations.

fopen()   :
create a new file or open a existing file

*fp = FILE *fopen(const char *filename, const char *mode);

fclose() :

function is used to close an already opened file.

int fclose(FILE *fp);

➢getc() and putc() are the simplest functions which can be used to read and write individual characters to a file.

➢fprintf() function directly writes into the file

➢fscanf() function reads from the file, which can then be printed on the console using standard printf() function

fseek() Contd

Stream: pointer to a file.

Offset: Number of bytes or characters from the origin.

Origin: The original position is set here. From this position, using the offset, the file pointer is set to a new position. Generally, three positions are used as origin:

SEEK_SET - Beginning of file
SEEK_CUR - Current position of the file pointer
SEEK_END - End of file

Return

Type: Integer
Value: On success Zero(0)
On failure Non-Zero

fseek() example

Code:
int main () {
FILE * fp;

fp = fopen ("file.txt" , "w");
if (fp==NULL)
printf ("Error in opening file"); else
 { fputs ("I am supporter of France." , fp); fseek (fp , 18 , SEEK_SET);
fputs ("Brazil" , fp);
fseek(fp , 0 , SEEK_CUR); fputs (" and Portugal" , fp); fclose (fp);
}
return 0;}

- Then using SEEK_SET and offset, the word France is replaced by Brazil. Then by using SEEK_CUR, and position is appended with the string.

ftell()

- **ftell()**

  It is used to get current position of the file pointer. The function prototype
  is:

  long int ftell (FILE * stream);

  Parameters

  stream: Pointer to a file.

  Return

  Type: long integer.
  Value: On success value of current position or offset bytes
  On failure -1. System specific error no isset

ftell() Example

Code:

```c
int main () {
FILE * fp;
long int len;
fp = fopen ("file.txt","r");
if (fp==NULL)
printf ("Error in opening file"); else {
fseek (fp, 0, SEEK_END);
len=ftell (fp);
fclose (fp);
printf ("The file contains %ld characters.\n",len);
}
return 0;
}
```

ftell() Example Contd.

•In this example, file.txt is opened and using fseek(), file pointer is set to the end of the file.

•Then, ftell() is used to get the currentposition, i.e. offset from the beginning of the file.

rewind()

rewind()

It is used to set the file pointer at the beginning of the file. Function prototype:

void rewind (FILE * stream);

Parameters

stream: Pointer to a file.

In any stage of the program, if we need to go back to the starting point of the file, we can use rewind() and send the file pointer as the parameter.

```c
int main () {
int n;
FILE * fp;
fp = fopen ("file.txt","w+");

if (fp==NULL)

printf ("Error in opening file");
 else { fputs ("France is my favorite team",fp);
 rewind (fp);
fputs("Brazil",fp);
fclose (fp);
}
r}eturn 0;
```

File position functions

➢ The C library function **int fseek(FILE *stream, long int offset, int whence)** sets the file position of the **stream** to the given **offset**.

➢ Following is the declaration for fseek()function.

int fseek(FILE *stream, long int offset, int whence)

Example

```c
#include <stdio.h>
 int main ()
{
FILE *fp;
fp = fopen("file.txt","w+");
fputs("This   is tutorialspoint.com", fp);
 fseek( fp,   7, SEEK_SET );
fputs(" C Programming Language", fp);
 fclose(fp);
return(0);
}
```

➢ Let us compile and run the above program that will create a file **file.txt** with the following content. Initially program creates the file and writes *This is tutorialspoint.com* but later we had reset the write pointer at 7th position from the beginning and used puts() statement which over-write the file with the following content−

Output : This is C Programming Language

Command Line Arguments

➢ It is possible to pass some values from the command line to your C programs when they are executed.

➢ These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code

➢ The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program.

➢ Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly–

Example Program

```c
#include <stdio.h>
int main( int argc, char *argv[] )
 {
if( argc == 2 )
 {
 printf("The argument supplied is %s\n", argv[1]);
 }
 else if( argc > 2 ) {
  printf("Too many arguments supplied.\n");
 }
else {
 printf("One argument expected.\n");    }
}
```

➢When the above code is compiled and executed without passing any argument, it produces the following result.

$./a.out

One argument expected

➢ It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

Searching

- Searching is one of the most common problems that arise in computing. Searching is the algorithmic process of finding a particular item in a collection of items. A search typically answers either True or False as to whether the item is present. On occasion it may be modified to return where the item is found. Search operations are usually carried out on a key field.

- Well, to search an element in a given array, there are two popular algorithms available:

➢ Linear Search
➢ Binary Search

Linear Search

- Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

- It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return -1.

- Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

➤ **Features of Linear Search Algorithm**

➤ It is used for unsorted and unordered small list of elements.

➤ It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.

➤ It has a very simple implementation.

Binary Search

➢ Binary Search is used with sorted array or list. In binary search, we follow the following steps:

➢ We start by comparing the element to be searched with the element in the middle of the list/array.

➢ If we get a match, we return the index of the middle element.

➢ If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.

➢ If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.

➢ If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

1

➢ **Features of Binary Search**

➢ It is great to search through large sorted arrays.

➢ It has a time complexity of **O(log n)** which is a very good time complexity

➢ It has a simple implementation.

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 elementsas follow;
  10, 3 ,6 12, 4, 17, 5, 9

- After shorting value must be;
  3, 4, 5, 6, 9, 10, 12, 17

➢ Above value sort by apply any sorting technique. C language have following technique to sort values;

1. Bubble Sort
2. Selection Sort
3. Insertion Sort

➢ **Bubble Sort in C**

1. 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 settle down.

2. Both worst case and average case complexity is O (n^2).

➢ **Selection Sort in C**

1. One of the simplest techniques is a selection sort.

2. As the name suggests, selection sort is the selection of an element and keeping it in sorted order. In selection sort, the strategy is to find the smallest number in the array and exchange it with the value in first position of array.

3. Now, find the second smallest element in the remainder of array and exchange it with a value in the second position, carry on till you have reached the end of array. Now all the elements have been sorted in ascending order.ep

➢ The selection sort algorithm is performed using following steps…

1. **Step 1:** Select the first element of the list (i.e., Element at first position in the list).
1. **Step 2:** Compare the selected element with all other elements in the list.
2. **Step 3:** For every comparision, if any element is smaller than selected element (for Ascending order), then these two are swapped.
3. **Step 4:** Repeat the same procedure with next position in the list till the entire list is sorted.

> **Insertion Sort in C**

- The insertion sort inserts each element in proper place. The strategy behind the insertion sort is similar to the process of sorting a pack of cards.

- You can take a card, move it to its location in sequence and move the remaining cards left or right as needed.

- In insertion sort, we assume that first element A[0] in pass 1 is already sorted. In pass 2 the next second element A[1] is compared with the first one and inserted into its proper place either before or after the first element. In pass 3 the third element A[2] is inserted into its proper place and so on.