



## Unit-1

# Introduction to C Programming Language



**D. SRINIVAS**

Department of CSE

[www.srinivas-materials.blogspot.com](http://www.srinivas-materials.blogspot.com)

[srinivascsdept@gmail.com](mailto:srinivascsdept@gmail.com)

+91 9347556447





## Outline



- **Introduction to components of a computer system:**
  - ↳ Compilers
  - ↳ Creating, compiling and executing a program etc.,
- **Introduction to Algorithms:**
  - ↳ Representation of Algorithm, Flowchart/Pseudo code with examples,
  - ↳ Program design and structured programming.
- **Introduction to C Programming Language:**
  - ↳ Simple input and output with scanf and printf, variables
  - ↳ Syntax and Logical Errors in compilation, object and executable code,
  - ↳ Operators, expressions and precedence, Expression evaluation,
  - ↳ Type conversion, Command line arguments
- **Conditional Branching and Loops:**
  - ↳ Writing and evaluation of conditionals and consequent branching with if, if-else, switch-case,
  - ↳ ternary operator, goto, break
  - ↳ Iteration with for, while, do-while loops

# Introduction to Programming



# Creating and Running Programs

- Creating and running programs takes place in 4 steps.
  1. Writing and Editing the program.
  2. Compiling.
  3. Linking the program with the required library functions.
  4. Executing the program.

**Step 1 Creating Source Code**

Press  
**F2** To  
Save

**Step 2 Compiling Source Code**

Press  
**Alt +**  
**F9**

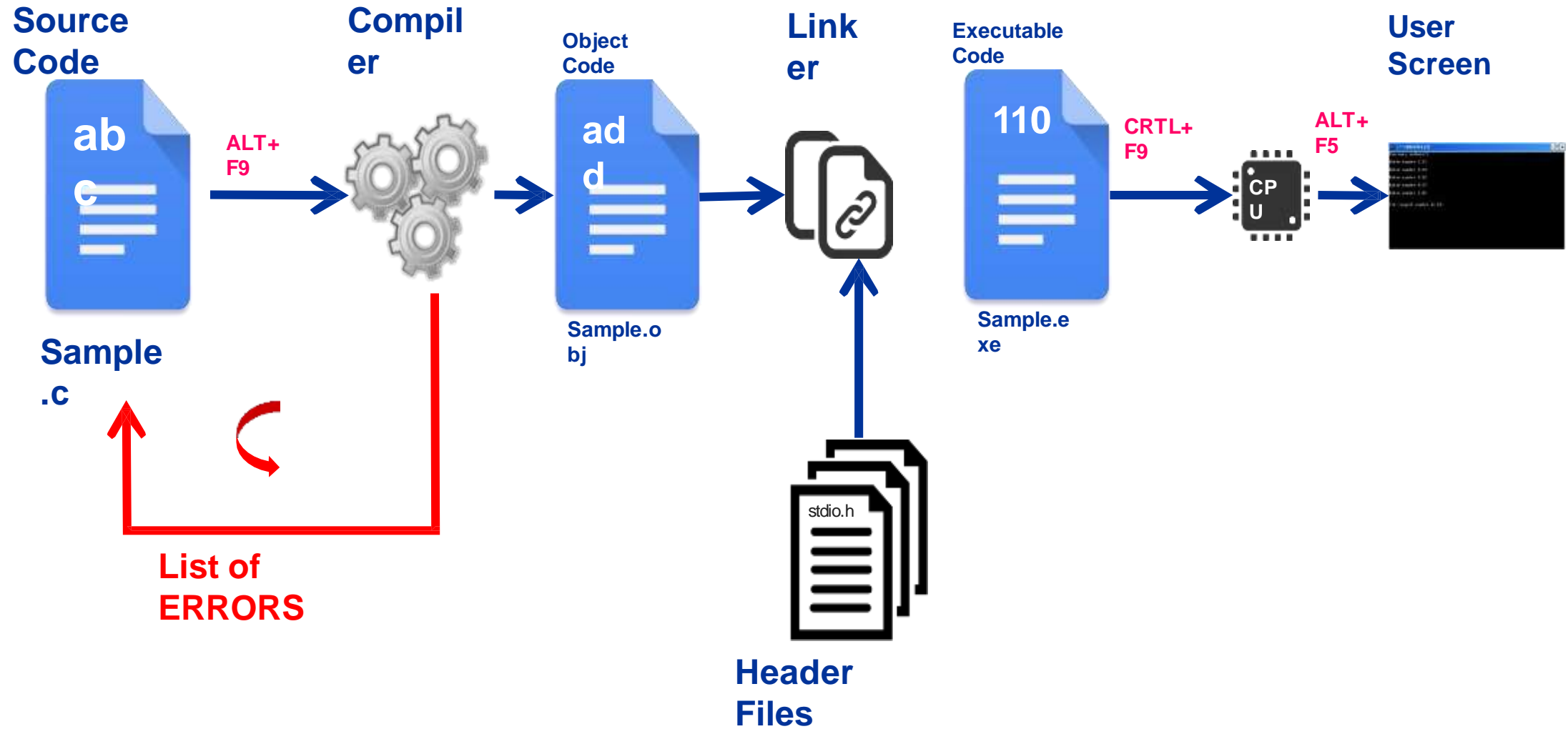
**Step 3 Linking with Library**

Press  
**Alt +**  
**F9**

**Step 4 Executing Source Code**

Press  
**Ctrl +**  
**F9**

# C Program Execution Process



# C Program Execution Process

- **1. Writing and Editing the program**

- ➔ Software used to write programs is known as a **text editor**, where you can type, edit and store the data.
- ➔ You can write a **C** program in text editor and save that file on to the disk with “**.c**” extension. This file is called **source file**.

- **2. Compiling Program**

- ➔ **Compiler** is used to convert High Level Language instructions into the Machine Language instructions.
- ➔ It could complete its task in two steps.

- i) Preprocessor

- ii) Translator

- Preprocessor:**

- It reads the source file and checks for special commands known as preprocessor commands ( instructions which starts with # symbols ).
    - The result of preprocessor is called as **translation unit**.
    - Preprocessor processes the source file before compilation only.

- Translator:**

- It is a program which reads the **translation unit** and converts the program into machine language and gives the **object module**.
    - This module is not yet ready to run because it does not have the required C and other functions included.

# C Program Execution Process

- **3. Linking a program with required library functions**

- C program is made up of different functions in which some functions can be written by the programmer, other functions like **input/output** functions and **mathematical** library functions, that exist **elsewhere** and must be attached to our program.
- The **linker** assembles all of these functions and produces the **executable** file which is ready to run on the computer.

- **4. Executing the program.**

- Once a program has been linked, it is ready for execution.
- Now, you can execute the program by using the run command.
- **Loader** is a program which is used to load the program from the disk to main memory.

1. Write a program (source code) using vi editor and save it with **.c** extension. Ex: **\$vi sample.c**
2. Run the compiler to convert a program into to “binary” code. Ex: **\$cc sample.c**
3. Compiler gives errors and warnings if any, then edit the source file, fix it, and re-compile.
4. Run it and see the output. Ex: **\$ ./a.out**

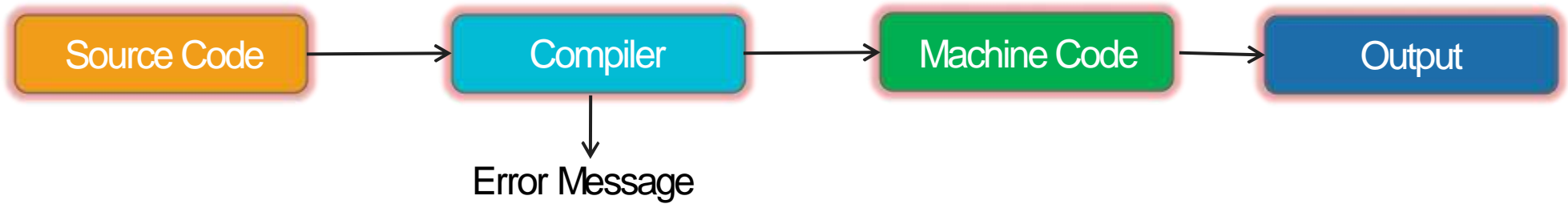
# Translators

- A program written in high-level language is called as source code. To convert the source code into machine code, translators are needed.
- A translator takes a program written in source language as input and converts it into a program in target language as output.
- It also detects and reports the error during translation.
- **Roles of translator are:**
  - ➔ Translating the high-level language program input into an equivalent machine language program.
  - ➔ Providing diagnostic messages wherever the programmer violates specification of the high-level language program.
- **Different type of translators**
- The different types of translator are as follows:
  - ➔ **Compiler**
  - ➔ **Interpreter**
  - ➔ **Assembler**

# Translators

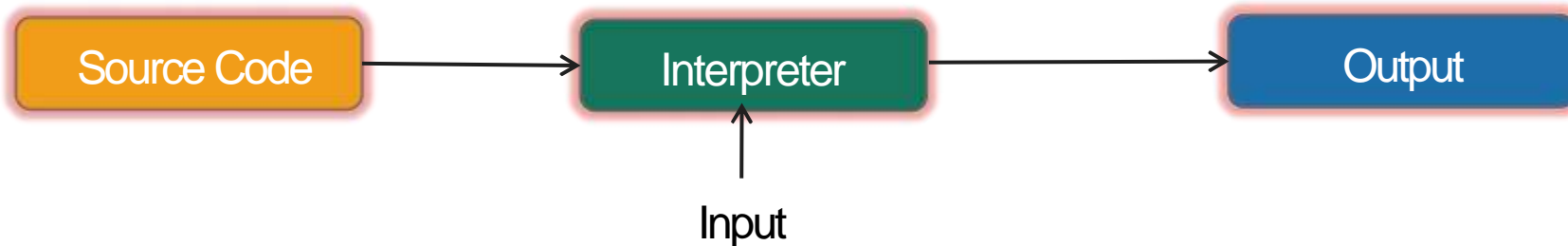
## • Compiler

- Compiler is a translator which is used to convert programs in high-level language to low-level language.
- It translates the entire program and also reports the errors in source program encountered during the translation.



## • Interpreter

- Interpreter is a translator which is used to convert programs in high-level language to low-level language. Interpreter translates line by line and reports the error once it encountered during the translation process.
- It directly executes the operations specified in the source program when the input is given by the user.
- It gives better error diagnostics than a compiler.



# Translators

- **Assembler**

➔ Assembler is a translator which is used to translate the assembly language code into machine language code.



# Introduction to Algorithms



# Program Development

- It is a multistep process that requires that:
  - ➔ 1. Understand the problem
  - ➔ 2. Develop Solution
    - 1. Structure Chart
    - 2. Algorithm / Pseudo code
    - 3. Flowchart
  - ➔ 3. Write the program
  - ➔ 4. Test the program
- **1. Understand the Problem**
  - ➔ The first step in solving any problem is to understand it.
  - ➔ To solve any problem first you must understand the problem by reading the requirements of the problem.
  - ➔ Once you understand it, review with user(customer) and system analyst.

# Program Development

- **2. Develop the Solution**

- ➔ To develop a solution to a problem the following tools are needed.

- 1. Structure Chart:**

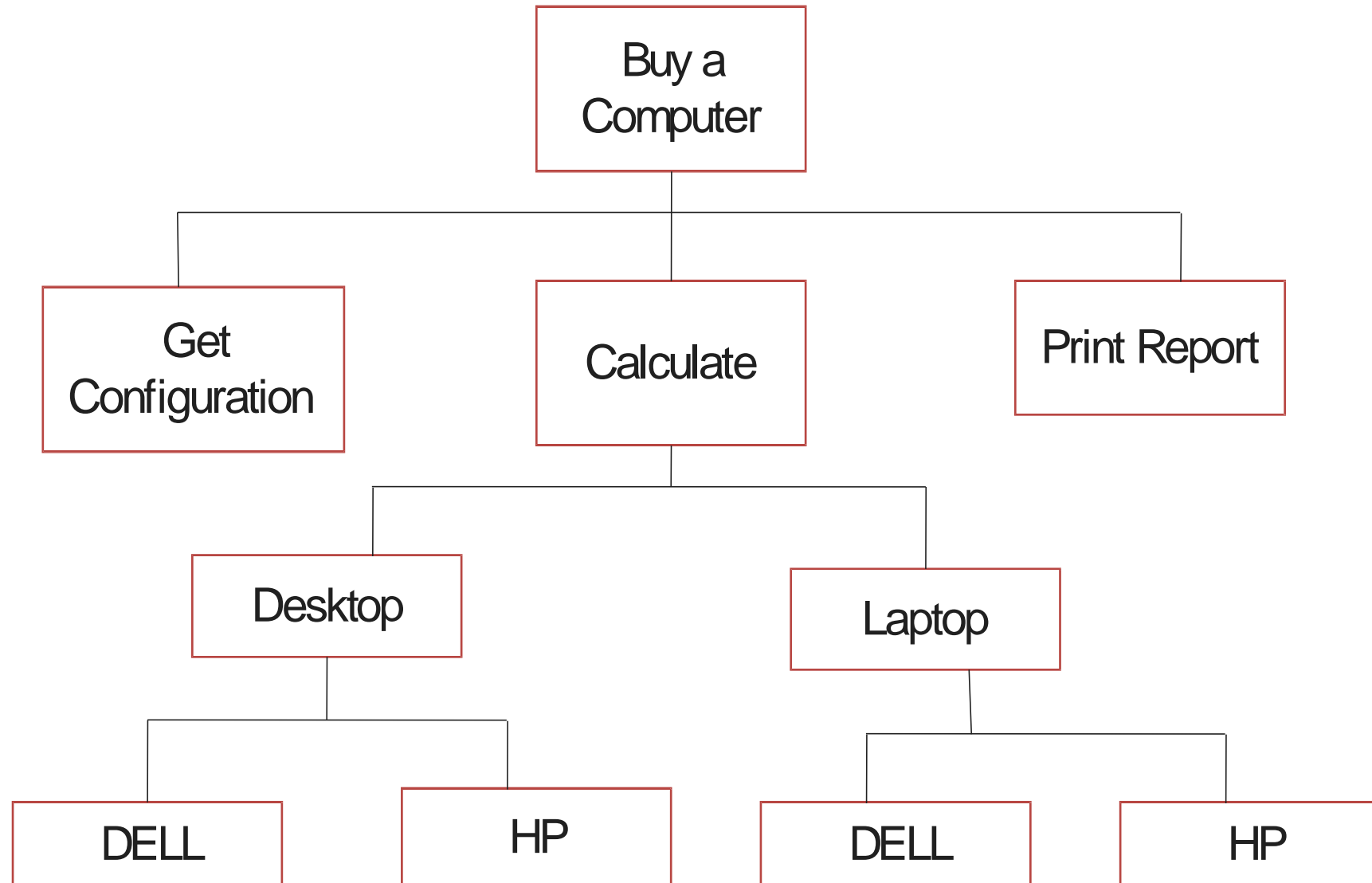
- ➔ It is also known as a hierarchy chart, shows the functional flow through your program.
    - ➔ It shows how the problem is broken into logical steps, each step will be a separate module.
    - ➔ It also shows the interaction between all the parts of your program.
    - ➔ It is like the architect's blueprint.

The below two are used to design the individual parts of the program.

- 1. Algorithm / Pseudo Code**

- 2. Flowchart**

# Example Structure Chart



# Algorithm

- **Algorithm:** It is an **ordered sequence** of **unambiguous** and **well-defined instructions** that **performs some task** and **halts in finite time**.

- Let's examine the four parts of this definition more closely.

1. **Ordered Sequence:** You can number the step.
2. **Unambiguous** and well defined instructions: Each instruction should be clear, well understand.
3. **Performs** some task
4. **Halts in finite time:** Algorithm must terminate at some point

## ► **Properties of an Algorithm:-**

1. **Finiteness:** An algorithm must terminate in a finite number of steps.
2. **Definiteness:** Each step of an algorithm must be precisely and unambiguously stated.
3. **Effectiveness:** Each step must be effective, and can be performed exactly in a finite amount of time.
4. **Generality:** The algorithm must be complete in itself.
5. **Input/Output:** Each algorithm must take zero, one or more inputs and produces one or more output.

# Algorithm

- **Three Categories of Algorithmic Operations**

- ➔ An algorithm must have the ability to alter the order of its instructions. An instruction that alters the order of an algorithm is called a control structure.

- Three categories of an algorithmic operations:

1. **Sequential operations:** Instructions are executed in order

2. **Conditional/Selection ("question asking") Operations:** A control structure that asks a true/false question and then selects the next instruction based on the answer.

3. **Iterative Operations (loops):** A control structure that repeats the execution of a block of instructions.

# Pseudo Code

- **Definition:** English-like statements that follow a loosely defined syntax and are used to convey the design of an algorithm.
- **Example1:** To determine whether a student is passed or not

## Pseudo Code:

1. If student's grade is greater than or equal to 60
  1. Print "passed"
2. else
  1. Print "failed"

## Algorithm:

### Begin

1. If grade  $\geq$  60
  1. Print  
"passed"
2. else
  1. Print  
"failed"

### End

# Pseudo Code

- **Example 2:** Write an algorithm to determine a student's final grade and indicate whether it is passing or failing. The final grade is calculated as the average of four marks.

## Pseudo Code:

1. Input set of 4 marks
2. Calculate their average by summing and dividing by 4
3. if average is below 50
4.     Print "FAIL"
5. else
6.     Print "PASS"

## Algorithm:

### Begin

Step 1: Input M1,M2,M3,M4

Step 2:     GRADE  $\leftarrow$   
          (M1+M2+M3+M4)/4

Step 3:     if (GRADE < 50) then

Step 4:             Print "FAIL"

Step 5:     else





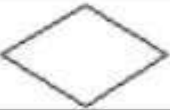


Step 6:             Print "PASS"

Step 7:     endif

### End

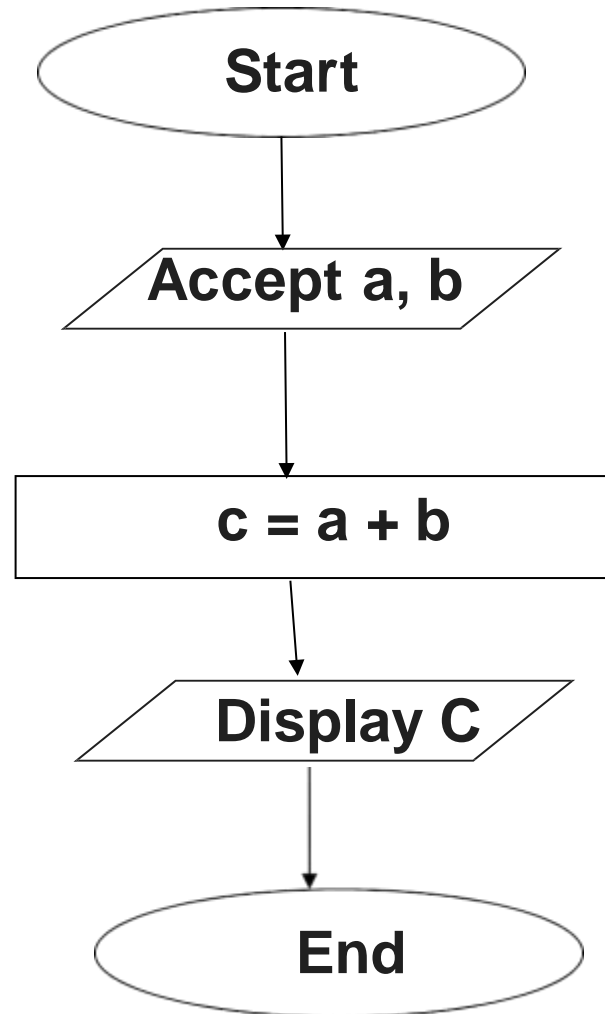
# Flow chart

- **Flowchart:** Pictorial representation of an algorithm is called flowchart.  
or
- A diagram that uses graphic symbols to depict the nature and flow of the steps in a process.

Symbol	Symbol Name	Description
	Flow Lines	Used to connect symbols
	Terminal	Used to start, pause or halt in the program logic
	Input/output	Represents the information entering or leaving the system
	Processing	Represents arithmetic and logical instructions
	Decision	Represents a decision to be made
	Connector	Used to Join different flow lines
	Sub function	used to call function

# Flow chart

- **Example:** Addition of two numbers



## Algorithm

**Start:**

**Step 1:** Read a, b values

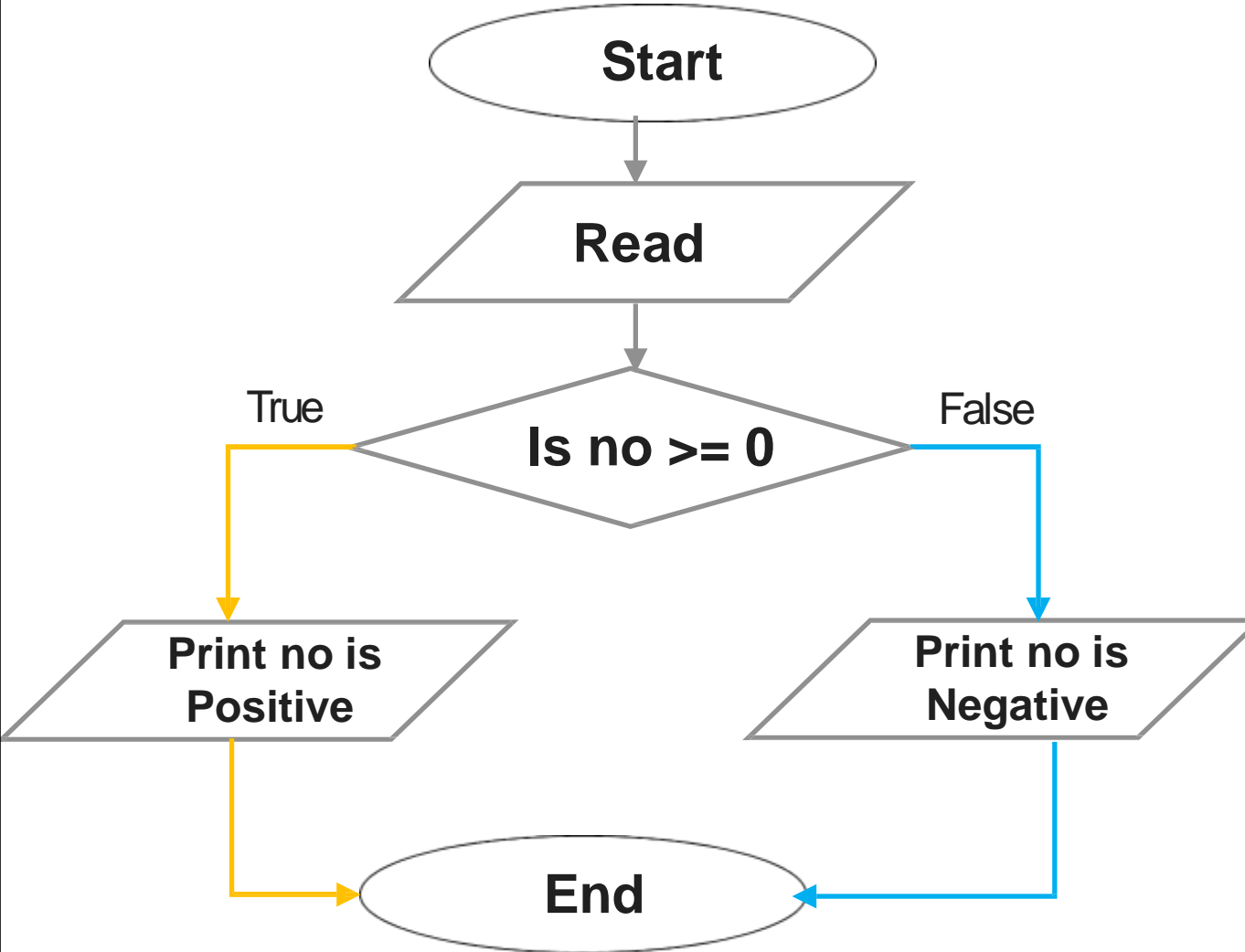
**Step 2:** Sum of a,b

**Step 3:** Print "C".

**Stop:**

# Flow chart

- **Example:** Number is positive or negative



## Algorithm

**Start:**

**Step 1:** Read no.

**Step 2:** If no is greater than equal zero, go to step 4.

**Step 3:** Print no is a negative number, go to step 5.

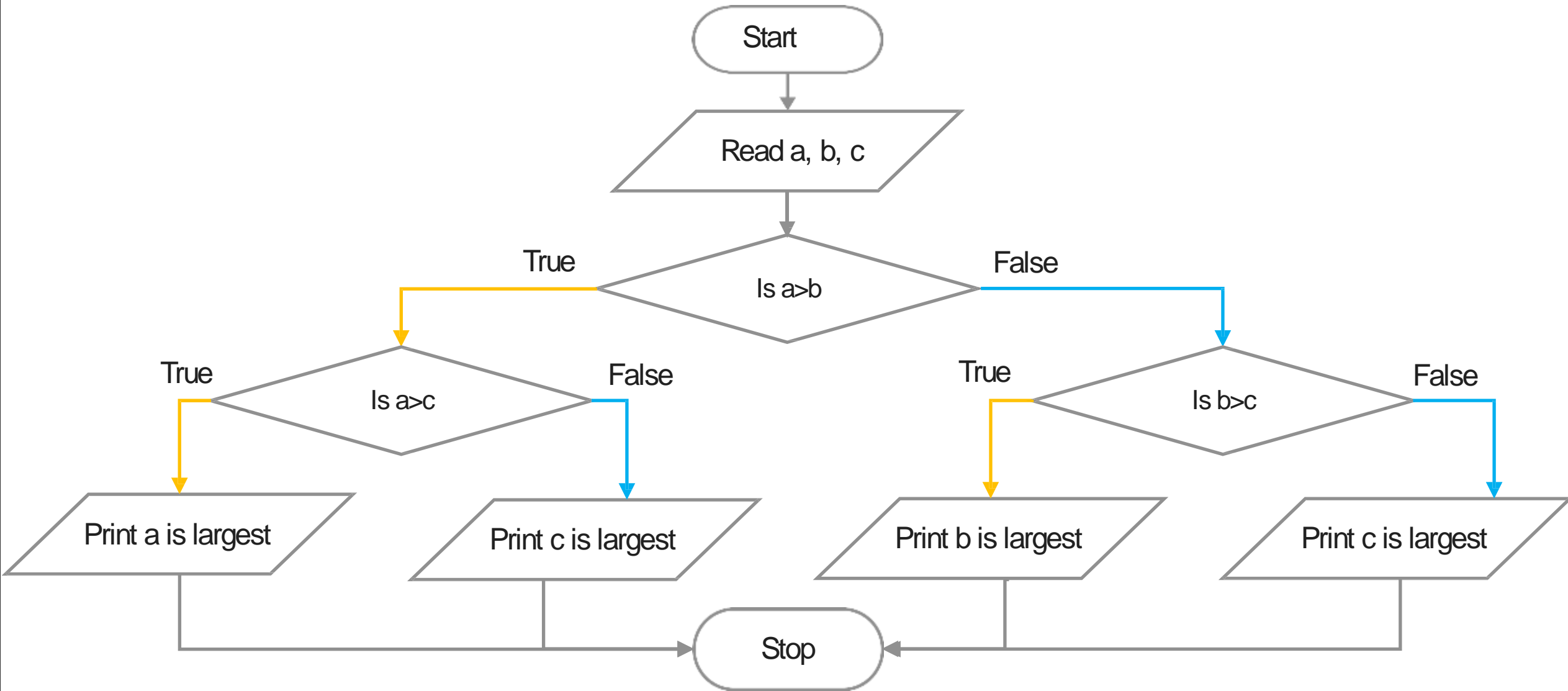
**Step 4:** Print no is a positive number.

**Step 5:** Stop.

**Stop:**

# Flow chart

- **Example:** Largest number from 3 numbers (Flowchart)



# Algorithm

- **Example:** Largest number from 3 numbers  
(Algorithm)

## Start:

Step 1: Read a, b, c.

Step 2: If  $a > b$ , go to step 5.

Step 3: If  $b > c$ , go to step 8.

Step 4: Print c is largest number, go to step 9.

Step 5: If  $a > c$ , go to step 7.

Step 6: Print c is largest number, go to step 9.

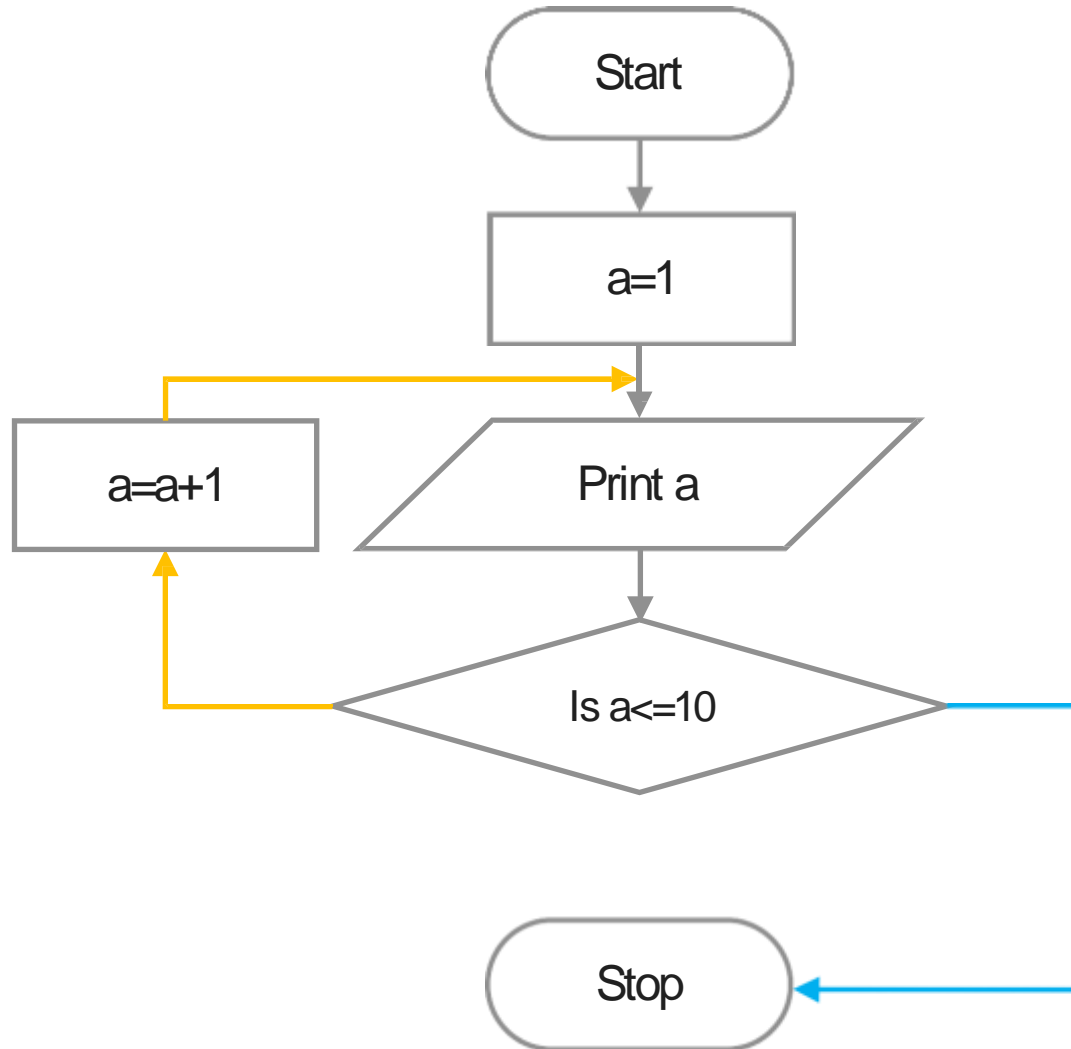
Step 7: Print a is largest number, go to step 9.

Step 8: Print b is largest number.

## Stop:

# Flow chart

- **Example:** Print 1 to 10



## Algorithm

**Start:**

Step 1: Initialize a to 1.

Step 2: Print a.

Step 3: Repeat step 2 until  $a \leq 10$ .

Step 3.1:  $a = a + 1$ .

**Stop:**

# Differences between Flowchart and Algorithm

Flowchart	Algorithm
Flowchart is a pictorial or graphical representation of a program.	Algorithm is a finite sequence of well defined steps for solving a problem.
It is drawn using various symbols.	It is written in the natural language like English.
Easy to understand.	Difficult to understand.
Easy to show branching and looping.	Difficult to show branching and looping.
Flowchart for big problem is impractical.	Algorithm can be written for any problem.

# Introduction to C Programming Languages



# C History

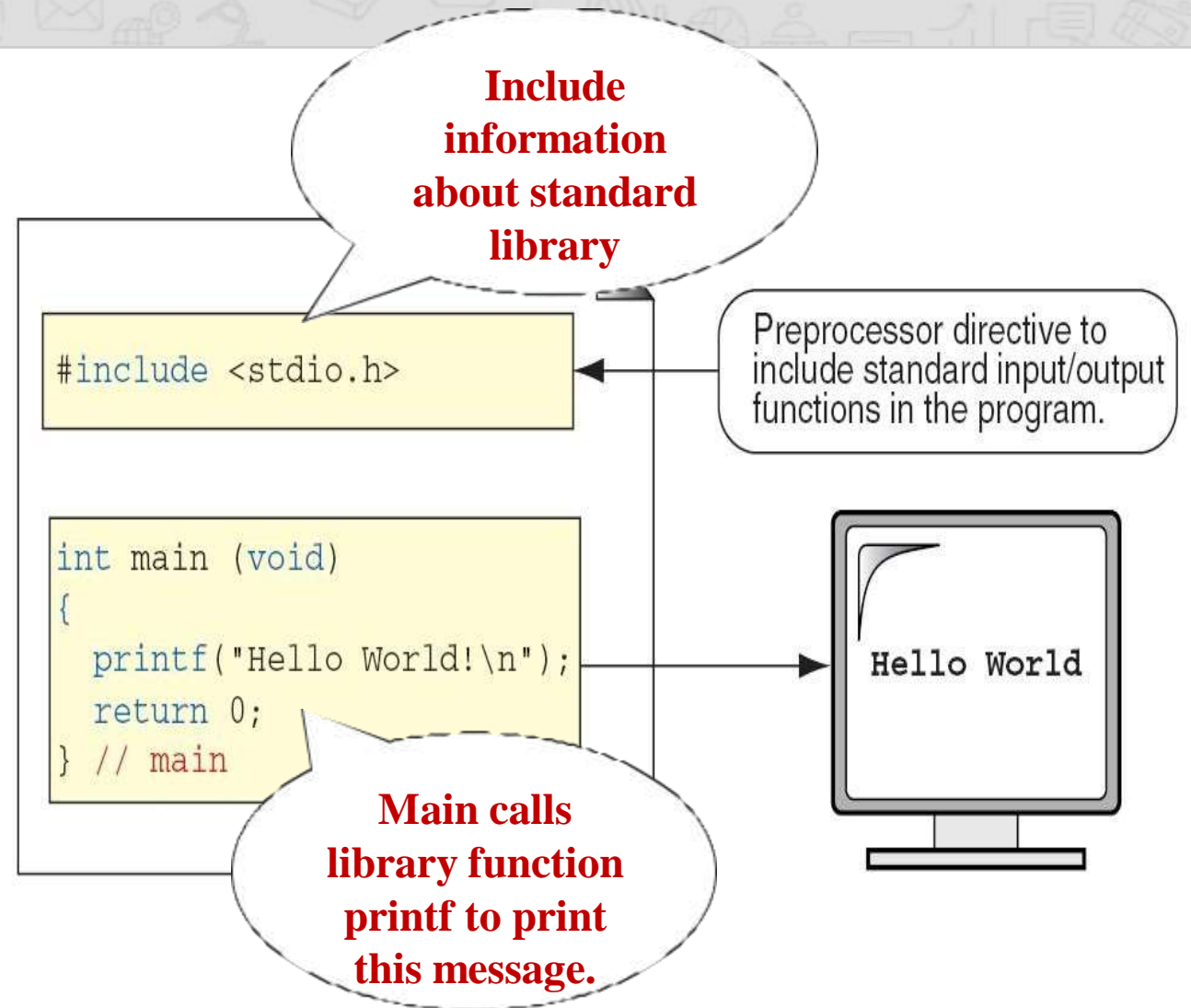
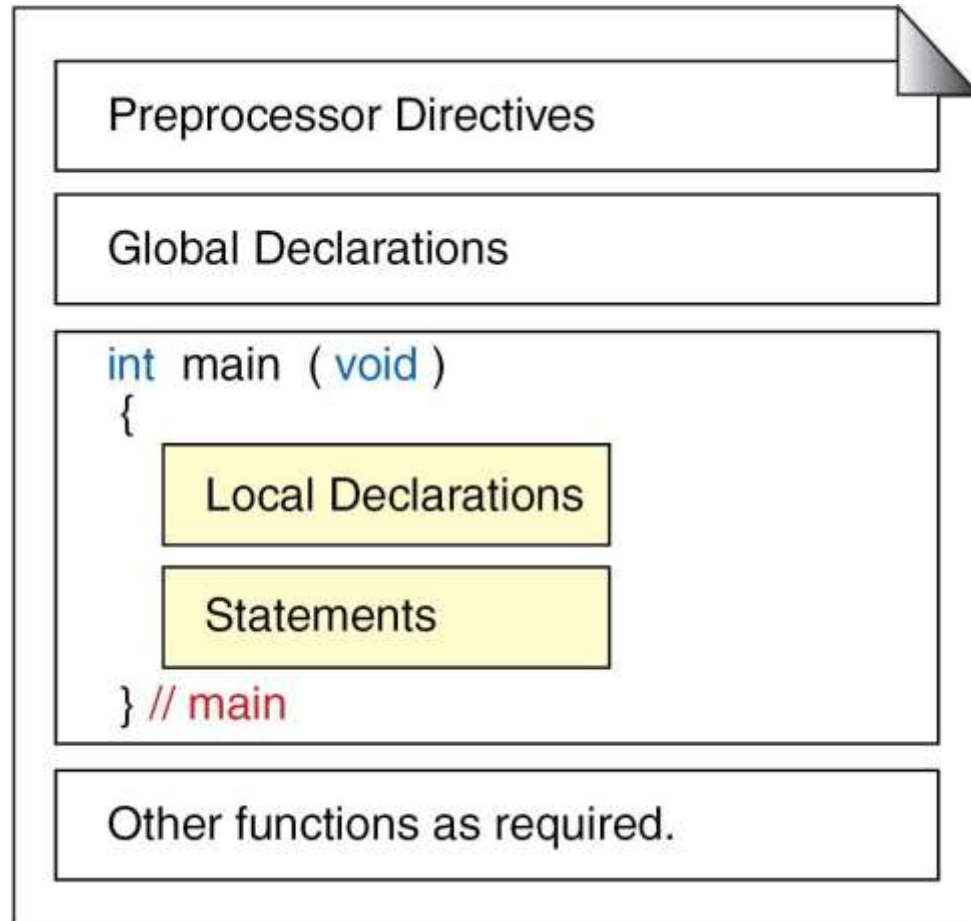
- ALGOL was the first computer language.
- In 1967, **Martin Richards** developed a language called BCPL (Basic Combined Programming Language) at University of Cambridge primarily, for writing system software.
- In 1969, language B was developed by Ken Thompson.
- 'B' was used to create early versions of UNIX operating system at Bell Laboratories.
- In 1972, C was developed by Dennis M. Ritchie at Bell Labs(AT&T) in USA.
- In 1988 C language was standardized by ANSI as ANSI C (ANSI- American National Standards Institute).
- UNIX operating system was coded almost entirely in C.

# C Features

The increasing popularity of C is due to its various features:

- **Robust:** C is a robust language with rich set of built-in functions and operators to write any complex programs.
- **C compilers combines the capabilities of low level languages with features of high level language.** Therefore it is suitable for writing the system software, application software and most of the compilers of other languages also developed with C language.
- **Efficient and Fast:** Programs written in C are efficient and fast. This is due to its variety of data types.
- **Portable:** C program written on one computer can also run on the other computer with small or no modification.  
**Example:** C program written in windows can also run on the Linux operating system.
- **Structured Programming:** Every program in C language is divided into small modules or functions so that it makes the program much simple, debugging, and also maintenance of the program is easy.
- **Ability to extend itself:** A C program is basically a collection of various functions supported by C library (also known as header files). We can also add our **own functions** to the **C library**. These functions can be reused in other applications or programs.

# Structure of C Program



# Preprocessor directives

- The **preprocessor directives** provide instructions to the preprocessor, to include functions from the system library, to define the symbolic constants and macro.
- The preprocessor command always starts with symbol **#**.
- Example: `#include<stdio.h>`
- **Header file** contains a collection of library files.
- `#include<stdio.h>` includes the information about the standard input/output library.
- The variables that are used in common by more than one function are called **Global Variables** and are declared in global declaration section.
- Every C program must have one **main()** function. All the statements of main are enclosed in braces.
- The program execution begins at main() function and ends at closing brace of the main function.
- C program can have any number of **user-defined functions** and they are generally placed immediately after the main () function, although they may appear in any order.

# Preprocessor directives

- All sections except the main () function may be absent when they are not required.
- In the previous program, main() function **returns** an integer value to the **operating system**.
- Each statement in C program must end with **;** specifies that the instruction is ended.
- A function can be called by **it's name**, followed by a parenthesized list of arguments and ended with semicolon.
- In previous program main() function calls printf() function.
- **Example:** `printf("Hello World!");`

# Comments

- To make the program more readable use the comments.
- They may used to make the program easier to understand.
- Two types of comments
  - ➔ 1. Block comment
  - ➔ 2. Line comment

## 1.Block comment :

- ➔ Any characters between `/*` and `*/` are ignored by the compiler.
- ➔ Comments may appear anywhere in a program.
- ➔ `/*` and `*/` is used to comment the multiple lines of code which is ignored by the compiler.
- ➔ Nested block comments are invalid like `/* /* */`
- ➔ Ex: `/* Write a program to add two integer numbers */`

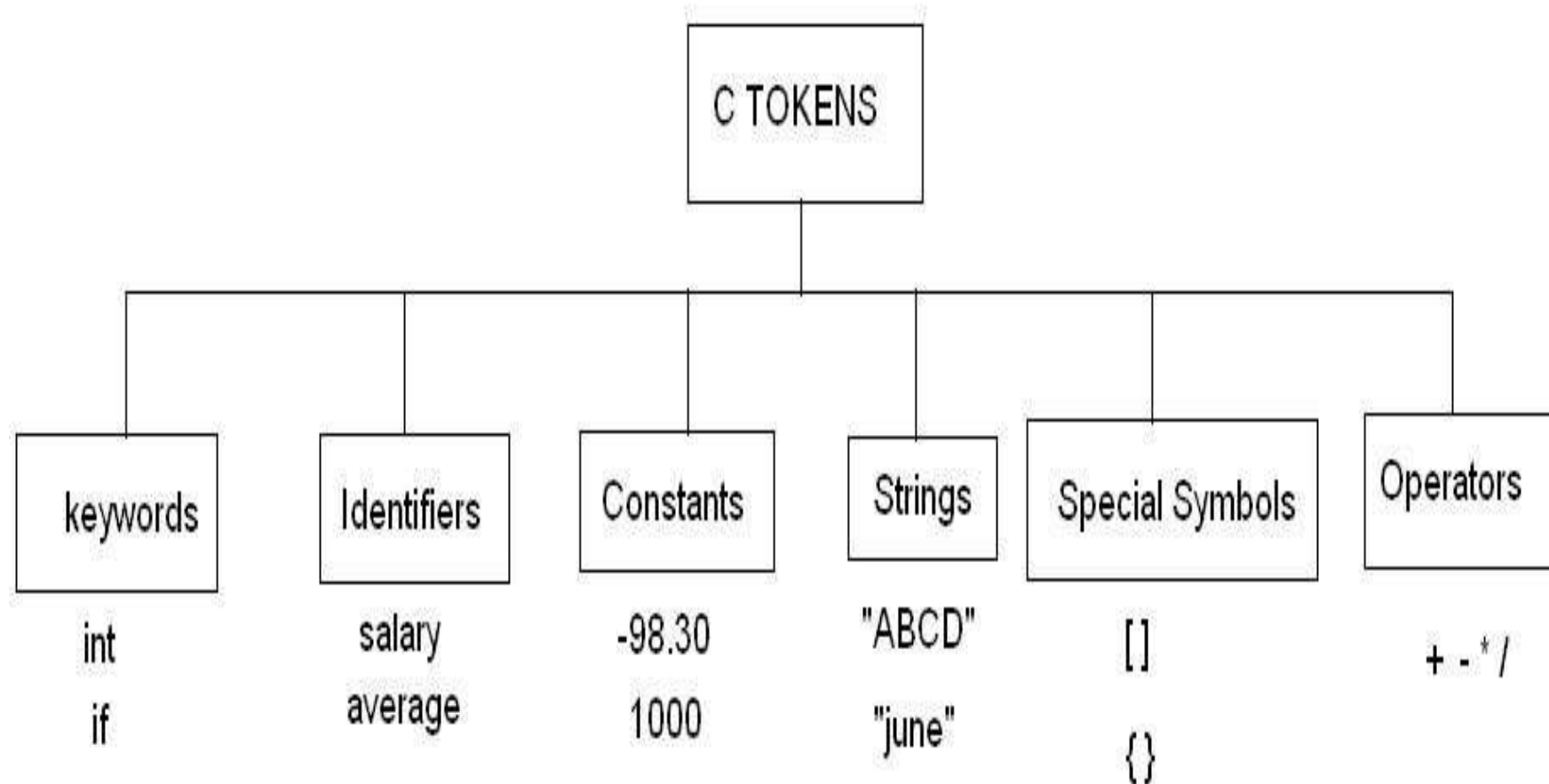
## 2. Line comment

- ➔ To comment a single line use two slashes `//`  

```
int                // Variables declaration & initialization
a=10,b=20,c;       // Adding two numbers
c=a+b;
```

# C Token

- In a passage of text, individual words and punctuation marks are called as tokens.
- The compiler splits the program into individual units, are known as C tokens. C has six types of tokens.



# C Token

- Characters are used to form words, numbers and expressions.
- Characters are categorized as
  - ➔ Letters
  - ➔ Digits
  - ➔ Special characters
  - ➔ White spaces.
- **Letters(26+26):** (Upper Case and Lower Case) A B C  
D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e  
f g h i j k l m n o p q r s t u v w x y z
- **Digits(10):** 0 1 2 3 4 5 6 7 8 9
- **Special Characters(36):**  
' " ( ) \* + - / : = ! & \$ % ; < > ? , . ^ # @ ~ ' { } [ ] \ |
- **White Spaces(5):** Blank Space, Horizontal Space, Carriage Return, New Line.

# A. Identifiers

- Identifiers are **names** given to various programming elements such as variables, constants, and functions.
- It should start with an alphabet or underscore, followed by the combinations of alphabets and digits.
- No special character is allowed except underscore.
- An Identifier can be of arbitrarily long. Some implementation of C recognizes only the first 8 characters and some other recognize first 32 Characters.
- **The following are the rules for writing identifiers in C:**
  - ➔ First character must be alphabetic character or underscore.
  - ➔ Must consist only of alphabetic characters, digits, or underscore.
  - ➔ Should not contain any special character, or white spaces.
  - ➔ Should not be C keywords.
  - ➔ Case matters (that is, upper and lowercase letters). Thus, the names **count** and **Count** refer to two different identifiers.

# A. Identifiers Cont...

Identifier	Legality
Percent	Legal
y2x5__fg7h	Legal
annual profit	Illegal: Contains White space
_1990_tax	Legal but not advised
savings#account	Illegal: Contains the illegal character #
double	Illegal: It is s a C keyword
9winter	Illegal: First character is a digit

# Variables

- Variable is a valid identifier which is used to store the value in the memory location, that value varies during the program execution.

## Types of variables:

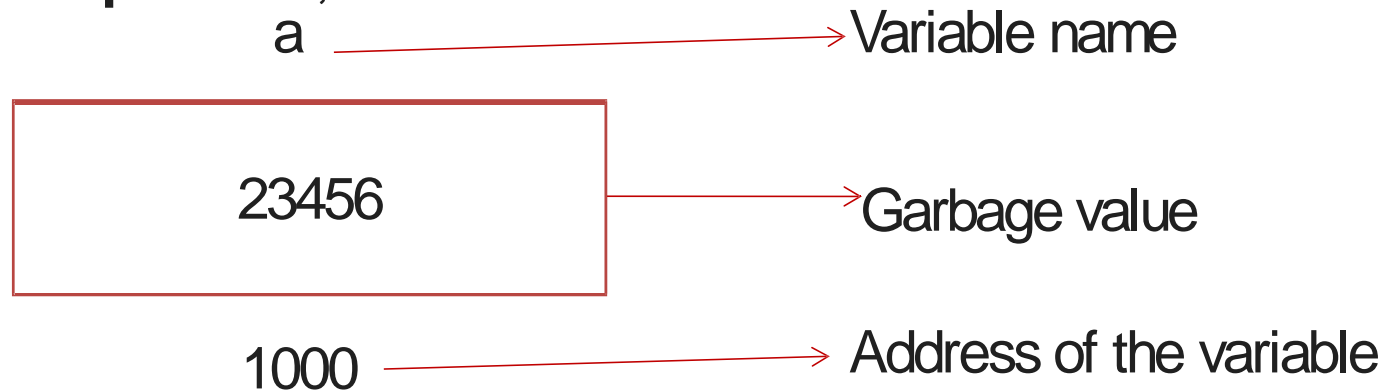
- ➔ Global Variables
- ➔ Local Variables
- **Global Variable:** The variables which are declared at the starting of the program are called as global variable. They are visible to all the parts of the program.
- **Local Variable:** The variables which are declared in a function are called local variables to that function. These variables visible only within the function.
- **Variable Declaration & Definition:**
  - ➔ Each variable in your program must be declared and defined.
  - ➔ In C, a declaration is used to name an object, such as a variable. Definitions are used to create the object.
  - ➔ When you create variables, the declaration gives them a symbolic name and the definition reserves memory for them.
  - ➔ A variable's type can be any of the data types, such as character, integer or real except void.
  - ➔ C allows multiple variables of the same type to be defined in one statement.
  - ➔ **Example:** `int a, b;`

# VariableCont..

- **Variable Initialization:**

- ➔ You can initialize a variable at the same time that you declare it by including an initializer.
- ➔ To initialize a variable when it is defined, the identifier is followed by the assignment operator and then the initializer.
- ➔ **Example:** `int count = 0;`

**Variable declaration and definition:** **Example:** `int a;`



## Variable initialization:

`datatype identifier = initial value;`

### Examples:

```
int a=10;  
float b=2.1;  
float pi=3.14;  
char ch='A';
```

- When you want to process some information, you can save the values temporarily in variables.

# Variables Cont...

**There are some restrictions on the variable names (same as identifiers):**

- ➔ First character must be alphabetic character or underscore.
  - ➔ Must consist only of alphabetic characters, digits, or underscore.
  - ➔ Should not contain any special character, or white spaces.
  - ➔ Should not be C keywords.
  - ➔ Case matters (that is, upper and lowercase letters).
  - ➔ Thus, the names **count** and **Count** refer to two different identifiers.
- **Note:** Variables must be declared before they are used, usually at the beginning of the function.

Variable Name	Legality
annual_profit	Legal
_1990_tax	Legal but not advised
savings#account	Illegal: Contains the illegal character #
double	Illegal: It is a C keyword

```
#include<stdio.h>
main()
{
int
a=10,b=20,c;
c=a+b;
printf("sum of a and b=%d\n",c);
return 0;
}
```

# B.Keywords

- C word is classified as either **keywords** or **identifiers**.
- Keyword are **reserved** words or **predefined** words
- Keywords have fixed meanings, these meanings cannot be changed.
- Keywords must be in **lowercase**.
- There are 32 keywords in C.

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

# C.Constants

- Constants are data values that cannot be changed during the program execution.
- Like variables, constants have a type.
- **Types of constants:**
  - ➔ Boolean constants:
    - A Boolean data type can take only two values **true** and **false**.
  - ➔ Character constants
    - Single character constants
    - string constants.
  - ➔ Numeric constants.
    - integer constant
    - real constants.
- **Type qualifier `const`**
  - One way to use the constant is with memory constants. Memory constants use a C type qualifier; **const**.
  - This indicates that the data cannot be changed.
  - ➔ **`const type identifier= value;`**
  - ➔ **`const float pi=3.14;`**

# D. Strings

- **Single character constants**

- A **single character constants** are enclosed in **single quotes**.

- Example: '1' 'X' '%' ' '

- Character constants have integer values called **ASCII** values.

```
printf("%d",ch);  
char ch='A';
```

```
similarly printf("%c",65)
```

**Output: 65**

**Output: A**

- **String Constants**

- **String** is a collection of characters or sequence of characters enclosed in **double quotes**.

- The characters may be letters, numbers, special characters and blank space.

- Example: "JNTUH" "2011" "A".

# Backslash \ escape characters

- **Backslash characters** are used in **output** functions.
- These backslash characters are preceded with the \ symbol.

- **Numeric Constants**

- ➔ **integer constant:**

- It is a sequence of digits that consists numbers from 0 to 9.

- ➔ **Example:**      23      -678      0      +78

## Rules:

1. integer constant have at least one digit.
2. No decimal points.
3. No commas or blanks are allowed.
4. The allowable range for integer constant is **-32768 to 32767**.

constant	meaning
'\a'	Alert(bell)
'\b'	Back space
'\f'	Form feed
'\n'	New line
'\r'	Carriage return
'\v'	Vertical tab
'\t'	Horizontal tab
'\'	Single quote
'\"'	Double quotes
'\?'	Question mark
'\\'	Backslash
'\0'	null

# Constants Cont...

➔ **Real constants:**

➔ The numbers containing fractional parts like 3.14

➔ **Example:**            1.9099            -0.89            +3.14

➔ Real constants are also expressed in exponential notation.

Mantissa **e** exponent

- A number is written as the combination of the mantissa, which is followed by the prefix **e** or **E**, and the exponent.

➔ **Example:**

87000000	=	8.7e7
- 550	=	-5.5e2
0.000000000031	=	3.1e-10.

Examples of real constants

Type	Representation	Value
double	0.	0.0
double	0.0	.0
float	-2.0f	-2.0
long double	3.14159276544L	3.14159276544

# Constants Cont...

## → **Coding Constants:**

→ Different ways to create constants.

## → **Literal constants:**

- A literal is an unnamed constant used to specify data.
- **Example:**      `a = b + 5;`

## → **Defined constants:**

- By using the preprocessor command you can create a constant.
- **Example:**      `#define pi 3.14`

## → **Memory constants:**

- Memory constants use a C type qualifier, `const`, to indicate that the data can not be changed.
- Its format is: `const type identifier = value;`
- **Example:**              `const float PI = 3.14159;`

# E.Operators

- C supports a rich set of operators.
- An **operator** is a symbol that tells the computer to perform mathematical or logical operations.
- Operators are used in C to **operate on data and variables**.

expression

X=Y+Z

Operators: =, +

Operands: x, y, z

- **Unary operators** are used on a **single operand** (- -, +, ++, --)
- **Binary operators** are used to apply in between **two operands** (+, -, /, \*, %)
- Conditional (or **ternary**) operator can be applied on **three operands**. (? : )

# Operators Cont...

- **Types of Operators:**
- C operators can be classified into a number of categories.
- They include:
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Assignment Operator
  - Increment and Decrement Operators
  - Conditional Operators
  - Bitwise Operators
  - Special Operators

# 1.Arithmetic Operators

- Arithmetic operators are used for **mathematical calculation**.

Operator	Meaning	Example	Description
+	Addition	$a + b$	Addition of a and b
-	Subtraction	$a - b$	Subtraction of b from a
*	Multiplication	$a * b$	Multiplication of a and b
/	Division	$a / b$	Division of a by b
%	Modulo division- remainder	$a \% b$	Modulo of a by b

**Syntax:** **operand1** arithmetic operator **operand2**

→ **Examples:**

$10 + 10 = 20$	(addition on integer numbers)
$10.0 + 10.0 = 20.0$	(addition on real numbers)
$10 + 10.0 = 20.0$	(mixed mode)
$14 / 3 = 4$	(ignores fractional part)

## 2.Relational Operators

- Relational operators are used to **compare two numbers and taking decisions** based on their relation.
- The value of a relational expression is either one or zero.
- It is **one** if the specified relation is **true** and **zero** if the relation is **false**.
- Relational operators are used by **if** , **while** and **for** statements.

Operator	Meaning	Example	Description
<	Is less than	a < b	a is less than b
<=	Is less than or equal to	a <= b	a is less than or equal to b
>	Is greater than	a > b	a is greater than b
>=	Is greater than or equal to	a >= b	a is greater than or equal to b
=	Is equal to	a = b	a is equal to b
!=	Is not equal to	a != b	a is not equal to b

**Syntax:**    **operand1** relational operator **operand2**

# 3. Logical Operators

- Logical operators are used to **test more than one condition** and make decisions.
- Yields a value either one or zero.

Operator	Meaning
&&	Logical AND (true only if both the operands are true)
	Logical OR (true if either one operand is true)
!	Logical NOT (negate the operand)

**Syntax:** **operand1** logical operator **operand2**  
or  
logical operator **operand**

**Example:**  $(x < y) \&\& (x == 8)$

a	b	a&&b	a    b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

a	!a
0	1
1	0

# 4. Assignment Operators

➤ Assignment operators are used to assign the result of an expression to a variable.

➤ Assignment Operator is **=**

**Syntax:**     **variable = expression;**

➤ **Types of assignment:**

➤ Single Assignment

Ex:                    a = 10;

➤ Multiple Assignment

Ex:                    a=b=c=0;

➤ Compound Assignment

Ex:                    c = a +

b;

Operator	Meaning	Example	Equivalent
+=	Addition with assignment	a +=5	a= a+5
-=	Subtraction with assignment	a -=6	a=a-6
*=	Multiplication with assignment	a *=5	a=a*5
/=	Division with assignment	a /=5	a=a/5
%=	Remainder with assignment	a %=5	a=a%5

**operand1** arithmetic assignment operator **operand2**

# 5. Increment and Decrement Operators

- We can add or subtract 1 to or from variables by using **increment (++)** and **decrement (--)** operators.
- The operator ++ **adds 1** to the operand and the operator -- **subtracts 1**.
- They can apply in two ways: **postfix** and **prefix**.
- **Syntax:** **increment or decrement operator operand**  
**operand increment or decrement operator**
- **Prefix form:** Variable is changed before expression is evaluated
- **Postfix form:** Variable is changed after expression is evaluated.

Operator	Meaning	Example	Equivalent	
++	Prefix or Pre Increment	++i	i=i+1;	i+=1
++	Postfix or Post Increment	i++	i=i+1;	i+=1
--	Prefix or Pre Decrement	--i	i=i-1;	i-=1
--	Postfix or Post Decrement	i--	i=i-1;	i-=1

**Syntax:** **operand1** arithmetic assignment operator **operand2**

# 5. Increment and Decrement Operators

Operator	Description
Pre increment operator (++x)	value of x is incremented before assigning it to the variable on the left

## Example

```
x=10;  
p=++x;
```

## Explanation

First increment value of x by one then assign.

## Output

```
x will be 11  
p will be 11
```

Operator	Description
Post increment operator (x++)	value of x is incremented after assigning it to the variable on the left

## Example

```
x=10;  
p=x++;
```

## Explanation

First assign value of x then increment value.

## Output

```
x will be 11  
p will be 10
```

## 6. Conditional (ternary) Operators ( ?: )

- C's only conditional (or **ternary**) operator requires three operands.

**Syntax:**     **conditional expression? expression1: expression2;**

- The conditional expression is any expression that results in a true (nonzero) or false (zero).
- If the result is true then expression1 executes, otherwise expression2 executes.
- **Example:**     a=1; b=2;  
                     x = (a<b)?a:b;

This is like

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

# 7. Bitwise Operators

- C has a special operator known as Bitwise operator for manipulation of data at bit level.
- Bitwise operator may not be applied for float and double. Manipulates the data which is in binary form.
- **Syntax:**      **operand1 bitwise operator operand2**

Operator	Meaning	Example
&	bitwise AND	a & b
	bitwise OR	a   b
^	bitwise exclusive OR	a ^ b
<<	shift left (shift left means multiply by 2)	a<< 2
>>	shift right (shift right means divide by 2)	a>>2

A	B	A&B	A B	A^B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- **Examples:**

&	Bitwise AND	0110 &	0011	➔	0010
	Bitwise OR	0110	0011	➔	0111
^	Bitwise XOR	0110	^ 0011	➔	0101
<<	Left shift	01101110	<< 2	➔	10111000
>>	Right shift	01101110	>> 3	➔	00001101
~	One's complement	~0011		➔	1100

# 7. Bitwise Operators Cont...

## ➤ Shift right:

➤ **>>** is a binary operator that requires two integral operands. the first one is value to be shifted, the second one specifies number of bits to be shifted.

➤ The general form is as follows:

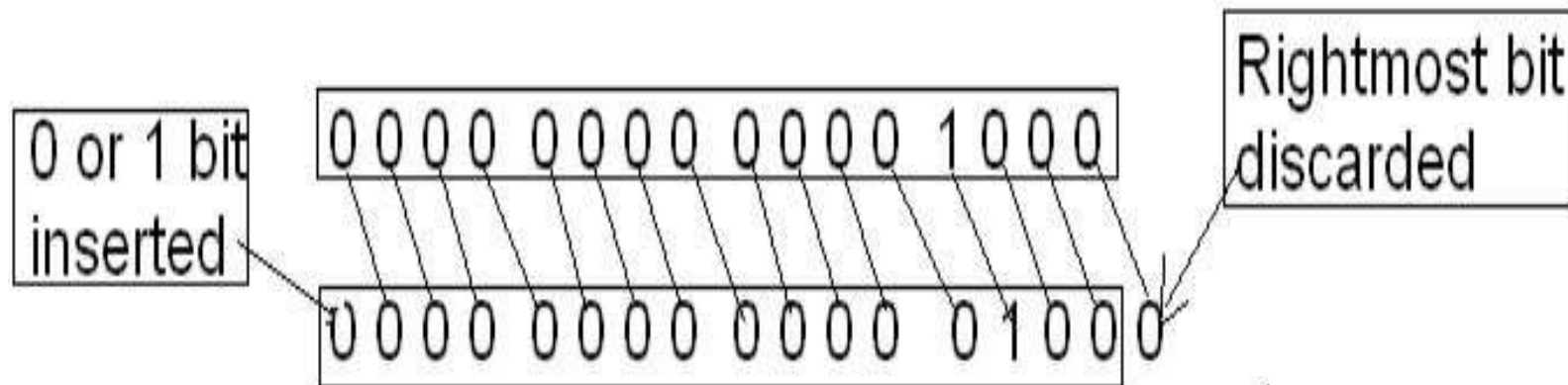
**variable >> expression;**

➤ When bits are shifted right, the bits at the rightmost end are deleted.

➤ Shift right operator divides by a power of 2. I.e.  $a \gg n$  results in  $a/2^n$ , where  $n$  is number of bits to be shifted.

`a=8; b=a>>1;`

**Example:**



# 7. Bitwise Operators Cont...

## ➤ Shift left:

➤ `<<` is a binary operator that requires two integral operands. the first one is value to be shifted, the second one specifies number of bits to be shifted.

➤ The general form is as follows:

**variable << expression;**

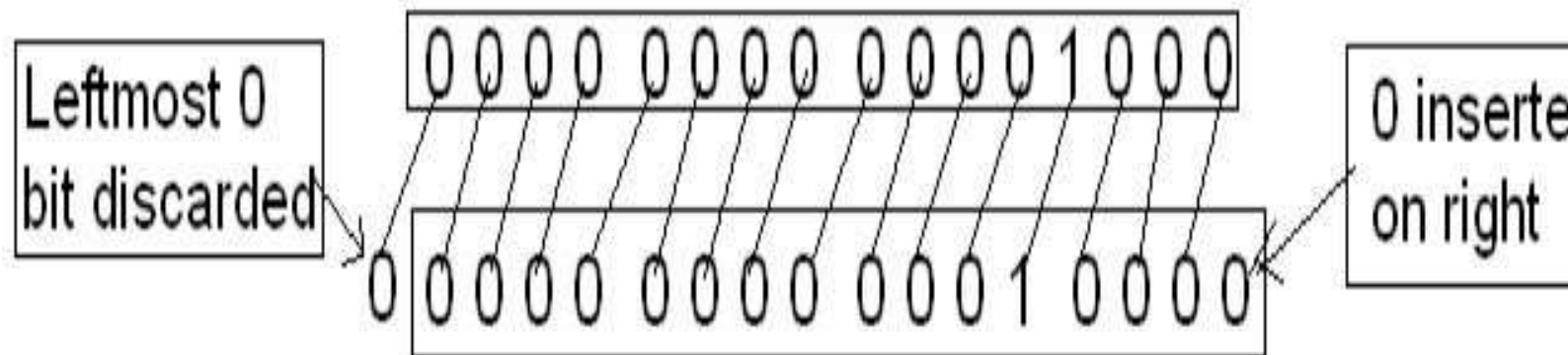
➤ When bits are shifted left, the bits at the leftmost end are deleted.

**Examp<sub>l</sub>**

`a=8;`

**e:** `b=a<<1; // assigns 16 after left shift operation`

➤ Shift left operator multiply by a power of 2, `a<<n` results in  $a*2^n$ , where **n** is number of bits to be shifted.



# 8.Special Operators

Operator	Meaning
&	Address operator, it is used to determine address of the variable.
*	Pointer operator, it is used to declare pointer variable and to get value from it.
,	Comma operator. It is used to link the related expressions together.
sizeof	It returns the number of bytes the operand occupies.
.	member selection operator, used in structure.
->	member selection operator, used in pointer to structure.

## comma operator :

- It doesn't operate on data but allows more than one expression to appear on the same line.

**Example:**     `int i = 10, j = 20;     printf ("%d %.2f %c", a,f,c);`  
                                  `j = (i = 12, i + 8);` //i is assigned 12 added to 8 produces 20

## sizeof Operator :

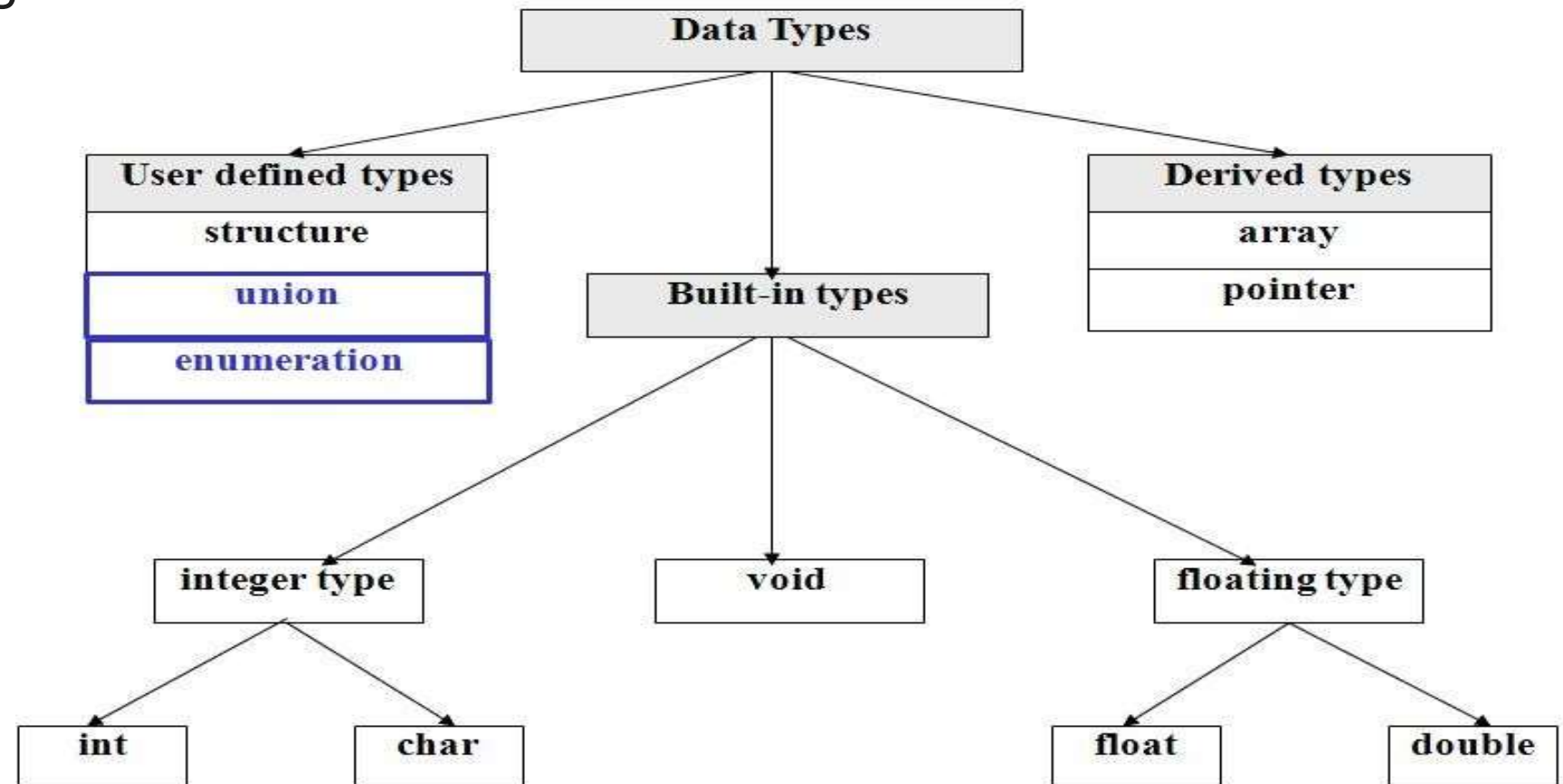
- It is a unary operator (operates on a single value).
- Produces a result that represent the size in bytes.

**Syntax:**                   **sizeof(datatype);**

**Example:**               `int a = 5;               sizeof (a);               //produces 2`  
                                  `sizeof(char);               // produces 1`  
                                  `sizeof(int);                // produces 2`

# Data types

- Data types are used to indicate the **type** of value represented or stored in a variable, the **number of bytes** to be reserved in memory, the **range of values** that can be represented in memory, and the **type of operation** that can be performed on a particular data value.
- ANSI C supports 3 categories of data types:
  - ➔ Built-in data types
  - ➔ Derived data types
  - ➔ User Defined data types



# Data types Cont...

- **Built-in data types:**

- ➔ Built-in data types are also known as primitive data types. C uses the
- ➔ following primitive data types.

<b>int</b>	integer quantity
<b>char</b>	character (stores a single character)
<b>float</b>	floating point number
<b>double</b>	floating point number

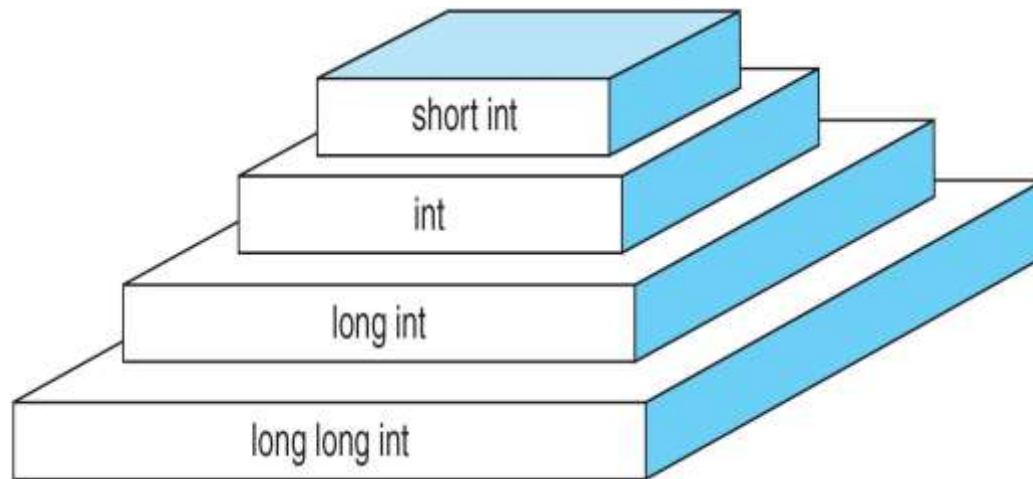
- **1.Integer data type:**

- ➔ An **integer number** (also called whole number) has no fractional part or decimal point.
- ➔ The keyword **int** is used to specify an integer variable.
- ➔ It occupies 2 bytes (16 bits) or 4 bytes (32 bits), depending on the machine architecture.
- ➔ 16-bit integer can have values in the range of **-32768 to 32767**
- ➔ One bit is used for sign.

# Data types Cont...

- **2.void data type:**

➔ Defines an empty data type which can then be associated with some data types. It is useful with pointers.



**Note**

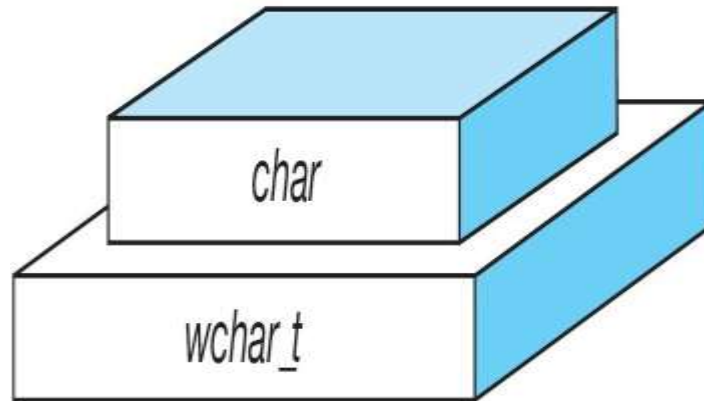
**$\text{sizeof (short)} \leq \text{sizeof (int)} \leq \text{sizeof (long)} \leq \text{sizeof (long long)}$**

**Integer  
Types**

# Data types Cont...

- **3.Character Data Type :**

- ➔ The shortest data type is character.
- ➔ The keyword `char` is used to declare a variable of a character type.
- ➔ It is stored in 1 byte in memory.
- ➔ Corresponding integer values for all characters are defined in ASCII (American Standard Code for Information Interchange).
- ➔ **Example:** character constant 'a' has an int value 97, 'b' has 98, 'A' has 65 etc.
- ➔ Character can have values in the range of **-128 to 127**.



**Character Types**

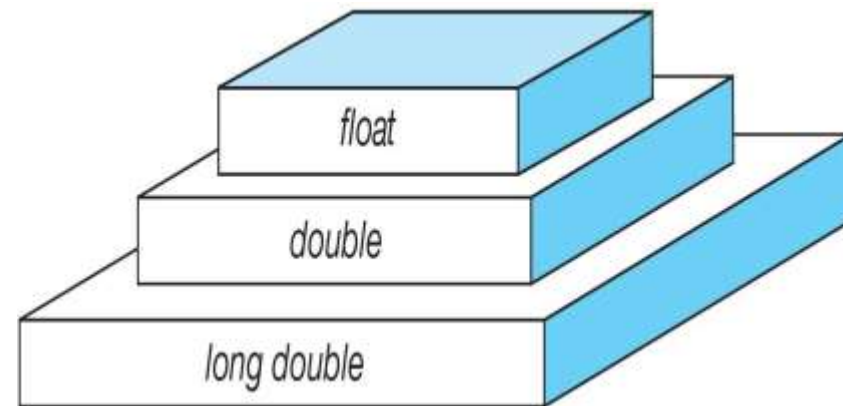
# Data types Cont...

- **4. floating point data type:**

- ➔ The keyword **float** is used to declare a variable of the type float.
- ➔ The float type variable is usually stored in 32 bits, with 6 digits of precision.
- ➔ A float variable can have values in the range of **3.4E-38** to **3.4 E+38**.

- **5. double data type:**

- ➔ A floating point number can also be represented by the double data type.
- ➔ The data type double is stored on most machines in 64 bits which is about 15 decimal places of accuracy.
- ➔ To declare a variable of the type double, use the keyword **double**.
- ➔ A double variable can have values in the range of **1.7E-308** to **+1.7E+308**.



*Note*

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

# Data types Cont...

- **Derived data types and User defined data types:**

- These are the combination of primitive data types. They are used to represent a collection of data.

- **They are:**

- Arrays
- Pointers
- Structures
- Unions
- Enumeration

**Note:** Number of bytes and range given to each data type is platform dependent.

- **Type Modifiers:**

- The basic data types may have various **modifiers** (or **qualifiers**) preceding them, except type 'void'.

- A **modifier** is used to alter the meaning of the base data type to fit the needs of various situations more precisely.

- The modifiers **signed, unsigned, long, short** may be applied to **integer** base types.

- The modifiers **unsigned and signed** may be applied to **characters**. The modifier **long** may also be applied to **double**.

- The difference between signed and unsigned integers is in the way high-order bit (sign bit) of the integer is interpreted.

- If sign bit is 0, then the number is positive; if it is 1, then the number is negative.

# Data types Cont...

type	size (byte )	smalle st numb er	large st numb er	precision	constant	forma t specifi er
long double	10	3.4E -4932	1.1E+4932	1.08E -19	1.2345L, 1.234E -5L	%Lf, %Le, %Lg
double	8	1.7E -308	1.7E+308	2.22E -16	1.2345, 1.234E -5	%lf, %le, %lg
float	4	1.7E -38	3.4E+38	1.19E -7	1.2345F, 1.234E -5F	%f, %e, %g
unsigned long	4	0	4294967295		123UL	%lu
long	4	-2147483648	2147483647		123L	%ld, %li
unsigned	4	0	4294967295		123U	%u
int	4	-2147483648	2147483647		123	%d, %i
unsigned short	2	0	65535		123U	%hu
short	2	-32768	32767		123	%hd, %hi
unsigned char	1	0	255		'a', 123, ' \n'	%c
char	1	-128 or 0	127 or 255		'a', 123, ' \n'	%c

# Precedence and Associativity

- **Precedence** is used to determine the order in which different operators in a complex expression are evaluated.
- **Associativity** is used to determine the order in which operators with the same precedence are valuated in a complex expression.
- Every operator has a precedence.
- The operators which has higher precedence in the expression is evaluated first.

**Example:**

$a=8+4*2;$

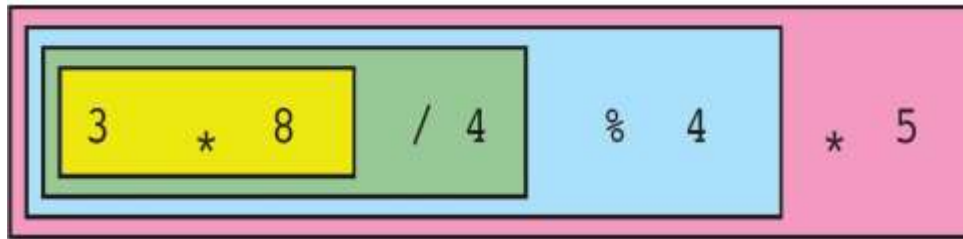
$a=?$

# Precedence and Associativity

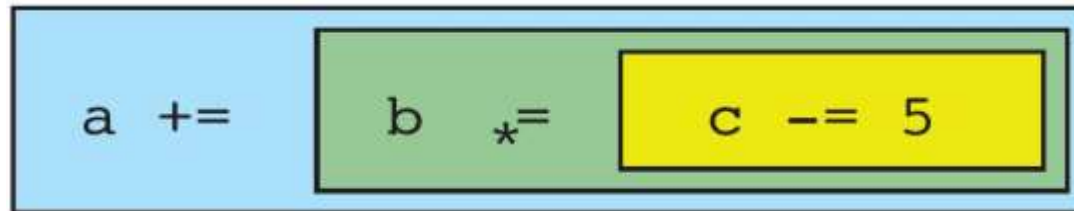
Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary, prefix	+ - ! ~ ++ -- (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	? :	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma	,	Left to right

# Expression Evaluation

- A side effect is an action that results from the evaluation of an expression.
- For example, in an assignment, C first evaluates the expression on the right of the assignment operator and then places the value in the left variable.
- Changing the value of the left variable is a side effect.

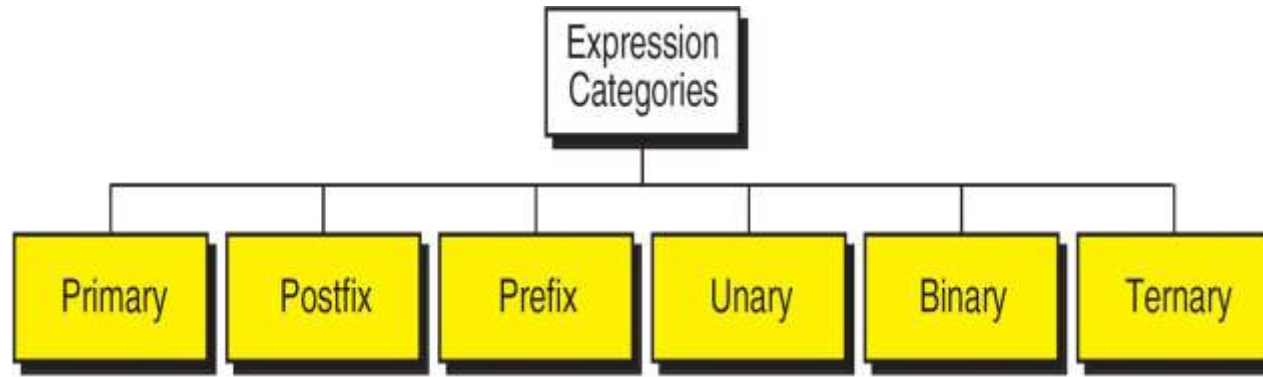


**Left-to-Right Associativity**



**Right-to-Left Associativity**

# Expression



## ➤ Expression

- ➔ An **expression** is a sequence of operands and operators that reduces to a single value.
- ➔ Expressions can be simple or complex.
- ➔ An **operator** is a syntactical token that requires an action be taken.
- ➔ An **operand** is an object on which an operation is performed; it receives an operator's action.

## ➤ Primary Expression:

- ➔ The most elementary type of expression is a primary expression.
- ➔ It consists of **only one operand with no operator**.
- ➔ In C, the operand in the primary expression can be a **name**, a **constant**, or a **parenthesized expression**.
- ➔ Name is any identifier for a variable, a function, or any other object in the language.
- ➔ The following are examples of primary expressions:
  - ➔ **Example:** a            price            sum            max
- ➔ Literal Constants is a piece of data whose value can't change during the execution of the program.

# Expression

- The following are examples of literal constants used in primary expression:

**Example:** 'A'      56      98      12.34

- Any value enclosed in parentheses must be reduced in a single value is called as primary expression.
- The following are example of parentheses expression:

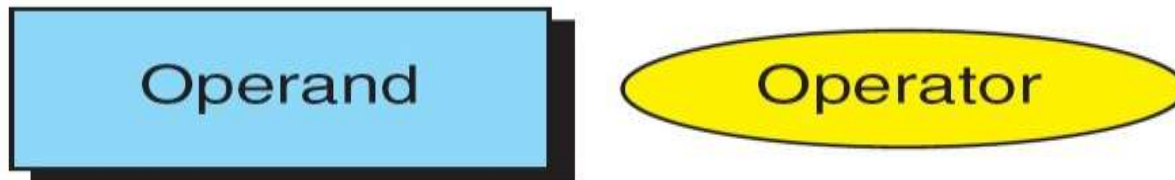
**Example:** (a\*x + b)      (a-b\*c)      (x+90)

## Post fix expression:

- It is an expression which contains **operand** followed by one **operator**. a--;

**Example:** a++;

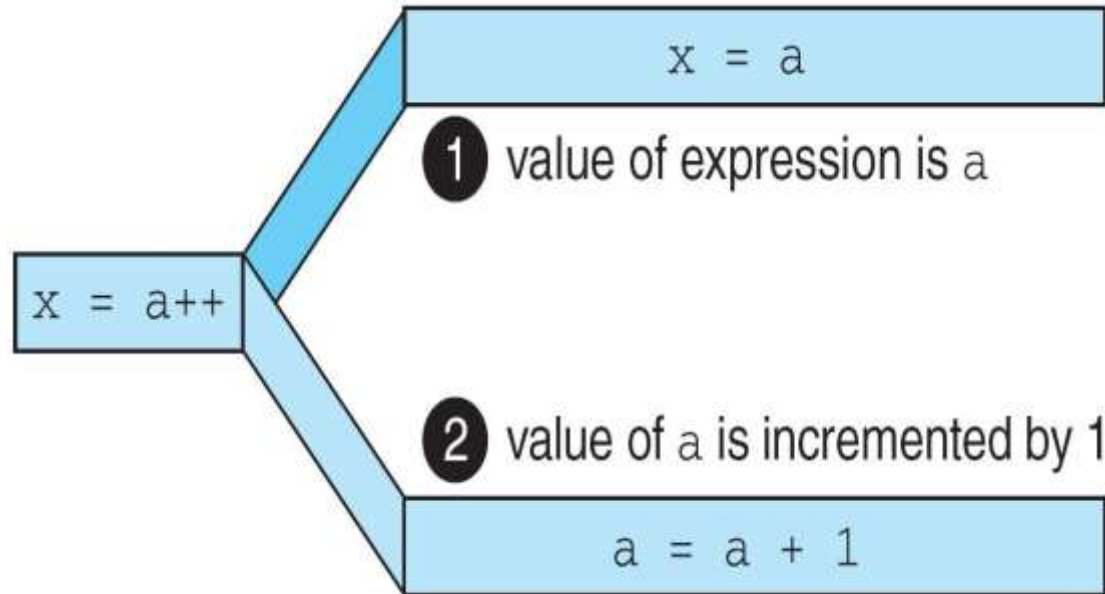
- The operand in a postfix expression must be a variable.
- (a++) has the same effect as (a = a + 1)
- If ++ is after the operand, as in a++, the increment takes place **after** the expression is evaluated.



# Expression

In the following figure:

1. Value of the variable `a` is assigned to `x`
2. Value of the `a` is incremented by 1.



```
#include<stdio.h>

void main()

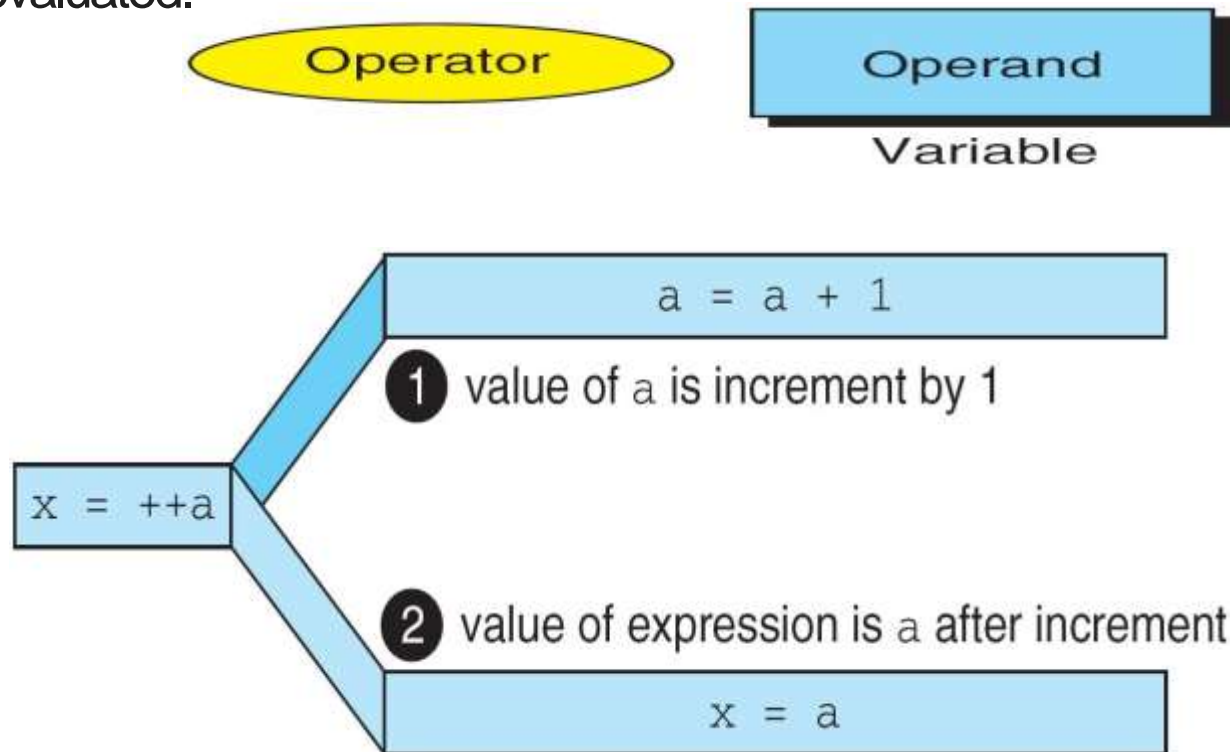
{
    a=10;
    x=a++;
    printf("x=%d, a=%d",x,a);
}
```

Output:  
x=10, a=11

# Expression

## Pre fix expression:

- It is an expression which contains **operator** followed by an **operand**.
- **Example:**            `++a;`                    `--a;`
- The operand of a prefix expression must be a variable.
- `(++a)` has the same effect as `(a = a + 1)`
- If `++` is before the operand, as in `++a`, the increment takes place **before** the expression is evaluated.



```
#include<stdio.h>

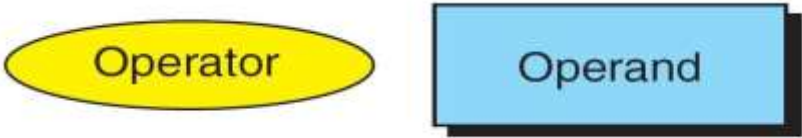
void main()
{
    a=10;
    x=++a;
    printf("x=%d, a=%d",x,a);
}
```

Output:  
x=11, a=11

# Expression

## Unary expression:

- It is an expression which consists of unary operator followed by the operand



Expression	Contents of a Before and After Expression	Expression Value
+a	3	+3
-a	3	-3
+a	-5	-5
-a	-5	+5

## Binary Expressions:

- In binary expression **operator** must be placed in between the **two operands**.
- Both operands of the modulo operator (%) must be integral types.



# Type conversion

- Up to this point, we have assumed that all of our expressions involved data of the same type.
- But, what happens when we write an expression that involves two different data types, such as multiplying an integer and a floating-point number?
- To perform these evaluations, one of the types must be converted.
- **Type Conversion:** Conversion of one data type to another data type.
- Type conversions are classified into:
  - ↳ Implicit Type Conversion
  - ↳ Explicit Type Conversion (Cast)
- **Implicit Conversion:**
  - In implicit type conversion, if the operands of an expression are of different types, the lower data type is automatically converted to the higher data type before the operation evaluation.
  - The result of the expression will be of higher data type.
  - The final result of an expression is converted to the type of the variable on the LHS of the assignment statement, before assigning the value to it.
  - Conversion during assignments:

```
char c = 'a'; int i;  
i = c; /* i is assigned by the ascii of 'a' */
```

# Type Conversion Cont...

- Arithmetic Conversion: If two operands of a binary operator are not the same type, **implicit** conversion occurs:

```
int i = 5 , j = 1;
```

```
float x = 1.0, y;
```

```
y = x / i;          /* y = 1.0 / 5.0 */
```

```
y = j / i;          /* y = 1 / 5 so y = 0 */
```

## Explicit Conversion or Type Casting:

- In explicit type conversion, the user has to enforce the compiler to convert one data type to another data type by using typecasting operator.

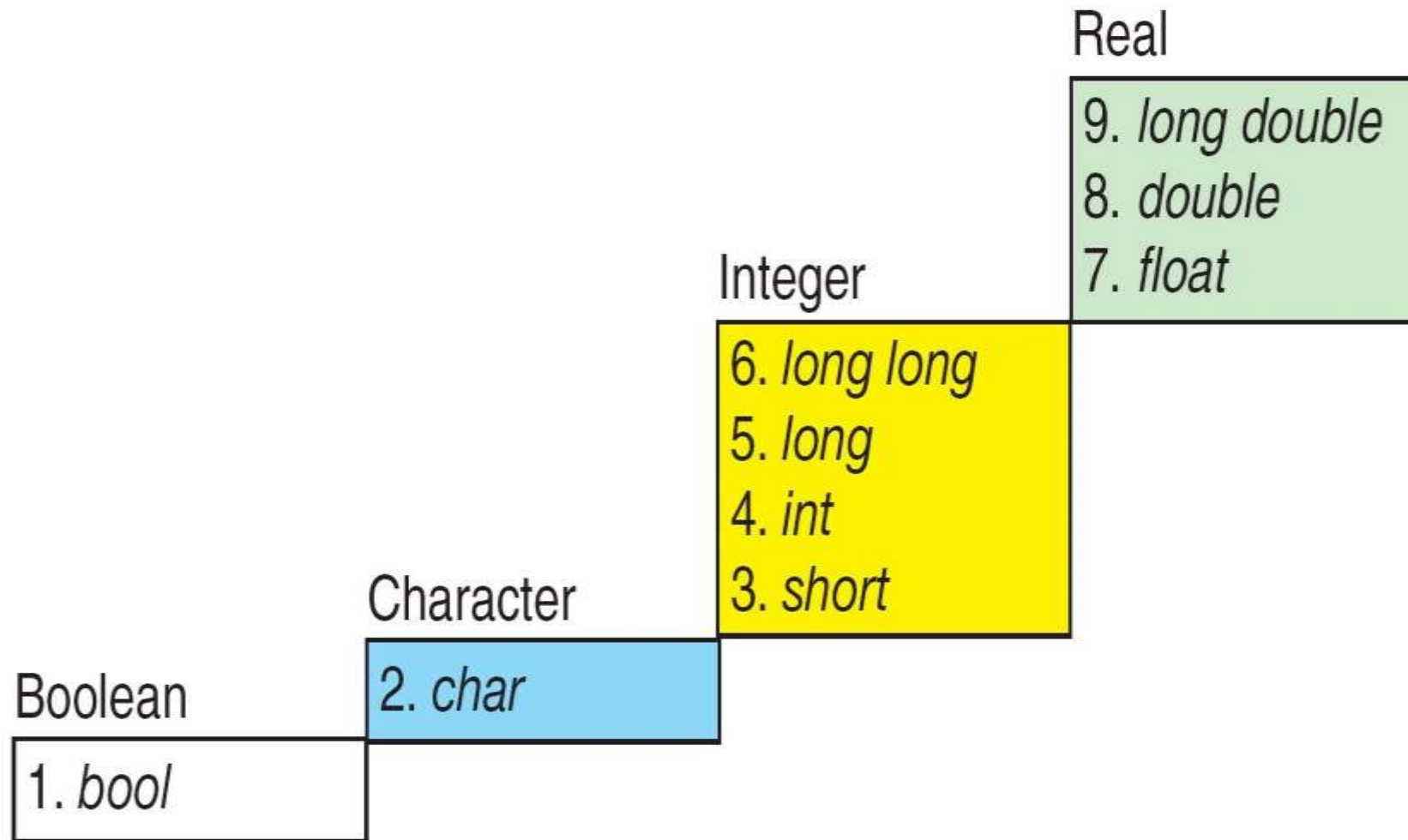
- This method of typecasting is done by prefixing the variable name with the **data type enclosed within parenthesis**.

**(data type) expression**

- Where **(data type)** can be any valid C data type and expression is any variable, constant or a combination of both.

- **Example:**     int x;  
                  x=(int)7.5;

# Type Conversion Cont...



**Conversion Rank (C Promotion Rules)**

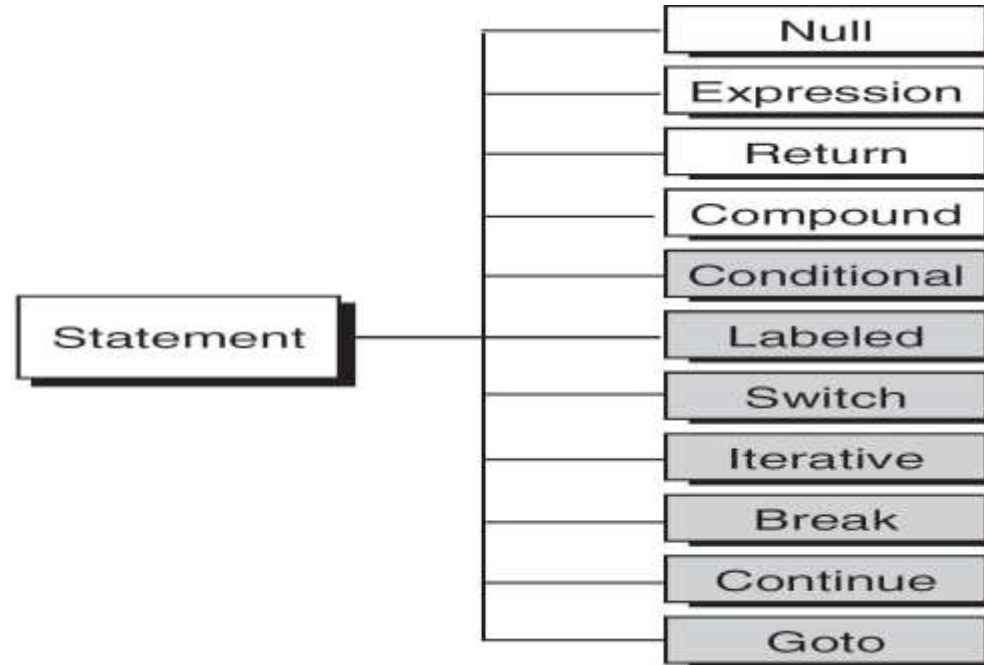
# Conditional branching and Loops



# Computer

## ➤ Statements

- ➔ A statement causes an action to be performed by the program.
- ➔ It translates directly into one or more executable computer instructions.
- ➔ Generally statement is ended with semicolon.
- ➔ Most statements need a semicolon at the end; some do not. Compound statements are used to group the statements into a single executable unit.
- ➔ It consists of one or more individual statements enclosed within the braces { }



# Decision Control Structures

- The decision is described to the computer as a conditional statement that can be answered either **true** or **false**.
- If the answer is true, one or more action statements are executed.
- If the answer is false, then a different action or set of actions is executed.
- **Types of decision control structures:**
  - if
  - if..else
  - nested if...else
  - else if ladder
  - dangling else
  - switch statement

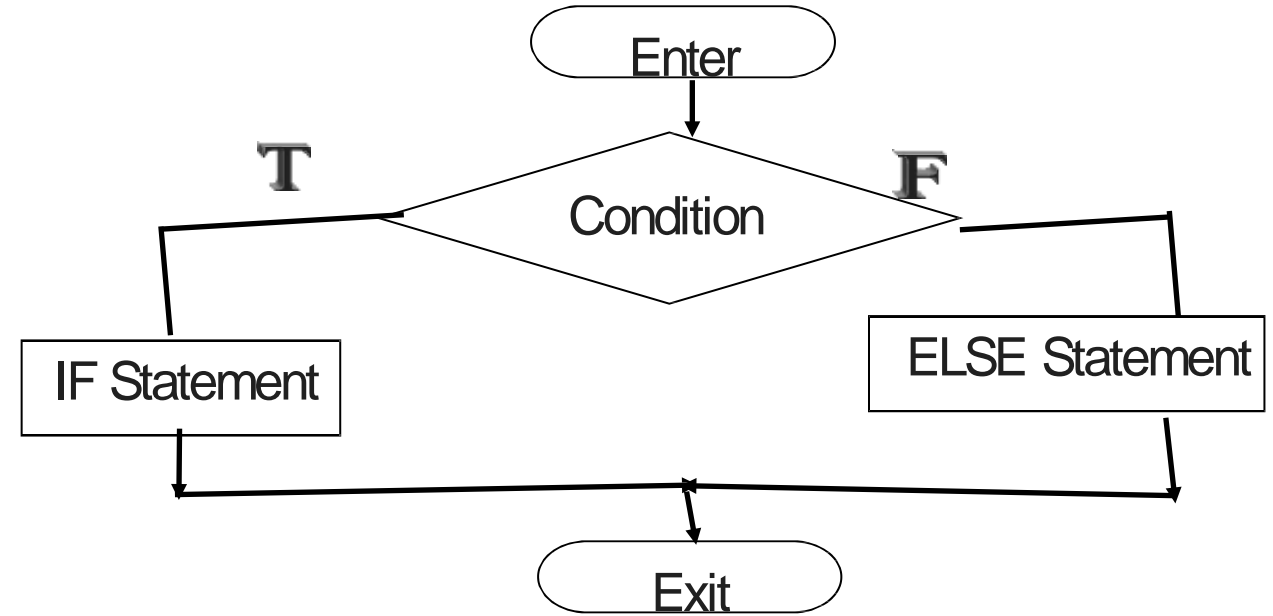
# Decision Control Statement: **if..else**

- The general form of a simple if statement is:

```
if (condition)
{
    statement-block;
}
else
{
    Statement-block;
}
```

- Rules:

- ➔ The expression or condition which is followed by **if** statement must be enclosed in **parenthesis**.
- ➔ No **semicolon** is needed for an **if...else** statement.
- ➔ Both the true and false statements can be any statement (even another if...else)
- ➔ Multiple statements under **if** and **else** should be enclosed between curly braces.
- ➔ No need to enclose a single statement in curly braces.



## Example:

```
main()
{
    int a=10,b=20;
    if(a>b)
    {printf("%d",a);}
    else
    {printf("%d",b);}
}
```

# Decision Control Statement: **else if Ladder**

- **Rules:**

- ➔ The conditions are evaluated from the top to down.
- ➔ As soon as a true condition is found the statement associated with it is executed and the control is transferred to the statement x by skipping the rest of the ladder.
- ➔ When all **n** conditions become false, final else containing default\_statement that will be executed

```
if (condition1)
    statements1;
else if (condition2)
    statements2;
    else if (condition3)
        statements3;
        else if (condition4)
            statements4;
            .....
            else if(conditionn)

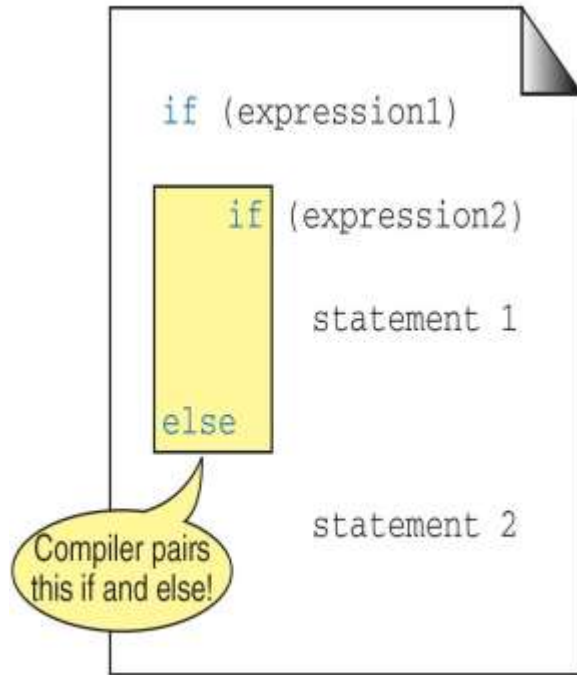
statementsn;

else

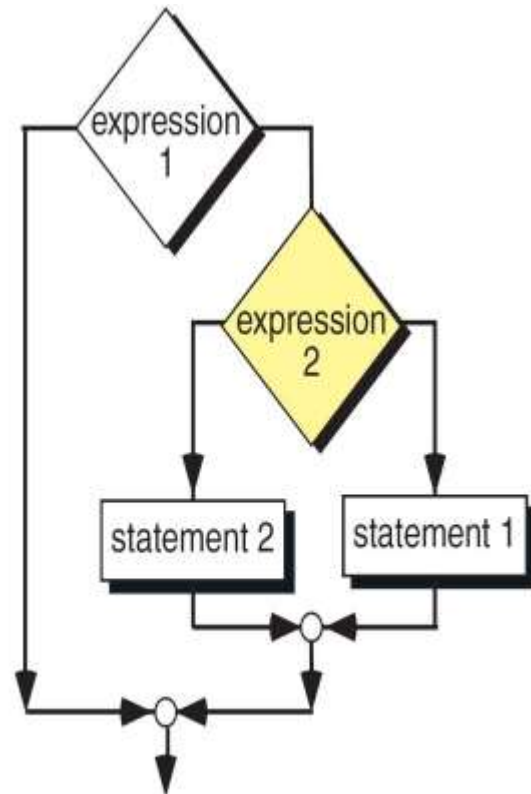
default_statement;
statement x;
```

# Dangling else

- **else** is always paired with the most recent unpaired **if**.



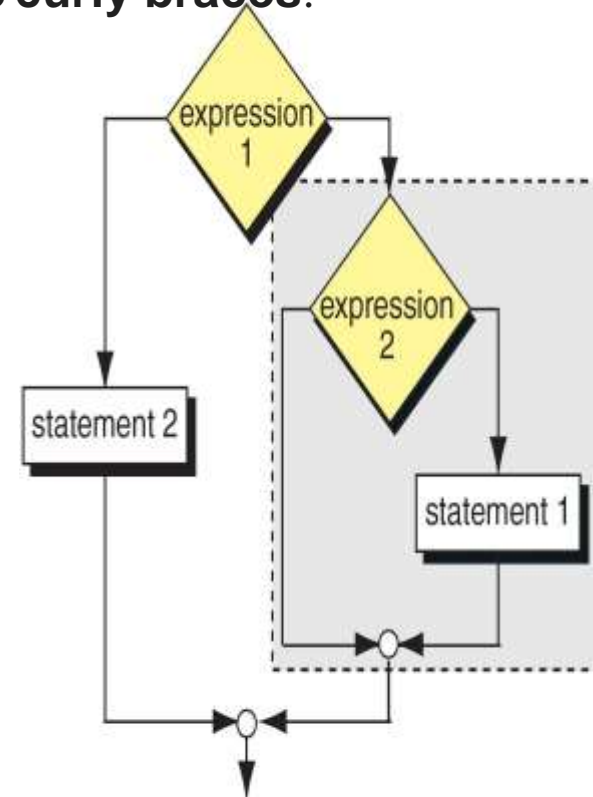
(a) Code



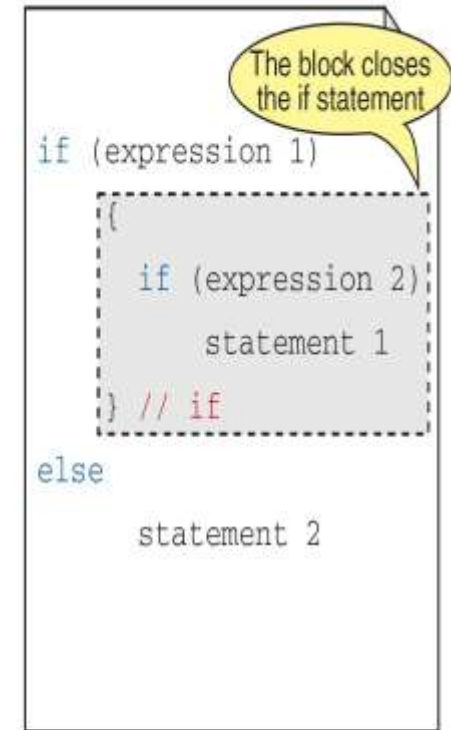
(b) Logic Flow

**Dangling else**

- To avoid **dangling else** problem place the **inner if statement** with in the **curly braces**.



(a) Logic Flow



(b) Code

**Dangling else Solution**

# Decision Control Statement: **switch**

- ➔ It is a **multi-way** conditional statement generalizing the **if...else** statement.
- ➔ It is a conditional control statement that allows some particular **group of statements to be chosen** from several available groups.
- ➔ A switch statement allows a single variable to be compared with several possible **case** labels, which are represented by constant values.
- ➔ If the variable matches with one of the constants, then an execution jump is made to that point.
- ➔ A case label cannot appear more than once and there can only be one default expression.
- ➔ Note: **switch** statement does not allow less than ( < ), greater than ( > ).
- ➔ ONLY the equality operator (==) is used with a switch statement.
- ➔ The control variable must be integral (int or char) only.
- ➔ When the switch statement is encountered, the control variable is evaluated.
- ➔ Then, if that evaluated value is equal to any of the values specified in a **case** clause, the statements immediately following the colon (":") begin to run.
- ➔ **Default case** is optional and if specified, default statements will be executed, if there is no match for the case labels.
- ➔ Once the program flow enters a **case** label, the statements associated with **case** have been executed, the program flow continues with the statement for the next case. (if there is no **break** statement after case label.)

# Decision Control Statement: **switch**

- **General format of switch:**

- ➔ If you want to execute only one case-label, C provides break statement.
- ➔ It causes the program to jump out of the switch statement, that is go to the closing braces (}) and continues the remaining code of the program.
- ➔ If we add break to the last statement of the case, the general form of switch case is as follows:

```
switch (printFlag)
{
    case 1:
        printf
            ("This is case 1");
        break;
    case 2:
        printf
            ("This is case 2");
        break;
    default:
        printf
            ("This is default");
        break;
} // switch
```

```
switch (expression)
{
    case constant-1: statement
                    : break;
    case constant-2: statement
                    : break;
    case constant-n: statement
                    : break;
    default         : statement
                    : break;
}
```



(a) printFlag is 1



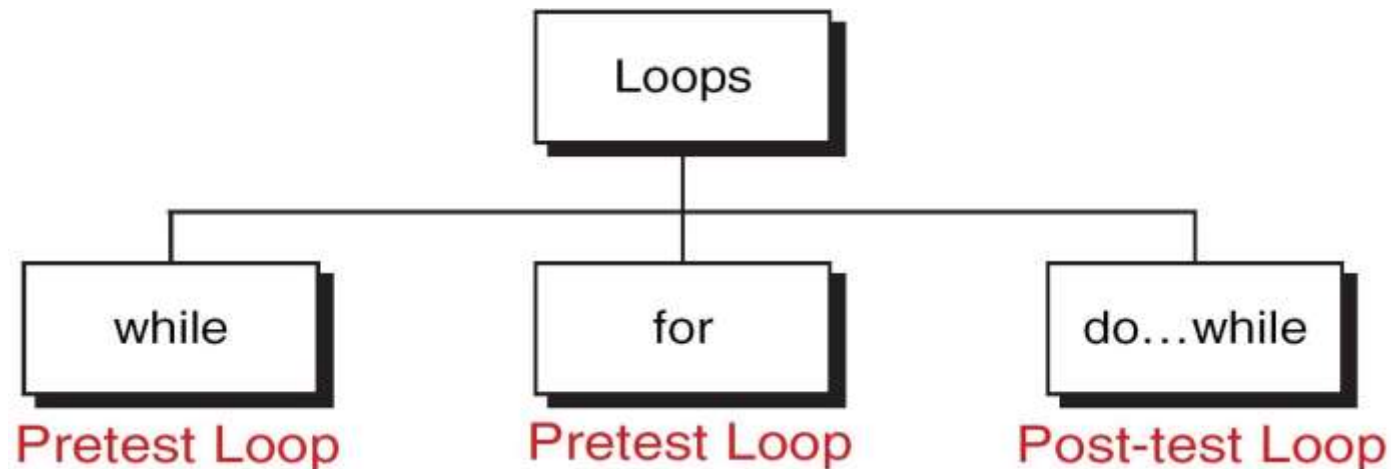
(b) printFlag is 2



(c) printFlag is not 1 or 2

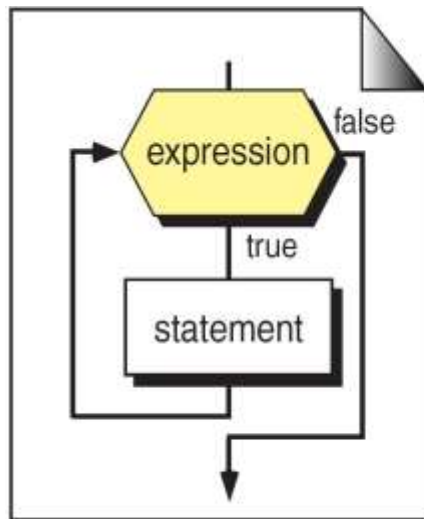
# Loops in C

- C has three loop statements: the while, the for, and the do...while. The first two are pretest loops, and the third is a post-test loop.
- We can use all of them for event-controlled and counter-controlled loops.
- A looping process, in general, would include the following four steps:
- Before a loop start, the loop control variable must be initialized; this should be done before the first execution of loop body.
- Test for the specified condition for execution of the loop, known as loop control expression.
- Executing the body of the loop, known as actions.
- Updating the loop control variable for performing next condition checking.

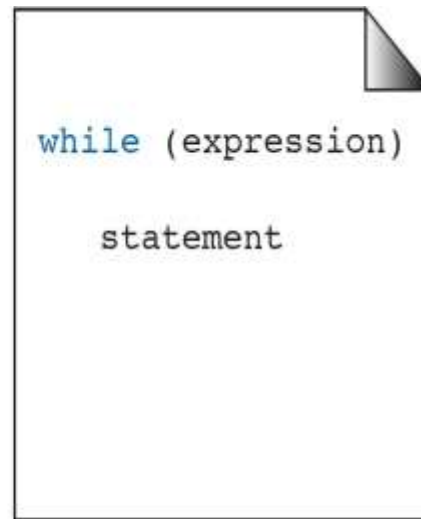


# Loops in C

- **while**
  - ➔ The "while" loop is a generalized looping structure that employs a variable or expression for testing the condition.
  - ➔ It is a repetition statement that allows an action to be repeated while some conditions remain true.
  - ➔ The body of while statement can be a single statement or compound statements.
  - ➔ It doesn't perform even a single operation if condition fails.

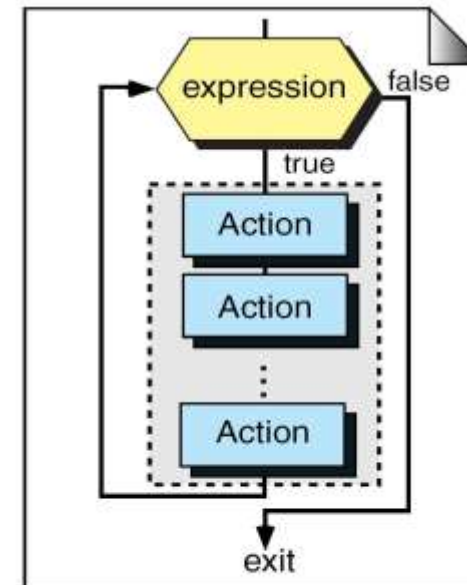


(a) Flowchart

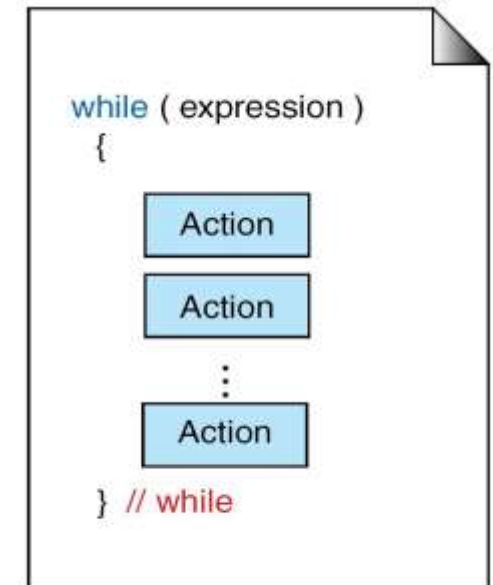


(b) Sample Code

The while Statement



(a) Flowchart



(b) C Language

Compound while Statement

# Loops in C Cont...

## Example 1: To print 1 to 10 natural numbers

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

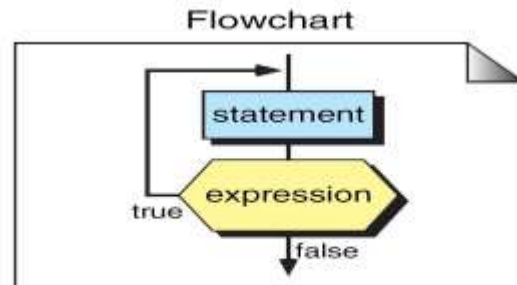
## Example 2: To print the reverse of the given number.

```
void main()
{
    int n, rem, rev = 0;
    printf(" Enter a positive number: ");
    scanf("%d",&n);
    while(n != 0)
    {
        rem = n%10;
        rev = rev*10+rem;
        n = n/10;
    }
    printf("The reverse of %d is %d",n,rev);
}
```

# Loops in C Cont...

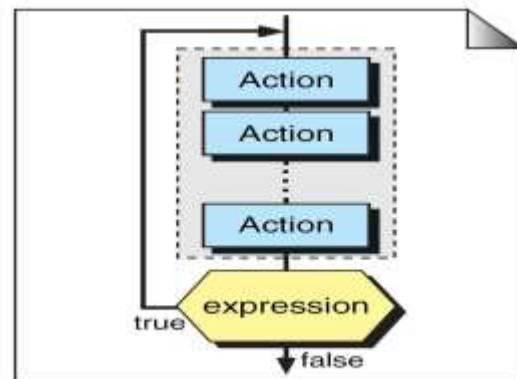
- **do-while**

- ➔ The “do while” loop is a repetition statement that allows an action to be done at least once and then condition is tested.
- ➔ On reaching do statement, the program proceeds to evaluate the body of the loop first.
- ➔ At the end of the loop, condition statement is evaluated.
- ➔ If the condition is true, it evaluates the body of the loop once again.
- ➔ This process continues up to the condition becomes false.



Sample Code

```
do
    statement
while (expression);
```



```
do
{
    Action
    Action
    ...
    Action
} while (expression);
```

# Loops in C Cont...

**Example 3: To print fibonacci sequence for the given number.**

```
#include<stdio.h>
main()
{
    int
    a=0,b=1,c,i;
    i=1;
    printf("%d%d",a,b)
    ; do
    {
        c=a+b;
        i++;
        printf("%3d",c);
        a=b;
        b=c;
    }while(i<=10);
}
```

**Example 4: To print multiplication table for 5.**

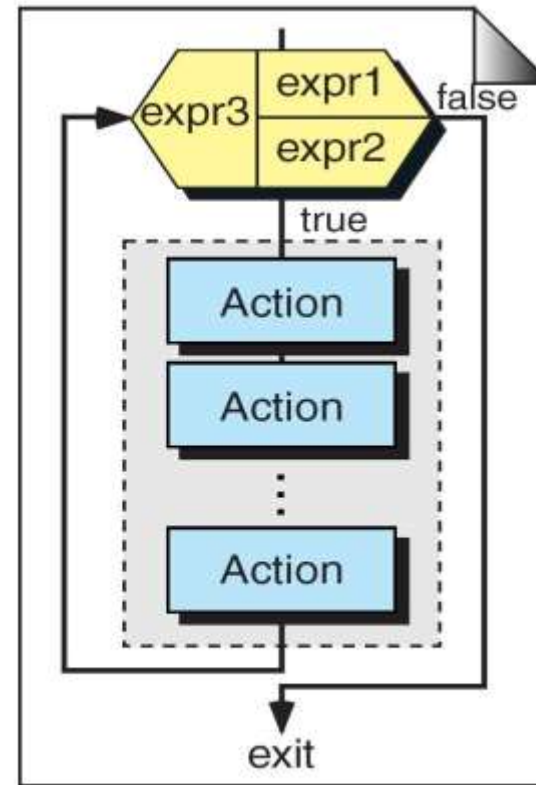
```
#include <stdio.h>
void main()
{
    int      i = 1, n=5;
    do
    {
        printf(" %d * %d = %d ", n, i,
            n*i); i = i + 1;
    } while ( i<= 5);
}
```

# Loops in C Cont...

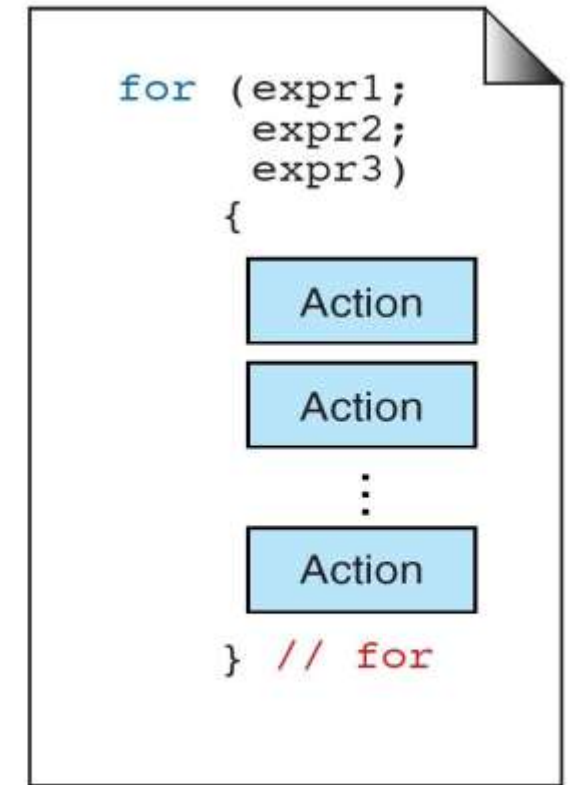
- **for**

- ➔ A for loop is used when a loop is to be executed a known number of times.
- ➔ We can do the same thing with a while loop, but the for loop is easier to read and more natural for counting loops.
- ➔ General form of the for is:

```
for( initialization; test-condition; updation)
{
    Body of the loop
}
```



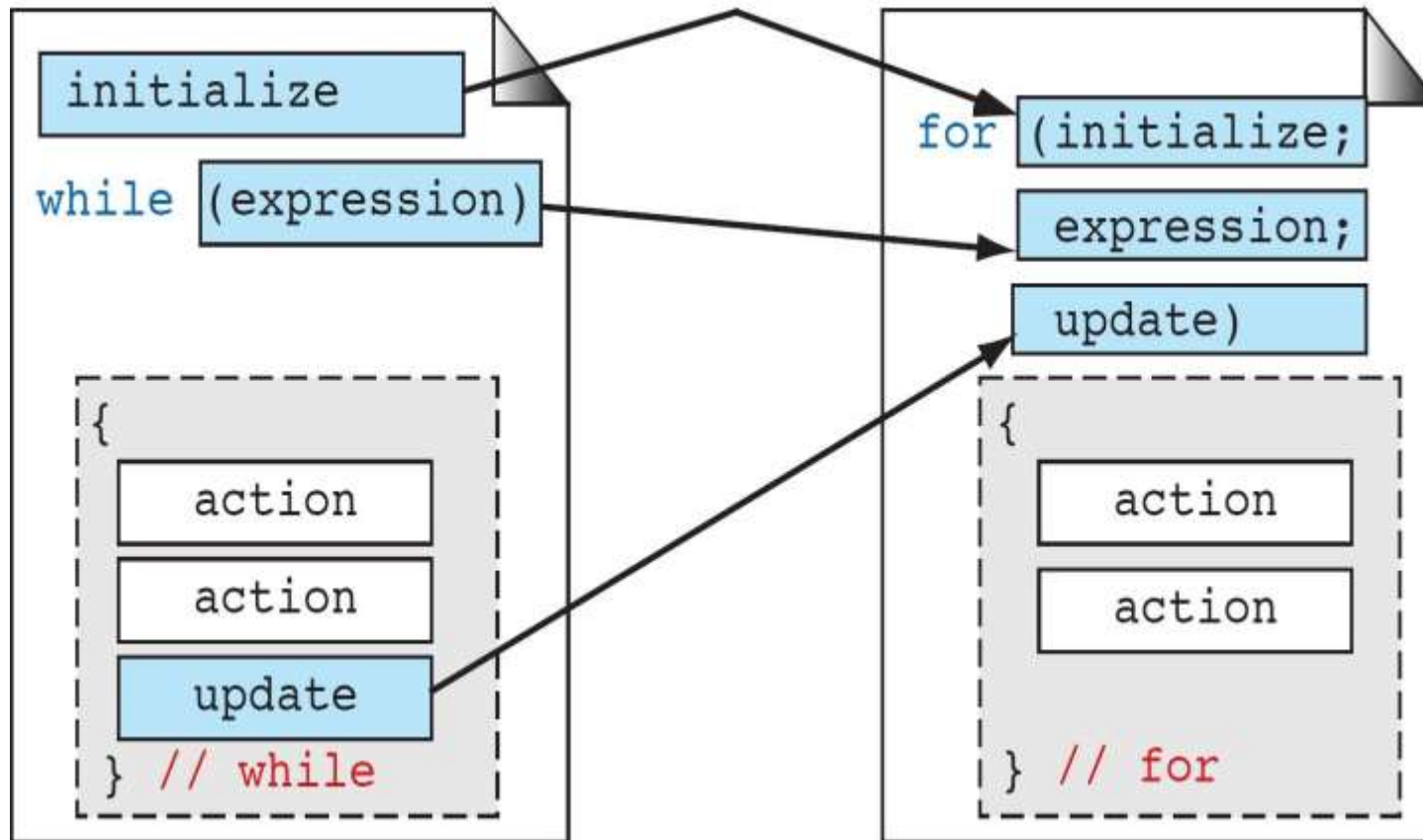
(a) Flowchart



(b) C Language

# Loops in C Cont...

- Compare between while and for loops



# Loops in C Cont...

- ➔ Option 3: The infinite loop
- ➔ One of the most interesting uses of the for loop is the creation of the infinite loop. Since none of the three expressions that form the for loop are required, it is possible to make an endless loop by leaving the conditional expression empty.
- ➔ For example: `for ( ; ; )`  
`printf("The loop will run forever\n");`
- ➔ Actually the for ( ; ; ) construct does not necessarily create an infinite loop because C's break statement, when encountered anywhere inside the body of a loop, causes immediate termination of the loop.
- ➔ Program control then picks up the code following the loop, as shown here:

```
for ( ; ; )  
{  
    ch = getchar( );           /* get a character */  
    if (ch == 'A')  
        break ;  
}  
printf ("you typed an A");
```

- ➔ This loop will run until A is typed at the keyboard.

# Loops in C Cont...

- ➔ Option 3: For loop with no body
- ➔ A statement, as defined by the C syntax, may be empty.
- ➔ This means that the body of the for may also be empty.
- ➔ This fact can be used to improve the efficiency of certain algorithms as well as to create time delay loops.
- ➔ The following statement shows how to create a time delay loop using a for loop:  
`for (t = 0; t < SOME VALUE; t++);`
- ➔ The operator comma , is used to separate the more than one expressions.
- ➔ A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand.
- ➔ Thus, in a for statement, it is possible to place multiple expressions in the various parts.



# Others statements/Jumping Statements

## • 1. break

- ➔ When a break statement is enclosed inside a block or loop, the loop is immediately exited and program continues with the next statement immediately following the loop.
- ➔ When loop are nested break only exit from the inner loop containing it.
- ➔ The format of the break statement is:

<pre>while (expr) {     ...     break;     ... }</pre>	<pre>do {     ...     break;     ... } while (expr);</pre>	<pre>for (expr1; expr2; expr3) {     ...     break;     ... }</pre>
<p>// while</p>	<p>while (expr);</p>	<p>// for</p>

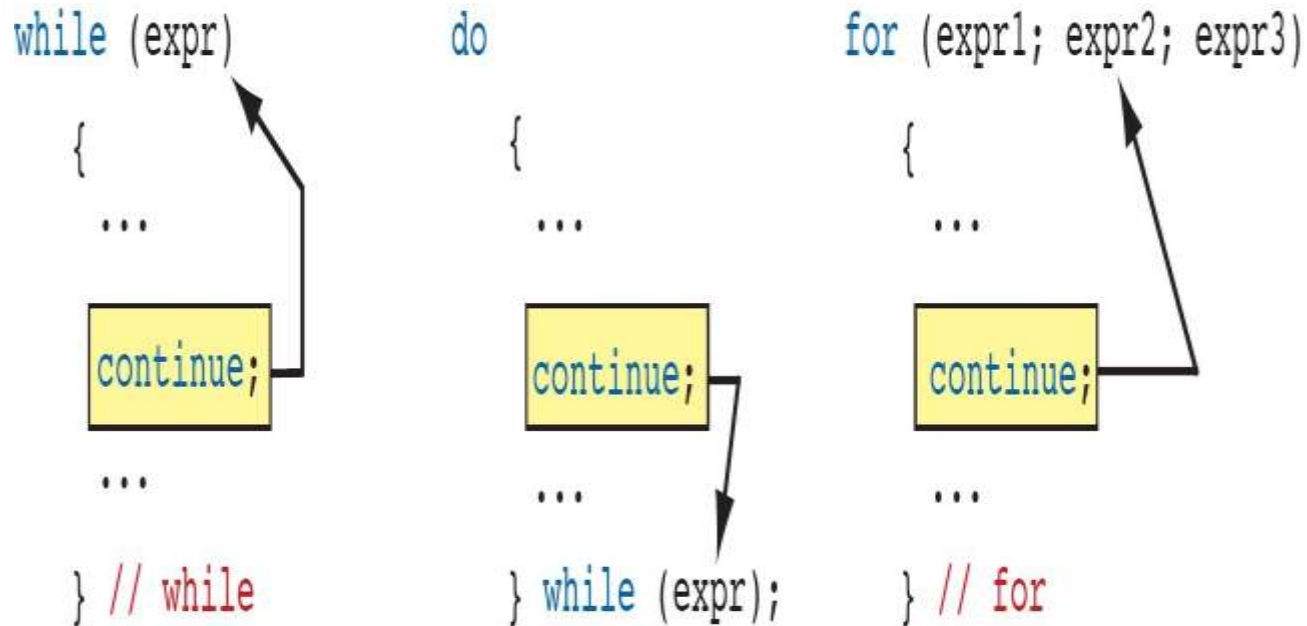
### Example 6: Program to demonstrate **break** statement.

```
#include<stdio.h>
main()
{
    int i;
    i=1;
    while(i<=10)
    {
        if(i==8)
            break;
        printf("%d",i);
        i=i+1;
    }
    printf("\n Thanking You");
}
```

# Others statements /Jumping Statements

## • 2. Continue

- ➔ When a continue statement is enclosed inside a block or loop, the loop is to be continued with the next iteration.
- ➔ The continue statement tells the compiler, skip the following statements and continue with the next iteration.
- ➔ The format of the continue statement is:



### Example 5: Program to demonstrate **continue** statement. `#include<stdio.h>`

```
main()
{
    int i;
    for(i=1;i<=5;i++)
    {
        if(i == 3)
            continue;
        printf(" %d",i);
    }
}
```

# Others statements/Jumping Statements

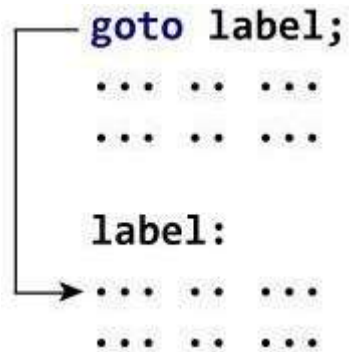
## • 3. goto

→ goto is an unconditional statement used to transfer the control from one statement to another statement in the program.

→ **Syntax:**

Label: Statements; goto label;

→ The format of the gotostatement is:



```
goto label;  
... ..  
... ..  
  
label:  
... ..  
... ..
```

### Example 5: Program to demonstrate goto statement.

```
#include <stdio.h>  
void main()  
{  
    int i;  
    clrscr();  
    for(i=1;i<=10;i++)  
    {  
        printf("%d ",  
            i); if(i==5)  
            goto end;  
    }  
    end:  
    printf("\nEnd of the program");  
}
```

***Thank  
You***



**D. SRINIVAS**

Computer Science and Engineering Department

✉ [srinivascsedept@gmail.com](mailto:srinivascsedept@gmail.com)

☎ +91-9347556447



## Unit-2

# Array, Strings, Structures and Pointers



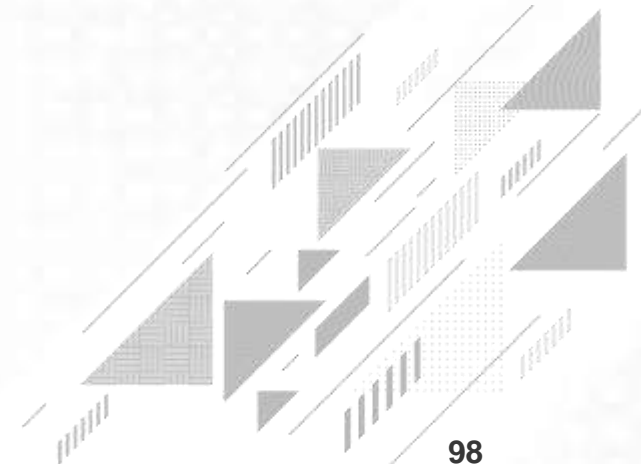
**D. SRINIVAS**

Department of CSE

[www.srinivas-materials.blogspot.com](http://www.srinivas-materials.blogspot.com)

✉ [srinivascsedpt@gmail.com](mailto:srinivascsedpt@gmail.com)

☎ +91 9347556447





# Outline



- **Arrays:**
  - ↳ one- and two-dimensional arrays,
  - ↳ creating, accessing and manipulating elements of arrays.
- **Strings:**
  - ↳ Introduction to strings,
  - ↳ handling strings as array of characters,
  - ↳ Basic string functions available in C (strlen, strcat, strcpy, strstr etc.),
  - ↳ arrays of strings.
- **Structures:**
  - ↳ Defining structures, initializing structures,
  - ↳ unions,
  - ↳ Array of structures
- **Pointers:**
  - ↳ Idea of pointers, defining pointers,
  - ↳ Pointers to Arrays and Structures,
  - ↳ Use of Pointers in self-referential structures,
  - ↳ usage of self-referential structures in linked list (no implementation),
  - ↳ Enumeration data type.

# Introduction to Arrays



# Arrays in C

- **How to create an array:**

- ➔ Creation of an consists two things: **Element Type** and **Array Size**.

- ➔ **Element Type:** What kind of data an array can hold?

An array can hold any one of the following data:

**integer, double, character data.**

- ➔ **Array Size:** How many elements an array can contain?

Once an array size is defined it cannot be changed at run-time

- **Using arrays in C:**

- ➔ In C, arrays can be classified based on how the data items are arranged for human understanding. Arrays are broadly classified into three categories,

1. **One** Dimensional Arrays

2. **Two** Dimensional Arrays

3. **Multi** Dimensional Arrays

# Arrays in C

- **One Dimensional Arrays:**

- ➔ One dimensional array is a linear list consisting of related and similar Data items.
- ➔ In memory all the data items are stored in contiguous memory locations one after the other.
- ➔ Syntax for **declaring** One Dimensional Arrays:

elementType **arrayName**[size];

- ➔ Where elementType specifies data type of each element in the array,arrayName specifies name of the variable you are declaring and size specifies number of elements allocated for this array.
- ➔ To declare regular variables we just specify a data type and a unique name.
- ➔ Example: int number;
- ➔ To declare an array, we just add an array size.

Example: **int temp[5];** //Creates an array of 5 integer elements.

Example: **double stockprice[31];** //Creates an array of 31 double elements.

# Arrays in C

- **Initializing One Dimensional Arrays:**

- ➔ If array is not initialized it contain garbage values.

- ➔ Types of array initializations:

- Option 1: Initializing all memory locations

- Option 2: Initialization without size

- Option 3: Partial array initialization

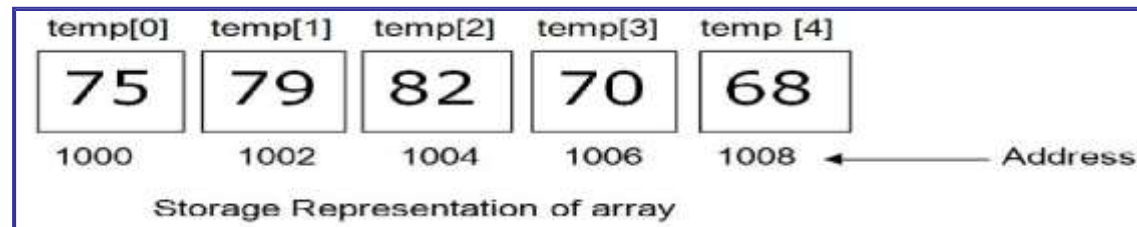
- Option 4: Initializing an entire array with zero.

- **Option 1: Initializing all memory locations:**

- ➔ If you know all the data at compile time, you can specify all your data within brackets:

`int temp [5] = {75, 79, 82, 70, 68};`

- ➔ During compilation, 5 contiguous memory locations are reserved by the compiler for the variable temp and all these locations are initialized as shown below.



- ➔ If the size of integer is 2 bytes, 10 bytes will be allocated for the variable temp.

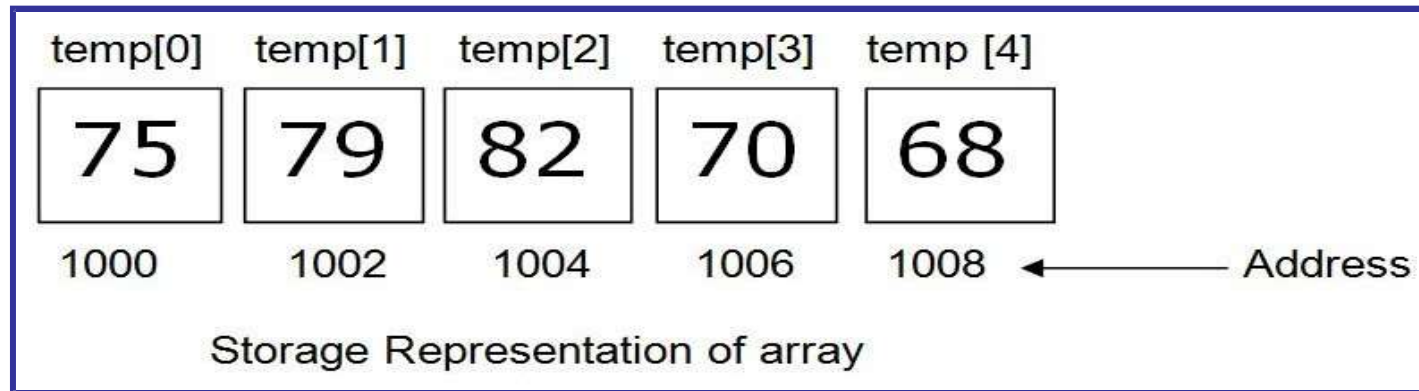
# Arrays in C

- **Option 2: Initialization without size:**

- ➔ If you omit the size of an array, but specify an initial set of data, then the compiler will automatically determine the size of an array.

`int temp [] = {75, 79, 82, 70, 68};`

- ➔ In the above declaration, even though you have not specified exact number of elements to be used in array temp, the array size will be set with the total number of initial values specified.
- ➔ Here, the compiler creates an array of 5 elements. The array temp is initialized as shown below.



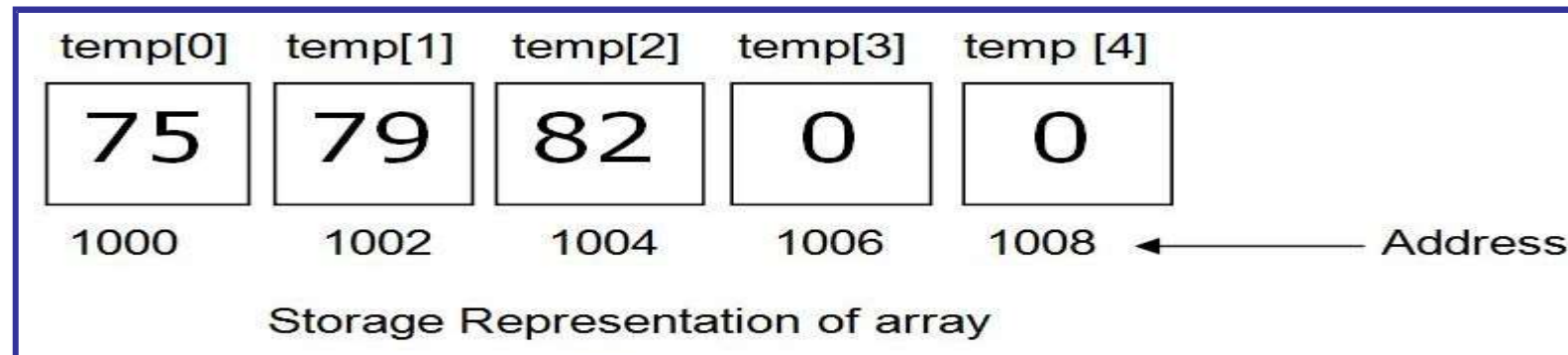
# Arrays in C

- **Option 3 Partial Array Initialization:**

- ➔ If the number of values to be initialized is less than the size of the array, then the elements are initialized in the order from 0<sup>th</sup> location.
- ➔ The remaining locations will be initialized to zero automatically.

`int temp [5] = {75, 79, 82};`

- ➔ Even though compiler allocates 5 memory locations, using the above declaration statement, the compiler initializes first three locations with 75, 79 and 82, and the next set of memory locations are automatically initialized to 0's by the compiler as shown below.



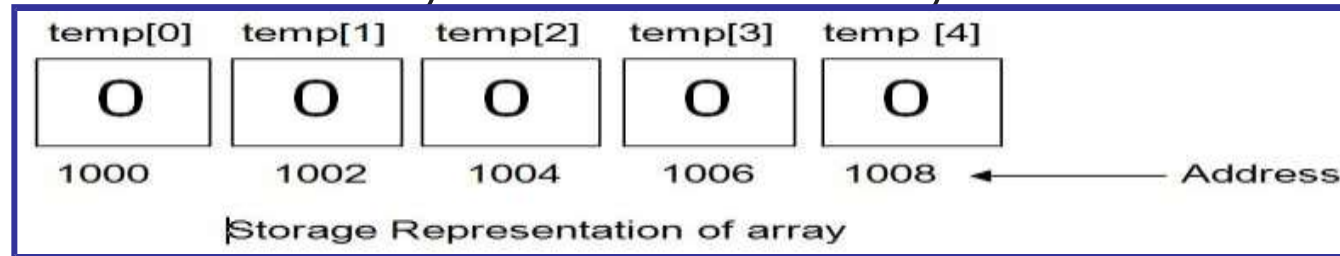
# Arrays in C

- **Option 4: Initializing an entire array with zero:**

- ➔ If you do not know any data ahead of time, but you want to initialize everything to 0, just use 0 within { }. For example:

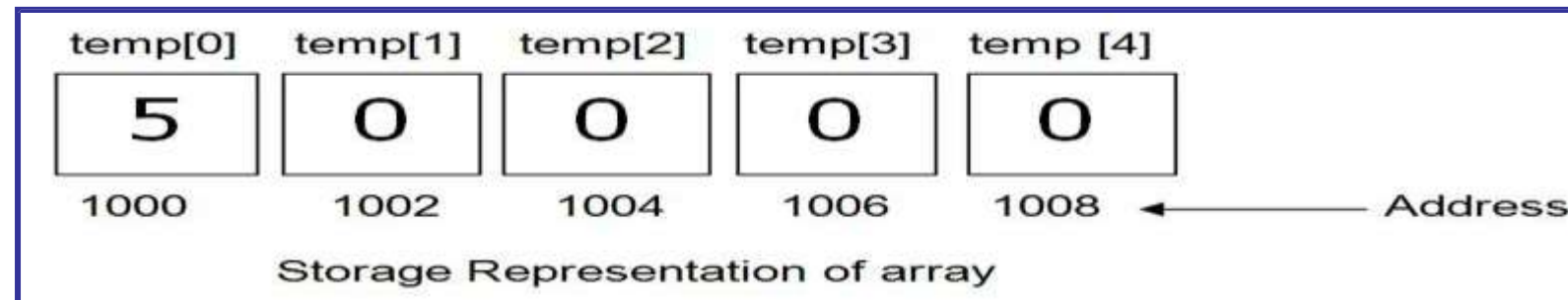
`int temp [5] = {0};`

- ➔ This will initialize every element within the array to 0 as shown below.



- ➔ **Example:**

`int temp [5] = {5};`



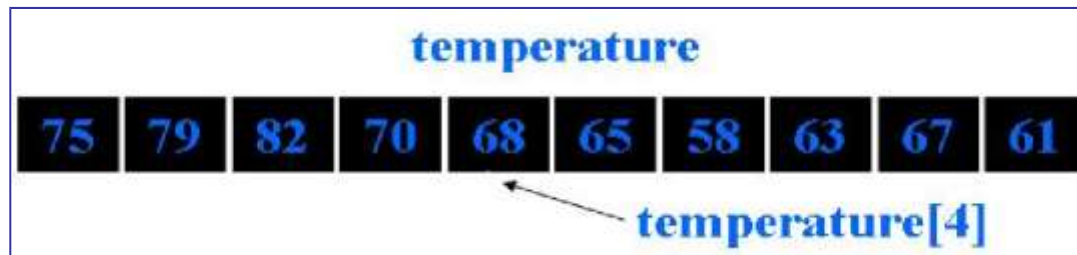
- ➔ The first value is supplied in the first element memory location, remaining all elements are placed with zero.

# Arrays in C

- **Accessing** elements of one dimensional array:

- ➔ You know how to declare and initialize an array. Now let's understand, how to access an array element.
- ➔ To access an array element use **name** of the array with the **subscript** in **brackets**.
- ➔ Suppose you have an array called temperature, for storing temperature in a year.
- ➔ Then the subscripts would be 0, 1, ..., 364.
- ➔ For example to access temperature of fifth day:

**temperature [4]**



- ➔ To **assign or store** the value 89 for the 150th day:  
`temperature [149] = 89`
- ➔ You can loop through the elements of an array by varying the subscript.
- ➔ To set all of the values to 0, say  
`for(i=0;i<365;i++)  
    temperature[i] = 0;`

# Arrays in C

- You can not use assignment statement directly with arrays.
- If `a[]` , `b[]` are two arrays then the assignment `a=b` is not valid.
- C does not provide array bounds checking.
- Hence, if you have
- `double stockPrice[5];`
- `printf ("%d", stockPrice[10]);`
- This will compile, but you have overstepped the bounds of your array.
- You may therefore get wrong value.
- As we can see above, the 5th element of an array is accessed as `'arr[5]'`.
- Note that for an array declared as `int arr[5]`.
- The five values are represented as: `arr[0]` `arr[1]` `arr[2]` `arr[3]` `arr[4]` and   
`not arr[1]` `arr[2]` `arr[3]` `arr[4]` `arr[5]`
- The first element of array always has a subscript of 0

**//Example for accessing array elements.**

```
#include<stdio.h> main()
{
    int arr[10];
    int i = 0;
    for(i=0;i<sizeof(arr);i++)
    {
        arr[i] = i;
    }
    int j = arr[4];
    printf("Value at 5th location is: %d",j);
}
```

# Arrays in C

**//Program to calculate sum of all the array elements.**

```
#include <stdio.h>
void main()
{
    int a[10];
    int i, size, total=0;
    printf(" Enter the size of the array : ");
    scanf("%d", &size);
    printf(" Enter the elements of an array : ");
    for (i = 0; i < size ; i++)
        scanf("%d",&a[i]);
    for (i = 0; i < size ; i++)
        total += a[i];
    printf("Sum of all array elements: %d", total);
}
```

# Arrays in C

- **Two Dimensional Arrays:**

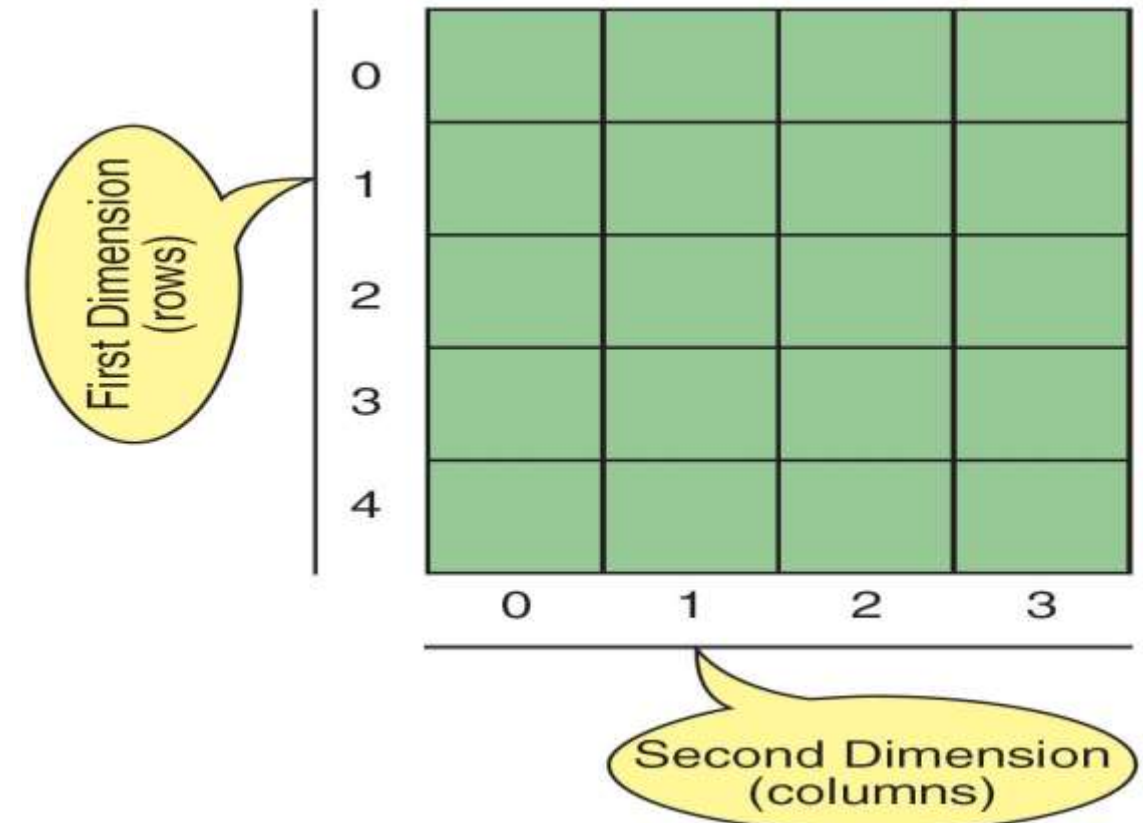
- ➔ Two-dimensional array are those type of array, which has finite number of rows and finite number of columns.
- ➔ An array of array is called a two-dimensional array and can be represented as a table with rows and columns.

'J'	'o'	'h'	'n'
'M'	'a'	'r'	'v'
'I'	'v'	'a'	'n'

is a  $3 \times 4$  array  
3 rows, 4 columns

- ➔ This is an array of size 3 names whose elements are arrays of size 4.

- ➔ Syntax: `elementType arrayName [rowsize][columnsize];`



# Arrays in C

- The declaration form of 2-dimensional array is

**elementType** **arrayName** **[row size][column size];**

- ➔ The elementType may be any valid type supported by C.
- ➔ The rule for giving the arrayName is same as the ordinary variable.
- ➔ The row size and column size should be an individual constant.
- ➔ The following declares a two-dimensional 3 by 3 array of integers and sets the first and last elements to be 10.

```
int matrix [3][3];  
matrix[0][0] = 10;  
matrix[2][2] = 10;
```

- ➔ In the declaration of two dimensional array the column size should be specified, so that it can arrange the elements in the form of rows and columns.
- ➔ Two-dimensional arrays in C are stored in "row-major format": the array is laid out contiguously, one row at a time.

	[0]	[1]	[2]
[0]	10		
[1]			
[2]			10

Two-dimensional Array representation

# Arrays in C

- **Initialization of Two Dimensional Arrays:**

- ➔ An array may be initialized at the time of declaration as follows:

```
char names [3][4] = {  
    {'J', 'o', 'h', 'n'},  
    {'M', 'a', 'r', 'y'},  
    {'I', 'v', 'a', 'n'}  
};
```

- ➔ An integer array may be initialized to all zeros as follows

```
int nums [3][4] = {0};
```

- ➔ An integer array may be initialized to different values as follows

```
int nums [3][4] = {  
    {1,2,3,4},  
    {5,6,7,8},  
    {9,0,10,11}  
};
```

# Arrays in C

→ To access an element of a 2D array, you need to specify both the row and the column:

```
printf ("%d", nums[1][2]);
```

output:

```
a[0][0]=0 a[0][1]=1 a[0][2]=2 a[0][3]=3
```

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

```
a[2][0]=0 a[2][1]=3 a[2][2]=4 a[2][3]=5
```

```
sum = 30
```

**//Program to print sum of elements of a matrix.**

```
#define M 3    /* Number of rows */  
#define N 4    /* Number of columns */
```

```
main() {  
    int a [ M ] [ N ], i, j, sum = 0;  
    for ( i = 0; i < M; ++i ) {  
        for ( j = 0; j < N, ++j ){  
            scanf ("%d", &a [i] [j]);  
        }  
        for ( i = 0; i < M; ++i ) {  
            for ( j = 0; j < N, ++j ) {  
                printf("a [ %d ] [ %d ] = %d  
            }  
            printf ("\n");  
        }  
        for ( i = 0; i < M; ++i ) {  
            for ( j = 0; j < N, ++j ) {  
                sum += a [ i ] [ j ];  
            }  
            printf("\nsum = %d\n\n");  
        }  
    }  
}
```

}

# Arrays in C

- **Multi Dimensional Arrays:**

- ➔ C allows three or more dimensions. The exact limit is determined by the compiler.
- ➔ The general form of multidimensional array is  
elementType **arrayName** [s1][s2][s3]...[sm];
- ➔ Where si is the size of the i<sup>th</sup> dimension.
- ➔ Array declarations read right-to-left
- ➔ For Example:      int a[3][5][4];
- ➔ It is represented as “an array of ten arrays of three arrays of two ints”
- ➔ In memory the elements are stored as shown in below figure.

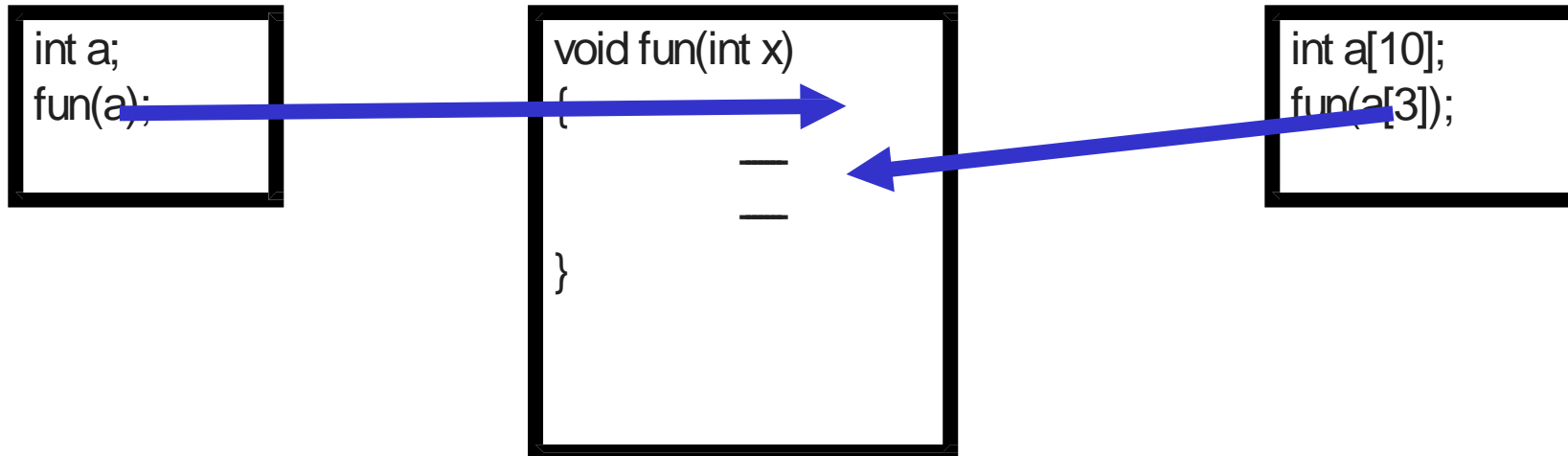
# Arrays in C

Example:

```
int table[3][5][4] = {  
    {  
        {000,001,002,003},  
        {010,011,012,013},  
        {020,021,022,023},  
        {030,031,032,033},  
        {040,041,032,043}  
    },  
    {  
        {100,101,102,103},  
        {110,111,112,113},  
        {120,121,122,123},  
        { 130 , 131 , 132 , 133  
        },  
        {140,141,142,143}  
    },  
    {  
        {200,201,202,203},  
        {210,211,212,213},  
        {220,221,222,223},  
        { 230 , 231 , 232 , 233  
        },  
        {240,241,242,243}  
    }  
};
```

# Arrays in C

- **Inter-function communication (Functions with Arrays):**
  - Like the values of variable, it is also possible to pass values of an array to a function.
  - There are two types of passing an array to the function:
    - 1. Passing Individual Elements
    - 2. Passing the whole array
- 1. Passing Individual Elements:



# Arrays in C

- **2. Passing the whole array:**

- ➔ To pass an array to a called function, it is sufficient to list the **name** of the array, without any subscripts, and the **size** of the array as arguments.
- ➔ For example, the function call **findMax(a, n);** will pass all the elements contained in the array a of size n.
- ➔ The called function expecting this must be appropriately defined.
- ➔ The findMax function header looks like: **int findMax(int x[], int size)**
- ➔ The pair of brackets informs the compiler that the argument x is an array of numbers. It is not necessary to specify the size of the array here.
- ➔ The function prototype takes of the form

**int findMax (int [], int );**

**int findMax (int a [], int );**

# Arrays in C

**//Program to read an array of elements and find max value.**

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

```
int findMax(int[],int);
```

```
void main()
```

```
{
```

```
int a[10], n ,i , max;
```

```
printf("\n Enter the size of the array ");
```

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

```
printf("\n Enter the elements of the array : ");
```

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

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

```
max=findMax(a, n);
```

```
printf("\n The Maximum value =%d", max);
```

```
}
```

```
int findMax(int x[],int size)
```

```
{
```

```
int temp;
```

```
temp=x[0];
```

```
for(i=1;i<size; i++)
```

```
{
```

```
if(x[i]>temp)
```

```
{
```

```
temp=x[i];
```

```
}
```

```
}
```

```
return temp;
```

```
}
```

```
Enter the size of the array 5
```

```
Enter the elements of the array : 10 4 56 44 12
```

```
The Maximum value =56
```

# Introduction to Strings



# Strings

- **Character Arrays and Strings:**

- ➔ String is a sequence of characters.
- ➔ If '\0' is present after a series of characters in an array, then that array becomes a string otherwise it is a character array.

➔ Example:

```
char arr[] = {'a', 'b', 'c'};           //This is an array
```

```
char arr[] = {'a', 'b', 'c', '\0' };    //This is a string
```

- **Strings:**

- ➔ A C string is a variable-length array of characters that is delimited by the null character.
- ➔ A string is a sequence of characters.
- ➔ A string literal is enclosed in double quotes.

# Strings

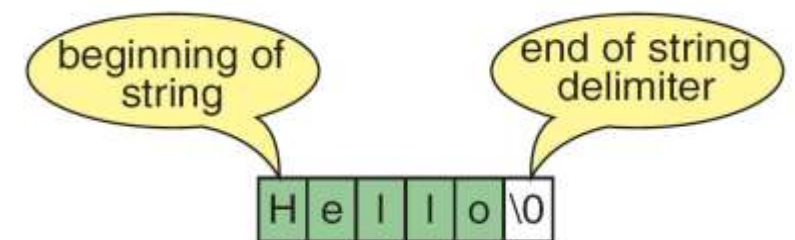
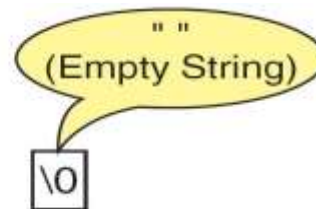
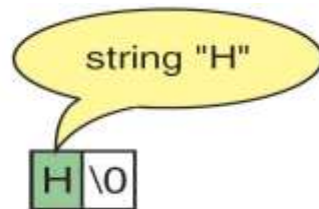
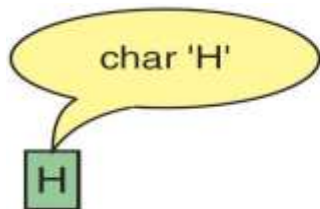
- **Declaring And Initializing String Variables**

- **Declaring a String:**

- ➔ A string variable is a valid C variable name and always declared as an array.
- ➔ The general form of declaration of a string variable is,  
char string name [size];
- ➔ The size determines the number of characters in the string name.
- ➔ When the compiler assigns a character string to a character array, it automatically supplies a null character('\0') at the end of the string.
- ➔ The size should be equal to the maximum number of characters in the string plus one.

- **Initializing a String: This can be done in two ways.**

1. char str1[7]="Welcome";
2. char str2[8]={'W','e','l','c','o','m','e','\0'};



Storing Strings and Characters

# Arrays of Strings

- Ragged arrays are very common with strings.
- Consider, for example, the need to store the days of the week in their textual format.
- We could create a two-dimensional array of seven days by ten characters, but this wastes space.

```
1  /* Demonstrates an array of pointers to strings.
2      Written by:
3      Date written:
4  */
5  #include <stdio.h>
6
7  int main (void)
8  {
9  // Local Declarations
10     char*  pDays[7];
11     char** pLast;
12
13 // Statements
14     pDays[0] = "Sunday";
15     pDays[1] = "Monday";
16     pDays[2] = "Tuesday";
17     pDays[3] = "Wednesday";
18     pDays[4] = "Thursday";
```

```
19     pDays[5] = "Friday";
20     pDays[6] = "Saturday";
21
22     printf("The days of the week\n");
23     pLast = pDays + 6;
24     for (char** pWalker = pDays;
25          pWalker <= pLast;
26          pWalker++)
27         printf("%s\n", *pWalker);
28     return 0;
29 } // main
```

# Strings Manipulation Functions

- The C Library provides a rich set of string handling functions that are placed under the header file `<string.h>` and `<ctype.h>`.

→ Some of the string handling functions are (`string.h`):

<code>strlen()</code>	<code>strcat()</code>	<code>strcpy()</code>	<code>strrchr()</code>
<code>strcmp()</code>	<code>strstr()</code>	<code>strchr()</code>	<code>strrev()</code>

→ Some of the string conversion functions are (`ctype.h`):

<code>toupper()</code>	<code>tolower()</code>	<code>toascii()</code>
------------------------	------------------------	------------------------

→ All I/O functions are available in `stdio.h`

<code>scanf()</code>	<code>printf()</code>	<code>gets()</code>	<code>puts()</code>
<code>getchar()</code>	<code>putchar()</code>		

# Strings Manipulation Functions

- **strlen () function:**

- This function counts and returns the number of characters in a string. It takes the form  
Syntax: `int n=strlen(string);`
- Where n is an integer variable, which receives the value of the length of the string. The counting ends at the first null character.

- **strcat () function:**

- The strcat function joins two strings together.
- It takes of the following form:  
`strcat(string1,string2);`
- string1 and string2 are character arrays.
- When the function strcat is executed, string2 is appended to string1.
- It does so by removing the null character at the end of string1 and placing string2 from there.
- strcat function may also append a string constant to a string variable. The following is valid.  
`strcat(part1,"Good");`
- C permits nesting of strcat functions.
- Example:  
`strcat(strcat(string1,string2),string3);`

# Strings Manipulation Functions

C O N \0

s1- before

C A T E N A T I O N \0

s2 - before

(a) `strcat( s1, s2 ) ;`

C O N C A T E N A T I O N \0

s1 - after

C A T E N A T I O N \0

s2 - after

String Concatenate

C O N \0

s1- before

C A T E N A T I O N \0

s2 - before

(b) `strncat( s1, s2, 3 ) ;`

C O N C A T \0

s1 - after

C A T E N A T I O N \0

s2 - after

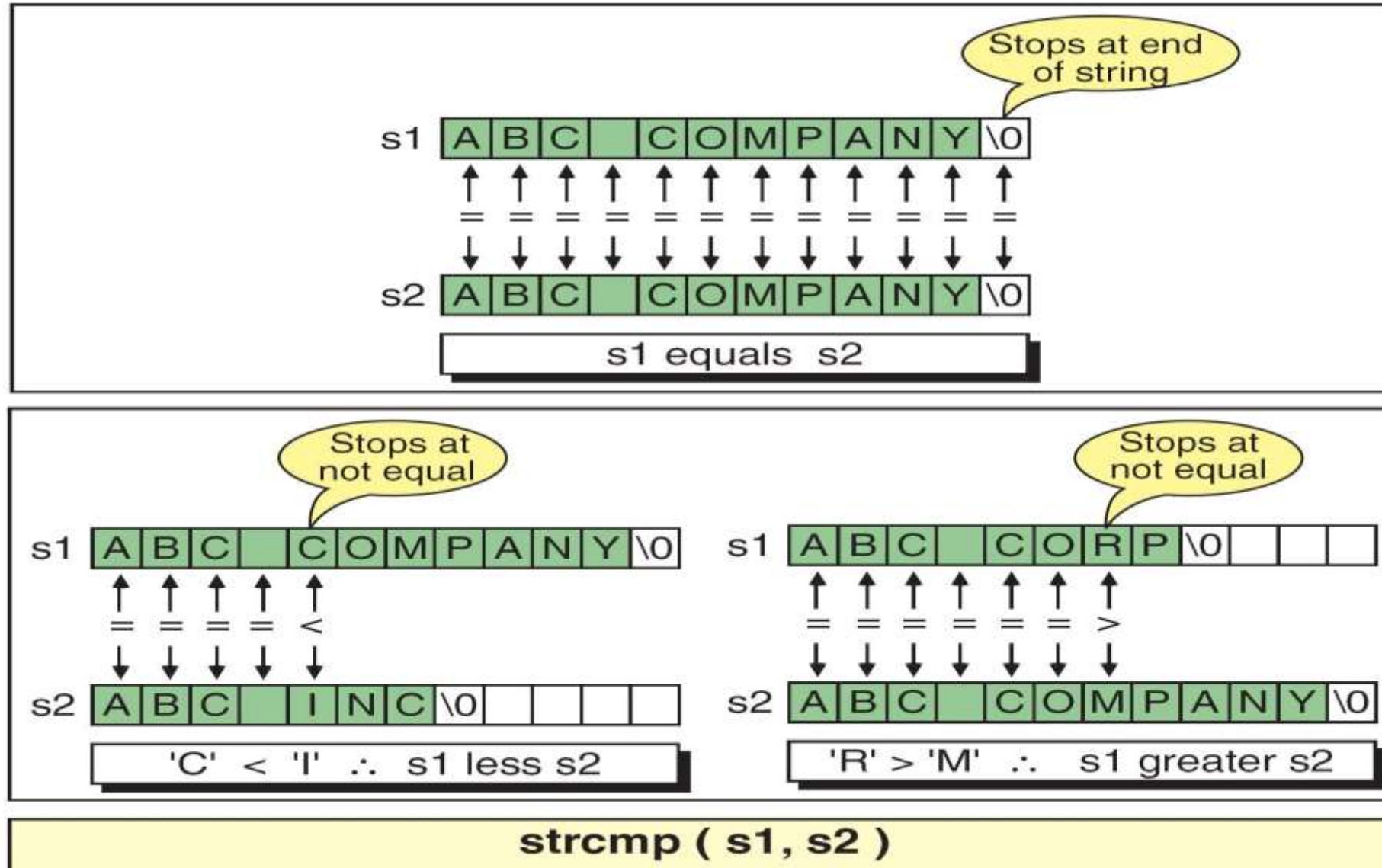
String N Concatenate

# Strings Manipulation Functions

- **strcmp () function:**

- ➔ The strcmp function compares two strings, it returns the value 0 if they are equal.
- ➔ If they are not equal, it returns the numeric difference between the first non matching characters in the strings.
- ➔ It takes the following form:  
`strcmp(str1,str2);`
- ➔ returning value less than 0 means "str1" is less than "str2"
- ➔ returning value 0 means "str1" is equal to "str2"
- ➔ returning value greater than 0 means "str1" is greater than "str2"
- ➔ string1 and string2 may be string variables or string constants.
- ➔ Example:  
`strcmp(name1,name2);`  
`strcmp(name1,"John");`  
`strcmp("their" ,"there");`

# Strings Manipulation Functions



# Strings Manipulation Functions

- **strcpy () function:**

- ➔ It copies the contents of one string to another string. It takes the following form:

- strcpy(string1,string2);**

- ➔ The above function assign the contents of string2 to string1.

- ➔ string2 may be a character array variable or a string constant.

- ➔ Example:               strcpy(city , "Delhi");  
                              strcpy(city1,city2):

- **strrev() function:**

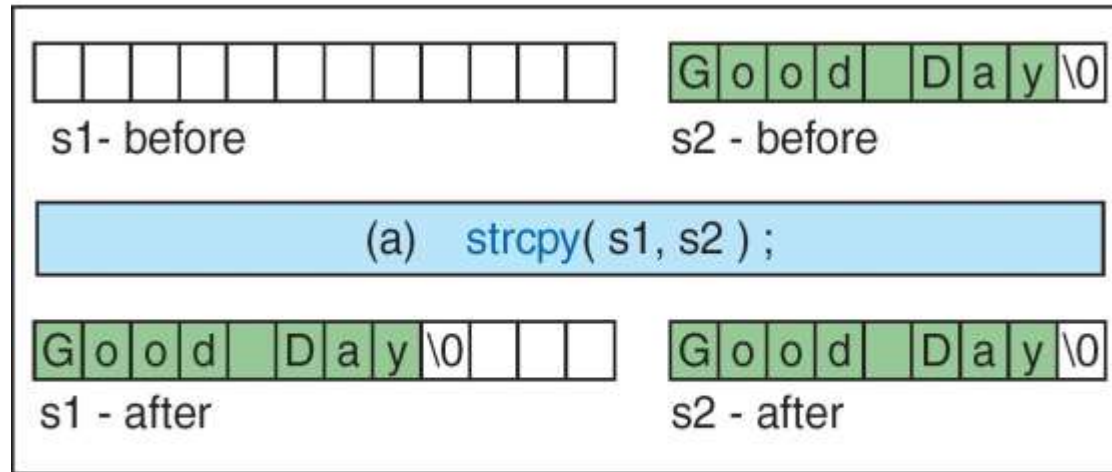
- ➔ Reverses the contents of the string. It takes of the form

- strrev(string);**

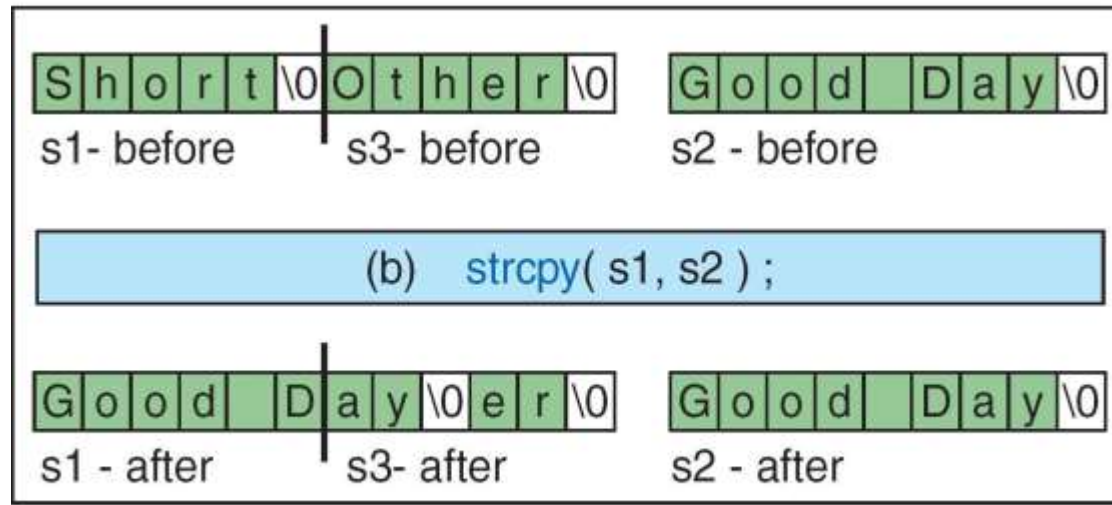
**Example:**

```
#include<stdio.h>
#include<string.h>
void main()
{char s[]="hello";
strrev(s);
puts(s);
}
```

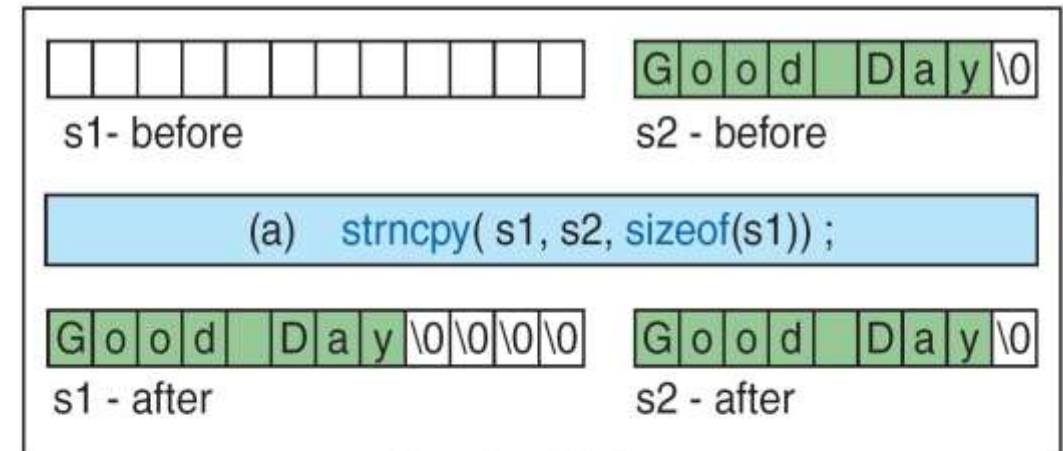
# Strings Manipulation Functions



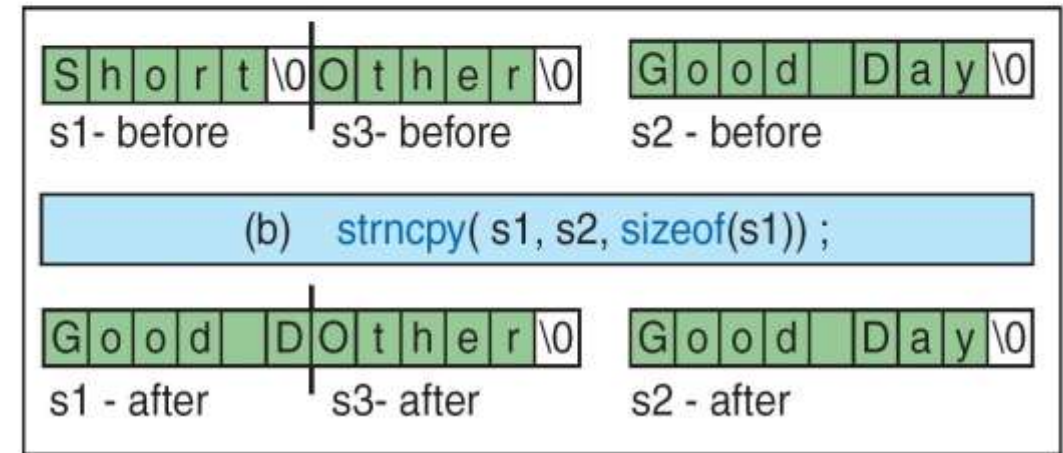
Copying Strings



Copying Long Strings



Copying Strings



Copying Long Strings

# Strings Manipulation Functions

- **strstr () function:**

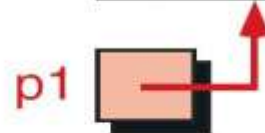
- ➔ It is a two-parameter function that can be used to locate a sub-string in a string.
- ➔ It takes the form:
  - ➔ `strstr (s1, s2);`
- ➔ Example: `strstr (s1,"ABC");`
- ➔ The function `strstr` searches the string `s1` to see whether the string `s2` is contained in `s1`. If yes, the function returns the position of the first occurrence of the sub-string. Otherwise, it returns a `NULL` pointer.

- **strchr() function:**

- ➔ It is used to determine the existence of a character in a string.
- ➔ Example: `strchr (s1,'m');` //It locates the first occurrence of the character 'm'.
- ➔ Example: `strrchr(s2,'m');` //It locates the last occurrence of the character 'm'.

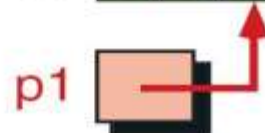
# Strings Manipulation Functions

s1 C O N C A T E N A T I O N \0



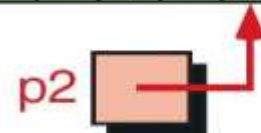
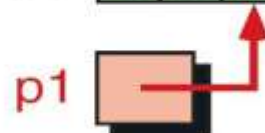
```
p1 = strchr ( s1, 'N' ) ;
```

s1 C O N C A T E N A T I O N \0



```
p2 = strrchr ( s1, 'N' ) ;
```

s1 C O N C A T E N A T I O N \0

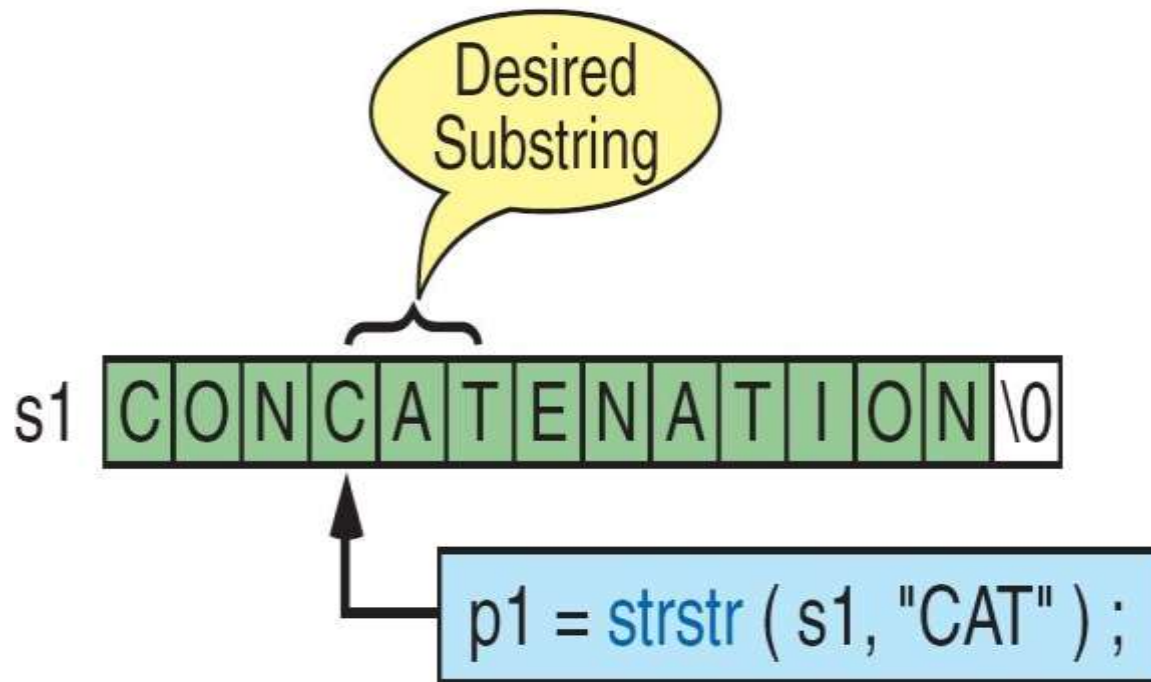


```
p3 = strchr ( (p1 + 1) , 'N' ) ;
```

# Strings Manipulation Functions

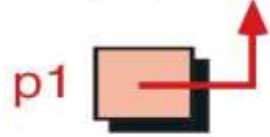
- **strcat () function:**

- ➔ It is used to join only two Strings at a time.
- ➔ It takes the form:
  - ➔ `strcat (s1, s2);`
  - ➔ Example: `strcat (s1,"CAT");`



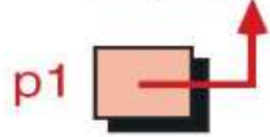
# Strings Manipulation Functions

s1 C O N C A T E N A T I O N \0



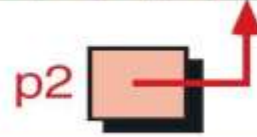
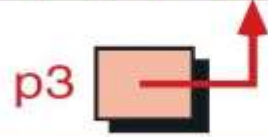
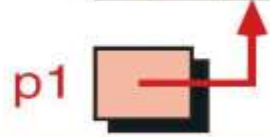
```
p1 = strchr ( s1, 'N' ) ;
```

s1 C O N C A T E N A T I O N \0



```
p2 = strchr ( s1, 'N' ) ;
```

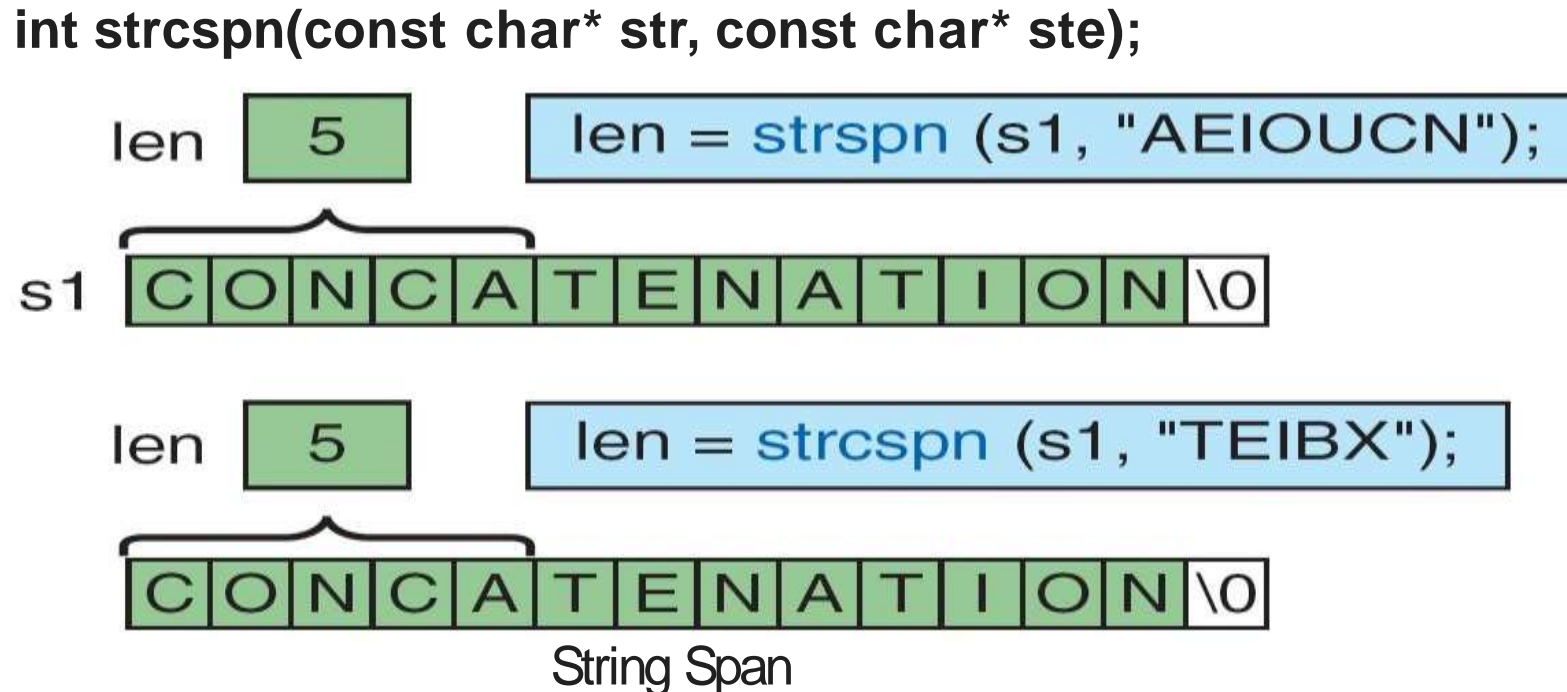
s1 C O N C A T E N A T I O N \0



```
p3 = strchr ( (p1 + 1) , 'N' ) ;
```

# Strings Manipulation Functions

- The basic string span function, `strspn`, searches the string, spanning characters that are in the set and stopping at the first character that is not in the set.
- They return the number of characters that matched those in the set.
- If no characters match those in the set, they return zero.
- The function declaration is shown below:  
**`int strspn(const char* str, const char* set);`**
- The second function, `strcspn`, is string complement span; its functions stop
- at the first character that matches one of the characters in the set.



# String Example program

**/\*Define functions- length of a string, copy, concatenate, convert into uppercase letters, compare two strings for alphabetical order- over strings and implement in a program\*/**

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

```
#include<string.h>
```

```
#include<stdlib.h>
```

```
#include<ctype.h>
```

```
main() {
```

```
    char str1[15],str2[15],str3[10];
```

```
    int n,c,len,i;
```

```
    printf("\n Enter the string1 ");
```

```
    gets(str1);
```

```
    puts(str1);
```

```
    printf("\n Enter the string2 ");
```

```
    gets(str2);
```

```
    puts(str2);
```

```
    printf("Enter the string 3 ");
```

```
    scanf("%s",str3);
```

```
    printf("%s",str3);
```

# String Example program

```
printf("\n*****");
printf("\n 1. String Length      ");
printf("\n 2. String Copy        ");
printf("\n 3. String Comparison     ");
printf("\n 4. String Concat           ");
printf("\n 5. UpperCase                ");
printf("\n*****");
printf("\n Enter the choice u want to perform");
scanf("%d",&n);
switch(n)
{
    case 1:    len=strlen(str1);
               printf("\n The length of the string entered is %d",len);
               break;
    case 2:    strcpy(str1,str2);
               printf("\n 1st string =%s,2nd string=%s",str1,str2);
               break;
    case 3:    c=strcmp(str1,str2);
               if(c==0)
                   printf("\n Both are equal");
               else
```

# String Example program

```
printf("\n Both are different");
break;
case 4:
printf("\n The resultant string is: %s",strcat(str1,str2))
break;
case 5:
for(i=0;i<strlen(str1);i++)
    str1[i]=toupper(str1[i]);
printf("%s",str1);
break;
default: printf("\n Enter correct choice");
}
}
```

## OUTPUT:

Enter the string1: abcd

abcd

Enter the string2: efgh

efgh

Enter the string3: pqr

pqr

\*\*\*\*\*

1. String Length
2. String Copy
3. String Comparison
4. String Concat
5. UpperCase

\*\*\*\*\*

Enter ur choice 4

The resultant string is: abcdefgh

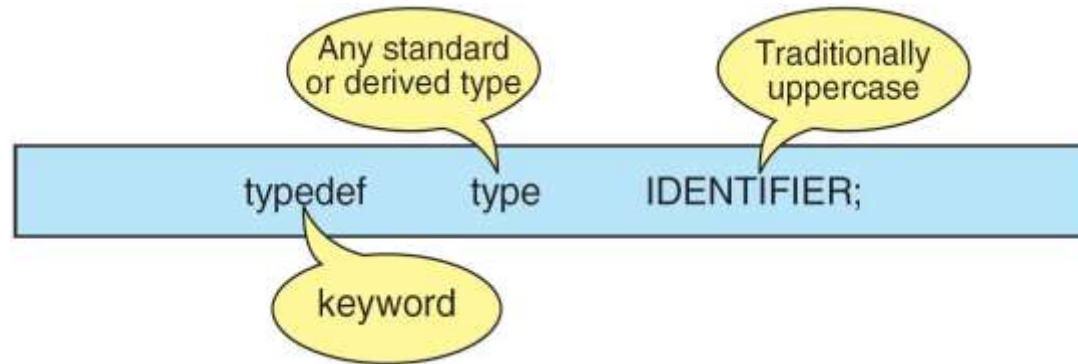
# Introduction to Structures



# The Type Definition (*typedef*)

- **typedef :**

- ➔ A type definition, **typedef**, gives a name to a data type by creating a new type that can then be used anywhere a type is permitted.
- ➔ Its purpose is to redefine the name of an existing variable type.



- ➔ The general syntax of the typedef is as follows,  
`typedef datatype IDENTIFIER;`
- ➔ where typedef is the keyword that tells the compiler about the type definition, data\_type refers to an existing data type and IDENTIFIER refers the “new” name given to the data type.
- ➔ Note that using typedef, we are not creating new data types.
- ➔ Instead we are creating only new name for the existing data type.
- ➔ These new data type names are called user-defined data types.

# The Type Definition (*typedef*)

- The general syntax of the typedef is as follows,
- `typedef datatype IDENTIFIER;`
- where typedef is the keyword that tells the compiler about the type definition, data\_type refers to an existing data type and IDENTIFIER refers the “new” name given to the data type.
- Note that using typedef, we are not creating new data types.
- Instead we are creating only new name for the existing data type.
- These new data type names are called user-defined data types.
- Suppose we want to store marks scored in various subjects in variables sub1, sub2 and sub3. These variables can be declared as follows,

```
int sub1, sub2, sub3;
```

- Using the user-defined data types, the variables can be declared as shown below,
- **typedef** int MARKS;

```
MARKS sub1, sub2, sub3;
```

# The Type Definition (*typedef*)

- //Example program to demonstrate typedef

```
#include<stdio.h>
int main()
{
    typedef int MARKS;
    MARKS sub1,sub2,sub3;
    printf("\n Enter Marks In three subjects: ");
    scanf("%d%d%d",&sub1,&sub2,&sub3);
    printf("\n Marks In three subjects: ");
    printf("%d %d %d ",sub1,sub2,sub3);
    return 0;
}
```

# What is Structure?

- Structure is a collection of logically related data items of different datatypes grouped together under single name.
- Structure is a **user defined datatype**.
- Structure helps to build a complex datatype which is more meaningful than an array.
- But, an array holds similar datatype record, when structure holds different datatypes records.
- Two fundamental aspects of Structure:
  - ➔ Declaration of Structure Variable
  - ➔ Accessing of Structure Member

# Syntax to Define Structure

- To define a structure, we need to use **struct** keyword.
- This keyword is reserved word in C language. We can only use it for structure and its object declaration.

Syntax

```
1 struct structure_name
2 {
3     member1_declaration;
4     member2_declaration;
5     . . .
6     memberN_declaration;
7 };
```

structure\_name is name of custom type

memberN\_declaration is individual member declaration

- Members can be normal variables, pointers, arrays or other structures.
- Member names within the particular structure must be distinct from one another.

# Create Structure variable

- A data type defines various properties about data stored in memory.
- To use any type we must declare its variable.
- Hence, let us learn how to create our custom structure type objects also known as **structure variable**.
- In C programming, there are two ways to declare a structure variable:
  1. Along with structure definition
  2. After structure definition

# Create Structure Variable – Cont.

## 1. Declaration along with the structure definition

### Syntax

```
1 struct structure_name
2 {
3     member1_declaration;
4 member2_declaration; 5
6     . . .
7     memberN_declaration;
8 } structure_variable;
```

### Example

```
1 struct student
2 {
3     char name[30]; // Student Name
4     int roll_no; // Student Roll No
5     float CPI; // Student CPI
6     int backlog; // Student Backlog
7 } student1;
```

# Create Structure Variable – Cont.

## 2. Declaration after Structure definition

### Syntax

```
1 struct structure_name structure_variable;
```

### Example

```
1 struct student
2 {
3     char name[30]; // Student Name
4     int roll_no; // Student Roll No
5     float CPI; // Student CPI
6 int backlog; // Student Backlog 7
7 };
8 struct student student1; // Declare structure variable
```

# Access Structure member (data)

- Structure is a complex data type, we cannot assign any value directly to it using assignment operator.
- We must assign data to individual **structure members** separately.
- C supports two operators to access structure members, using a structure variable.
  1. Dot/period operator (.)
  2. Arrow operator (->)

# Access Structure member (data) – Cont.

## 1. Dot/period operator (.)

→ It is known as member access operator. We use **dot operator** to access members of simple structure variable.

### Syntax

2. 

```
1 structure_variable.member_name;
```

→ In C language it is illegal to access a structure member from a pointer to structure variable using dot operator.

→ We use **arrow operator** to access structure member from pointer to structure.

### Example

```
1 // Assign CPI of student1
2 student1.CPI = 7.46;
```

### Syntax

```
1 pointer_to_structure->member_name;
```

### Example

```
1 // Student1 is a pointer to student type
2 student1 -> CPI = 7.46;
```

## Write a program to read and display student information

### using struct

```
1  #include <stdio.h>
2  struct student
3  {
4      char name[40]; // Student name
5      int roll; // Student enrollment
6      float CPI; // Student mobile number
7  int backlog; 8
9  };
10 int main()
11 {
12     struct student student1; // Simple structure variable
13     // Input data in structure members using dot operator
14     printf("Enter Student Name:");
15     scanf("%s", student1.name);
16     printf("Enter Student Roll Number:");
17     scanf("%d", &student1.roll);
18     printf("Enter Student CPI:");
19     scanf("%f", &student1.CPI);
20     printf("Enter Student Backlog:");
21     scanf("%d", &student1.backlog);
22     // Display data in structure members using dot operator
23     printf("\nStudent using simple structure variable.\n");
24     printf("Student name: %s\n", student1.name);
25     printf("Student Enrollment: %d\n", student1.roll);
26     printf("Student CPI: %f\n", student1.CPI);
27     printf("Student Backlog: %i\n", student1.backlog);
28 }
```

### Output

```
Enter Student Name:aaa
Enter Student Roll Number:111
Enter Student CPI:7.89
Enter Student Backlog:0
```

```
Student using simple structure variable.
Student name: aaa
Student Enrollment: 111
Student CPI: 7.890000
Student Backlog: 0
```

## Write a program to declare time structure and read two different time period and display sum of it.

### Program

```
1  #include<stdio.h>
2  struct time {
3      int hours;
4      int minutes;
5      int seconds;
6  };
7  int main() {
8      struct time t1,t2;
9      int h, m, s;
10     //1st time
11     printf ("Enter 1st time.");
12     printf ("\nEnter Hours: ");
13     scanf ("%d",&t1.hours);
14     printf ("Enter Minutes: ");
15     scanf ("%d",&t1.minutes);
16     printf ("Enter Seconds: ");
17     scanf ("%d",&t1.seconds);
18     printf ("The Time is
19     %d:%d:%d",t1.hours,t1.minutes,t1.seconds);
20     //2nd time
21     printf ("\n\nEnter the 2nd time.");
22     printf ("\nEnter Hours: ");
23     scanf ("%d",&t2.hours);
24     printf ("Enter Minutes: ");
25     scanf ("%d",&t2.minutes);
26     printf ("Enter Seconds: ");
```

```
27     scanf ("%d",&t2.seconds);
28     printf ("The Time is
29     %d:%d:%d",t2.hours,t2.minutes,t2.seconds);
30     ds);
31     h = t1.hours + t2.hours;
32     m = t1.minutes + t2.minutes;
33     s = t1.seconds + t2.seconds;
34     printf ("\nSum of the two time's is
35     %d:%d:%d",h,m,s);
36     return 0;
37 }
```

### Output

```
Enter 1st time.
Enter Hours: 1
Enter Minutes: 20
Enter Seconds: 20
The Time is 1:20:20

Enter the 2nd time.
Enter Hours: 2
Enter Minutes: 10
Enter Seconds: 10
The Time is 2:10:10
Sum of the two time's is 3:30:30
```

# Structure using Pointer

- Reference/address of structure object is passed as function argument to the definition of function.

## Program

```
1  #include <stdio.h>
2  struct student {
3      char name[20];
4      int rollno;
5      float cpi;
6  };
7  int main()
8  {
9      struct student *studPtr, stud1;
10     studPtr = &stud1;
11     printf("Enter Name: ");
12     scanf("%s", studPtr->name);
13     printf("Enter RollNo: ");
14     scanf("%d", &studPtr->rollno);
15     printf("Enter CPI: ");
16     scanf("%f", &studPtr->cpi);
17     printf("\nStudent Details:\n");
18     printf("Name: %s\n", studPtr->name);
19     printf("RollNo: %d", studPtr->rollno);
20     printf("\nCPI: %f", studPtr->cpi);
21     return 0;
22 }
```

## Output

```
Enter Name: ABC
Enter RollNo: 121
Enter CPI: 7.46

Student Details:
Name: ABC
RollNo: 121
CPI: 7.460000
```

# Nested Structure

- When a **structure contains another structure**, it is called **nested structure**.
- For example, we have two structures named **Address** and **Student**. To make Address nested to Student, we have to define Address structure before and outside Student structure and create an object of Address structure inside Student structure.

## Syntax

```
1 struct structure_name1
2 {
3     member1_declaration;
4     member2_declaration;
5     ...
6     memberN_declaration;
7 };
8 struct structure_name2
9 {
10     member1_declaration;
11     member2_declaration;
12     ...
13     struct structure1 obj;
14 };
```

## Write a program to read and display student information using nested of structure.

### Program

```
1  #include<stdio.h>
2  struct Address
3  {
4      char HouseNo[25];
5      char City[25];
6      char PinCode[25];
7  };
8  struct Student
9  {
10     char name[25];
11     int roll;
12     float cpi;
13     struct Address Add;
14 };
15 int main()
16 {
17     int i;
18     struct Student s;
19     printf("\n\tEnter Student Name : ");
20     scanf("%s",s.name);
21     printf("\n\tEnter Student Roll Number : ");
22     scanf("%d",&s.roll);
23     printf("\n\tEnter Student CPI : ");
24     scanf("%f",&s.cpi);
25     printf("\n\tEnter Student House No : ");
26     scanf("%s",s.Add.HouseNo);
27
28
29
30
31
32
33
34
35
36
37
38     printf("\n\tStudent Name : %s",s.name);
39     %s",s.Add.City);
40     printf("\n\tStudent Pincode :
41     %s",s.Add.PinCode);
42 return 0;
43 }
```

### Output

```
Details of Students
Student Name : aaa
Student Roll Number : 111
Student CPI : 7.890000
Student House No : 39
Student City : rajkot
Student Pincode : 360001
```

# Array of Structure

- It can be defined as the collection of multiple structure variables where each variable contains information about different entities.
- The array of structures in C are used to store information about **multiple entities of different data types**.

## Syntax

```
1 struct structure_name
2 {
3     member1_declaration;
4 member2_declaration; 5
6     ...
6     memberN_declaration;
7 } structure_variable[size];
```

## Write a program to read and display N student information using array of structure.

### Program

```
1  #include<stdio.h>
2  struct student {
3      char name[20];
4      int rollno;
5  float cpi;
6  };
7  int main( ) {
8      int i,n;
9      printf("Enter how many records u want to store : ");
10     scanf("%d",&n);
11     struct student sarr[n];
12     for(i=0; i<n; i++)
13     {
14         printf("\nEnter %d record : \n",i+1);
15         printf("Enter Name : ");
16         scanf("%s",sarr[i].name);
17         printf("Enter RollNo. : ");
18         scanf("%d",&sarr[i].rollno);
19         printf("Enter CPI : ");
20         scanf("%f",&sarr[i].cpi);
21     }
22     printf("\n\tName\tRollNo\tMarks\t\n");
23     for(i=0; i<n; i++) {
24         printf("\t%s\t\t%d\t\t%.2f\t\n", sarr[i].name,
25 sarr[i].rollno, sarr[i].cpi);
26     }
27     return 0;
28 }
```

### Output

Enter how many records u want to store : 3

Enter 1 record :

Enter Name : aaa

Enter RollNo. : 111

Enter CPI : 7.89

Enter 2 record :

Enter Name : bbb

Enter RollNo. : 222

Enter CPI : 7.85

Enter 3 record :

Enter Name : ccc

Enter RollNo. : 333

Enter CPI : 8.56

Name	RollNo	Marks
aaa	111	7.89
bbb	222	7.85
ccc	333	8.56

Write a program to declare time structure and read two different time period and display sum of it using function.

#### Program

```
1  #include<stdio.h>
2  struct Time {
3      int hours;
4      int minutes;
5  int seconds; 6
7  };
8  struct Time input(); // function declaration
9  int main()
10 {
11     struct Time t;
12     t=input();
13     printf("Hours : Minutes : Seconds\n %d : %d :
14 %d",t.hours,t.minutes,t.seconds);
15 return 0; 16 }
17 struct Time input() // function definition
18 {
19     struct Time tt;
20     printf ("Enter Hours: ");
21     scanf ("%d",&tt.hours);
22     printf ("Enter Minutes: ");
23     scanf ("%d",&tt.minutes);
24     printf ("Enter Seconds: ");
25     scanf ("%d",&tt.seconds);
26 return tt; // return structure variable 27 }
```

#### Output

```
Enter Hours: 1
Enter Minutes: 20
Enter Seconds: 20
Hours : Minutes : Seconds
1 : 20 : 20
```

# Unions

- Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location.
- The union can also be defined as many members, but only one member can contain a value at a particular point in time.
- Union is a user-defined data type, but unlike structures, they share the same memory location.

## Syntax

```
1 union structure_name1
2 {
3     member1_declaration;
4     member2_declaration;
5     ...
6     memberN_declaration;
7 };
8 union structure_name2
9 {
10    member1_declaration;
11    member2_declaration;
12    ...
13    union structure1 obj;
14 };
```

# Unions

- Access members of a union

- ➔ We use the . operator to access members of a union.
- ➔ And to access pointer variables, we use the -> operator..

In the above example,

- ➔ To access price for car1, car1.price is used.
- ➔ To access price using car3, either (\*car3).price or car3->price can be used.

## Syntax

```
1 union car
2 {
3     char name[50];
4     int price;
5 } car1, car2, *car3;
```

# Difference between unions and structures

- differences between structures and unions

→ Here, the size of sJob is 40 bytes because

- the size of name[32] is 32 bytes
- the size of salary is 4 bytes
- the size of workerNo is 4 bytes

## Output

```
size of union = 32
size of structure = 40
```

## Program

```
1 #include <stdio.h>
2 union unionJob
3 {
4     char name[32];
5     float salary;
6     int workerNo;
7 } uJob;
8
9 struct structJob
10 {
11     char name[32];
12     float salary;
13     int workerNo;
14 } sJob;
15 void main()
16 {
17     printf("size of union = %d bytes",
18     sizeof(uJob));
19     printf("\nsize of structure = %d
20 bytes", sizeof(sJob));
21 }
```

# Differences between Structures and Unions

	STRUCTURE	UNION
Keyword	The keyword <b>struct</b> is used to define a structure	The keyword <b>union</b> is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is <b>greater than or equal to the sum of sizes of its members.</b>	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of <b>union is equal to the size of largest member.</b>
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

# Introduction to Pointers



# What is Pointer?

- A normal variable is used to store value.
- A pointer is a variable that **store address / reference** of another variable.
- Pointer is **derived data type** in C language.
- A pointer contains the memory address of that variable as their value. Pointers are also called **address variables** because they contain the addresses of other variables.

# Advantages and Disadvantages of pointers :

- **Advantages (Benefits) of pointers :**

- ➔ Pointers provide direct access to memory
- ➔ Pointers provide a way to return more than one value to the functions
- ➔ Reduces the storage space and complexity of the program
- ➔ Reduces the execution time of the program
- ➔ Provides an alternate way to access array elements
- ➔ Pointers can be used to pass information back and forth between the calling function and called function.
- ➔ Pointers allows us to perform dynamic memory allocation and deallocation.
- ➔ Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.
- ➔ Pointers allows us to resize the dynamically allocated memory block.
- ➔ Addresses of objects can be extracted using pointers

- **Disadvantages (Drawbacks) of pointers :**

- ➔ Uninitialized pointers might cause segmentation fault.
- ➔ Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak
- ➔ Pointers are slower than normal variables.
- ➔ If pointers are updated with incorrect values, it might lead to memory corruption.

# Declaration & Initialization of Pointer

## Syntax

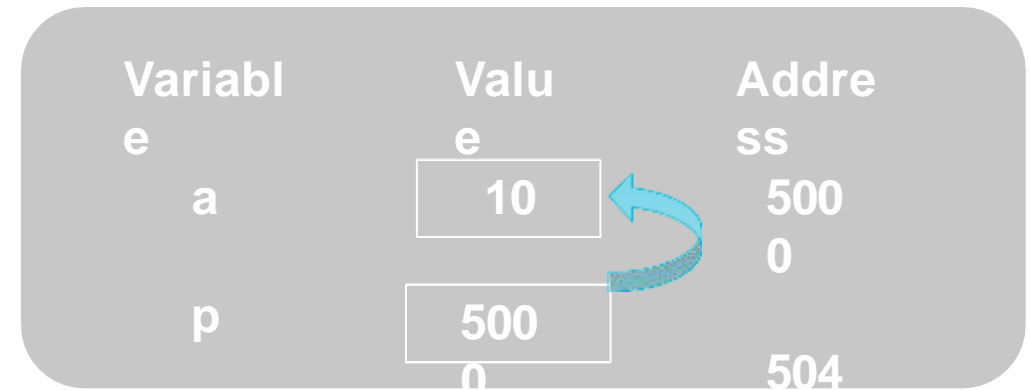
```
1 datatype *ptr_variablename;
```

## Example

```
1 void main()  
2 {  
3     int a=10, *p; // assign memory address of a  
4     // to pointer variable p  
5     p = &a;  
6     printf("%d %d %d", a, *p, p);  
7 }
```

## Output

```
10 10 5000
```



- **p** is integer pointer variable
- **&** is address of or referencing operator which returns memory address of variable.
- **\*** is indirection or dereferencing operator which returns value stored at that memory address.
- **&** operator is the inverse of **\*** operator
- **x = a** is same as **x = \*(&a)**

# Why use Pointer?

- C uses pointers to create **dynamic data structures**, data structures built up from blocks of memory allocated from the heap at run-time. Example linked list, tree, etc.
- C uses pointers to handle variable parameters passed to functions.
- Pointers in C provide an alternative way to **access information stored in arrays**.
- Pointer use in **system level programming** where memory addresses are useful. For example shared memory used by multiple threads.
- Pointers are used for file handling.
- This is the reason why C is versatile.

# Pointer to Pointer – Double Pointer

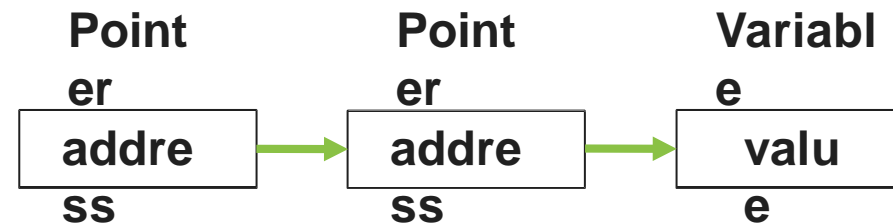
- Pointer holds the address of another variable of same type.
- When a pointer holds the **address of another pointer** then such type of pointer is known as **pointer-to-pointer** or **double pointer**.
- The first pointer contains the address of the second pointer, which points to the location that contains the actual value.

## Syntax

```
1 datatype **ptr_variablename;
```

## Example

```
1 int **ptr;
```



Write a program to print variable, address of pointer variable and pointer to pointer variable.

#### Program

```
1  #include <stdio.h>
2  int main () {
3      int var;
4      int *ptr;
5      int **pptr;
6      var = 3000;
7      ptr = &var; // address of var
8      pptr = &ptr; // address of ptr using address of operator &
9      printf("Value of var = %d\n", var );
10     printf("Value available at *ptr = %d\n", *ptr );
11         printf("Value available at **pptr = %d\n", **pptr);
12 return 0; 13 }
```

#### Output

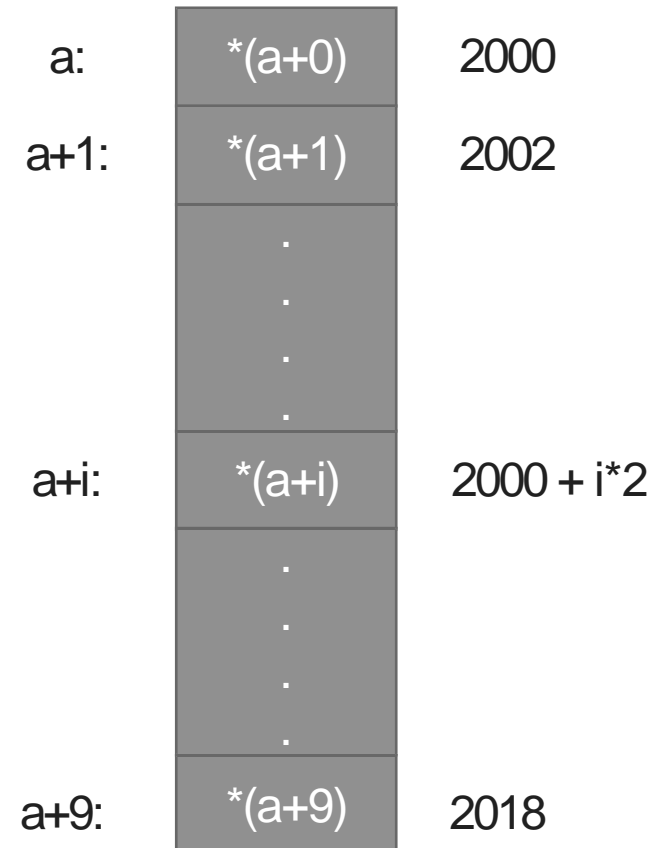
```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

# Relation between Array & Pointer

- When we declare an array, compiler allocates continuous blocks of memory so that all the elements of an array can be stored in that memory.
- The address of first allocated byte or the address of first element is assigned to an array name.
- Thus array name works as **pointer variable**.
- The address of first element is also known as **base address**.

# Relation between Array & Pointer – Cont.

- Example: `int a[10], *p;`
- `a[0]` is same as `*(a+0)`, `a[2]` is same as `*(a+2)` and `a[i]` is same as `*(a+i)`



# Array of Pointer

- As we have an array of char, int, float etc, same way we can have an array of pointer.
- Individual elements of an array will store the address values.
- So, an array is a collection of values of similar type. It can also be a collection of references of similar type known by single name.

## Syntax

```
1 datatype *name[size];
```

## Example

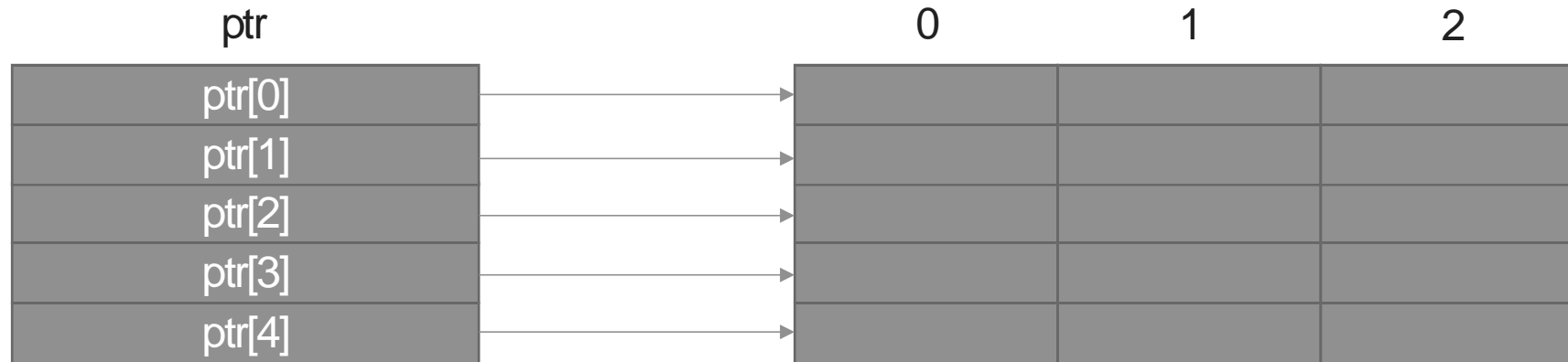
```
1 int *ptr[5]; //declares an array of integer pointer of size 5
```

# Array of Pointer – Cont.

- An array of pointers ptr can be used to point to different rows of matrix as follow:

## Example

```
1 for(i=0; i<5; i++)  
2 {  
3     ptr[i]=&mat[i][0];  
4 }
```



- By dynamic memory allocation, we do not require to declare two-dimensional array, it can be created dynamically using array of pointers.

Write a program to swap value of two variables using pointer / call by reference.

#### Program

```
1 int main()
2 {
3     int num1,num2;
4     printf("Enter value of num1 and num2: ");
5 scanf("%d %d",&num1, &num2); 6
7 //displaying numbers before swapping
8 printf("Before Swapping: num1 is: %d, num2 is: %d\n",num1,num2); 9
10 //calling the user defined function swap()
11 swap(&num1,&num2); 12
13 //displaying numbers after swapping
14     printf("After Swapping: num1 is: %d, num2 is: %d\n",num1,num2);
15 return 0; 16
    }
```

#### Output

```
Enter value of num1 and num2: 5
10
Before Swapping: num1 is: 5, num2 is: 10
After Swapping: num1 is: 10, num2 is: 5
```

# Pointer and Function

- Like normal variable, pointer variable can be passed as function argument and function can return pointer as well.
- There are two approaches to passing argument to a function:
  - ➔ Call by value
  - ➔ Call by reference / address

# Call by Value

- In this approach, the values are passed as function argument to the definition of function.

## Program

```
1 #include<stdio.h>
2 void fun(int,int);
3 int main()
4 {
5     int A=10,B=20;
6     printf("\nValues before calling %d, %d",A,B);
7     fun(A,B);
8     printf("\nValues after calling %d, %d",A,B);
9 return 0; 10 }
11 void fun(int X,int Y)
12 {
13     X=11;
14     Y=22;
15 }
```

## Output

```
Values before calling 10, 20
Values after calling 10, 20
```

Address	48252	24688		
Value	10	20	10 <sup>11</sup>	-20 <sup>22</sup>
Variable	A	B	X	Y

# Call by Reference / Address

- In this approach, the references / addresses are passed as function argument to the definition of function.

## Program

```
1  #include<stdio.h>
2  void fun(int*,int*);
3  int main()
4  {
5      int A=10,B=20;
6      printf("\nValues before calling %d, %d",A,B);
7      fun(&A,&B);
8      printf("\nValues after calling %d, %d",A,B);
9  return 0;
10 }
11 void fun(int *X,int *Y)
12 {
13     *X=11;
14     *Y=22;
15 }
```

## Output

```
Values before calling 10, 20
Values after calling 11, 22
```

Address	48252	24688		
Value	-10 <sup>11</sup>	-20 <sup>22</sup>	48252	24688
Variable	A	B	*X	*Y

# Pointer to Function

- Every function has reference or address, and if we know the reference or address of function, we can access the function using its **reference or address**.
- This is the way of accessing function using pointer.

## Syntax

```
1 return-type (*ptr-function)(argument list);
```

- **return-type**: Type of value function will return.
- **argument list**: Represents the type and number of value function will take, values are sent by the calling statement.
- **(\*ptr-function)**: The parentheses around **\*ptr-function** tells the compiler that it is pointer to function.
- If we write **\*ptr-function** without parentheses then it tells the compiler that **ptr-function** is a function that will return a pointer.

Write a program to sum of two numbers using pointer to function.

#### Program

```
1  #include<stdio.h>
2  int Sum(int,int);
3  int (*ptr)(int,int);
4  int main()
5  {
6      int a,b,rt;
7      printf("\nEnter 1st number : ");
8      scanf("%d",&a);
9      printf("\nEnter 2nd number : ");
10     scanf("%d",&b);
11     ptr = Sum;
12     rt = (*ptr)(a,b);
13     printf("\nThe sum is : %d",rt);
14     return 0;
15 }
16 int Sum(int x,int y)
17 {
18     return x + y;
19 }
```

#### Output

Enter 1st number : 5

Enter 2nd number : 10

The sum is : 15

# Enumerated Types

- **Enum:**

- ➔ The enumerated type is a user-defined type based on the standard integer type.
- ➔ In an enumerated type, each integer value is given an identifier called an enumeration constant.
- ➔ **Declaring an Enumerated Type:**
- ➔ To declare an enumerated type, we must declare its identifier and its values. Because it is derived from integer type, its operations are the same as for integers.
- ➔ Syntax for defining an enumerated type is as follows,

```
enum typeName
{
    member1;
    member2;
    ....
    ....
};
```

- ➔ Where enum is the keyword that tells the compiler about enumerated type definition, enum type\_Name together represent the user defined data type and member1, member2... are integer constants but represented using descriptive names. These are called enumerator constants or enumerators.

# Enumerated Types

- **Enum:**

- The definition is terminated with a semicolon.
- The syntax for declaring the variables are shown below:

```
enum typeName var;
```

**Following are some of the examples of enumerator type:**

```
enum color
{
    RED,
    BLUE,
    GREEN
};
enum color c1, c2;
```

```
enum days
{
    SUNDAY,
    MONDAY
    ,
    ...
    SATURDAY
} d1;
```

# Enumerated Types

- **Assigning Values to Enumerated Types**

- ➔ After an enumerated variable has been declared, we can store values in it.
- ➔ While, the compiler automatically assigns values to enumerated types starting with 0, the next values are initialized with a value by adding 1 to previous value.
- ➔ For example,

```
enum color {RED, BLUE, GREEN, WHITE};
```

- ➔ Here red representing the value 0, blue is 1, green is 2, white is 3.
- ➔ You can also create variables from the enumerated type.
- ➔ For example, **enum color skyColor;**
- ➔ We can override it and assign our own values.
- ➔ For example, to make JAN start with 1 we could use the following declaration.

- ➔ **enum month**

```
{  
    JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC  
}m1;
```

- ➔ Note that we need not to assign every enumerator constant value. If we omit the initializes, the compiler assigns the next value by adding 1.

# Enumerated Types

- **Assigning Values to Enumerated Types**

→ Consider the following enumerated declaration,

**enum days**

```
{  
    sun=3, mon, tue, wed=0, thu, fri, sat  
} d1, d2;
```

**Output:**

```
0      1      2  
2
```

**//Example program to demonstrate enum**

```
#include<stdio.h> main()  
{  
    enum color  
    {  
        RED,  
        GREEN,  
        BLUE  
    }c1;  
    printf("%d%d%d",RED,GREEN,BLUE);  
    c1 = BLUE;  
    printf("\n%d",c1);  
}
```

***Thank  
You***



**D. SRINIVAS**

Computer Science and Engineering Department

✉ [srinivascsedept@gmail.com](mailto:srinivascsedept@gmail.com)

☎ +91-9347556447



# UNIT-3

# Functions and

# Dynamic Memory

# Allocation



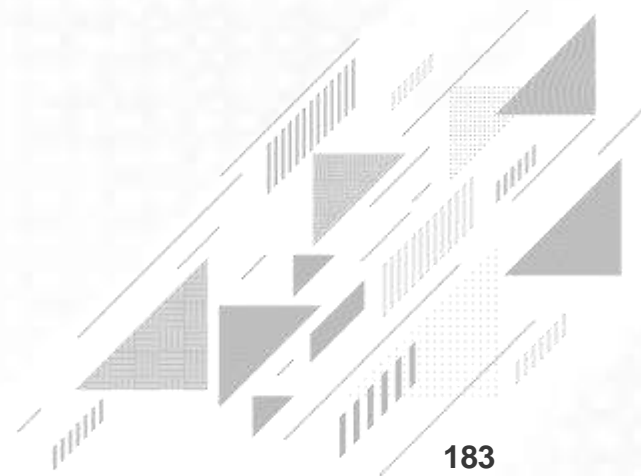
**D. SRINIVAS**

Department of CSE

[www.srinivas-materials.blogspot.com](http://www.srinivas-materials.blogspot.com)

✉ [srinivascsedpt@gmail.com](mailto:srinivascsedpt@gmail.com)

☎ +91 9347556447





## Outline

- **Functions:**

- ↳ Designing structured programs,
- ↳ Declaring a function, Signature of a function,
- ↳ Parameters and return type of a function,
- ↳ passing parameters to functions,
- ↳ call by value Passing arrays to functions,
- ↳ passing pointers to functions, idea of call by reference,
- ↳ Some C standard functions and libraries

- **Recursion:**

- ↳ Simple programs,
- ↳ such as Finding Factorial,
- ↳ Fibonacci series etc.,
- ↳ Limitations of Recursive functions

- **Dynamic memory allocation:**

- ↳ Allocating and freeing memory,
- ↳ Allocating memory for arrays of different datatypes.

# Introduction to Functions



# Function

- A **function** is a group of programming statements that perform a specific task.
- It divides a large program into smaller parts.
- A **function** is something like hiring a person to do a specific job for you.
- Every C program can be thought of as a collection of these functions.
- Program execution in C language starts from the main function.

Syntax

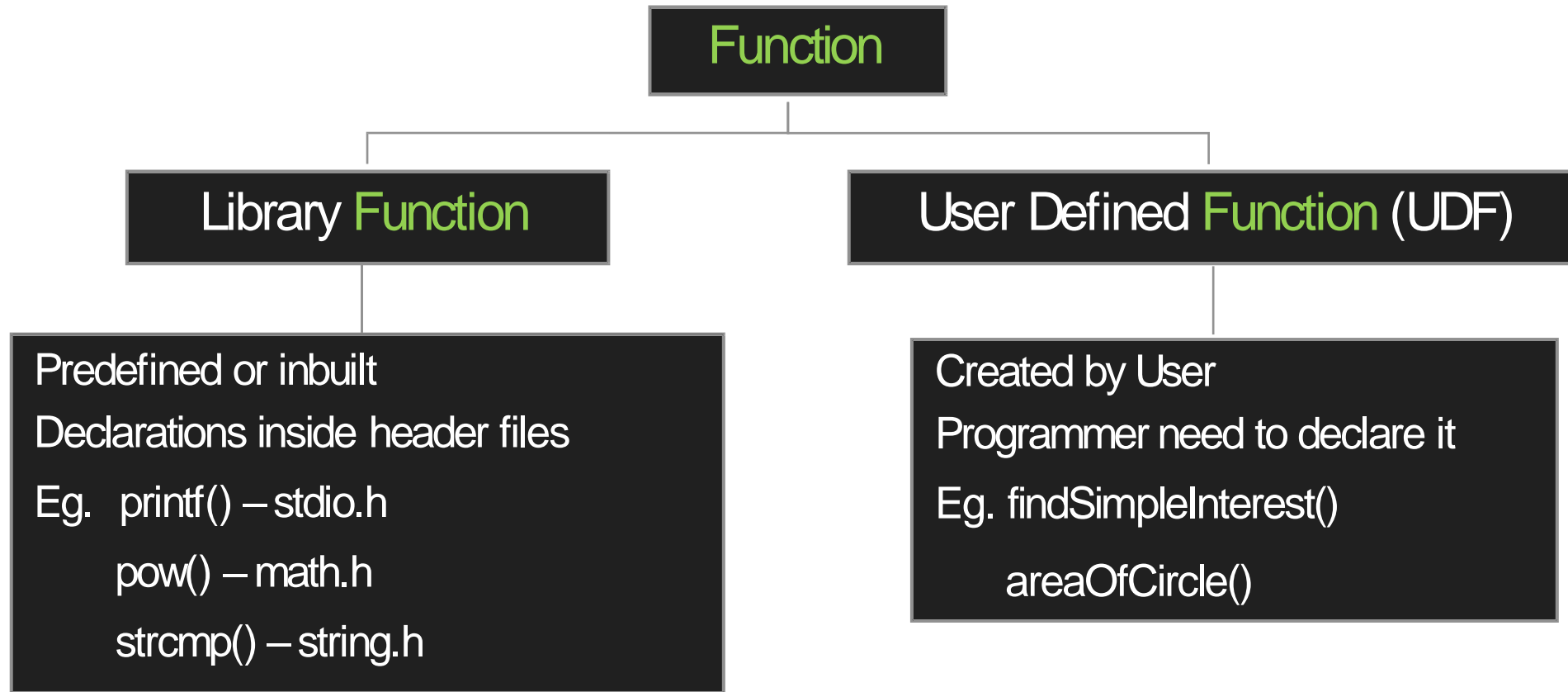
```
void main()  
{  
    // body part  
}
```

- Why **function** ?
  - ➔ Avoids rewriting the same code over and over.
  - ➔ Using functions it becomes easier to write programs and keep track of what they doing.

# Advantages of Function

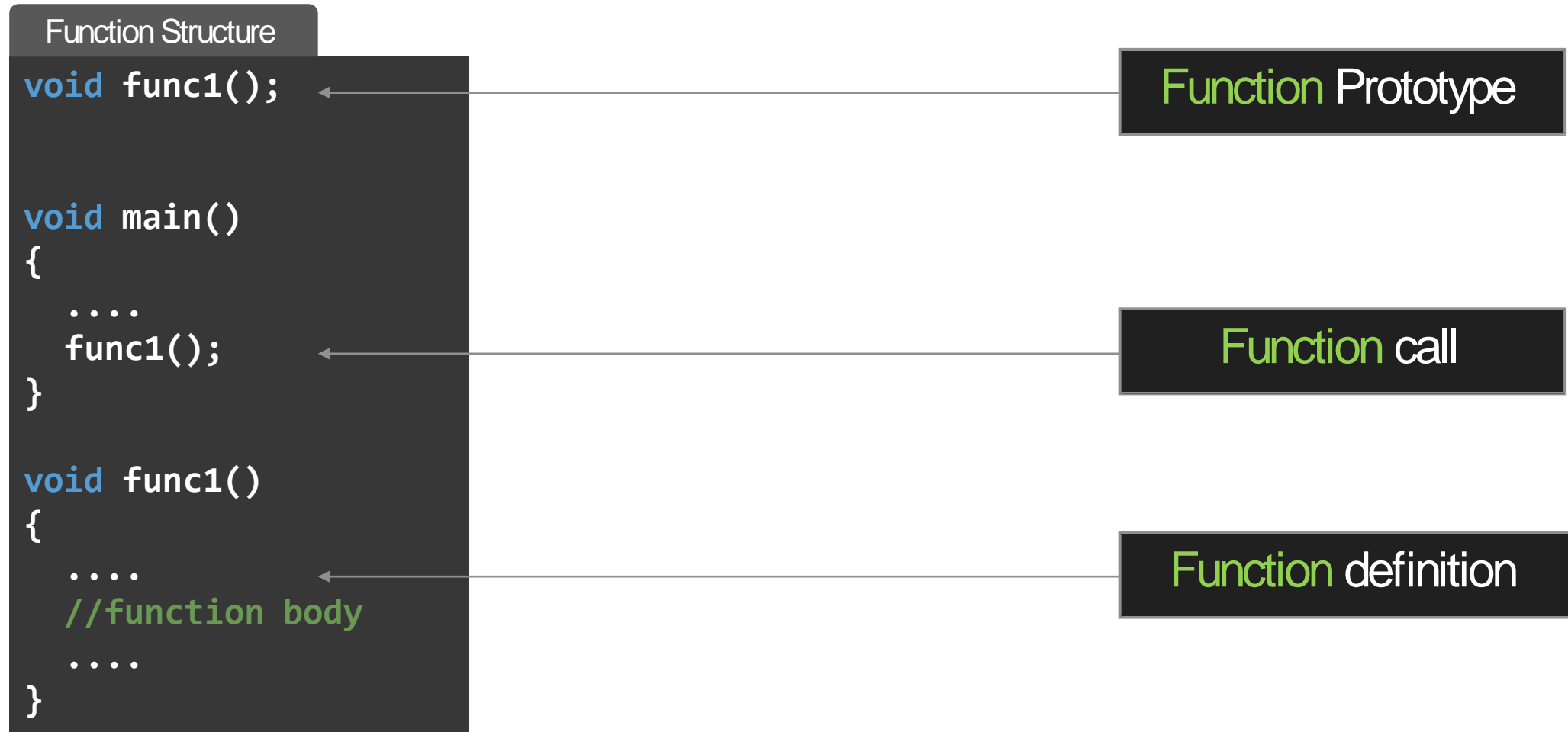
- Using **function** we can avoid rewriting the same logic or code again and again in a program.
- We can track or understand large program easily when it is divide into **functions**.
- It provides reusability.
- It help in testing and debugging because it can be tested for errors individually in the easiest way.
- Reduction in size of program due to code of a **function** can be used again and again, by calling it.

# Types of Functions



# Program Structure for Function

- When we use a user-defined function program structure is divided into three parts.



# Function Phototype

- A **function** Prototype also know as function declaration.
- A **function** declaration tells the compiler about a function name and how to call the function.
- It defines the function before it is being used or called.
- A **function** prototype needs to be written at the beginning of the program.

## Syntax

```
return-type function-name (arg-1, arg 2, ...);
```

## Example

```
void addition(int, int);
```

# Function Definition

- A **function** definition defines the functions header and body.
- A **function** header part should be identical to the function prototype.
  - ➔ Function return type
  - ➔ Function name
  - ➔ List of parameters
- A **function** body part defines function logic.
  - ➔ Function statements

## Syntax

```
return-type function-name (arg-1, arg 2, ...)  
{  
    //... Function body  
}
```

## Example

```
void addition(int x, int y)  
{  
    printf("Addition  
is=%d", (x+y)); }  
}
```

# Program on Function

- WAP to add two number using add(int, int) Function

## Program

```
#include <stdio.h>
void add(int, int); // function declaration

void main()
{
    int a = 5, b = 6;
    add(a, b); // function call
}

void add(int x, int y) // function definition
{
    printf("Addition is = %d", x + y);
}
```

## Output

```
Addition is = 11
```

# Actual parameters and Formal parameters

- Values that are passed to the called function from the main function are known as **Actual** parameters.
- The variables declared in the function prototype or definition are known as **Formal** parameters.
- When a method is called, the **formal** parameter is temporarily "bound" to the **actual** parameter.

## Actual parameters

```
void main()
{
    int a = 5, b = 6;
    add(a, b); // a and b are the
               // actual parameters in this call.
}
```

## Formal parameters

```
void add(int x, int y) // x and y are
                        // formal parameters.
{
    printf("Addition is = %d", x + y);
}
```

# Programs on Functions

- WAP to find Factorial of a Number.

## Program

```
#include <stdio.h>
int fact(int);
int main()
{
    int n, f;
    printf("Enter the number :\n");
    scanf("%d", &n);
    f = fact(n);
    printf("factorial = %d", f);
}
int fact(int n)
{
    int i, fact = 1;
    for (i = 1; i <= n; i++)
        fact = fact * i;
    return fact;
}
```

## Output

```
Enter the number :
5
factorial = 120
```

# Programs on Functions

- WAP to check Number is Prime or not

## Program

```
#include <stdio.h>
int checkPrime(int);
void main()
{
    int n1, prime;
    printf("Enter the number :");
    scanf("%d", &n1);
    prime = checkPrime(n1);
    if (prime == 1)
        printf("The number %d is a prime\n", n1);
    else
        printf("The number %d is not a\n", n1);
}
```

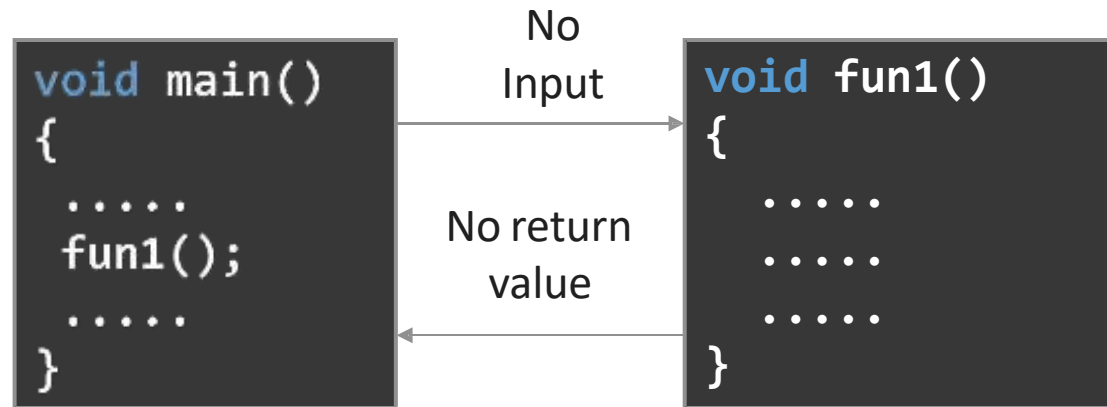
## Program contd.

## Output

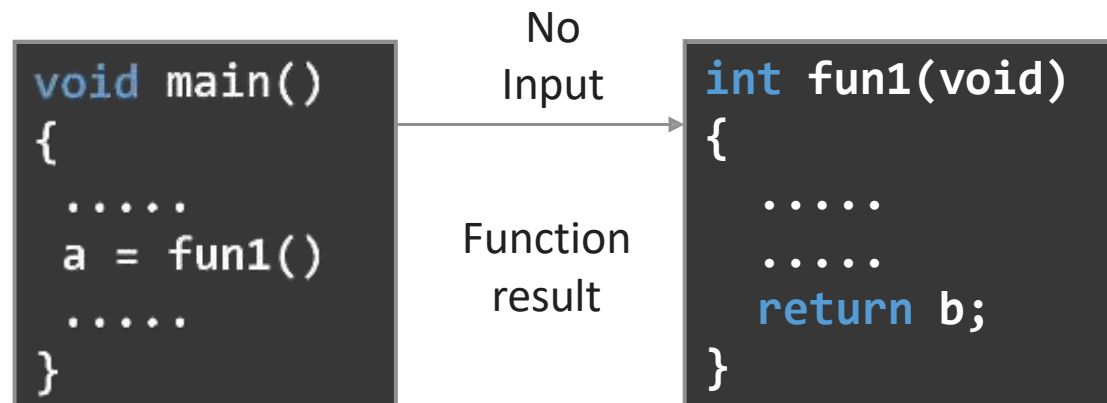
```
Enter the number :7
The number 7 is a prime number.
```

# Category of Function

## (1) Function with no argument and no return value

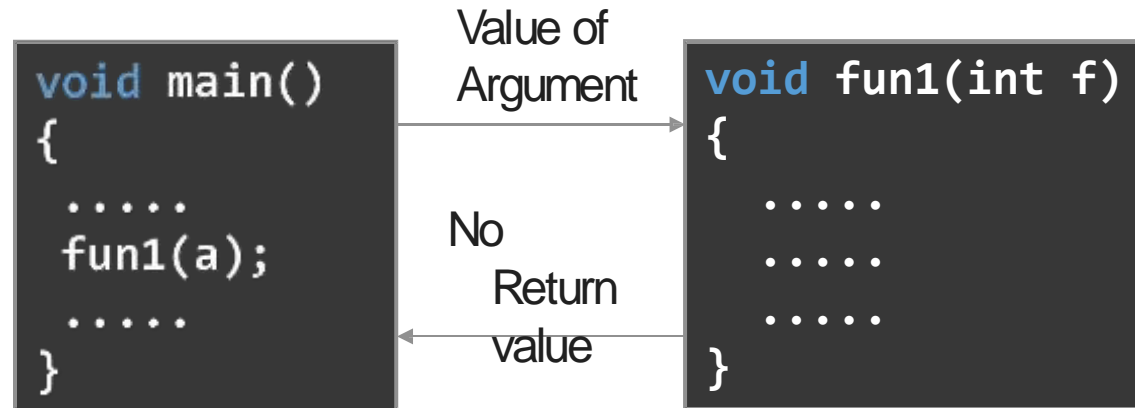


## (2) Function with no argument and returns value

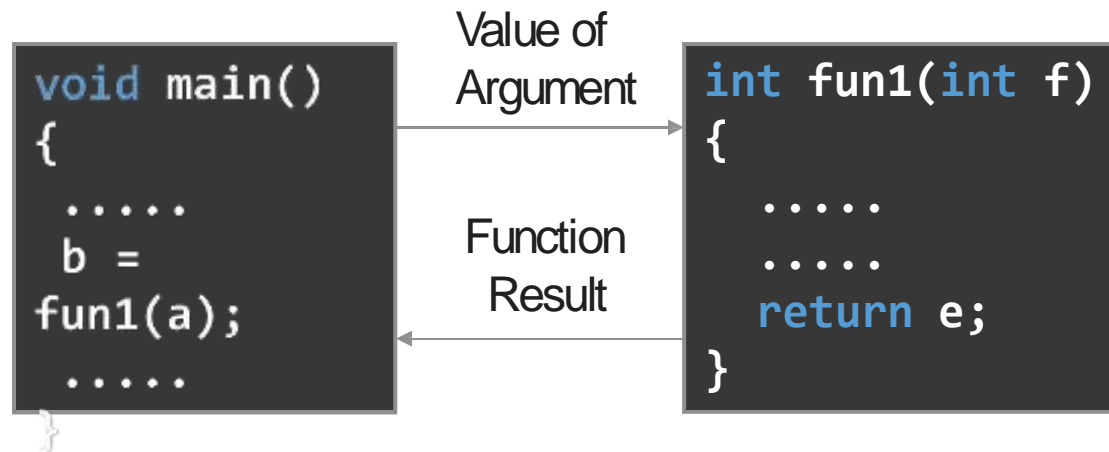


# Category of Function cont...

## (3) Function with argument and but no return value



## (4) Function with argument and returns value



# Passing Parameters to Functions

- There are two ways of passing parameters to the functions.

1. Call by value and 2. Call by reference

- **Call by value:**

- ➔ When a function is called with actual parameters, the values of actual parameters are copied into the formal parameters.
- ➔ If the values of the formal parameters changes in the function, the values of the actual parameters are not changed.
- ➔ This way of passing parameters is called call by value (pass by value).
- ➔ In the below example, the values of the arguments to swap () 10 and 20 are copied into the parameters x and y.
- ➔ Note that the values of x and y are swapped in the function.
- ➔ But, the values of actual parameters remain same before swap and after swap.

- **Note:** In call by value any changes done on the formal parameter will not affect the actual parameters.

# Example Program on call by value

```
#include<stdio.h>

void swap (int , int );      /*function prototype */

void main ()

{   int a=10, b=20;
    swap (a, b);  /*function calling*/
    printf ("From main The Values of a and b a=%d, b=%d ", a, b);
}

void swap (int x, int y)      /* function definition */
{   int temp;
    temp=x;
    x=y;
    y=temp;
    printf ("\n The Values of a and b after swapping a=%d, b =%d", x, y);
}
```

# Passing Parameters to Functions

- **Call by reference:**

- ➔ When a function is called with actual parameters, the values of actual parameters are copied into the formal parameters.
- ➔ If the values of the formal parameters change in the function, the values of the actual parameters are not changed.
- ➔ This way of passing parameters is called call by reference (pass by address).
- ➔ In the below example, the values of the arguments to swap () 10 and 20 are copied into the parameters x and y.
- ➔ Note that the values of x and y are swapped in the function.
- ➔ But, the values of actual parameters remain the same before swap and after swap.

- **Note:** In call by reference any changes done on the formal parameter will affect the actual parameters.

# Example Program on call by reference

```
#include<stdio.h>

void swap (int * , int * );    /*function prototype */

void main ()

{   int a=10, b=20;

    swap (&a, &b); /*function calling*/

    printf ("From main The Values of a and b a=%d, b=%d ", a, b);

}

void swap (int *x, int *y)    /* function definition */

{   int temp;

    temp=*x;

    *x=*y;

    *y=temp;

    printf ("\n The Values of a and b after swapping a=%d, b =%d", x, y);

}
```

# Differences between Call by Value and Call by Reference

Call by Value	Call by Reference
When Function is called the values of variables are passed.	When a function is called address of variables is passed.
Formal parameters contain the value of actual parameters.	Formal parameters contain the address of actual parameters.
Change of formal parameters in the function will not affect the actual parameters in the calling function	The actual parameters are changed since the formal parameters indirectly manipulate the actual parameters
Execution is slower since all the values have to be copied into formal parameters.	Execution is faster since only addresses are copied.

# Introduction to Recursion



# Recursion

- Any function which calls itself is called **recursive function** and such function calls are called **recursive calls**.
- **Recursion** cannot be applied to all problems, but it is more useful for the tasks that can be defined in terms of a similar subtask.
- It is idea of representing problem a with smaller problems.
- Any problem that can be solved **recursively** can be solved iteratively.
- When **recursive** function call itself, the memory for called function allocated and different copy of the local variable is created for each function call.
- Some of the problem best suitable for recursion are
  - ➔ Factorial
  - ➔ Fibonacci
  - ➔ Tower of Hanoi

# Working of Recursion

Working

```
void  
func1();  
  
void  
main()  
{  
    ....  
    func1();  
    ....  
}  
void  
func1()  
{  
    ....  
    func1();  
    ....  
}
```

Function  
call

Recursive  
function call

# Properties Recursion

- A **recursive** function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have.
- **Base Case or Base criteria**
  - ➔ It allows the recursion algorithm to stop.
  - ➔ A base case is typically a problem that is small enough to solve directly.
- **Progressive approach**
  - ➔ A recursive algorithm must change its state in such a way that it moves forward to the base case.

# Programs on Recursion

- Factorial of a Number using Recursion

## Program

```
#include <stdio.h>
int fact(int);
void main()
{
    int n, f;
    printf("Enter the
    number:\n");
    scanf("%d", &n); f =
    fact(n);
    printf("factorial = %d", f);
}
int fact(int n)
{
    if (n == 0)
        return 1; else
    if (n == 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

## Output

```
Enter the
number: 5
factorial = 120
```

## ► Fibonacci Series of a Number using Recursion

## Program

```
#include <stdio.h>
int fibonacci(int);
void main()
{
    int n, m = 0, i;
    printf("Enter
    Total terms\n");
    scanf("%d", &n);
    printf("Fibonacci
    series\n");
    for (i = 1; i <=
    n; i++)
    { printf("%d ",
    fibonacci(m));
        m++;
    }
}
```

```
int fibonacci(int n)
{
    if (n == 0 || n == 1)
        return n;
    else
        return
        (fibonacci(n - 1) +
        fibonacci(n - 2));
}
```

## Output

```
Enter Total terms
5
Fibonacci series
0 1 1 2 3
```

# Iteration vs Recursion

ITERATION	RECURSION
Iteration explicitly uses repetition structure.	Recursion achieves repetition by calling the same function repeatedly.
Iteration is terminated when the loop condition fails	Recursion is terminated when base case is satisfied.
May have infinite loop if the loop condition never fails	Recursion is infinite if there is no base case or if base case never reaches.
Iterative functions execute much faster and occupy less memory space.	Recursive functions are slow and take a lot of memory space compared to iterative functions
No. of CPU Cycles repeated	No. of times Function executed

# Dynamic Memory Allocation



# Dynamic Memory Allocation(DMA)

- If memory is allocated at runtime (during execution of program) then it is called dynamic memory.
- It allocates memory from **heap** (*heap*: it is an empty area in memory)
- Memory can be accessed only through a pointer.

When DMA is needed?

- It is used when number of variables are not known in advance or **large** in size.
- Memory can be allocated at any time and can be released at any time during runtime.

# 1.malloc() Function

- **malloc** ( ) is used to allocate a fixed amount of memory during the execution of a program.
- **malloc** ( ) allocates **size\_in\_bytes** of memory from heap, if the allocation succeeds, a pointer to the block of memory is returned else **NULL** is returned.
- Allocated memory space may not be contiguous.
- Each block contains a **size**, a pointer to the next block, and the space itself.
- The blocks are kept in ascending order of storage address, and the last block points to the first.
- The memory is not initialized.

Syntax	Description
<pre>ptr_var = (cast_type *) malloc (size_in_bytes);</pre>	<p>This statement returns a pointer to <b>size_in_bytes</b> of uninitialized storage, or <b>NULL</b> if the request cannot be satisfied.</p> <p><b>Example:</b> <code>fp = (int *)malloc(sizeof(int) *20);</code></p>

# Write a C program to allocate memory using malloc.

## Program

```
#include <stdio.h>
void main()
{
    int *fp; //fp is a pointer variable
    fp = (int *)malloc(sizeof(int)); //returns a pointer to int size storage
    *fp = 25; //store 25 in the address pointed by fp
    printf("%d", *fp); //print the value of fp, i.e. 25
    free(fp); //free up the space pointed to by fp
}
```

## Output

25

# 2.calloc() function

- **calloc()** is used to allocate a block of memory during the execution of a program
- **calloc()** allocates a region of memory to hold **no\_of\_blocks** of **size\_of\_block** each, if the allocation succeeds then a pointer to the block of memory is returned else **NULL** is returned.
- The memory is initialized to **ZERO**.

Syntax	Description
<pre>ptr_var = (cast_type *) calloc (no_of_blocks, size_of_block);</pre>	<p>This statement returns a pointer to <b>no_of_blocks</b> of size <b>size_of_blocks</b>, it returns <b>NULL</b> if the request cannot be satisfied.</p> <p><b>Example:</b> <code>int n = 20; fp = (int *)calloc(n, sizeof(int));</code></p>

# Write a C program to allocate memory using calloc.

## Program

```
#include <stdio.h>
void main()
{
    int i, n; //i, n are integer variables
    int *fp; //fp is a pointer variable
    printf("Enter how many numbers: ");
    scanf("%d", &n);
    fp = (int *)calloc(n, sizeof(int)); //calloc returns a pointer to n blocks
    for(i = 0; i < n; i++) //loop through until all the blocks are read
    {
        scanf("%d", fp); //read and store into location where fp points
        fp++; //increment the pointer variable
    }
    free(fp); //frees the space pointed to by fp
}
```

# 3.realloc() function

- **realloc()** changes the size of the object pointed to by pointer fp to specified size.
- The contents will be unchanged up to the minimum of the old and new sizes.
- If the new size is larger, the new space will be uninitialized.
- **realloc()** returns a pointer to the new space, or **NULL** if the request cannot be satisfied, in which case \*fp is unchanged.

Syntax	Description
<pre>ptr_var = (cast_type *) realloc (void *fp, size_t);</pre>	<p>This statement returns a pointer to new space, or NULL if the request cannot be satisfied.</p> <p><b>Example:</b> <code>fp = (int *)realloc(fp,sizeof(int)*20);</code></p>

# Write a C program to allocate memory using realloc.

## Program

```
#include <stdio.h>
void main()
{
    int *fp; //fp is a file pointer
    fp = (int *)malloc(sizeof(int)); //malloc returns a pointer to int size storage
    *fp = 25; //store 25 in the address pointed by fp
    fp =(int *)realloc(fp, 2*sizeof(int)); //returns a pointer to new space
    printf("%d", *fp); //print the value of fp
    free(fp); //free up the space pointed to by fp
}
```

## Output

25

## 4.free() function

- **Free()** deallocates the space pointed to by fp.
- It does nothing if fp is **NULL**.
- fp must be a pointer to space previously allocated by **calloc**, **malloc** or **realloc**.

Syntax	Description
<code>void free(void *);</code>	<p>This statement free up the memory not needed anymore.</p> <p><b>Example:</b> <code>free(fp);</code></p>

***Thank  
You***



**D. SRINIVAS**

Computer Science and Engineering Department

✉ [srinivascsedept@gmail.com](mailto:srinivascsedept@gmail.com)

☎ +91-9347556447



# UNIT-4

# Algorithms ,

# Searching and

# Sorting



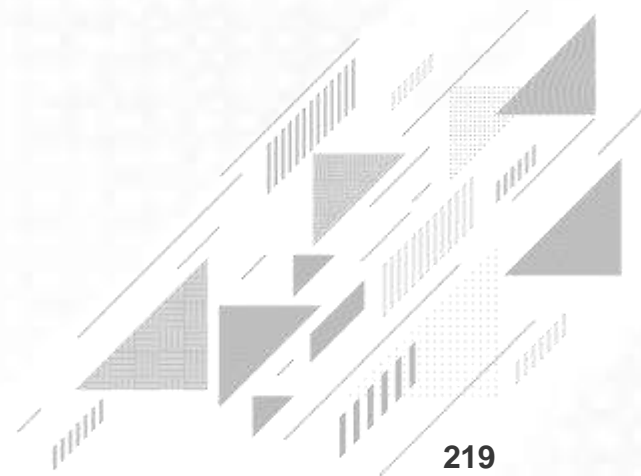
**D. SRINIVAS**

Department of CSE

[www.srinivas-materials.blogspot.com](http://www.srinivas-materials.blogspot.com)

✉ [srinivascsedpt@gmail.com](mailto:srinivascsedpt@gmail.com)

☎ +91 9347556447





## Outline

- **Algorithms:**

- ↪ Algorithms for finding roots of a quadratic equations, finding minimum and maximum numbers of a given set, finding if a number is prime number, etc.

- **Sorting:**

- ↪ Basic searching in an array of elements (linear and binary search techniques),
  - ↪ Basic algorithms to sort array of elements (Bubble, Insertion and Selection sort algorithms),
  - ↪ Basic concept of order of complexity through the example programs

# Introduction to Algorithms



# Algorithms

- **Algorithm:** It is an **ordered sequence** of **unambiguous** and **well-defined instructions** that **performs some task** and **halts in finite time**.
- Let's examine the four parts of this definition more closely.
  1. **Ordered Sequence:** You can number the step.
  2. **Unambiguous** and well defined instructions: Each instruction should be clear, well understand.
  3. **Performs** some task
  4. **Halts in finite time:** Algorithm must terminate at some point.
- **Properties of an Algorithm:-**
  1. **Finiteness:** An algorithm must terminate in a finite number of steps.
  2. **Definiteness:** Each step of an algorithm must be precisely and unambiguously stated.
  3. **Effectiveness:** Each step must be effective, and can be performed exactly in a finite amount of time.
  4. **Generality:** The algorithm must be complete in itself.
  5. **Input/Output:** Each algorithm must take zero, one or more inputs and produces one or more output.

# Algorithm to find all the roots of a quadratic equation

Start:

Step1: Input the value of a, b, c.

Step2: Calculate  $d = b^2 - 4ac$

Step3: If ( $d < 0$ )

Step3.1: Display "Roots are Imaginary " calculate  $r1 = \frac{-b + i \sqrt{d}}{2a}$  and  $r2 = \frac{-b - i \sqrt{d}}{2a}$ .

Step4: else if ( $d = 0$ )

Step4.1: Display "Roots are Equal" and calculate  $r1 = r2 = \frac{-b}{2a}$

Step5: else

Step5.1: Display "Roots are real" and  
calculate  $r1 = \frac{-b + \sqrt{d}}{2a}$  and  $r2 = \frac{-b - \sqrt{d}}{2a}$

Step4: Print r1 and r2.

Stop:

# Algorithm to find the minimum and maximum numbers in a given set of numbers

**Start:**

**Step1:** Initialize two variables "min" and "max" to the first element in the set.

**Step2:** Iterate through the rest of the set, comparing each element to the current values of "min" and "max".

**Step3:** If the current element is smaller than "min", set "min" to the current element.

**Step4:** If the current element is larger than "max", set "max" to the current element.

**Step5:** After iterating through the entire set, "min" and "max" will contain the minimum and maximum values, respectively.

**Stop:**

# Algorithm to Find Prime Number

Start:

Step1: Take num as input.

STEP 2: Initialize a variable temp to 0.

STEP 3: Iterate a “for” loop from 2 to num/2.

STEP 4: If num is divisible by loop iterator, then increment temp.

STEP 5: If the temp is equal to 0,

Return “Num IS PRIME”.

Step6: Else,

Return “Num IS NOT PRIME”.

Stop:

# Introduction to Searching & Sortings



# SEARCHING

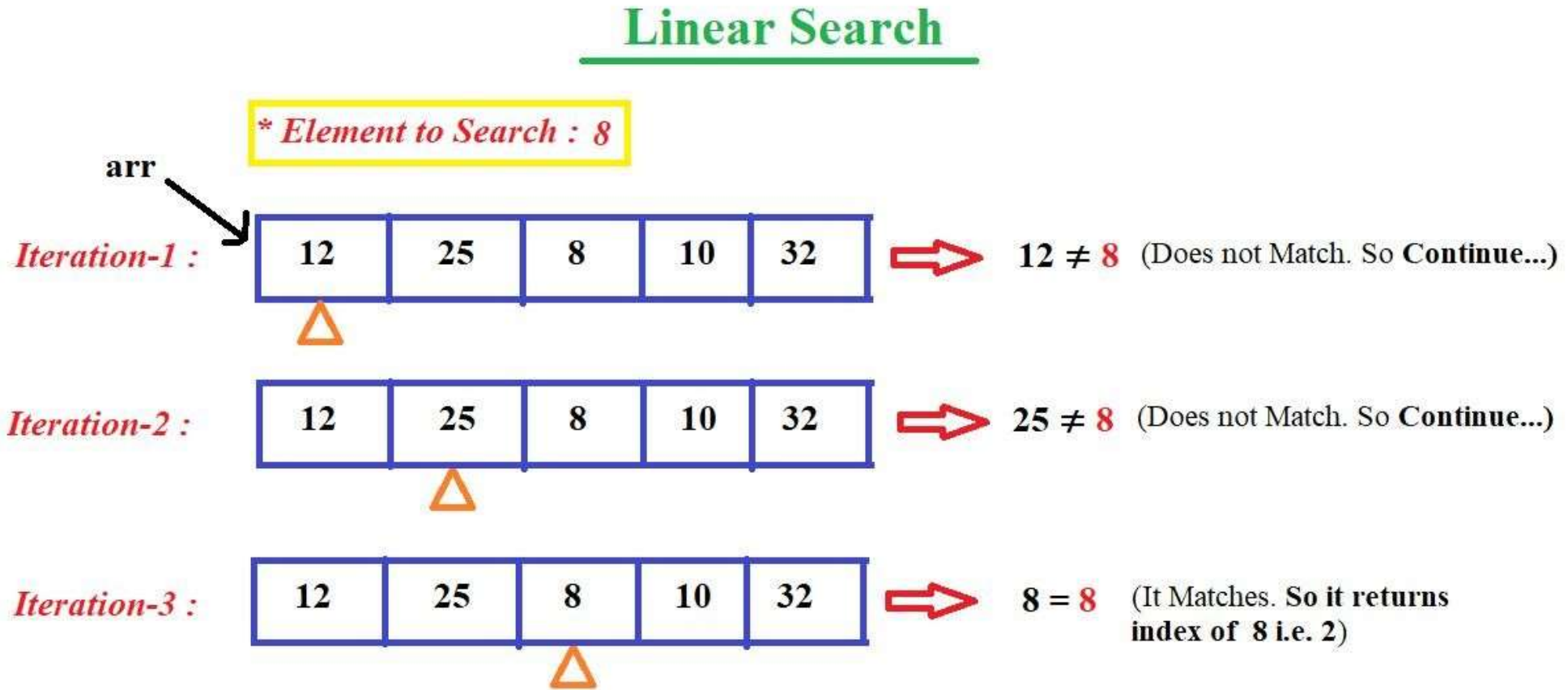
- Searching is an operation or a technique that helps find the place of a given element or value in the list.
- Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not.
- Some of the standard searching techniques that are being followed in data structures are listed below:
  1. Linear Search
  2. Binary Search

# Linear Search

- The algorithm proceeds as follows:
  1. Start at the first element of the list.
  2. Compare the current element with the target value.
  3. If the current element is equal to the target value, return the index of the current element.
  4. If the end of the list is reached without finding the target value, return -1 to indicate that the target value was not found.
- **Features of Linear Search Algorithm**
  1. It is used for unsorted and unordered small list of elements.
  2. It has a time complexity of  $O(n)$ , which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
  3. It has a very simple implementation.

# Linear Search

- Example



# Linear Search

- Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.
- It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns the index of the element in the array, else it return -1.
- Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Algorithm Linear Search ( Array A, Value x)

Step 1: Set i to 1

Step 2: if  $i > n$  then go to step 7

Step 3: if  $A[i] = x$  then go to step 6

Step 4: Set i to  $i + 1$

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8:

Exit

# Write a C program to find key element in the list using Linear Search.

## Program

```
#include <stdio.h>
int main()
{
    int a[10], i, item, n;
    printf("\nEnter number of elements of an array:\n");
    scanf("%d", &n);
    printf("\nEnter elements: \n");
    for (i=0; i<n; i++)
        scanf("%d", &a[i]);
    printf("\nEnter item to search: ");
    scanf("%d", &item);
    for (i=0; i<=9; i++)
        if (item == a[i])
        {
            printf("\nItem found at location %d", i+1);
            break;
        }
    if (i > 9)
        printf("\nItem does not exist.");
    return 0;
}
```

## Output

Enter number of elements of an array:

8

Enter elements:

2 3 5 7 8 6 4 1

Enter item to search: 1

Item found at location 8

# Binary Search

- Binary Search is used with sorted array or list.
- In binary search, we follow the following steps:
  - ➔ 1. We start by comparing the element to be searched with the element in the middle of the list/array.
  - ➔ 2. If we get a match, we return the index of the middle element.
  - ➔ 3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
  - ➔ 4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
  - ➔ 5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

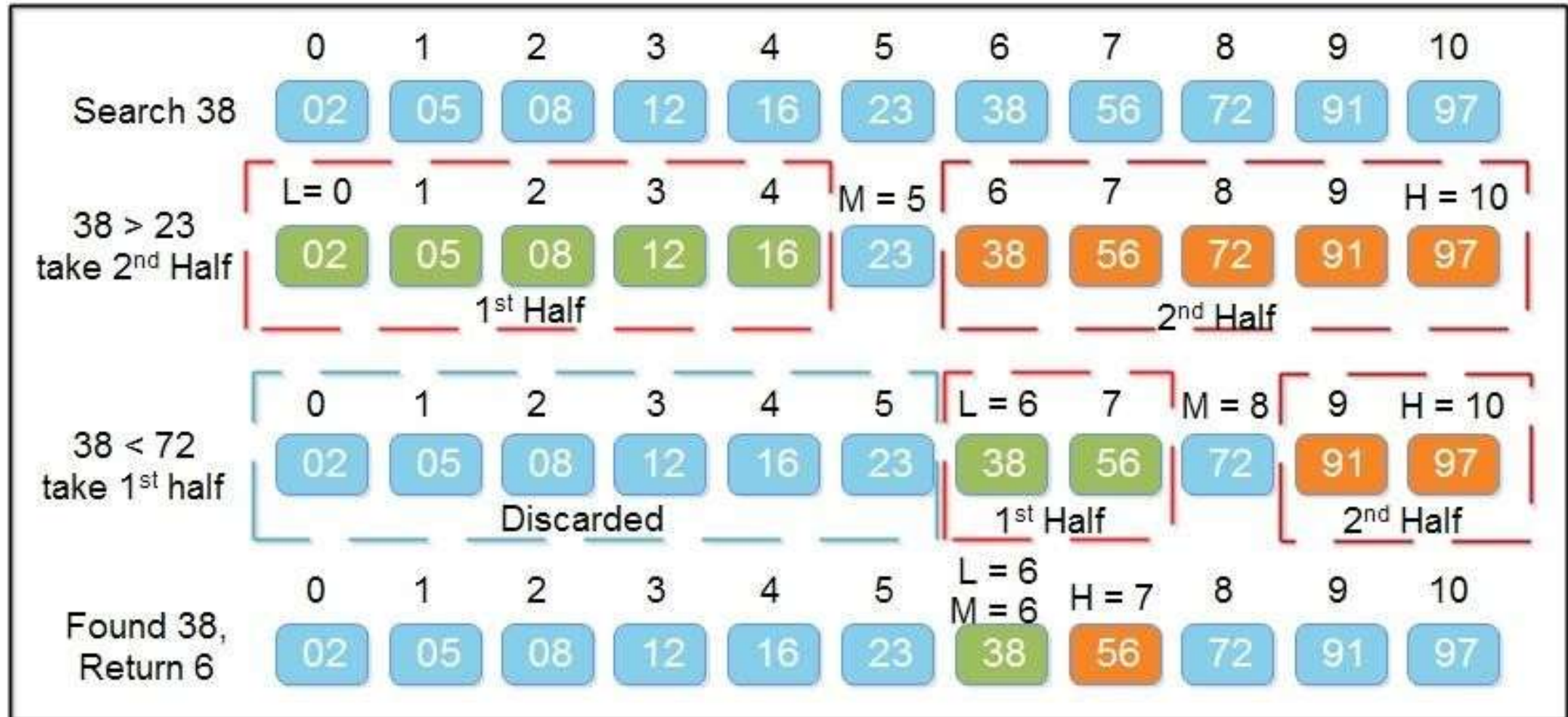
# Binary Search

- **Features of Binary Search Algorithm**
  1. It is great to search through large sorted arrays.
  2. It has a time complexity of  $O(\log n)$  which is a very good time complexity.
  3. It has a simple implementation.

# Binary Search

- Example

## Binary Search in C



# Write a C program to find key element in the list using Binary Search.

## Program

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i, arr[10], search, first, last, middle;
    printf("Enter 10 elements (in ascending order): ");
    for(i=0; i<10; i++)
        scanf("%d", &arr[i]);
    printf("\nEnter element to be search: ");
    scanf("%d", &search);
    first = 0;
    last = 9;
    middle = (first+last)/2;
    while(first <= last)
    {
        if(arr[middle]<search)
            first = middle+1;
        else if(arr[middle]==search)
        {
            printf("\nThe number, %d found at Position %d", search, middle+1);
            break;
        }
    }
```

```
        else
            last = middle-1;
        middle = (first+last)/2;
    }
    if(first>last)
        printf("\nThe number, %d is not found in given Array", search);
    getch();
    return 0;
}
```

## Output

```
Enter 10 elements (in ascending order): 1 5 8 9 10 15 20
21 25 28
Enter element to be search: 15
The number, 15 found at Position 6
```

# Sorting Algorithms

- A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order.
- Efficient sorting is important to optimizing the use of other algorithms that require sorted lists to work correctly and for producing human - readable input. Sorting algorithms are often classified by :
  - ↳ \* Computational complexity (worst, average and best case) in terms of the size of the list (N). For typical sorting algorithms good behaviour is  $O(N\log N)$  and worst case behaviour is  $O(N^2)$  and the average case behaviour is  $O(N)$ .
  - ↳ \* Memory Utilization
  - ↳ \* Stability -Maintaining relative order of records with equal keys.
  - ↳ \* No. of comparisons.
  - ↳ \* Methods applied like Insertion, exchange, selection, merging etc. Sorting is a process of linear ordering of list of objects.
  - ↳ Sorting techniques are categorized into
    - **Internal Sorting:** takes place in the main memory of a computer.
      - Ex. eg : -Bubble sort, Insertion sort, Shell sort, Quick sort, Heap sort, etc.

# Sorting Algorithms

- **External Sorting:** takes place in the secondary memory of a computer, Since the number of objects to be sorted is too large to fit in main memory.
  - eg : -Merge Sort, Multiway Merge, Polyphase merge.

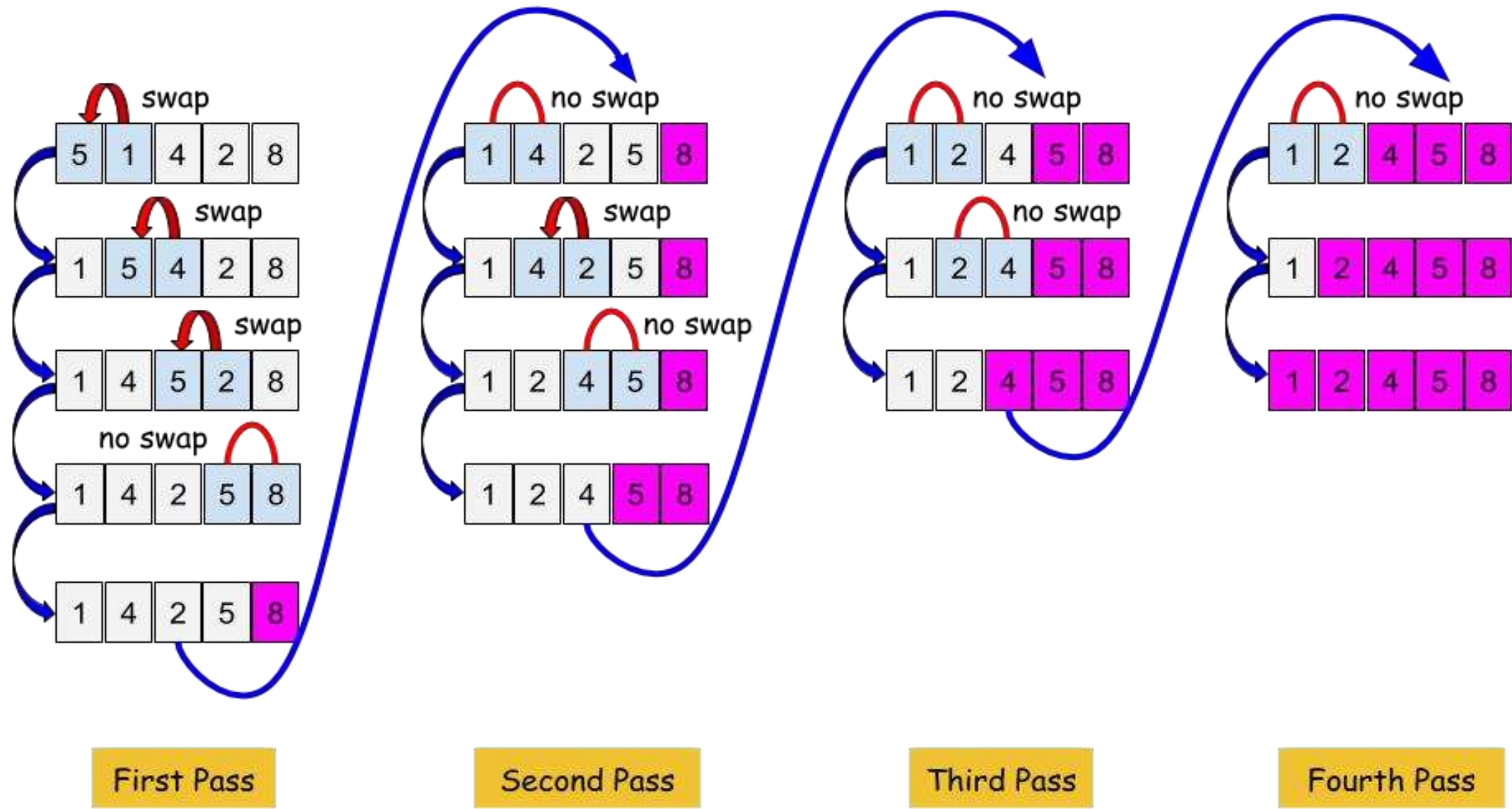
- **Sorting Techniques**

- ↳ **Bubble Sort**
- ↳ **Selection Sort**
- ↳ **Insertion Sort**
- ↳ **Merge Sort**
- ↳ **Quicksort**
- ↳ **Counting Sort**
- ↳ **Radix Sort**
- ↳ **Bucket Sort**
- ↳ **Heap Sort**
- ↳ **Shell Sort**

# Bubble Sort

- Bubble sort is a simple sorting algorithm that repeatedly iterates through the list, compares adjacent elements and swaps them if they are in the wrong order. The algorithm continues until no more swaps are needed.
- The basic steps of the bubble sort algorithm are as follows:
  - ➔ Start at the beginning of the list.
  - ➔ Compare the first two elements. If the first element is greater than the second element, swap them.
  - ➔ Move to the next pair of adjacent elements and repeat step 2.
  - ➔ Continue this process until the end of the list is reached.
  - ➔ If any swaps were made during the previous iteration, repeat steps 2-4 until no swaps are made.

# Bubble Sort



# Write a C program to sort elements in the list using Bubble Sort.

## Program

```
#include <stdio.h>
int main()
{
    int array[100], n, i, j, swap;
    printf("Enter number of elements\n");
    scanf("%d", &n);

    printf("Enter %d integers\n", n);

    for (i = 0; i < n; i++)
        scanf("%d", &array[i]);

    for (i = 0 ; i < n - 1; i++)
    {
        for (j = 0 ; j < n - i - 1; j++)
        {
            if (array[j] > array[j+1])
            {
                swap      = array[j];
                array[j]   = array[j+1];
                array[j+1] = swap;
            }
        }
    }
}
```

```
    }

    printf("Sorted list in ascending order:\n");

    for (i = 0; i < n; i++)
        printf("%d\n", array[i]);

    return 0;
}
```

## Output

```
Enter number of elements
5
Enter 5 integers
8
10
2
0
5
Sorted list in ascending order:
0      2      5      8      10
```

# Bubble Sort

- **Advantages:**

- ➔ Bubble sort is easy to understand and implement.
- ➔ Bubble sort is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the input list.
- ➔ Bubble sort has a space complexity of  $O(1)$ , meaning that it does not require any additional memory beyond the input list.

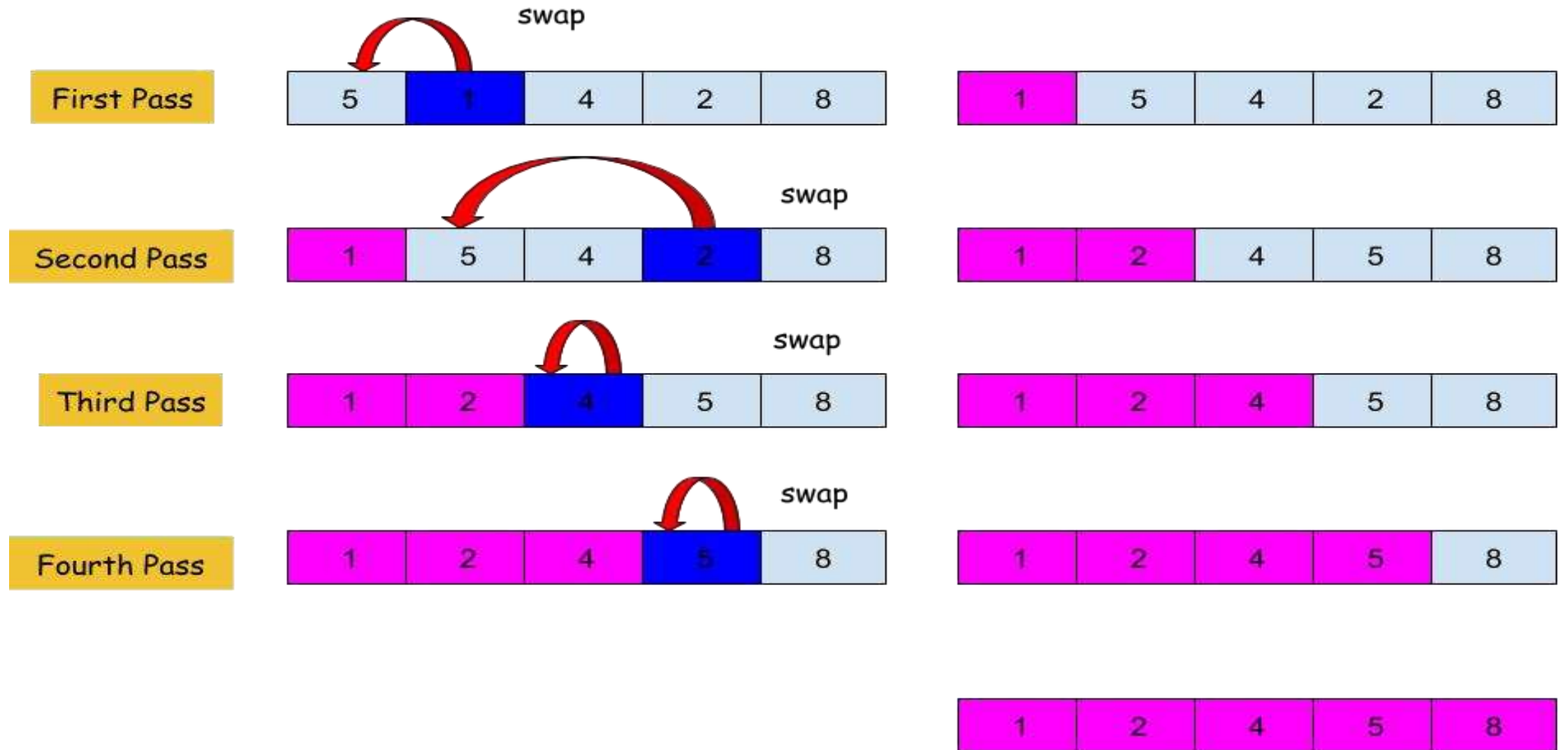
- **Disadvantages:**

- ➔ **Bubble sort has a time complexity of  $O(n^2)$** , where  $n$  is the number of elements in the input list. This means that as the size of the list increases, the time taken to sort the list increases exponentially. For large lists, bubble sort is much slower than other sorting algorithms with better time complexity, such as merge sort or quicksort.
- ➔ **Bubble sort is not adaptive**, meaning that it does not take advantage of the fact that the input list may already be partially sorted. Even if the input list is nearly sorted, bubble sort still requires  $O(n^2)$  time to sort the list.
- ➔ **Bubble sort is not efficient for large lists**, and it is generally only used for educational purposes or for sorting small lists with few elements.

# Selection Sort

- Selection sort is a simple sorting algorithm that works by repeatedly selecting the smallest element from the unsorted portion of the list and swapping it with the first element of the unsorted portion.
- The algorithm proceeds as follows:
  1. Find the smallest element in the unsorted portion of the list.
  2. Swap the smallest element with the first element of the unsorted portion of the list.
  3. Move the boundary between the sorted and unsorted portions of the list one element to the right.
  4. Repeat steps 1-3 until the entire list is sorted.

# Selection Sort



# Write a C program to sort elements in the list using Selection Sort.

## Program

```
#include <stdio.h>
int main()
{
    int array[100], n, c, d, position, t;
    printf("Enter number of elements\n");
    scanf("%d", &n);
    printf("Enter %d integers\n", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
    for (c = 0; c < (n - 1); c++)
    {
        position = c;
        for (d = c + 1; d < n; d++)
        {
            if (array[position] > array[d])
                position = d;
        }
        if (position != c)
        {
            t = array[c];
            array[c] = array[position];
            array[position] = t;
        }
    }
}
```

```
printf("Sorted list in ascending order:\n");
for (c = 0; c < n; c++)
    printf("%d\n", array[c]);
return 0;
}
```

## Output

```
Enter number of elements
5
Enter 5 integers
8
2
1
0
40
Sorted list in ascending order:
0        1        2        8        40
```

# Selection Sort

- **Advantages:**

- ➔ Selection sort is easy to understand and implement.
- ➔ Selection sort is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the input list.
- ➔ Selection sort has a space complexity of  $O(1)$ , meaning that it does not require any additional memory beyond the input list.

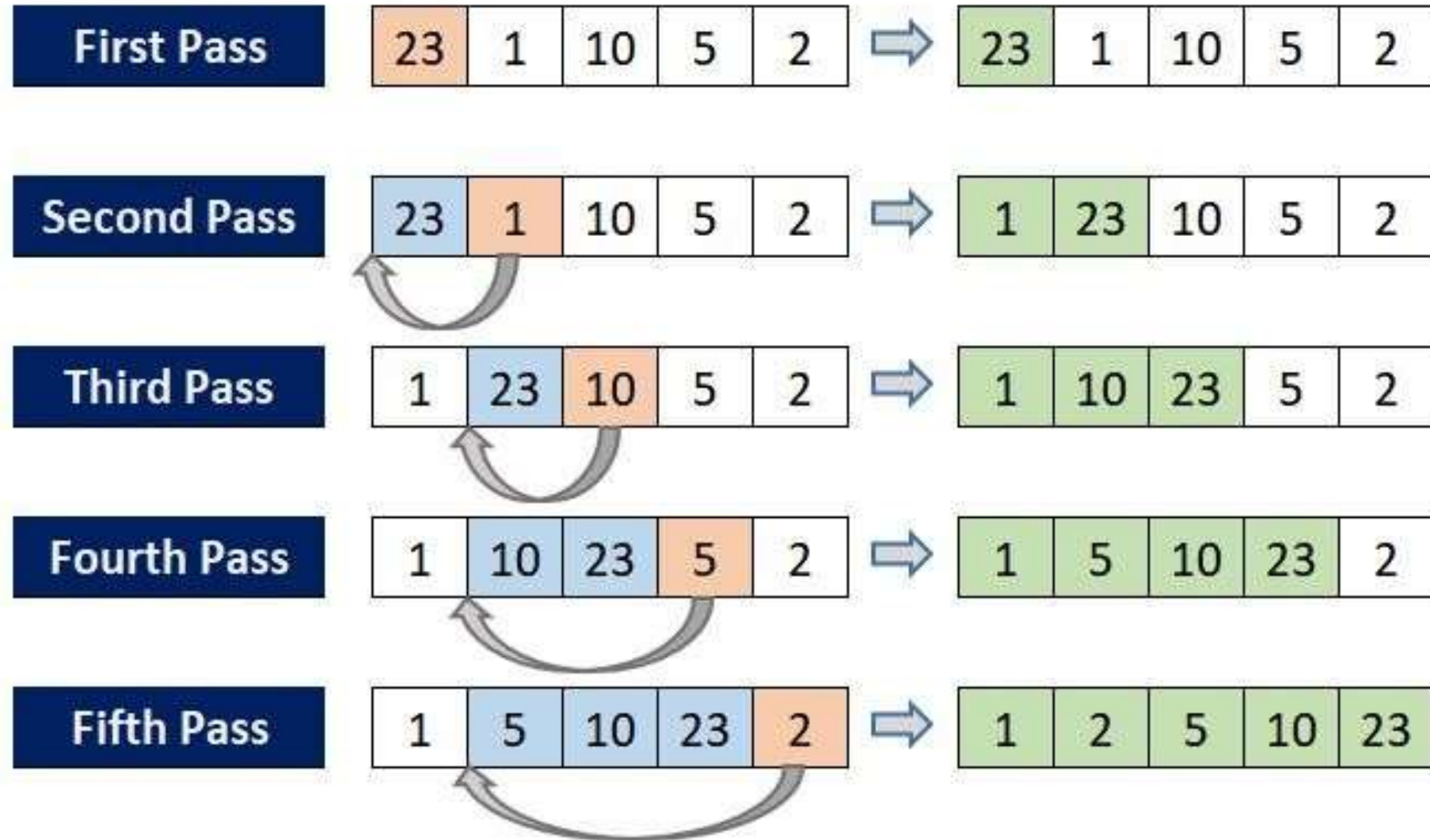
- **Disadvantages:**

- ➔ Selection sort has a time complexity of  $O(n^2)$ , where  $n$  is the number of elements in the input list. This means that as the size of the list increases, the time taken to sort the list increases exponentially. For large lists, selection sort is much slower than other sorting algorithms with better time complexity, such as merge sort or quicksort.
- ➔ Selection sort is not adaptive, meaning that it does not take advantage of the fact that the input list may already be partially sorted. Even if the input list is nearly sorted, selection sort still requires  $O(n^2)$  time to sort the list.
- ➔ Selection sort is not efficient for large lists, and it is generally only used for educational purposes or for sorting small lists with few elements.

# Insertion Sort

- Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.
- The algorithm proceeds as follows:
  - ➔ **Step 1** - If the element is the first element, assume that it is already sorted. Return 1.
  - ➔ **Step2** - Pick the next element, and store it separately in a **key**.
  - ➔ **Step3** - Now, compare the **key** with all elements in the sorted array.
  - ➔ **Step 4** - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
  - ➔ **Step 5** - Insert the value.
  - ➔ **Step 6** - Repeat until the array is sorted.

# Insertion Sort



# Write a C program to sort elements in the list using Insertion Sort.

## Program

```
#include <stdio.h>
int main()
{
    int n, array[1000], c, d, t, flag = 0;
    printf("Enter number of elements\n");
    scanf("%d", &n);
    printf("Enter %d integers\n", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
    for (c = 1; c <= n - 1; c++) {
        t = array[c];
        for (d = c - 1; d >= 0; d--) {
            if (array[d] > t) {
                array[d+1] = array[d];
                flag = 1;
            }
        }
        else
            break;
    }
    if (flag)
        array[d+1] = t;
}
```

```
printf("Sorted list in ascending order:\n");
for (c = 0; c <= n - 1; c++) {
    printf("%d\n", array[c]);
}

return 0;
}
```

## Output

```
Enter number of elements
5
Enter 5 integers
6
8
0
1
30
Sorted list in ascending order:
0      1      6      8      30
```

# Insertion Sort

- **Advantages:**

- ➔ Simple implementation
- ➔ Efficient for small data sets
- ➔ Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

- **Disadvantages:**

- ➔ The disadvantage of the insertion sort is that it does not perform as well as other, better sorting algorithms
- ➔ With  $n^2$  steps required for every  $n$  element to be sorted, the insertion sort does not deal well with a huge list.
- ➔ The insertion sort is particularly useful only when sorting a list of few items.

# Time and Space Complexity of all Sorting Algorithms

## Comparison of Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Quick Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(\log(n))$
Heap Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$

# Introduction to Storage Classes



# Storage Classes

- **Storage** class decides the scope, lifetime and memory allocation of variable.
- Scope of a variable is the boundary within which a variable can be used.

Storage Specifier	Storage	Initial Value	Scope	Life	Example
<b>Auto</b> {auto}	Stack	Garbage	Within block	End of block	<code>int a;</code> <code>auto int a;</code>
<b>Register</b> {register}	CPU register	Garbage	Within block	End of block	<code>register int var;</code>
<b>External</b> {extern}	Data segment	Zero	Global Multiple file	Till end of program	<code>extern int var;</code>
<b>Static</b> {static}	Data segment	Zero	Within block	Till end of program	<code>static extern int var;</code> <code>static int</code>

***Thank  
You***



**D. SRINIVAS**

Computer Science and Engineering Department

✉ [srinivascsedept@gmail.com](mailto:srinivascsedept@gmail.com)

☎ +91-9347556447



# UNIT-5

# Preprocessors and Files



**D. SRINIVAS**

Department of CSE

[www.srinivas-materials.blogspot.com](http://www.srinivas-materials.blogspot.com)

✉ [srinivascsedpt@gmail.com](mailto:srinivascsedpt@gmail.com)

☎ +91 9347556447



## Outline

- **Preprocessor:**

- ↳ Introduction to Preprocessors
- ↳ Types of Preprocessors
- ↳ Commonly used Preprocessor commands like include, define, undef, if, ifdef, ifndef.

- **Files:**

- ↳ Text and Binary files,
- ↳ Creating and Reading and writing text and binary files,
- ↳ Appending data to existing files,
- ↳ Writing and reading structures using binary files,
- ↳ Random access using fseek, ftell and rewind functions

# Introduction to Preprocessors



# Preprocessor

- The C compiler is made of two functional parts: a preprocessor and a translator.
- The preprocessor is a program which processes the source code before it passes through the compiler.
- The translator is a program which converts the program into machine language and gives the object module.
- **There are 4 main types of preprocessor directives:**
  - Macros
  - File Inclusion
  - Conditional Compilation
  - Other directives

# Types of Pre-processors

- 1. Macros:

- Macros are a piece of code in a program which is given some name.
- Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code.
- The '#define' directive is used to define a macro.

- a. Pass by Symbolic Constants as macro:

- Macro definition without arguments is referred as a constant.
- The body of the macro definition can be any constant value including integer, float, double, character, or string.
- However, character constants must be enclosed in single quotes and string constants in double quotes.
- Example:

```
#define PI 3.14159
```

- Here "PI" replaces with "3.14159".

- b. Pass by Function macro:

- C handles function macros by simply rescanning a line after macro expansion.
- Therefore, if an expansion results in a new statement with a macro, the second macro will be properly expanded.
- For Example:

```
#define sqre(a) (a*a)
```

# Types of Preprocessors

## → Program

//Example for macro to calculate square of a given number

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

```
#define square(x) (x*x) /* macro definition */
```

```
void main()
```

```
{
```

```
    int a=10;
```

```
    printf("The square of %d=%d", a, square(a));
```

```
}
```

OUTPUT:

The square of 5 = 25

# Types of Pre-processors

## → C. Predefined Macros

- ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

## → Program:

```
#include <stdio.h>
void main()
{
    printf("File   :%s\n", __FILE__ );
    printf("Date   :%s\n", __DATE__ );
    printf("Time    :%s\n", __TIME__ );
    printf("Line    :%d\n", __LINE__ );
    printf("ANSI :%d\n", __STDC__ );
}
```

## → Output:

```
File :test.c
Date :April 1 2022
Time :03:36:24
Line :8
ANSI :1
```

Sr.No.	Macro & Description
1. <b>__DATE__</b>	The current date as a character literal in "MMM DD YYYY" format.
2. <b>__TIME__</b>	The current time as a character literal in "HH:MM:SS" format.
3. <b>__FILE__</b>	This contains the current filename as a string literal.
4. <b>__LINE__</b>	This contains the current line number as a decimal constant.
5. <b>__STDC__</b>	Defined as 1 when the compiler complies with the ANSI standard.

# Types of Pre-processors –Cont...

- **2.File Inclusion:**

- ➔ This type of preprocessor directive tells the compiler to include a file in the source code program.
- ➔ There are two types of files which can be included by the user in the program:
  - ➔ **A. Header File or Standard files:**
    - These files contains definition of pre-defined functions like printf(), scanf() etc.
    - These files must be included for working with these functions.
    - It is used to direct the preprocessor to include header files from the system library.
  - ➔ **Syntax:**
    - #include< *filename* >
  - ➔ **B. User defined files:**
    - When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed.
    - These types of files are user defined files.
    - It is used to direct the preprocessor look for the files in the current working directory and standard library.
    - These files can be included as:
  - ➔ **Syntax:**
    - #include" *filename* "

# Types of Pre-processors –Cont...

- **3.Conditional Compilation:**

- ➔ Conditional Compilation directives are type of directives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions.
- ➔ It allows us to control the compilation process by including or excluding statements.
- ➔ Cast expressions, size of, enumeration constants cannot be evaluated in preprocessor directives.
- ➔ Its structure is similar to if statement.
- ➔ This can be done with the help of two preprocessing commands '**ifdef**' and '**endif**'.
- ➔ Syntax for conditional compilation:
  - **#if** expression1  
code to be included for true
  - **#elif** expression2  
code to be included for true
  - **#else**  
code to be included false
  - **#endif**
  - **#if !defined( NULL )**

# Types of Pre-processors –Cont...

## • 4. Other Directives:

- Apart from the above directives there are two more directives which are not commonly used.
- These are:
  - **#undef Directive:**
    - The #undef directive is used to undefine an existing macro.
    - This directive works as:
      - first job of a preprocessor is file inclusion that is copying of one or more files into programs.
      - The files are usually header files and external files containing functions and data declarations.
  - **Syntax:**
    - #undef LIMIT
  - **#pragma Directive:**
    - This directive is a special purpose directive and is used to turn on or off some features.
    - **#pragma startup** and **#pragma exit**:
      - These directives helps us to specify the functions that are needed to run before program startup( before the control passes to main()) and just before program exit (just before the control returns from main()).

# Types of Pre-processors –Cont...

<i>Preprocessor</i>	<i>Syntax/Description</i>
Macro	<b>Syntax:</b> #define This macro defines constant value and can be any of the basic data types.
Header file inclusion	<b>Syntax:</b> #include <file_name> The source code of the file “file_name” is included in the main program at the specified place.
Conditional compilation	<b>Syntax:</b> #ifdef, #endif, #if, #else, #ifndef Set of commands are included or excluded in source program before compilation with respect to the condition.
Other directives	<b>Syntax:</b> #undef, #pragma #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program.

# Introduction to Files



# Files

## FILE:

- A file is an external collection of related data treated as a unit.
- The primary purpose of a file is to keep a record of data.
- Record is a group of related fields. Field is a group of characters they convey meaning.
- Files are stored in auxiliary or secondary storage devices. The two common forms of secondary storage are disk (hard disk, CD and DVD) and tape.
- Each file ends with an end of file (EOF) at a specified byte number, recorded in file structure.
- C has predefined structure to hold this information.
- The stdio.h header file defines this file structure; its name is **FILE**.

## • File Name

- File name is a string of characters that make up a valid filename.
- Every operating system uses a set of rules for naming its files.
- When we want to read or write files, we must use the operating system rules when we name a file.
- The file name may contain two parts, a primary name and an optional period with extension.
- input.txt

**Example:** program.c

# File Management

- In real life, we want to store data permanently so that later we can retrieve it and reuse it.
- A file is a collection of characters stored on a secondary storage device like hard disk, or pen drive.
- There are two kinds of files that programmer deals with:
  - ➔ **Text Files** are human readable and it is a stream of plain English characters
  - ➔ **Binary Files** are computer readable, and it is a stream of processed characters and ASCII symbols

Hello, this is a text file. Whatever written here can be read easily without the help of a computer.

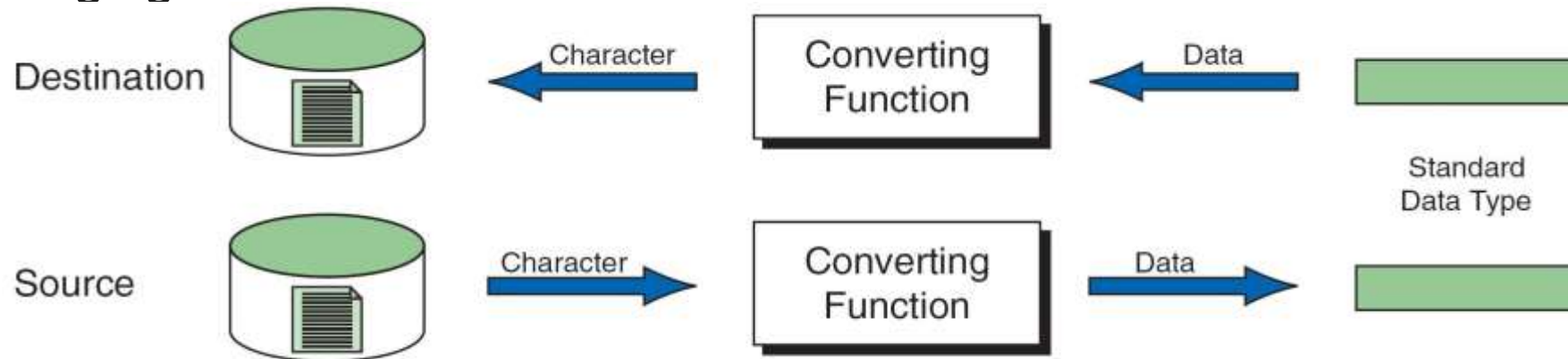
Binary File

# Text Files And Binary Files:

- **Text File:**

- ➔ It is a file in which data are stored using only characters; a text file is written using text stream.
- ➔ Non-character data types are converted to a sequence of characters before they are stored in the file.
- ➔ In the text format, data are organized into lines, terminated by newline character.
- ➔ The text files are in human readable form and they can be created and read using any text editor.
- ➔ Text files are read and written using input / output functions that convert characters to data types: scanf and printf, getchar and putchar, fgets and fputs.

- The following figure shows the data transfer in text file:



**Reading and Writing Text Files**

# File Opening Modes

→ We can perform different operations on a file based on the file opening modes

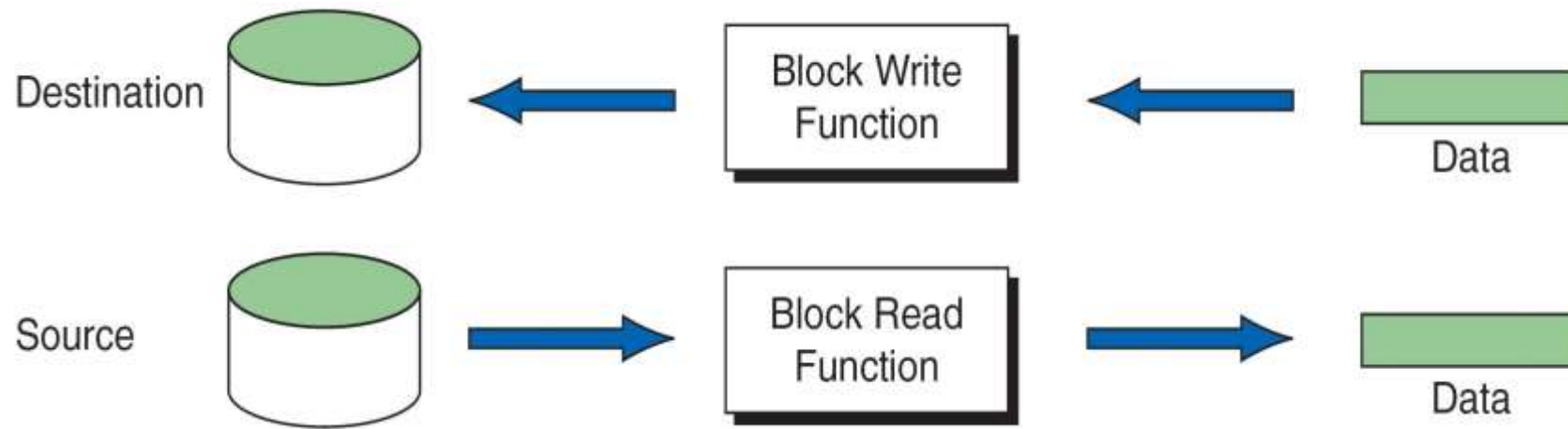
Text Mode	Binary Mode	Description
<b>r</b>	<b>rb</b>	Open the file for reading only. If it exists, then the file is opened with the current contents; otherwise an error occurs.
<b>w</b>	<b>wb</b>	Open the file for writing only. A file with specified name is created if the file does not exists. The contents are deleted, if the file already exists.
<b>a</b>	<b>ab</b>	Open the file for appending (or adding data at the end of file) data to it. The file is opened with the current contents safe. A file with the specified name is created if the file does not exists.
<b>r+</b>	<b>r+b</b>	The existing file is opened to the beginning for both reading and writing.
<b>w+</b>	<b>w+b</b>	Same as w except both for reading and writing.
<b>a+</b>	<b>a+b</b>	Same as a except both for reading and writing.

**Note:** The main difference is w+ truncate the file to zero length if it exists or create a new file if it doesn't. While r+ neither deletes the content nor create a new file if it doesn't exist.

# Text Files And Binary Files:

- **Binary File:**

- ➔ A binary file is a collection of data stored in the internal format of the computer.
- ➔ The binary files are not in human readable form.
- ➔ There are no lines or newline characters.
- ➔ Binary files are read and written using binary streams known as block input / output functions.
- ➔ The following figure shows the data transfer in binary file:



**Block Input and Output  
Binary Files**

# Text Files And Binary Files:

- **Differences between Text File and Binary File**

<b>Text File</b>	<b>Binary File</b>
Data is stored as lines of characters with each line terminated by newline.	Data is stored on the disk in the same way as it is represented in the computer memory.
Human readable format.	Not in human readable format.
There is a special character called end-of-file(EOF) marker at the end of the file.	There is an end-of-file marker.
Data can be read using any of the text editors.	Data can be read only by specific programs written for them.

# File Handling Functions or Operations

- In general, there are five steps to processing a file.
  - ➔ 1. Creating a file
  - ➔ 2. Opening a file
  - ➔ 3. Reading a file
  - ➔ 4. Writing a file
  - ➔ 5. Closing a file
- **1.Creating a file (fopen):**
  - ➔ The function that prepares a file for processing is fopen.
  - ➔ It does two things: First, it makes the connection between the physical file and the file stream in the program.
  - ➔ Second, it creates a program file structure to store the information needed to process the file.
  - ➔ To open a file, we need to specify the physical filename and its mode.
  - ➔ **Syntax:**  
`fopen ("filename", "mode");`
  - ➔ The file mode is a string that tells C compiler how we intend to use the file: reading, writing or append.
  - ➔ For example: `fptr1 = fopen ("mydata", "r");`
  - ➔ Once the files are open, they stay open until you close them or end the program.

# File Handling Functions or Operations

- **2.Opening a file (fopen):**

- This function used to open existing file with respective mode.
- The function that prepares a file for processing is fopen.
- It does two things: First, it makes the connection between the physical file and the file stream in the program.
- Second, it creates a program file structure to store the information needed to process the file.
- To open a file, we need to specify the physical filename and its mode.
- Syntax:  
`fopen ("filename", "mode");`
- The file mode is a string that tells C compiler how we intend to use the file: reading, writing or append.
- For example: `fptr1 = fopen ("mydata", "r");`
- Once the files are open, they stay open until you close them or end the program.

# File Handling Functions or Operations

- **3. Reading a file :**

- **Read a character:** `getc ()` and `fgetc ()`

- ➔ The `getc` functions read the next character from the stream, which can be a user-defined stream or `stdin`, and converts it into an integer.
- ➔ This function has one argument which is the file pointer declared as `FILE` or `stdin` (in case of standard input stream).
- ➔ If the read detects an end of file, the function returns `EOF`, `EOF` is also returned if any error occurs.
- ➔ The functionality of `getc` / `fgetc` is same.
- ➔ **Syntax:**

```
int getc (FILE *spIn);      or      int  
                             getc(stdin);  
int fgetc (FILE *spIn);    or      int  
                             fgetc(stdin);
```

- **Read a string:** `gets ()` and `fgets ()`

- ➔ The `gets` functions read the string from the file, which can be a user-defined stream or `stdin`.
- ➔ This function has one argument which is the file pointer declared as `FILE` or `stdin` (in case of standard input stream).
- ➔ If the read detects an end of file, the function returns `EOF`, `EOF` is also returned if any error occurs.
- ➔ The functionality of `gets` / `fgets` is same.
- ➔ **Syntax:**

```
int gets (FILE *spIn);      or      int gets(stdin);  
int fgets(FILE *spIn);    or      int fgets(stdin);
```

# File Handling Functions or Operations

- **Read an integer: `getw ()` and `fgetw ()`**

- The `getc` functions read the next integer from the stream, which can be a user-defined stream or `stdin`.
- This function has one argument which is the file pointer declared as `FILE` or `stdin` (in case of standard input stream).
- If the read detects an end of file, the function returns `EOF`, `EOF` is also returned if any error occurs.
- The functionality of `getw` / `fgetw` is same.
- **Syntax:**  

<code>int getw (FILE *splt);</code>	or	<code>int getw(stdin);</code>
<code>int fgetw (FILE *splt);</code>	or	<code>int fgetw(stdin);</code>

# File Handling Functions or Operations

- 3. Reading a file :
- Formatted input -**fscanf ()**:
  - It is used to read data from a user-specified stream.
  - The general format of fscanf() is:  
**fscanf (stream-pointer, "format string", list);**
  - The first argument is the stream pointer, it is the pointer to the streams that has been declared and associated with a text file. Remaining is same as scanf function arguments.
  - The following example illustrates the use of an input stream.

```
int a, b; FILE
*fptr1;
fptr1 = fopen ("mydata", "r");
fscanf (fptr1, "%d %d", &a, &b);
```
  - The fscanf function would read values from the file "pointed" to by fptr1 and assign those values to a and b.
  - The only difference between scanf and fscanf is that scanf reads data from the stdin (input stream) and fscanf reads input from a user specified stream(stdin or file).
  - The following example illustrates how to read data from keyboard using fscanf,  
**fscanf (stdin, "%d", &a);**

# File Handling Functions or Operations

- 4.Writing a file :
- Writing a character: `putc ()` and `fputc ()`
  - The `putc` function writes a character to the stream which can be a user-defined stream, `stdout`, or `stderr`.
  - The functionality of `putc/ fputc` is same.
  - The functions, `putc` or `fputc` takes two arguments.
  - The first parameter is the character to be written and the second parameter is the file.
  - The second parameter is the file pointer declared as `FILE` or `stdout` or `stderr`.
  - If the character is successfully written, the function returns it. If any error occurs, it returns `EOF`.
  - **Syntax:**  
`int putc (char, *fp);`  
`int fputc (char, *fp);`

# File Handling Functions or Operations

- **Read an integer: `getw ()` and `fgetw ()`**

- The `getc` functions read the next integer from the stream, which can be a user-defined stream or `stdin`.
- This function has one argument which is the file pointer declared as `FILE` or `stdin` (in case of standard input stream).
- If the read detects an end of file, the function returns `EOF`, `EOF` is also returned if any error occurs.
- The functionality of `getw` / `fgetw` is same.
- **Syntax:**

<code>int getw (FILE *splt);</code>	or	<code>int getw(stdin);</code>
<code>int fgetw (FILE *splt);</code>	or	<code>int fgetw(stdin);</code>

# File Handling Functions or Operations

- **Formatted Output -Writing to Files: `fprintf ()`**

- It can handle a group of mixed data simultaneously.
- The first argument of these functions is a file pointer which specifies the file to be used.
- The general form of `fprintf` is:  
`fprintf (stream-pointer, "format string", list);`
- Where stream-pointer is a file pointer associated with a file that has been opened for writing.
- The format string contains output specifications for the items in the list.
- The list may include variables, constants and strings.
- The following example illustrates the use of an Output stream.  

```
int a = 5, b = 20;  
FILE *fptr2;  
fptr2 = fopen ("results", "w");  
fprintf (fptr2, "%d %d", a, b) ;
```
- The `fprintf` functions would write the values stored in `a` and `b` to the file "pointed" to by `fptr2`.
- `fprintf` function works like `printf` except that it specifies the file in which the data will be displayed.
- The file can be standard output (`stdout`) or standard error (`stderr`) also.
- Example,  
`fprintf (stdout, "%d", 45);`

# File Handling Functions or Operations

- **5.Closing a file (fclose):**

- ➔ When we no longer need a file, we should be close it to free system resources, such as buffer space.
- ➔ Closing a file ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken.
- ➔ Another instance where we have to close a file is to reopen the same file in a different mode.
- ➔ A file is closed using the close function, fclose.
- ➔ **Syntax:**  
`fclose (filepointer);`
- ➔ fclose () returns 0 on success (or) -1 on error.
- ➔ Once a file is closed, its file pointer can be reused for another file.

# Programs

- Write a C program to fprintf() and fscanf() a text file.

## Program

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

int main()
{
    int num;
    FILE *fptr;
    fptr = fopen("C:\\program.txt","w");

    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    printf("Enter num: ");
    scanf("%d",&num);

    fprintf(fptr,"%d",num);
    fclose(fptr);

    return 0;
}
```

## Program

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

int main()
{
    int num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.txt","r")) == NULL){
        printf("Error! opening file");

        exit(1);
    }

    fscanf(fptr,"%d", &num);

    printf("Value of n=%d", num);
    fclose(fptr);

    return 0;
}
```

# File Handling Functions or Operations

- **Reading And Writing binary Functions:**

- C language uses the block input and output functions to read and write data to binary files.
- As we know that data are stored in memory in the form of 0's and 1's.
- When we read and write the binary files, the data are transferred just as they are found in memory and hence there are no format conversions.

- **File Read: `fread ()`**

- It reads a specified number of bytes from a binary file and places them into memory at the specified location.
- The function declaration is as follows:  
`int fread (void *pInArea, int elementsize, int count, FILE *sp);`
- The first parameter, pInArea, is a pointer to the input area in memory. The data read from the file should be stored in memory.
- For this purpose, it is required to allocate the sufficient memory and address of the first byte is stored in pInArea.
- The next two elements, elementSize and count, are multiplied to determine how much data are to be transferred.
- The size is normally specified using the sizeof operator and the count is normally one when reading structures.
- The last argument is the pointer to the file we want to read from.
- This function returns the number of items read. If no items have been read or when error has occurred or EOF encountered, the function returns 0.

# File Handling Functions or Operations

- **File Write: `fwrite ()`**

- ➔ It writes specified number of items to a binary file.
- ➔ The function declaration is as follows,  
`int fwrite (void *pOutArea, int elementSize, int count, FILE *sp);`
- ➔ The parameters for file write correspond exactly to the parameters for the file read function.

# Programs

- Write a C program to fread() and fwrite() a binary file.

## Program

```
#include <stdio.h>
#include <stdlib.h>
struct threeNum
{
    int n1, n2, n3;
};
void main()
{
    int n;
    struct threeNum num;
    FILE *fptr;
    if ((fptr = fopen("C:\\\\program.bin", "wb")) == NULL){
        printf("Error! opening file");
        exit(1);
    }
    for(n = 1; n < 5; ++n)
    {
        num.n1 = n;
        num.n2 = 5*n;
        num.n3 = 5*n + 1;
        fwrite(&num, sizeof(struct threeNum), 1, fptr);
    }
    fclose(fptr);
}
```

## Program

```
#include <stdio.h>
#include <stdlib.h>
struct threeNum
{
    int n1, n2, n3;
};
void main()
{
    int n;
    struct threeNum num;
    FILE *fptr;
    if ((fptr = fopen("C:\\\\program.bin", "rb")) == NULL){
        printf("Error! opening file");
        exit(1);
    }

    for(n = 1; n < 5; ++n)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\\tn2: %d\\tn3: %d\\n", num.n1, num.n2,
num.n3);
    }
    fclose(fptr);
}
```

# Programs

- Write a C program to copy a given file.

## Program

```
#include <stdio.h>
void main()
{
    FILE *fp1, *fp2; //p and q is a FILE type pointer
    char ch; //ch is used to store temporary data
    fp1 = fopen("file1.c", "r"); //open file "file1.c" in read mode
    fp2 = fopen("file2.c", "w"); //open file "file2.c" in write mode
    do { //repeat step 9 and 10 until EOF is reached
        ch = getc(fp1); //get character pointed by p into ch
        putc(ch, fp2); //print ch value into file, pointed by pointer q
    } while (ch != EOF); //condition to check EOF is reached or not
    fclose(fp1); //free up the file pointer p
    fclose(fp2); //free up the file pointer q
    printf("File copied successfully...");
}
```

# Programs- Cont...

- Write a C program to merge the two files

Program

```
#include <stdio.h>
void main()
{
    FILE *p1,*p2,*p3;
    char ch;
    p1 = fopen("file1.txt","r");
    p2 = fopen("file2.txt","r");
    p3 = fopen("file3.txt","w");
    if (fp1 == NULL || fp2 == NULL)
    {
        puts("Could not open files");
        exit(0);
    }
    while ((c = fgetc(fp1)) != EOF)
    {
        fputc(c, fp3);
    }
    while ((c = fgetc(fp2)) != EOF)
    {
        fputc(c, fp3);
    }
}
```

Program (contd.)

```
printf("Merged file1.txt and
file2.txt into file3.txt");
fclose(fp1);
fclose(fp2);
fclose(fp3);
return 0;
}
```

Output

```
Merged file1.txt and file2.txt into
file3.txt
```

# Programs- Cont...

- Write a C program to count lines, words, tabs, and characters

Program

```
#include <stdio.h>
void main()
{
    FILE *p;
    char ch;
    int ln=0,t=0,w=0,c=0;
    p = fopen("text1.txt","r");
    ch = getc(p);
    while (ch != EOF) {
        if (ch == '\n')
            ln++;
        else if(ch == '\t')
            t++;
        else if(ch == ' ')
            w++;
        else
            c++;
        ch = getc(p);
    }
    printf("Lines = %d, tabs = %d, words = %d, characters = %d\n", ln, t, w, c);
}
```

Program  
(contd.)

```
C++;

    ch = getc(p);
}
fclose(p);
printf("Lines = %d, tabs = %d, words = %d, characters = %d\n", ln, t, w, c);
}
```

Output

```
Lines = 22, tabs = 0, words = 152,
characters = 283
```

# Random Access File Functions

- **Rewind File (rewind):**

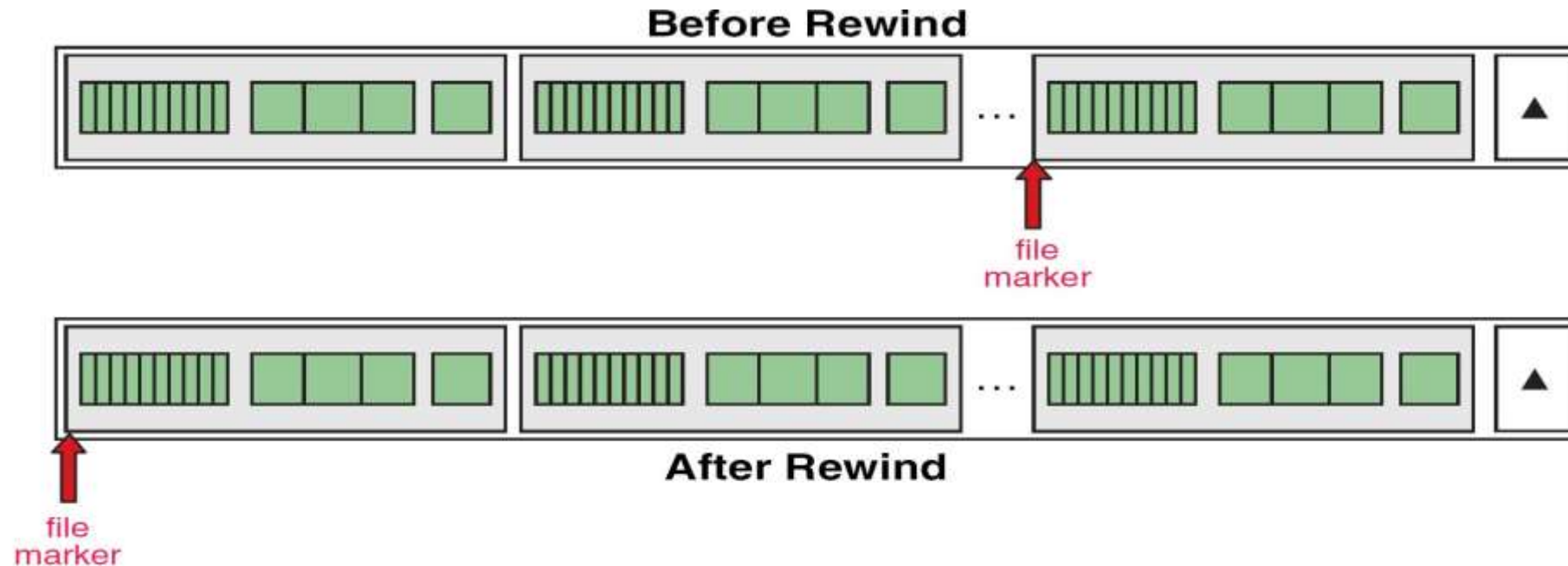
- ➔ It simply sets the file position indicator to the beginning of the file.

- ➔ Syntax:

```
void rewind(FILE *stream);
```

- ➔ It helps us in reading a file more than once, without having to close and open the file.

- ➔ A common use of the rewind function is to change a work file from a write state to a read state.



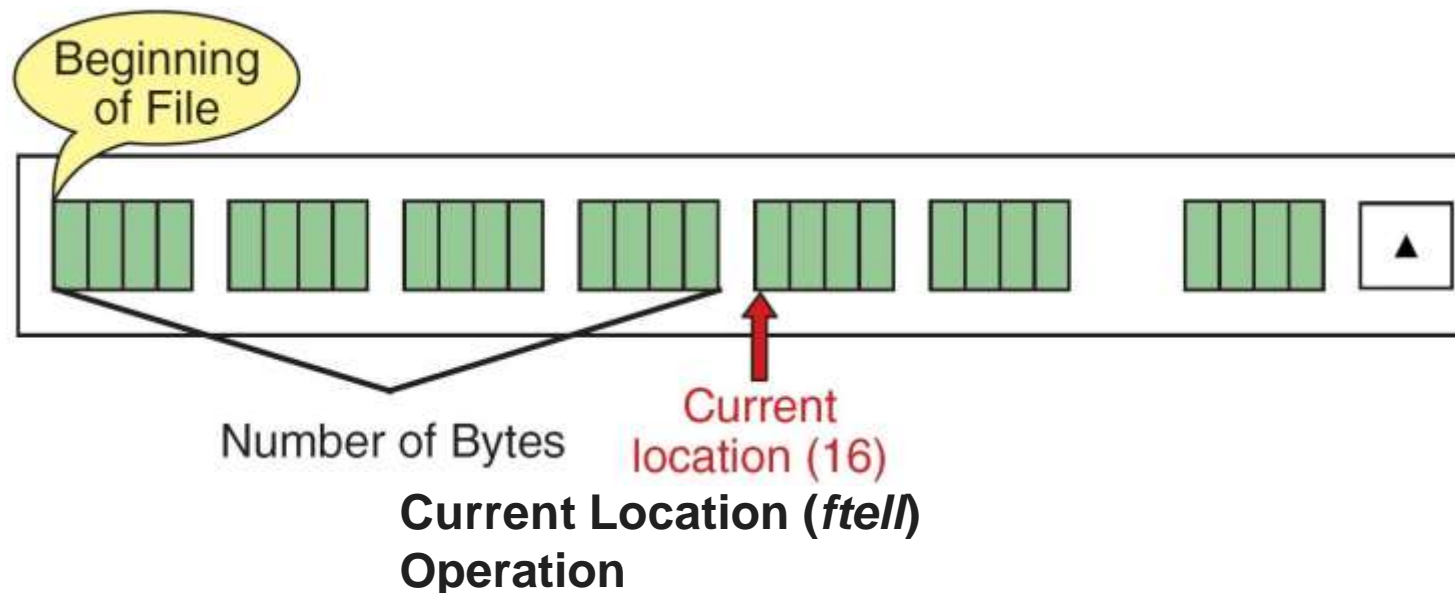
# Random Access File Functions

- Current Location (**ftell**):

- It reports the current position of the file marker in the file, relative to the beginning of the file.
- It measures the position in the file by the number of bytes, relative to zero, from the beginning of the file.
- **Syntax:**

```
int pos=long int ftell(FILE *stream);
```

- It also returns the number of bytes from the beginning of the file.
- If ftell encounters an error, it returns -1.



# Random Access File Functions

- Re Position: (**fseek**):

- ➔ It is used to move the file position to a desired location within the file.



**Syntax:**

```
int fseek(FILE *stream, long offset, int wherefrom);
```

- ➔ The offset specifies the number of positions to be moved from the location specified by position.

- ➔ The position can take one of the following three values:

**Value**

**Meaning**

0

Beginning of file.

1

Current position.

2

End of file.

- ➔ The offset may be positive( means forward), or negative (means backward).

- ➔ When the operation is successful, fseek returns a zero.

- ➔ If we attempt to move the file pointer beyond the file boundaries, an error occurs and fseek returns -1.

# Random Access File Functions

## Operations of the fseek function

Statement	Meaning
<code>fseek(fp,0L,0);</code>	Go to the beginning.
<code>fseek(fp,0L,1);</code>	Stay at the current position.
<code>fseek(fp,0L,2);</code>	Go to the end of the file, past the last character of the file.
<code>fseek(fp,m,0)</code>	Move to (m+1)th byte in the file.
<code>fseek(fp,m,1);</code>	Go forward by m bytes.
<code>fseek(fp,-m,1);</code>	Go backward by m bytes from the current position.
<code>fseek(fp,-m,2);</code>	Go backward by m bytes from the end. (positions the file to the character from the end.)

***Thank  
You***



**D. SRINIVAS**

Computer Science and Engineering Department

✉ [srinivascsedept@gmail.com](mailto:srinivascsedept@gmail.com)

☎ +91-9347556447