

# **18CSS101J - PROGRAMMING FOR PROBLEM SOLVING**

# **UNIT - 1**

## **SLO-1**

# **Evolution of Programming & Languages**

# Programming Language

- A programming language is a formal language comprising a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms.

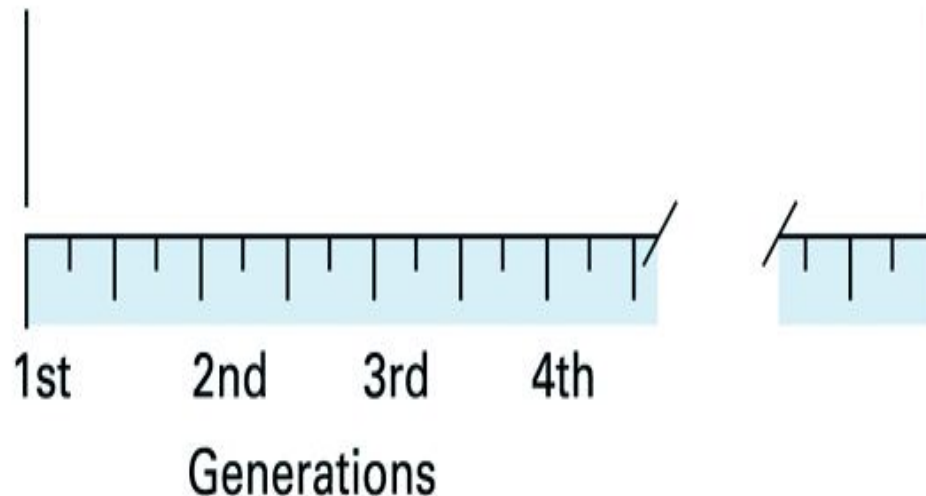
# 1. 1 Evolution of Programming & Languages

- ❑ 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

# “Generations” of Programming Languages

Problems solved in an environment in which the human must conform to the machine's characteristics

Problems solved in an environment in which the machine conforms to the human's characteristics



# **“Generations” of Programming Languages**

- First Generation
  - Machine language
- Second Generation
  - Assembly language
- Third Generation
  - “High-level” languages such as Pascal, C, COBOL, Fortran
- Fourth Generation
  - Scripting languages such as SQL, Applescript, VBScript
- Fifth Generation?
  - Natural language? Automatic code generation? Object-oriented languages?

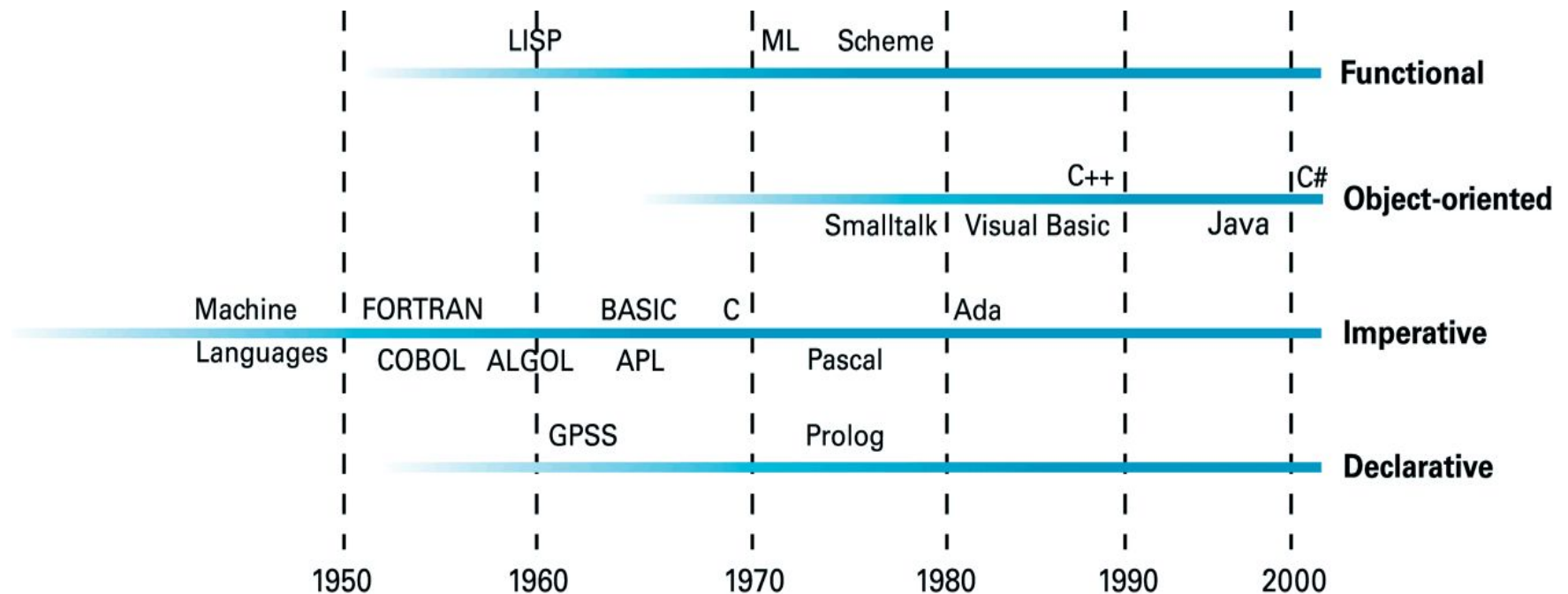
# Four types of programming languages

- Functional
  - Lisp, ML, Scheme
  - Good for evaluating expressions.
- Declarative
  - Prolog
  - Good for making logical inferences.
- Imperative
  - C, Pascal, Fortran, COBOL
  - Good at performing calculations, implementing algorithms.
- Object-oriented
  - C++, Java, C#, Visual Basic
  - Much like imperative languages, but have support for “communication” among objects.



# History of Programming Languages

- An **imperative sentence** is a type of **sentence** that gives instructions or advice, and expresses a command, an order, a direction, or a request..
- An **imperative sentence** may end with an exclamation mark or a period.



# Difference between Compiler

# interpreter

1. It translates the full source code at a time.

1. It translates one line of code at a time.

1. It translates one line of code at a time.

2. Comparatively slower.

3. It uses more memory to perform.

3. The interpreter uses less memory than the compiler to perform.

4. Error detection is difficult for the compiler.

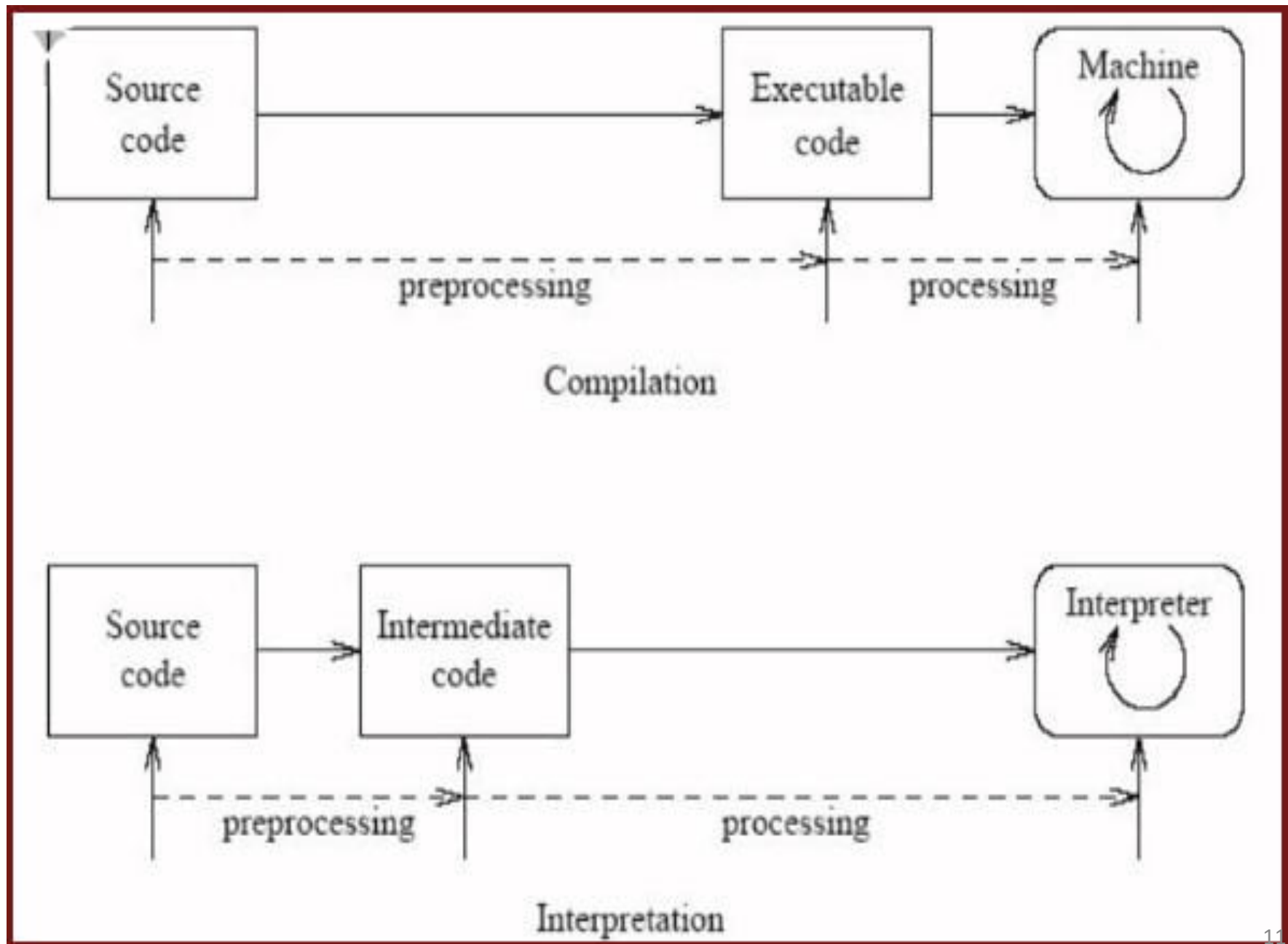
4. Error detection is easier for the interpreter.

5. It shows error alert after scanning the full program.

5. Whenever it finds any error it stops there.

6. Before execution of a program, the compilation is done.

6. Compilation and execution for a program are done simultaneously.



# **SLO - 2**

## **Problem solving through programming**

## 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

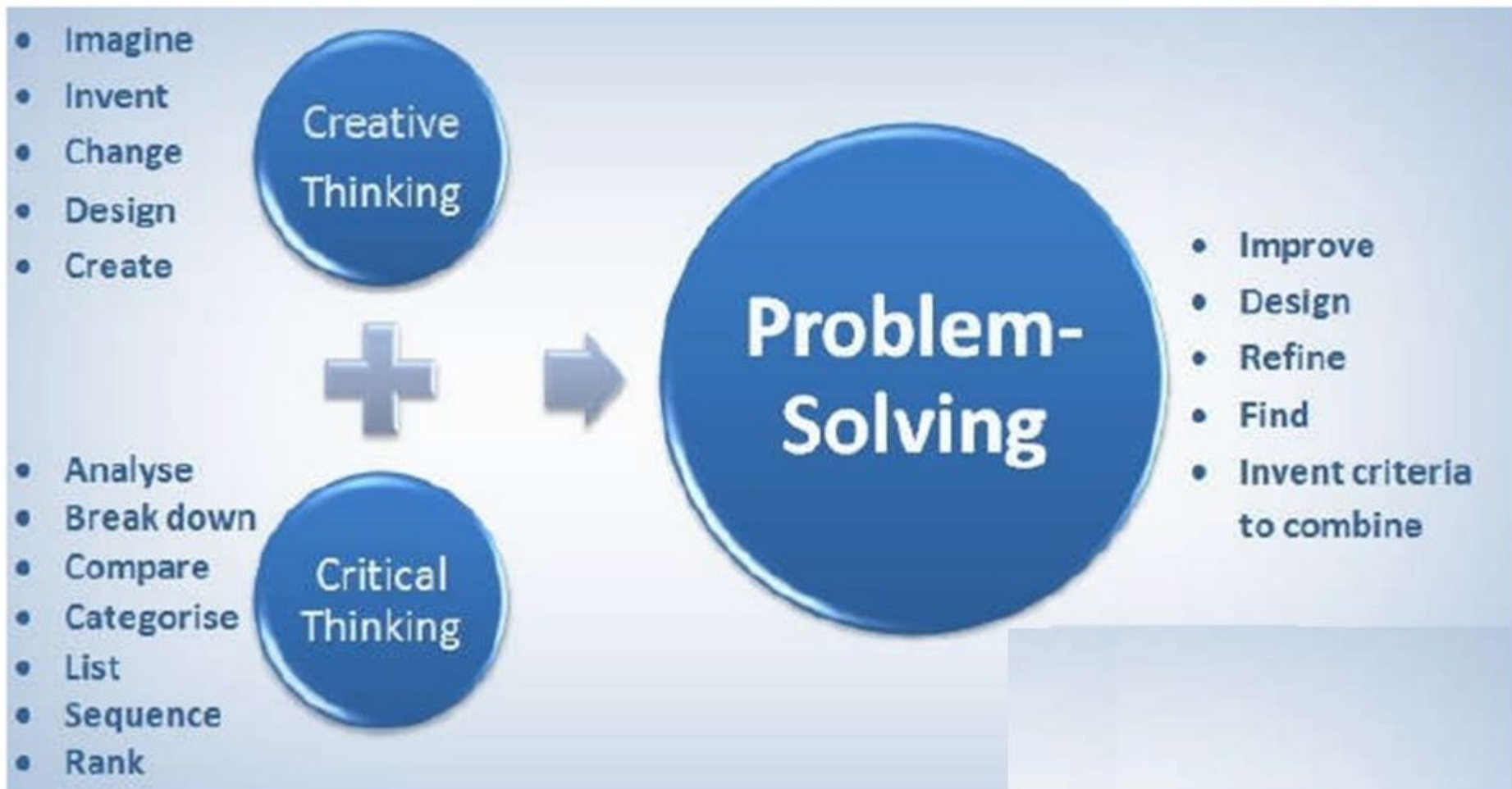
# 1. 2 Problem Solving through Programming Contd...

## *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...





## 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

# **SLO - 1**

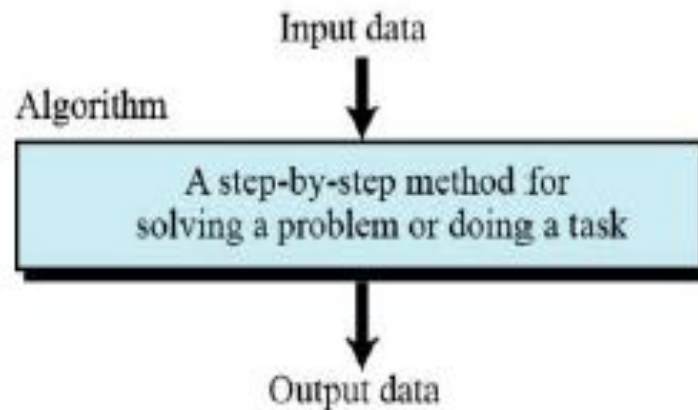
## **Creating Algorithms**

## 1.3 Creating Algorithms

❑ An informal definition of an algorithm is:



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

❑ ***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

### *Example*

#### **Convert Temperature from Fahrenheit (°F) to Celsius (°C)**

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

### *Example*

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

- ❑ **Step 1:** Start
- ❑ **Step 2:** 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.  
$$\text{sum} \leftarrow \text{num1} + \text{num2}$$

## □ 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

# **SLO – 2**

## **Drawing Flowcharts**

## 1. 4 Drawing Flowcharts

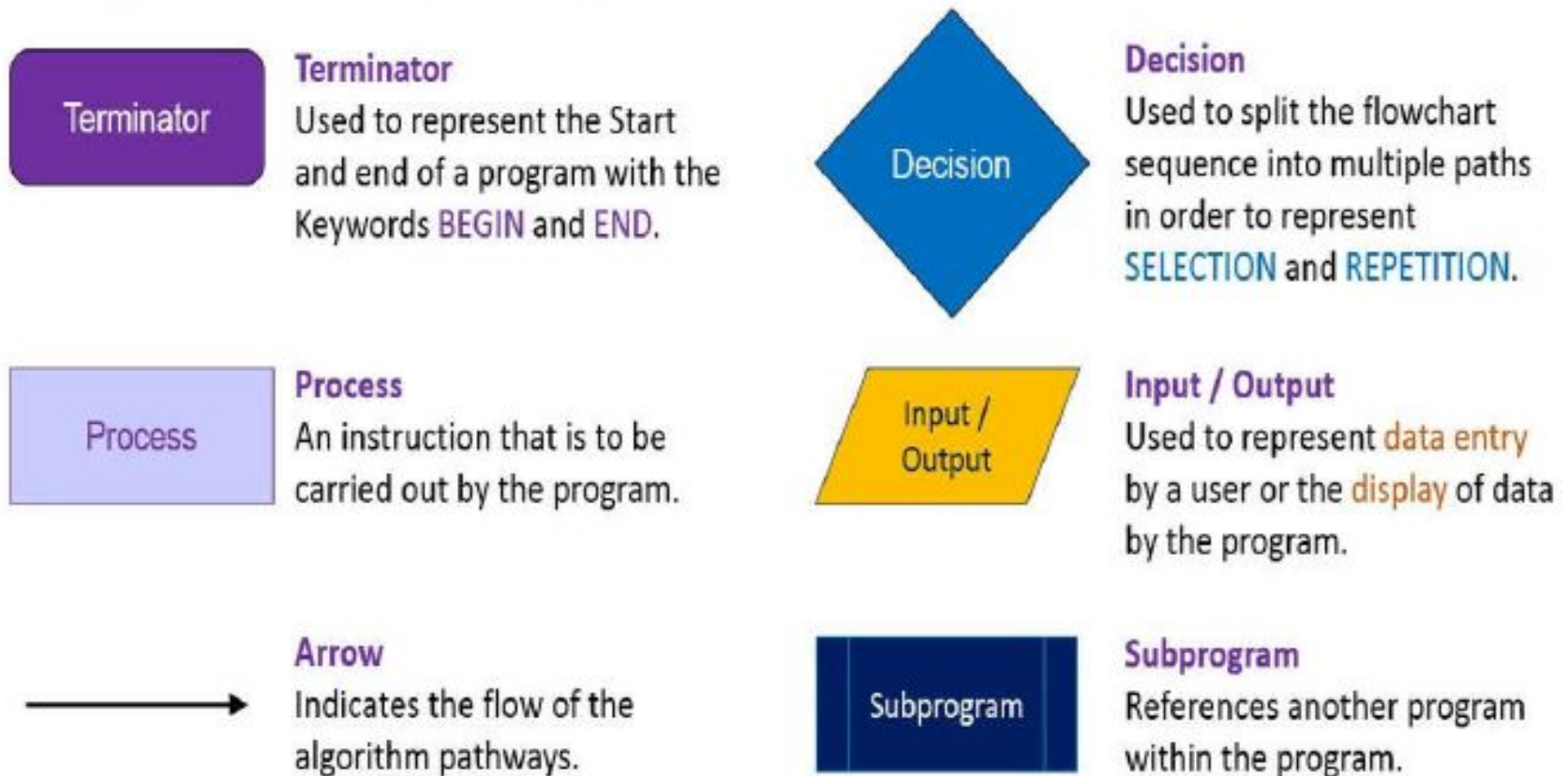
- ❑ Diagrammatic representation
- ❑ Illustrates sequence of operations to be performed
- ❑ Each step represented by a different symbol
  - ❑ Each Symbol contains short description of the Process
- ❑ Symbols linked together by arrows
- ❑ Easy to understand diagrams
- ❑ Clear Documentation
- ❑ Helps clarify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



# 1. 4 Drawing Flowcharts Contd...



## Terminator

Indicates the beginning or end of a program flow in your diagram.



## Process

Indicates any processing function.



## Decision

Indicates a decision point between two or more paths in a flowchart.



## Delay

Indicates a delay in the process.



## Data

Can represents any type of data in a flowchart.



## Document

Indicates data that can be read by people, such as printed output.



## Multiple documents

Indicates multiple documents.



## Subroutine

Indicates a predefined (named) process, such as a subroutine or a module.



## Preparation

Indicates a modification to a process, such as setting a switch or initializing a routine.



## Display

Indicates data that is displayed for people to read, such as data on a monitor or projector screen.



## Manual input

Indicates any operation that is performed manually (by a person).



## Manual loop

Indicates a sequence of commands that will continue to repeat until stopped manually.



## Loop limit

Indicates the start of a loop. Flip the shape vertically to indicate the end of a loop.



## Stored data

Indicates any type of stored data.



## Connector

Indicates an inspection point.



## Off-page connector

Use this shape to create a cross-reference and hyperlink from a process on one page to a process on another page.



## Off-page connector



## Off-page connector



## Off-page connector



## Or

Logical OR



## Summing junction

Logical AND



## Collate

Indicates a step that organizes data into a standard format.



## Sort

Indicates a step that organizes items list sequentially.



## Merge

Indicates a step that combines multiple sets into one.



## Database

Indicates a list of information with a standard structure that allows for searching and sorting.



## Internal storage

Indicates an internal storage device.



## 1. 4 Drawing Flowcharts Contd...

### ❑ Guidelines for Preparing Flowchart

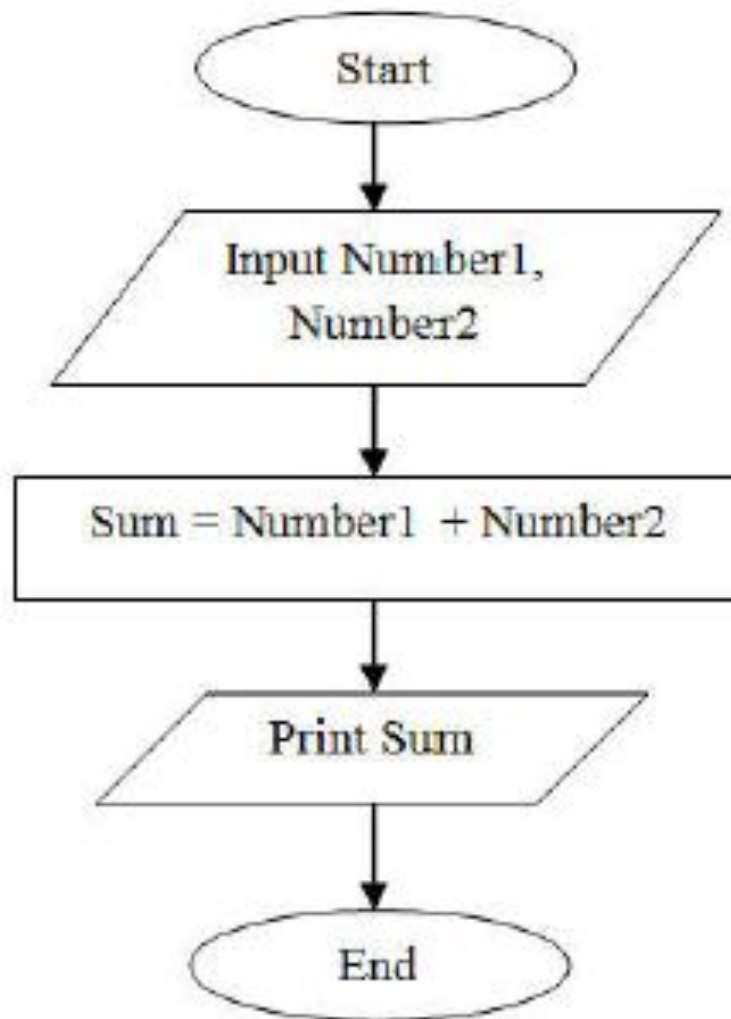
- ❑ Logical order of requirements
- ❑ Ensure that Flowchart has logical *Start* and *Stop*
- ❑ Direction is from Top to bottom
- ❑ Only one flow line is used with Terminal Symbol
- ❑ Only one flow line should come out of a Process symbol
- ❑ Only one flow line should enter a Decision symbol but multiple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

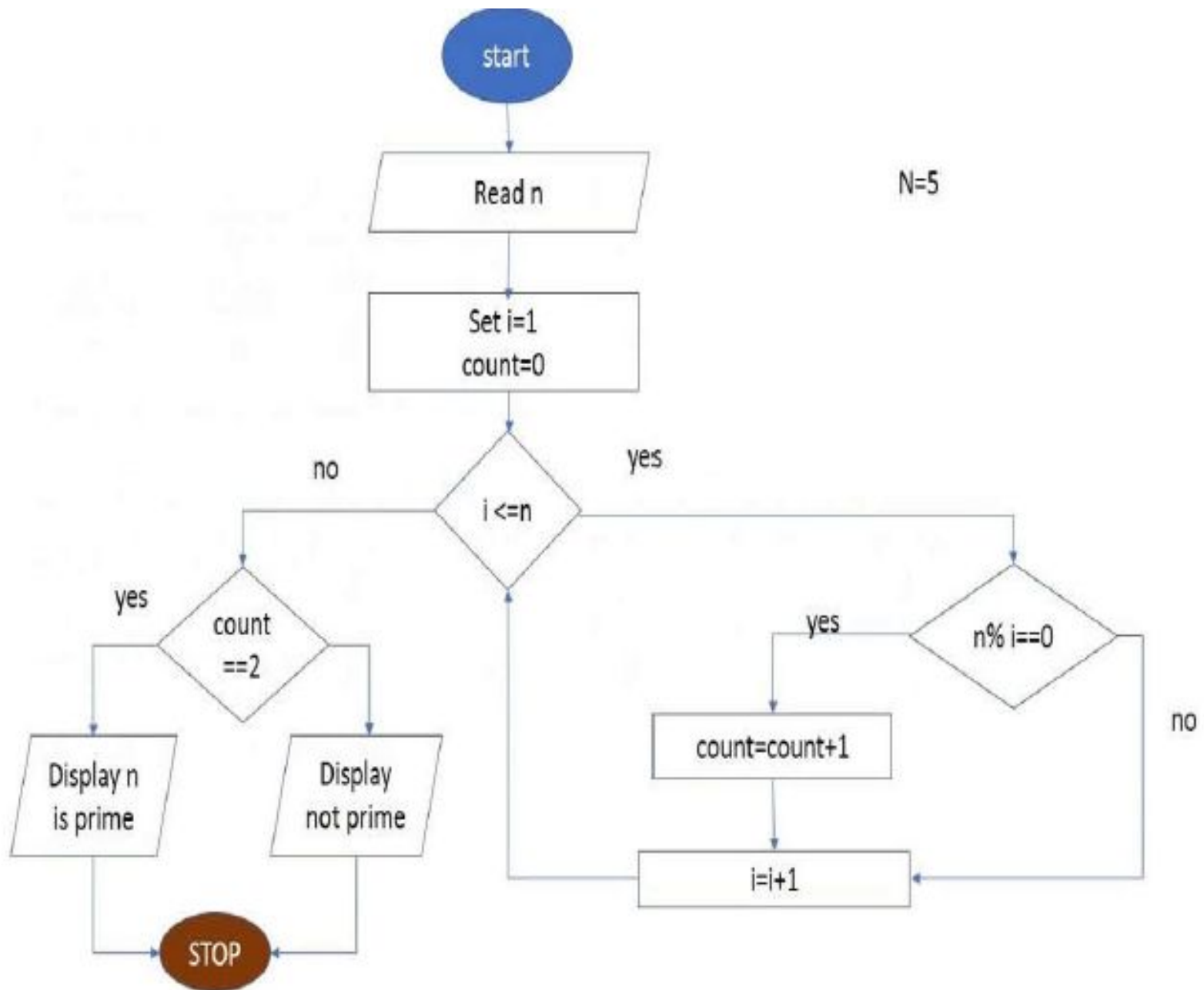
# 1. 4 Drawing Flowcharts Contd...

## ☐ **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...





# **SLO :1**

## **Pseudocode**

# Writing Pseudocode

Pseudo – Imitation / False

Code - Instructions

Outline or a rough draft of the Program.

Simple English syntaxes.

**Goal:** To provide a high level description of the Algorithm

**Benefit:** Enables programmer to concentrate on Algorithm

- Similar to programming code .
- Explanation of the Algorithm .
- No specific Programming language symbolizations .
- Pseudo Code can be transformed into actual program code .

# How to write a Pseudo-code?

## Guidelines of Pseudocode

1. Sequence of operations need to be decided and arranged then should write the pseudocode accordingly.
2. Start with statement that establishes the aim of the Program.

### Example

**The program checks the given number is Odd or Even.**

3. Indent the statement , as it helps to understand the execution mechanism of any decision control. The Pseudocode should ease the readability.

### Example

if “yes”

Print response

“I am in”

if “No”

Print response

“I am not in”

# How to write a Pseudo-code...

4. Use proper Naming Conventions.
5. Use suitable sentence casings such i) CamelCase –Methods.  
ii) Uppercase-Constants  
iii) Lowercase-Variables.
6. Don't be abstract with the pseudocode. Make the pseudocode elaborate and explanatory.
7. Use Structured programming such as i) if-then ii) for iii) while iv) case
8. Make sure the pseudocode is very clear, understandable, finite and comprehend.
9. The pseudocode should be understandable even for a layman or client. Don't make too much of technical complex terms.



# **Do's and Don'ts of Pesudocode**

## **Do's**

1. Use Control Structures.
2. Use appropriate naming conventions.
3. Indentation and white spaces are the key.
4. Keep it simple

## **Don'ts**

1. Don't be too generalized.
2. Don't make the pseudocode abstract.

# Advantages

- The Pseudocode can be easily converted into programming.
- Easy to understand for even non-programmers
- Syntax errors are not so important.
- Changes in the requirement can be easily incorporated in the pseudocode.
- Implements Structured concepts
- No special symbols
- No specific platform required, it can be typed in MS Word.

# Disadvantage

- No standard is followed
- Compilation and Execution of the program cannot be done.
- It may be time consuming to produce result.

# Examples of Pseduocode

**1. Write a Pesudocode to add 2 numbers together and then display the result**

Line1: Start Program

Line 2: Enter two numbers, A, B

Line 3: Add the numbers together

Line 4: Print Sum

Line 5: End Program

**2. Write a Pesudocode to compute the area of a rectangle:**

Line1 : Get the length, l, and width, w

Line2 : Compute the area =  $l * w$

Line3: Display the area

# Examples of Pseduocode Continues....

## 3. Compute the perimeter of a rectangle:

Line1 : Enter length, l

Line2 : Enter width, w

Line3 : Compute Perimeter =  $2 * l + 2 * w$

Line4 : Display Perimeter of a rectangle

# Examples of Pseduocode Continues....

## 4.Input examination marks and award grades

Use Variables: mark of type integer

If mark  $\geq 80$  DISPLAY "Distinction"

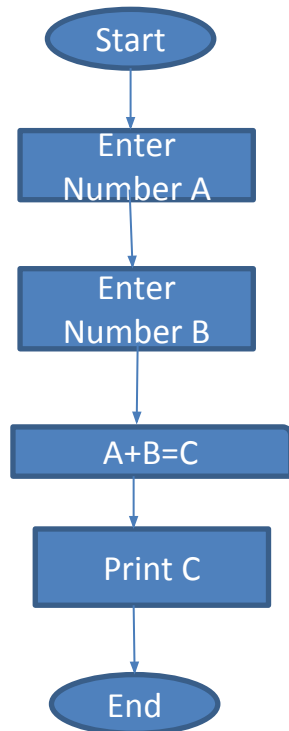
If mark  $\geq 60$  and mark  $< 80$  DISPLAY "First Class"

If mark  $\geq 50$  and mark  $< 60$  DISPLAY "Second Class"

If mark  $< 50$  DISPLAY "Fail"

End Program

# Flowchart to add 2 numbers and display the result.



- Line1: Start Program
- Line 2: Enter two numbers, A, B
- Line 3: Add the numbers together
- Line 4: Print C
- Line 5: End Program

## **SLO - 2**

# **Evolution of C Language, its usage Histroy**



# History and Evolution of C

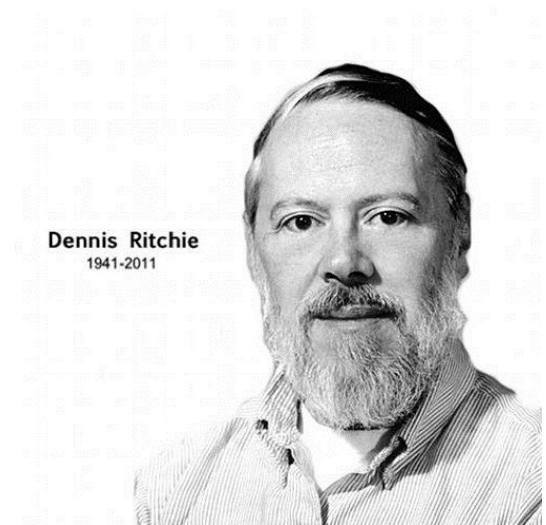
- C evolved from two previous languages, BCPL (Basic Combined Programming Language) and B.
- BCPL developed in 1967 by Martin Richards as a language for writing OSes and compilers.
- Ken Thompson modeled many features in his language, B, after their counterparts in BCPL, and used B to create an early versions of UNIX operating system at bell Laboratories in 1970 on a DEC PDP-7 computer.
- Both BCPL and B were typeless languages: the only data type is machine word and access to other kinds of objects is by special operators or function calls.
- The C language developed from B by Dennis Ritchie at Bell Laboratories and was originally implemented on a DEC PDP-11 computer in 1972.
- It was named C for new language (after B).
- Initially, C used widely as the development language of the UNIX OS.
- Today, almost all new major OS are written in C including Windows.

# C STANDARDS

- The rapid expansion of C over various types of computers led to many variations - similar but incompatible.
- Need to be standardized. In 1983, the X3J11 technical committee was created under the American National Standards Institute (ANSI) Committee on Computer and Information Processing (X3) to provide an unambiguous and machine-independent definition of the language and approved in 1989, called ANSI C.
- Then, the document is referred to as ANSI/ISO 9899:1990.
- The second edition of Kernighan and Ritchie, published in 1988, this version called ANSI C, then used worldwide.
- The more general ANSI then adopted by ISO/IEC, known as ISO/IEC C.
- Historically, from ISO/IEC, C programming language evolved from C89/C90/C95, C99 and the latest is C11.

# History and Evolution of C

1. C – General Purpose Programming Language
2. Developed by Dennis Ritchie in 1972
3. Developed at Bell Laboratories
4. Developed to overcome the problems of previous languages such as B, BCPL, etc.
5. Structured Programming Language
6. C Program
  - ☐ Collection of Functions
  - ☐ Supported by C library



# Father of C Programming : Dennis Ritchie

<b>Born on</b>	<b>September 09,1941</b>
Born In	Bronxville ,New York
Full name	Dennis MacAlistair Ritchie
Nick name	DMR
Nationality	American
Graduate From	Harvard University
Graduate In	Physics and Applied Mathematics
Dead On	October 12 2011

# Evolution of C

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

# Features of C Language

Features of C are

1. Simple
2. Machine Independent.
3. Mid-level programming language
4. Structured Programming language
5. Rich Library
6. Memory management
7. Speed
8. Pointer
9. Recursion
10. Extensible.

# Advantages of C

1. Compiler based Language
2. Programming – Easy & Fast
3. Powerful and Efficient
4. Portable
5. Supports Graphics
6. Supports large number of Operators
7. Used to Implement Data structures

# Disadvantages of C

1. Not a strongly typed Language
2. Use of Same operator for multiple purposes
3. Not Object Oriented

## Typed languages

- **Untyped languages**, also known as dynamically **typed languages**, are programming **languages** that do not make you define the **type** of a variable. JavaScript is **untyped language**.
- This means that a JavaScript variable can hold a value **of any data type**. To declare variables in JavaScript, you need to use the **var** keyword.

## Typeless languages

- Data types are **particular types** of data such as string that can contain text, or a Boolean that can only contain true/false values



# **SLO - 1**

## **Input and Output Functions : printf, scanf**

# Input and Output Statements

1. **Input**- To provide the data for the program
2. **Output**-To display the data on the screen
3. C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.
4. The Standard input-output file – `stdio.h`.
5. The input statement is called `scanf()`.
6. The output statement is called `printf()`.

# Input and Output

## Basic Input & Output Functions

**printf**

**scanf**

**getchar**

**putchar**

# printf

- It is a function in the standard library <stdio.h>
- It converts internal values to characters

**The syntax is :**

**printf( format string, arg1, arg2,...)**

- The format string contains, within double quotes, formatting information for the arguments to be printed viz., arg1, arg2,..
- There is one group of characters for each argument.
- Each such group contains a percentage sign %, followed by a conversion character

## printf: cont'd.

For example,

```
printf("%d\n", number);
```

- In this case, we had only one argument `number`, and its formatting information is given by `%d`, indicating that `number` is a decimal integer.
- The `\n` indicates that after printing `number` we should move to a newline.

# Structure of a C Program

Preprocessor Directives

Global Declarations

```
int main ( void )
```

```
{
```

Local Declarations

Statements

```
} // main
```

Other functions as required.

# Main Function

- Every program has one function **main**
- Header for main: **return type main ()** or **void main()**
- Program is the sequence of statements between the **curly brackets { }** following main
- Statements are executed one at a time from the one immediately following to main to the one before the }

# printf: examples

(dipsrm.c)

```
#include <stdio.h>
main()
{
    char univer_name[20] = "SRM UNIVERSITY";
    int x = 123;
    float y = 123.45;
    printf("%d %f %s", x, y, univer_name);
}
```

Output:

```
123 123.449997 SRM UNIVERSITY_
```



# printf: examples

**format1.c**

```
#include <stdio.h>
main()
{
int i = 12345, j = -13579, k = -24680;
long ix = 123456789;
short sx = -2222;
unsigned ux = 5555;
float a = 32.2, b = -6.285;
    printf("%d %d %d %ld %d %u\n", i, j, k, ix, sx, ux);
    printf(" i = %3d j = %3d %3d \n", i, j, k);
    printf(" $%4.2f %7.2%f\n", a, b);
}
```

Output:

```
12345 -13579 -24680 123456789 -2222 5555
 i = 12345 j = -13579 -24680
 $32.20    -6.28%
```

## **printf: cont'd.**

summarize the formatting specifications :

- Each conversion specification begins with a % and ends with a conversion character.
- Between the % and the conversion character there may be, in order:
  - A minus sign, which specifies left adjustment of the converted argument
  - A number that specifies the minimum field width  
A period, which separates the field width from the precision
  - A number that specifies the precision  
An u if the integer is to be printed as short or l if as long

# scanf

- This is a function in the standard library *<stdio.h>*
- It reads characters from standard input
- **The syntax is:**

*scanf( format string, arg1, arg2,...)*

- The format string contains, within double quotes, formatting information for the way the arguments are to be stored
- It controls how arguments are stored
- There is one group of characters for each argument
- Each such group contains a percentage sign %, followed by a conversion character

## scanf: cont'd.

- Note that each argument is an address, indicating where in the memory the corresponding argument is to be stored

For example,

```
scanf("%d\n", &n);
```

- In this case we had only one argument **&n**, and its formatting information is given by **%d**, indicating that **n** is to be stored as a decimal integer.
- Once again the **\n** indicates that after reading **n** we should move to a newline.

# scanf: examples

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

```
main()
{
    char univer_name[20];
    int x ;
    float y ;

    scanf("%d %f %s", &x, &y, univer_name);
}
```

## multiple word string input through scanf( )

```
char name[50];  
printf("Enter your full name: ");  
scanf("%[^\\n]s",name);
```

Here `[^\\n` indicates that `scanf( )` will keep receiving characters into `name[ ]` until a `\\n` is encountered.

**Note :** *The caret (^) sign in math means exponent.*

*Example :  $x^2$*

## **scanf: examples cont'd**

The following data it can be entered during execution

12345 0.05 SRMIST

or even one per line as:

12345

0.05

SRMIST

Note that we have a different conversion character for each of the three different argument types

## scanf: examples cont'd.

If we input the following data from the terminal  
10 256.875 T

After executing the following program

```
#include <stdio.h>
main()
{
    char c;
    int i;
    float x;
    scanf("%3d %5f %c", &i, &x, &c);
}
```

10 will be assigned to i, **256.8** to x and the character **T** to c.



# Functions *getchar* and *putchar*

- When we need to read or write data character by character, functions *getchar* and *putchar* are very handy.
- To read a single character in variable *c*, we give:

*c = getchar();*  
(equivalent to *scanf("%c", &c);*)

- To write the character in variable *c*, we give:

*putchar(c);*  
(equivalent to *printf("%c", c);*)

# **SLO : 2**

## **Variables, Identifiers**

# IDENTIFIERS

- Identifiers are names given to program elements such as variables, arrays and functions.

## Rules for forming identifier name:

- it cannot include any special characters or punctuation marks (like #, \$, ^, ?, .., etc) except the underscore "\_".
- There cannot be two successive underscores

## **Rules for Identifiers**

1. First character must be alphabetic character or underscore.
2. Must consist only of alphabetic characters, digits, or underscores.
3. First 63 characters of an identifier are significant.
4. Cannot duplicate a keyword.

# VARIABLES

- By default, C automatically a numeric variable signed.
- Character variables can include any letter from the alphabet or from the ASCII chart and numbers 0 – 9 that are put between single quotes.

# Variables

- *Variables are named memory locations that have a type, such as integer or character, which is inherited from their type.*
- *The type determines the values that a variable may contain and the operations that may be used with its values.*

**Variable Declaration**

**Variable Initialization**

# Variables

Variable's  
type

Variable's  
identifier

```
char code;  
int i;  
long long national_debt;  
float payRate;  
double pi;
```

Program

## Examples of Variable Declarations and Definitions

```
bool    fact;
short   maxItems;           // Word separator: Capital
long    long national_debt; // Word separator: underscore
float   payRate;           // Word separator: Capital
double  tax;
float    complex voltage;
char     code, kind;        // Poor style—see text
int      a, b;              // Poor style—see text
```



# Variable Initialization

```
char code = 'B';  
int i = 14;  
long long natl_debt = 10000000000000;  
float payRate = 14.25;  
double pi = 3.1415926536;
```

Program

B code  
14 i  
10000000000000 natl\_debt  
14.25 payRate  
3.1415926536 pi

Memory

## **Note**

**When a variable is defined, it is not initialized.  
We must initialize any variable requiring  
prescribed data when the function starts.**

# Print Sum of Three Numbers

```
1  /* This program calculates and prints the sum of
2     three numbers input by the user at the keyboard.
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7
8  int main (void)
9  {
10 // Local Declarations
11     int a;
12     int b;
13     int c;
14     int sum;
15
```

## Print Sum of Three Numbers (continued)

```
16 // Statements
17 printf("\nWelcome. This program adds\n");
18 printf("three numbers. Enter three numbers\n");
19 printf("in the form: nnn nnn nnn <return>\n");
20 scanf("%d %d %d", &a, &b, &c);
21
22 // Numbers are now in a, b, and c. Add them.
23 sum = a + b + c;
24
25 printf("The total is: %d\n\n", sum);
26
27 printf("Thank you. Have a good day.\n");
28 return 0;
29 } // main
```

## Print Sum of Three Numbers (continued)

Results:

```
Welcome. This program adds  
three numbers. Enter three numbers  
in the form: nnn nnn nnn <return>  
11 22 33
```

```
The total is: 66
```

```
Thank you. Have a good day.
```

# Example

```
int emp_num;  
float salary;  
char grade;  
double balance_amount;  
unsigned short int acc_no;
```

## Examples of Valid and Invalid Names

Valid Names		Invalid Name	
a	// Valid but poor style	\$sum	// \$ is illegal
student_name		2names	// First char digit
_aSystemName		sum-salary	// Contains hyphen
_Bool	// Boolean System id	stdnt Nmbr	// Contains spaces
INT_MIN	// System Defined Value	int	// Keyword

# **SLO - 1**

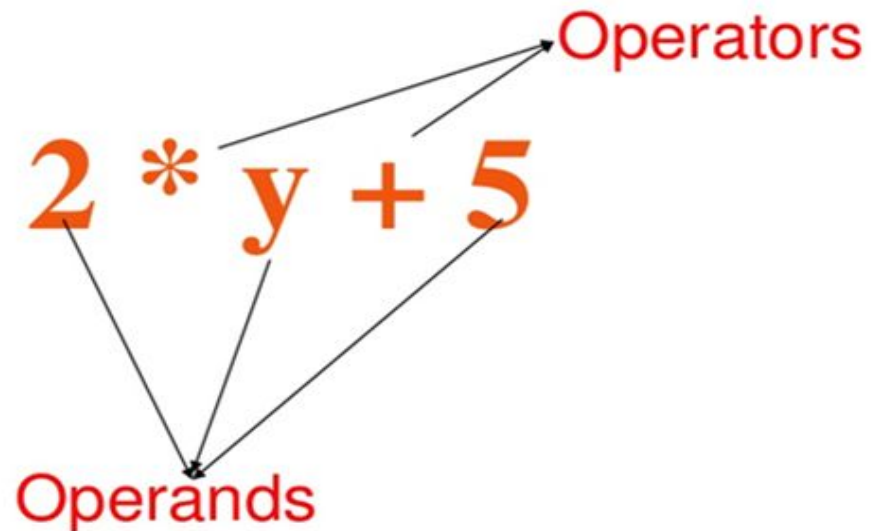
## **Expressions**



# Expressions

- In programming, an **expression** is any legal combination of symbols that represents a value.
- For example, in the C language  $2*y+5$  is a legal **expression**.
- Every **expression** consists of at least one operand and can have one or more operators

~~•  $X=3+/(R+M))$~~



# EXPRESSION

- linked to each other by the use of operators to compute a value. An operand can be a function reference, a variable, an array element or a constant.

## Example:

- $a-b$ ;
- In the above expression, minus character (-) is an operator, and a, and b are the two operands.

- ❑ **There are four types of expressions exist in C:**
  - Arithmetic expressions Ex:  $12/8 + 8 * 2$
  - Relational expressions Ex:  $a != b$
  - Logical expressions Ex:  $x > 10 \parallel y < 11$
  - Conditional expressions  
Syntax:  $exp1 ? exp2 : exp3$
- ❑ Each type of expression takes certain types of operands and uses a specific set of operators. Evaluation of a particular expression produces a specific value.

# Arithmetic Expressions

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

# Arithmetic Expressions

Algebraic expression	C expression
$\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$	<code>area=sqrt(s*(s-a)*(s-b)*(s-c))</code>
$\text{Sin} \left( \frac{b}{\sqrt{a^2 + b^2}} \right)$	<code>sin(b/sqrt(a*a+b*b))</code>
$\tau_1 = \sqrt{\left\{ \frac{\sigma_x - \sigma_y}{2} \right\}^2 + \tau_{xy}^2}$	<code>tow1=sqrt((rowx-rowy)/2+tow*x*y*y)</code>
$\tau_1 = \sqrt{\left\{ \frac{\sigma_x - \sigma_y}{2} \right\}^2 + \tau_{xy}^2}$	<code>tow1=sqrt(pow((rowx-rowy)/2,2)+tow*x*y*y)</code>
$y = \frac{\alpha + \beta}{\sin \theta} +  x $	<code>y=(alpha+beta)/sin(theta*3.1416/180)+abs(x)</code>

# Precedence of operators

## BODMAS RULE-

- Brackets Of Division Multiplication Addition Subtraction
  - Brackets will have the highest precedence and have to be evaluated first,
  - then comes of , then comes division, multiplication, addition and
  - finally subtraction.
- C language uses some rules in evaluating the expressions and they are called as precedence rules or sometimes also referred to as hierarchy of operations, with some operators with highest precedence and some with least.
- The 2 distinct priority levels of arithmetic operators in c are-
- Highest priority : \* / %
- Lowest priority : + -

# **Rules for evaluation of expression**

1. First parenthesized sub expression from left to right are evaluated.
2. If parentheses are nested, the evaluation begins with the innermost sub expression
3. The precedence rule is applied in determining the order of application of operators in evaluating sub expressions
4. The associatively rule is applied when 2 or more operators of the same precedence level appear in a sub expression.
5. Arithmetic expressions are evaluated from left to right using the rules of precedence
6. When parentheses are used, the expressions within parentheses assume highest priority

# Hierarchy of operators

Operator	Description	Associativity
( ), [ ]	Function call, array element reference	Left to Right
+, -, ++, -- ,!, ~, *, &	Unary plus, minus, increment, decrement, logical negation, 1's complement, pointer reference, address	Right to Left
*, / , %	Multiplication, division, modulus	Left to Right



# Example 1

Evaluate  $x1 = (-b + \sqrt{b*b - 4*a*c}) / (2*a)$   
@  $a=1, b=-5, c=6$

$$\begin{aligned} &= -(-5) + \sqrt{(-5)(-5) - 4*1*6} / (2*1) \\ &= (5 + \sqrt{(-5)(-5) - 4*1*6}) / (2*1) \\ &= (5 + \sqrt{25 - 4*1*6}) / (2*1) \\ &= (5 + \sqrt{25 - 4*6}) / (2*1) \\ &= (5 + \sqrt{25 - 24}) / (2*1) \\ &= (5 + \sqrt{1}) / (2*1) \\ &= (5 + 1.0) / (2*1) \\ &= (6.0) / (2*1) \\ &= 6.0 / 2 \\ &= 3.0 \end{aligned}$$

## Example 2

Evaluate the expression when  $a=4$

$$b = a - ++a$$

$$= a - 5$$

$$= 5 - 5$$

$$= 0$$

# Order of evaluation

Highest () [] ->

! ~ ++ -- (type \*) & sizeof

/ %

+ -

<< >>

< <= > >=

== !=

&

^

|

&&

||

? :

= += -= \*= /=

Lowest ,

## **SLO - 2**

# **Single line and Multiline comments**

# USING COMMENTS

- It is a good programming practice to place some comments in the code to help the reader understand the code clearly.
- Comments are just a way of explaining what a program does. It is merely an internal program documentation.
- The compiler ignores the comments when forming the object file. This means that the comments are non-executable statements.

# Continued....

## C supports two types of commenting:

- `//` is used to comment a **single statement**. This is known as a line comment. A line comment can be placed anywhere on the line and it does not require to be specifically ended as the end of the line automatically ends the line.
- `/*` is used to comment **multiple statements / multiline**. A `/*` is ended with `*/` and all statements that lie within these characters are commented.

# EXAMPLE

// This is my first program in C

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

```
int main()
```

```
{
```

```
    printf("\n Welcome to the world of C ");
```

```
        /* the above statement prints the  
           statements given in double quotes*/
```

```
    return 0;
```

```
}
```

# **SLO - 1**

## **Constants, Keywords**



## Define : Constants

- C Constants are also like normal variables. But, only difference is, their values **can not be modified** by the program once they are defined.
- Constants refer to fixed values. They are also called as literals
- Constants may be belonging to any of the data type.

### Syntax:

```
const data_type variable_name; (or)  
const data_type *variable_name;
```

# **TYPES OF C CONSTANT**

1. Integer constants
2. Real or Floating point constants
3. Octal & Hexadecimal constants
4. Character constants
5. String constants
6. Backslash character constants

<b>Constant type</b>	<b>data type (Example)</b>
<b>Integer constants</b>	<b>int (53, 762, -478 etc )</b> <b>unsigned int (5000u, 1000U etc)</b> <b>long int, long long int</b> <b>(483,647 2,147,483,680)</b>
<b>Real or Floating point constants</b>	<b>float (10.456789)</b> <b>double (600.123456789)</b>
<b>Octal constant</b>	<b>int (Example: 013 /*starts with 0 */)</b>
<b>Hexadecimal constant</b>	<b>int (Example: 0x90 /*starts with 0x*/)</b>
<b>character constants</b>	<b>char (Example: 'A', 'B', 'C')</b>
<b>string constants</b>	<b>char (Example: "ABCD", "Hai")</b>

# **RULES FOR CONSTRUCTING C CONSTANT**

## **1. INTEGER CONSTANTS IN C:**

- An integer constant must have at least one digit.
- It must not have a decimal point.
- It can either be positive or negative.
- No commas or blanks are allowed within an integer constant.
- If no sign precedes an integer constant, it is assumed to be positive.
- The allowable range for integer constants is -32768 to 32767.

## **2. REAL CONSTANTS IN C:**

- A real constant must have at least one digit
- It must have a decimal point
- It could be either positive or negative
- If no sign precedes an integer constant, it is assumed to be positive.
- No commas or blanks are allowed within a real constant.

# RULES FOR CONSTRUCTING C CONSTANT

## 3. CHARACTER AND STRING CONSTANTS IN C:

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single quotes.
- The maximum length of a character constant is 1 character.
- String constants are enclosed within double quotes.

## 4. BACKSLASH CHARACTER CONSTANTS IN C:

- There are some characters which have special meaning in C language.
- They should be preceded by backslash symbol to make use of special function of them.

## Given below is the list of special characters and their purpose

Backslash_character	Meaning
<b>\b</b>	Backspace
<b>\f</b>	Form feed
<b>\n</b>	New line
<b>\r</b>	Carriage return
<b>\t</b>	Horizontal tab
<b>\”</b>	Double quote
<b>\’</b>	Single quote
<b>\\</b>	Backslash
<b>\v</b>	Vertical tab
<b>\a</b>	Alert or bell
<b>\?</b>	Question mark
<b>\N</b>	Octal constant (N is an octal constant)
<b>\XN</b>	Hexadecimal constant (N – hex.dcm1 cnst)

# HOW TO USE CONSTANTS IN A C PROGRAM?

- We can define constants in a C program in the following ways.
1. By “const” keyword
  2. By “#define” preprocessor directive

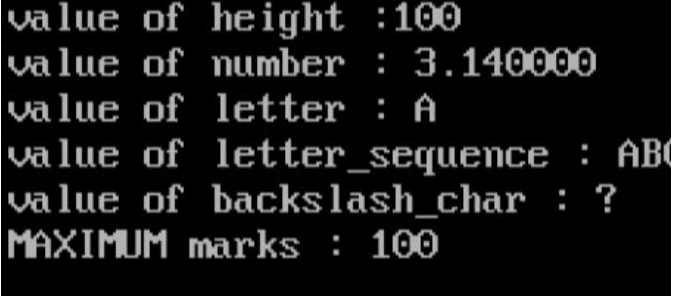
Please note that when you try to change constant values after defining in C program, it will through error.

## EXAMPLE PROGRAM USING CONST KEYWORD AND #DEFINE PREPROCESSOR DIRECTIVE IN C

```
#include <stdio.h>
# define Max_Marks 100
Int m=20;
void main()
{
const int height = 100; /*int constant*/
const float number = 3.14; /*Real constant*/
const char letter = 'A'; /*char constant*/
const char letter_sequence[10] = "ABC"; /*string constant*/
const char backslash_char = '\\'; /*special char cnst*/
clrscr();
printf("value of height :%d \n", height );
printf("value of number : %f \n", number );
printf("value of letter : %c \n", letter );
printf("value of letter_sequence : %s \n", letter_sequence);
printf("value of backslash_char : %c \n", backslash_char);
printf("MAXIMUM marks : %d \n", Max_Marks);

getch();
}
```

### OUTPUT

A screenshot of a terminal window showing the output of the C program. The text is displayed in a monospaced font on a black background. The output lines are: 'value of height :100', 'value of number : 3.140000', 'value of letter : A', 'value of letter\_sequence : ABC', 'value of backslash\_char : ?', and 'MAXIMUM marks : 100'.

```
value of height :100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
MAXIMUM marks : 100
```



# KEYWORDS

- Keywords cannot be used as identifiers (Keywords : if, for, while etc...)
- The names are case sensitive. So, example, “FIRST” is different from “first” and “First”.
- It must begin with an alphabet or an underscore.
- It can be of any reasonable length. Though it should not contain more than 31 characters.
- Example: roll\_number, marks, name, emp\_number, basic\_pay, HRA, DA, dept\_code

# KEYWORDS

- Keywords are pre-defined words in a C compiler.
- Each keyword is meant to perform a specific function in a C program.
- Since keywords are referred names for compiler, they can't be used as variable name.
- C language supports 32 keywords

# KEYWORDS

<u>auto</u>	<u>double</u>
<u>int</u>	<u>struct</u>
<u>const</u>	<u>float</u>
<u>short</u>	<u>unsigned</u>
<u>break</u>	<u>else</u>
<u>long</u>	<u>switch</u>
<u>continue</u>	<u>for</u>
<u>signed</u>	<u>void</u>

<u>case</u>	<u>enum</u>
<u>register</u>	<u>typedef</u>
<u>default</u>	<u>goto</u>
<u>sizeof</u>	<u>volatile</u>
<u>char</u>	<u>extern</u>
<u>return</u>	<u>union</u>
<u>do</u>	<u>if</u>
<u>static</u>	<u>while</u>

## **SLO - 2**

# **Values, Names, Scope, Binding, Storage Classes**

# Values

There are two kinds of expressions in C –

- lvalue – Expressions that refer to a memory location are called "lvalue" expressions. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- rvalue – The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right-hand side but not on the left-hand side of an assignment.
- Variables are lvalues and so they may appear on the left-hand side of an assignment.
- Numeric literals are rvalues and so they may not be assigned and cannot appear on the left-hand side. Take a look at the following valid and invalid statements –

`int g = 20; // valid statement`

`10 = 20; // invalid statement; would generate compile-time error`

# Names

Function names and variable names are both identifiers in C.

**The following are all the characters you can use to make a valid variable name:**

1. Characters A through Z and a through z.
2. Digit characters 0 through 9, which can be used in any position except the first of a variable name.
3. The underscore character (\_).

For instance, `stop_sign`, `Loop3`, and `_pause` are all valid variable names.

**Now, let's see what you cannot use in variable naming:**

- A variable name cannot contain any C arithmetic signs.
- A variable name cannot contain any dots (.).
- A variable name cannot contain any apostrophes (').
- A variable name cannot contain any other special symbols such as \*, @, #, ?, and so on.

Some invalid variable names,

for example, are, `4flags`, `sum-result`, `method*4`, and `what_size?`.

# Scope of Variable

- **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

# Scope of Variable

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



# Scope of Variable

- 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

# Scope of Variable

*/\* 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;
```

```
}
```

# Scope of Variable

- 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

# Scope of Variable

```
/* 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;  
}
```

# 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

- Design Time
- Compile Time
- Link Time
- Run Time

# Binding

- **Design Time**

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

- **Compile Time**

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

# Binding

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

# Storage Classes

- Each and every variable declared in C contains not only its data type but also has a storage class specified with it.
- If we do not specify storage class of a variable compiler will assume its storage class as default i.e automatic.



# **Storage class of a variable tell us about characteristics of storage classes.**

- Storage place of the variable i.e memory or CPU registers.
- The initial value of a variable.
- The scope or the visibility of a variable
- The life time of a variable i.e how long the variable exists.

# Four types of storage classes in C

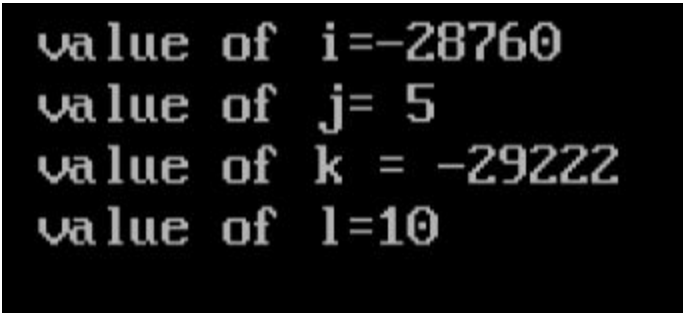
1. Automatic storage class(with specifier **auto**).
2. Register storage class(with specifier **register**).
3. Static storage class(with specifier **static**).
4. External storage class(with specifier **extern**).

# Automatic storage class(Local variables)

Storage	Memory
Default initial value	Garbage Value i.e unpredictable value
Scope or visibility	Local or visible to the block in which variable is declared eg. if variable is declared in a function it is only visible to that function.
Life Time	It retains its value till the control remains in the block in which it is declared. Eg: if the variable is declared in a function, it will retain its value till the control is in that function.

# Demonstration of auto storage class with default value

```
#include<stdio.h>
main()
{
auto int i , j=5;
int k , l = 10;
clrscr();
printf("\n value of i=%d \n value of j= %d", i , j);
printf("\n value of k = %d \n value of l=%d",k,l);
getch();
}
```



```
value of i=-28760
value of j= 5
value of k = -29222
value of l=10
```

# Demonstration of scope and visibility and lifetime of auto storage class.

```
#include<stdio.h>
void main()
{
    auto int x = 5;
    {
        auto int x = 10;
        {
            auto int x = 15;
            {
                clrscr();
                printf("%d",x);
            }
        }
        printf(" %d ",x);
    }
    printf(" %d",x);
    getch();
}
```

**OUTPUT:**

**15 10 5**

# Register Storage Class

Storage	CPU registers
Default initial value	Garbage Value i.e unpredictable value
Scope or visibility	Local or visible to the block in which variable is declared eg. if variable is declared in a function it is only visible to that function.
Life Time	It retains its value till the control remains in the block in which it is declared. Eg: if the variable is declared in a function, it will retain its value till the control is in that function.

## Demonstration of register storage class

```
#include<stdio.h>
void main()
{
    register int i;
    clrscr();
    for(i=1;i<=100;i++)
    {
        printf("%d ",i);
    }
    getch();
}
```

Speed of the program increases by using a variable as register storage class.

# Static Storage Class

Storage	Memory
Default initial value	zero
Scope or visibility	Local or visible to the block in which variable is declared e.g. if variable is declared in a function it is only visible to that function.
Life Time	It retains its value between different function calls i.e. when function is called for the first time, static variable is created with initial value zero and in subsequent calls it retains its present value.



# Demonstration of static storage class

```
#include<stdio.h>
void main()
{
    int i,r;
    clrscr();
    for(i=1;i<=5;i++)
    {
        r=tot (i);
        printf("\t %d",r);
    }
    getch();
}
tot (int x)
{
    static int s=0;
    s=s+x;
    return (s);
}
```

main() calls the function tot() five times, each time sending the value of i varying from 1 to 5. In the function value is added to s and the accumulated result is returned. Once this program is executed following result is returned.

## output

**1   3   6   10   15**

# External Storage Class

- Global variables in same file
- Variables declared in the other file and referred in current file

# External variable

Storage	Memory
Default initial value	zero
Scope or visibility	local to the Block in which it is referred
Life Time	Till the termination of block in which it is referred. Block may be a whole program.

# Extern-Example

First File: EXTERNAL.c

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

```
int count ;
```

```
extern void write_extern();
```

```
main() {
```

```
    count = 5;
```

```
    write_extern();
```

```
}
```

Second File: EXTERNAL2.c

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

```
extern int count;
```

```
void write_extern(void) {
```

```
    printf("count is %d\n", count);
```

```
}
```

Output:

count is 5

**Auto**

**Every time new value**

**Static**

**Retains value between calls**

**Extern**

**Variable is defined Outside**

**Register**

**Value is Stored in CPU Register**

# **SLO - 1**

## **Numeric Data types : int**

# Data Types

*A type defines a set of values and a set of operations that can be applied on those values.*

**There are 4 basic *data types* :**

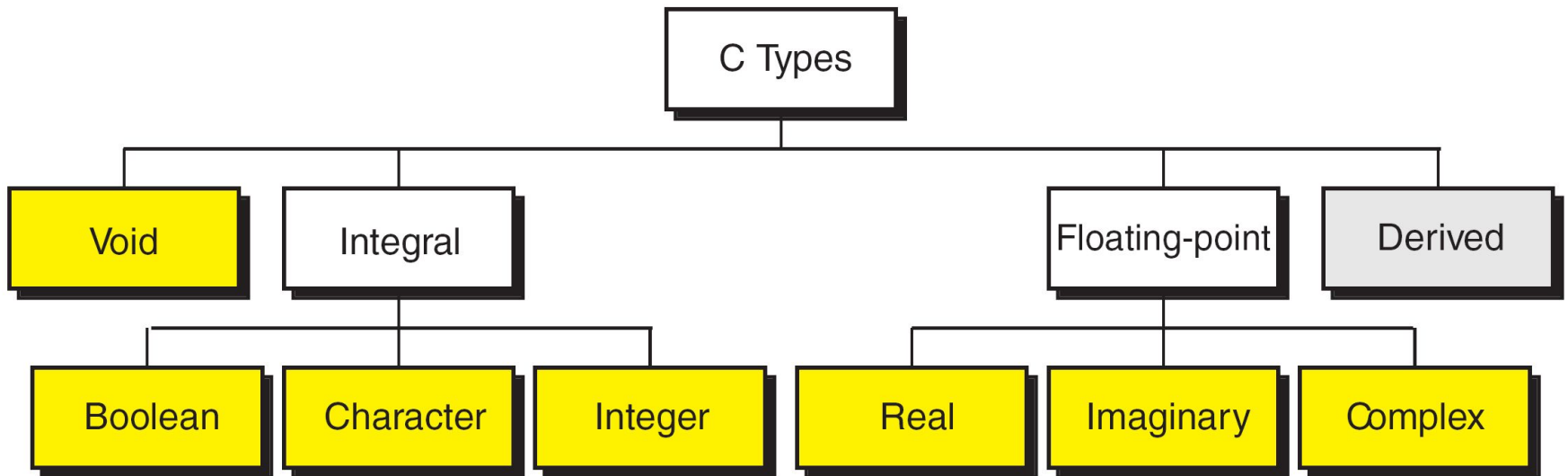
**int**

**float**

**double**

**char**

# Data Types

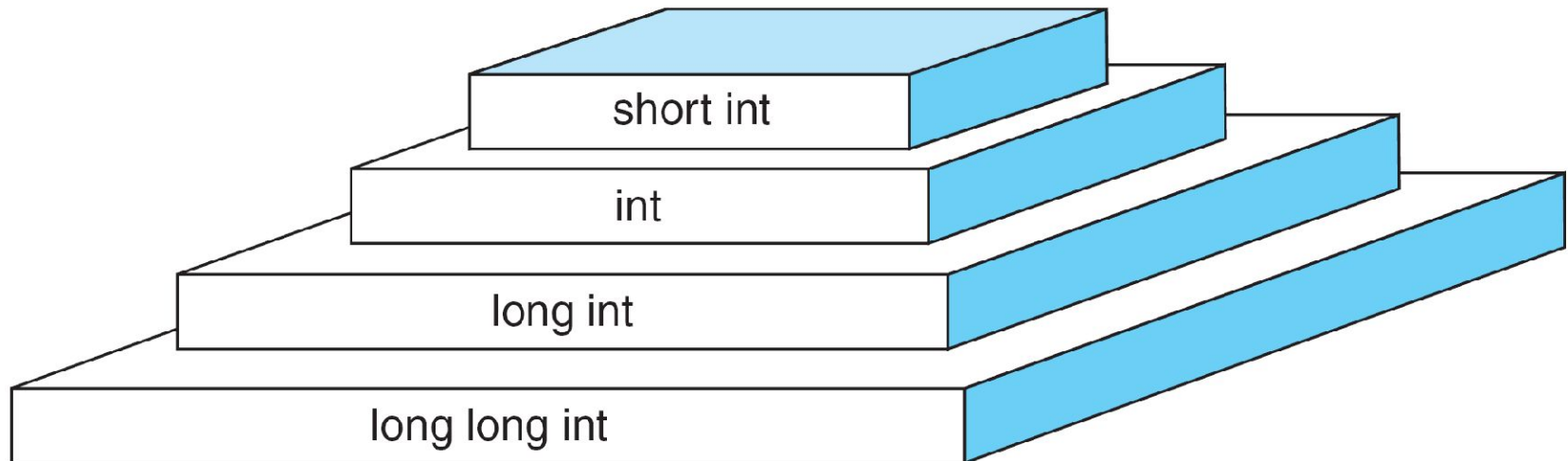




# Integer Types

## int

- used to declare numeric program variables of integer type
- whole numbers, positive and negative
- keyword: int
  - int number;
  - number = 12;



## *Note*

**$\text{sizeof (short)} \leq \text{sizeof (int)} \leq \text{sizeof (long)} \leq \text{sizeof (long long)}$**

# Typical Integer Sizes and Values for Signed Integers

Type	Byte Size	Minimum Value	Maximum Value
short int	2	-32,768	32,767
int	4	-2,147,483,648	2,147,483,647
long int	4	-2,147,483,648	2,147,483,647
long long int	8	-9,223,372,036,854,775,807	9,223,372,036,854,775,806

# Examples of Integer Constants

Representation	Value	Type
+123	123	int
-378	-378	int
-32271L	-32,271	long int
76542LU	76,542	unsigned long int
12789845LL	12,789,845	long long int

**SLO : 2**

**Numeric Data types : floating point**

# Floating-point Types

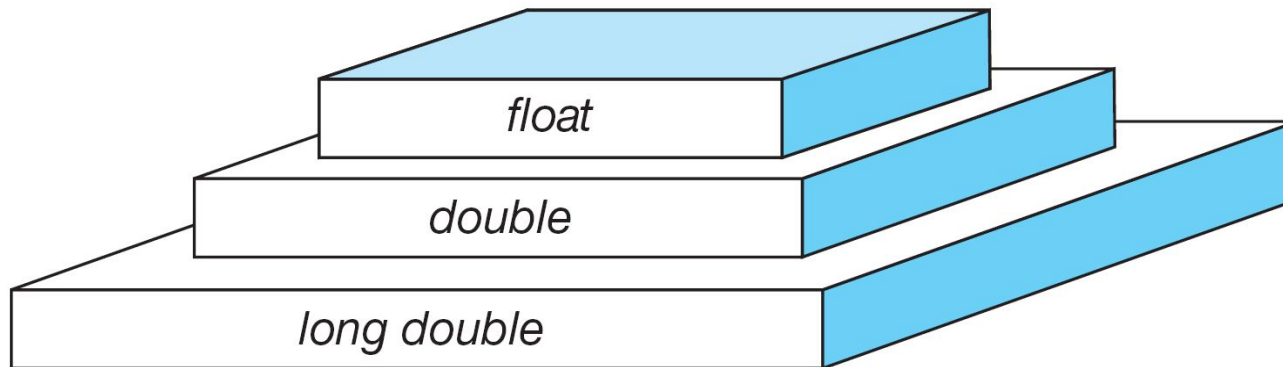
## float

fractional parts, positive and negative

keyword: float

float height;

height = 1.72;



## **Note**

**$\text{sizeof (float)} \leq \text{sizeof (double)} \leq \text{sizeof (long double)}$**

# Examples of Real Constants

Representation	Value	Type
0.	0.0	double
.0	0.0	double
2.0	2.0	double
3.1416	3.1416	double
-2.0f	-2.0	float
3.1415926536L	3.1415926536	long double



## **SLO - 1**

# **Non-numeric data types : char and string**

# Character Types

## char

equivalent to 'letters' in English language

Example of characters:

Numeric digits: 0 - 9

Lowercase/uppercase letters: a - z and A - Z

Space (blank)

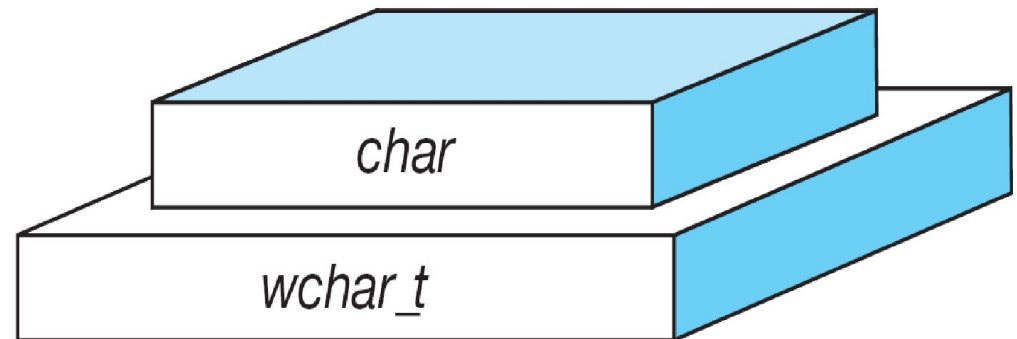
Special characters: , . ; ? " / ( ) [ ] { } \* & % ^ < > etc

single character

keyword: char

```
char my_letter;  
my_letter = 'U';
```

The declared character must be enclosed within a single quote!



**wchar\_t** is a wide character. It is used to represent characters which require more memory to represent them than a regular char

# Type Summary

Category	Type	C Implementation
Void	Void	<i>void</i>
Integral	Boolean	<i>bool</i>
	Character	<i>char, wchar_t</i>
	Integer	<i>short int, int, long int, long long int</i>
Floating-Point	Real	<i>float, double, long double</i>
	Imaginary	<i>float imaginary, double imaginary, long double imaginary</i>
	Complex	<i>float complex, double complex, long double complex</i>

# Constants

*Constants are data values that cannot be changed during the execution of a program. Like variables, constants have a type.*

## *Note*

**A character constant is enclosed in single quotes.**

# Symbolic Names for Control Characters

ASCII Character	Symbolic Name
null character	'\0'
alert (bell)	'\a'
backspace	'\b'
horizontal tab	'\t'
newline	'\n'
vertical tab	'\v'
form feed	'\f'
carriage return	'\r'
single quote	'\''
double quote	'\"'
backslash	'\\'

# Some Strings

```
" " // A null string
"h"
"Hello World\n"
"HOW ARE YOU"
"Good Morning!"
L"This string contains wide characters."
```

## Null Characters and Null Strings

'\0'

" "



Null character



Empty string

## *Note*

**Use single quotes for character constants.  
Use double quotes for string constants.**



## **SLO - 2**

# **Increment & Decrement Operators**

# Increment & Decrement Operators

C supports 2 useful operators namely

1. Increment ++
  2. Decrement -- operators
- The ++ operator adds a value 1 to the operand
  - The -- operator subtracts 1 from the operand
- ++a or a++
- a or a--

# Rules for ++ & -- operators

1. These require variables as their operands
2. When postfix either ++ or -- is used with the variable in a given expression, the expression is evaluated first and then it is incremented or decremented by one
3. When prefix either ++ or -- is used with the variable in a given expression, it is incremented or decremented by one first and then the expression is evaluated with the new value

# Examples for ++ & -- operators

Let the value of a =5 and b=++a then

a = b =6

Let the value of a = 5 and b=a++ then

a =5 but b=6

i.e.:

1. a prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left
2. a postfix operator first assigns the value to the variable on left and then increments the operand.

## **SLO - 1**

# **Comma, Arrow and Assignment Operator**

# Comma Operator

- Comma operator represented by ‘,’ ensures the evaluation from left to right, one by one, of two or more expressions, separated with commas, and **result** of entire expression is value of **rightmost expression**. For example:

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

```
printf("Value of exp. \"x = a + b, b + c, c + a;\" is %d\\n\",  
      (x = a + b, b + c, c + a));
```

```
/* value of x is value of rightmost sub-exp. c + a */
```

# Comma Operator

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

```
int x;
```

```
printf("Value of exp. \"x = a + b, b + c, c + a;\" is %d\\n",  
      x = a + b, b + c, c + a);
```

```
/* value of x is value of leftmost sub-exp. a + b */
```

- Actually, in the above printf() statement, this time there are three arguments, separated with commas, following format string for one format specifier.
- All expressions are evaluated but value of  $x = a + b$  is printed.
- To avoid such problems, enclose the entire expression in pair of parenthesis.

# Comma Operator

This operator is of great worth in multiple initializations, updating values in for and while loop statements. For example:

```
/*comma2.c----usage of comma operator */
#include <stdio.h>
int main(void)
{
    int x, y, z;

    for (x = getval(), y = x + 1; y >= 0 ; x++, y = x + 1) {
        ...
        x = getval();
    }

    return 0;
}
```



# Arrow Operator

- In C, this operator enables the programmer to access the data elements of a Structure or a Union.
- This operator(->) is built using a minus(-) operator and a greater than(>) relational operator.
- Moreover, it helps us access the members of the struct or union that a pointer variable refers to.

## Syntax of Arrow operator(->)

**(pointer variable)->(variable) = value;**

- The operator is used along with a pointer variable. That is, it stores the value at the location(variable) to which the pointer/object points.

# Arrow operator to access the data member of a C Structure

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

```
struct Stu_details
```

```
{
```

```
    char *name;
```

```
    char *reg_no;
```

```
    int age;
```

```
};
```

```
int main()
```

```
{
```

```
    struct Stu_details* M;
```

```
    clrscr();
```

```
    M->name = "RAM";
```

```
    M->reg_no="RA1111111111";
```

```
    M->age=22;
```

```
    printf("\n\tStudent Details:");
```

```
    printf("\n\t*****");
```

```
    printf("\n\tName : %s", M->name);
```

```
    printf("\n\tReg_no: %s", M->reg_no);
```

```
    printf("\n\tAge : %d",M->age);
```

```
    getch();
```

```
    return 0;
```

```
}
```

## OUTPUT

```
Student Details:
```

```
*****
```

```
Name : RAM
```

```
Reg_no: RA1111111111
```

```
Age : 22
```

# Assignment operators

- Assignment operators are used to assigning value to a variable.
- The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value.
- The value on the right side must be of the same data-type of the variable on the left side otherwise the compiler will raise an error.

## Syntax:

**v** **op** = **exp**;

Where, **v** = variable,

**op** = shorthand assignment operator

**exp** = expression

**Ex:**     **x=x+3**

**x+=3**

# Two categories of assignment operators

There are 2 categories of assignment operators in C language.

They are,

1. Simple assignment operator ( Example: = )
2. Compound assignment operators ( Example: +=, -=, \*=, /=, %=, &=, ^= )

Operators	Example/Description
=	sum = 10; 10 is assigned to variable sum
+=	sum += 10; This is same as sum = sum + 10
-=	sum -= 10; This is same as sum = sum – 10
*=	sum *= 10; This is same as sum = sum * 10
/=	sum /= 10; This is same as sum = sum / 10
%=	sum %= 10; This is same as sum = sum % 10
&=	sum&=10; This is same as sum = sum & 10
^=	sum ^= 10; This is same as sum = sum ^ 10

# Shorthand Assignment operators

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

## **SLO - 2**

# **Bitwise and sizeof Operator**

# Bitwise operators

- These operators allow manipulation of data at the bit level

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

# Bitwise operators

- In arithmetic-logic unit (which is within the CPU), 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.



# Bitwise AND operator &

- The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.

# Example : Bitwise AND

Let us suppose the bitwise AND operation of two integers 12 and 25.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

00001100  
& 00011001

---

00001000 = 8 (In decimal)

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a&b);
    return 0;
}
```

Output

Output = 8

# Bitwise OR operator |

- The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1.
- In C Programming, bitwise OR operator is denoted by |

# Example : Bitwise OR

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12  
and 25

```
00001100
| 00011001
```

---

00011101 = 29 (In decimal)

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a | b);
    return 0;
}
```

**Output**

Output = 29