

# 18CSS101J – Programming for Problem Solving

## Unit I

## COURSE LEARNING RATIONALE (CLR)

*The purpose of learning this course is to:*

<b>CLR -1:</b>	Think and evolve a logically to construct an algorithm into a flowchart and a pseudocode that can be programmed
<b>CLR -2:</b>	Utilize the logical operators and expressions to solve problems in engineering and real-time
<b>CLR -3:</b>	Store and retrieve data in a single and multidimensional array
<b>CLR -4:</b>	Utilize custom designed functions that can be used to perform tasks and can be repeatedly used in any application
<b>CLR -5:</b>	Create storage constructs using structure and unions. Create and Utilize files to store and retrieve information
<b>CLR -6:</b>	Create a logical mindset to solve various engineering applications using programming constructs in C

## COURSE LEARNING OUTCOMES (CLO)

*At the end of this course, learners will be able to:*

<b>CLO -1:</b>	Identify methods to solve a problem through computer programming. List the basic data types and variables in C
<b>CLO -2:</b>	Apply the logic operators and expressions. Use loop constructs and recursion. Use array to store and retrieve data
<b>CLO -3:</b>	Analyze programs that need storage and form single and multi-dimensional arrays. Use preprocessor constructs in C
<b>CLO -4:</b>	Create user defined functions for mathematical and other logical operations. Use pointer to address memory and data
<b>CLO -5:</b>	Create structures and unions to represent data constructs. Use files to store and retrieve data
<b>CLO -6:</b>	Apply programming concepts to solve problems. Learn about how C programming can be effectively used for solutions

## LEARNING RESOURCES

S. No	TEXT BOOKS
1.	<i>Zed A Shaw, Learn C the Hard Way: Practical Exercises on the Computational Subjects You Keep Avoiding (Like C), Addison Wesley, 2015</i>
2.	<i>W. Kernighan, Dennis M. Ritchie, The C Programming Language, 2nd ed. Prentice Hall, 1996</i>
3.	<i>Bharat Kinariwala, Tep Dobry, Programming in C, eBook</i>
4.	<u><a href="http://www.c4learn.com/learn-c-programming-language/">http://www.c4learn.com/learn-c-programming-language/</a></u>

# **UNIT I**

## **INTRODUCTION**

Evolution of Programming & Languages - Problem Solving through Programming - Creating Algorithms - Drawing Flowcharts - Writing Pseudocode - Evolution of C language, its usage history - Input and output functions: `Printf` and `scanf` - Variables and identifiers - Expressions - Single line and multiline comments - Constants, Keywords - Values, Names, Scope, Binding, Storage Classes - Numeric Data types: **integer** -

# **UNIT I**

## **INTRODUCTION**

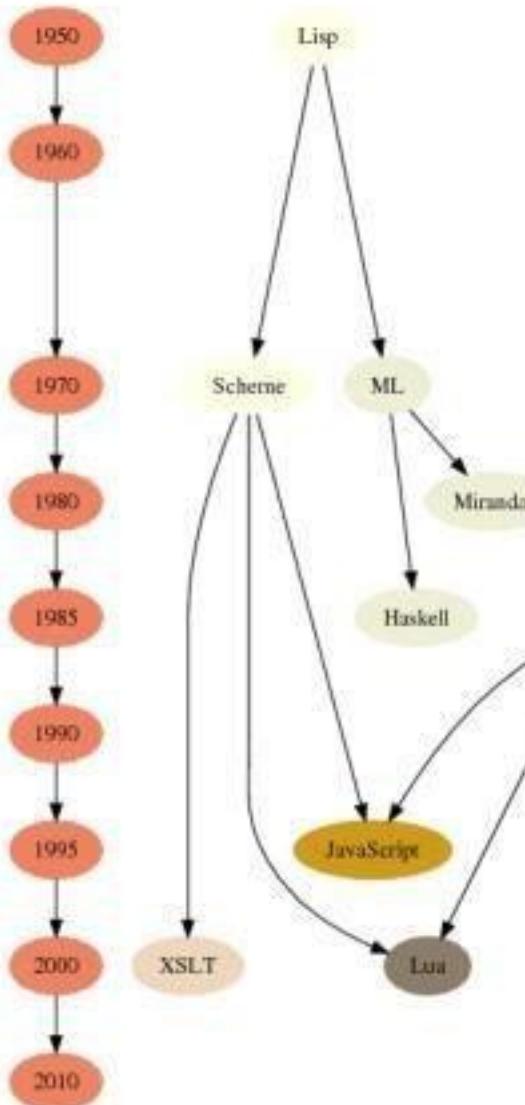
**floating point** - Non-Numeric Data types: **char** and **string** -  
Increment and decrement operator - Comma, Arrow and  
Assignment operator - Bitwise and **Sizeof** operator

# 1. 1 Evolution of Programming & Languages

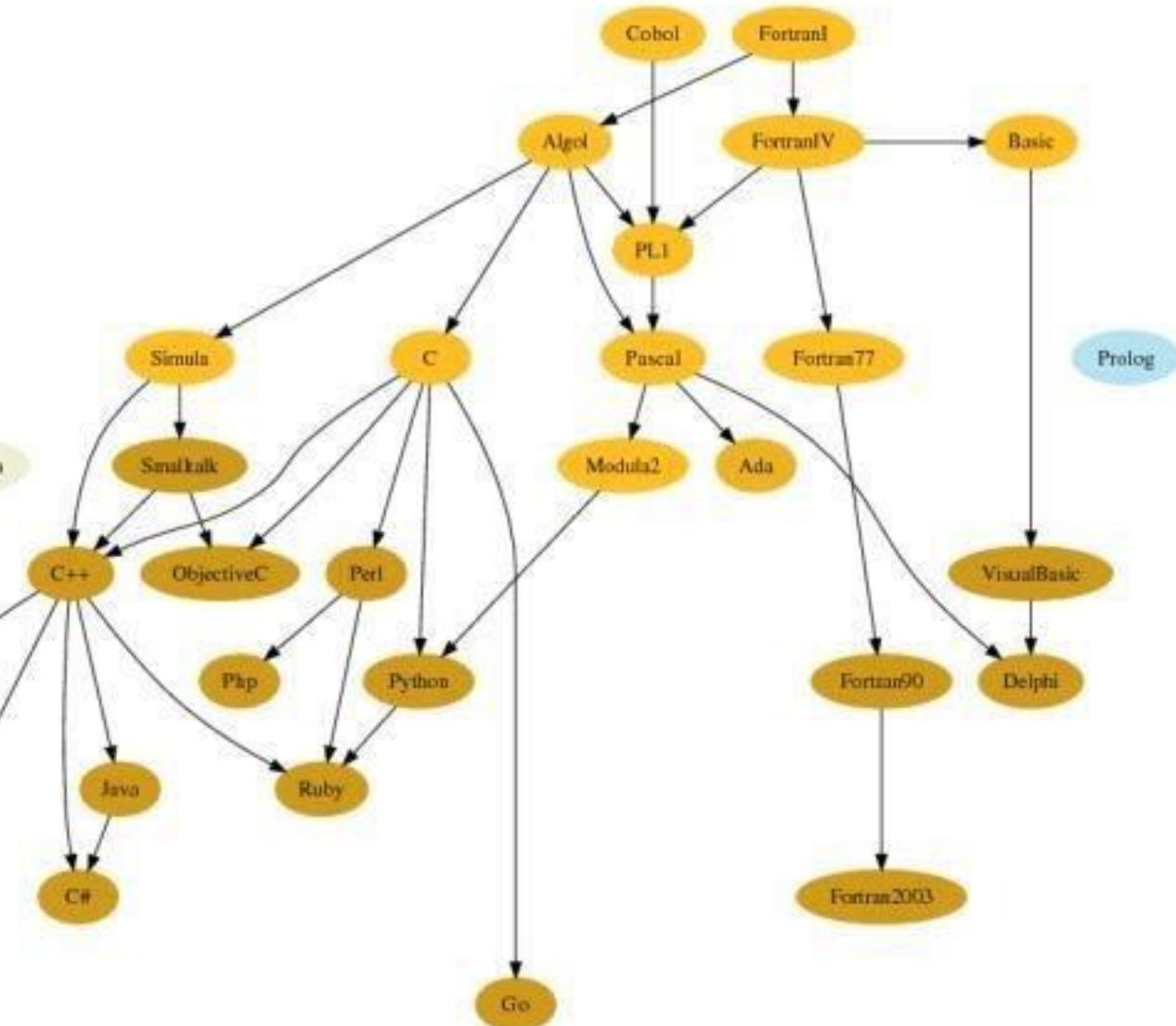
- ❑ A Computer needs to be given instructions in a programming language that it understands
- ❑ Programming Language
  - ❑ Artificial language that controls the behavior of computer
  - ❑ Defined through the use of syntactic and semantic rules
  - ❑ Used to facilitate communication about the task of organizing and manipulating information
  - ❑ Used to express algorithms precisely

# Evolution of programming languages

## Functional



## Imperative



# 1. 1 Evolution of Programming & Languages Contd...

Period	Programming Languages
1950's	Creation of high-level languages
1960's	Forth, Simula I, Lisp, Cobol
1970's	Pascal, C language
1980's	ML, Smalltalk, C++
1990's	Java, Perl, Python languages
2000	Internet Programming
2010	Concurrency and asynchronicity. JavaScript and Go language

## 1. 2 Problem Solving through Programming

□ ***Problem*** - Defined as any question, something involving doubt, uncertainty, difficulty, situation whose solution is not immediately obvious

□ ***Computer Problem Solving***

- Understand and apply logic
- Success in solving any problem is only possible after we have made the effort to understand the problem at hand
- Extract from the problem statement a set of precisely defined tasks

# 1. 2 Problem Solving through Programming Contd...

## i. *Creative Thinking*

- ❑ Proven method for approaching a challenge or opportunity in an imaginative way
- ❑ Process for innovation that helps explore and reframe the problems faced, come up with new, innovative responses and solutions and then take action
- ❑ It is generative, nonjudgmental and expansive
- ❑ Thinking creatively, a lists of new ideas are generated

# 1. 2 Problem Solving through Programming Contd...

## *ii. Critical Thinking*

- ❑ Engages a diverse range of intellectual skills and activities that are concerned with evaluating information, our assumptions and our thinking processes in a disciplined way so that we can think and assess information more comprehensively
- ❑ It is Analytical, Judgmental and Selective
- ❑ Thinking critically allows a programmer in making choices

# 1. 2 Problem Solving through Programming Contd...

- Imagine
- Invent
- Change
- Design
- Create

Creative Thinking



Critical Thinking



Problem-Solving

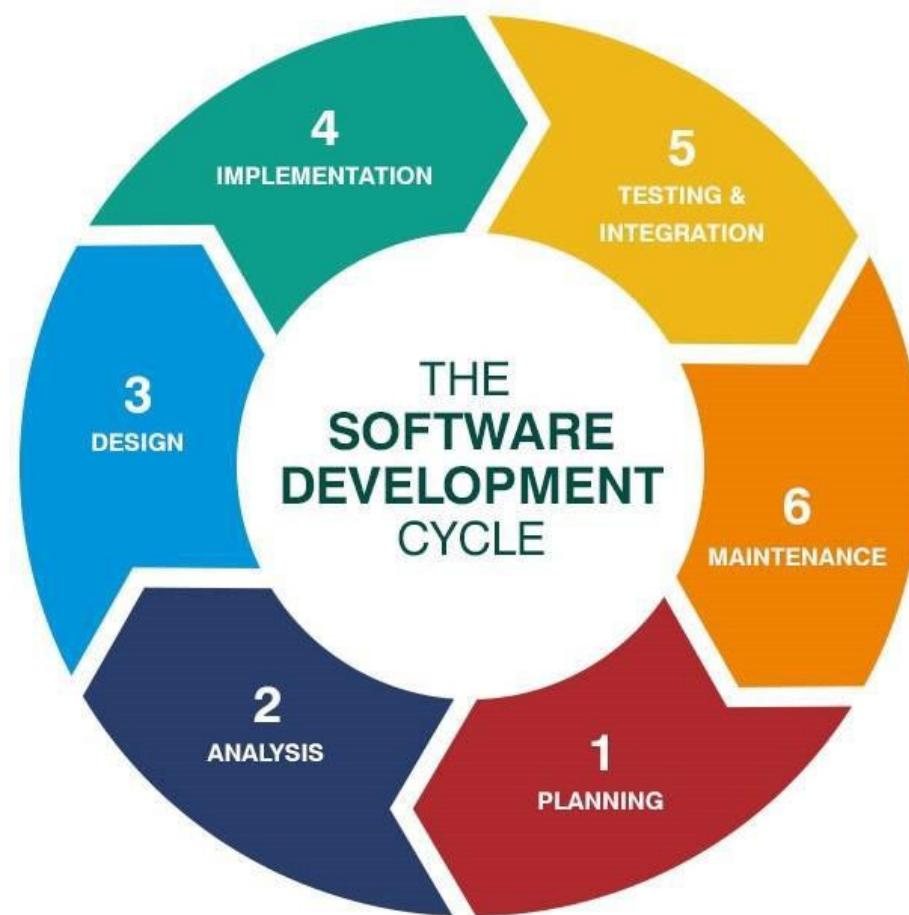
- Analyse
- Break down
- Compare
- Categorise
- List
- Sequence
- Rank

- Improve
- Design
- Refine
- Find
- Invent criteria to combine

## 1. 2 Problem Solving through Programming Contd...

- ***Program*** - Set of instructions that instructs the computer to do a task
- ***Programming Process***
  - a) *Defining* the Problem
  - b) *Planning* the Solution
  - c) *Coding* the Program
  - d) *Testing* the Program
  - e) *Documenting* the Program

# 1. 2 Problem Solving through Programming Contd...



# 1. 2 Problem Solving through Programming Contd...

- ❑ A typical programming task can be divided into two phases:

## i. *Problem solving phase*

- ❑ Produce an ordered sequence of steps that describe solution of problem this sequence of steps is called an *Algorithm*

## ii. *Implementation phase*

- ❑ Implement the program in some programming language

## ❑ *Steps in Problem Solving*

- a) Produce a general algorithm (one can use *pseudocode*)

## 1. 2 Problem Solving through Programming Contd...

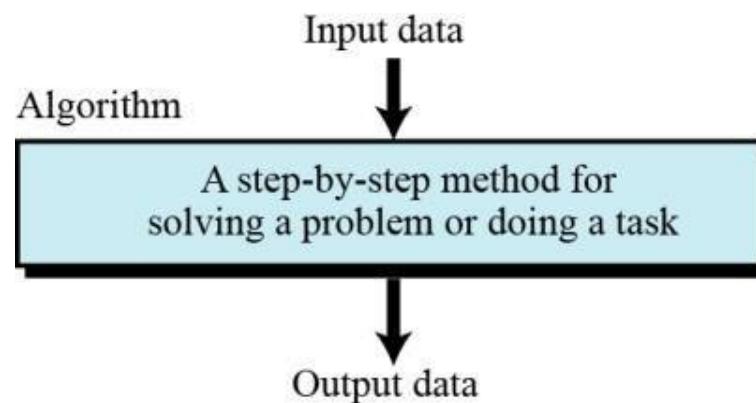
- b) Refine the algorithm successively to get step by step detailed ***algorithm*** that is very close to a computer language
- c) **Pseudocode** is an artificial and informal language that helps programmers develop algorithms
  - ❑ Pseudocode is very similar to everyday English

## 1. 3 Creating Algorithms

- An informal definition of an algorithm is:

i

Algorithm: a step-by-step method for solving a problem or doing a task.



## 1. 3 Creating Algorithms Contd...

- ❑ What are Algorithms for?
  - ❑ A way to communicate about your problem/solution with others
  - ❑ A possible way to solve a given problem
  - ❑ A "formalization" of a method, that will be proved
  - ❑ A mandatory first step before implementing a solution
- ❑ ***Algorithm Definition*** - “A finite sequence of unambiguous, executable steps or instructions, which, if followed would ultimately terminate and give the solution of the problem”

# 1. 3 Creating Algorithms

## ❑ Notations

- ❑ Starting point
- ❑ Step Numbers – Positions in Algorithm
- ❑ Incoming Information - Input
- ❑ Control Flow – Order of evaluating Instructions
- ❑ Statements
- ❑ Outgoing Information - Output
- ❑ Ending Point

## 1. 3 Creating Algorithms Contd...

### *Properties of an algorithm*

- Finite:** The algorithm must eventually terminate
- Complete:** Always give a solution when one exists
- Correct (sound):** Always give a correct solution

### *Rules of Writing an Algorithm*

- Be consistent
- Have well Defined input and output
- Do not use any syntax of any specific programming language

## 1. 3 Creating Algorithms Contd...

- ❑ Algorithm development process consists of five major steps
  - ❑ **Step 1:** Obtain a description of the problem
  - ❑ **Step 2:** Analyze the problem
  - ❑ **Step 3:** Develop a high-level algorithm
  - ❑ **Step 4:** Refine the algorithm by adding more detail
  - ❑ **Step 5:** Review the algorithm

## 1. 3 Creating Algorithms Contd...

### *Example*

#### □ **Problem**

- a) Develop an algorithm for finding the largest integer among a list of positive integers
- b) The algorithm should find the largest integer among a list of any values
- c) The algorithm should be general and not depend on the number of integers

## 1. 3 Creating Algorithms Contd...

### □ ***Solution***

- a) To solve this problem, we need an intuitive approach
- b) First use a small number of integers (for example, five), then extend the solution to any number of integers
- c) The algorithm receives a list of five integers as input and gives the largest integer as output

(12 8 13 9 11) **Input data**



Largest **12** [12] 12 8 13 9 11 List  
Step 1

Largest **12** [12] 12 8 13 9 11 List  
Step 2

Largest **13** [13] 12 8 13 9 11 List  
Step 3

Largest **13** [13] 12 8 13 9 11 List  
Step 4

Largest **13** [13] 12 8 13 9 11 List  
Step 5

FindLargest



(13) **Output data**

(12    8    13    9    11) **Input data**



Set Largest to the first number.

Step 1

If the second number is greater than Largest, set Largest to the second number.

Step 2

If the third number is greater than Largest, set Largest to the third number.

Step 3

If the fourth number is greater than Largest, set Largest to the fourth number.

Step 4

If the fifth number is greater than Largest, set Largest to the fifth number.

Step 5

FindLargest



(13) **Output data**

(12    8    13    9    11) **Input data**



Step 0 Set Largest to  $-\infty$

Step 1 If the current number is greater than Largest, set Largest to the current number.

•••

Step 5 If the current number is greater than Largest, set Largest to the current number.

FindLargest

(13 ) **Output data**

Input data ( $n$  integers)



Set Largest to  $-\infty$

Repeat the following step  $n$  times:

If the current integer is greater than Largest, set Largest to the current integer.

FindLargest

Largest

## 1. 3 Creating Algorithms Contd...

***Example 2:*** Print 1 to 20

- **Step 1:** Start
- **Step 2:** Initialize X as 0,
- **Step 3:** Increment X by 1,
- **Step 4:** Print X,
- **Step 5:** If X is less than 20 then go back to step 2.
- **Step 6:** Stop

## 1. 3 Creating Algorithms Contd...

### *Example 3*

**Convert Temperature from Fahrenheit ( $^{\circ}\text{F}$ ) to Celsius ( $^{\circ}\text{C}$ )**

- Step 1:** Start
- Step 2:** Read temperature in Fahrenheit
- Step 3:** Calculate temperature with formula  $\text{C}=5/9*(\text{F}-32)$
- Step 4:** Print C
- Step 5:** Stop

## 1. 3 Creating Algorithms Contd...

### *Example 4*

#### **Algorithm to Add Two Numbers Entered by User**

- **Step 1:** Start
- **Step2:** Declare variables num1, num2 and sum.
- **Step 3:** Read values num1 and num2.
- **Step 4:** Add num1 and num2 and assign the result to sum.  
$$\text{sum} \leftarrow \text{num1} + \text{num2}$$
- **Step 5:** Display sum Step 6: Stop

## 1. 3 Creating Algorithms Contd...

### □ Write an Algorithm to:

- 1) Find the Largest among three different numbers
- 2) Find the roots of a Quadratic Equation
- 3) Find the Factorial of a Number
- 4) Check whether a number entered is Prime or not
- 5) Find the Fibonacci Series

## 1. 4 Drawing Flowcharts

- Diagrammatic representation
- Illustrates sequence of operations to be performed
- Each step represented by a different symbol
  - Each Symbol contains short description of the Process
- Symbols linked together by arrows
- Easy to understand diagrams
- Clear Documentation
- Helps clarify the understanding of the process

# Flowchart Symbols

Flowcharts are used to illustrate algorithms in order to aid in the visualisation of a program.

Flowcharts are to be read top to bottom and left to right in order to follow an algorithms logic from start to finish. Below is an outline of symbols used in flowcharts.

## Terminator

### Terminator

Used to represent the Start and end of a program with the Keywords **BEGIN** and **END**.

## Process

### Process

An instruction that is to be carried out by the program.



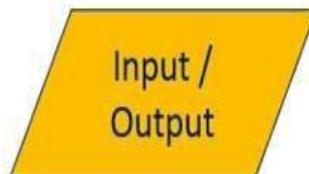
### Arrow

Indicates the flow of the algorithm pathways.



### Decision

Used to split the flowchart sequence into multiple paths in order to represent **SELECTION** and **REPETITION**.



### Input / Output

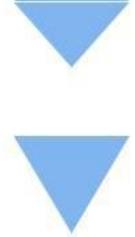
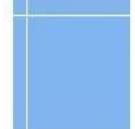
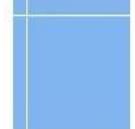
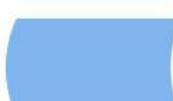
Used to represent **data entry** by a user or the **display** of data by the program.



### Subprogram

References another program within the program.

# 1.4 Drawing Flowcharts Contd...

	<b>Terminator</b> Indicates the beginning or end of a program flow in your diagram.		<b>Subroutine</b> Indicates a predefined (named) process, such as a subroutine or a module.		<b>Connector</b> Indicates an inspection point.		<b>Collate</b> Indicates a step that organizes data into a standard format.
	<b>Process</b> Indicates any processing function.		<b>Preparation</b> Indicates a modification to a process, such as setting a switch or initializing a routine.		<b>Off-page connector</b> Use this shape to create a cross-reference and hyperlink from a process on one page to a process on another page.		<b>Sort</b> Indicates a step that organizes items list sequentially.
	<b>Decision</b> Indicates a decision point between two or more paths in a flowchart.		<b>Display</b> Indicates data that is displayed for people to read, such as data on a monitor or projector screen.		<b>Off-page connector</b>		<b>Merge</b> Indicates a step that combines multiple sets into one.
	<b>Delay</b> Indicates a delay in the process.		<b>Manual input</b> Indicates any operation that is performed manually (by a person).		<b>Off-page connector</b>		<b>Database</b> Indicates a list of information with a standard structure that allows for searching and sorting.
	<b>Data</b> Can represent any type of data in a flowchart.		<b>Manual loop</b> Indicates a sequence of commands that will continue to repeat until stopped manually.		<b>Off-page connector</b>		<b>Internal storage</b> Indicates an internal storage device.
	<b>Document</b> Indicates data that can be read by people, such as printed output.		<b>Loop limit</b> Indicates the start of a loop. Flip the shape vertically to indicate the end of a loop.		<b>Or</b> Logical OR		<b>Summing junction</b> Logical AND
	<b>Multiple documents</b> Indicates multiple documents.		<b>Stored data</b> Indicates any type of stored data.				

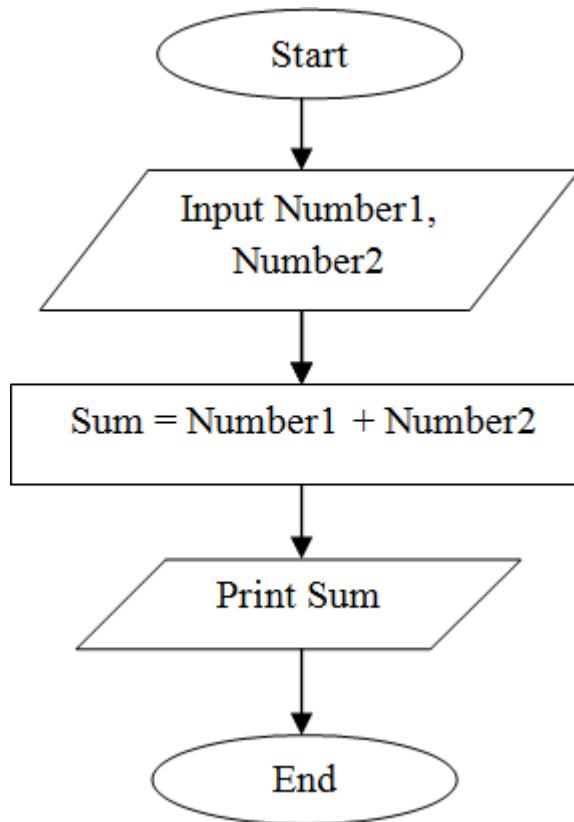
## 1. 4 Drawing Flowcharts Contd...

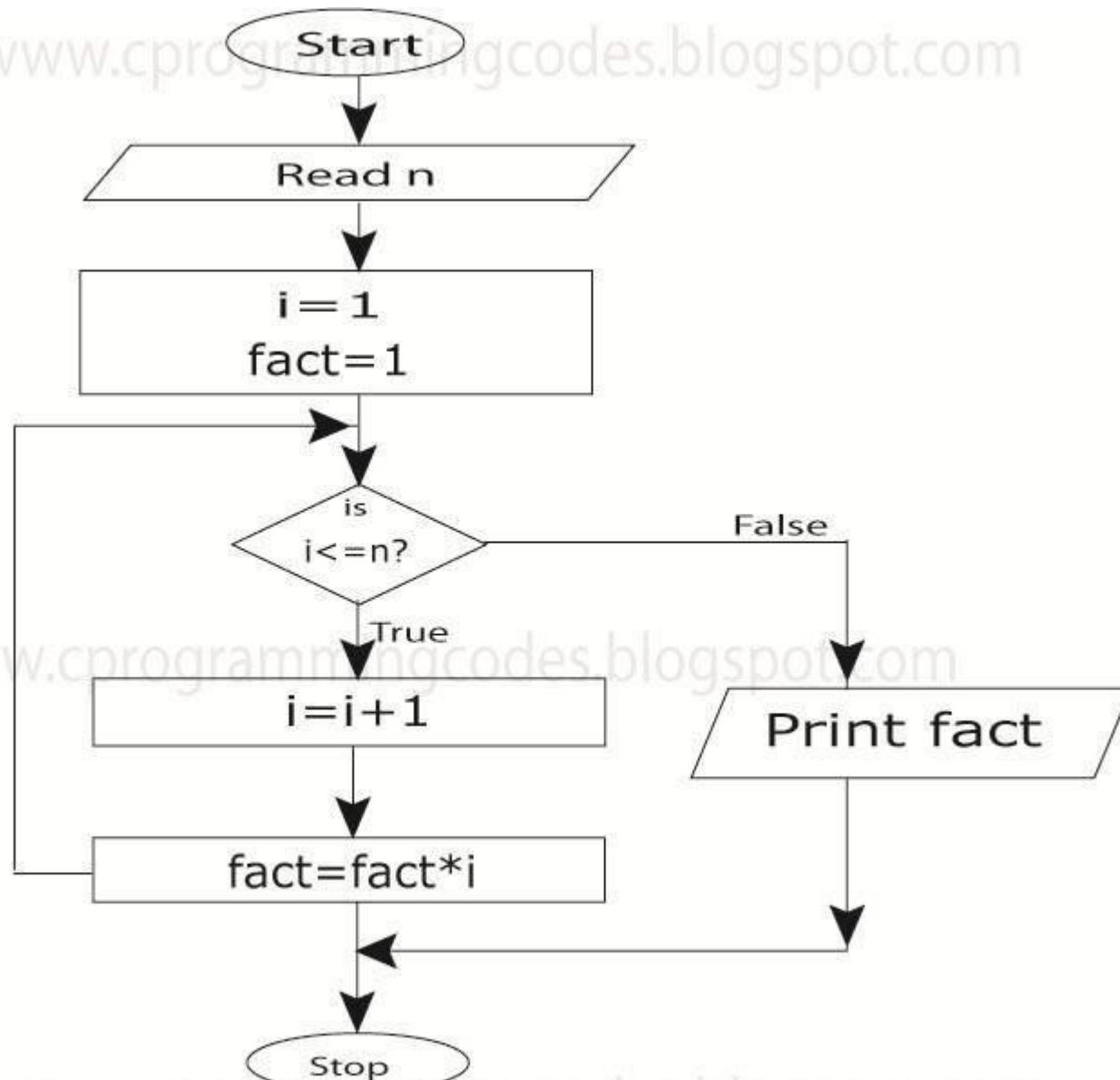
- **Guidelines for Preparing Flowchart**
  - Logical order of requirements
  - Ensure that Flowchart has logical *Start* and *Stop*
  - Direction is from Top to bottom
  - Only one flow line is used with Terminal Symbol
  - Only one flow line should come out of a Process symbol
  - Only one flow line should enter a Decision symbol but multiple lines may leave the Decision symbol

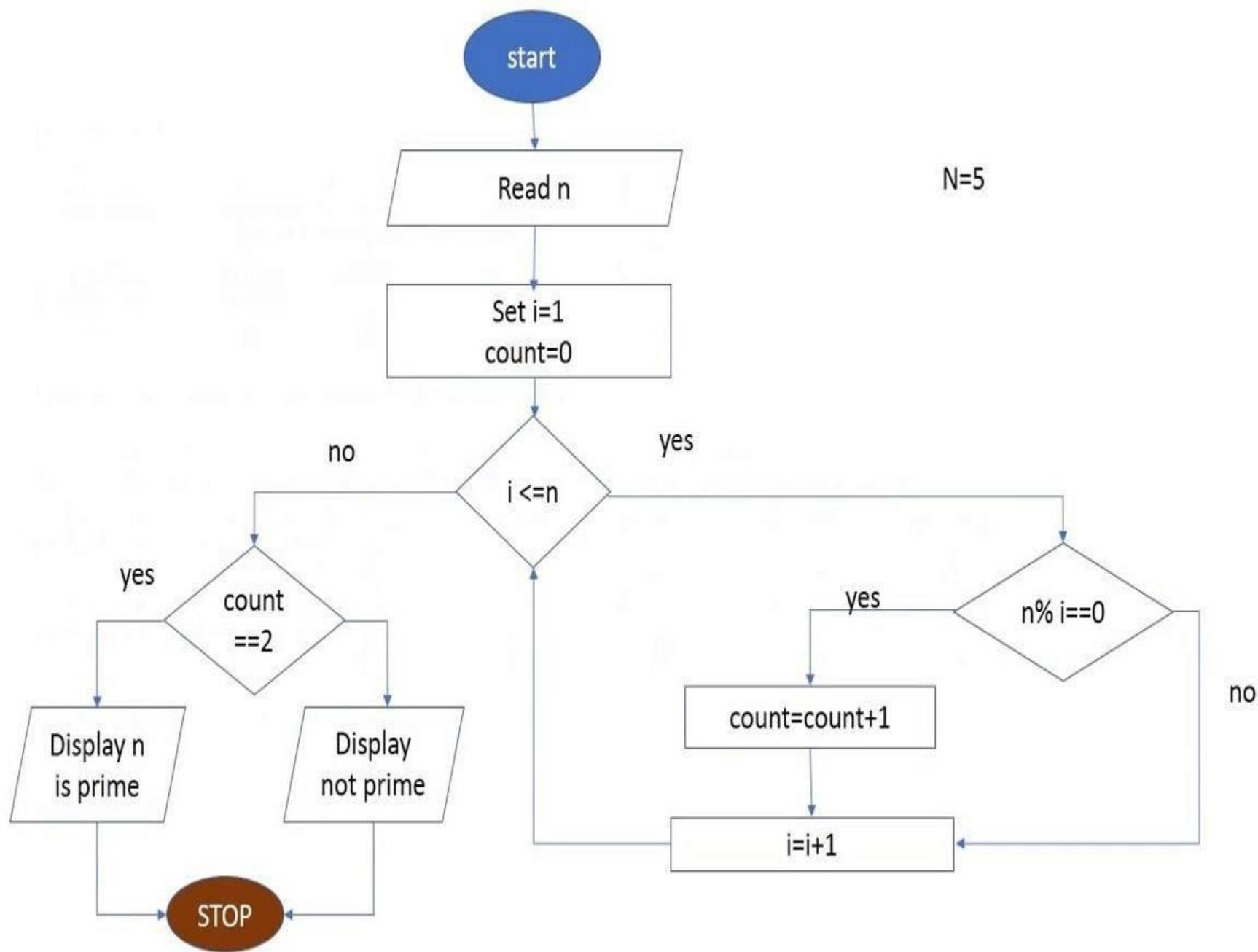
## 1. 4 Drawing Flowcharts Contd...

- **Guidelines for Preparing Flowchart Contd...**
  - Write briefly within Symbols
  - Use connectors to reduce number of flow lines
  - Avoid intersection of flow lines
  - Test Flowchart through simple test data
  - Clear, Neat and easy to follow

# 1. 4 Drawing Flowcharts Contd...







# 1. 5 Writing Pseudocode

- ❑ Pseudo – Imitation / False
- ❑ Code – Instructions
- ❑ **Goal:** To provide a high level description of the Algorithm
- ❑ **Benefit:** Enables programmer to concentrate on Algorithm
- ❑ Similar to programming code
- ❑ Description of the Algorithm
- ❑ No specific Programming language notations
- ❑ Pseudo Code transformed into actual program code

# 1. 5 Writing Pseudocode Contd...

## a) Guidelines for Writing Pseudo Code

- Write only one Statement per line
- Example – Pseudo Code for calculating Salary
  1. **READ** name, hourly rate, hours worked, deduction rate
  2. Gross pay = hourly rate \* hours worked
  3. deduction = gross pay \* deduction rate
  4. net pay = gross pay – deduction
  5. **WRITE** name, gross, deduction, net pay

# 1. 5 Writing Pseudocode Contd...

## b) Capitalize Initial Keyword

- Keywords to be written in capital letters
- Examples: **READ, WRITE, IF, ELSE, WHILE, REPEAT, PRINT**

## c) Indent to show Hierarchy

- Indentation shows the structure boundaries
- Sequence
- Selection
- Looping

# **1. 5 Writing Pseudocode Contd...**

## **d) End Multiline structures**

- Each structure must end properly
- Example: IF statement must end with ENDIF

## **e) Keep Statements Language independent**

- Resist the urge to write Pseudo Code in any programming language

# 1. 5 Writing Pseudocode Contd...

## ❑ Advantages

- ❑ Easily typed in a Word document
- ❑ Easily modified
- ❑ Simple to Use and understand
- ❑ Implements Structured Concepts
- ❑ No special symbols are used
- ❑ No specific syntax is used
- ❑ Easy to translate into Program

## ❑ Disadvantages

- ❑ No accepted Standard
- ❑ Cannot be compiled and executed

# 1. 5 Writing Pseudocode Contd...

## Write an Pseudo Code to:

- 1) Add three numbers and Display the result
- 2) Calculate Sum and product of two numbers
- 3) Input examination marks and award grades according to the following criteria:
  - a)  $> = 80$  Distinction
  - b)  $> = 60$  First Class
  - c)  $> = 50$  Second Class
  - d)  $< 40$  Fail

# **1. 5 Writing Pseudocode Contd...**

## **1. Pseudo Code to Add Three Numbers**

- Use Variables: sum, num1, num2, num3 of type integer
- ACCEPT num1,num2,num3
- Sum = num1+num2+num3
- Print sum
- End Program

## 1. 5 Writing Pseudocode Contd...

### 2. Calculate Sum and product of two numbers

- Use Variables: sum, product, num1, num2 of type real
- DISPLAY “Input two Numbers”
- ACCEPT num1,num2
- Sum = num1+num2
- Print “The sum is”, sum
- product = num1\*num2
- Print “The product is”, product
- End Program

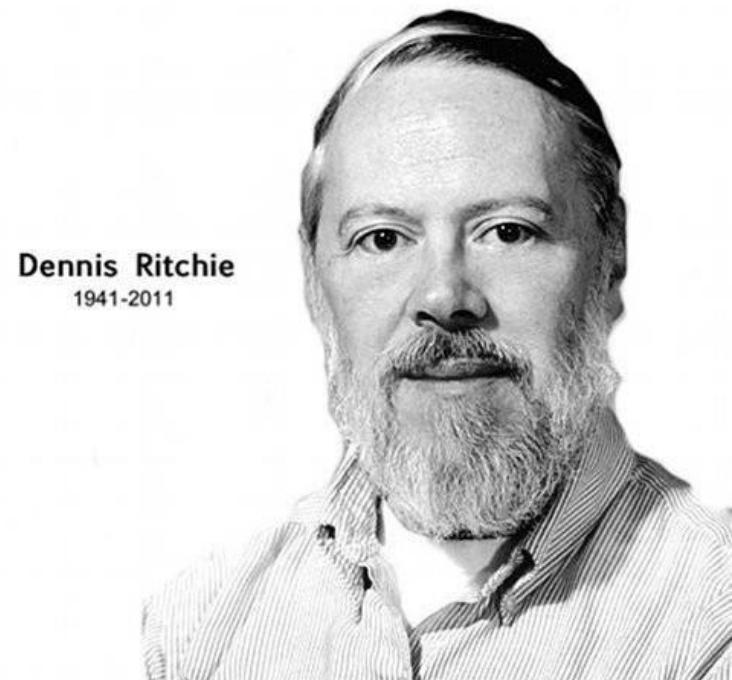
# **1. 5 Writing Pseudocode Contd...**

## **3. Input examination marks and award grades**

- Use Variables: mark of type integer
- If mark  $\geq 80$  DISPLAY “Distinction”
- If mark  $\geq 60$  and mark  $< 80$  DISPLAY “First Class”
- If mark  $\geq 50$  and mark  $< 60$  DISPLAY “Second Class”
- If mark  $< 50$  DISPLAY “Fail”
- End Program

# 1.6 History & Evolution of C

- ❑ C – General Purpose Programming Language
- ❑ Developed by Dennis Ritchie in 1972
- ❑ Developed at Bell Laboratories
- ❑ Principles taken from BCPL and CPL
- ❑ Structured Programming Language
- ❑ C Program
  - ❑ Collection of Functions
  - ❑ Supported by C library



# 1. 6 History & Evolution of C Cont...

*Father of C Programming : Dennis Ritchie*

<b>Born On</b>	September 9 1941
<b>Born in</b>	Bronxville – New York
<b>Full Name</b>	Dennis MacAlistair Ritchie
<b>Nickname</b>	DMR
<b>Nationality</b>	American
<b>Graduate From</b>	Harvard University
<b>Graduate In</b>	Physics and Applied Mathematics
<b>Webpage</b>	<a href="http://cm.bell-labs.com/who/dmr/">http://cm.bell-labs.com/who/dmr/</a>
<b>Dead On</b>	October 12 2011

# 1. 6 History & Evolution of C Cont...

1960	Algol	• International Group
1967	BCPL	• Martin Richards
1970	B	• Ken Thomson
1972	Traditional C	• Dennis Ritchie
1978	K&R C	• kernighan & Ritchie
1989	ANSI C	• ANSI Commitee
1990	ANSI/ISO C	• ISO Commitee
1999	C99	• Standard Commitee

*Evolution of C*

## 1. 6 History & Evolution of C Cont...

### ***Why the Name “C” was given ?***

- Many of C's principles and ideas were derived from the earlier language B
- BCPL and CPL are the earlier ancestors of B Language (CPL is common Programming Language)
- In 1967, BCPL Language ( Basic CPL ) was created as a scaled down version of CPL
- As many of the **features were derived from “B” Language the new language was named as “C”.**

# 1. 6 History & Evolution of C Cont...

## ❑ Characteristics of ‘C’

- ❑ Low Level Language Support
- ❑ Structured Programming
- ❑ Extensive use of Functions
- ❑ Efficient use of Pointers
- ❑ Compactness
- ❑ Program Portability
- ❑ Loose Typing

# 1. 6 History & Evolution of C Cont...

## ❑ Advantages of C

- ❑ Compiler based Language
- ❑ Programming – Easy & Fast
- ❑ Powerful and Efficient
- ❑ Portable
- ❑ Supports Graphics
- ❑ Supports large number of Operators
- ❑ Used to Implement Datastructures

## 1. 6 History & Evolution of C Cont...

### ❑ Disadvantages of C

- ❑ Not a strongly typed Language
- ❑ Use of Same operator for multiple purposes
- ❑ Not Object Oriented

# 1.7 Structure of ‘C’ Program

❑ Structure based on Set of rules defined by the Compiler

## ❑ **Sections**

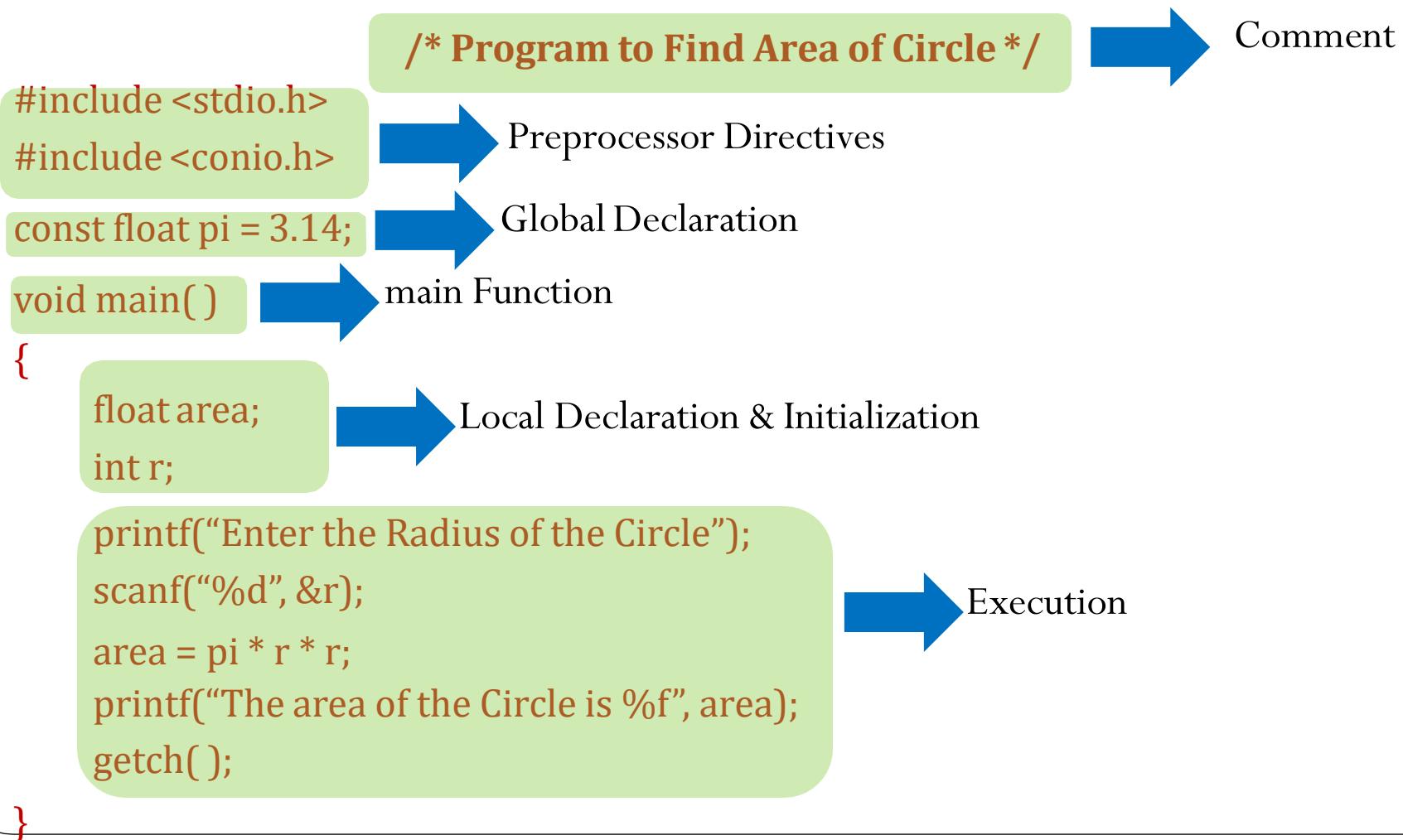
- |                       |                       |
|-----------------------|-----------------------|
| 1) Documentation      | 5) Local Declaration  |
| 2) Preprocessor       | 6) Program Statements |
| 3) Global Declaration |                       |
| 4) main( ) function   |                       |

## 1.7 Structure of 'C' Program Contd...

### □ Rules for Writing a C Program

- a) All statements should be written in lower case
- b) All statements should end with a semicolon
- c) Upper case letters are used for symbolic constants
- d) Blank spaces can be inserted between words
- e) No blank space while declaring a variable, keyword, constant
- f) Can write one or more statement in same line separated by comma
- g) Opening and closing of braces should be balanced

# 1.7 Structure of 'C' Program Contd...



# 1.7 Structure of 'C' Program Contd...

## 1) Documentation Section

- Used for providing Comments
- Comment treated as a single white space by Compiler
- Ignored at time of Execution: Not Executable
- Comment: Sequence of Characters given between /\* and \*/
- **Example:** Program Name, Statement description

/\* Program to Find Area of a Circle \*/

## 1.7 Structure of 'C' Program Contd...

### 2) Preprocessor Section

- Also called as Preprocessor Directive
- Also called as Header Files
- Not a part of Compiler
- Separate step in Compilation Process
- Instructs Compiler to do required Preprocessing
- Begins with **#** symbol
- Preprocessor written within **< >**

# 1.7 Structure of ‘C’ Program Contd...

## □ Examples

- #include <stdio.h>
- #include <conio.h>
- #include <math.h>
- #include <stdlib.h>
- #define PI 3.1412

# 1.7 Structure of 'C' Program Contd...

Directive	Description
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.

## 1. 7 Structure of 'C' Program Contd...

Directive	Description
#error	Prints error message on stderr.
#pragma	Issues special commands to the compiler, using a standardized method.

## **1.7 Structure of 'C' Program Contd...**

### **3) Global Declaration Section**

- Used to Declare Global variable (or) Public variable
- Variables are declared outside all functions
- Variables can be accessed by all functions in the program
- Same variable used by more than one function

# 1.7 Structure of 'C' Program Contd...

## 4) main( ) Section

- ❑ main( ) written in all small letters (No Capital Letters)
- ❑ Execution starts with a Opening Brace : {
- ❑ Divided into two sections: Declaration & Execution
  - ❑ **Declaration :** Declare Variables
  - ❑ **Executable:** Statements within the Braces
- ❑ Execution ends with a Closing Brace : }
- ❑ **Note:** main( ) does not end with a semicolon

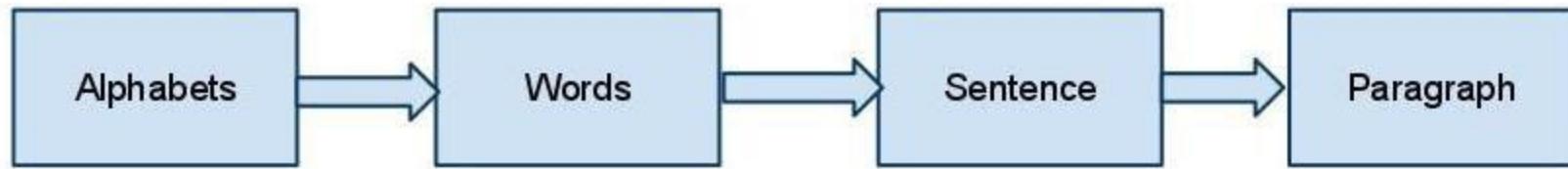
## **1.7 Structure of 'C' Program Contd...**

### **5) Local Declaration Section**

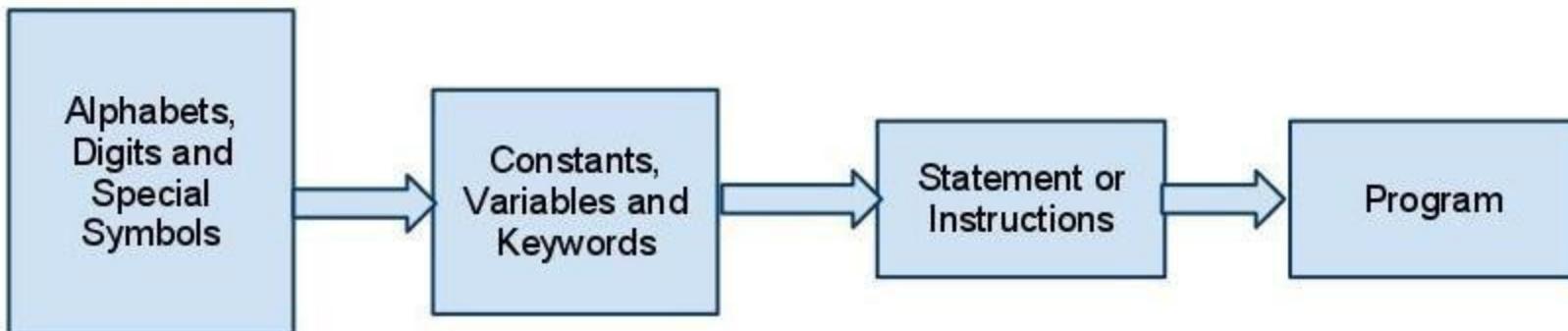
- Variables declared within the main( ) program
- These variables are called Local Variables
- Variables initialized with basic data types

# 1. 8 C Programming Fundamentals

Steps in Learning English Language



Steps in Learning C

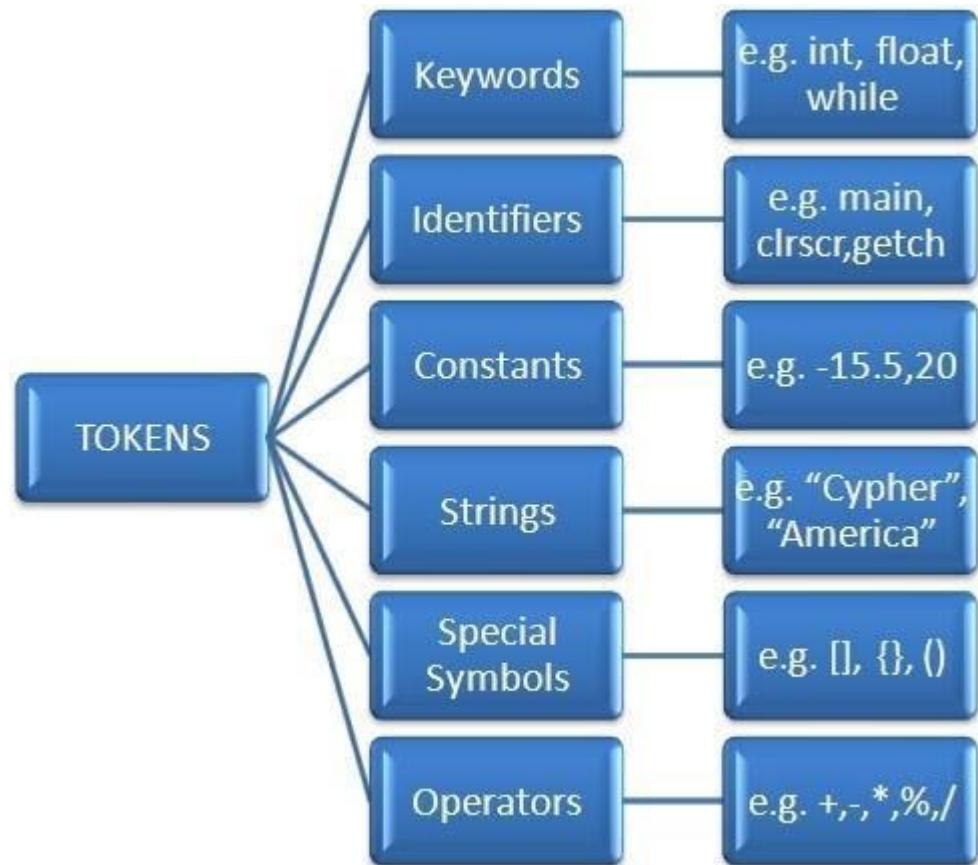


## 1. 8 C Programming Fundamentals Contd...

Alphabets	A, B, ....., Y, Z a, b, ....., y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ` ! @ # % ^ & * ( ) _ - + =   \ { } [ ] : ; " ' < > , . ? /

# 1. 8 C Programming Fundamentals Contd...

- ❑ **C Token** - Smallest individual unit of a C program
- ❑ C program broken into many C tokens
- ❑ Building Blocks of C program



# 1. 9 Single Line and Multiline Comments

## □ Comment – Definition

- Used to provide information about lines of code
- Provide clarity to the C source code
- Allows others to better understand what the code was intended to
- Helps in debugging the code
- Important in large projects containing hundreds or thousands of lines of source code
- Types – Single line and multiline comment

# 1. 9 Single Line and Multiline Comments Contd...

## a) Single Line Comment

- Represented by double slash \\

```
#include<stdio.h>

int main( ){
    //printing information
    printf("Hello C");

    return 0;
}
```

# 1. 9 Single Line and Multiline Comments Contd...

## b) Multi-Line Comment

- Represented by slash asterisk \\* ... \*\

```
#include<stdio.h>

int main( ){
    /*printing information
     Multi Line Comment*/
    printf("Hello C");

    return 0;
}
```

# 1. 9 Single Line and Multiline Comments Contd...

<b>Single-Line Comments</b>	<b>Multi-Line Comment</b>
Starts with /* and ends with */	Starts with //
All Words and Statements written between /* and */ are ignored	Statements after the symbol // upto the end of line are ignored
Comment ends when */ Occures	Comment Ends whenever ENTER is Pressed and New Line Starts
e.g /* Program for Factorial */	e.g // Program for Fibonacci

# 1. 10 Keywords

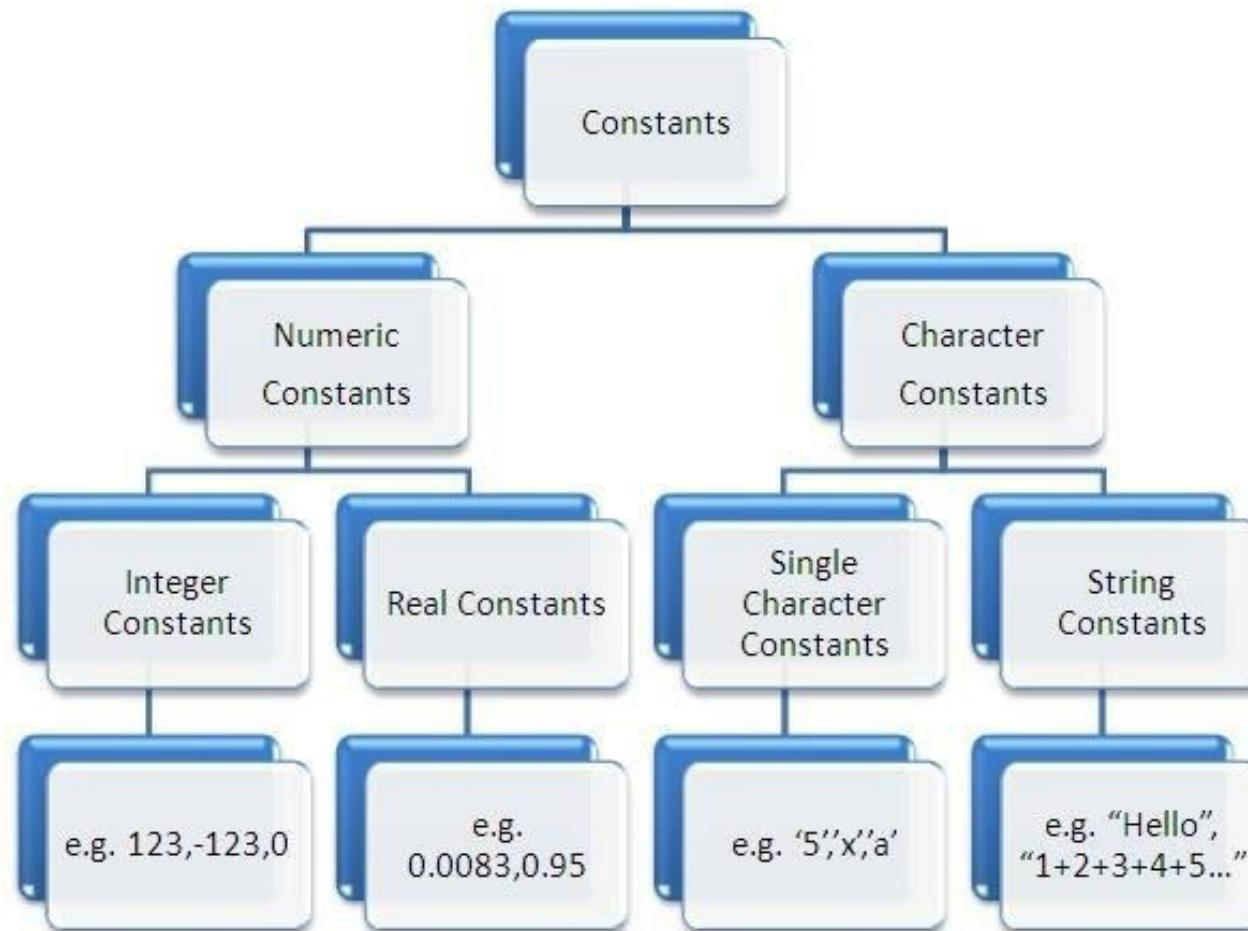
- ❑ **Keywords** – Conveys special meaning to Compiler
- ❑ Cannot be used as variable names

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

## 1. 11 Constants

- ❑ Definition :Value does not change during execution
- ❑ Can be a Number (or) a Letter
- ❑ **Types**
  - ❑ Integer Constants
  - ❑ Real Constants
  - ❑ Character Constant
    - ❑ Single Character Constants
    - ❑ String Constants

## 1.11 Constants Contd...



# 1. 12 Variables & Identifiers

## □ *Identifier*

- A string of alphanumeric characters that begins with an alphabetic character or an underscore character
- There are 63 alphanumeric characters, i.e., 53 alphabetic characters and 10 digits (i.e., 0-9)
- Used to represent various programming elements such as variables, functions, arrays, structures, unions
- The underscore character is considered as a letter in identifiers (Usually used in the middle of an identifier)

# 1. 12 Variables & Identifiers Contd...

## □ Rules for Identifiers

- Combination of alphabets, digits (or) underscore
- First character should be a Alphabet
- No special characters other than underscore can be used
- No comma / spaces allowed within variable name
- A variable name cannot be a keyword
- Variable names are case sensitive

## □ ***Variable Definition*** :Value changes during execution

- Identifier for a memory location where data is stored

## 1. 12 Variables & Identifiers Contd...

- Variable name length cannot be more than 31 characters
- **Examples:** AVERAGE, height, a, b, sum, mark\_1, gross\_pay
- **Variable Declaration**
  - A variable must be declared before it is used
  - Declaration consists of a data type followed by one or more variable names separated by commas.
- Syntax

*datatype variablename;*

## 1. 12 Variables & Identifiers Contd...

- Examples

*int a, b, c, sum;*

*float avg;*

*char name;*

- **Variable Initialization**

- Assigning a value to the declared variable
- Values assigned during declaration / after declaration

# 1. 12 Variables & Identifiers Contd...

## □ Examples

i. *int a, b, c;*

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

ii. *int a=10 ,b=10, c=10;*

## □ Scope of Variables

### □ Local Variables

### □ Global Variables

## 1. 13 Scope of Variables

### □ *Definition*

- A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed
- **Variable Scope** is a region in a program where a variable is declared and used
- The **scope** of a variable is the range of program statements that can access that variable
- A variable is **visible** within its scope and **invisible** outside it

## 1. 13 Scope of Variables Contd...

- ❑ There are three places where variables can be declared
  - a) Inside a function or a block which is called **local** variables
  - b) Outside of all functions which is called **global** variables
  - c) In the definition of function parameters which are called **formal** parameters

## 1. 13 Scope of Variables Contd...

### a) *Local Variables*

- Variables that are declared inside a function or block are called local variables
- They can be used only by statements that are inside that function or block of code
- Local variables are created when the control reaches the block or function containing the local variables and then they get destroyed after that
- Local variables are not known to functions outside their own

**/\* Program for Demonstrating Local Variables \*/**

```
#include <stdio.h>

int main ( )
{
    /* local variable declaration */

    int a, b;

    int c;
    /* actual initialization */

    a = 10; b = 20;

    c = a + b;

    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

    return 0;
}
```

## 1. 13 Scope of Variables Contd...

### b) *Global Variables*

- ❑ Defined outside a function, usually on top of the program
- ❑ Hold their values throughout the lifetime of the program
- ❑ Can be accessed inside any of the functions defined for the program
- ❑ Can be accessed by any function
  - ❑ That is, a global variable is available for use throughout the entire program after its declaration

## /\* Program for Demonstrating Global Variables \*/

```
#include <stdio.h>
    /* global variable declaration */
int g;
int main ( )
{
    /* local variable declaration */
    int a, b;
    /* actual initialization */
    a = 10; b = 20;
    g = a + b;
    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
    return 0;
}
```

## 1. 13 Scope of Variables Contd...

- ❑ **Note:** A program can have same name for local and global variables but the value of local variable inside a function will take preference

## 1. 14 Binding

- ❑ A Binding is an association between an entity and an attribute
  - ❑ Between a variable and its type or value
  - ❑ Between a function and its code
- ❑ Binding time is the point at which a binding takes place
- ❑ Types of Binding
  - a) Design Time
  - b) Compile Time
  - c) Link Time
  - d) Run Time

## 1. 14 Binding Contd...

### a) *Design Time*

- ❑ Binding decisions are made when a language is designed
- ❑ Example
  - ❑ Binding of + to addition in C

### b) *Compile Time*

- ❑ Bindings done while the program is compiled
- ❑ Binding variables to datatypes
- ❑ Example
  - ❑ int a; float b; char c;

## 1. 14 Binding Contd...

### c) *Link Time*

- ❑ Compiled code is combined into a full program for C
- ❑ Example
  - ❑ Global and Static variables are bound to addresses

### d) *Run Time*

- ❑ Any binding that happens at run time is called *Dynamic*
- ❑ Any binding that happens before run time is called *Static*
- ❑ Values that are dynamically bound can change

## 1. 15 Storage Classes in C

- ❑ What is a Variable?
- ❑ Storage Class Specifiers tells the Compiler about the following:
  - ❑ Where to Store a Variable
  - ❑ What is the Initial value of the Variable
  - ❑ What is the Lifetime of a Variable
- ❑ **Variable Scope:** Area or block where the variables can be accessed
  - a) Automatic Variables
  - b) External Variables
  - c) Static Variables
  - d) Register variables

## 1. 15 Storage Classes in C Contd...

### a) Automatic Variable (Or) Auto Variable (Or) Local Variable

- ❑ Default Storage class
- ❑ Defined inside a Function
- ❑ **Scope of the Variable:** Local to the function block where the variable is defined
- ❑ Lifetime of the variable's content vanishes after execution
- ❑ Keyword **Auto** used to Erase content of the variable

```
/* Program to Demonstrate Automatic (Or) Local Variables*/
#include<stdio.h>
#include<conio.h>
void main( )
{
    int n = 10;
    block1();
    block2();
    printf("In Main Block n=%d", n);
    getch( );
}
block1()
{
    int n = 20;
    printf("In Block 1 n=%d", n);
}
```

```
block2( )  
{  
    int n = 30;  
    printf("In Block 2 n=%d", n);  
}
```

### ***Output***

In Block 1 n=20

In Block 2 n= 30

In Main Block n=10

## 1. 15 Storage Classes in C Contd...

### b) External Variable (Or) Global Variable

- Available to all the Functions
- Defined outside the Function
- Keyword ***Declare*** is used to define the global variable (Optional)
- **Scope of the Variable:** Global to all the function blocks
- Lifetime of the variable's content vanishes after the entire program is executed

**/\* Program to Demonstrate External / Global Variables \*/**

```
#include<stdio.h>
#include<conio.h>
int n = 10;
void main( )
{
    block1();
    block2();
    clrscr( );
    printf("In Main Block n=%d", n);
    getch( );
}
block1()
{
    printf("In Block 1 n=%d", n);
    return;
}
```

```
block2( )  
{  
    printf("In Block 2 n=%d", n);  
    return;  
}
```

### ***Output***

In Block 1 n=10

In Block 2 n= 10

In Main Block n=10

# 1. 15 Storage Classes in C Contd...

## c) Static Variables

- ❑ Keyword **Static** is used to define the variable (Compulsory)
- ❑ Variable declared as static is initialized to NULL
- ❑ Value of the Static variable remains the same throughout the program
- ❑ **Scope of the Variable:** Local or Global depending on where it is declared
- ❑ **Static Global:** Defined outside the Function
- ❑ **Static Local:** Defined inside the Function

## /\* Program to Demonstrate Static Variables \*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int x;
    static int y;
    clrscr( );
    printf("x=%dy=%d", x, y);
    getch( );
}
```

### ***Output***

x = 28722

y = 0

# 1. 15 Storage Classes in C Contd...

## d) Register Variables

- Variables stored in the CPU registers instead of Memory
- Keyword ***Register*** is used to define the variable
- **Scope of the Variable:** Local
- **Advantages**
  - CPU register access is faster than memory access
- **Disadvantages**
  - Number of CPU registers is less
  - Less Number of variables can be stored in CPU registers

## /\* Program to Demonstrate Register Variables \*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    register int n=1;
    clrscr( );
    for(n=1; n<=10; n++)
        printf("%d", n);
    getch( );
}
```

### ***Output***

1 2 3 4 5 6 7 8 9 10

## 1. 16 Datatypes

- ❑ Defines a variable before use
- ❑ Specifies the type of data to be stored in variables
- ❑ Basic Data Types – 4 Classes
  - a) int – Signed or unsigned number
  - b) float – Signed or unsigned number having Decimal Point
  - c) double – Double Precision Floating point number
  - d) char – A Character in the character Set
- ❑ Qualifiers

# 1. 16 Datatypes Contd...

Variable Type	Keyword	Bytes Required	Range	Format
Character (signed)	Char	1	-128 to +127	%c
Integer (signed)	Int	2	-32768 to +32767	%d
Float (signed)	Float	4	-3.4e38 to +3.4e38	%f
Double	Double	8	-1.7e308 to +1.7e308	%lf
Long integer (signed)	Long	4	2,147,483,648 to 2,147,438,647	%ld
Character (unsigned)	Unsigned char	1	0 to 255	%c
Integer (unsigned)	Unsigned int	2	0 to 65535	%u
Unsigned long integer	unsigned long	4	0 to 4,294,967,295	%lu
Long double	Long double	10	-1.7e932 to +1.7e932	%Lf

# 1. 16 Datatypes Contd...

## a) Integer Data Type

- Whole numbers with a range
- No fractional parts
- Integer variable holds integer values only
- **Keyword:** int
- **Memory:** 2 Bytes (16 bits) or 4 Bytes (32 bits)
- **Qualifiers:** Signed, unsigned, short, long
- **Examples:** 34012, 0, -2457

## 1. 16 Datatypes Contd...

### b) Floating Point Data Type

- Numbers having Fractional part
- Float provides precision of 6 digits
- Integer variable holds integer values only
- **Keyword:** float
- **Memory:** 4 Bytes (32 bits)
- **Examples:** 5.6, 0.375, 3.14756

# 1. 16 Datatypes Contd...

## c) Double Data Type

- Also handles floating point numbers
- Double provides precision of 14 digits
- Integer variable holds integer values only
- Keyword:** float
- Memory:** 8 Bytes (64 bits) or 10 Bytes (80 bits)
- Qualifiers:** long, short

## 1. 16 Datatypes Contd...

### d) Character Data Type

- handles one character at a time
- **Keyword:** char
- **Memory:** 1 Byte (8 bits)

## 1. 17 Expressions

- ❑ **Expression :** An Expression is a collection of operators and operands that represents a specific value
- ❑ **Operator :** A symbol which performs tasks like arithmetic operations, logical operations and conditional operations
- ❑ **Operands :** The values on which the operators perform the task
- ❑ Expression Types in C
  - a) Infix Expression
  - b) Postfix Expression
  - c) Prefix Expression

## I1. 17 Expressions Contd...

### a) *Infix Expression*

- The operator is used between operands
- **General Structure :** Operand1 Operator Operand2
- **Example :** a + b

### b) *Postfix Expression*

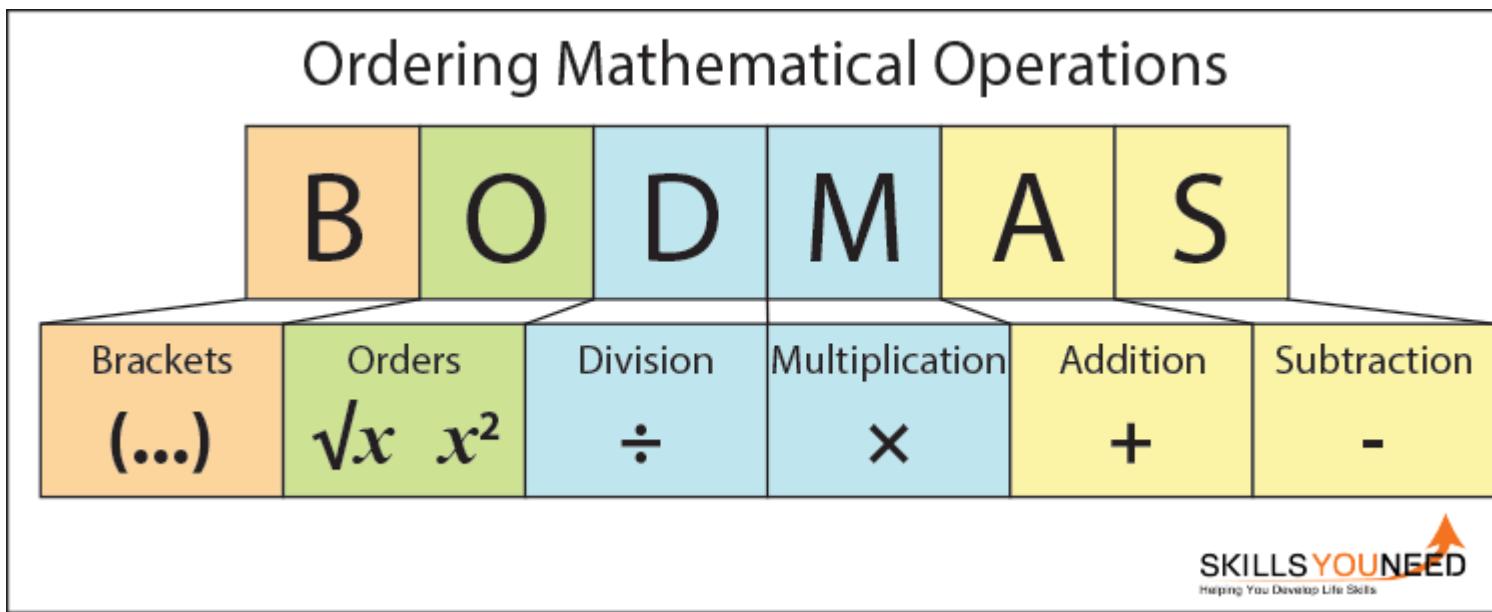
- Operator is used after operands
- **General Structure :** Operand1 Operand2 Operator
- **Example :** ab+

## 1. 17 Expressions Contd...

### c) *Prefix Expression*

- Operator is used before operands
- **General Structure :** Operator Operand1 Operand2
- **Example :** +ab

# 1. 17 Expressions Contd...



Precedence order	Operator	Associativity
1	( ) [ ] →	Left to right
2	++ -- - (unary) ! ~ * & sizeof	Right to left
3	* / %	Left to right
4	+ -	Left to right
5	<< >>	Left to right
6	< <= > >=	Left to right
7	= !=	Left to right
8	& (bitwise AND)	Left to right
9	^ (bitwise XOR)	Left to right
10	(bitwise OR)	Left to right

# 1. 18 Input and Output Functions

- ❑ Ability to Communicate with Users during execution

## ❑ **Input Operation**

- ❑ Feeding data into program
- ❑ Data Transfer from Input device to Memory

## ❑ **Output Operation**

- ❑ Getting result from Program
  - ❑ Data Transfer from Memory to Output device
- ❑ Header File : #include<**stdio.h**>

# **1. 18 Input and Output Functions Contd...**

## **□ Input / Output Function Types**

- a) Formatted Input / Output Statements
- b) Unformatted Input / Output Statements

## Console Input / Output Functions

Formatted Functions			Unformatted Functions		
Type	Input	Output	Type	Input	Output
Char	scanf( )	printf( )	char	getch( ) getche( ) getchar( )	putch( ) putchar( )
Int	scanf( )	printf( )	int	-	-
Float	scanf( )	printf( )	float	-	-
String	scanf( )	printf( )	string	gets( )	puts( )

# 1. 18 Input and Output Functions Contd...

## a) Formatted Input / Output Statements

- Reads and writes all types of data values
- Arranges data in particular format
- Requires **Format Specifier** to identify Data type
- Basic Format Specifiers
  - %d – Integer
  - %f – Float
  - %c – Character
  - %s - String

# 1. 18 Input and Output Functions Contd...

## i. The scanf ( ) Function

- Reads all types of input data
- Assignment of value to variable during Runtime
- Syntax

**scanf(“Control String/Format Specifier”, &arg1, &arg2,... &argn)**

- Control String / Format Specifier
- arg1, arg2,,, arg n – Arguments (Variables)
- & - Address

# 1. 18 Input and Output Functions Contd...

/\* Giving Direct Input in  
Program \*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a;
    a=10;
}
```

/\*Getting Input using scanf( )  
function \*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a;
    scanf("%d", &a);
}
```

# 1. 18 Input and Output Functions Contd...

```
/* Getting Multiple Input using  
   scanf( ) function */  
  
#include<stdio.h>  
  
#include<conio.h>  
  
void main( )  
{  
    int a, b, c;  
    scanf("%d%d%d",&a,&b,&c);  
}
```

```
/* Getting Multiple Different Inputs  
   using scanf( ) function */  
  
#include<stdio.h>  
  
#include<conio.h>  
  
void main( )  
{  
    int a, b;  
    float c;  
    scanf("%d%d%f",&a,&b,&c);  
}
```

# 1. 18 Input and Output Functions Contd...

```
/* Getting Multiple Input using scanf ( ) function */  
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int a, b;  
    float c;  
    scanf("%d %d", &a, &b);  
    scanf("%f", &c);  
}
```

# 1. 18 Input and Output Functions Contd...

## ii. The printf( ) Function

- To print Instructions / Output onto the Screen
- Requires Format Specifiers & Variable names to print data

### □ Syntax

```
printf("Control String/Format Specifier",arg1,arg2,... argn)
```

- Control String / Format Specifier
- arg1, arg2.,., arg n – Arguments (Variables)

## 1. 18 Input and Output Functions Contd...

```
/* Example 1 – Using printf( ) & scanf( ) function */

#include<stdio.h>
#include<conio.h>
void main()
{
    int a;
    printf("Enter the Value of a");
    scanf("%d", &a);
    printf("Value of a is %d", a);
    getch();
}
```

## 1. 18 Input and Output Functions Contd...

```
/* Example 2 – Using printf ( ) & scanf ( ) function */  
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int a, b, c;  
    printf("Enter the Value of a, b & c");  
    scanf("%d %d %d", &a, &b, &c);  
    printf("Value of a, b & c is %d%d%d", a, b, c);  
    getch();  
}
```

*/\* Example 3 – Using printf ( ) & scanf ( ) function \*/*

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

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

```
void main()
```

```
{
```

```
    int a, b;
```

```
    float c;
```

```
    printf("Enter the Value of a & b");
```

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

```
    printf("Enter the Value of a & b");
```

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

```
    printf("Value of a, b is %d%d", a, b);
```

```
    printf("Value of c is %f", c);
```

```
    getch();
```

```
}
```

## 1. 18 Input and Output Functions Contd...

```
/* Example 4 – Using printf ( ) & scanf ( ) function */

#include<stdio.h>
#include<conio.h>
void main()
{
    int a, b;
    float c;
    printf("Enter the Value of a, b & c");
    scanf("%d %d%f", &a, &b, &c);
    printf("Value of a, b & c is %d%d%f", a, b, c);
    getch();
}
```

# 1. 18 Input and Output Functions Contd...

**Try it Out Yourself ! Write a C program to:**

- 1) Add two numbers
- 2) To Multiply two floating point numbers
- 3) To compute Quotient and Remainder
- 4) To Swap two numbers

# 1. 18 Input and Output Functions Contd...

## b) Unformatted Input / Output Statements

- Works only with Character Data type
- No need of Format Specifier

### □ Unformatted Input Statements

- i. **getch ( )** – Reads alphanumeric characters from Keyboard
- ii. **getchar ( )** – Reads one character at a time till enter key is pressed

## 1. 18 Input and Output Functions Contd...

**iii. gets ( )** – Accepts any string from Keyboard until Enter Key is pressed

### □ Unformatted Output Statements

**i. putch ( )** – Writes alphanumeric characters to Monitor (Output Device)

**ii. putchar ( )** – Prints one character at a time

**iii. puts ( )** – Prints a String to Monitor (Output Device)

## 1. 19 Operators in C

- ❑ C supports rich set of built in Operators
- ❑ Used to manipulate Constants (Data) & Variables
- ❑ Part of Mathematical (or) Logical expressions
- ❑ Operators vs Operands
- ❑ **Operator – Definition**
  - ❑ Symbol (or) Special character that instructs the compiler to perform mathematical (or) Logical operations

# 1. 19 Operators in C Contd...

## □ Classification of Operators

- a) Increment & Decrement Operators
- b) Comma Operator
- c) Arrow Operator
- d) Assignment Operators
- e) Bitwise Operators
- f) Sizeof Operator

# 1. 19 Operators in C Contd...

## a) Increment and Decrement Operators

- ❑ Increment and decrement operators are unary operators that add or subtract one from their operand
- ❑ C languages feature two versions (pre- and post-) of each operator
  - ❑ Operator placed before variable (Pre)
  - ❑ Operator placed before variable (Post)
- ❑ The increment operator is written as ++ and the decrement operator is written as --

# 1. 19 Operators in C Contd...

## a) Increment and Decrement Operators Contd...

### □ Classification

- Pre Increment Operator
- Post Increment Operator
- Pre Decrement Operator
- Post Decrement Operator

# 1. 19 Operators in C Contd...

## a) Increment and Decrement Operators Contd...

### □ Syntax

*(pre)++variable\_name;*

*(pre)- -variable\_name;*

(Or)

*variable\_name++ (post);*

*variable\_name - (Post);*

### □ Examples

- ++count, ++a, ++i, ++count
- Count++, a++, i++, count++

# 1. 19 Operators in C Contd...

## a) Increment and Decrement Operators Contd...

S. No	Operator type	Operator	Description
1	Pre Increment	<code>++i</code>	Value of i is incremented before assigning it to variable i.
2	Post Increment	<code>i++</code>	Value of i is incremented after assigning it to variable i.
3	Pre Decrement	<code>-- i</code>	Value of i is decremented before assigning it to variable i.
4	Post Decrement	<code>i --</code>	Value of i is decremented after assigning it to variable i.

## /\* Program for Post Increment \*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int i = 1;
    while (i++ < 5)
    {
        printf("%d", i);
    }
    getch();
}
```

## *Output*

1 2 3 4

## 1. 19 Operators in C Contd...

### a) Increment and Decrement Operators Contd...

- ❑ **Step 1 :** In this program, value of i “0” is compared with 5 in while expression.
- ❑ **Step 2 :** Then, value of “i” is incremented from 0 to 1 using post-increment operator.
- ❑ **Step 3 :** Then, this incremented value “1” is assigned to the variable “i”.
- ❑ Above 3 steps are continued until while expression becomes false and output is displayed as “1 2 3 4 5”.

## /\* Program for Pre Increment \*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int i = 1;
    while (++i<5)
    {
        printf("%d", i );
    }
    getch ( );
}
```

### *Output*

2 3 4

## 1. 19 Operators in C Contd...

### a) Increment and Decrement Operators Contd...

- ❑ **Step 1 :** In above program, value of “i” is incremented from 0 to 1 using pre-increment operator.
- ❑ **Step 2 :** This incremented value “1” is compared with 5 in while expression.
- ❑ **Step 3 :** Then, this incremented value “1” is assigned to the variable “i”.
- ❑ Above 3 steps are continued until while expression becomes false and output is displayed as “1 2 3 4”.

/\* Program for Post Decrement \*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int i = 10;
    while (i--<5)
    {
        printf("%d", i );
    }
    getch ( );
}
```

*Output*

10 9 8 7 6

## 1. 19 Operators in C Contd...

### a) Increment and Decrement Operators Contd...

- ❑ **Step 1 :** In this program, value of i “10” is compared with 5 in while expression.
- ❑ **Step 2 :** Then, value of “i” is decremented from 10 to 9 using post-decrement operator.
- ❑ **Step 3 :** Then, this decremented value “9” is assigned to the variable “i”.
- ❑ Above 3 steps are continued until while expression becomes false and output is displayed as “9 8 7 6 5”.

## /\* Program for Pre Decrement \*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int i = 10;
    while (--i<5)
    {
        printf("%d", i);
    }
    getch ( );
}
```

### *Output*

9 8 7 6

## 1. 19 Operators in C Contd...

### a) Increment and Decrement Operators Contd...

- ❑ **Step 1 :** In above program, value of “i” is decremented from 10 to 9 using pre-decrement operator.
- ❑ **Step 2 :** This decremented value “9” is compared with 5 in while expression.
- ❑ **Step 3 :** Then, this decremented value “9” is assigned to the variable “i”.
- ❑ Above 3 steps are continued until while expression becomes false and output is displayed as “9 8 7 6”.

# 1. 19 Operators in C Contd...

## b) Comma Operator

- Special operator which separates the declaration of multiple variables
- Has Lowest Precedence i.e it is having lowest priority so it is evaluated at last
- Returns the value of the rightmost operand when multiple comma operators are used inside an expression
- Acts as Operator in an Expression and as a Separator while Declaring Variables

## 1. 19 Operators in C Contd...

### b) Comma Operator Contd...

```
#include<stdio.h>

int main( )
{
    int i, j;
    i=(j=10,j+20);
    printf("i = %d\n j = %d\n" , i,j );
    return 0;
}
```

## 1. 19 Operators in C Contd...

### c) Arrow Operator (->)

- ❑ Arrow operator is used to access the structure members when we use pointer variable to access it
- ❑ When pointer to a structure is used then arrow operator is used

# 1. 19 Operators in C Contd...

## d) Assignment Operators

- Assigns result of expression to a variable
- Performs Arithmetic and Assignment operations
- Commonly used Assignment operator: **=**
- **Syntax**

***variable = expression;***

### □ **Examples**

- num = 25; age = 18; pi = 31.4; area = 3.14 \* r \* r;

# 1. 19 Operators in C Contd...

## □ Shorthand Assignment Operators

Simple Assignment Operator	Shorthand Operator
$a = a + 1$	$a+=1$
$a = a - 1$	$a-=1$
$a = a * 2$	$a*=2$
$a = a / b$	$a/=b$
$a = a \% b$	$a\%b$
$c = c * (a + b)$	$c *= (a + b)$
$b = b / (a + b)$	$b /= (a + b)$

## /\* Program for Assignment Operations \*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int a;
    a = 11;
    a+ = 4;
    printf("Value of A is %d\n",a);
    a = 11;
    a- = 4;
    printf("Value of A is %d\n",a);
    a = 11;
    a* = 4;
    printf("Value of A is %d\n",a);
    a = 11; a/ = 4;
```

```
    printf("Value of A is %d\n",a);  
    a = 11;  
    a% = 4;  
        printf("Value of A is %d\n",a);  
    getch ( );  
}
```

### ***Output***

Value of A is 15

Value of A is 7

Value of A is 44

Value of A is 2

Value of A is 3

## 1. 19 Operators in C Contd...

### e) Bitwise Operators

- ❑ In arithmetic-logic unit, mathematical operations

Like addition, subtraction, multiplication and division are done in bit-level

- ❑ To perform bit-level operations in C programming, bitwise operators are used
- ❑ Bit wise operators in C language are & (bitwise AND), | (bitwise OR), ~ (bitwise NOT), ^ (XOR), << (left shift) and >> (right shift)

## 1. 19 Operators in C Contd...

### e) Bitwise Operators Contd...

Operator	Operation
&	Bitwise AND
	Bitwise OR
~	One's Complement
>>	Shift right
<<	Shift left
^	Exclusive OR

## Bitwise operators: truth table

<b>a</b>	<b>b</b>	<b>a&amp;b</b>	<b>a b</b>	<b>a^b</b>	<b>~a</b>
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it ( $A \& B$ ) = 12, i.e., 0000 1100 exists in both operands.	1100
	Binary OR Operator copies a bit if it exists in either ( $A   B$ ) = 61, i.e., 0011 1101 operand.	1101
^	Binary XOR Operator copies the bit if it is set in one ( $A ^ B$ ) = 49, i.e., 0011 0001 operand but not both.	0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	$(\sim A) = -60$ , i.e., 1100 0100 in 2's complement form.
<<	Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

# PROGRAM TO DEMONSTRATE BITWISE OPERATIONS

```
#include <stdio.h>

int main()
{
    unsigned int a = 60; /* 60 = 0011 1100 */
    unsigned int b = 13; /* 13 = 0000 1101 */
    int c = 0; c = a & b; /* 12 = 0000 1100 */
    printf("Line 1 - Value of c is %d\n", c );
    c = a | b; /* 61 = 0011 1101 */
    printf("Line 2 - Value of c is %d\n", c );
    c = a ^ b; /* 49 = 0011 0001 */
    printf("Line 3 - Value of c is %d\n", c );
    c = ~a; /* -61 = 1100 0011 */
    printf("Line 4 - Value of c is %d\n", c );
    c = a << 2; /* 240 = 1111 0000 */
    printf("Line 5 - Value of c is %d\n", c );
    c = a >> 2; /* 15 = 0000 1111 */
    printf("Line 6 - Value of c is %d\n", c );
    return 0;
}
```

## ***Output***

Line 1 - Value of c is 12

Line 2 - Value of c is 61

Line 3 - Value of c is 49

Line 4 - Value of c is -61

Line 5 - Value of c is 240

Line 6 - Value of c is 15

# 1. 19 Operators in C Contd...

## f) **Sizeof Operators**

- This operator returns the size of its operand in bytes
- The sizeof operator always precedes its operand
  - Used to calculate the size of data type or variables
  - Can be nested
  - Returns the size in integer format
- Syntax looks more like a function but it is considered as an operator in c programming

## /\* Program for Sizeof Operators \*/

```
#include<stdio.h>

int main( )
{
    int a;
    float b;
    double c;
    char d;

    printf("Size of Integer      :%d\n\n",sizeof(a));
    printf("Size of Floating Point      :%d\n\n",sizeof(b));
    printf("Size of Double      :%d\n\n",sizeof(c));
    printf("Size of Charcter      :%d\n\n",sizeof(d));
    return 0;
}
```

## ***Output***

Size of Integer :2

Size of Floating Point :4

Size of Double :8

Size of Character :1

# 18CSS101J – Programming for Problem Solving

## Unit II

## **UNIT II**

Relational and logical Operators - Condition Operators,  
Operator Precedence - Expressions with pre / post increment  
Operator - Expression with conditional and assignment  
operators - If statement in expression - L value and R value in  
expression - Control Statements – if and else - else if and  
nested if, switch case - Iterations, Conditional and  
Unconditional Branching - For loop - While loop - do while,  
goto, break, continue – Array - Initialization and Declaration-

## **UNIT II**

- Initialization: one Dimensional Array-Accessing, Indexing one Dimensional Array Operations - Array Programs – 1D

## 2. 1 Operators in C

- a) Relational Operators
- b) Logical Operators
- c) Conditional Operators

## 2. 1 Operators in C Contd...

### a) Relational Operators

- Binary Operators (or) Boolean Operators
- Produces an integer result
  - **Condition True** : Integer value is 1
  - **Condition False** : Integer value is 0
- Compares
  - Values between two variables
  - Values between variables and constants

## 2. 1 Operators in C Contd...

### a) Relational Operators Contd...

**Relational Expression / Boolean Expression** : An expression containing a relational operator

Relational Operations		
Operation	Operations	Example
<	Less than	$a < b$
>	Greater than	$a > b$
$\leq$	Less than or equal to	$a \leq b$
$\geq$	Greater than equal to	$a \geq b$
=	Equal to	$a == b$
$\neq$	Not equal to	$a != b$

## 2. 1 Operators in C Contd...

### a) Relational Operators Contd...

- Consider  $a = 10$  and  $b = 4$ . The relational expression returns the following integer values

Relational Expression	Result	Return Values
$a < b$	False	0
$a > b$	True	1
$a \leq b$	False	0
$a \geq b$	True	1
$a == b$	False	0
$a != b$	True	1

## /\* Program for Relational Operations \*/

```
#include<stdio.h>
int main( )
{
    int a,b;
    printf("Enter the two Values\n");
    scanf("%d%d", &a, &b);
    printf("a>b is %d\n", (a>b));
    printf("a<b is %d\n", (a<b));
    printf("a>=b is %d\n", (a>=b));
    printf("a<=b is %d\n", (a<=b));
    printf("a==b is %d\n", (a==b));
    printf("a!=b is %d\n", (a!=b));
    return 0;
}
```

## ***Output***

4

2

a > b is 1

a < b is 0

a > = b is 1

a < = b is 0

a = = b is 0

a ! = b is 1

## 2. 1 Operators in C Contd...

### b) Logical Operators

- Combines two or more relations
- Used for testing one or more conditions

<b>SYMBOLS</b>	<b>MEANINGS</b>
“&&”	<b>LOGICAL AND:</b> true when all expression are T false otherwise F
“  ”	<b>LOGICAL OR:</b> true when either expression is T false when both are F
“!”	<b>NOT:</b> negation ( T → F ) and vice-versa

## 2. 1 Operators in C Contd...

### b) Logical Operators Contd...

#### □ Logical Expression / Compound Relational Expression :

An expression which combines two or more relational expression

Op1	Op2	Op1 && Op2	Op1    Op2
F (0)	F (0)	F (0)	F (0)
F (0)	T (1)	F (0)	T (1)
T (1)	F (0)	F (0)	T (1)
T (1)	T (1)	T (1)	T (1)

## 2. 1 Operators in C Contd...

### b) Logical Operators Contd...

- Consider  $a = 10$  and  $b = 4$ . The Logical expression returns the following integer values

Relational Expression	Result	Return Values
$a < 5 \&\& b > 2$	True	1
$a < 5 \&\& b < 2$	False	0
$a > 5 \&\& b < 2$	False	0
$a > 5    b < 2$	True	1
$a < 5    b < 2$	False	0
$a > 5    b < 2$	True	1

## /\* Program for Logical Operations \*/

```
#include<stdio.h>
int main( )
{
    int age,height;
    printf("Enter Age of Candidate:\n");
    scanf("%d", &age);
    printf("Enter Height of Candidate:\n");
    scanf("%d", &height);
    if ((age>=18) && (height>=5))
        printf("The Candidate is Selected");
    else
        printf("Sorry, Candidate not Selected");
    return 0;
}
```

## ***Output 1***

Enter Age of Candidate: 18

Enter Height of Candidate: 6

The Candidate is Selected

## ***Output 2***

Enter Age of Candidate: 19

Enter Height of Candidate: 4

Sorry, Candidate not Selected

## 2. 1 Operators in C Contd...

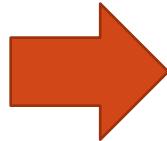
### c) Conditional Operators

- ? and : are the Conditional Operators
- Also called as Ternary Operators
- Shorter form of if-then-else statement
- *Syntax*

**Expression 1 ? Expression 2 : expression 3**

- If expression 1 is true then the value returned will be expression 2
- Otherwise the value returned will be expression 3

```
#include<stdio.h>
int main( )
{
    int x, y;
    scanf("%d", &x);
    y=(x > 5 ? 3 : 4);
    printf("%d", y);
    return 0;
}
```



```
#include<stdio.h>
int main( )
{
    int x, y;
    scanf("%d", &x);
    if(x > 5)
        y=3;
    else
        y=4;
    printf("%d", y);
    return 0;
}
```

*/\* Program for Addition (or) Multiplication \*/*

```
#include<stdio.h>
int main( )
{
    int a, b, result, choice;
    printf("Enter first number \n");
    scanf("%d",&a);
    printf("Enter second number\n");
    scanf("%d",&b);
    printf("Enter 1 for addition or 2 for multiplication\n");
    scanf("%d",&choice);
    result = (choice==1)?a+b:(choice==2)?a*b:printf("Invalid");
    if(choice==1||choice==2)
        printf("The result is %d\n\n",result);
    return 0;
}
```

## ***Output***

Enter first number

10

Enter second number

3

Enter 1 for addition or 2 for multiplication

2

The result is 30

/\* Program to find the maximum of 3 Numbers \*/

```
#include <stdio.h>

int main( )

{
    int a, b, c, max;

    printf("Enter three numbers: ");
    scanf("%d%d%d",&a, &b, &c);
    max = (a > b && a > c) ? a : (b > c) ? b : c;
    printf("\n Maximum between %d, %d and %d = %d", a, b, c, max);
    return 0;
}
```

## ***Output***

Enter three numbers: 30 10 40

Maximum between a, b and c = 40

## 2. 2 Operator Precedence

- ***Operator Precedence*** is used to determine the order of operators evaluated in an expression
  - Every operator has precedence (Priority)
  - Operator with higher precedence is evaluated first and the operator with least precedence is evaluated last
- ***Associativity*** is used when two operators of same precedence appear in an expression
  - Determines the order of evaluation of those operators
  - Associativity can be either Left to Right or Right to Left

## 2.2 Operator Precedence Contd...

- ❑ Operators are listed in descending order of precedence
- ❑ An Expression can contain several operators with equal precedence
  - ❑ Evaluation proceeds according to the associativity of the operator i.e.,
    - ❑ From Right to Left (or)
    - ❑ From Left to Right
- ❑ **Note:** Order of operations is not defined by the language

Precedence order	Operator	Associativity
1	( ) [ ] →	Left to right
2	++ -- - (unary) ! ~ * & sizeof	Right to left
3	* / %	Left to right
4	+ -	Left to right
5	<< >>	Left to right
6	< <= > >=	Left to right
7	= !=	Left to right
8	& (bitwise AND)	Left to right
9	^ (bitwise XOR)	Left to right
10	(bitwise OR)	Left to right

## 2. 3 Expressions using Pre/Post Increment Operator

- Increment operators increase the value of the variable by one
- Decrement operators decrease the value of the variable by one

### □ *Syntax*

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

Decrement operator: `--var_name;` (or) `var_name --;`

### □ Example

Increment operator : `++ i ;    i ++ ;`

Decrement operator : `-- i ;    i -- ;`

## /\* Expressions using Pre-Increment Operator\*/

```
#include<stdio.h>
int main( )
{
    int x,i;
    i=10;
    x=++i;
    printf("x: %d",x);
    printf("i: %d",i);
    return 0;
}
```

### ***Output***

x: 11

i: 11

## /\* Expressions using Post-Increment Operator\*/

```
#include<stdio.h>
int main( )
{
    int x,i;
    i=10;
    x=i++;
    printf("x: %d",x);
    printf("i: %d",i);
    return 0;
}
```

### ***Output***

x: 10

i: 11

## /\* Expressions using Pre-Decrement Operator\*/

```
#include<stdio.h>
int main( )
{
    int x,i;
    i=10;
    x=--i;
    printf("x: %d",x);
    printf("i: %d",i);
    return 0;
}
```

### ***Output***

x: 9

i: 9

## /\* Expressions using Post-Decrement Operator \*/

```
#include<stdio.h>
int main( )
{
    int x,i;
    i=10;
    x=i--;
    printf("x: %d",x);
    printf("i: %d",i);
    return 0;
}
```

### ***Output***

x: 10

i: 9

## /\* Expressions using Increment / Decrement Operators \*/

```
#include<stdio.h>
int main()
{
    int p,q,x,y;
    printf("Enter the value of x \n");
    scanf("%d",&x);
    printf("Enter the value of y \n");
    scanf("%d",&y);
    printf("x=%d\ny=%d\n",x,y);
    p=x++;
    q=y++;
    printf("x=%d\ty=%d\n",x,y);
    printf("x=%d\tq=%d\n",p,q);
    p=--x;
    q=--y;
    printf("x=%d\ty=%d\n",x,y);
    printf("p=%d\tq=%d\n",p,q);
    return 0;
}
```

## ***Output***

Enter the value of x 10

Enter the value of y 20

x = 10

y = 20

x = 11    y = 21

p = 10    q = 20

x = 10    y = 20

p = 10    q = 20

## 2. 4 Expressions using Conditional Operator

- ❑ Any operator is used on three operands or variable is known as **Ternary Operator**
- ❑ It can be represented with ? : . It is also called as **conditional operator**



## /\* Program for Printing Odd or Even Number \*/

```
#include<stdio.h>

int main( )
{
    int num;
    printf("Enter the Number : ");
    scanf("%d",&num);
    (num%2==0)?printf("Even\n"):printf("Odd");
}
```

### ***Output***

Enter the Number : 10

Even

## /\* Program for Eligibility to Vote \*/

```
#include<stdio.h>
int main()
{
    int age;
    printf(" Please Enter your age here: \n ");
    scanf(" %d ", &age);
    (age >= 18) ? printf(" You are eligible to Vote ") : printf(" You are not
                                                                eligible to Vote ");
    return 0;
}
```

### ***Output***

Please Enter your age here: 19

You are eligible to Vote

/\* Program for Finding Biggest of 2 Numbers \*/

```
#include<stdio.h>
int main( )
{
    int a, b, max;
    printf("Enter a and b: ");
    scanf("%d%d", &a, &b);
    max = a > b ? a : b; printf("Largest of the two numbers = %d\n", max);
    return 0;
}
```

### ***Output***

Enter a and b: 10 20

Largest of the two numbers = 20

## **2. 5 Expressions using Assignment Operator**

- ❑ Assignment Operator is used to assign value to an variable
- ❑ Assignment Operator is denoted by equal to sign
- ❑ Assignment Operator is binary operator which operates on two operands
- ❑ Assignment Operator have Two Values – L-Value and R-Value
  - ❑ Operator = copies R-Value into L-Value
- ❑ Assignment Operator have lower precedence than all available operators but has higher precedence than comma Operator

Operator	Example	Equivalent Expression
=	$m = 10$	$m = 10$
+=	$m += 10$	$m = m + 10$
-=	$m -= 10$	$m = m - 10$
*=	$m *= 10$	$m = m * 10$
/ =	$m /=$	$m = m/10$
% =	$m \% = 10$	$m = m \% 10$
<<=	$a <<= b$	$a = a << b$
>>=	$a >>= b$	$a = a >> b$
>>>=	$a >>>= b$	$a = a >>> b$
& =	$a \& = b$	$a = a \& b$
^ =	$a ^ = b$	$a = a ^ b$
=	$a   = b$	$a = a   b$

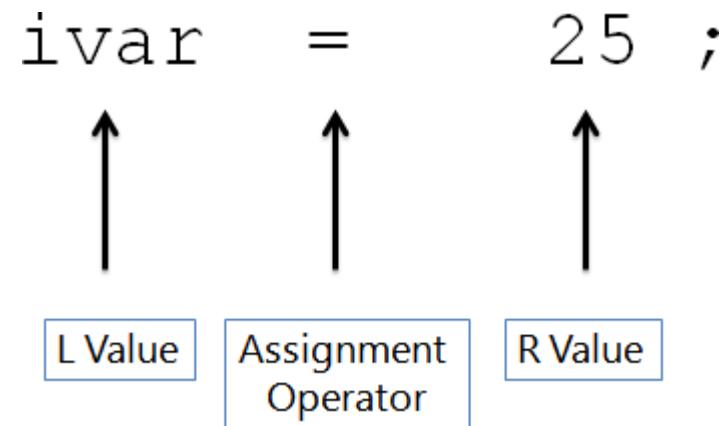
## 2. 6 L-Value and R-Value of Expression

a) L-Value stands for **left value**

- L-Value of Expressions refer to a memory locations
- In any assignment statement L-Value of Expression must be a container(i.e. must have ability to hold the data)
- Variable is the only container in C programming thus L Value must be any Variable.
- L Value cannot be a Constant, Function or any of the available data type in C

## 2. 6 L-Value and R-Value of Expression Contd...

- Diagram Showing L-Value of Expression :



## 2. 6 L-Value and R-Value of Expression Contd...

```
#include<stdio.h>
int main( )
{
    int num;
num = 5;
    return(0);
}
```

```
#include<stdio.h>
int main( )
{
    int num;
5 = num; //Error
    return(0);
}
```

```
#include<stdio.h>
int main( )
{
    const num;
num = 20; //Error
    return(0);
}
```

*Example of L-  
Value Expression*

*L-value cannot be  
a Constant*

*L-value cannot be  
a Constant  
Variable*

## 2. 6 L-Value and R-Value of Expression Contd...

```
#include<stdio.h>
#define MAX 20
int main( )
{
    MAX = 20; //Error
    return(0);
}
```

*L-value cannot be  
a MACRO*

```
#include<stdio.h>
enum {JAN,FEB,MARCH};
int main( )
{
    JAN = 20; //Error
    return(0);
}
```

*L-value cannot be  
a Enum Constant*

## 2. 6 L-Value and R-Value of Expression Contd...

- b) R Value stands for **Right value** of the expression
- In any **Assignment statement** R-Value of Expression must be anything which is capable of returning Constant Expression or Constant Value

ivar = 25 ;



L Value

Assignment  
Operator

R Value

## 2. 6 L-Value and R-Value of Expression Contd...

### Examples of R-Value of Expression

Variable	Constant
Function	Macro
Enum Constant	Any other data type

- R value may be a Constant or Constant Expression
- R value may be a MACRO
- R Value may be a variable

## 2. 7 Control Statements

- Also called as Conditional Statement
- Decides order of execution based on conditions
- Helps repeat a group of statements
- Modifies control flow of program
- Decision Making
- Branching

## 2. 7 Control Statements Contd...

### *Types of Branching Statements*

- a) if statement
  - i. Simple if
  - ii. if...else statement
  - iii. nested if...else statement
  - iv. else...if statement
- b) switch statement
- c) goto statement

## 2. 7 Control Statements Contd...

### a) if statement

- Condition "True" - Statement block will be executed
- Condition "False" - Statement block will not be executed.

#### Variations

- i. Simple if
- ii. if...else statement
- iii. nested if...else statement
- iv. else...if statement

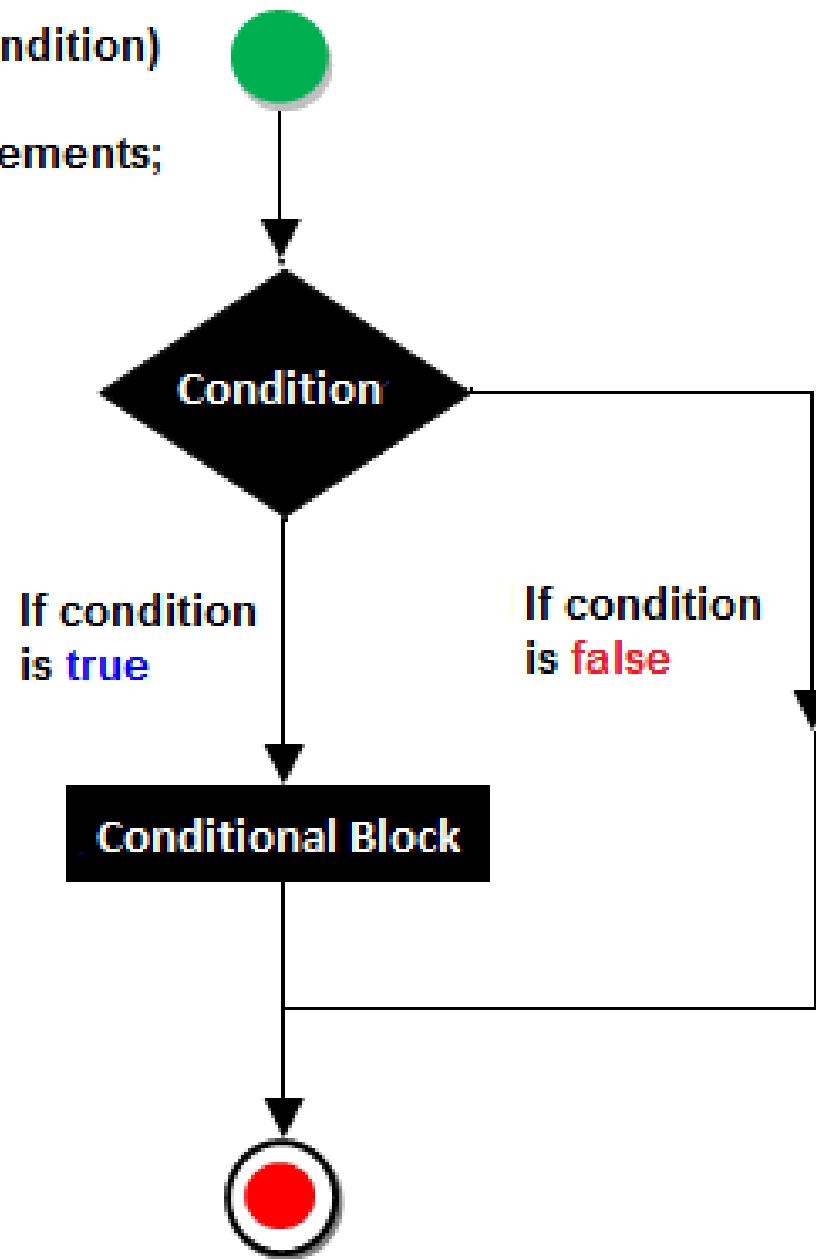
## 2. 7 Control Statements Contd...

### i. Simple if statement

- Basic if statement
- What is a condition?
- Executes statement block only if condition is true
- Syntax

```
if (condition)
{
    Statements;
}
```

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



*/\* Simple if – Program to check whether a number is Odd\*/*

```
#include<stdio.h>
int main( )
{
    int number;
    printf("Enter the Number: ");
    scanf("%d, &number);
    if(number%2==0)
    {
        printf("The Number is Even");
    }
    return 0;
}
```

### *Output*

Enter a value : 10342

The number is Even

## 2. 7 Control Statements Contd...

### **□ Try it Out Yourself ! Write a C program to:**

- 1) Check whether the given number is Even
- 2) To check whether the given number is Greater
- 3) To check whether the given number is Smaller
- 4) To check whether the given number is positive
- 5) To check whether the given number is negative
- 6) To check whether the given number is zero
- 7) To check whether two numbers are equal

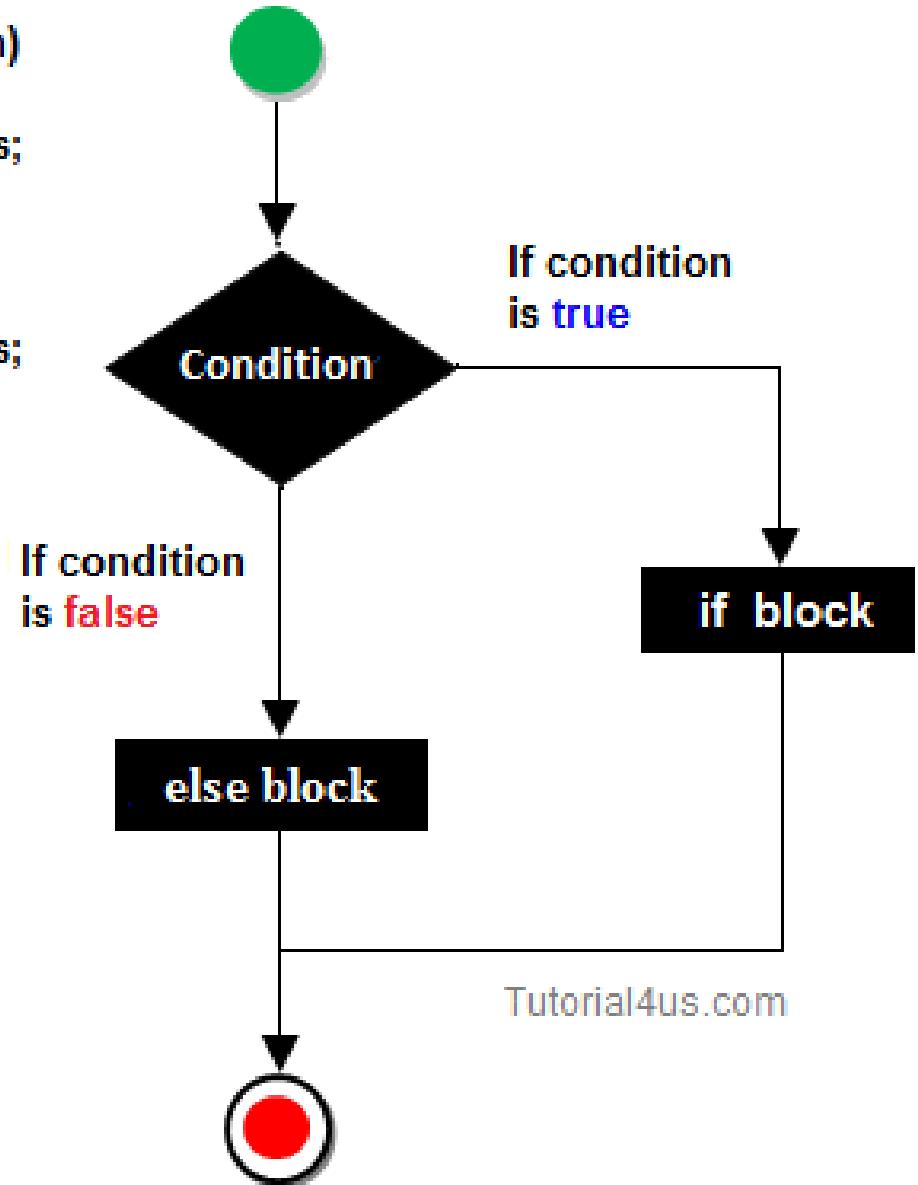
## **2. 7 Control Statements Contd...**

### **ii. If else statement**

- Extension of basic if statement
- Takes care of True and False condition
- Number of Statement Blocks - 2
  - Block 1 – True Condition
  - Block 2 – False Condition

```
if (condition)
{
    Statements;
}
Else
{
    Statements;
}
```

```
if( condition)
{
    statements;
}
else
{
    statements;
}
```



/\* if else -To check whether a number is Odd or Even\*/

```
#include<stdio.h>
int main( )
{
    int number;
    printf("Enter the Number: ");
    scanf("%d, &number);
    if(number%2==0)
    {
        printf("The Number is Even");
    }
    else
    {
        printf("The Number is Odd");
    }
    return 0;
}
```

## ***Output 1***

Enter the Number : 10341

The number is Odd

## ***Output 2***

Enter the Number : 10342

The number is Even

## 2. 7 Control Statements Contd...

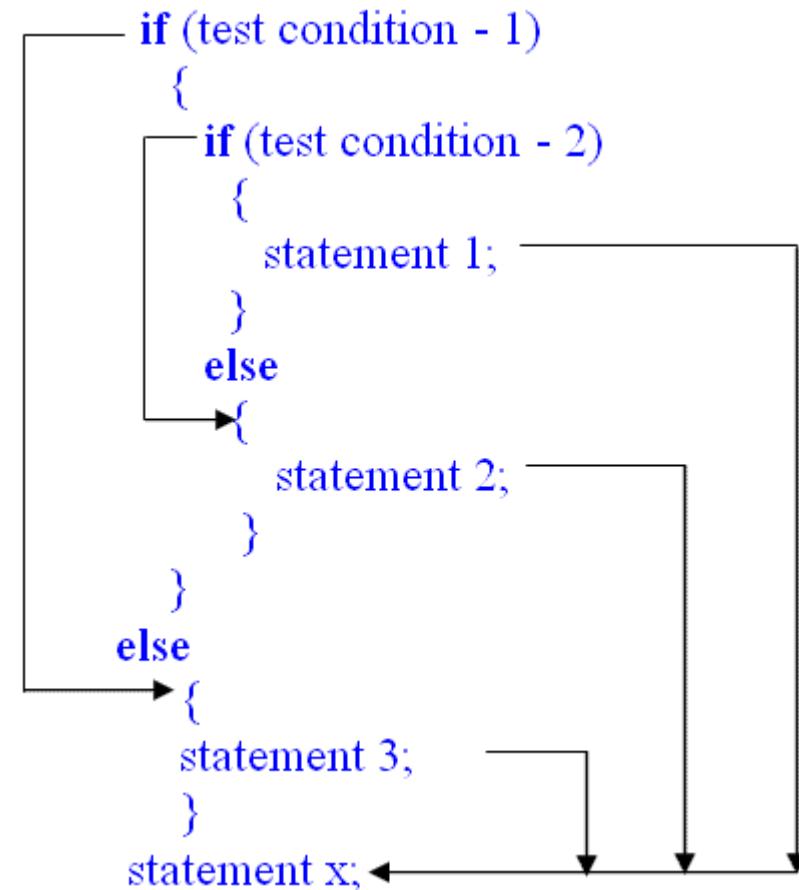
### **□ Try it Out Yourself ! Write a C program to:**

- 1) To check whether the given number is Greater or Smaller
- 2) To check whether the given number is +ve or -ve
- 3) To check whether two numbers are equal or not

## 2.7 Control Statements Contd...

### iii. Nested if else statement

- ❑ Used when a series of decisions are involved
- ❑ Makes a choice between several alternatives
- ❑ New if else statement block is used within existing if else statement block



/\*Program for Nested if else \*/

```
#include <stdio.h>

void main( )
{
    char username;
    int password;
    printf("Username:");
    scanf("%c",&username);
    printf("Password:");
    scanf("%d",&password);
```

```
if(username=='a')
{
    if(password==12345)
    {
        printf("Login successful");
    }
    else
    {
        printf("Password is incorrect, Try again.");
    }
}
else
{
    printf("Username is incorrect, Try again.");
}
return 0;
}
```

## ***Output 1***

Username: a

Password: 12345

Login Successful

## ***Output 2***

Username: a

Password: 54321

Password is incorrect, Try again.

## ***Output 3***

Username: b

Password: 54321

Username is incorrect, Try again.

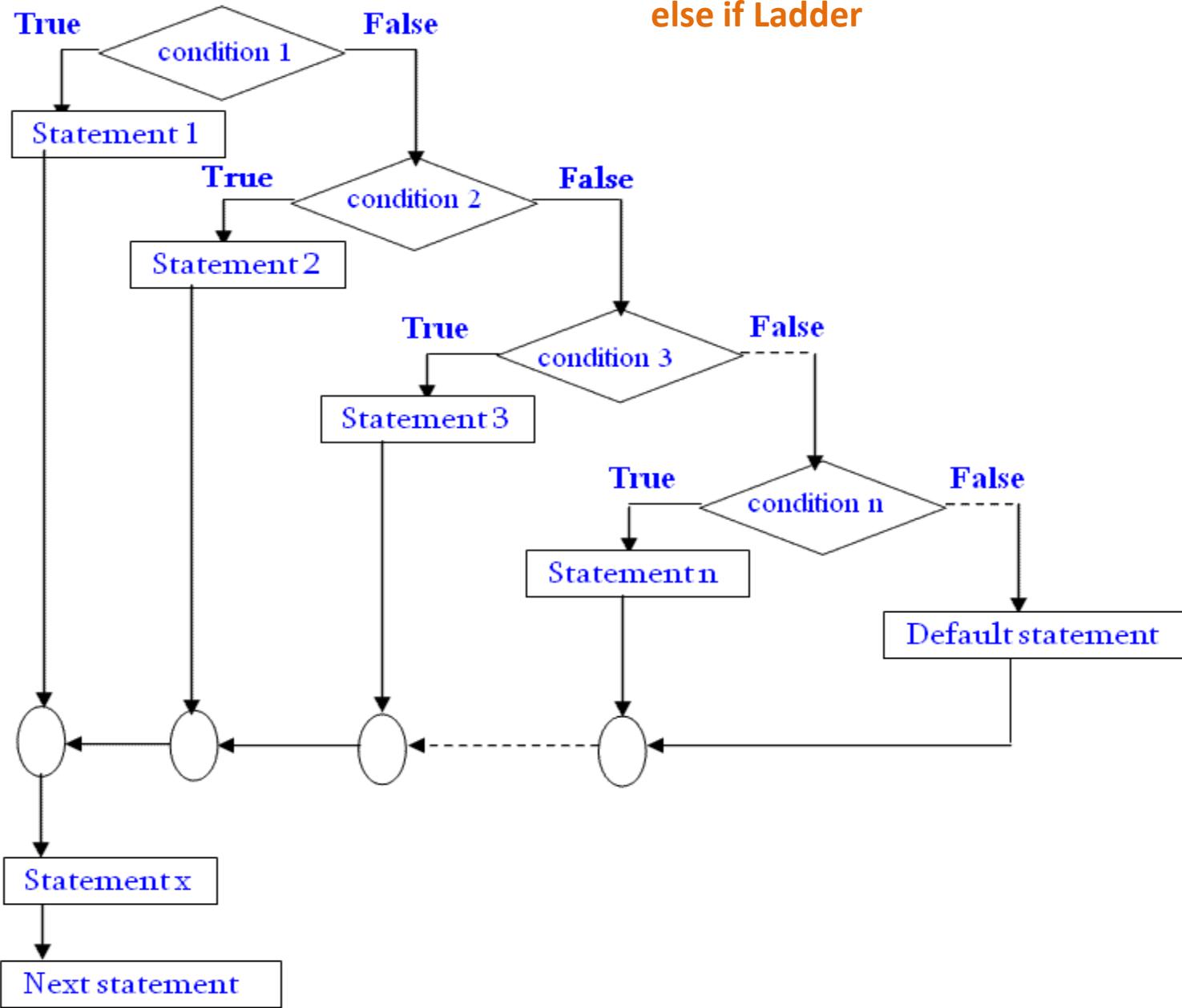
## 2. 7 Control Statements Contd...

- ❑ **Step 1:** First if condition will be true, if the user has typed 'a' as a username then the program control moves to second if condition and checks for the password
  - ❑ if it true it will print 'login successful'
  - ❑ else it will execute block statement 'Password is Incorrect, Try again.'
- ❑ **Step 2:** If the first if condition is false then it executes last else block thus printing 'Username is Incorrect, Try again.'

## 2. 7 Control Statements Contd...

❑ **Step 3:** In this above example we have use username as single character to use multiple character username we need to use string data type

## else if Ladder



## /\*Program for if else ladder\*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a;
    printf("Enter a Number: ");
    scanf("%d",&a);
    if(a > 0)
    {
        printf("Given Number is Positive");
    }
    else if(a == 0)
    {
        printf("Given Number is Zero");
    }
    else if(a < 0)
    {
        printf("Given Number is Negative");
    }
    getch();
}
```

## 2. 7 Control Statements Contd...

### b) Switch statement

- Allows to make decisions from a number of choices
- Also called as Switch-Case-Default Statement
- Faster than nested if else statement
- Easier to understand
- **Rules for writing switch ( ) statement**
  - Expression in switch must be an integer value or a character constant
  - No real numbers used in Expression

## **2. 7 Control Statements Contd...**

- Each case block and default block must end with break statements
- Default is optional
- Case keyword must end with colon ( : )
- Default may be placed anywhere in the switch
- No two case constants are identical

## 2. 7 Control Statements Contd...

```
switch(variable or expression)
{
    case constant 1:
        statements;
    break;
    ....
    case constant N;
        statements;
    break;
    default:
        statements;
}
```

## /\* Program for Switch Case \*/

```
#include<stdio.h>
int main( )
{
    int a, b, choice;
    printf("\nEnter Two Numbers:");
    scanf("%d%d", &a,&b);
    printf("\n Enter 1 for Addition");
    printf("\n Enter 2 for Subtraction");
    printf("\n Enter 3 for Multiplication");
    printf("\n Enter 4 for Division");
    printf(" Enter your Choice");
    scanf("%d",&choice);
```

```
switch (choice)
{
    case 1:
        printf("Sum is : %d", a+b);
        break;

    case 2:
        printf("Difference is : %d", a-b);
        break;

    case 3:
        printf("Multiplication is : %d", a*b);
        break;

    case 4:
        printf("Difference is : %d", a/b);
        break;
}
```

```
        default:  
            printf("Invalid Choice:");  
        }  
    getch();  
}
```

Enter two numbers

20

10

Enter 1 for Addition

Enter 2 for Subtraction

Enter 3 for Multiplication

Enter 4 for Division

Enter your Choice: 3

Product is : 200

## 2. 7 Control Statements Contd...

### **❑ Nested Switch statement**

- ❑ Inner switch( ) can be a part of an outer switch( )
- ❑ Inner switch( ) and outer switch( ) case constants may be the same

## /\* Program for Nested Switch Case \*/

```
#include<stdio.h>

int main( )
{
    int square, i, n, fact = 1, choice;

    printf("\n Enter Any Number: ");
    scanf("%d", &n);

    printf(" 1. Square \n");
    printf(" 2. Factorial \n");
    printf(" 3. Find Odd or Even \n");
    printf(" 4. Exit \n");
    printf(" Enter your Choice");
    scanf("%d", &choice);
```

```
switch (choice)
{
    case 1:
        square = n * n;
        printf("The Square of the Given number is %d\n",
               square);
        break;
    case 2:
        for(i=1;i<=n;i++)
        {
            fact = fact * i;
        }
        printf("The Factorial of a given number is %d\n", fact);
        break;
```

```
switch (n%2)
{
    case 0:
        printf("Given Number is Even\n");
    case 1:
        printf("Given Number is Odd\n");
    }
    case 3:
        exit(0);
    default:
        printf("Invalid Choice. Please try again\n");
    }
return 0;
}
```

Enter any number

5

1. Square
2. Factorial
3. Find Odd or Even
4. Exit

Enter your choice

2

The factorial of a given number is: 120

## 2. 7 Control Statements Contd...

### c) The goto statement

❑ Transfers control from one point to another

❑ Syntax

    goto label;

                statements;

.....

    label

                statements;

## 2. 8 Looping Statements

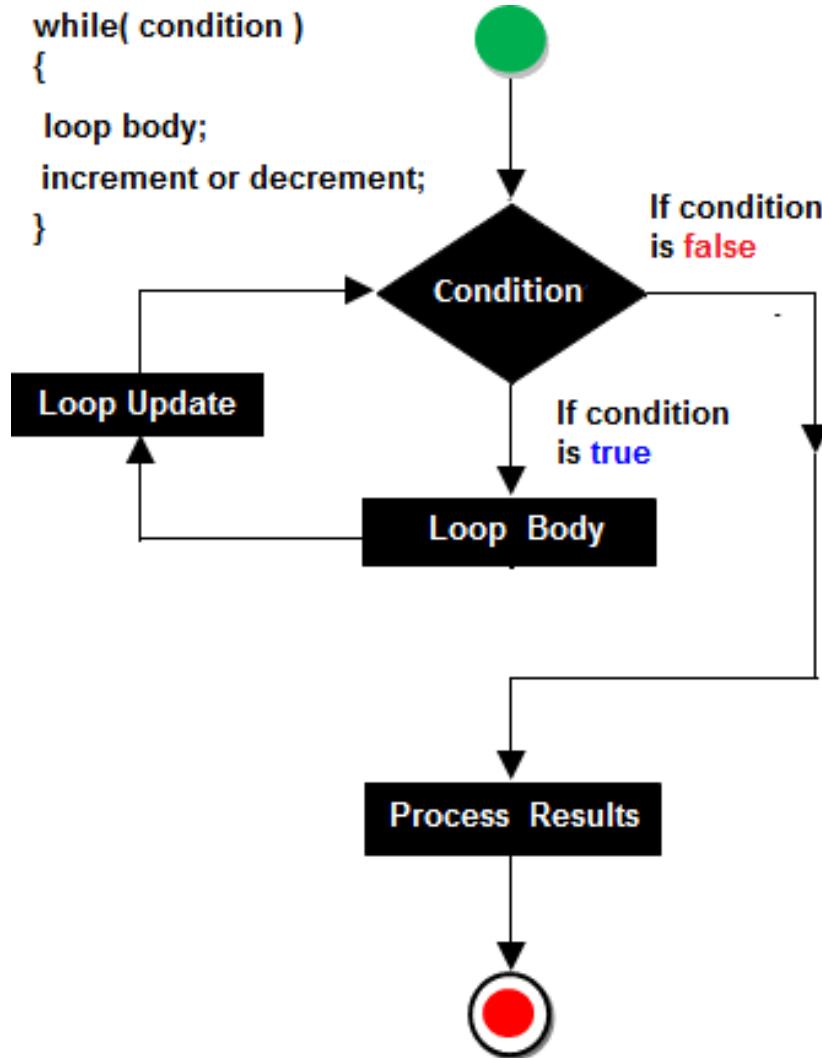
- Loop – A segment of the program that is executed repeatedly until a condition is satisfied
- Classification – Entry Controlled & Exit Controlled
- **Types**
  - a) while do loop
  - b) do while loop
  - c) for loop
    - i. Nested for loop

## 2. 8 Looping Statements Contd...

### a) The While Loop

- Simplest looping structure in C
- Statements in the program may need to repeat for many times. e.g., calculate the value of  $n!$
- Loop consists of two segments
  - Control Statement
  - Body of the Loop
- How while loop works?

```
while( condition )  
{  
    loop body;  
    increment or decrement;  
}
```



Initialize loop counter variable;  
while (condition)  
{  
 Statements;  
 increment / Decrement loop  
 counter variable;  
}

*/\* Program to Add 3 Numbers \*/*

```
#include<stdio.h>
int main( )
{
    int a, b, c, sum;
    printf("\n Enter the Three Numbers: ");
    scanf("%d%d%d", &a,&b,&c);
    sum = a+b+c;
    printf("The sum of 3 Numbers is %d", sum);
    return 0;
}
```

### ***Output***

Enter the Three Numbers: 10 20 30

The sum of 3 Numbers is: 60

## /\* Program to Add n Numbers \*/

```
#include<stdio.h>
int main( )
{
    int i=1,n, sum=0;
    printf("\n Enter the value for n: ");
    scanf("%d", &n);
    while (i<=n)
    {
        sum = sum + i;
        i++;
    }
    printf("The sum of n Numbers is: %d", sum);
    return 0;
}
```

## ***Output***

Enter the value for n: 5

The sum of n Numbers is: 15

## **2. 8 Looping Statements Contd...**

### **□ Try it Out Yourself ! Write a C program to:**

- 1) To print all even numbers from 1 to 100
- 2) To print all even numbers from 1 to n
- 3) To print table for any number
- 4) To calculate the sum of its digits
- 5) To check whether the entered number is Prime or not
- 6) To get a number as input and print it in reverse.
- 7) To check whether the number is Armstrong number

## 2. 8 Looping Statements Contd...

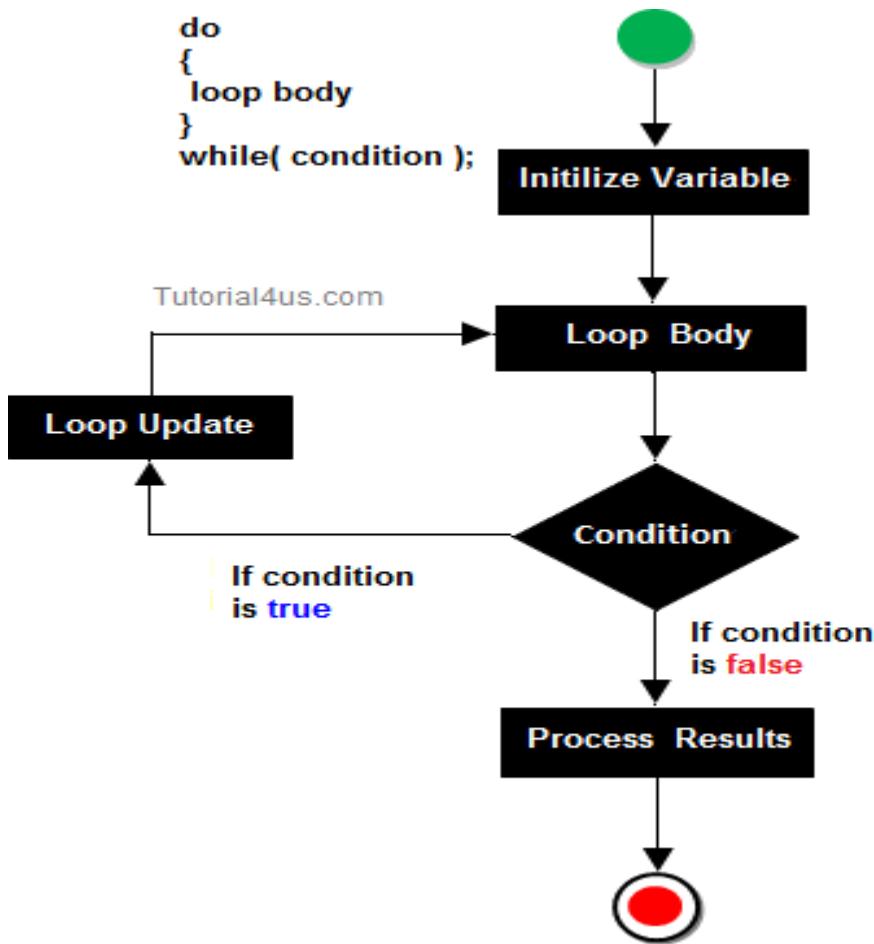
### b) The Do While Loop

□ The body of the loop is executed at least once

□ Syntax

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

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

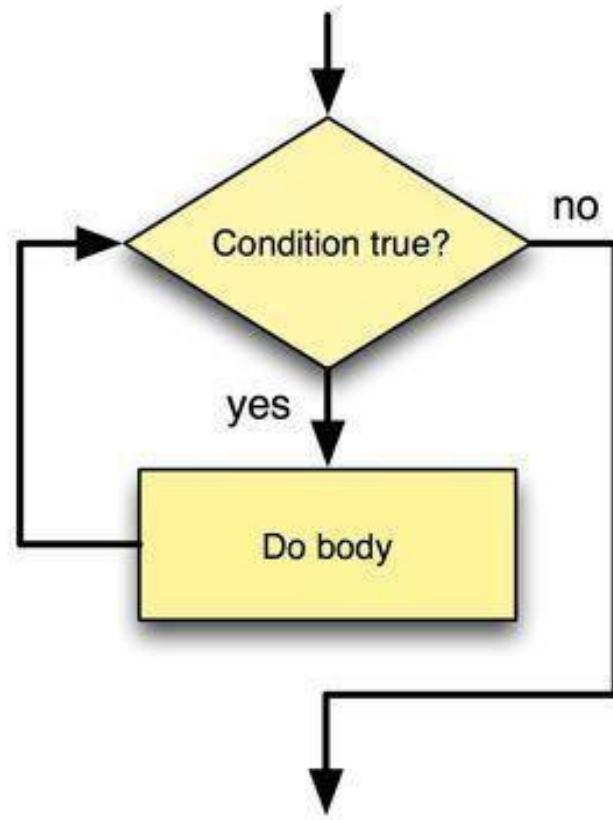


Initialize loop counter variable;  
do  
{  
 Statements;  
 increment / Decrement loop  
 counter variable;  
}  
while (condition)

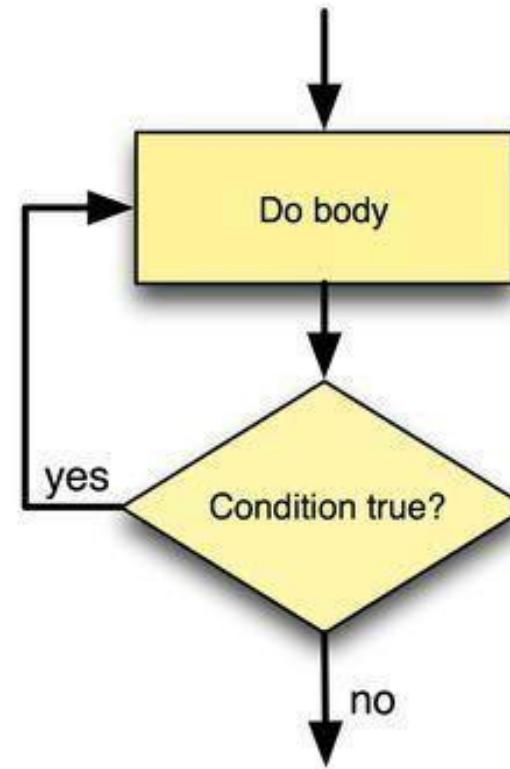
## 2. 8 Looping Statements Contd...

While Do Loop	Do While Loop
Entry Controlled Loop	Exit Controlled Loop
Test condition is checked before body of the loop is executed	Test condition is checked after the body of the loop is executed
Loop will not be executed if condition is false	Loop will be executed at least once even if condition is false
Top tested loop	Bottom tested loop

## 2. 8 Looping Statements Contd...



while flowchart



do/while flowchart

## 2. 8 Looping Statements Contd...

### c) The for loop

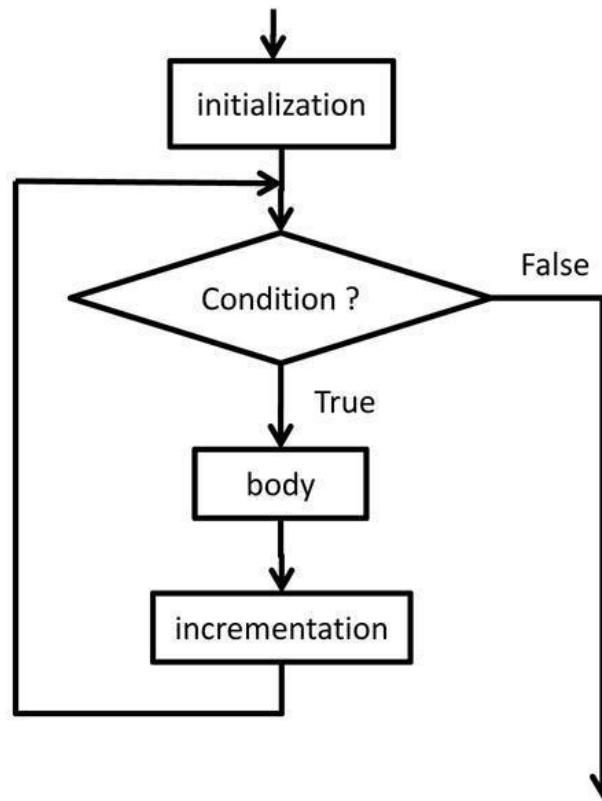
- Most commonly and popularly used loop structure
- Structure of the for loop
  - Initialize loop counter variable
  - Check for condition
  - Increment / Decrement the loop counter variable

#### □ Syntax

*for(initialization; condition; increment / decrement)*

## 2. 8 Looping Statements Contd...

```
for( initialization; condition; incrementation )  
    body;
```



## 2. 8 Looping Statements Contd...

### □ Examples

i.    for(*i* = 0; *i* < *n*; *i*++)

{

    Statements;

}

ii.    for(*count* = 0; *count* > *n*; *count*--)

{

    Statements;

}

/\* Program to Add n Numbers using for loop \*/

```
#include<stdio.h>
int main( )
{
    int i, n, sum=0;
    printf("\n Enter the value for n: ");
    scanf("%d", &n);
    for (i =1; i<=n; i++)
    {
        sum = sum + i;
    }
    printf("The sum of n Numbers is: %d", sum);
    return 0;
}
```

## ***Output***

Enter the value for n: 5

The sum of n Numbers is: 15

## **2. 8 Looping Statements Contd...**

### **□ Try it Out Yourself ! Write a C program to:**

- 1) To print all even numbers from 1 to 100
- 2) To print all even numbers from 1 to n
- 3) To print table for any number
- 4) To calculate the sum of its digits
- 5) To check whether the entered number is Prime or not
- 6) To get a number as input and print it in reverse.
- 7) To check whether the number is Armstrong number

# break

The break statement ends the loop immediately when it is encountered.

Its syntax is:

```
break;
```

The break statement is almost always used with if...else statement inside the loop.

```
while (testExpression) {  
    // codes  
    if (condition to break) {  
        break;  
    }  
    // codes  
}  
  
do {  
    // codes  
    if (condition to break) {  
        break;  
    }  
    // codes  
}  
while (testExpression);
```

```
for (init; testExpression; update) {  
    // codes  
    if (condition to break) {  
        break;  
    }  
    // codes  
}
```

## continue

The continue statement skips the current iteration of the loop and continues with the next iteration.

Its syntax is: continue;

The continue statement is almost always used with the if...else statement.

```
→ while (testExpression) {  
    // codes  
    if (testExpression) {  
        continue;  
    }  
    // codes  
}
```

```
do {  
    // codes  
    if (testExpression) {  
        continue;  
    }  
    // codes  
}  
→ while (testExpression);
```

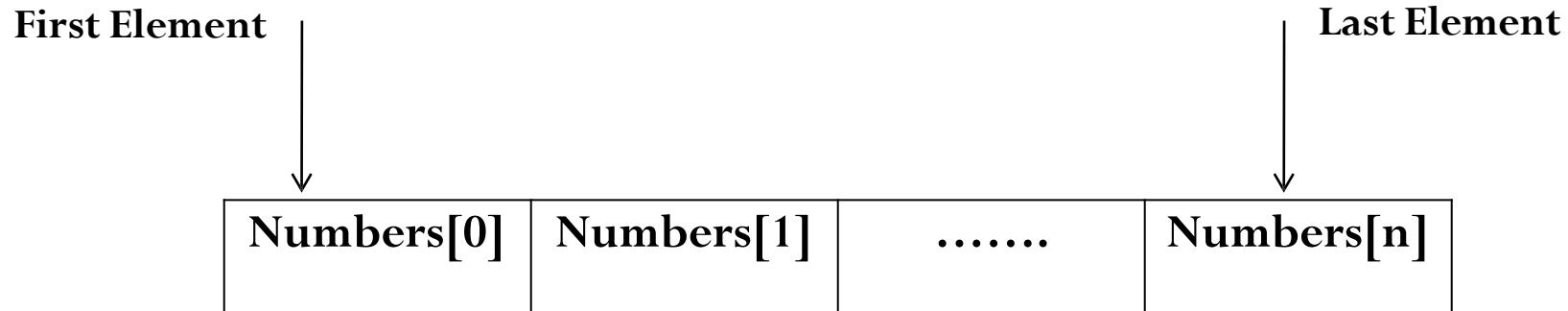
```
→ for (init; testExpression; update) {  
    // codes  
    if (testExpression) {  
        continue;  
    }  
    // codes  
}
```

## 2. 9 Arrays

### □ Definition

An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations.

- ✓ Array is used to store a collection of data
- ✓ Array is a collection of variables of the same type.



## 2. 9 Arrays Contd...

- Need for Arrays
  - Used to represent a list of numbers / names
  - Used to represent tabular data in 2, 3 or more dimensions
  - Important Data Structure in any programming language
- **Definition**
  - Collection of elements of similar data types
  - Each element is located in separate memory locations
  - Each Array element share a common name

## 2. 9 Arrays Contd...

### □ Characteristics of Arrays

- All elements in the arrays share a common name
- Elements distinguished by index number
- Index (or) element number of an array plays vital role for calling each element
- Specific array elements can be modified
- Value of array element can be assigned to variables
- Array elements stored in continuous memory locations

## 2. 9 Arrays Contd...

- Storage space for array depends on its data type and size

***Total bytes = sizeof (Data type) x Size of Array***

- Example

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

Total bytes = sizeof (int) x 5 = 2 x 5 = 10 bytes

## 2. 9 Arrays Contd...

### a) Array Declaration

- Syntax

**Datatype arrayname [size/subscript];**

- **Data Type:** int, float, double, char, structure, union

- **Array Name:** Name given to the Array variable

- **Size / Subscript:** Number of values an Array can hold

- **Examples**

int numbers[5];

float marks[50];

char name[20];

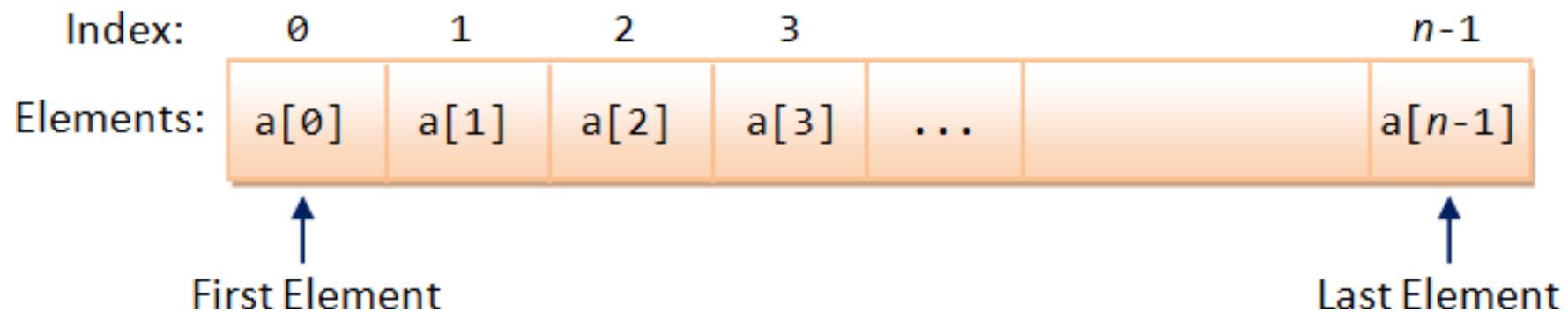
double a[i];

## 2.9 Arrays Contd...

### □ Illustration

```
int a[n];
```

Array Name: a  
Array Length: n



## 2. 9 Arrays Contd...

- ❑ **Static Array:** Array size (range) declared in the program
- ❑ **Dynamic Array:** Array size given during execution

STATIC ARRAYS	DYNAMIC ARRAYS
Range / Size of an array included in the Array definition	Range / Size of an array not included in the Array definition
Static Arrays cannot be changed	Dynamic Arrays can be changed

## 2. 9 Arrays Contd...

### b) Array Initialization

- **Initialization:** Assigning values to array elements
  - Values specified in curly braces separated by commas
- Examples

```
int a[ 5] = {1, 2, 3, 4, 5};
```

```
float b[3] = { 40.5, 59.0, 98.5};
```

```
char name[6] = " IST";
```

- Array element index start from 0

## 2. 9 Arrays Contd...

- Array elements are called by array names followed by the element numbers
- `int a[ 5 ] = {1, 2, 3, 4, 5};`
  - a[0] refers to 1<sup>st</sup> element i.e., 1
  - a[1] refers to 2<sup>nd</sup> element i.e., 2
  - a[2] refers to 3<sup>rd</sup> element i.e., 3
  - a[3] refers to 4<sup>th</sup> element i.e., 4
  - a[4] refers to 5<sup>th</sup> element i.e., 5

## 2. 9 Arrays Contd...

### c) Getting Input for Arrays

- Use for loops to get input in arrays
- Use for loops with regard to the Array's dimension

- Input for One Dimensional Arrays – 1 for

- loop   for(*i* = 0; *i* < 5; *i*++)

- {

- scanf*(“%d”, &*a*[*i*]);

- }

## 2. 9 Arrays Contd...

- Input for Two Dimensional Arrays – 2 for loops

```
for(i=0;i<5;i++)
```

```
{
```

```
    for(j=0;j<5;j++)
```

```
{
```

```
        scanf("%d",&a[i][j]);
```

```
}
```

```
}
```

## 2. 9 Arrays Contd...

### d) Printing Output in Arrays

- Use for loops to print array output
- Use for loops with regard to the Array's dimension
  - Printing One Dimensional Array Output – 1 for

loop   for(*i*=0;*i*<5;*i*++)

{

    printf("%d",*a*[*i*

}

## 2. 9 Arrays Contd...

- ❑ Printing Two Dimensional Array Output – 2 for loops

```
for(i = 0; i < 5; i++)
```

```
{
```

```
    for(j=0; j < 5; j++)
```

```
{
```

```
        printf("%d", a[i][j]);
```

```
}
```

```
}
```

## /\* Program 1 : Array Declaration & Initialization \*/

```
#include<stdio.h>
int main( )
{
    int i, arr[5];
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    arr[3] = 40;
    arr[4] = 50;
    for(i=0; i<=n; i++)
    {
        printf("%d\n", a[i]);
    }
    return 0;
}
```

## *Output*

10

20

30

40

50

## /\* Program 2 : Array Declaration & Initialization \*/

```
#include<stdio.h>

int main( )
{
    int i, arr[5];
    arr[5] = {10, 20, 30, 40, 50};
    for(i=0; i<=n; i++)
    {
        printf("%d", a[i]);
    }
    return 0;
}
```

## *Output*

10

20

30

40

50

## /\* Program 3 : Array Declaration & Initialization \*/

```
#include<stdio.h>

int main( )
{
    int i, n, arr[5];
    scanf("%d", &n);
    printf("Enter the Elements of Array\n");
    for(i=0; i<n; i++)
    {
        scanf("%d", &a[i]);
    }
    printf("The Elements of the Array are\n");
```

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

### ***Output***

Enter the Elements of the Array

10 20 30 40 50

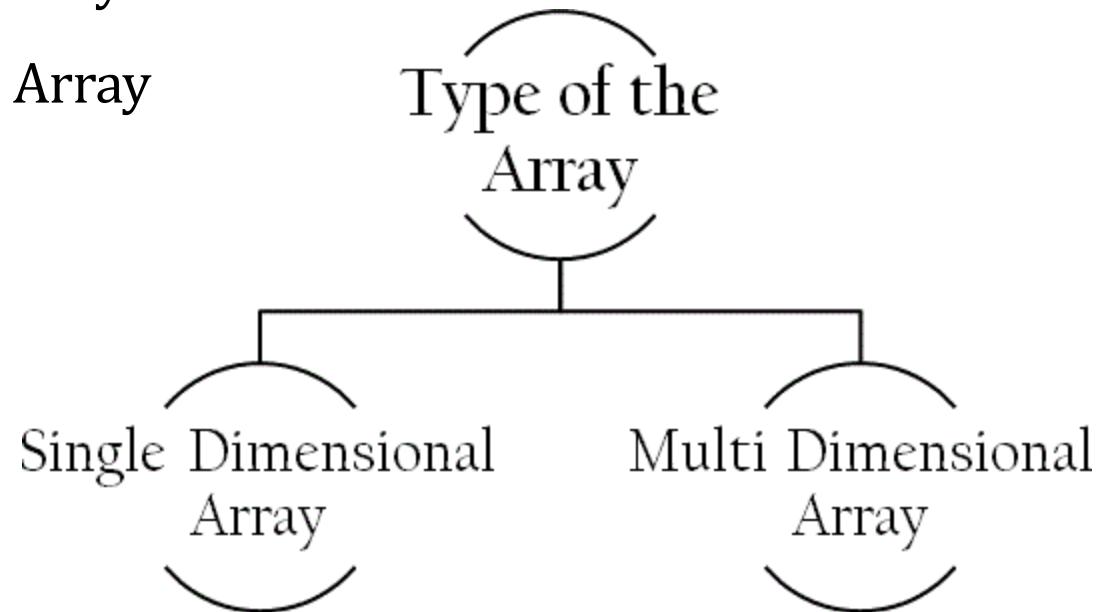
The Elements of the Array are

10 20 30 40 50

## 2. 9 Arrays Contd...

### e) Classification of Arrays

- i. One-Dimensional Array
- ii. Two-Dimensional Array
- iii. Multi-Dimensional Array



## 2. 9 Arrays Contd...

### i. One Dimensional Array

- Data stored under a single variable using one subscript
- 1-D Array Declaration – Syntax

*datatype arrayname [size/subscript];*

□ **Example:** int a [5];

- 1-D Array initialization – Syntax

*datatype arrayname [size] = { list of values};*

**Example:** int a [5] = { 10, 20, 30, 40, 50};

## /\* Program 1 : One Dimensional Array \*/

```
#include<stdio.h>
int main ( )
{
int a[10], n, i, sum;
clrscr();
printf("Enter the Number of Elements\n");
scanf("%d", &n);
for(i = 0; i < n; i++)
{
    scanf("%d", & a [i]);
}
sum = 0;
for(i = 0; i < n; i++)
```

**a [10]**

a [0]	40
a [1]	22
a [2]	34
a [3]	12
a [4]	64
a [5]	
a [6]	
a [7]	
a [8]	
a [9]	

**n**

5

**i**

0

**sum**

0

## /\* Program 1 : One Dimensional Array \*/

```
{  
    sum = sum + a[i];  
}  
printf("The Sum is: %d", sum);  
return 0;  
}
```

### **Output**

Enter the Number of Elements

5

40 22 34 12 64

The Sum is 182

a [10]

a [0]	40
a [1]	22
a [2]	34
a [3]	12
a [4]	64
a [5]	
a [6]	
a [7]	
a [8]	
a [9]	

n

5

i

4

sum

182

## /\* Program 2 : 1-D Array for Sorting \*/

```
#include<stdio.h>
int main( )
{
    int i, j, temp, n, a[10];
    printf("Enter the Number of Elements:");
    scanf("%d", &n);
    printf("Enter the Elements to be Sorted\n");
    for(i=0; i<n; i++)
    {
        scanf("%d\n", &a[i]);
    }
    for(i=0; i<n; i++)
    {
        for(j=i+1; j<n; j++)
        {

```

```
        if(a[i] >a[j])
        {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }

print("The Sorted Elements are: \n");
for(i=0; i<n; i++)
{
    printf("%d\n", a[i]);
}
return 0;
}
```

## ***Output***

Enter the Number of Elements:5

Enter the Elements to be Sorted

25

12

45

68

7

The Sorted Elements are:

7

12

25

45

68

**THANK YOU**

# 18CSS101J – Programming for Problem Solving

## Unit III

## **UNIT III**

Initializing and accessing of 2D Array – Initializing multidimensional Array – Array programs 2D – Array contiguous memory – Array advantages and Limitations – Array construction for real-time application common programming errors – **String Basics** – String Declaration and Initialization – String Functions: gets(), puts(), getchar(), putchar(), printf() - String Functions: atoi, strlen, strcat strcmp – String Functions: sprintf, sscanf, strrev, strcpy, strstr, strtok – Arithmetic characters on strings.

## **UNIT III**

- Functions declaration and definition : Types: Call by Value, Call by Reference – Function with and without Arguments and no Return values – Functions with and without Arguments and Return Values – Passing Array to function with return type – Recursion Function

## 3.1 INITIALIZING AND ACCESSING OF 2D ARRAY

### 3.1.1 ACCESSING OF 2D ARRAY

- Two Dimensional Array requires ***Two Subscript*** Variables
- Two Dimensional Array stores the values in the form of matrix.
- One Subscript Variable denotes the “***Row***” of a matrix.
- Another Subscript Variable denotes the “***Column***” of a matrix.

### 3.1.2 INITIALIZING OF 2D ARRAY

An array of two dimensions can be declared as follows:

***data\_type array\_name[size1][size2];***

Here ***data\_type*** is the name of some type of data, such as int. Also, ***size1*** and ***size2*** are sizes of the array’s first and second dimensions respectively.

## DECLARATION AND USE OF 2D ARRAY

*int a[3][4];*

Use :

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

## MEANING OF 2D ARRAY

- Matrix is having 3 rows ( i takes value from 0 to 2 )
- Matrix is having 4 Columns ( j takes value from 0 to 3 )
- Above Matrix  $3 \times 4$  matrix will have 12 blocks having 3 rows & 4 columns.
- Name of 2-D array is 'a' and each block is identified by the row & column number.
- Row number and Column Number Starts from 0.
- If  $a[0][0]$  means 0<sup>th</sup> row 0<sup>th</sup> column,  $a[0][1]$  means 0<sup>th</sup> row 1<sup>st</sup> column,  
 $a[0][2]$  means 0<sup>th</sup> row 2<sup>nd</sup> column like wise it goes on.....

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

## INITIALIZING 2D ARRAY

- Row wise assignment
- Combine assignment
- Selective assignment

### Method 1 : INITIALIZING ALL ELEMENTS ROWWISE

For initializing 2D Array we can need to assign values to each element of an array using the below syntax.

```
int a[3][2] = {  
    { 1 , 4 },  
    { 5 , 2 },  
    { 6 , 5 }  
};
```

Consider the below program –

```
#include<stdio.h>
int main() {
    int i, j;
    int a[3][2] = { { 1, 4 },
                    { 5, 2 },
                    { 6, 5 } };
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 2; j++) {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

**Output :**

1 4  
5 2  
6 5

We have declared an array of size  $3 \times 2$ , It contain overall 6 elements.

Row 1 : { 1 , 4 },

Row 2 : { 5 , 2 },

Row 3 : { 6 , 5 }

We have initialized each row independently

$a[0][0] = 1$

$A[0][1] = 4$

## Method 2 : COMBINE ASSIGNMENT

Initialize all Array elements but initialization is much straight forward. All values are assigned sequentially and row-wise

```
int a[3][2] = {1 , 4 , 5 , 2 , 6 , 5 };
```

So here it automatically assigns that number 3 has row and number 2 has column.

# Output :

1 4

Consider the below program –

```
#include <stdio.h>
int main() {
    int i, j;
    int a[3][2] = { 1, 4, 5, 2, 6, 5 };
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 2; j++) {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    } return 0;
}
```

5 2  
6 5

## Method 3: SELECTIVE ASSIGNMENT

```
int a[3][2] = {  
    { 1 },  
    { 5 , 2 },  
    { 6 }  
};
```

Now we have again going with the way 1 but we are removing some of the elements from the array. In this case we have declared and initialized 2-D array like this

# Consider the below program –

```
#include <stdio.h>
int main() {
    int i, j;
    int a[3][2] = { { 1 },
                    { 5, 2 },
                    { 6 }};
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 2; j++) {
            printf("%d ", a[i][j]);
        } printf("\n");
    }
    return 0;
}
```

**Output :**

1 0

5 2

6 0

## **3.2 INITIALIZING MULTIDIMENSIONAL ARRAY**

Arrays with more than one dimension are called multidimensional arrays.

**Declaration of three-Dimensional arrays:**

*data\_type array\_name[size1][size2][size3];*

Arrays do not have a shaped like squares and cubes; each dimension of the array be given in different size as follows:

Eg: int non\_cube[2][[6][8];

You can initialize a three dimensional array in a similar way like a two dimensional array. Here's an example,

```
int test[2][3][4] = {
    { {3, 4, 2, 3}, {0, -3, 9, 11}, {9, 1, 6, 2} },
    { {13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9} }
};
```

In this program we mention size1 – row, size2 – column, size3 – number of elements in that array.

C Program to store values entered by the user in a three-dimensional array and display it.

```
#include <stdio.h>

int main()
{
    int i, j, k, test[2][3][2]; // this array can store 12 elements
    printf("Enter 12 values: \n");
    for(i = 0; i < 2; ++i) {
        for (j = 0; j < 3; ++j) {
            for(k = 0; k < 2; ++k ) {
                scanf("%d", &test[i][j][k]);
            }
        }
    }
}
```

```
}

printf("\nDisplaying values:\n");//Displaying values with
proper index.

for(i = 0; i < 2; ++i) {
    for (j = 0; j < 3; ++j) {
        for(k = 0; k < 2; ++k ) {
            printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
        }
    }
}
return 0;
}
```

**OUTPUT:**

**Enter 12 values:**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**Displaying Values:**

**test[0][0][0] = 1**

**test[0][0][1] = 2**

**test[0][1][0] = 3**

**test[0][1][1] = 4**

**test[0][2][0] = 5**

**test[0][2][1] = 6**

**test[1][0][0] = 7**

**test[1][0][1] = 8**

**test[1][1][0] = 9**

**test[1][1][1] = 10**

**test[1][2][0] = 11**

**test[1][2][1] = 12**

### 3.3 ARRAY PROGRAMS – 2D

- a) Basic 2d array program
- b) Store and display values
- c) Sum of two matrices using Two dimensional arrays
- d) Transpose of Matrix
- e) Multiplication of 2d matrices.

```
#include<stdio.h>
int main(){
    int disp[2][3]; /* 2D array
declaration*/
    int i, j; /*Counter variables
for loop*/
    for(i=0; i<2; i++) {
        for(j=0;j<3;j++) {
            printf("Enter value for
disp[%d][%d]:", i, j);
            scanf("%d", &disp[i][j]);
        } }
```

```
//Displaying array elements
printf("Two Dimensional array
elements:\n");
for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
        printf("%d ", disp[i][j]);
        if(j==2){
            printf("\n");
        }
    }
}
return 0; }
```

# Basic 2D Array Program

## OUTPUT

Enter value for disp[0][0]:1

Enter value for disp[0][1]:2

Enter value for disp[0][2]:3

Enter value for disp[1][0]:4

Enter value for disp[1][1]:5

Enter value for disp[1][2]:6

Two Dimensional array elements:

1 2 3

4 5 6

## STORE AND DISPLAY VALUES OF ARRAY

```
#include <stdio.h>
const int CITY = 1;
const int WEEK = 7;
int main()
{
    int temperature[CITY][WEEK];
    for (int i = 0; i < CITY; ++i) {
        for(int j = 0; j < WEEK; ++j) {
            printf("City %d, Day %d: ",
i+1, j+1);
            scanf("%d",
&temperature[i][j]);
        }
    }
    printf("\nDisplaying      values:
\n\n");
    for (int i = 0; i < CITY; ++i) {
        for(int j = 0; j < WEEK; ++j)
        {
            printf("City %d, Day %d =
%d\n",
i+1, j+1,
temperature[i][j]);
        }
    }
    return 0;
}
```

## OUTPUT

City 1, Day 1: 33  
City 1, Day 2: 34  
City 1, Day 3: 35  
City 1, Day 4: 33  
City 1, Day 5: 32  
City 1, Day 6: 31  
City 1, Day 7: 30

Displaying values:  
City 1, Day 1 = 33  
City 1, Day 2 = 34  
City 1, Day 3 = 35  
City 1, Day 4 = 33  
City 1, Day 5 = 32  
City 1, Day 6 = 31  
City 1, Day 7 = 30

## Sum of 2D Array

```
#include <stdio.h>
int main()
{
    float a[2][2], b[2][2], c[2][2];
    int i, j; // Taking input using nested
for loop
    printf("Enter      elements      of      1st
matrix\n");
    for(i=0; i<2; ++i)
        for(j=0; j<2; ++j)
        {
            printf("Enter a%d%d: ", i+1, j+1);
            scanf("%f", &a[i][j]);
        } // Taking input using nested for loop
```

```
    printf("Enter      elements      of      1st
matrix\n");
    for(i=0; i<2; ++i)
        for(j=0; j<2; ++j)
        {
            printf("Enter b%d%d: ", i+1, j+1);
            scanf("%f", &b[i][j]);
        } // adding corresponding elements of
two arrays
    for(i=0; i<2; ++i)
        for(j=0; j<2; ++j)
        {
            c[i][j] = a[i][j] + b[i][j];
        }
```

```
// Displaying the sum printf("\nSum Of Matrix:");
for(i=0; i<2;
++i)
for(j=0; j<2; ++j)

{
    printf("%.1f\t", c[i][j]);
    if(j==1)
        printf("\n");
}
return 0;
}
```

**OUTPUT:**

Enter elements of 1st matrix  
Enter a11: 2;  
Enter a12: 0.5;  
Enter a21: -1.1;  
Enter a22: 2;  
Enter elements of 2nd matrix  
Enter b11: 0.2;  
Enter b12: 0;  
Enter b21: 0.23;  
Enter b22: 23;  
Sum Of Matrix:  
**2.2 0.5**  
**-0.9 25.0**

# Transpose of Matrix

```
#include <stdio.h>
int main()
{
    int a[10][10], transpose[10][10], r, c,
i, j;
    printf("Enter rows and columns of
matrix: ");
    scanf("%d %d", &r, &c); // Storing
elements
    printf("\nEnter      elements      of
matrix:\n");
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            printf("Enter      element      a%d%d:
", i+1, j+1);
            scanf("%d", &a[i][j]);
        }
    printf("\nEnter      elements      of
matrix: \n");
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            printf("%d ", a[i][j]);
            if (j == c-1)
                printf("\n\n");
        }
    // Finding the transpose of
matrix a
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            transpose[j][i] = a[i][j];
        }
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            printf("%d ", transpose[j][i]);
            if (j == c-1)
                printf("\n");
        }
}
```

## output

```
// Displaying the transpose of matrix a
printf("\nTranspose of Matrix:\n");
for(i=0; i<c; ++i)
    for(j=0; j<r; ++j)
    {
        printf("%d ",transpose[i][j]);
        if(j==r-1)
            printf("\n\n");
    }
return 0;
}
```

Enter rows and columns of matrix: 2 3  
Enter element of matrix:  
Enter element a11: 2  
Enter element a12: 3  
Enter element a13: 4  
Enter element a21: 5  
Enter element a22: 6  
Enter element a23: 4  
Entered Matrix:  
**2 3 4**  
**5 6 4**  
Transpose of Matrix:  
**2 5**  
**3 6**  
**4 4**

# Multiplication of 2D Matrix

```
#include<stdio.h>
int main()
int m, n, p, q, c, d, k, sum = 0;
int first[10][10], second[10][10],
multiply[10][10];
printf("Enter number of rows and
columns of first matrix\n");
scanf("%d%d", &m, &n);
printf("Enter elements of
matrix\n");
for (c = 0; c < m; c++)
for (d = 0; d < n; d++)
scanf("%d", &first[c][d]);
```

first

```
printf("Enter number of rows and
columns of second matrix\n");
scanf("%d%d", &p, &q);
for (c = 0; c < p; c++)
for (d = 0; d < q; d++)
scanf("%d", &second[c][d]);
for (c = 0; c < m; c++) {
for (d = 0; d < q; d++) {
for (k = 0; k < p; k++) {
sum = sum +
first[c][k]*second[k][d];
}
multiply[c][d] = sum;
sum = 0;
} }
```

```
printf("Product      of      the matrices:\n");
```

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

```
return 0;  
}
```

### OUTPUT

Enter the number of rows and columns  
of first matrix: 3 3

Enter the elements of first matrix: 1 2 0

0 1 1

2 0 1

Enter the number of rows and  
columns of second matrix:

3

3

Enter the elements of second  
matrix:

1 1 2

2 1 1

1 2 1

Product of entered matrices:-

5 3 4

3 3 2

3 4 5

## **4. ARRAY CONTIGUOUS MEMORY**

- When Big Block of memory is reserved or allocated then that memory block is called as Contiguous Memory Block.
- Alternate meaning of Contiguous Memory is continuous memory.
- Suppose inside memory we have reserved 1000-1200 memory addresses for special purposes then we can say that these 200 blocks are going to reserve contiguous memory.

## **How to allocate contiguous memory:**

- Using static array declaration.
- Using alloc() / malloc() function to allocate big chunk of memory dynamically.

## **Contiguous Memory Allocation**

- Two registers are used while implementing the contiguous memory scheme. These registers are base register and limit register.

When OS is executing a process inside the main memory then content of each register are as –

- ❑ **Register** - Content of register.
- ❑ **Base register-** Starting address of the memory location where process execution is happening.
- ❑ **Limit register-** Total amount of memory in bytes consumed by process.

When process try to refer a part of the memory then it will firstly refer the base address from base register and then it will refer relative address of memory location with respect to base address

## **3.5 ADVANTAGE AND LIMITATIONS OF ARRAY**

### **1. Advantages:**

- It is better and convenient way of storing the data of same datatype with same size.
- It allows us to store known number of elements in it.
- It allocates memory in contiguous memory locations for its elements. It does not allocate any extra space/ memory for its elements. Hence there is no memory overflow or shortage of memory in arrays.
- Iterating the arrays using their index is faster compared to any other methods like linked list etc.
- It allows to store the elements in any dimensional array - supports multidimensional array.

## **3.5.2 Limitations of Array:**

### **A) *Static Data***

- Array is Static data Structure
- Memory Allocated during Compile time.
- Once Memory is allocated at Compile Time it Cannot be Changed during Run-time.

### **B) *Can hold data belonging to same Data types***

- Elements belonging to different data types cannot be stored in array because array data structure can hold data belonging to same data type.
- Example : Character and Integer values can be stored inside separate array but cannot be stored in single array

### **C) Inserting data in Array is Difficult**

- Inserting element is very difficult because before inserting element in an array have to create empty space by shifting other elements one position ahead.
- This operation is faster if the array size is smaller, but same operation will be more and more time consuming and non-efficient in case of array with large size.

### **D) Deletion Operation is difficult**

- Deletion is not easy because the elements are stored in contiguous memory location.
- Like insertion operation , we have to delete element from the array and after deletion empty space will be created and thus we need to fill the space by moving elements up in the array.

## **E) Bound Checking**

- If we specify the size of array as 'N' then we can access elements upto 'N-1' but in C if we try to access elements after 'N-1' i.e Nth element or N+1th element then we does not get any error message.
- Process of Checking the extreme limit of array is called *Bound checking* and C does not perform Bound Checking.
- If the array range exceeds then we will get garbage value as result.

## **E) Shortage of Memory**

- Array is Static data structure. Memory can be allocated at *compile time* only Thus if after executing program we need more space for storing additional information then we cannot allocate additional space at run time.
- Shortage of Memory , if we don't know the size of memory in advance

## **F) Wastage of Memory**

- Wastage of Memory , if array of large size is defined.

## ***3.6 ARRAY CONSTRUCTION FOR REAL-TIME APPLICATION COMMON PROGRAMMING ERRORS***

### **(i) Constant Expression Require**

```
#include<stdio.h>
void main()
{
int i=10;
int a[i];
}
```

In this example we see what's that error?

- We are going to declare an array whose size is equal to the value of variable.
- If we changed the value of variable then array size is going to change.
- According to array concept, we are allocating memory for array at compile time so if the size of array is going to vary then how it is possible to allocate memory to an array.
- *i is initialized to 10 and using a[i] does not mean a[10] because 'i' is Integer Variable whose value can be changed inside program.*

- **Value of Const Variable Cannot be changed**
- we know that value of Const Variable cannot be changed once initialized so we can write above example as below –

```
#include<stdio.h>
void main()
{
    const int i=10;
    int a[i];
}
```

or

**int a[10];**

## (ii) Empty Valued 1D Array

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

```
{  
int arr[];  
}
```

Instead of it Write it as –

```
#include<stdio.h> void main()  
{  
int a[] = {1,1};  
}
```

- Consider this example, we can see the empty pair of square brackets means we haven't specified size of an 1D Array. In this example array 'arr' is undefined or empty.
- Size of 1D Array should be Specified as a Constant Value.

```
#include<stdio.h>  
void main()  
{  
int a[] = {};// This also  
Cause an Error  
}
```

### (iii) 1D Array with no Bound Checking

```
#include<stdio.h>
void main()
{
int a[5];
printf("%d",a[7]);
}
```

Here Array size specified is 5.

- So we have Access to Following Array Elements –  
a[0],a[1],a[2],a[3] and a[4].
- But accessing a[5] causes Garbage Value to be used because C Does not performs Array Bound Check.

If the maximum size of array is “MAX” then we can access following elements of an array –

Elements accessible for Array Size "MAX" = arr[0]  
=.  
= arr[MAX-1]

#### **4. Case Sensitive**

```
#include<stdio.h>
void main()
{
int a[5];
printf("%d",A[2]);
}
```

Array Variable is Case Sensitive so A[2] does not print anything it Displays Error Message : “Undefined Symbol A”

## 3.7 STRING BASICS

- **Strings in C are represented by arrays of characters.**
- String is nothing but the collection of the individual array elements or characters stored at contiguous memory locations
  - i) **Character array** – 'P','P','S'
  - ii) **Double quotes** - “PPS” is a example of String.
    - If string contains the double quote as part of string then we can use escape character to keep double quote as a part of string.
    - “PP\S” is a example of String

### *iii) Null Character*

- The end of the string is marked with a special character, the *null character*, which is a character all of whose bits are zero i.e., a NULL.
- String always Terminated with NULL Character ('/0')

***char name[10] = {'P','P','S','\0'}***

- NULL Character is having ASCII value 0
- ASCII Value of '\0' = 0
- As String is nothing but an array , so it is Possible to Access Individual Character

***name[10] = "PPS";***

- It is possible to access individual character

***name[0] = 'P';***

***name[1] = 'P';***

***name[2] = 'S';***

***name[3] = '\0';***

#### **iv) MEMORY**

Each Character Occupy 1 byte of Memory

$$\begin{aligned}\text{Size of "PPS"} &= \text{Size of 'P'} + \\ &= \text{Size of 'P'} + \\ &= \text{Size of 'S'};\end{aligned}$$

Size of "PPS" is 3 BYTES

Each Character is stored in consecutive memory location.

Address of 'P' = 2000

Address of 'P' = 2001

Address of 'S' = 2002

## **8. STRING DECLARATION AND INITIALIZATION**

- **3.8.1 String Declaration:**

- String data type is not supported in C Programming. String means Collection of Characters to form particular word. String is useful whenever we accept name of the person, Address of the person, some descriptive information. We cannot declare string using String Data Type, instead of we use array of type character to create String.
- Character Array is Called as ‘String’.
- Character Array is Declared Before Using it in Program.

***char String\_Variable\_name [ SIZE ];***

***Eg: char city[30];***

Point	Explanation
Significance	- We have declared array of character[i.e String]
Size of string	- 30 Bytes
Bound checking	- C Does not Support Bound Checking i.e if we store City with size greater than 30 then C will not give you any error
Data type	- char
Maximum size	- 30

## ***Precautions to be taken while declaring Character Variable :***

- String / Character Array Variable name should be legal C Identifier.
- String Variable must have Size specified.
  - **char city[];**
- Above Statement will cause compile time error.
  - Do not use String as data type because String data type is included in later languages such as C++ / Java. C does not support String data type
  - **String city;**
- When you are using string for other purpose than accepting and printing data then you must include following header file in your code –
  - **#include<string.h>**

## **2. *Initializing String [Character Array] :***

- Whenever we declare a String then it will contain garbage values inside it. We have to initialize String or Character array before using it. Process of Assigning some legal default data to String is Called Initialization of String. There are different ways of initializing String in C Programming –
  - 1) Initializing Unsized Array of Character
  - 2) Initializing String Directly
  - 3) Initializing String Using Character Pointer

## ***Way 1 : Unsized Array and Character***

- ***Unsized Array*** : Array Length is not specified while initializing character array using this approach
- Array length is Automatically calculated by Compiler
- Individual Characters are written inside Single Quotes , Separated by comma to form a list of characters. Complete list is wrapped inside Pair of Curly braces
- ***NULL Character*** should be written in the list because it is ending or terminating character in the String/Character Array
- `char name [] = {'P','P','S','\0'};`

## ***Way 2 : Directly initialize String Variable***

- In this method we are directly assigning String to variable by writing text in double quotes.
- In this type of initialization , we don't need to put NULL or Ending / Terminating character at the end of string. It is appended automatically by the compiler.
- char name [ ] = "PPS";

### ***Way 3 : Character Pointer Variable***

- Declare Character variable of pointer type so that it can hold the base address of “String”
- Base address means address of first array element i.e (address of name[0] )
- NULL Character is appended Automatically
- `char *name = "PPS";`

### **3.9 STRING FUNCTIONS**

- gets()
- puts()
- Getchar()
- Putchar()
- Printf()
- Atoi()
- Strlen()
- Strcat ()
- Strcmp()
- Sprintf()
- Sscanf()
- Strrev()
- Strcpy()
- Strstr()
- strtok()

### **3.9.1 GETS():**

Syntax for Accepting String :

char \* gets ( char \* str ); OR  
gets( <variable-name> )

**Example :**

```
#include<stdio.h>
void main()
{
    char name[20];
    printf("\nEnter the Name : ");
    gets(name); }
```

**Output:**

Enter the name: programming in c

## ***Explanation :***

- Whenever gets() statement encounters then characters entered by user (the string with spaces) will be copied into the variable.
- If user start accepting characters , and if new line character appears then the newline character will not be copied into the string variable(i.e name).
- A terminating null character is automatically appended after the characters copied to string vriable (i.e name)
- gets() uses stdin (Standered Input Output) as source, but it does not include the ending newline character in the resulting string and does not allow to specify a maximum size for string variable (which can lead to buffer overflows).

## ***Some Rules and Facts :***

### **A. %s is not Required :**

Like scanf statement %s is not necessary while accepting string.

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

and here is gets() syntax which is simpler than scanf() –

```
gets(name);
```

### **B. Spaces are allowed in gets() :**

```
gets(name);
```

Whenever the above line encounters then interrupt will wait for user to enter some text on the screen. When user starts typing the characters then all characters will be copied to string and when user enters newline character then process of accepting string will be stopped.

### **Sample Input Accepted by Above Statement:**

- Value Accepted : Problem solving\n
- Value Stored : Problem solving (\n Neglected)

## **2. *PUTS():***

### **Way 1 :Messaging**

- `puts(" Type your Message / Instruction ");`
- Like Printf Statement `puts()` can be used to display message.

### **Way 2 : Display String**

- `puts(string_Variable_name) ;`

### **Notes or Facts :**

- `puts` is included in header file “`stdio.h`”
- As name suggest it used for Printing or Displaying Messages or Instructions.

**Example :**

```
#include< stdio.h>
#include< conio.h>
void main()
{
    char string[] = "This is an example string\n";
    puts(string);    // String is variable Here
    puts("String");  // String is in Double Quotes
    getch();
}
```

**Output :**

*String is : This is an example string*

*String is : String*

### 3.9.3 GETCHAR():

- **Getchar()** function is also one of the function which is used to accept the single character from the user.
- The characters accepted by getchar() are buffered until **RETURN** is hit means getchar() does not see the characters until the user presses return. (i.e Enter Key)

#### Syntax for Accepting String and Working :

```
/* getchar accepts character & stores in ch */  
    char ch = getchar();
```

- When control is on above line then getchar() function will accept the single character. After accepting character control remains on the same line. When user presses the enter key then getchar() function will read the character and that character is assigned to the variable 'ch'.

Parameter	Explanation
Header File	- stdio.h
Return Type	- int (ASCII Value of the character)
Parameter	- Void
Use	- Accepting the Character

## **Example 1 :**

In the following example we are just accepting the single character and printing it on the console –

```
main()
{
    char ch;
    ch = getchar();
    printf("Accepted Character : %c",ch);
}
```

**Output :**

Accepted Character : A

## Example 2 : Accepting String (Use One of the Loop)

```
#include<stdio.h>
void main()
{
int i = 0;
char name[20];
printf("\nEnter the Name : ");
while((name[i] = getchar())!='\n')
    i++; /*while loop will accept the one character at a time and check
it with newline character. Whenever user enters newline character
then control comes out of the loop.*/
getch(); }
```

### 3.9.4 PUTCHAR():

Displaying String in C Programming

Syntax :

int putchar(int c);

Way 1 : Taking Character as Parameter

`putchar('a') ; // Displays : a`

- Individual Character is Given as parameter to this function.
- We have to explicitly mention Character.

## Way 2 : Taking Variable as Parameter

putchar(a);

// Display Character Stored in a

- ❑ Input Parameter is Variable of Type “Character”.
- ❑ This type of putchar() displays character stored in variable.

## Way3:Displaying Particular Character from Array

putchar(a[0]) ;

// Display a[0] th element from array

- ❑ Character Array or String consists of collection of characters.
- ❑ Like accessing individual array element , characters can be displayed one by one using putchar().

**Example:**

```
#include< stdio.h>
#include< conio.h>
int main()
{
    char string[] = "C programming\n";
    int i=0;
    while(string[i]!='\0')
    {
        putchar(string[i]);
        i++;
    }
    return 0;
}
```

**Output:**

C programming

### **3.9.5 *PRINTF()*:**

Syntax :

Way 1 : Messaging

```
printf (" Type your Message / Instruction " );
```

Way 2 : Display String

```
printf ("Name of Person is %s ", name ) ;
```

***Notes or Facts :***

printf is included in header file “**stdio.h**”

As name suggest it used for **Printing or Displaying Messages or Instructions**

**Uses :**

- Printing Message
- Ask user for entering the data ( Labels . Instructions )
- Printing Results

### **3.10.1 ATOI FUNCTION:**

- Atoi = A to I = Alphabet to Integer
- **Convert String of number into Integer**

**Syntax:**

**num = atoi(String);**

num - Integer Variable

String- String of Numbers

**Example :**

```
char a[10] = "100";
int value = atoi(a);
printf("Value = %d\n", value);
return 0;
```

**Output :**

**Value : 100**

## Significance :

- Can Convert any String of Number into Integer Value that can Perform the arithmetic Operations like integer
- Header File : **stdlib.h**

## Ways of Using Atoi Function :

### Way 1 : Passing Variable in Atoi Function

```
int num;  
char marks[3] = "98";  
num = atoi(marks);  
printf("\nMarks : %d",num);
```

### Way 2 : Passing Direct String in Atoi Function

```
int num;  
num = atoi("98");  
printf("\nMarks : %d",num);
```

## ***OTHER INBUILT TYPECAST FUNCTIONS IN C PROGRAMMING LANGUAGE:***

- Typecasting functions in C language performs data type conversion from one type to another.
- Click on each function name below for description and example programs.
  - [atof\(\)](#) Converts string to float
  - [atoi\(\)](#) Converts string to int
  - [atol\(\)](#) Converts string to long
  - [itoa\(\)](#) Converts int to string
  - [ltoa\(\)](#) Converts long to string

### **3.10.2 STRLEN FUNCTION:**

- **Finding length of string**

<b>Point</b>		<b>Explanation</b>
No of Parameters	-	1
Parameter Taken	-	Character Array Or String
Return Type	-	Integer
Description	-	Compute the Length of the String
Header file	-	string.h

## *Different Ways of Using strlen() :*

- There are different ways of using strlen function. We can pass different parameters to strlen() function.

### *Way 1 : Taking String Variable as Parameter*

```
char str[20];
int length ;
printf("\nEnter the String : ");
gets(str);
length = strlen(str);
printf("\nLength of String : %d ", length);
```

#### **Output:**

Enter the String : hello  
Length of String : 5

## Way 2 : Taking String Variable which is Already Initialized using Pointer

```
char *str = "priteshtaral";
int length ;
length = strlen(str);
printf("\nLength of String : %d ", length);
```

## Way 3 : Taking Direct String

```
int length ;
length = strlen("pritesh");
printf("\nLength of String : %d",length);
```

## Way 4 : Writing Function in printf Statement

```
char *str = "pritesh";
printf("\nLength of String : %d", strlen(str));
```

### **3. STRCAT FUNCTION:**

What strcat Actually does ?

- Function takes 2 Strings / Character Array as Parameter
- Appends second string at the end of First String.

Parameter Taken - 2 Character Arrays / Strings Return

Type - Character Array / String

**Syntax :**

**char\* strlen ( char \* s1, char \* s2);**

## ***Ways of Using Strcat Function :***

### **Way 1 : Taking String Variable as Parameter**

```
char str1[20] = “Don” , str2[20] = “Bosqo”;  
strcat(str1,str2);  
puts(str1);
```

### **Way 2 : Taking String Variable which is Already Initialized using Pointer**

```
char *str1 = “Ind”,*str2 = “ia”;  
strcat(str1,str2);// Result stored in str1        puts(str1); // Result :  
India
```

### **Way 3 : Writing Function in printf Statement**

```
printf(“nString: ”, strcat(“Ind”, “ia”));
```

### **3.10.4 STRCMP FUNCTION:**

What strcmp Actually Does ?

- Function takes two Strings as parameter.
- It returns integer .

Syntax :

`int strcmp ( char *s1, char *s2 );`

#### **Return Type**

-ve Value

-

#### **Condition**

String1 < String2

+ve Value

-

String1 > String2

0 Value

-

String1 = String2

## Example 1 : Two strings are Equal

```
char s1[10] = "SAM",s2[10]="SAM" ;  
int len;  
len = strcmp (s1,s2);
```

Output

0

*/\*So the output will be 0. if u want to print the string then give condition like\*/*

```
char s1[10] = "SAM",s2[10]="SAM" ;  
int len;  
len = strcmp (s1,s2);  
if (len == 0)  
printf ("Two Strings are Equal");
```

Output:

Two Strings are Equal

## Example 2 : String1 is Greater than String2

```
char s1[10] = "SAM",s2[10]="sam";  
int len;  
len = strcmp (s1,s2);  
printf ("%d",len); // -ve value
```

**Output:**

-32

**Reason :**

ASCII value of “SAM” is smaller than “sam”

ASCII value of ‘S’ is smaller than ‘s’

### Example 3 : String1 is Smaller than String2

```
char s1[10] = "sam",s2[10]="SAM" ;
int len;
len = strcmp (s1,s2);
printf ("%d",len); //+ve value
```

**Output:**

**85**

**Reason :**

ASCII value of “SAM” is greater than “sam”

ASCII value of ‘S’ is greater than ‘s’

## **1. *SPRINTF FUNCTION:***

- **sends formatted output to String.**

### **Features :**

- Output is Written into String instead of Displaying it on the Output Devices.
- Return value is integer ( i.e Number of characters actually placed in array / length of string ).
- String is terminated by '\0'.
- Main Per pose : Sending Formatted output to String.
- Header File : **Stdio.h**

### **Syntax :**

**int sprintf(char \*buf,char format,arg\_list);**

**Example :**

```
int age = 23 ;  
char str[100];  
sprintf( str , "My age is %d",age);  
puts(str);
```

**Output:**

My age is 23

**Analysis of Source Code:** Just keep in mind that

- Assume that we are using printf then we get output “My age is 23”
- What does printf does ? — Just Print the Result on the Screen
- Similarly Sprintf stores result “My age is 23” into string str instead of printing it.

### **3.11.2 *SSCANF FUNCTION*:**

**Syntax :**

**int sscanf(const char \*buffer, const char \*format[, address, ...]);**

What it actually does ?

- Data is read from array Pointed to by buffer rather than stdin.
- Return Type is Integer
- Return value is nothing but number of fields that were actually assigned a value

## Example

```
#include <stdio.h>
int main ()
{
    char buffer[30] = "Fresh2refresh 5 ";
    char name [20];
    int age;
    sscanf (buffer,"%s %d",name,&age);
    printf ("Name : %s \n Age : %d \n",name,age);
    return 0;
}
```

## Output:

Name : Fresh2refresh Age : 5

### **3. STRSTR FUNCTION:**

- Finds first occurrence of sub-string in other string

Features :

- Finds the first occurrence of a sub string in another string
- Main Purpose : **Finding Substring**
- Header File : **String.h**
- Checks whether s2 is present in s1 or not
- On success, strstr returns a pointer to the element in s1 where s2 begins (points to s2 in s1).
- On error (if s2 does not occur in s1), strstr returns null.

### Syntax :

**char \*strstr(const char \*s1, const char \*s2);**

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char string[55] ="This is a test string;
    char *p;
    p = strstr (string,"test");
    if(p)
    {
        printf("string found\n");
    }
}
```

```
printf("First string \"test\" in \"%s\" to \"\" \"%s \"",string,p);
}
else
    printf("string not found\n");
return 0;
}
```

**Output:**

string found

First string “test” in “This is a test string” to “test string”.

## Example 2

### Parameters as String initialized using Pointers

```
#include<stdio.h>
#include<string.h>
int main(void)
{
    char *str1 = "c4learn.blogspot.com", *str2 = "spot", *ptr;
    ptr = strstr(str1, str2);
    printf("The substring is: %sn", ptr);
    return 0;
}
```

Output :

The substring is: spot.com

## Example 3

### Passing Direct Strings

```
#include<stdio.h>
#include<string.h>
void main()
{
    char *ptr;
    ptr = strstr("c4learn.blogspot.com","spot");
    printf("The substring is: %sn", ptr);
}
```

Output :

The substring is: spot.com

### **3.11.4 STRREV():**

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

**char \*strrev(char \*string);**

- strrev() function is nonstandard function which may not available in standard library in C.

### **Algorithm to Reverse String in C :**

- Start
- Take 2 Subscript Variables ‘i’,‘j’
- ‘j’ is Positioned on Last Character
- ‘i’ is positioned on first character
- str[i] is interchanged with str[j]
- Increment ‘i’
- Decrement ‘j’
- If ‘i’ > ‘j’ then goto step 3
- Stop

## Example

```
#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;
}
```

## Output:

String before strrev() : Hello  
String after strrev() : olleH

### **3.11.5 STRCPY FUNCTION:**

**Copy second string into First**

What strcmp Actually Does ?

- Function takes two Strings as parameter.
- Header File : String.h.
- It returns string.
- Purpose : Copies String2 into String1.
- Original contents of String1 will be lost.
- Original contents of String2 will remains as it is.

**Syntax :**

**char \* strcpy ( char \*string1, char \*string2 ) ;**

- `strcpy ( str1, str2)` – It copies contents of str2 into str1.
- `strcpy ( str2, str1)` – It copies contents of str1 into str2.
- If destination string length is **less than source string**, **entire source string value won't be copied into destination string**.
- For example, consider destination string length is 20 and source string length is 30. Then, only 20 characters from source string will be copied into destination string and remaining 10 characters won't be copied and will be truncated.

## Example 1

```
char s1[10] = "SAM" ;  
char s2[10] = "MIKE" ;  
strcpy (s1,s2);  
puts (s1) ; // Prints : MIKE  
puts (s2) ; // Prints : MIKE
```

### Output:

MIKE

MIKE

## Example 2

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

### Output

source string = hihello

target string =

target string after strcpy( ) = hihello

### **3.11.6 STRTOK FUNCTION**

- tokenizes/parses the given string using delimiter.

#### **Syntax**

**char \* strtok ( char \* str, const char \* delimiters );**

*For example, we have a comma separated list of items from a file and we want individual items in an array.*

- *Splits str[] according to given delimiters and returns next token.*
- *It needs to be called in a loop to get all tokens.*
- *It returns NULL when there are no more tokens.*

## Example

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str[] = "Problem_Solving_in_c";//Returns first token
    char* token = strtok(str, "_");//Keep printing tokens while
one of the delimiters present in str[].

    while (token != NULL) {
        printf("%s\n", token);
```

```
token = strtok(NULL, "_");  
    }
```

```
return 0;  
}
```

## Output:

Problem  
Solving  
in  
C

### **3.12 ARITHMETIC CHARACTERS ON STRING**

- C Programming Allows you to Manipulate on String
- Whenever the Character is variable is used in the expression then it is automatically Converted into Integer Value called ASCII value.
- All Characters can be Manipulated with that Integer Value.(Addition,Subtraction)

**Examples :**

**ASCII value of : 'a' is 97**

**ASCII value of : 'z' is 121**

## Possible Ways of Manipulation :

Way 1: Displays ASCII value[ Note that %d in Printf]

```
char x = 'a';
```

```
printf("%d",x); // Display Result = 97
```

Way 2 : Displays Character value[Note that %c in Printf]

```
char x = 'a';
```

```
printf("%c",x); // Display Result = a
```

Way 3 : Displays Next ASCII value[ Note that %d in Printf ]

```
char x = 'a' + 1 ;
```

```
printf("%d",x); //Display Result = 98 (ascii of 'b' )
```

Way 4 Displays Next Character value [Note that %c in Printf ]

```
char x = 'a' + 1;  
printf("%c",x); // Display Result = 'b'
```

Way 5 : Displays Difference between 2 ASCII in Integer [Note %d in Printf ]

```
char x = 'z' - 'a';  
printf("%d",x);/*Display Result = 25 (difference between ASCII of z and a ) */
```

Way 6 : Displays Difference between 2 ASCII in Char [Note that %c in Printf ]

```
char x = 'z' - 'a';  
printf("%c",x);/*Display Result =( difference between ASCII of z and a ) */
```

### 3.13 FUNCTION DECLARATION AND DEFINITION:

- ❑ A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.
- ❑ 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.

### 3.13.1 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

```
main()
{
    display();
}
```

We have written functions in the above specified sequence, however functions are called in which order we call them.

```
printf("In mumbai");
}
```

Here functions are called this sequence –

```
main() display() pune() india()
mumbai() india();
}
```

```
void display()
{
    pune();
}
```

```
void india()
{
    mumbai();
}
```

## Why Function is used???

### Advantages of Writing Function in C Programming

#### 1. Modular and Structural Programming can be done

- We can divide c program in **smaller modules.**
- We can call module whenever require. e.g suppose we have written calculator program then we can write 4 modules (i.e add,sub,multiply,divide)
- Modular programming **makes C program more readable.**
- Modules once created , **can be re-used in other programs.**

#### 2. It follows Top-Down Execution approach , So main can be kept very small.

- Every C program starts **from main function.**
- Every function is **called directly or indirectly through main**
- Example : **Top down approach.** (functions are executed from top to bottom)

### **3. Individual functions can be easily built, tested**

- As we have developed C application in modules **we can test each and every module.**
- Unit testing** is possible.
- Writing code in function will **enhance application development process.**

### **3. Program development become easy**

### **4. Frequently used functions can be put together in the customized library**

- We can put frequently used functions in **our custom header file.**
- After creating header file we can re use header file. We can include header file in other program.

### **5. A function can call other functions & also itself**

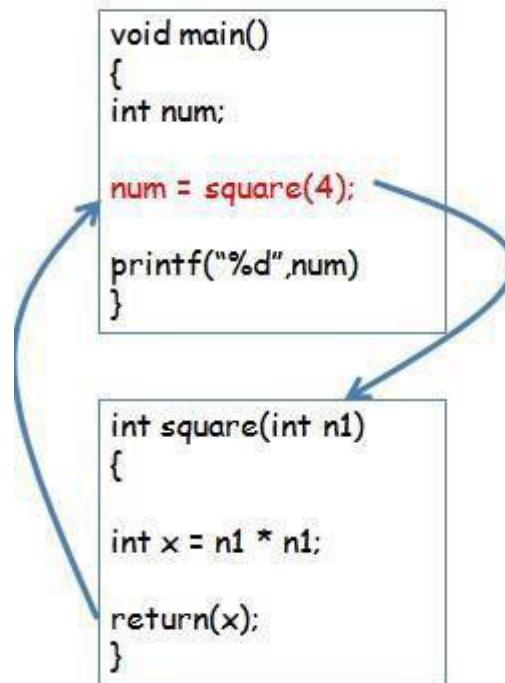
- Function can call other function.
- Function can call itself , which is called as “recursive” function.
- Recursive functions are also useful in order to write system functions.

### **6. It is easier to understand the Program topic**

- We can get overall idea of the project just by reviewing function names.

## How Function works in C Programming?

- ❑ C programming is modular programming language.
- ❑ We must divide C program in the different modules in order to create more readable, eye catching ,effective, optimized code.
- ❑ In this article we are going to see how function is C programming works ?



## Explanation : How function works in C Programming ?

- Firstly Operating System will call our main function.
- When control comes inside main function , execution of main starts (i.e execution of C program starts)
- Consider Line 4 :  
`num = square(4);`
- We have called a function square(4). [ See : How to call a function ? ].
- We have passed “4” as parameter to function.

*Note : Calling a function halts execution of the current function , it will execute called function*

- after execution control returned back to the calling function.
- Function will return 16 to the calling function.(i.e. main)
- Returned value will be copied into variable.
- printf will gets executed.
- main function ends.
- C program terminates.

## **FUNCTION PROTOTYPE DECLARATION IN C PROGRAMMING**

- ❑ Function prototype declaration is necessary in order to provide information the compiler about function, about return type, parameter list and function name etc.

### **Important Points :**

- ❑ Our program starts from main function. Each and every function is called directly or indirectly through main function
- ❑ Like variable we also need to declare function before using it in program.
- ❑ In C, declaration of function is called as prototype declaration
- ❑ Function declaration is also called as function prototype

### **Points to remember**

- ❑ Below are some of the important notable things related to prototype declaration –
- ❑ It tells name of function, return type of function and argument list related information to the compiler
- ❑ Prototype declaration always ends with semicolon.
- ❑ Parameter list is optional.
- ❑ Default return type is integer.

## Header Files

Global Declaration Section  
(Write Prototype Declaration)



Write Main Function in this  
section

[www.c4learn.com](http://www.c4learn.com)

Write All Function Definitions  
in this Section

## Syntax

`return_type function_name ( type arg1, type arg2..... );`

prototype declaration comprised of three parts i.e name of the function, return type and parameter list

### Examples of prototype declaration

- ❑ Function with two integer arguments and integer as return type is represented using syntax **int sum(int,int);**
- ❑ Function with integer argument and integer as return type is represented using syntax **int square(int);**
- ❑ In the below example we have written function with no argument and no return type **void display(void);**
- ❑ In below example we have delclared function with no argument and integer as return type **int getValue(void);**

## **Positioning function declaration**

- If function definition is written after main then and then only we write prototype declaration in global declaration section
- If function definition is written above the main function then ,no need to write prototype declaration

### **Case 1 : Function definition written before main**

```
#include<stdio.h>
void displayMessage()
{
    printf("welcome");
}
void main()
{
    displayMessage();
}
```

## **Case 2 : Function definition written after main**

```
#include<stdio.h> //Prototype Declaration
void displayMessage();
void main()
{
    displayMessage();
}

void displayMessage()
{
    printf("welcome");
}
```

### **Need of prototype declaration**

- Program Execution always starts from main , but during lexical analysis (1st Phase of Compiler) token generation starts from left to right and from top to bottom.
- During code generation phase of compiler it may face issue of backward reference.

## 3.14 TYPES OF CALLING

- ❑ While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.
- ❑ When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.
- ❑ To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example –

```
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);
int main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret; /* calling a function to get max value */
    ret = max(a, b);
    printf( "Max value is : %d\n", ret );
    return 0;
} /* 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;
}
```

output  
Max value is : 200

*We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result –*

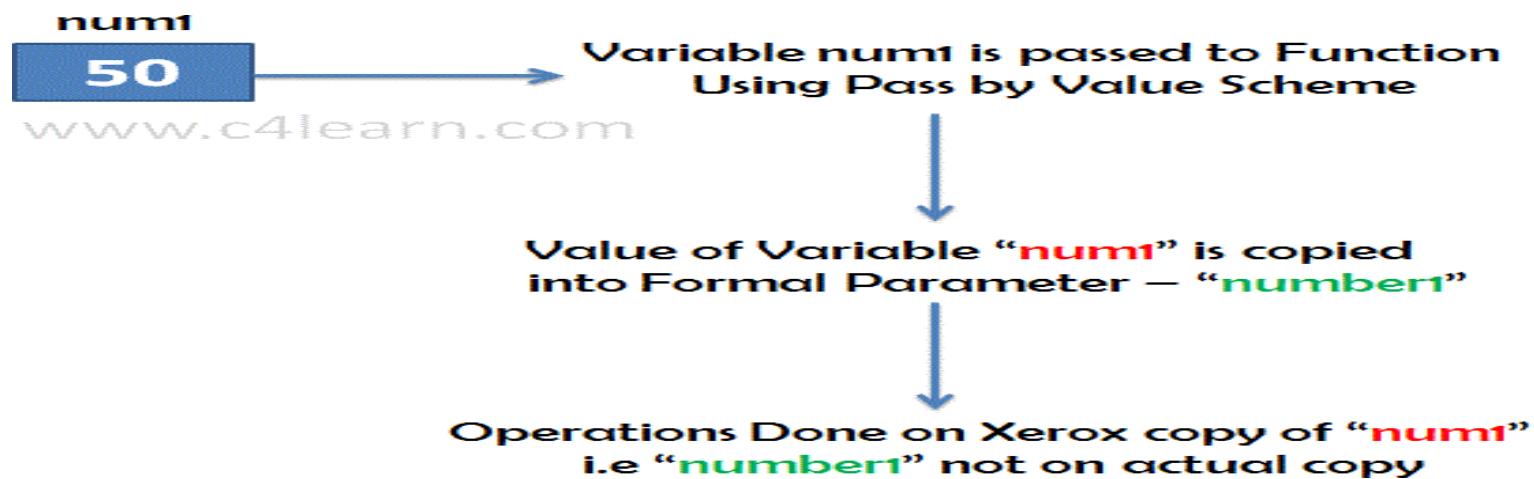
*Max value is : 200*

- ❑ If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.
- ❑ Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- ❑ While calling a function, there are two ways in which arguments can be passed to a function –
- ❑ By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Sr.No.	Call Type & Description
1	<p><b>Call by value</b> This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.</p> <pre data-bbox="654 472 1326 822">/* function definition to swap the values */ void swap (int x, int y) {     int temp; temp = x;     /* save the value of x */     x = y; /* put y into x */     y = temp; /* put temp into y */     return; }</pre>
2	<p><b>Call by reference</b> 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.</p> <pre data-bbox="654 1038 1326 1388">/* function definition to swap the values */ void swap (int x, int y) {     int temp;     temp = *x;     *x = *y;     *y = temp;     return; }</pre>

## 1. CALL BY VALUE

- ❑ 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 function respectively.
- ❑ As their scope is limited to only function so they cannot alter the values inside main function.



```
#include <stdio.h>
void swap(int x, int y); /* function declaration */
int main ()
{
    int a = 100; /* local variable definition */
    int b = 200;
    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );/*calling a function to swap the values */
    swap(a, b);
    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );
    return 0;
}
```

**Output:**

Before swap, value of a :100  
Before swap, value of b :200  
After swap, value of a :100  
After swap, value of b :200

## 2. CALL BY REFERENCE

- ❑ The **call by reference** method of passing arguments to a function 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. It means the changes made to the parameter affect the passed argument.
- ❑ To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

```
#include <stdio.h>
void swap(int *x, int *y); /* function declaration */
int main ()
{
    int a = 100; /* local variable definition */
    int b = 200;
    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b ); /*calling a function to swap the values. * &a indicates pointer to a ie. address of variable a and * &b indicates pointer to b ie. address of variable b.*/
    swap(&a, &b);
    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );
    return 0;
}
```

### **output**

**Before swap, value of a :100**

**Before swap, value of b :200**

**After swap, value of a :200**

**After swap, value of b :100**

### **3.15 .1 FUNCTION WITH ARGUMENTS AND NO RETURN VALUE:**

- ❑ Function accepts argument but it does not return a value back to the calling Program .
- ❑ It is Single ( One-way) Type Communication
- ❑ Generally Output is printed in the Called function

**Function declaration : void function();**

**Function call : function();**

**Function definition : void function() { statements; }**

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);
int main()
{
    int n; printf("Enter a positive integer: ");
    scanf("%d",&n);
    // n is passed to the function checkPrimeAndDisplay(n);
    return 0;
}
```

```
// void indicates that no value is returned from the function
void checkPrimeAndDisplay(int n)
{
int i, flag = 0;
for(i=2; i <= n/2; ++i)
{
if(n%i == 0){
flag = 1;
break; }
}
if(flag == 1)
printf("%d is not a prime number.",n);
else
printf("%d is a prime number.", n);
}
```

## OUTPUT

Enter a positive integer : 4  
4 is not a prime number

### **3.15.2 FUNCTION WITH NO ARGUMENTS AND NO RETURN VALUE IN C**

When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return a value, the calling function does not receive any data from the called function.

Syntax :

**Function declaration** : void function();

**Function call** : function();

**Function definition** : void function()

{

statements;

}

```
#include<stdio.h>
void area(); // Prototype Declaration
void main()
{
    area();
}
```

```
void area()
{
    float area_circle;
    float rad;
    printf("\nEnter the radius : ");
    scanf("%f",&rad);
    area_circle = 3.14 * rad * rad ;
    printf("Area of Circle = %f",area_circle);
}
```

Output :

Enter the radius : 3

Area of Circle = 28.260000

### **3.16.1 FUNCTION WITHOUT ARGUMENTS AND RETURN VALUE**

There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. A example for this is getchar function it has no parameters but it returns an integer an integer type data that represents a character.

**Function declaration :** int function();

**Function call :** function();

**Function definition :** int function() { statements; return x; }

```
#include<stdio.h>
int sum();
int main()
{
    int addition;
    addition = sum();
    printf("\nSum of two given values = %d", addition);
```

```
return 0;  
}  
  
int sum()  
{  
    int a = 50, b = 80, sum;  
    sum = a + b;  
    return sum;  
}
```

## OUTPUT

Sum of two given values = 130

### **3.16.2 FUNCTION WITH ARGUMENTS AND RETURN VALUE**

**Syntax :**

**Function declaration :** int function ( int );

**Function call :** function( x );

**Function definition:** int function( int x ) { statements; return x; }

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

```
float calculate_area(int);
```

```
int main()
```

```
{
```

```
int radius;
```

```
float area;
```

```
printf("\nEnter the radius of the circle : ");
```

```
scanf("%d",&radius);
```

```
area = calculate_area(radius);
```

```
printf("\nArea of Circle : %f ",area);
```

```
return(0);  
}  
float calculate_area(int radius)  
{  
    float areaOfCircle;  
    areaOfCircle = 3.14 * radius * radius;  
    return(areaOfCircle);  
}
```

**Output:**

Enter the radius of the circle : 2

Area of Circle : 12.56

## 3.17 PASSING ARRAY TO FUNCTION IN C

### Array Definition :

Array is collection of elements of similar data types .

### Passing array to function :

Array can be passed to function by two ways :

- Pass Entire array
- Pass Array element by element

#### 1 . Pass Entire array

- Here entire array can be passed as a argument to function .
- Function gets **complete access** to the original array .
- While passing entire array Address of first element is passed to function , any changes made inside function , directly **affects the Original value** .
- Function Passing method : “**Pass by Address**”

## 2 . Pass Array element by element:

- ❑ Here individual elements are passed to function as argument.
- ❑ Duplicate **carbon copy of Original variable** is passed to function .
- ❑ So any changes made inside function **does not affects the original value.**
- ❑ Function doesn't get complete access to the original array element.
- ❑ Function passing method is "**Pass by Value**".

## Passing entire array to function :

- ❑ Parameter Passing Scheme : **Pass by Reference**
- ❑ Pass **name of array** as function parameter .
- ❑ Name contains the base address i.e ( Address of 0th element )
- ❑ Array values are updated in function .
- ❑ Values are reflected inside main function also.

```
#include<stdio.h>
#include<conio.h>
void fun(int arr[])
{
    int i;
    for(i=0;i< 5;i++)
        arr[i] = arr[i] + 10;
}
void main()
{
    int arr[5],i;
    clrscr();
    printf("\nEnter the array elements : ");
}
```

```
for(i=0;i< 5;i++)
scanf("%d",&arr[i]);
printf("\nPassing entire array .....");
fun(arr); // Pass only name of array
for(i=0;i< 5;i++)
printf("\nAfter Function call a[%d] : %d",i,arr[i]);
getch();
}
```

### output

Enter the array elements : 1 2 3 4 5

Passing entire array .....

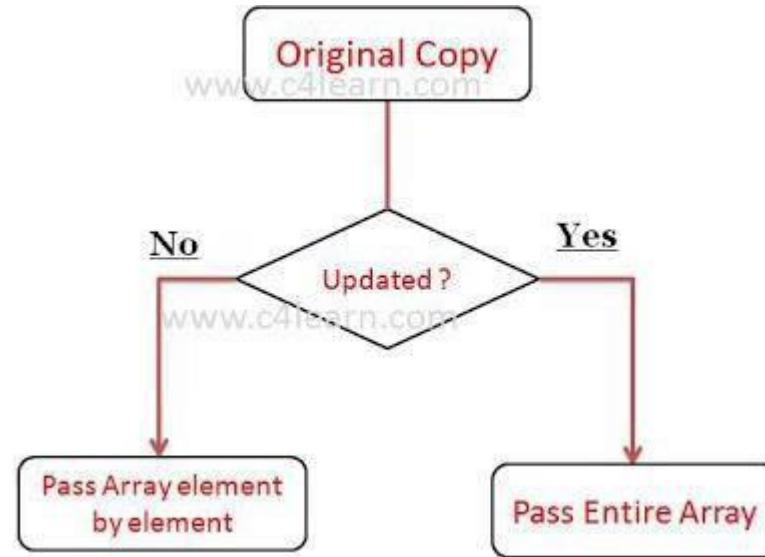
After Function call a[0] : 11

After Function call a[1] : 12

After Function call a[2] : 13

After Function call a[3] : 14

After Function call a[4] : 15



## Passing Entire 1-D Array to Function in C Programming

- Array is passed to function Completely.
- Parameter Passing Method : **Pass by Reference**
- It is Also Called “**Pass by Address**”
- Original Copy is Passed to Function
- Function Body Can Modify **Original Value.**

Example :

```
#include<stdio.h>
#include<conio.h>
void modify(int b[3]);
void main()
{
int arr[3] = {1,2,3};
modify(arr);
for(i=0;i<3;i++)
printf("%d",arr[i]);
getch();
}
void modify(int a[3])
{
int i;
for(i=0;i<3;i++)
a[i] = a[i]*a[i];
}
```

**Output :**

1 4 9

Here “arr” is same as “a” because Base Address of Array “arr” is stored in Array “a”

**Alternate Way of Writing Function Header :**

void modify(**int** a[3]) **OR** void modify(**int** \*a)

## **Passing array element by element to function :**

- ❑ Individual element is passed to function using **Pass By Value** parameter passing scheme
- ❑ Original Array elements remains same as Actual Element is never Passed to Function. thus function body cannot modify **Original Value**.
- ❑ Suppose we have declared an array ‘arr[5]’ then its individual elements are arr[0],arr[1]...arr[4]. Thus we need 5 function calls to pass complete array to a function.

## **Tabular Explanation :**

Consider following array

Iteration	Element Passed to Function	Value of Element
1	arr[0]	11
2	arr[1]	22
3	arr[2]	33
4	arr[3]	44
5	arr[4]	55

## **C Program to Pass Array to Function Element by Element :**

```
#include< stdio.h>
#include< conio.h>
void fun(int num)
{
printf("\nElement : %d",num);
}
void main()
{
int arr[5],i;
clrscr();
printf("\nEnter the array elements : ");
for(i=0;i< 5;i++)
scanf("%d",&arr[i]);
printf("\nPassing array element by element.....");
for(i=0;i< 5;i++)
fun(arr[i]);
getch(); }
```

### **Output :**

Enter the array elements : 1 2 3 4 5

Passing array element by element.....

Element : 1

Element : 2

Element : 3

Element : 4

Element : 5

### **DISADVANTAGE OF THIS SCHEME :**

- ❑ This type of scheme in which we are calling the function again and again but with **different array element is too much time consuming**. In this scheme we need to call function by pushing the current status into the system stack.
- ❑ It is better to pass complete array to the function so that we can save some system time required for pushing and popping.

### 3.18 RECURSION CONCEPT :

- Recursion is **basic concept** in Programming.
- When Function is defined in terms of itself then it is called as **“Recursion Function”**.
- Recursive function is a **function which contains a call to itself**.

```
int factorial(int n)
{
    if(n==0)
        return(1);
    else
        return( n * factorial(n-1));
}
```

$$\text{Factorial } (n) = \begin{cases} 1 & \\ n * \text{Factorial } (n-1) & \end{cases}$$

Example:

```
#include <stdio.h>
int sum(int n);
int main()
{
    int number, result;
    printf("Enter a positive integer: ");
    scanf("%d", &number);
    result = sum(number);
    printf("sum=%d", result);
}

int sum(int num)
{
    if (num!=0)
        return num + sum(num-1); // sum() function calls itself
    else
        return num;
}
```

### Output

Enter a positive integer: 3 6

## How does recursion work?

```
void recurse()  
{  
    ... ... ...  
    recurse(); ————— recursive  
    ... ... ...  
}  
  
int main()  
{  
    ... ... ...  
    recurse(); —————  
    ... ... ...  
}
```

```

int main() {
    ...
    result = sum(number) ← [3]
    ...
}
int sum(int n)
{
    if(n!=0) [3] ← [2]
        return n + sum(n-1);
    else
        return n;
}
int sum(int n)
{
    if(n!=0) [2] ← [1]
        return n + sum(n-1);
    else
        return;
}
int sum(int n)
{
    if(n!=0) [1] ← [0]
        return n + sum(n-1);
    else
        return n;
}
int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return n;
}

```

$3+3 = 6$   
is returned

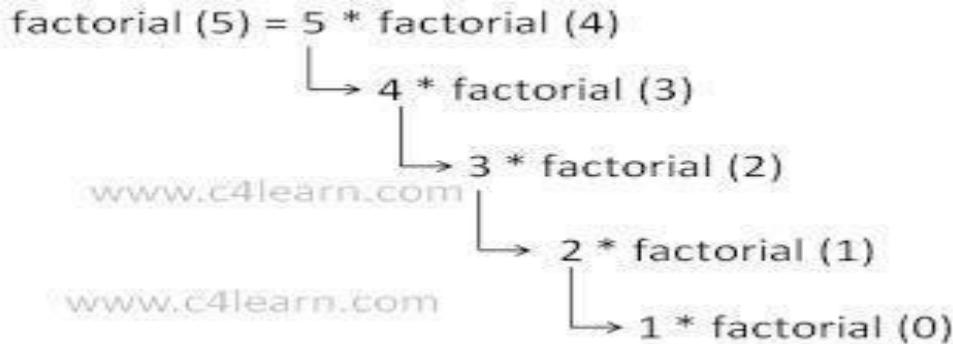
$1+2 = 3$   
is returned

$0+1 = 1$   
is returned

$0$   
is returned

## **Warning of using recursive function in C Programming :**

- ❑ Recursive function must have at least one terminating condition that can be satisfied.
- ❑ Otherwise, the recursive function will call itself repeatedly until the run time stack overflows.



## **Advantages and Disadvantages of Recursion**

- ❑ Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.
- ❑ If the speed of the program is vital then, you should avoid using recursion. Recursions use more memory and are generally slow.

Instead, you can use [loop](#).

Thank you

# 18CSS101J – Programming for Problem Solving

## Unit IV

# **UNIT IV**

## **INTRODUCTION**

Passing Array Element to Function-Formal and Actual Parameters-Advantages of using Functions-Processor Directives and #define Directives-Nested Preprocessor Macro-Advantages of using Functions-Functions-Pointers and address operator-Size of Pointer Variable and Pointer-Operator-Pointer Declaration and dereferencing-pointers-Void Pointers and size of Void Pointers-Arithmetic Operations-Incrementing Pointers-Pointers-Constant Pointers-Pointers to array elements and strings-Function Pointers-Array of Function Pointers-Accessing Array of Function Pointers-Null Pointers-Pointers

## INTRODUCTION

- Arrays can be passed as a parameter to a function.
- When it is being passed, the name of the array is enough as an argument instead passing it as an entire array.
- Since the name of the array itself holds the address, no need to include & in the function call.
- Since it passes address , in case of any modification occurs in called function automatically reflects back in the calling function.

# Introduction

- Multidimensional arrays are also allowed to be passed as arguments to functions.
- The first dimension is omitted when a multidimensional array is used as a formal parameter in a function.

# Formal and Actual Parameters

## One-Dimensional Array

- **Actual Parameters:**

```
int a[10];
```

**Function call:**

```
add(a);
```

here,

a is the base address of the array

add is the function

# Formal and Actual Parameters

## One-Dimensional Array

- **Formal Parameters:**

within the function body

```
void add(int a[])
```

```
{
```

```
....
```

```
}
```

int a[] - is a formal parameter, since it is a one dimensional array size doesn't matter.

# Formal and Actual Parameters

## Two-Dimensional Array

- **Actual Parameters:**

int a[10][10];

**Function call:**

add(a);

here,

a is the base address of the array ‘a’

b is the base address of the array ‘b’

add is the function

# Formal and Actual Parameters

## One-Dimensional Array

- **Formal Parameters:**

within the function body

```
void add(int a[][10])
```

```
{
```

```
....
```

```
}
```

int a[][10] - is a formal parameter, the first dimension is omitted only column value is taken into the account.

# Passing Single element

- **Example**

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

**Function call:**

```
add(a[1]);
```

Here,

add is the function name,

a[1] is the second element of the array, value 1 is passed to the array.

# Adding Constant five to the given array using functions(One-Dimensional)

## • Main Function

```
int main()
{
    int a[50],i,n;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
add(a,n);
    for(i=1;i<=n;i++)
        printf("%d\n",a[i]);
    return 0;
}
```

# Adding Constant five to the given array using functions(One-Dimensional)

- **Add Function:**

```
void add(int a[],int n)
{
    int r;
    for(r=1;r<=n;r++)
    {
        a[r]=a[r]+5;
    }
}
```

# Output

- **Input:**

5

2 4 6 8 10

- **Output:**

7

9

11

13

15

# Main Function

```
int main()
{
    int num[2][2], i, j;
    printf("Enter 4 numbers:\n");
    for (i = 0; i < 2; ++i)
    {
        for (j = 0; j < 2; ++j)
        {
            scanf("%d", &num[i][j]);
        }
    }
    // passing multi-dimensional array to displayNumbers function
    displayNumbers(num);
    return 0;
}
```

# Display Numbers Function

```
void displayNumbers(int num[2][2])
{
    // Instead of the above line,
    // void displayNumbers(int num[][][2]) is also valid
    int i, j;
    printf("Displaying:\n");
    for (i = 0; i < 2; ++i)
    {
        for (j = 0; j < 2; ++j)
        {
            printf("%d\n", num[i][j]);
        }
    }
}
```

# The advantages of using functions are:

- Divide and conquer
  - Manageable program development
- Software reusability
  - Use existing functions as building blocks for new programs
  - Abstraction - hide internal details (library functions)
- Avoid code repetition

# Preprocessor Directives MACROS

- The C preprocessor is a macro processor that is used automatically by the C compiler to transform your program before actual compilation.
- A macro is a segment of code which is replaced by the value of 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 **#**.

# List of preprocessor directives

#include  
#define  
#undef  
#ifdef  
#ifndef  
#if  
#else  
#elif  
#endif  
#error  
#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 us, then in a standard list of system directories.

## #include "file"

- This variant is used for header files of your own program.
- It searches for a file named file first in the current directory, then in the same directories used for system header files.
- The current directory is the directory of the current input file.

## #include anything else

- This variant is called a computed #include.
- Any #include directive whose argument does not fit the above two forms is a computed include.

## Macro's (#define)

#define token value There are two types of macros:

1. Object-like Macros
2. Function-like Macros

### Object-like Macros

- The object-like macro is an identifier that is replaced by value.
- It is widely used to represent numeric constants.

For example:

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

Output:

3.14000

# Function-like Macros

- The function-like macro looks like function call.
- For example:`#define MIN(a,b) ((a)<(b)?(a):(b))`
- Here, MIN is the macro name.

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))
void main()
{
    printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
}
```

## Output:

Minimum between 10 and 20 is: 10

# #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); }
```

## Output

Compile Time Error: 'PI' undeclared

## #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 expression or condition. If condition is true, it executes the code.

### 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**

## **Example**

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 1
void main()
{
#if NUMBER==0
printf("Value of Number is: %d",NUMBER);
#else print("Value of Number is non-zero");
#endif getch();
}
```

## **Output**

Value of Number is non-zero

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

## Output

I am in func  
I am in main  
I am in func

# Advantages of Macros

- Time efficiency.
- Not need to pass arguments like function.
- It's preprocessed.
- Easier to Read

# Disadvantages of Macros

- Very hard to debug in large code.
- Take more memory compare to function

# POINTERS

## Definition

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand.

# POINTERS

- Syntax

Datatype \*pointervariable;

- Syntax Example

1. int \*ip; /\* pointer to an integer \*/
2. double \*dp; /\* pointer to a double \*/
3. float \*fp; /\* pointer to a float \*/
4. char \*ch /\* pointer to a character \*/

# POINTERS

- Example

```
int var = 20; /* actual variable declaration */
```

```
int *ip; /* pointer variable declaration */
```

```
ip = &var; /* store address of var in pointer  
variable */
```

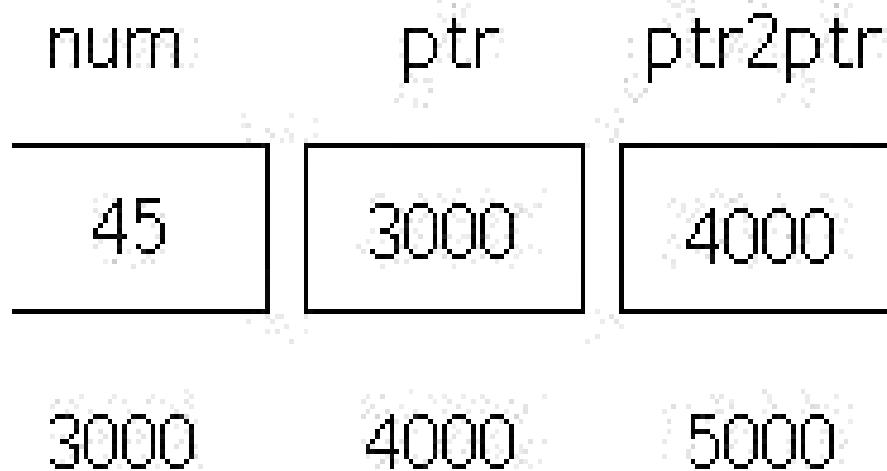
# POINTERS

## Reference operator (&) and Dereference operator (\*)

- 1.& is called reference operator. It gives the address of a variable.
- 2.\* is called dereference operator. It gives the value from the address

# Pointer to Pointer

Double (\*\*) is used to denote the double pointer.  
Double Pointer Stores the address of the Pointer Variable. Conceptually we can have Triple ..... n pointers.



# Example 1

```
int main()
{
    int num = 45 , *ptr , **ptr2ptr ;
    ptr    = &num; //3000
    ptr2ptr = &ptr; //4000
    printf("%d",**ptr2ptr);
    return(0);
}
```

Output 45

# Pointer to Constant Objects

These type of pointers are the one which cannot change the value they are pointing to. This means they cannot change the value of the variable whose address they are holding.

const datatype \*pointername;

(or)

datatype const \*pointername;

# Example

The pointer variable is declared as a const. We can change address of such pointer so that it will point to new memory location, but pointer to such object cannot be modified (\*ptr).

# Example

The screenshot shows the Turbo C++ IDE interface. The title bar reads "Turbo C++ IDE". The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. A tab labeled "CPOINTER.C" is open, showing the following C code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int num[2]={20,70};
    const int *ptr;
    clrscr();
    ptr=&num[0];
    ptr++;
    num[1]=20;
    *ptr=30;
    printf("%d\n%d",*ptr,ptr);
    getch();
}
```

The line `*ptr=30;` is highlighted with a green selection bar. The status bar at the bottom left shows the time as 11:14. The message window at the bottom displays the compilation output:

[1] Message 2=[1]

Compiling CPOINTER.C:  
•Error CPOINTER.C 11: Cannot modify a const object

The keyboard navigation keys F1 through F10 are visible at the bottom of the message window.

# Constant Pointers

Constant pointers are the one which cannot change address they are pointing to. This means that suppose there is a pointer which points to a variable (or stores the address of that variable). If we try to point the pointer to some other variable, then it is not possible.

```
int* const ptr=&variable;
```

(or)

```
int *const ptr=&variable // ptr is a constant pointer to int
```

# Null pointer

- NULL Pointer is a pointer which is pointing to nothing.
- Pointer which is initialized with NULL value is considered as NULL pointer.

```
datatype *pointer_variable=0;
```

```
datatype *pointer_variable=NULL;
```

# Example

The screenshot shows the Turbo C++ IDE interface. The title bar reads "Turbo C++ IDE". The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. The current file is "CONPOI.C", which contains the following C code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int *ptr=NULL;
    clrscr();
    printf("%d",ptr);
    getch();
}
```

The status bar at the bottom displays "5:16" and various keyboard shortcuts: F1 Help, Alt-F8 Next Msg, Alt-F7 Prev Msg, Alt-F9 Compile, F9 Make, and F10 Menu.

# Pointer Arithmetic

- C allows you to perform some arithmetic operations on pointers.
- **Incrementing a pointer**

Incrementing a pointer is one which increases the number of bytes of its data type.

```
int *ptr;  
int a[]={1,2,3};  
ptr=&a;  
ptr++;  
ptr=&a;  
ptr=ptr+1;
```

# Pointer Arithmetic

- **Decrementing a Pointer**

Decrementing a pointer is one which decreases the number of bytes of its data type.

## Using Unary Operator

```
int *ptr;  
int a[]={1,2,3};  
ptr=&a;  
ptr--;
```

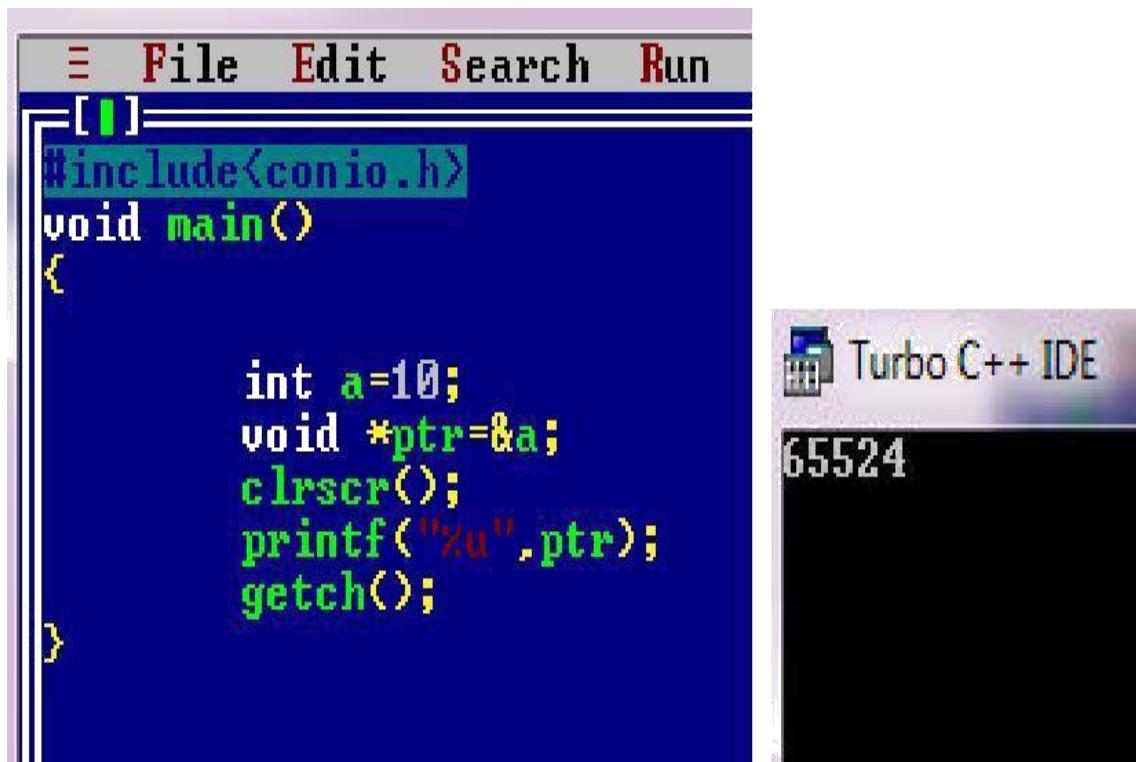
# Limitations of Pointer Arithmetic

- Addition of 2 pointers is not allowed
- Addition of a pointer and an integer is commutative  $\text{ptr} + 5 \overset{\text{?}}{=} 5 + \text{ptr}$
- Subtraction of 2 pointers is applicable.
- subtraction of a pointer and an integer is not commutative  $\text{ptr} - 5 \overset{\text{?}}{\neq} 5 - \text{ptr}$ .
- Only integers can be added to pointer. It is not valid to add a float or double value to a pointer.
- A pointer variable cannot be assigned a non address value except zero.
- Multiplication and division Operators cannot be applied on pointers.
- Bitwise operators cannot be applied on pointers.
- A pointer and an integer can be subtracted.
- A pointer and an integer can be added.

# Void pointer

1. Void pointer is a generic pointer and can point to any type of object. The type of object can be char, int, float or any other type.

- Example



The image shows two windows from the Turbo C++ IDE. The left window is the code editor with the following C code:

```
#include<conio.h>
void main()
{
    int a=10;
    void *ptr=&a;
    clrscr();
    printf("%u",ptr);
    getch();
}
```

The right window is the terminal window displaying the output of the program, which is the memory address of variable 'a': 65524.

2.A pointer to any type of object can be assigned to a void pointer.

The screenshot shows the Turbo C++ IDE interface. The title bar reads "Turbo C++ IDE". The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. The current file is "CONPOI.C". The code in the editor is:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=10;
    int *iptr=&a;
    void *vptr=iptr;
    clrscr();
    printf("int* is implicitly converted to void* %d\n",vptr,iptr);
    getch();
}
```

The status bar at the bottom shows the time as "10:37" and various keyboard shortcuts: F1 Help, F2 Save, F3 Open, Alt-F9 Compile, F9 Make, F10 Menu.

OUTPUT

The screenshot shows the execution output window of the Turbo C++ IDE. It displays the message: "int\* is implicitly converted to void\* 65524".

### 3. A void pointer cannot be dereferenced

The screenshot shows the Turbo C++ IDE interface. The title bar reads "Turbo C++ IDE" and the file name is "CONPOI.C". The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. The status bar at the bottom has buttons for F1 Help, Space, View source, Edit source, and F10 Menu.

```
#include<conio.h>
void main()
{
    int a=10;
    void *ptr=&a;
    clrscr();
    printf("%d",*ptr);
    getch();
}
```

The message window displays the compilation results:

[ 1 ] = Message = 2=[ ↑ ]=

Compiling CONPOI.C:  
•Error CONPOI.C 9: Not an allowed type  
Warning CONPOI.C 11: 'ptr' is assigned a value that is never used

# Relational operations

- A pointer can be compared with a pointer of same type or zero.
- Various relational operators are ==., !=,<,<=,>,>=
- Ex: float a=1.0,b=2.0,\*fptr1,\*fptr2;
- fptr1=&a;fptr2=&b;
- int result;

**result=fptr1!=fptr2;**

# Example

The screenshot shows a window titled "Turbo C++ IDE" with a menu bar containing File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. The main window displays a C program named "CONPOI.C". The code uses pointer arithmetic to compare two variables and prints the result. The status bar at the bottom shows the time as 9:21.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    float *ptr1,*ptr2,a=1.0,b=2.0;
    int result;
    ptr1=&a;ptr2=&b;
    result=ptr1!=ptr2;
    clrscr();
    printf("%d",result);
    getch();
}
```

F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

# Pointers and Arrays

- Pointers and arrays are closely related , An array variable is actually just a pointer to the first element in the array.
- Accessing array elements using pointers is efficient way than using array notation.
- When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array.
- Base address i.e address of the first element of the array is also allocated by the compiler.

# Pointers and Arrays Cont...

Suppose we declare an array **arr**,

`int arr[5]={ 1, 2, 3, 4, 5 };` Assuming that the base address of **arr** is 1000 and each integer requires two bytes, the five elements will be stored as follows

element	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Address	1000	1002	1004	1006	1008

# Pointers and Arrays Cont...

- Here variable **arr** will give the base address, which is a constant pointer pointing to the element, **arr[0]**.
- Therefore **arr** is containing the address of **arr[0]** i.e 1000. In short, arr has two purpose- it is the name of an array and it acts as a pointer pointing towards the first element in the array.

# Pointers and Arrays Cont...

```
int *p;
```

```
p = arr;
```

or

```
p = &arr[0];
```

Now we can access every element of array **arr** using **p++** to move from one element to another.

# Pointers and Arrays Cont...

- $a[0]$  is the same as  $*a$
- $a[1]$  is the same as  $*(a + 1)$
- $a[2]$  is the same as  $*(a + 2)$
- If  $pa$  points to a particular element of an array,  $(pa + 1)$  always points to the next element,  $(pa + i)$  points  $i$  elements after  $pa$  and  $(pa - i)$  points  $i$  elements before.

# Example 1

```
#include<stdio.h>
void main()
{
    int a[10],i,n;
    printf("Enter n");
    scanf("%d",&n);
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    for(i=0;i<n;i++)
        printf("a[%d]=%d\n",i,*(&a[i]));
}
```

# Output

Enter n 5

1 2 3 4 5

a[0]=1

a[1]=2

a[2]=3

a[3]=4

a[4]=5

# Pointer to Multidimensional Array

- A multidimensional array is of form,  $a[i][j]$ . Lets see how we can make a pointer point to such an array.
- As we know now, name of the array gives its base address. In  $a[i][j]$ , **a** will give the base address of this array, even  $a+0+0$  will also give the base address, that is the address of **a[0][0]** element.
- Here is the generalized form for using pointer with multidimensional arrays.
- $\ast(\ast(a + i) + j)$  is same as  $a[i][j]$ .

# Example 2

```
#include <stdio.h>
#define ROWS 4
#define COLS 3
int main ()
{
    int i,j;
    // declare 4x3 array
    int matrix[ROWS][COLS] = {{1, 2, 3},
                               {4, 5, 6},
                               {7, 8, 9},
                               {10, 11, 12}};
    for (i = 0; i < ROWS;i++)
    {
        for (j = 0;j < COLS;j++)
        {
            printf("%d\t",matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

# Output

1      2      3

4      5      6

7      8      9

10     11     12

`matrix[0][0] = *(*matrix))`

`matrix[i][j] = *((*(matrix)) + (i * COLS + j))`

`matrix[i][j] = *(*(matrix + i) + j)`

`matrix[i][j] = *(matrix[i] + j)`

`matrix[i][j] = (*(matrix + i))[j]`

`&matrix[i][j] = ((*(matrix)) + (i * COLS + j))`

## Example 3

```
void array_of_arrays_ver(int arr[][COLS])
{
    int i,j;
    for (i = 0; i < ROWS;i++)
    {
        for (j = 0;j < COLS;j++)
        {
            printf("%d\t",arr[i][j]);
        }
        printf("\n");
    }
}
void ptr_to_array_ver(int (*arr)[COLS])
{
    int i,j;
    for (i = 0; i < ROWS;i++)
    {
        for (j = 0;j < COLS;j++)
        {
            printf("%d\t",(*arr)[j]);
        }
        arr++;
        printf("\n");
    }
}
```

# Output

```
C:\Users\prakash\Desktop\dektopfile\unit5pdd\pro3.exe
Printing Array Elements by Array of Arrays Version Function:
1      2      3
4      5      6
7      8      9
10     11     12
Printing Array Elements by Pointer to Array Version Function:
1      2      3
4      5      6
7      8      9
10     11     12
Process returned 0 (0x0)  execution time : 0.026 s
Press any key to continue.
```

# Double Pointer

```
#include <stdio.h>
#define ROWS 4
#define COLS 3
int main ()
{
    // matrix of 4 rows and 3 columns
    int matrix[ROWS][COLS] = {{1, 2, 3},
                               {4, 5, 6},
                               {7, 8, 9},
                               {10, 11, 12}};
    int** pmat = (int **)matrix;
    printf("&matrix[0][0] = %u\n", &matrix[0][0]);
    printf("&pmat[0][0] = %u\n", &pmat[0][0]);
    return 0;
}
```

# Output

`&matrix[0][0] = 2675498000`

`&pmat[0][0] = 1`

# Passing an array to a function

- Single element of an array can be passed in similar manner as passing variable to a function
- **C program to pass a single element of an array to function**

```
#include <stdio.h>
void display(int age)
{
    printf("%d", age);
}

int main()
{
    int ageArray[] = { 2, 3, 4 };
    display(ageArray[2]); //Passing array element ageArray[2] only.
    return 0;
}
```

OUTPUT

4

# Passing an entire one-dimensional array to a function

```
#include<stdio.h>
int total(int[],int);
void main()
{
    int a[10],n,sum,i;
    printf("Enter n\n");
    scanf("%d",&n);
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    sum=total(a,n);
    printf("Sum=%d",sum);
}
int total(int a[],int n)
{
    int sum=0,i;
    for(i=0;i<n;i++)
        sum=sum+a[i];
    return sum;
}
```

# Output

Enter n

5

1

2

3

4

5

**Sum=15**

# Passing Multi-dimensional Arrays to Function

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

# Example 4

```
#include <stdio.h>

void displayNumbers(int num[2][2]);

int main()
{
    int num[2][2], i, j;
    printf("Enter 4 numbers:\n");
    for (i = 0; i < 2; ++i)
        for (j = 0; j < 2; ++j)
            scanf("%d", &num[i][j]);

    // passing multi-dimensional array to displayNumbers function
    displayNumbers(num);
    return 0;
}
```

```
void displayNumbers(int num[2][2])
{
    // Instead of the above line,
    // void displayNumbers(int num[][2]) is also valid
    int i, j;
    printf("Displaying:\n");
    for (i = 0; i < 2; ++i)
        for (j = 0; j < 2; ++j)
            printf("%d\n", num[i][j]);
}
```



# Output

Enter 4 numbers:

1

2

3

4

Displaying:

1

2

3

4

# Pointers and Strings

- **Strings as arrays:** In C, the abstract idea of a string is implemented with just an array of characters. For example, here is a string:
- `char label[] = "Single";` What this array looks like in memory is the following:



# Pointers and Strings cont...

- where the beginning of the array is at some location in computer memory, for example, location 1000.
- A character array can have more characters than the *abstract string* held in it, as below:
- `char label[10] = "Single";` giving an array that looks like:



- we can access each character in the array using subscript notation, as in:

```
printf("Third char is: %c\n", label[2]);
```

- which prints out the third character, **n**.

# Disadvantage Of Creating Strings Using The Character Array

- A disadvantage of creating strings using the character array *syntax* is that you must say ahead of time how many characters the array may hold. For example, in the following array definitions, we state the number of characters (either implicitly or explicitly) to be allocated for the array.

```
char label[] = "Single"; /* 7 characters */
```

```
char label[10] = "Single";
```

- Thus, you must specify the maximum number of characters you will ever need to store in an array.
- This type of array allocation, where the size of the array is determined at compile-time, is called *static allocation*.

# String as Pointers

- Another way of accessing a *contiguous* chunk of memory, instead of with an array, is with a *pointer*.
- However, pointers only hold an address, they cannot hold all the characters in a character array.
- This means that when we use a `char *` to keep track of a string, the character array containing the string must already exist (having been either statically- or dynamically-allocated).

```
char label[] = "Single";  
char label2[10] = "Married";  
char *labelPtr;  
  
labelPtr = label;
```

# Example1

```
#include <stdio.h>

int main()
{
    char *sample = "From whence cometh my help?\n";
    while(putchar(*sample++))
    ;
    return(0);
}
```

## Output:

From whence cometh my help?\n

# Passing Strings

Below is the definition of a function that prints a label and a call to that function:

```
void PrintLabel(char the_label[])
{
    printf("Label: %s\n", the_label);
}

int main(void)
{
    char label[] = "Single";
    ...
    PrintLabel(label);
    ...
}
```

# Passing Strings Cont...

- Since label is a character array, and the function PrintLabel() expects a character array, the above makes sense.
- However, if we have a pointer to the character array label, as in:

```
char *labelPtr = label;
```

then we can also pass the pointer to the function, as in:

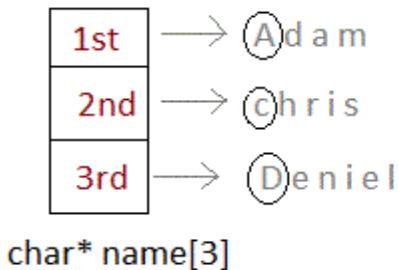
- PrintLabel(labelPtr);

# Array of Pointers

- Pointers are very helpful in handling character array with rows of varying length.

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

## Using Pointer



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

## Without Pointer

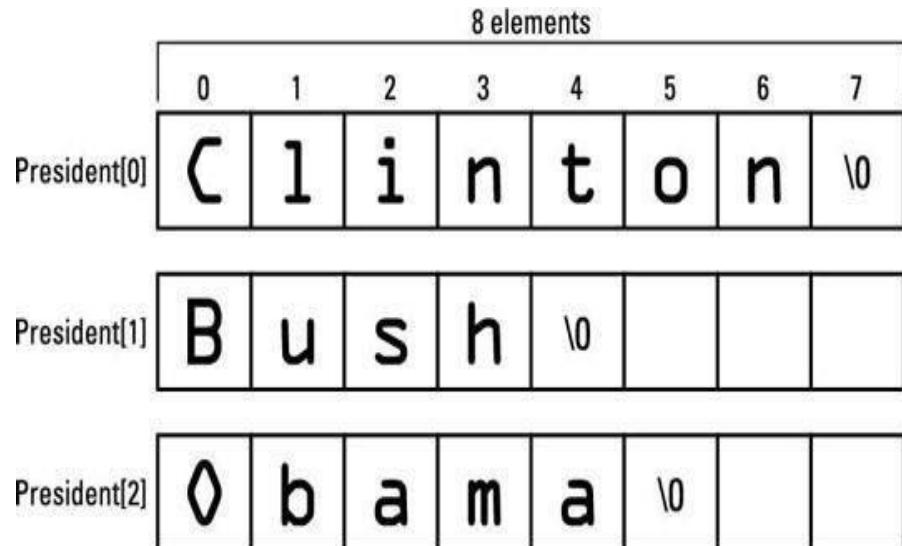
A	d	a	m			
c	h	r	i	s		
D	e	n	i	e	l	

char name[3][20]

extends till 20  
memory locations

# Example1

```
#include <stdio.h>
#define SIZE 3
int main()
{
    char president[SIZE][8] = {
        "Clinton",
        "Bush",
        "Obama"
    };
    int x,index;
    for(x=0;x<SIZE;x++)
    {
        index = 0;
        while(president[x][index] != '\0')
        {
            putchar(president[x][index]);
            index++;
        }
        putchar('\n');
    }
    return(0);
}
```



**Output:**

ClintonnBushnObaman

# Function Pointers

- In C, like normal data pointers (int \*, char \*, etc), we can have pointers to functions.
- Initialization

```
return_type function_pointer(argu)=&function_name  
void (*fun_ptr)(int) = &fun;
```

- Function Definition

```
void fun(int a)  
{  
    printf("Value of a is %d\n", a);  
}
```

# Example1

```
#include<stdio.h>

void fun(int a)
{
    printf("a=%d\n",a);
}

void main()
{
    void (*fun1)(int)=&fun;
    (*fun1)(15);
}
```

**Output:**

a=15

# Function Pointers Cont...

If we remove bracket, then the expression

“void (\*fun\_ptr)(int)”

becomes

“void \*fun\_ptr(int)”

which is declaration of a function that returns void pointer.

## Interesting facts

- Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
- Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
- A function’s name can also be used to get functions’ address.

# Passing Function Pointer as an argument

```
#include<stdio.h>
void fun1(void(*test)())
{
int a=20;
test(a);
}
void test(int a)
{
printf("a=%d\n",a);
}
void main()
{
fun1(&test);
}
```

**Output:**

a=20

# Array of Function Pointer

```
#include<stdio.h>
void add(int a,int b)
{
    printf("add=%d\n",a+b);
}
void sub(int a,int b)
{
    printf("sub=%d\n",a-b);
}
void mul(int a,int b)
{
    printf("mul=%d\n",a*b);
}
```

```
void main()
{
    void
    (*fun[])(int,int)={add,sub,mul};
    int ch;
    int a,b;
    printf("Enter a and b\n");
    scanf("%d%d",&a,&b);
    printf("Enter the
operation\n");
    scanf("%d",&ch);
    if(ch>2)
        printf("Wrong Input\n");
    else
        (*fun[ch])(a,b);
}
```



# Output

Enter a and b

4 5

Enter the operation

2

mul=20

# Structures created and accessed using 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 accessed through pointer in two ways:

1. Referencing pointer to another address to access memory
2. Using dynamic memory allocation

# Referencing pointer to another address to access the memory

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

```
#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:\n");
    printf("age=%d\nweight=%f", (*personPtr).age, (*personPtr).weight);
    return 0;
}
```

# Output

Enter integer: 3

Enter number: 6

Displaying:

age=3

weight=6.000000

->

- **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`

# 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)`

# Example1

```
#include <stdio.h>
#include <stdlib.h>
struct person {
    int age;
    float weight;
    char name[30];
};
int main()
{
    struct person *ptr;
    int i, num;
    printf("Enter number of persons: ");
    scanf("%d", &num);
    ptr = (struct person*) malloc(num *
sizeof(struct person));
    // Above statement allocates the memory for n
    structures with 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);
}

printf("Displaying Information:\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:

aaa

12

45

Enter name, age and weight of the person respectively:

bbb

87

65

Displaying Information:

aaa 12 45.00

bbb 87 65.00

# Self-Referential Structure

- A self referential structure is used to create data structures like linked lists, stacks, etc. Following is an example of this kind of structure:

```
struct struct_name
{
    datatype datatypename;
    struct_name * pointer_name;
};
```

# Self-Referential Structure Cont...

- A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type.
- For example, a linked list is supposed to be a self-referential data structure. The next node of a node is being pointed, which is of the same struct type. For example,

```
typedef struct listnode {  
    void *data;  
    struct listnode *next;  
} linked_list;
```

In the above example, the listnode is a self-referential structure – because the \*next is of the type struct listnode.

# Self-Referential Structure Cont...

```
typedef struct listnode {  
    void *data;  
    struct listnode *next;  
} linked_list;
```



**THANK YOU**

# 18CSS101J – Programming for Problem Solving

## Unit V

COURSE LEARNING RATIONALE (CLR)		<i>The purpose of learning this course is to:</i>
<b>CLR -1:</b>	Think and evolve a logically to construct an algorithm into a flowchart and a pseudocode that can be programmed	
<b>CLR -2:</b>	Utilize the logical operators and expressions to solve problems in engineering and real-time	
<b>CLR -3:</b>	Store and retrieve data in a single and multidimensional array	
<b>CLR -4:</b>	Utilize custom designed functions that can be used to perform tasks and can be repeatedly used in any application	
<b>CLR -5:</b>	Create storage constructs using structure and unions. Create and Utilize files to store and retrieve information	
<b>CLR -6:</b>	Create a logical mindset to solve various engineering applications using programming constructs in C	

COURSE LEARNING OUTCOMES (CLO)		<i>At the end of this course, learners will be able to:</i>
<b>CLO -1:</b>	Identify methods to solve a problem through computer programming. List the basic data types and variables in C	
<b>CLO -2:</b>	Apply the logic operators and expressions. Use loop constructs and recursion. Use array to store and retrieve data	
<b>CLO -3:</b>	Analyze programs that need storage and form single and multi-dimensional arrays. Use preprocessor constructs in C	
<b>CLO -4:</b>	Create user defined functions for mathematical and other logical operations. Use pointer to address memory and data	
<b>CLO -5:</b>	Create structures and unions to represent data constructs. Use files to store and retrieve data	
<b>CLO -6:</b>	Apply programming concepts to solve problems. Learn about how C programming can be effectively used for solutions	

## LEARNING RESOURCES

S. No	TEXT BOOKS
1.	<i>Zed A Shaw, Learn C the HardWay: Practical Exercises on the Computational Subjects You Keep Avoiding (Like C), AddisonWesley, 2015</i>
2.	<i>W.Kernighan, Dennis M. Ritchie, The C Programming Language, 2nd ed. Prentice Hall, 1996</i>
3.	<i>Bharat Kinariwala, Tep Dobry, Programming in C, eBook</i>
4.	<u><a href="http://www.c4learn.com/learn-c-programming-language/">http://www.c4learn.com/learn-c-programming-language/</a></u>

## **UNIT V**

### **INTRODUCTION**

Initializing Structure, Declaring Structure variable- Structure using typedef, Accessing members – Nested structure Accessing elements in a structure array – Array of structure Accessing elements in a structure array –Passing Array of Structure to function- Array of Pointers to structures- Bit Manipulation of structure and pointer to structure – Union Basic and declaration – Accessing Union Members Pointers to union

## **UNIT V**

### **INTRODUCTION**

Dynamic memory allocation, malloc, realloc , free – Allocating  
Dynamic Array- Multidimensional array using dynamic  
memory allocation- file: opening, defining, closing, File Modes,  
File Types- Writing contents into a file – Reading file contents –  
Appending an existing file- File permissions and rights-  
changing permissions and rights.

# INTRODUCTION TO STRUCTURE

- Problem:  
– How to group together a collection of data items of different types that are logically related to a particular entity??? (**Array**)

*Solution:* **Structure**

# STRUCTURE

- ❑ A Structure is a collection of variables of different data types under a singlename.
- ❑ The variables are called **members** of the structure.
- ❑ The structure is also called a user-defined data type.

# Defining a Structure

- Syntax:

```
struct structure_name  
{  
    data_type member_variable1; data_type  
    member_variable2;  
    .....;data_type member_variableN;  
};
```

Once *structure\_name* is declared as new data type, then variables of that type can be declared as:

```
struct structure_name structure_variable;
```

**Note: The members of a structure do not occupy memory until they are associated with a structure\_variable.**

- Example

```
struct student
{
    char name[20];
    int roll_no;
    float marks;
    char gender;
    long int phone_no;
};
```

*struct student st;*

- Multiple variables of *struct student* type can be declared as:

*struct student st1, st2, st3;*

# Defining a structure...

- Each variable of structure has its own copy of member variables.
- The member variables are accessed using the dot (.) operator or memberoperator.
- For example: *st1.name* is member variable *name* of *st1* structure variable while *st3.gender* is member variable *gender* of *st3* structure variable.

# Defining a structure...

```
struct student
{
    charname[20];
    int roll_no;
    float marks;
    char gender;
    long int phone_no;
}st1, st2, st3;
```

```
struct
{
    charname[20]; int
    roll_no; floatmarks;
    char gender;
    long int phone_no;
}st1, st2, st3;
```

# Structure initialization

- Syntax:

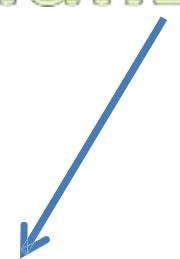
`struct structure_name structure_variable={value1, value2, ..., valueN};`

- Note: C does not allow the initialization of individual structure members within the structure definition template.

```
char name[20];
int roll_no;
float marks;
char gender;
long int phone_no;
};

void main()
{
    struct student st1={"ABC", 4, 79.5, 'M', 5010670};
    clrscr();
    printf("Name\t\tRoll No.\tMarks\tGender\tPhone No.");
    printf("\n.....\n");
    printf("\n %s\t\t %d\t\t %f\t%c\t %ld", st1.name, st1.roll_no, st1.marks,
          st1.gender, st1.phone_no);
    getch();
}
```

Initialization



# Partial Initialization

- We can initialize the first few members and leave the remaining blank.
- However, the uninitialized members should be only at the end of the list.
- The uninitialized members are assigned default values as follows:
  - **Zero** for integer and floating point numbers.
  - **'\0'** for characters and strings.

```
struct  
stude  
nt  
char name[20];  
int roll;  
char remarks;  
float marks;  
};  
void main()  
{  
struct student s1={"name", 4};  
clrscr();  
printf("Name=%s", s1.name);  
printf("\n Roll=%d", s1.roll);  
printf("\n Remarks=%c", s1.remarks);  
printf("\n Marks=%f", s1.marks);  
getch();  
}
```

# Accessing member of structure/ Processing a structure

- By using dot (.) operator or period operator or member operator.
- Syntax:

*structure\_variable.member*

- Here, *structure\_variable* refers to the name of a *struct* type variable and *member* refers to the name of a member within the structure.

# Question

- Create a structure named *student* that has *name*, *roll* and *mark* as members. Assume appropriate types and size of member. Write a program using structure to read and display the data entered by the user.

```
struct student
{
    char name[20];
    int roll;
    float mark;
};

void main()
{
    struct students;
    clrscr();
    printf("Enter name:\t");
    gets(s.name);
    printf("\n Enter roll:\t");
    scanf("%d", &s.roll);
    printf("\n Enter marks:\t");
    scanf("%f", &s.mark);
    printf("\n Name \t Roll \t Mark\n");
    printf(".....\n");
    printf("\n%s\t%d\t%f", s.name, s.roll, s.mark);
    getch();
}
```

# Copying and Comparing Structure Variables

- Two variables of the same structure type can be copied in the same way as ordinary variables.
- If *student1* and *student2* belong to the same structure, then the following statements are valid:  
*student1=student2; student2=student1;*
- However, the statements such as:  
*student1==student2 student1!=student2*  
are not permitted.
- If we need to compare the structure variables, we may do so by comparing members individually.

```
structstudent
{
    char name[20];
    int roll;
};

void main()
{
    struct student student1={"ABC", 4,};
    struct student student2;
    clrscr();
    student2=student1;
```

Here, structure has been declared global i.e. outside of main() function. Now, any function can access it and create a structure variable.

```
printf("\nStudent2.name=%s",
student2.name);
printf("\nStudent2.roll=%d",
student2.roll);
if(strcmp(student1.name,student2.name)==0 &&
    (student1.roll==student2.roll))
{
    printf("\n\n student1 and student2
are same.");
}
getch();
}
```

# How structure elements are stored?

- The elements of a structure are always stored in contiguous memory locations.
- A structure variable reserves number of bytes equal to sum of bytes needed to each of its members.
- Computer stores structures using the concept of “**word boundary**”. In a computer with two bytes word boundary, the structure variables are stored left aligned and consecutively one after the other (with at most one byte unoccupied in between them called **slackbyte**).

## How structure elements are stored?

- When we declare structure variables, each one of them may contain slack bytes and the values stored in such slack bytes are undefined.
- Due to this, even if the members of two variables are equal, their structures do not necessarily compare.
- That's why C does not permit comparison of structures.

# Array of structure

- Let us consider we have a structure as:

```
struct student  
{  
    char name[20];  
    int roll;  
    char remarks;  
    float marks;  
};
```

- If we want to keep record of 100 students, we have to make 100 structure variables like st1, st2, ..., st100.
- In this situation we can use array of structure to store the records of 100 students which is easier and efficient to handle (because loops can be used).

# Array of structure...

- Two ways to declare an array of structure:

- *struct student*

- {
- *char name[20]; int roll;*
- *char remarks; float marks;*
- *}st[100];*

```
struct student
{
    char name[20];
    int roll;
    char remarks;
    float marks;
};

struct student st[100];
```

- Write a program that takes roll\_no, fname, lname of 5 students and prints the same records in ascending order on the basis of roll\_no

# Reading values

```
for(i=0; i<5; i++)
{
    printf("\n Enter roll number:"); scanf("%d",
&s[i].roll_no);

    printf("\n Enter first name:"); scanf("%s",
&s[i].f_name);

    printf("\n Enter Lastname:"); scanf("%s",
&s[i].l_name);
}
```

# Question

- Define a structure of employee having data members name, address, age and salary. Take the data for n employees in an array and find the average salary.
- Write a program to read the *name*, *address*, and *salary* of 5 employees using array of structure. Display information of each employee in alphabetical order of their name.

# Array within Structure

- We can use single or multi dimensional arrays of type *int* or *float*.

- Eg.      *struct student*

```
{  
char name[20];  
int roll;  
float marks[6];  
};
```

```
struct students[100];
```

- Here, the member `marks` contains six elements, `marks[0]`, `marks[1]`, ..., `marks[5]` indicating marks obtained in six different subjects.
- These elements can be accessed using appropriate subscripts.
- For example, `s[25].marks[3]` refers to the marks obtained in the fourth subject by the 26<sup>th</sup> student.

## Array within structure...

# Reading Values

```
for(i=0;i<n;i++)
{
    printf("\n Enter information about student%d",i+1); printf("\n
Name:\t");
    scanf("%s", s[i].name); printf("\n
Class:\t"); scanf("%d", &s[i]._class);
    printf("\n Section:"); scanf("%c",
&s[i].section);
    printf("\n Input marks of 6subjects:\t"); for(j=0;j<6;j++)
{
    scanf("%f", &temp);
    s[i].marks[j]=temp;
}
}
```

# Structure within anotherStructure (Nested Structure)

- Let us consider a structure *personal\_record* to store the information of a personas:
- *struct personal\_record*

```
{  
    char name[20]; int day_of_birth;  
    int month_of_birth; int year_of_birth;  
    float salary;  
}person;
```

# Structure within anotherStructure (Nested Structure)...

- In the structure above, we can group all the items related to birthday together and declare them under a substructure as:

```
struct Date  
{  
    int day_of_birth; int month_of_birth; int  
    year_of_birth;  
};
```

```
struct personal_record  
{  
    char name[20]; struct Date birthday; float  
    salary;  
}person;
```

# Structure within anotherStructure (Nested Structure)...

- Here, the structure *personal\_record* contains a member named *birthday* which itself is a structure with 3 members. This is called structure within structure.
- The members contained within the inner structure can be accessed as:  
*person.birthday.day\_of\_birth*  
*person.birthday.month\_of\_birth* *person.birthday.year\_of\_birth*
- The other members within the structure *personal\_record* are accessed as usual:  
*person.name* *person.salary*

```
printf("Enter name:\t"); scanf("%s",
person.name); printf("\nEnter day of
birthday:\t");
scanf("%d", &person.birthday.day_of_birth);
printf("\nEnter month of birthday:\t");
scanf("%d", &person.birthday.month_of_birth);
printf("\nEnter year of birthday:\t");
scanf("%d", &person.birthday.year_of_birth);
printf("\nEnter salary:\t");
scanf("%f", &person.salary);
```

Structure within another Structure (Nested Structure)...

- ***Note:- More than one type of structures can be nested...***

```
struct  
date  
{  
int day;  
int month;  
int year;  
};
```

```
struct name  
{  
char first_name[10];  
char middle_name[10];  
char last_name[10];  
};
```

```
struct personal_record  
{  
float salary;  
struct date birthday, deathday;  
struct name full_name;  
};
```

# Assignment

- Create a structure named ***date*** that has ***day***, ***month*** and ***year*** as its members. Include this structure as a member in another structure named ***employee*** which has ***name***, ***id*** and ***salary*** as other members. Use this structure to read and display employee's name, id, date of birthday and salary.

# Pointer to Structure

- A structure type pointer variable can be declared as:

```
structbook  
{  
    char name[20];  
    int pages; float price;  
};  
struct book*bptr;
```

- However, this declaration for a pointer to structure does not allocate any memory for a structure but allocates only for a pointer, so that to access structure's members through pointer *bptr*, we must allocate the memory using *malloc()* function.

- Now, individual structure members are accessed as:

*bptr->name*      *bptr->pages*      *bptr->price*

*(\*bptr).name*      *(\*bptr).pages*      *(\*bptr).price*

- Here, -> is called arrow operator and there must be a pointer to the structure on the left side of this operator.

```
struct book*bptr;  
  
bptr=(struct book *)malloc(sizeof(struct book));  
  
printf("\n Enter name:\t");  
scanf("%s", bptr->name);  
printf("\n Enter no. of pages:\t");  
scanf("%d", &bptr->pages);  
printf("\n Enter price:\t");  
scanf("%f", &bptr->price=temp)
```

# Pointer to Structure...

- Also, the address of a structure type variable can be stored in a structure type pointer variable as follows:

```
struct book
{
    char name[20]; int pages;
    float price;
};

struct book b, *bptr; bptr=&b;
```

- Here, the base address of *b* is assigned to *bptr* pointer.

# Pointer to Structure...

- Now the members of the structure book can be accessed in 3 ways:

*b.name*

*b.pages*

*b.price*

*bptr->name*

*bptr->pages*

*bptr->price*

*(\*bptr).name*

*(\*bptr).pages*

*(\*bptr).price*

# Pointer to array of structure

- Let we have a structure as follows:

```
struct book
{
    char name[20]; int pages;
    float price;
};

struct book b[10], *bptr;
```

- Then the assignment statement `bptr=b;` assigns the address of the zeroth element of `b` to `bptr`.

# Pointer to array of structure...

- The members of  $b[0]$  can be accessed as:

$bptr->name$        $bptr->pages$        $bptr->price$

- Similarly members of  $b[1]$  can be accessed as:

$(bptr+1)->name$      $(bptr+1)->pages$      $(bptr+1)->price$

- The following *for* statement can be used to print all the values of array of structure  $b$  as:

$for(bptr=b;bptr<b+10;bptr++)$

$printf("%s %d %f", bptr->name, bptr->pages, bptr->price);$

# Problem

- Define a structure of employee having data members name, address, age and salary. Take data for n employee in an array **dynamically** and find the average salary.
- Define a structure of student having data members name, address, marks in C language, and marks in information system. Take data for n students in an array dynamically and find the total marks obtained.

# Function and Structure

- We will consider four cases here:
  - *Passing the individual members to functions*
  - *Passing whole structure to functions*
  - *Passing structure pointer to functions*
  - *Passing array of structure to functions*

## Passing structure member to functions

- Structure members can be passed to functions as actual arguments in function call like ordinary variables.
- Problem: Huge number of structure members
- Example: Let us consider a structure *employee* having members *name*, *id* and *salary* and pass these members to a function:

**display(emp.name,emp.id,emp.salary);**

```
Void display(char e[],int id ,float sal)
{
    printf("\nName\t\tID\t\tSalary\n");
    printf("%s\t%d\t%.2f",e,id,sal);
}
```

# Passing whole structure to functions

- Whole structure can be passed to a function by the syntax:
  - ***function\_name(structure\_variable\_name);***
- The called function has the form:
  - ***return\_type function\_name(struct tag\_name structure\_variable\_name)***
  - {
  - ....;
  - }

display(emp);

```
void display(struct employee e)
{
    printf("\nName\tID\tSalary\n");
    printf("%s\t%d\t%.2f", e.name, e.id, e.salar);
}
```

## Passing structurepointer to functions

- In this case, address of structure variable is passed as an actual argument to a function.
- The corresponding formal argument must be a structure type pointer variable.
- Note: Any changes made to the members in the called function are directly reflected in the calling function.

display(&emp);

```
void display(struct employee*e)
{
    printf("\nName\tID\tSalary\n");
    printf("%s\t%d\t%.2f",e->name,e->id,e->salary);
}
```

# Passing array of structures to function

- Passing an array of structure type to a function is similar to passing an array of any type to a function.
- That is, the name of the array of structure is passed by the calling function which is the base address of the array of structure.
- **Note:** The function prototype comes after the structure definition.

display(emp); //emp is array name of size2

```
void display(struct employee ee[])
{
int i;
printf("\n Name\t\t ID\t\t Salary\n");
for(i=0;i<2;i++)
{
    printf("%s\t\t%d\t\t%.2f\n",ee[i].name,ee[i].id,ee[i].salary);
}
}
```

## Structure using `typedef`

- It allows us to introduce synonyms for data types which could have been declared some other way.
- It is used to give New name to the Structure.
- New name is used for Creating instances, Passing values to function, declaration etc...

## Example:

```
#include<stdio.h>
int main()
{
    typedef int Number;
    Number num1 = 40,num2 = 20;
    Number answer;
    answer = num1 + num2;
    printf("Answer : %d",answer);
    return(0);
}
```

### Output :

Answer : 60

- In the above program we have used `typedef` to create alias name to data type. We have created alias name to ‘int’ data type. We have given new name to integer data type i.e ‘Number’.
- In the second example, **Record** is **tag-name**. ‘**employee**’ is nothing but **New Data Type**. We can now create the variables of type ‘**employee**’ Tag name is optional.

# Different Ways of Declaring Structure using Typedef :

```
typedef struct
{
    char ename[30];
    int ssn;
    int deptno;
}
employee;
```

```
typedef struct
Record
{
    char
    ename[30];
    int ssn;
    int deptno;
```

# Live Example : Using Typedef For Declaring Structure

```
#include<stdio.h>
typedef struct b1
{
    char bname[30];
    int ssn;
    int pages;
}
book;
book b1 = {"Let Us
C",1000,90};
int main()
{
    printf("\nName of Book :s",b1.bname);
    printf("\nSSN of Book : %d",b1.ssn);
    printf("\nPages in Book : %d",b1.pages);
    return(0);
}
```

**OUTPUT:**

Name of Book : Let Us C

SSN of Book : 1000

Pages in Book : 90

## Pointer to Structure Array

- Like we have array of integers, array of pointers etc, we can also have array of structure variables. And to use the array of structure variables efficiently,
- we use **pointers of structure type**. We can also have pointer to a single structure variable, but it is mostly used when we are dealing with array of structure variables.

# Accessing Structure Members with Pointer

```
#include <stdio.h>
struct Book
{ char name[10];
int price; }
int main()
{
struct Book a; //Single structure variable struct Book* ptr; //Pointer of Structure type ptr = &a;
struct Book b[10]; //Array of structure variables struct Book* p; //Pointer of Structure type
p = &b;
return 0;
}
```

# Accessing Structure Members with Pointer

- To access members of structure using the structure variable, we used the dot . operator.
- But when we have a pointer of structure type, we use arrow -> to access structure members.

# Accessing Structure Members with Pointer

```
#include <stdio.h>
struct my_structure{
char name[20];
int number;
int rank;
};
int main()
{
    struct my_structure variable = {"StudyTonight", 35, 1}; struct my_structure *ptr; ptr = &variable;
    printf("NAME: %s\n", ptr->name);
    printf("NUMBER: %d\n", ptr->number);
    printf("RANK: %d", ptr->rank);
    return 0;
}
```

# Bit Manipulation

- Suppose we want to store the gender of the person , then instead of wasting the complete byte we can manipulate single bit. We can consider Single bit as a flag. Gender of Person can be stored as **[M/F]** . By Setting the **flag bit**
- **1** we can set Gender as “**Male**” and “**Female**” by setting bit as 0
- 1. To pack **Several data objects** in the single memory word , Bits are used.
- 2. Flags can be used in order to store the Boolean values ( **T / F** ).
- 3. A method to define a structure of packed information is known as **bit fields**.

## Syntax : Bit Manipulation

```
struct databits
```

```
{  
int b1 : 1;  
int b2 : 1;  
int b3 : 1;  
int b4 : 4;  
int b5 : 9;  
}data1;
```

## Explanation :

- In the above example, we can say that we have allocated specific number of bits to each structure member.
- Integer can store 2 bytes (\*) , so 2 bytes are manipulated in bits and 1 bit is reserved for b1. Similarly b2,b3 will get single bit.
- Similarly we can structure efficiently to store boolean values or smaller values that requires little memory.

# How to access the Individual Bits ?

- data1.b1
- data1.b2
- data1.b3
- data1.b4
- data1.b5

# How to initialize Structure ?

```
struct databits  
{  
--  
---  
}data1 = { 1,1,0,10,234 };
```

**Initialized Result –**

```
data1.b1 = 1  
data1.b2 = 1  
data1.b3 = 0  
data1.b4 = 10  
data1.b5 = 234
```

## Unions

- Unions are quite similar to the structures in C.
- Union is also a derived type as structure.
- Union can be defined in same manner as structures just the keyword used in defining union in union where keyword used in defining structure was struct.

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};
int main( )
{
    union Data data;
    printf( "Memory size occupied by data : %d\n", sizeof(data));
    return 0;
}
```

Instead of union if we put struct keyword the output will be  
Memory size occupied by data : 26

The memory occupied by a union will be large enough to hold the largest member of the union

## OUTPUT

Memory size occupied by data : 20

# Accessing Union Members

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};
int main( )
{
    union Data data;
    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");
    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);
    return 0;
}
```

## OUTPUT

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

The values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

## Pointer to Union

```
#include<stdio.h>
union team
{
    char *name;
    int members;
    char captain[20]; };
int main()
{
    union team t1,*sptr = &t1;
    t1.name = "India";
    printf("\nTeam : %s",(*sptr).name);
    printf("\nTeam : %s",sptr->name);
    return 0;
}
```

### Output :

Team = India  
Team = India

# Dynamic Arrays

- Dynamic arrays are created using pointer variables and memory management functions malloc,calloc and realloc.
- The concept of dynamic arrays is used in creating and manipulating data structures such as linked list, stacks and queues.

What if we don't know how much space we will need ahead of time?

Example:

- ✓ ask user how many numbers to read in
- ✓ read set of numbers in to array (of appropriate size)
- ✓ calculate the average (look at all numbers)
- ✓ calculate the variance (based on the average)

Problem: how big do we make the array??

using static allocation, have to make the array as big as the user might specify  
(might not be big enough)

## Dynamic Memory Allocation

1. Allow the program to allocate some variables (notably arrays), during the program, based on variables in program (dynamically)
2. Previous example: ask the user how many numbers to read, then allocate array of appropriate size
3. Idea: user has routines to request some amount of memory, the user then uses this memory, and returns it when they are done memory allocated in the *Data Heap*

## **Dynamic Memory Allocation (cont.)**

### Memory Management Functions

calloc - routine used to allocate arrays of memory

malloc - routine used to allocate a single block of memory

realloc - routine used to extend the amount of space allocated previously

free - routine used to tell program a piece of memory no longer needed

#### **note:**

✓ memory allocated dynamically does not go away at the end of functions, you  
MUST explicitly free it up

## malloc function

It is a process by which the required memory address is obtained without an explicit declaration.

The required memory space is obtained by using the memory allocation functions like malloc() and calloc().

malloc() function: Used to allocate a single block of memory to store values of specific data types.

# malloc Function

Syntax:

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

ptr : Pointer variable

type : Data type

size : Number of bytes to be allotted

Ex: int \*ptr;

```
ptr= (int *)malloc(20);
```

The allotted space can be used to store 10 int type variables

## Example program for malloc() function:

```
#include<stdio.h>
#include<malloc.h>
#include<conio.h>
void main()
{
    float *fp;
    fp=(float *)malloc(10);
    printf("Enter a float value : ");
    scanf("%f", &fp);
    printf("The address of pointer in memory is : %u", fp);
    printf("The value stored in memory is : %f", *fp);
    getch();
}
```

It is used to allocate memory in multiple blocks of same size during program execution.

### Syntax:

ptr = (type \*)calloc(n,m);

ptr = Pointer variable

type = Data type

n = Number of blocks to be allotted

m = Number of bytes in each block of memory

Ex:

float \*ptr;

ptr=(float \*)calloc(20,4)

## Example program 1 for calloc() function:

```
#include<stdio.h>
#include<calloc.h>
#include<conio.h>
void main()
{
    float *fp;
    fp=(float *)calloc(10,4);
    printf("Enter a float value : ");
    scanf("%f", &fp);
    printf("The address of pointer in memory is : %u", fp);
    printf("The value stored in memory is : %f", *fp);
    getch();
}
```

## Example program 2 for calloc() function:

```
float *nums;  
int N;  
int l;  
  
printf("Read how many numbers:");  
scanf("%d",&N);  
nums = (float *) calloc(N, sizeof(float));  
/* nums is now an array of floats of size N */  
for (l = 0; l < N; l++) {  
    printf("Please enter number %d: ",l+1);  
    scanf("%f",&(nums[l]));  
}  
/* Calculate average, etc. */
```

# Releasing Memory (`free`)

prototype: `void free(void *ptr)`

memory at location pointed to by `ptr` is released (so we could use it again in the future)

program keeps track of each piece of memory allocated by where that memory starts

if we free a piece of memory allocated with `calloc`, the entire array is freed (released)

results are problematic if we pass as address to `free` an address of something that was not allocated dynamically (or has already been freed)

# free Example

```
float *nums;  
int N;  
printf("Read how many numbers:");  
scanf("%d", &N);  
nums = (float *) calloc(N, sizeof(float));  
/* use array nums */  
/* when done with nums: */  
free(nums);  
/* would be an error to say it again - free(nums) */
```

# The Importance of free

```
void problem() {  
    float *nums;  
    int N = 5;  
    nums = (float *) calloc(N, sizeof(float));  
    /* But no call to free with nums */  
} /* problem ends */
```

When function problem called, space for array of size N allocated, when function ends, variable nums goes away, but the space nums points at (the array of size N) does not (allocated on the heap) - furthermore, we have no way to figure out where it is) Problem called *memory leakage*

**realloc()** function: It is used to modify or reallocate the memory space which is previously allocated.

**Syntax:** `ptr=realloc(ptr,size);`

**Ex:** `int *p;`

```
p=(int *)malloc(50);  
-----  
-----  
-----
```

`p=realloc(p,100); //50 bytes is modified as 100 bytes.`

**free()** function: It is used to release the memory space which is allocated using `malloc()` or `calloc()` function

**Syntax:** `free(ptr);`

# realloc

## Example

```
float *nums;  
int l;  
nums = (float *) calloc(5, sizeof(float));  
/* nums is an array of 5 floating point values */  
for (l = 0; l < 5; l++)  
    nums[l] = 2.0 * l;  
/* nums[0]=0.0, nums[1]=2.0, nums[2]=4.0, etc. */  
nums = (float *) realloc(nums, 10 * sizeof(float));  
/* An array of 10 floating point values is allocated, the first 5 floats from the old nums are copied  
as the first 5 floats of the new nums, then the old nums is released */
```

# File Handling in C

## What is a File?

- A *file* is a collection of related data that a computer treats as a single unit.
- Computers store files to secondary storage so that the contents of files remain intact when a computer shuts down.
- When a computer reads a file, it copies the file from the storage device to memory; when it writes to a file, it transfers data from memory to the storage device.
- C uses a structure called `FILE` (defined in `stdio.h`) to store the attributes of a file.

## Steps in Processing a File

1. Create the stream via a pointer variable using the **FILE** structure:

**FILE** \***p**;

2. Open the file, associating the stream name with the file name.
3. Read or write the data.
4. Close the file.

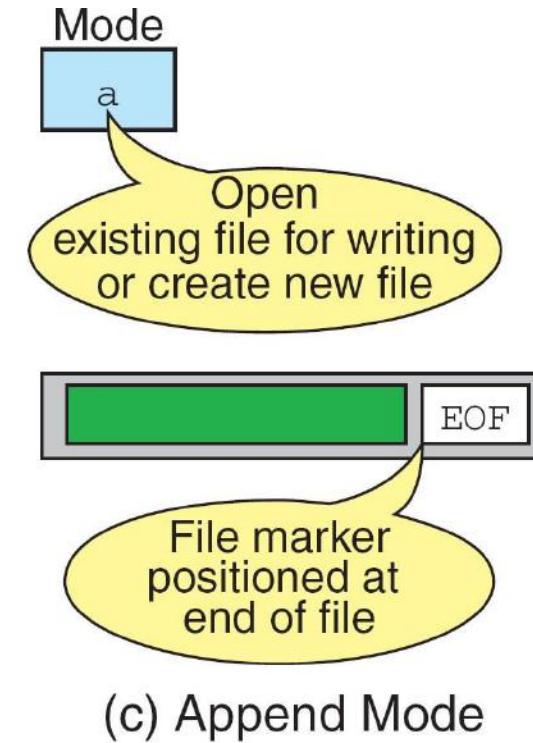
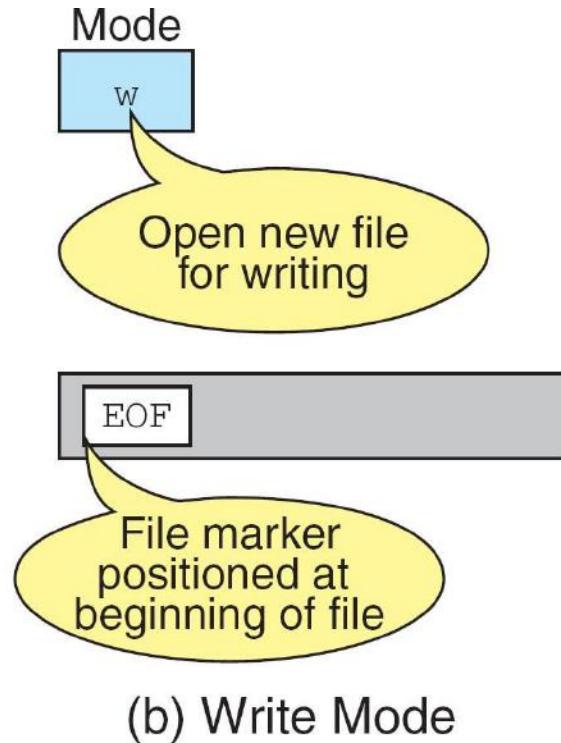
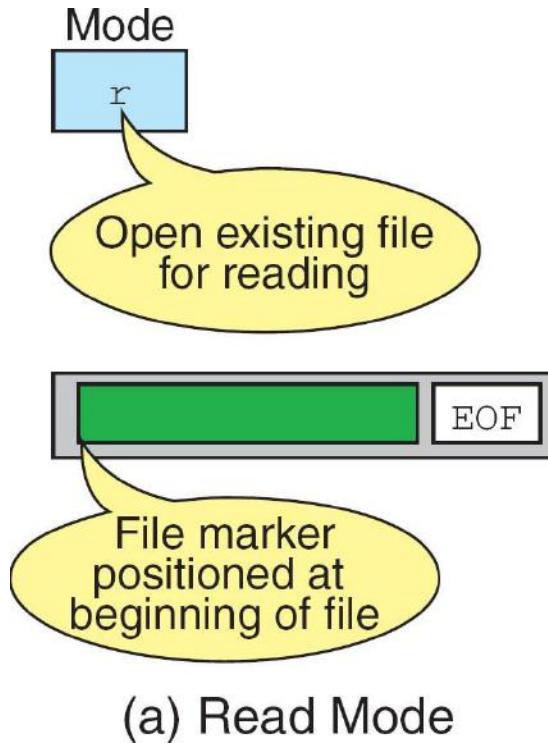
## The basic file operations are

- fopen - open a file- specify how its opened (read/write) and type (binary/text)
- fclose - close an opened file
- fread - read from a file
- fwrite - write to a file
- fseek/fsetpos - move a file pointer to somewhere in a file.
- ftell/fgetpos - tell you where the file pointer is located.

# File Open Modes

Mode	Meaning
r	<p>Open text file in read mode</p> <ul style="list-style-type: none"><li>• If file exists, the marker is positioned at beginning.</li><li>• If file doesn't exist, error returned.</li></ul>
w	<p>Open text file in write mode</p> <ul style="list-style-type: none"><li>• If file exists, it is erased.</li><li>• If file doesn't exist, it is created.</li></ul>
a	<p>Open text file in append mode</p> <ul style="list-style-type: none"><li>• If file exists, the marker is positioned at end.</li><li>• If file doesn't exist, it is created.</li></ul>

# More on File Open Modes



Additionally,

- r+ - open for reading and writing, start at beginning
- w+ - open for reading and writing (overwrite file)
- a+ - open for reading and writing (append if file exists)

# File Open

- The file open function (**fopen**) serves two purposes:
  - It makes the connection between the physical file and the stream.
  - It creates “a program file structure to store the information” C needs to process the file.
- Syntax:  
`filepointer=fopen ("filename", "mode");`

## More On **fopen**

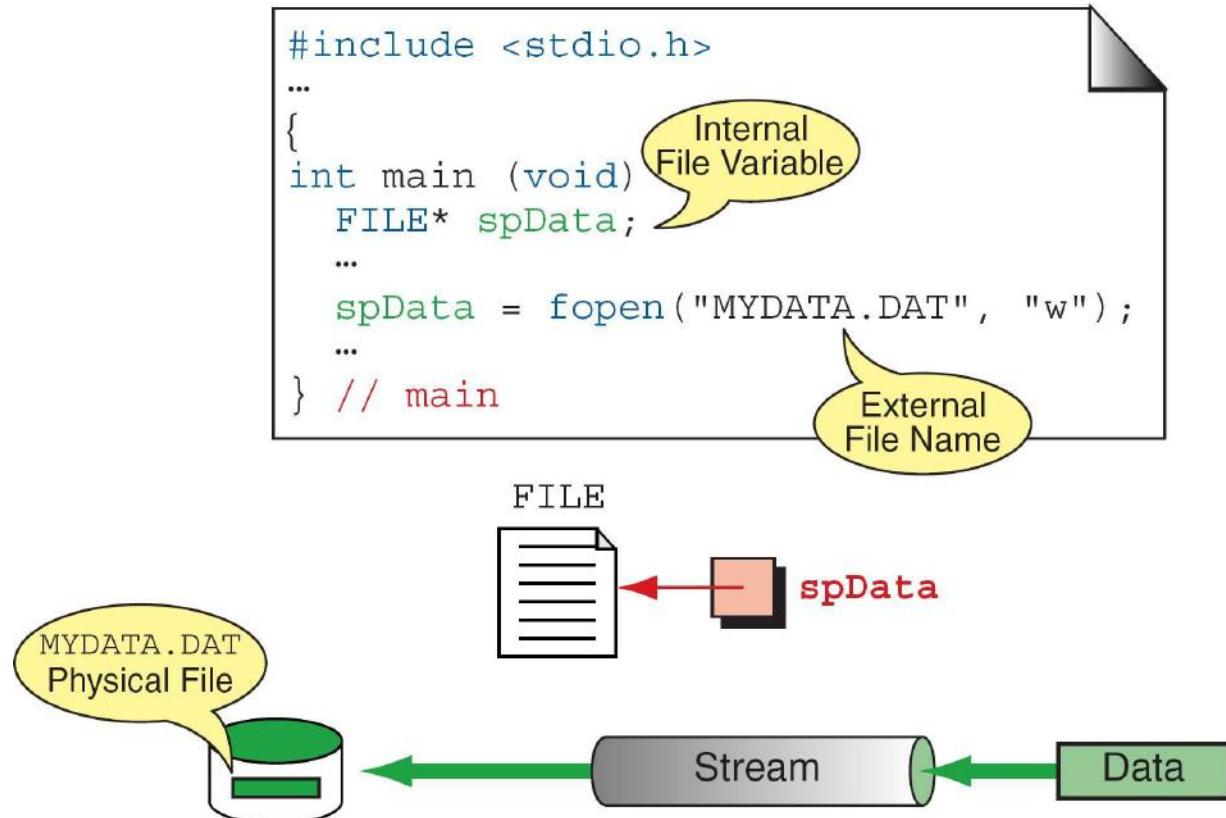
- The file mode tells C how the program will use the file.
- The filename indicates the system name and location for the file.
- We assign the return value of **fopen** to our pointer variable:

```
spData = fopen("MYFILE.TXT", "w");    fopen
```

```
spData = fopen("A:\\MYFILE.TXT", "w");
```

```
spData = fopen("A:\\\\MYFILE.TXT", "w");
```

# More On **fopen**



## Closing a File

- When we finish with a mode, we need to close the file before ending the program or beginning another mode with that same file.
- To close a file, we use `fclose` and the pointer variable:  
`fclose (spData) ;`

# fprintf()

## Syntax:

```
fprintf (fp,"string",variables);
```

## Example:

```
int i = 12;  
float x = 2.356;  
char ch = 's';  
FILE *fp;  
fp=fopen("out.txt","w");  
fprintf (fp, "%d %f %c", i, x, ch);
```

# fscanf()

Syntax:

fscanf (fp,"string",identifiers);

Example:

```
FILE *fp;  
Fp=fopen("input.txt","r");  
int i;  
fscanf (fp,"%d",i);
```

## getc()

Syntax:

~~Syntax~~ identifier = getc (file pointer);

Example:

~~Example~~ FILE \*fp;

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

char ch;

ch = getc (fp);

## putc()

write a single character to the output file, pointed to by fp.

### Example:

```
FILE *fp;  
char ch;  
putc (ch,fp);
```

# End of File

- There are a number of ways to test for the end-of-file condition. Another way is to use the value returned by the *fscanf* function:

```
FILE *fptr1;  
int istatus ;  
fscanf istatus = fscanf (fptr1, "%d", &var) ;  
if ( istatus == feof(fptr1) )  
{  
    printf ("End-of-file encountered.\n") ;  
}
```

# Reading and Writing Files

```
#include <stdio.h>
int main ( )
{
    FILE *outfile, *infile ;
    int b = 5, f ;
    float a = 13.72, c = 6.68, e, g ;
    outfile = fopen ("testdata", "w") ;
    fprintf (outfile, "%f %d %f ", a, b, c) ;
    fclose (outfile) ;
    infile = fopen ("testdata", "r") ;
    fscanf (infile,"%f %d %f", &e, &f, &g) ;
    printf ("%f %d %f \n ", a, b, c) ;
    printf ("%f %d %f \n ", e, f, g) ;
}
```

# Example

```
#include <stdio.h>
#include<conio.h>
void main()
{
    char ch;
    FILE *fp;
    fp=fopen("out.txt","r");
    while(!feof(fp))
    {
        ch=getc(fp);
        printf("\n%c",ch);
    }
    getch();
}
```

# *fread ()*

Declaration:

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Remarks:

fread reads a specified number of equal-sized data items from an input stream into a block.

ptr = Points to a block into which data is read

size = Length of each item read, in bytes

n = Number of items read

stream = file pointer

# *Example*

**Example:**

```
#include <stdio.h>
int main()
{
    FILE *f;
    char buffer[11];
    if (f = fopen("fred.txt", "r"))
    {
        fread(buffer, 1, 10, f);
        buffer[10] = 0;
        fclose(f);
        printf("first 10 characters of the file:\n%s\n", buffer);
    }
    return 0;
}
```

# fwrite()

Declaration:

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE*stream);  
size_t fwrite(const void *ptr size_t size, size_t n, FILE*stream);
```

Remarks:

fwrite appends a specified number of equal-sized data items to an output file.

ptr = Pointer to any object; the data written begins at ptr

size = Length of each item of data

n = Number of data items to be appended

stream = file pointer

# Example

Example:

```
#include <stdio.h>
int main()
{
    char a[10]={'1','2','3','4','5','6','7','8','9','a'};
    FILE *fs;
    fs=fopen("Project.txt","w");
    fwrite(a,1,10,fs);
    fclose(fs);
    return 0;
}
```

# fseek()

This function sets the file position indicator for the stream pointed to by stream or you can say it seeks a specified place within a file and modify it.

<b>SEEK_SET</b>	Seeks from beginning of file
<b>SEEK_CUR</b>	Seeks from current position
<b>SEEK_END</b>	Seeks from end of file

## Example:

```
#include <stdio.h>

int main()
{
    FILE * f;
    f = fopen("myfile.txt", "w");
    fputs("Hello World", f);
    fseek(f, 6, SEEK_SET);      SEEK_CUR,  SEEK_END
    fputs(" India", f);
    fclose(f);
    return 0;
}
```

# ftell()

```
offset = ftell(file pointer);
```

"ftell" returns the current position for input or output on the file

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

```
int main(void)
{
    FILE *stream;
    stream = fopen("MYFILE.TXT", "w");
    fprintf(stream, "This is a test");
    printf("The file pointer is at byte %ld\n", ftell(stream));
    fclose(stream);
    return 0;
}
```

THANK YOU.....