Infographics

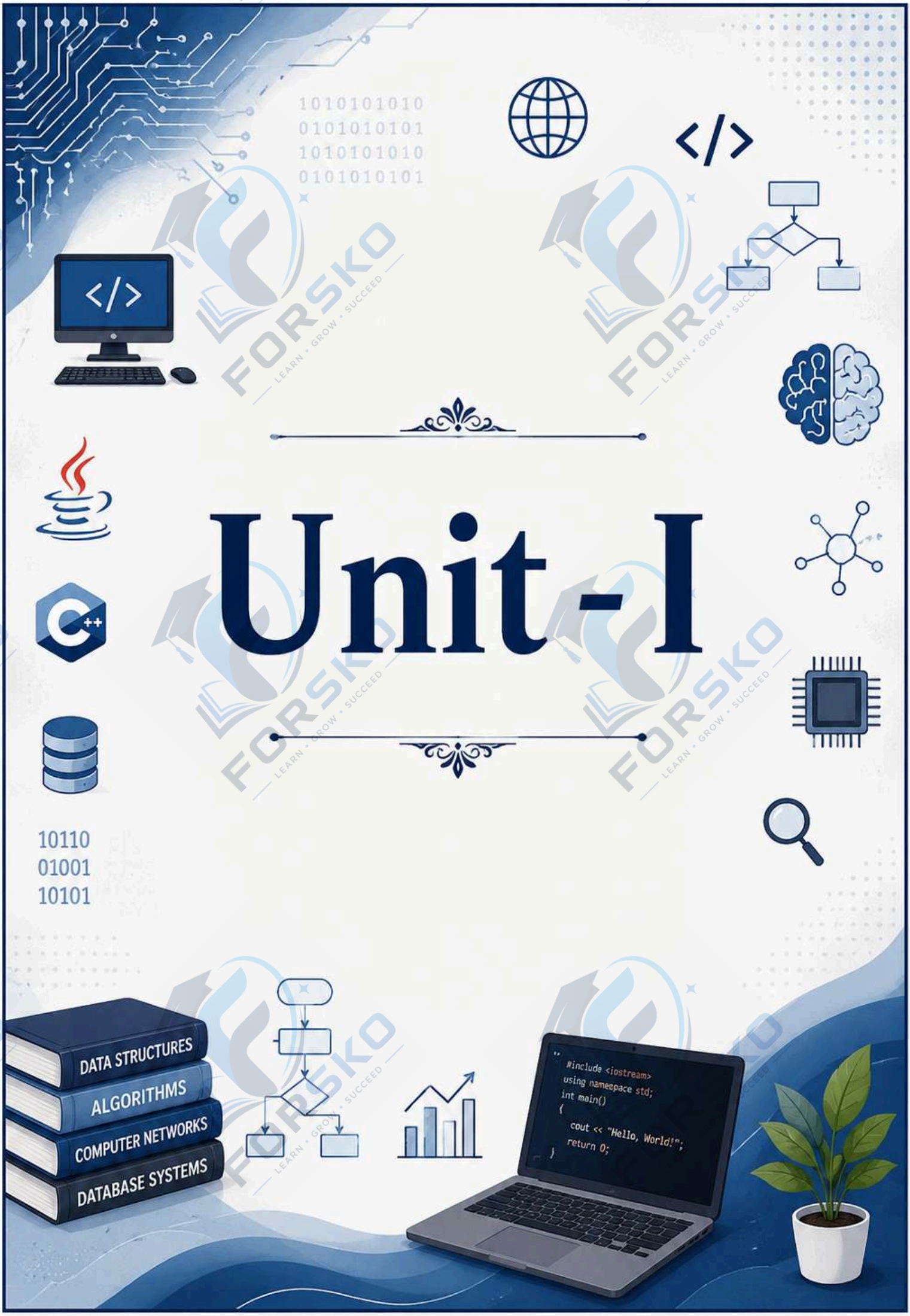
Visual Learning
Better understanding through visual content.

Simplified Concepts
Complex topics explained in simple and visual form.

Quick Revision
Easy to revise important points and diagrams.

Exam Ready
Helps in faster revision and better exam preparation.

Unit-I


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using nameepace std;
int main()
{
cout << "Hello, World!";
return 0;
}

ALGORITHM

An ALGORITHM is a finite, ordered set of well-defined instructions to solve a problem or perform a task.

CHARACTERISTICS

Input
It should have zero or more inputs.

Output
It should have at least one output.

Definiteness
Each step must be clear, precise and unambiguous.

Finiteness
It must terminate after a finite number of steps.

Effectiveness
Each step must be basic and feasible to perform.

Generality
It should be applicable to a class of problems.

EXAMPLE : ALGORITHM TO FIND SUM OF TWO NUMBERS

1. Start
2. Read two numbers A and B
3. Calculate SUM ← A + B
4. Display SUM
5. Stop

TIPS FOR WRITING GOOD ALGORITHM

- Use simple and clear language.
- Write steps in logical order.
- Make sure the algorithm terminates.
- Each step should be independent and effective.
- Use meaningful names for variables and constants.

IMPORTANCE

- Provides a step-by-step solution.
- Acts as a bridge between human thinking and computer program.
- Makes complex problems easier to understand.
- Helps in debugging and program maintenance.

APPLICATIONS

- Solving mathematical problems
- Data processing
- Control systems
- Artificial Intelligence, etc.

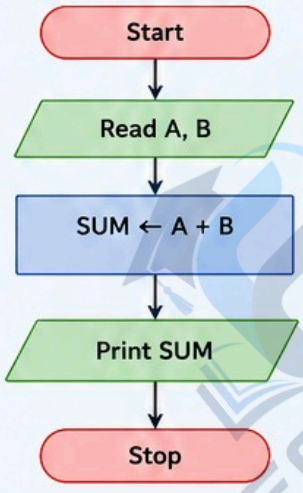
FLOWCHARTING

A FLOWCHART is a graphical representation of an algorithm. It uses standard symbols connected by arrows to show the flow of control.

COMMON FLOWCHART SYMBOLS

SYMBOL	NAME	PURPOSE
	Terminator	Indicates Start or Stop.
	Input / Output	Used for input or output operations.
	Process	Represents a processing step or calculation.
	Decision	Used for decision making (Yes / No).
→	Flow Line	Shows the direction of flow.
	Connector	Connects flowchart from one part to another.

EXAMPLE FLOWCHART : FIND SUM OF TWO NUMBERS



HOW TO READ

1. Start
2. Read two numbers A and B
3. Add A and B and store in SUM
4. Display SUM
5. Stop

ADVANTAGES

- Easy to understand
- Provides a clear picture of logic
- Helps in coding
- Useful for communication

ALGORITHM vs FLOWCHART

Algorithm is written in words or pseudocode. → Flowchart is a graphical representation.

Algorithm is easier to write for simple steps. → Flowchart is easier to understand visually.

TYPES OF PROGRAMMING LANGUAGES

Programming languages can be classified in different ways.
Below are the most common types with examples and uses.

1. BASED ON LEVEL OF ABSTRACTION

(A) LOW-LEVEL LANGUAGES

Close to machine language.
Dependent on computer hardware.

Includes :
Machine Language
Assembly Language

Features
- Very fast execution
- Difficult to write and understand
- Machine dependent

Examples
Machine Code, Assembly (x86, ARM)

(B) HIGH-LEVEL LANGUAGES

Closer to human language.
Independent of computer hardware.

```
if (x > 0) {
   sum = a + b;
   print(sum);
}
```

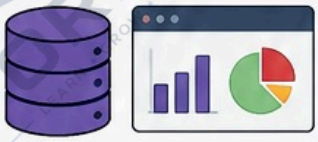
Includes :
Procedural, Object-Oriented, Functional, Scripting etc.

Features
- Easy to write and understand
- Portable (machine independent)
- Requires translator (compiler/interpreter)

Examples
C, C++, Java, Python, JavaScript, Pascal

(C) FOURTH-GENERATION LANGUAGES (4GL)

High-level, non-procedural.
Focus on what to do, not how to do.



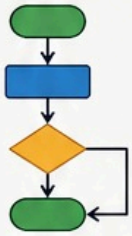
Includes :
Query languages, Report generators, Application generators

Features
- Very easy to use
- Increases programmer productivity
- Less control over system resources

Examples
SQL, MATLAB, R, SAS, ABAP

2. BASED ON PARADIGM (STYLE OF PROGRAMMING)

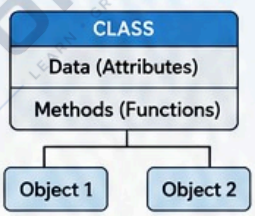
(A) PROCEDURAL LANGUAGES



Program is divided into procedures/functions.

Examples
C, Pascal, Fortran

(B) OBJECT-ORIENTED LANGUAGES



Uses classes and objects.
Supports inheritance, encapsulation, polymorphism.

Examples
C++, Java, Python, C#

(C) FUNCTIONAL LANGUAGES

Based on mathematical functions and avoids changing-state and mutable data.

Examples
Haskell, Lisp, Scala, F#

(D) SCRIPTING LANGUAGES

```
print("Hello")
for i in range(5):
   print(i)
```

Interpreted at runtime.
Often used for automation, web, and rapid development.

Examples
Python, JavaScript, PHP, Ruby, Shell (Bash)

3. BASED ON METHOD OF EXECUTION

(A) COMPILED LANGUAGES

Entire program is translated at once.
Errors are shown after compilation.

Examples
C, C++, Go, Rust

(B) INTERPRETED LANGUAGES

Program is translated and executed line by line.
Errors are shown line by line.

Examples
Python, JavaScript, PHP, Ruby

QUICK SUMMARY

- **Low-level** → Close to machine
- **High-level** → Close to human
- **4GL** → Problem-oriented
- **Paradigm** → How we structure and write programs
- **Execution** → How program is translated and run

WHY DIFFERENT TYPES?

- Different needs require different approaches.
- Improves efficiency, productivity and code maintainability.
- Helps choose the right language for the right problem.
- Drives innovation in software and technology.

HISTORY OF C LANGUAGE

> C is not a sector-specific language. It is a language for programming.
> – Dennis Ritchie

C is a **general-purpose, structured** programming language.
It is simple, efficient, portable and forms the foundation of many modern languages.

Why C is Special?

- Simple and efficient
- Portable (machine independent)
- Close to hardware
- Foundation of many modern languages
- Widely used in system programming

TIMELINE OF C LANGUAGE

Bell Labs
AT&T, USA

1970

1. 1970 – The Need for a New Language

Dennis Ritchie at Bell Labs needed a language to develop system software for the **UNIX** operating system.
Existing languages like Assembly were machine-dependent and hard to use.

Dennis Ritchie
Develops 'C'

1972

2. 1972 – Birth of C Language

Dennis Ritchie developed the C language.
C was created to be simple, efficient and portable.
It was used to rewrite **UNIX** almost completely in C.

The First Book
on C

1978

3. 1978 – The First Standard Book

Brian W. Kernighan and Dennis Ritchie published
"The C Programming Language" (also known as K&R Book).
This book became the definitive reference for C programmers.

ANSI Standard
for C

1989

4. 1989 – ANSI Standard (C89)

American National Standards Institute (ANSI) standardized C as "ANSI C".
Official name: ISO/IEC 9899:1990.
This version is also known as C89.

Influence on
Other Languages

1990s

5. 1990s – C and the World

C influenced many languages like **C++, Java, C#, JavaScript, PHP, Python** etc.
It became widely used in operating systems, compilers, databases, embedded systems and more.

C99
Standard

1999

6. 1999 – C99 Standard

ISO/IEC 9899:1999 (C99) introduced new features like inline functions, variable length arrays, // comments, and improved data types.

C11
Standard

2011

7. 2011 – C11 Standard

ISO/IEC 9899:2011 (C11) added features like _Atomic, threads, improved library functions and Unicode support.

C – Still
Going Strong

Today

8. Today and Beyond

C is still one of the most powerful and widely used languages.
It is the backbone of many technologies and continues to evolve.

MAJOR MILESTONES AT A GLANCE

1970	Idea of a new language at Bell Labs
1972	C language developed by Dennis Ritchie
1978	"The C Programming Language" published
1989	ANSI C (C89) standard
1999	C99 standard
2011	C11 standard
Today	C continues to power modern world

DID YOU KNOW?

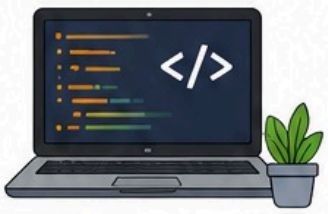
UNIX, the popular operating system, was mostly written in C.
Many parts of Windows, Linux, MySQL, Git, Python (core) and many other systems are written in C.

C LANGUAGE IN NUMBERS

- Developed in 1972
- Used in almost all Operating Systems
- Used in Compilers, Databases, Embedded Systems
- Basis for many modern programming languages

> C is a system programming language. It gives you access to everything a machine has to offer.
> – Dennis Ritchie

STRUCTURE OF C PROGRAM



A C program is written in a particular order.
It consists of different sections, and each section has a specific purpose.

1. Documentation Section

Used for programmer's comments.
Helps in understanding the program.

2. Link Section

Includes preprocessor directives.
It links header files to the program.

3. Definition Section

Used to define symbolic constants using #define keyword.

4. Global Declaration Section

Used to declare global variables and user-defined functions.

5. main() Function

Execution of a C program begins from the main() function.

6. Local Declaration Section

Variables declared inside main() or any function are local to that function.

7. Executable Section

Contains logic and statements.
Actual processing is done here.

8. Return Statement

Returns a value from the main() function to the calling environment.

9. End of Program

Indicates the end of program execution.

GENERAL STRUCTURE

```
/*  1. Documentation Section                          1
    This is a sample C program */

#include <stdio.h>        // 2. Link Section          2

#define PI 3.14159        // 3. Definition Section    3

int globalVar = 10;       // 4. Global Declaration Section  4
void display();

int main()                // 5. main() Function       5
{
   int localVar;          // 6. Local Declaration Section
   localVar = 5;                                      6

   printf("Hello, C Program!\n");
   display();             // 7. Executable Section    7

   return 0;              // 8. Return Statement      8
}

/* User-defined function */
void display()
{
   printf("This is a function.\n");
}

// 9. End of Program                                  9
```

FLOW OF EXECUTION

Preprocessor (#include, #define) → Global Declarations → main() Function Starts → Local Declarations → Execution of Statements → return 0; (Return Statement) → End of Program

EXAMPLE PROGRAM

```c
/* This is a sample C program */
#include <stdio.h>
#define PI 3.14159
int globalVar = 10;
void display();

int main()
{
    int localVar = 5;
    printf("Value of PI: %.2f\n", PI);
    printf("Global Variable: %d\n", globalVar);
    display();
    return 0;
}

void display()
{
    printf("This is a function.\n");
}
```

KEY POINTS

- Every C program must have a main() function.
- Execution always starts from main().
- #include is used to include header files.
- #define is used to define constants.
- return 0; indicates successful program termination.

GOOD PRACTICE

- Use meaningful comments.
- Indent your code properly.
- Use meaningful variable and function names.
- Keep your code modular using functions.

> Simplicity, efficiency, and portability make C language timeless.

C LANGUAGE – FUNDAMENTALS

Character Set, Identifiers, Keywords, Constants and Variables, Symbolic Constants, Qualifiers

1. CHARACTER SET

C language uses a set of basic characters to build programs.

Category	Characters
Alphabets	A – Z a – z
Digits	0 1 2 3 4 5 6 7 8 9
White Space	space tab newline \t \n
Special Symbols	! @ # $ % ^ & * ( ) { } [ ] : ; , . ? / \ < > = + - _ | ~

C is case-sensitive. 'A' and 'a' are different.

2. IDENTIFIERS

An identifier is a name given to program elements like variables, functions, arrays, structures, etc.

Rules

- Can contain letters, digits and underscore (_)
- Must start with a letter or underscore
- Cannot start with a digit
- No special symbols allowed (except _)
- Cannot be a keyword
- Case-sensitive

Examples

Valid Identifiers

✓ sum ✓ _value
✓ total_marks ✓ count1
✓ area2d

Invalid Identifiers

✗ 1value
✗ total-marks
✗ %value ✗ for
✗ for ✗ int

```
int total = 25;   // total is an identifier
```

3. KEYWORDS

Keywords are reserved words in C language.
They have special meaning and cannot be used as identifiers.

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

Note: Total 32 keywords in C (ANSI C).

4. CONSTANTS AND VARIABLES

Constants

Constants are fixed values that do not change during the execution of a program.

Types of Constants

- Integer Constants : 25, -10, 0
- Floating Constants : 3.14, -0.5
- Character Constants : 'A', '7', '\n'
- String Constants : "Hello"
- Enumeration Constants
- Symbolic Constants

Variables

Variables are named memory locations that store data whose value can change during program execution.

Example

```
int age = 20;
float price = 99.50;
char grade = 'A';
```

Difference

Constants	Variables
Value cannot change	Value can change
Uses fixed memory	Uses memory location
Literals or #define	Declared using data types

5. SYMBOLIC CONSTANTS

Symbolic constants are names given to constant values using preprocessor directive #define.

```
#define PI 3.14159
#define MAX 100
```

Advantages

- ✓ Improves readability
- ✓ Easy to modify
- ✓ Avoids magic numbers
- ✓ Used by preprocessor

#define

Example

```
#include <stdio.h>
#define PI 3.14159

int main()
{
    float r = 5.0, area;
    area = PI * r * r;
    printf("Area = %f", area);
    return 0;
}
```

6. QUALIFIERS

Qualifiers are used with data types to modify the properties of data or variables.

const	Makes a variable's value constant (read-only). `const int MAX = 100;`
volatile	Informs compiler that the variable's value can change unexpectedly. `volatile int flag;`
static	Retains value between function calls. Also used for limited scope. `static int count = 0;`
register	Suggests to store variable in CPU register for faster access. `register int i;`
extern	Declares a variable defined in another file. `extern int globalVar;`

QUICK SUMMARY

- Character set – Basic building blocks of C program.
- Identifiers – Names given to program elements.
- Keywords – Reserved words with special meaning.
- Constants – Fixed values.
- Variables – Changeable values.
- Symbolic constants – Named constants using #define.
- Qualifiers – Modify the meaning or storage of data.

EXAMPLE AT A GLANCE

```
#include <stdio.h>
#define PI 3.14159

int main()
{
    const int age = 20;        // constant
    int radius = 5;            // variable
    float area = PI * radius * radius;
    static int count = 0;      // qualifier
    count++;
    printf("Area = %f \n", area);
    return 0;
}
```

TIPS TO REMEMBER

- ☑ Always use meaningful identifiers.
- ☑ Do not use keywords as identifiers.
- ☑ Use constants for fixed values.
- ☑ Choose proper qualifiers to control variable behavior.
- ☑ Practice writing small programs using all these concepts.

★ Well-written programs start with the right building blocks!

C LANGUAGE
TYPE CONVERSION
OPERATORS AND EXPRESSIONS

1. TYPE CONVERSION

Type conversion is the process of converting a value from one data type to another.

A. IMPLICIT (AUTOMATIC) CONVERSION

Done automatically by the compiler when needed.

Generally from lower type to higher type to prevent data loss.

char → short → int → long → float → double

(small) ..... (large)

B. EXPLICIT (MANUAL) CONVERSION

Done manually by the programmer using type casting.

General form:

```
(data_type) expression
```

EXAMPLES

```
int i = 10;
float f = i;        // int to float
double d = f;       // float to double

printf("%d %f %lf", i, f, d);
```

```
int i = 25;
float f = 3.75;

int x = (int)f;     // float to int
char ch = (char)i;  // int to char

printf("%d %d %c", x, i, ch);
```

NOTE

- Implicit conversion is safe.
- Explicit conversion may cause loss of data.

DATA LOSS EXAMPLE

```
float f = 9.87;
int i = (int)f;     // i becomes 9
                    (decimal part lost)
```

2. OPERATORS OVERVIEW

Operators are special symbols that perform operations on operands (values or variables).

Arithmetic Operators	+ - * / % (Addition, Subtraction, Multiplication, Division, Modulus)
Relational Operators	== != > < >= <= (Equal to, Not equal to, Greater than, Less than, Greater or equal, Less or equal)
Logical Operators	&& || ! (Logical AND, OR, NOT)
Assignment Operators	= += -= *= /= %= etc.
Increment / Decrement Operators	++ -- (Increase or decrease value by 1)
Bitwise Operators	& | ^ ~ << >> (Bitwise AND, OR, XOR, NOT, Left shift, Right shift)
Conditional (Ternary) Operator	?: (condition ? expr1 : expr2)
Special Operators	sizeof & * , (Size of, Address of, Pointer dereference, Comma operator)

3. EXPRESSIONS

An expression is a combination of operands (constants, variables) and operators that evaluates to a single value.

EXAMPLE

```
int a = 10, b = 5, c;
c = a + b * 2;      // Valid expression
```

operand operator operand operator

TYPES OF EXPRESSIONS

- Arithmetic Expression
- Relational Expression
- Logical Expression
- Assignment Expression
- Mixed Expression

EXPRESSION EVALUATION (ORDER OF PRECEDENCE)

Expressions are evaluated according to operator precedence and associativity.

Example Expression: `a = 10 + 20 * 5 / 2 - 3;`

Evaluation Steps:

1. 20 * 5 = 100
2. 100 / 2 = 50
3. 10 + 50 = 60
4. 60 - 3 = 57

So, a = 57

GENERAL PRECEDENCE ORDER (High → Low)

1. () Parentheses
2. * / % Multiplicative
3. + - Additive
4. Relational
5. == !=
6. &&
7. = += -= etc. Assignment

4. OPERATORS IN DETAILS

Category	Operator	Meaning	Example	Result
Arithmetic	+	Addition	a + b	10 + 5 = 15
	-	Subtraction	a - b	10 - 5 = 5
	*	Multiplication	a * b	10 * 5 = 50
	/	Division	a / b	10 / 5 = 2
	%	Modulus	a % b	10 % 3 = 1
Relational	==	Equal to	a == b	10 == 5 → 0 (false)
	!=	Not equal to	a != b	10 != 5 → 1 (true)
	>	Greater than	a > b	10 > 5 → 1 (true)
	<	Less than	a < b	10 < 5 → 0 (false)
	>=	Greater or equal	a >= b	10 >= 5 → 1 (true)
	<=	Less or equal	a <= b	10 <= 5 → 0 (false)
Logical	&&	Logical AND	(a > 0 && b > 0)	
	||	Logical OR	(a > 0 || b > 0)	
	!	Logical NOT	!(a > 0)	
Assignment	=	a = 5		
	+=	a += 3 → a = a + 3		
	-=	a -= 3 → a = a - 3		
	*=	a *= 3 → a = a * 3		
	/=	a /= 3 → a = a / 3		
	%=	a %= 3 → a = a % 3		
Increment/ Decrement	++	a++ (post) / ++a (pre)		
	--	a-- (post) / --a (pre)		
Bitwise	&	Bitwise AND	a & b	
	|	Bitwise OR	a | b	
	^	Bitwise XOR	a ^ b	
	~	Bitwise NOT	~a	
	<<	Left Shift	a << 1	
	>>	Right Shift	a >> 1	
Conditional	?:	(a > b) ? a : b		
Special	sizeof	sizeof(a)	Size of a	
	&	&a	Address of a	
	*	*p	Value at address p	
	,	a = 5, b = 10	(comma operator)	

5. QUICK EXAMPLES

Arithmetic Expression

```
int a = 10, b = 3;
int c = a + b * 2;    // 10 + (3*2)
printf("%d", c);      // 16
```

Relational & Logical Expression

```
int x = 5, y = 10;
if (x < y && x != 0)
   printf("Condition True");
```

Type Conversion

```
float f = 7.89;
int i = (int)f;       // i = 7
```

Conditional Operator

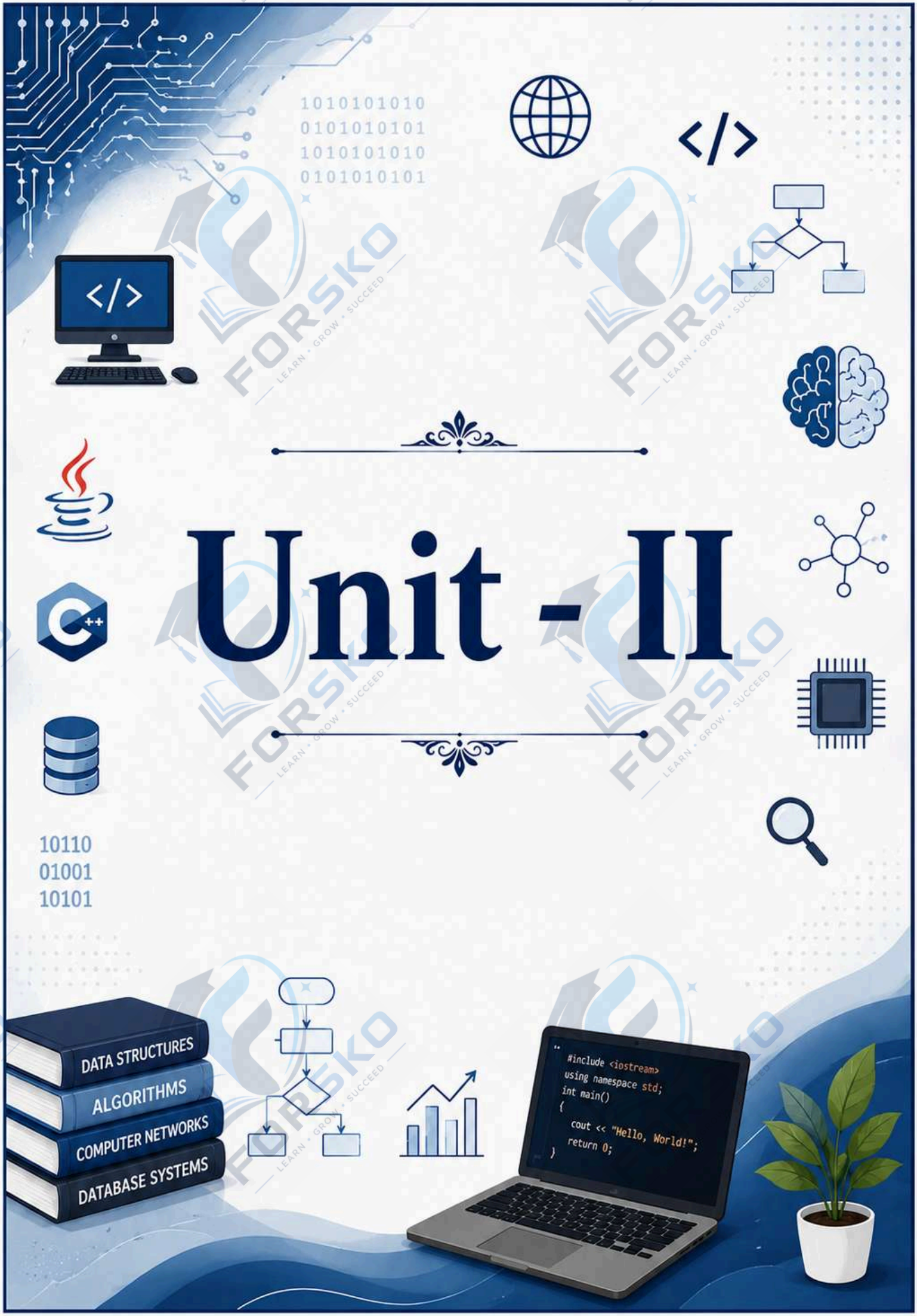
```
int max = (a > b) ? a : b;
```

TIPS TO REMEMBER

- ☑ Use parentheses to make expressions clear.
- ☑ Be careful with integer division and modulus.
- ☑ Type casting is useful but use it wisely.
- ☑ Understand operator precedence to avoid logic errors.

Understand Conversions. Use Operators Smartly. Write Better C Programs!

Unit - II


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

CONTROL STRUCTURES
BRANCHING: if, if-else

Control structures determine the order in which statements are executed.
Branching (Selection) allows a program to choose different paths based on conditions.

1. if STATEMENT

Executes a block of code only if the condition is TRUE.

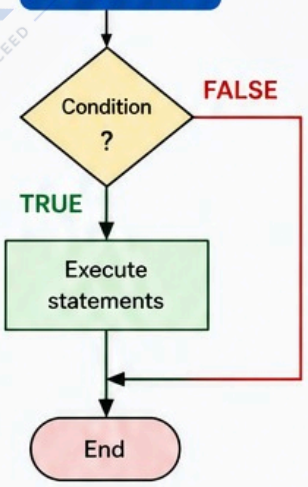
SYNTAX

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

How it works:

- If the condition is TRUE, the statements inside the if block are executed.
- If the condition is FALSE, the if block is skipped.

FLOWCHART



EXAMPLE

```
int marks = 75;

if (marks >= 50)
{
    printf("You are Pass.\n");
}
```

OUTPUT

```
You are Pass.
```

Used when we want to perform an action only when a condition is true.

2. if – else STATEMENT

Executes one block of code if the condition is TRUE and another block if the condition is FALSE.

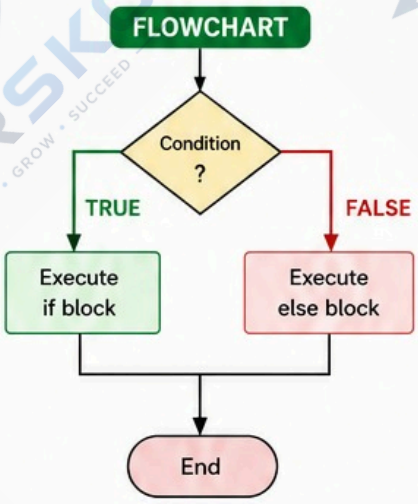
SYNTAX

```
if (condition)
{
   // statements if condition is TRUE;
}
else
{
   // statements if condition is FALSE;
}
```

How it works:

- If the condition is TRUE, the if block is executed.
- If the condition is FALSE, the else block is executed.

FLOWCHART



EXAMPLE

```
int num = 7;

if (num % 2 == 0)
{
    printf("Even Number\n");
}
else
{
    printf("Odd Number\n");
}
```

OUTPUT

```
Odd Number
```

Used when we want to choose between two alternatives.

QUICK COMPARISON

Feature	if Statement	if – else Statement
Purpose	Executes code when condition is TRUE	Executes one block when TRUE and another when FALSE
Number of paths	One path	Two paths
Else part	Not present	Present
Use case	When we only care about TRUE condition	When we need to handle both TRUE and FALSE cases

REAL-LIFE ANALOGY

if Statement:
If the light is green, Go. Otherwise do nothing.

if – else Statement:
If the road is clear, Go. Else, Stop.

KEY POINTS

- The condition in if or if-else must be inside parentheses ().
- A condition is TRUE when it is non-zero, and FALSE when it is zero.
- You can use relational operators (>, <, >=, <=, ==, !=) in conditions.
- Indentation improves readability but is not required in C.

MULTIPLE EXAMPLE

Example 1: if

```
int age = 18;

if (age >= 18)
{
    printf("You can vote.\n");
}
```

Output: You can vote.

Example 2: if – else

```
int x = -5;
if (x > 0)
{
    printf("Positive\n");
}
else
{
    printf("Non-positive\n");
}
```

Output: Non-positive

TIPS

- Start with simple conditions.
- Test both TRUE and FALSE cases.
- Use meaningful variable names for clarity.

CONDITIONAL OPERATOR (? :)

A short way to write if–else in a single line.

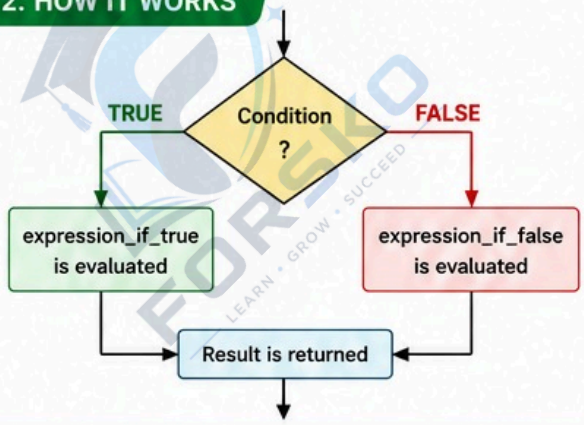
The conditional (ternary) operator evaluates a condition and returns one of two expressions based on whether the condition is TRUE or FALSE.

1. SYNTAX

```
condition ? expression_if_true : expression_if_false
```

- `condition` — Evaluated first. Must be a relational or logical expression.
- `expression_if_true` — Executed (returned) if condition is TRUE.
- `expression_if_false` — Executed (returned) if condition is FALSE.

2. HOW IT WORKS



3. EQUIVALENT TO if–else

Using Conditional Operator

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

≡

Using if–else

```
if (a > b)
    max = a;
else
    max = b;
```

4. EXAMPLES

```
// Find greater of two numbers
int max = (a > b) ? a : b;
printf("Max = %d", max);

// Check even or odd
printf("%d is %s", n, (n % 2 == 0) ? "Even" : "Odd");

// Assign grade
char grade = (marks >= 60) ? 'A' : 'B';

// Find absolute value
int abs_val = (x >= 0) ? x : -x;
```

5. IMPORTANT NOTES

- The condition is evaluated first.
- If TRUE, `expression_if_true` is evaluated and returned.
- If FALSE, `expression_if_false` is evaluated and returned.
- Only one of the two expressions is evaluated.
- Both expressions must be of compatible data types (or implicitly convertible).
- The operator is right-associative.
  Example: `a ? b : c ? d : e` is same as `a ? b : (c ? d : e)`

Great for short and simple decisions!

6. MORE EXAMPLES

Expression	Result (if condition is TRUE)	Result (if condition is FALSE)
`(x > y) ? x : y`	Returns x	Returns y
`(age >= 18) ? "Adult" : "Minor"`	"Adult"	"Minor"
`(ch == 'M') ? "Male" : "Female"`	"Male"	"Female"
`(n % 2 == 0) ? 1 : 0`	1	0
`(score >= 90) ? "A+" : "Below A+"`	"A+"	"Below A+"

7. COMMON USES

- Finding max, min of two numbers
- Checking even/odd
- Assigning grades
- Setting default values
- Replacing simple if–else statements to make code concise

8. PROGRAM EXAMPLE

```
#include <stdio.h>
int main() {
   int a, b, max, min;

   printf("Enter two numbers: ");
   scanf("%d %d", &a, &b);

   max = (a > b) ? a : b;    // conditional operator
   min = (a < b) ? a : b;

   printf("Maximum = %d\n", max);
   printf("Minimum = %d", min);

   return 0;
}
```

Sample Run

```
Enter two numbers: 12 7
Maximum = 12
Minimum = 7
```

9. TIPS

- Use it only for simple decisions.
- Overusing it can reduce readability.
- For complex logic, use regular if–else.
- Always ensure proper parentheses for clarity.

Think: `Condition ? Do this if TRUE : Do this if FALSE`

SWITCH STATEMENT

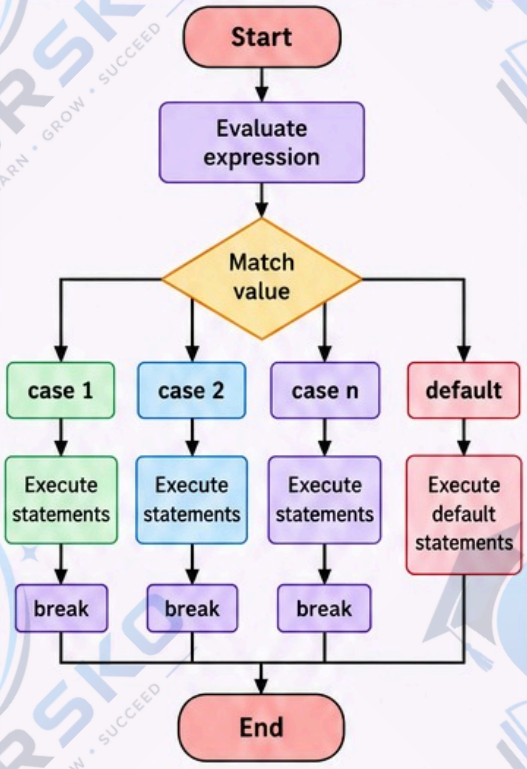
Multiple Choices, One Variable!

The **switch statement** is a **multi-way selection** control structure that allows a program to execute **one block of code** from many alternatives based on the value of an expression.

1 SYNTAX

```
switch (expression)
{
  case constant1:
     // statements
     break;

  case constant2:
     // statements
     break;

  ...

  default:
     // statements
}
```

2 FLOWCHART



3 EXAMPLE PROGRAM

```c
#include <stdio.h>

int main() {
   int day;
   printf("Enter a number (1-7): ");
   scanf("%d", &day);

   switch (day) {
      case 1:
         printf("Monday\n");
         break;
      case 2:
         printf("Tuesday\n");
         break;
      case 3:
         printf("Wednesday\n");
         break;
      case 4:
         printf("Thursday\n");
         break;
      case 5:
         printf("Friday\n");
         break;
      case 6:
         printf("Saturday\n");
         break;
      case 7:
         printf("Sunday\n");
         break;
      default:
         printf("Invalid number!\n");
   }

   return 0;
}
```

Sample Run

```
Enter a number (1-7): 3
Wednesday
```

4 HOW IT WORKS

1. The **expression** is evaluated once.
2. Its value is compared with each **case** constant (from top to bottom).
3. When a match is found, the corresponding block of statements is executed.
4. **break;** stops further execution and exits the **switch**.
5. If no match is found, the **default** block is executed (if present).

5 IMPORTANT NOTES

- The expression in **switch** must be of integer type (int, char, enum).
- Case constants must be unique and compile-time constants.
- Use **break;** to prevent fall-through (unintended execution of next case).
- **default** is optional.

Use switch when you have many possible choices for a single variable!

6 SWITCH vs IF-ELSE

Feature	switch	if-else
Purpose	Multiple choices for one variable	General purpose conditions
Condition type	Only int, char, enum	Any valid expression (relational, logical, etc.)
Comparison	Equality (==) only	Various (==, !=, >, <, >=, <=)
Readability	Better for many fixed values	Better for complex conditions
Use case	Menu, options, fixed choices	Range checks, complex logic

7 COMMON USES

- Menu-driven programs
- Simple calculator
- Days of the week
- Grade system
- Options in games and applications

8 QUICK TIPS

- Always use **break;** in each case.
- Keep **case** constants unique.
- Use **default** to handle invalid input.
- Use **switch** for cleaner and more readable code when dealing with multiple fixed choices.

One expression → Many choices → Clearer code!

Write Smart. Use **switch!**

STATEMENTS IN C

Statements are instructions that tell the computer what to do.
Every C program is made up of statements.

TYPES OF STATEMENTS IN C

1. EXPRESSION STATEMENT

An expression followed by a semicolon (;).
It evaluates an expression.

Syntax

```
expression ;
```

Example

```
a = b + 5;
x++;
```

Used for calculations, assignments, function calls.

2. COMPOUND STATEMENT (BLOCK)

A group of statements enclosed in curly braces { }.
Treated as a single statement.

Syntax

```
{
    statement1;
    statement2;
    ...
}
```

Example

```
{
    int x = 10;
    x = x + 5;
    printf("%d", x);
}
```

3. SELECTION (BRANCHING) STATEMENTS

Used to make decisions and choose different paths.

- if statement
- if–else statement
- else–if ladder
- switch statement

Example

```
if (x > 0)
    printf("Positive");
else
    printf("Non-positive");
```

4. ITERATION (LOOPING) STATEMENTS

Used to repeat a block of code multiple times.

- while loop
- do–while loop
- for loop

Example (for loop)

```
for (int i = 1; i <= 5; i++)
{
    printf("%d\n", i);
}
```

5. JUMP STATEMENTS

Used to transfer control to another part of the program.

- break – exits a loop or switch
- continue – skips to next iteration
- goto – jumps to a labeled statement
- return – returns from a function

Example

```
break;
continue;
return 0;
```

6. DECLARATION STATEMENT

Used to declare variables, functions, arrays, structures, etc.

Tells the compiler about the name and type.

Example

```
int a;
float price;
char name[20];
```

SUMMARY TABLE

Type of Statement	Purpose	Syntax / Form	Example
Expression	Evaluates an expression	`expression ;`	`x = y + 3;`
Compound	Group of statements	`{ statement1; statement2; ... }`	`{ int a = 5; a++; }`
Selection	Decision making	`if / if-else / else-if / switch`	`if (x > 0) { ... }`
Iteration	Repetition	`while / do-while / for`	`for (i=0; i<n; i++) { ... }`
Jump	Transfer control	`break / continue / goto / return`	`break; return 0;`
Declaration	Declare variables/functions	`type name;`	`int age;`

SUMPLE PROGRAM USING DIFFERENT STATEMENTS

```
#include <stdio.h>
int main()
{
    int i, sum = 0;                          // Declaration
    printf("Enter a number: ");
    scanf("%d", &i);                         // Expression (function calls)
    if (i > 0)                               // Selection
    {
        for (int j = 1; j <= i; j++)         // Iteration
        {
            if (j == 5)
                continue;                    // Jump
            sum += j;                        // Expression
        }
        printf("Sum = %d\n", sum);           // Expression
    }
    else
        printf("Invalid number!\n");         // Selection
    return 0;                                // Jump
}
```

KEY POINTS

- Every statement ends with a semicolon (;), except compound statements.
- Use braces { } to group multiple statements.
- Indentation improves readability but is not required.
- Choose the right statement based on what your program needs to do.

QUICK TIPS

- Use selection statements for decisions.
- Use iteration statements for repetition.
- Use jump statements carefully.
- Use meaningful variable names.
- Test your program with different values.

All programs in C are built using these statements.
Understand them well to write any program easily!

SPECIAL STATEMENTS & OPERATORS IN C

Comma Operator • goto Statement • break Statement • continue Statement

1 COMMA OPERATOR (,)

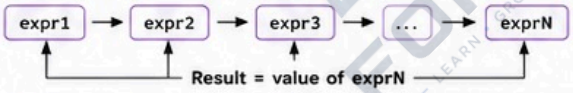
The comma operator evaluates multiple expressions from left to right and returns the value of the last expression.

SYNTAX

```
expr1 , expr2 , expr3 , ... , exprN
```

HOW IT WORKS

All expressions are evaluated in order (left to right). The result of the whole expression is the value of the last expression.



EXAMPLE

```
int a, b, c, d, result;
result = (a = 5, b = 10, c = a + b, d = c * 2);

printf("%d", result);    // Output: 30
```

NOTE

Parentheses are important to control order.

- Commonly used in for loops.
- Improves compactness but may reduce readability if overused.

USE CASES

- Initializing multiple variables in a for loop.
- Writing compact code.

FOR LOOP EXAMPLE

```
for (i = 0, sum = 0; i < n; i++,
    sum += i) {
  // loop body
}
```

2 GOTO STATEMENT

The goto statement transfers control to a labeled statement within the same function.

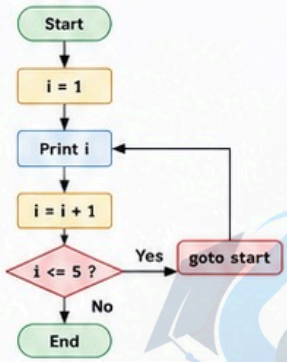
SYNTAX

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

EXAMPLE

```
#include <stdio.h>
int main() {
   int i = 1;
start:
   printf("i = %d\n", i);
   i++;
   if (i <= 5)
      goto start;    // jumps to 'start'
   return 0;
}
```

FLOW DIAGRAM



NOTES

- Avoid using goto in normal programs.
- It can make code hard to read and maintain.
- Use only when necessary (e.g., error handling, escaping from multiple loops).

USE CASES

- Jumping out of deeply nested loops.
- Error handling / cleanup code.
- Rarely used in modern programming.

3 BREAK STATEMENT

The break statement terminates the nearest enclosing loop or switch statement.

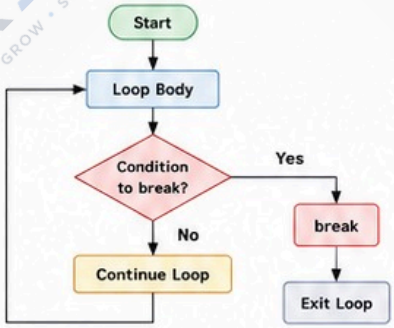
SYNTAX

```
break;
```

EXAMPLE (in loop)

```
for (i = 1; i <= 5; i++) {
   if (i == 3)
      break;       // exit loop when i is 3
   printf("%d ", i);
}
// Output: 1 2
```

FLOW DIAGRAM (in loop)



NOTES

- Immediately exits the loop/switch.
- Control comes to the statement after the loop/switch.

USE CASES

- Exit a loop when a condition is met.
- End a switch case early.

EXAMPLE (in switch)

```
swtich (ch) {
   case 'a': printf("Apple"); break;
   case 'b': printf("Banana"); break;
   default:  printf("Other");
}
```

4 CONTINUE STATEMENT

The continue statement skips the rest of the current iteration and proceeds to the next iteration of the loop.

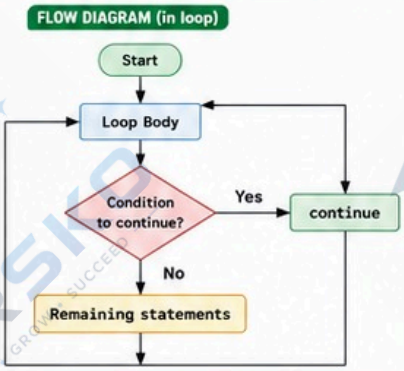
SYNTAX

```
continue;
```

EXAMPLE

```
for (i = 1; i <= 5; i++) {
   if (i == 3)
      continue;      // skip when i is 3
   printf("%d ", i);
}
// Output: 1 2 4 5
```

FLOW DIAGRAM (in loop)



NOTES

- Skips the remaining statements in the current iteration.
- Next iteration starts immediately.

USE CASES

- Skip unwanted values in a loop.
- Process only specific conditions.

EXAMPLE

```
for (i = 1; i <= 5; i++) {
   if (i % 2 == 0)
      continue;      // skip even numbers
   printf("%d ", i);
}
// Output: 1 3 5
```

QUICK COMPARISON

Feature	Comma Operator	goto	break	continue
Type	Operator	Jump Statement	Loop/Switch Control	Loop Control
Purpose	Evaluate multiple expressions	Jump to a labeled statement	Exit from loop or switch	Skip to next iteration
Where Used	Expressions, for loops	Within the same function	Loops, switch	Loops only
Effect	Returns last expression's value	Transfers control to label	Terminates the loop/switch	Skips current iteration

TIPS

- Prefer structured programming (avoid goto).
- Use break and continue wisely for clean and efficient code.
- Comma operator is useful in for loops but keep it readable.

Good code is not about using every feature, but about using the right feature at the right place.

NESTED LOOPS IN C

A loop inside another loop is called a **Nested Loop**.
The inner loop executes completely for each iteration of the outer loop.

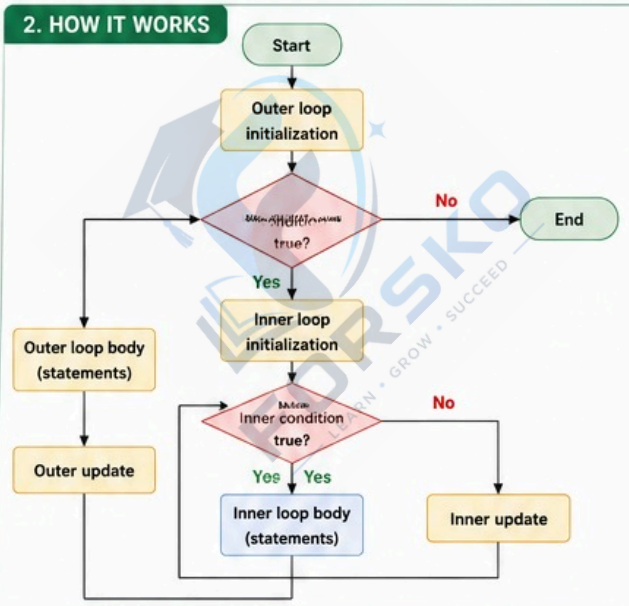
1. SYNTAX

```
for (outer initialization;
      outer condition; outer update)
{
    for (inner initialization;
          inner condition; inner update)
    {
        // statements in inner loop
    }

    // statements in outer loop
}
```

Outer loop controls the number of times the inner loop will run.

Inner loop executes completely inside the outer loop.

2. HOW IT WORKS



3. EXAMPLE CODE

Print a 3 × 4 rectangle of *

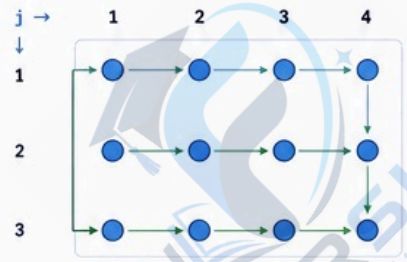
```
#include <stdio.h>
int main() {
  int i, j;
  for (i = 1; i <= 3; i++) {     // outer loop (rows)
    for (j = 1; j <= 4; j++) {   // inner loop (columns)
        printf("*");
    }
    printf("\n");  // new line after each row
 }
  return 0;
}
```

Output

```
* * * *
* * * *
* * * *
```

4. VISUAL REPRESENTATION

Flow of Execution



i → outer loop (rows)

j → inner loop (columns)

● → execution sequence

5. MORE EXAMPLES

A. Right Triangle

```
for (i = 1; i <= 5; i++) {
   for (j = 1; j <= i; j++) {
      printf("*");
   }
   printf("\n");
}
```

Output

```
*
**
***
****
*****
```

B. Inverted Right Triangle

```
for (i = 5; i >= 1; i--) {
   for (j = 1; j <= i; j++) {
      printf("*");
   }
   printf("\n");
}
```

Output

```
*****
****
***
**
*
```

C. Number Pattern

```
for (i = 1; i <= 4; i++) {
   for (j = 1; j <= i; j++) {
      printf("%d ", j);
   }
   printf("\n");
}
```

Output

```
1
1 2
1 2 3
1 2 3 4
```

6. NESTED LOOPS WITH DIFFERENT LOOP TYPES

Outer Loop	Inner Loop	Use When...	Example Use
for	for	Number of iterations is known for both loops.	Matrix operations, patterns
for	while	Outer count known, inner depends on a condition.	Reading variable-length data
while	for	Outer depends on a condition, inner has fixed count.	Menu-driven programs
while	while	Both depend on conditions.	Complex data processing
do-while	for/while	At least one execution required in outer loop.	User input validation

7. KEY POINTS

- The inner loop runs completely for each iteration of the outer loop.
- Total iterations = (Outer iterations) × (Inner iterations)
- Use nested loops for tables, patterns, matrices, and multi-dimensional tasks.
- Keep track of loop variables to avoid infinite loops.

8. COMMON USES

- Printing patterns
- Working with 2D arrays (matrices)
- Multiplication tables
- Searching and sorting in 2D data
- Game boards (tic-tac-toe, chess)

9. TIPS

- Always test with small values first.
- Use meaningful variable names (i, j, row, col).
- Indent your code properly for readability.
- Be careful with loop conditions and updates.

Practice different patterns and problems to master Nested Loops!

Unit - III


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

ARRAY – DECLARATION AND INITIALIZATION
ONE DIMENSIONAL ARRAY

An array is a collection of elements of the same data type stored in contiguous memory locations. Each element is accessed using its index.

1. DECLARATION

Declaration allocates memory for the array.

Syntax:

```
data_type array_name[size];
```

- data_type → data type of elements
- array_name → name of the array
- size → total number of elements

Example:

```
int arr[5];
```

This declares an integer array named `arr` that can store 5 elements.

Valid index range: 0 to 4

2. INITIALIZATION (Assigning values at the time of declaration)

A. Using Initialization List

Values are assigned inside curly braces {} at the time of declaration.

Syntax:

```
data_type array_name[size]={val1, val2, ..., valN};
```

Example:

```
int arr[5] = {10, 20, 30, 40, 50};
```

Memory representation:

- If size is not specified, compiler detects it automatically.
- `int arr[] = {10, 20, 30}; // size = 3`

B. Partial Initialization

If fewer values are provided, remaining elements are set to 0 (for int, float, char).

Example:

```
int arr[5] = {10, 20};
```

Memory representation:

Unspecified elements are initialized to 0 automatically.

KEY POINTS

- Array index always starts from 0.
- All elements of an array must be of the same data type.
- Array size must be a positive integer (constant).
- Access any element using: `array_name[index]`

Example Program

```
#include <stdio.h>
int main() {
    int marks[5] = {85, 90, 78, 92, 88};
    printf("Marks of first student: %d", marks[0]);
    return 0;
}
```

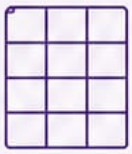
Output:

Marks of first student: 85

TWO DIMENSIONAL ARRAY (2D ARRAY)

Declaration and Initialization

What is a 2D Array?



A two dimensional array is an array of arrays arranged in the form of rows and columns (matrix). Each element is accessed using two indices: `[row][column]`.

Structure of 2D Array

Row ↓ \ Column →	0	1	2	...	(n-1)
0	a[0][0]	a[0][1]	a[0][2]	...	a[0][n-1]
1	a[1][0]	a[1][1]	a[1][2]	...	a[1][n-1]
2	a[2][0]	a[2][1]	a[2][2]	...	a[2][n-1]
⋮	⋮	⋮	⋮	⋮	⋮
(m-1)	a[m-1][0]	a[m-1][1]	a[m-1][2]	...	a[m-1][n-1]

If the array has m rows and n columns, then total number of elements = m × n

1. DECLARATION

Syntax:

```
data_type  array_name[row_size][column_size];
```

Example:

```
int arr[3][4];    // 3 rows and 4 columns
float mat[2][3];  // 2 rows and 3 columns
char table[4][5]; // 4 rows and 5 columns
```

2. INITIALIZATION

A 2D array can be initialized in the following ways:

(a) Initialization at the time of Declaration

Syntax:

```
data_type array_name[row_size][column_size] = {
   {val11, val12, val13, ...},     // 1st row
   {val21, val22, val23, ...},     // 2nd row
   .
   .
   {valm1, valm2, valm3, ...}      // mth row
};
```

Example:

```
int arr[3][3] = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
};
```

arr =

1	2	3
4	5	6
7	8	9

(b) Partial Initialization (Remaining elements become 0)

Example:

```
int arr[3][3] = {
   {1, 2},
   {4},
   {7, 8, 9}
};
```

arr =

1	2	0
4	0	0
7	8	9

(c) Initialization After Declaration

Example:

```
int arr[2][3];
arr[0][0] = 1;   arr[0][1] = 2;  arr[0][2] = 3;
arr[1][0] = 4;   arr[1][1] = 5;  arr[1][2] = 6;
```

arr =

1	2	3
4	5	6

KEY POINTS

- 2D arrays are stored in row-major order (row by row).
- Indexing starts from 0. The first element is `arr[0][0]`.
- Use two indices: `arr[i][j]` where i = row, j = column.
- Total elements = rows × columns.

QUICK EXAMPLE SUMMARY

Declaration: `int a[2][3];`
Initialization: `int a[2][3] = { {1,2,3}, {4,5,6} };`
Access: `a[1][2]` → 6

TIP: Always make sure the number of values matches the size (rows × columns) while initializing at the time of declaration.

STRUCTURE

Definition, Declaration, Initialization

A structure in C is a **user-defined data type** that allows you to group different data types under a single name.

1. DEFINITION

Definition tells the compiler how the structure looks.

```
struct StructureName
{
    dataType member1;
    dataType member2;
    dataType member3;
    ...
};
```

Example

```
struct Student
{
    int rollNo;
    char name[50];
    float marks;
};
```

Key Points

- The keyword `struct` is used.
- Structure members can be of different data types.
- Ends with a semicolon.

2. DECLARATION

Declaration creates variables of the defined structure type.

```
struct StructureName variable1;
struct StructureName variable2,
variable3, ...;
```

Example

```
struct Student s1;
struct Student s2, s3;
```

Memory Representation

s1	rollNo	name[50]	marks
s2	rollNo	name[50]	marks
s3	rollNo	name[50]	marks

Key Points

- Declares variables of structure type.
- No memory is wasted.
- Members can be accessed using dot (.) operator.

3. INITIALIZATION

Initialization assigns initial values to the members of a structure.

Method 1: Using Assignment

```
struct Student s1;
s1.rollNo = 101;
strcpy(s1.name, "Saquib");
s1.marks = 82.5;
```

Method 2: Using Initialization List

```
struct Student s2 = {101, "Saquib", 82.5};
```

Explanation (Method 2)

- Values are assigned in the same order as members are declared.
- For string, use " ".
- Works at the time of declaration.

Example

```
struct Student
{
    int rollNo;
    char name[50];
    float marks;
};

struct Student s1 = {101, "Saquib", 82.5};
```

Accessing Members

Syntax:

```
variableName.memberName
```

Example:

```
printf("Roll No: %d\n", s1.rollNo);
printf("Name: %s\n", s1.name);
printf("Marks: %.2f\n", s1.marks);
```

Important Notes

- Use . (dot) operator to access structure members.
- Structures can be passed to functions, returned, and used in arrays.
- Structures help in organizing related data in a single unit.

Benefits

- Organizes related data under one name.
- Improves code readability and maintenance.
- Useful for real-world entities (e.g., Student, Book).

ARRAY OF STRUCTURE

Store Multiple Records of Different Data Types

DEFINITION

An array of structure is an array in which each element is a structure variable. It is used to store multiple records, where each record can have different data types.

1. STRUCTURE EXAMPLE

Suppose we want to store information of students (name, roll number and marks).

```
struct Student {
    int roll;
    char name[30];
    float marks;
};
```

Members:

- roll (int)
- name (char array)
- marks (float)

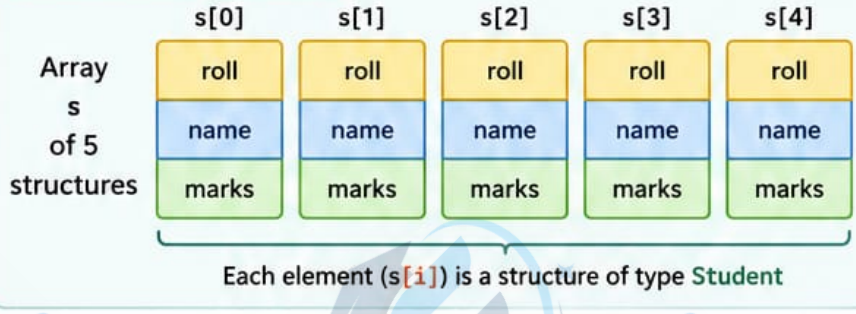
2. DECLARATION

Declare an array of structures just like any other array.

```
struct Student s[5];
```

- `Student` → Structure name
- `s[5]` → Array of 5 structures

VISUALIZATION



3. INITIALIZATION

Initialize an array of structures during declaration.

```
struct Student s[3] = {
    {101, "Aman", 85.5},
    {102, "Neha", 90.0},
    {103, "Rohan", 78.5}
};
```

Each set of values initializes one structure element.

4. ACCESSING ELEMENTS

Access members of a structure element using dot (.) operator.

```
s[1].roll;   // roll of 2nd student
s[1].name;   // name of 2nd student
s[1].marks;  // marks of 2nd student
```

Index starts from 0. `s[1]` is the second structure.

5. EXAMPLE PROGRAM

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

struct Student {
    int roll;
    char name[30];
    float marks;
};
int main() {
    struct Student s[3] ={
        {101, "Aman", 85.5},
        {102, "Neha", 90.0},
        {103, "Rohan", 78.5}
    };

    printf("\nStudent Details:\n");
    printf("Roll\tName  \tMarks\n");

    for(int i = 0; i < 3; i++) {
        printf("%d\t%s\t%.1f\n", s[i].roll, s[i].name, s[i].marks);
    }
    return 0;
}
```

Output

```
Student Details:
Roll    Name    Marks
101     Aman    85.5
102     Neha    90.0
103     Rohan   78.5
```

KEY POINTS

- Each element of the array is a complete structure.
- Useful for handling multiple records (as in databases).
- Access using: `arrayName[index].memberName`
- Can be initialized at the time of declaration or input at runtime.
- Array size must be fixed.

SUMMARY

Array of structure = Array in which each element is a structure.
It helps in storing and processing collection of similar records efficiently.

NESTED STRUCTURE & UNION

Organizing Complex Data in C

1. NESTED STRUCTURE

A structure that contains another structure as a member is called a **nested structure**.

EXAMPLE

Storing information of a Student which includes his Address.

```
struct Address {
   char city[20];
   char state[20];
   int pin;
};

struct Student {
   int roll;
   char name[30];
   struct Address addr;  // nested structure
   float percentage;
};
```

MEMORY REPRESENTATION

Memory layout of struct Student

addr (Address)

roll (int)	name (char[30])	city (char[20])	state (char[20])	pin (int)	percentage (float)

All members are stored in sequence.

ACCESSING MEMBERS

Use dot (.) operator for normal members and again dot (.) for members of the nested structure.

```
struct Student s;

s.roll = 101;
strcpy(s.name, "Aman");
strcpy(s.addr.city, "Pune");
strcpy(s.addr.state, "Maharashtra");
s.addr.pin = 411001;
s.percentage = 85.6;
```

Accessing nested structure members:

```
s.addr.city
s.addr.state
s.addr.pin
```

KEY POINTS (NESTED STRUCTURE)

- A structure can contain another structure as a member.
- Helps in representing real-world relationships.
- Improves code organization and readability.
- Access using: `outerMember.innerMember`

REAL-WORLD EXAMPLE

Student Record with Address

Roll
Name
Address (City, State, Pin)
Percentage

Address

City State Pin

2. UNION

A union is a user-defined data type in which all members share the **same memory location**.

Only one member can store a value at a time.

EXAMPLE

A variable that can store an integer, float or character but only one at a time.

```
union Data {
   int i;
   float f;
   char ch;
};

union Data d;
```

Members:

```
int i
float f
char ch
```

(Share same memory)

MEMORY REPRESENTATION

All members share the same memory location.

Memory

i (int)	f (float)	ch (char)

Size = Size of the largest member

USAGE

Store integer

```
d.i = 10;    // 10 is stored
```

Memory: 10

Store float (overwrites integer)

```
d.f = 3.14;
// 3.14 overwrites previous value
```

Memory: 3.14

Store character (overwrites float)

```
d.ch = 'A';
// 'A' overwrites previous value
```

Memory: 'A'

KEY POINTS (UNION)

- All members of a union share the same memory.
- Only one member can hold a value at a time.
- Size of union = size of its largest member.
- Useful when a variable can have one of several different data types (not all together).

REAL-WORLD EXAMPLE

Payment can be done by Credit Card, UPI ID or Cash Amount – only one at a time.

Payment

Card Details OR UPI ID OR

Cash Amount

SUMMARY

Nested Structure allows a structure inside another structure.
Union allows different data types to use the same memory location.

POINTERS
DECLARATION & INITIALIZATION

A pointer is a variable that stores the memory address of another variable.

1. DECLARATION OF POINTERS

A pointer is declared using * before the variable name.

Data Type → type * → * Pointer Name → pointer_name;

```
type *pointer_name;
```

Examples

```
int     *p;     // pointer to integer
char    *ch;    // pointer to character
float   *fptr;  // pointer to float
double  *dptr;  // pointer to double
int     **pp;   // pointer to pointer (int **)
```

Key Points

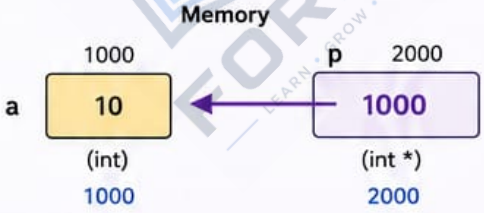
- & is the address-of operator.
- * is the indirection (dereference) operator.
- A pointer must be of the same data type as the variable it points to.
- Always initialize pointers to avoid unpredictable behavior.

2. INITIALIZATION OF POINTERS

A pointer can be initialized in different ways.

(A) Initialize with the address of a variable

```
int a = 10;
int *p = &a;   // p stores address of a
```



(B) Initialize with NULL (points to nothing)

```
int *p = NULL;   // recommended for safety
```

(C) Initialize with 0 (same as NULL)

```
int *p = 0;
```

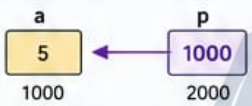
(D) Initialize with another pointer

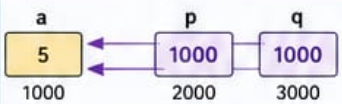
```
int *p, *q;
p = &a;   // p stores address of a
q = p;    // q now stores same address as p
```

DECLARATION SUMMARY

Declaration	Meaning	Example	Reads As
`type *ptr;`	Pointer to type	`int *p;`	p is a pointer to int
`type **ptr;`	Pointer to pointer to type	`char **pp;`	pp is a pointer to pointer to char
`type *ptr1, *ptr2;`	Multiple pointers	`int *a, *b;`	a and b are pointers to int
`type (*ptr);`	Pointer to type (using parentheses)	`int (*p);`	p is a pointer to int

INITIALIZATION EXAMPLES

Declaration & Initialization	Description	Example Diagram
`int a = 5;` `int *p = &a;`	p is initialized with address of a	
`int *p = NULL;`	p points to nothing (NULL)	
`int *p = 0;`	Same as NULL	

`int *p, *q;`
`p = &a;`
`q = p;`
	q is initialized with value of p	

TIPS

- Use NULL to initialize pointers when you don't have a valid address.
- Dereference using *ptr to access the value.
- Always check for NULL before using a pointer.

COMMON MISTAKES

- Using an uninitialized pointer.
- Dereferencing NULL pointer.
- Using wrong data type pointer for a variable.

SUMMARY

Declaration creates the pointer variable.
Initialization gives it a valid address or a safe starting value (NULL).

POINTERS ARITHMETIC

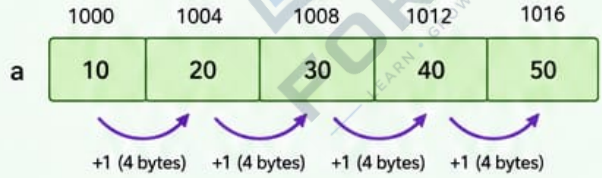
Moving Through Memory

Pointers store memory addresses. Pointer arithmetic moves the address by multiples of the data type size.

1. BASIC CONCEPT

When a pointer is incremented or decremented, it moves to the next or previous memory location according to the size of the data it points to.

Array of int (each int = 4 bytes)



2. POINTER ARITHMETIC OPERATORS

p + n	Moves n elements forward
p - n	Moves n elements backward
p++	Moves pointer to next element
p--	Moves pointer to previous element
p + n - p	Difference between two pointers (same array) = number of elements
p1 - p2	Valid only for pointers to the same array

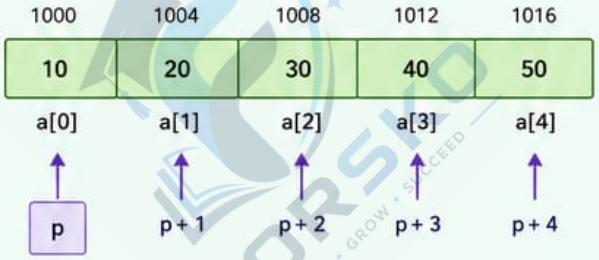
3. EXAMPLES

Let `int a[5] = {10, 20, 30, 40, 50}; int *p = a;`

```
p = p + 2;      // p now points to a[2] (value 30)
p++;            // p now points to a[3] (value 40)
p = p - 1;      // p now points to a[2] (value 30)
p--;            // p now points to a[1] (value 20)
```

4. VISUAL REPRESENTATION

`int a[5] = {10, 20, 30, 40, 50};  p = a`



5. COMMON POINTER ARITHMETIC EXAMPLES

Expression	Meaning	Example (int a[5])	Result / Points To
p + 2	Move 2 elements forward	a + 2	a[2] (address of 3rd element)
p - 2	Move 2 elements backward	a + 2 - 2	a[0] (address of 1st element)
p++	Move to next element	p++	Points to a[1]
p--	Move to previous element	p--	Points to a[0]
p + n - p	Difference in elements	(a + 3) - a	3 (elements)
p1 - p2	Distance between pointers	&a[4] - &a[1]	3 (elements)

IMPORTANT NOTES

- Pointer arithmetic is valid only within the same array (or one past the end).
- Adding or subtracting a number moves the pointer by multiples of sizeof(type).
- You cannot perform arithmetic on void pointers.
- Pointer comparison is allowed only within the same array.

INVALID OPERATIONS

- p1 + p2 (adding two pointers)
- p * n (multiplying pointer)
- p / n (dividing pointer)
- Pointer arithmetic outside array bounds

KEY TAKEAWAY

Pointer arithmetic lets you navigate through arrays efficiently by moving addresses based on the size of the data type.

SUMMARY

Pointers behave like integers when used in arithmetic. They move in units of the data type they point to, not in bytes.

p + n → move forward
p - n → move backward


1010101010
0101010101
1010101010
0101010101
</>
</>
Unit - IV
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

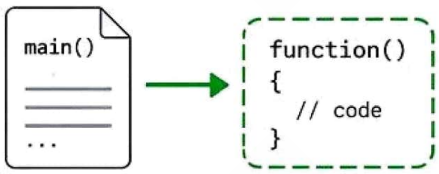
FUNCTIONS IN C

Functions break a large program into smaller, reusable blocks of code.

1 DEFINITION OF FUNCTION

A function in C is a block of code that performs a specific task.

It can be used repeatedly in a program, which improves modularity, readability and reusability of code.



2 FUNCTION PROTOTYPE

A function prototype is a declaration of a function before its use in the program.

It tells the compiler about the function's name, return type and parameters.

Syntax:

```
return_type function_name(data_type parameter_list);
```

Example:

```
int add(int a, int b);   // function prototype
```

Note:
The prototype ends with a semicolon (;)

3 CATEGORIES OF FUNCTIONS

A. BASED ON ORIGIN (SOURCE)

1. Library (Built-in) Functions

Predefined functions provided by C library. We can use them by including the appropriate header file.

Example:

```
printf(), scanf(), sqrt(), pow(), strlen()
```

2. User-Defined Functions

Functions created by the programmer to perform specific tasks.

Example:

```
int add(int a, int b) { ... }
```

B. BASED ON PARAMETERS

1. No Arguments, No Return Value
Function does not take arguments and does not return any value.

Example:
```
void show();
```

2. Arguments, No Return Value
Function takes arguments but does not return any value.

Example:
```
void printSum(int a, int b);
```

3. No Arguments, Return Value
Function does not take arguments but returns a value.

Example:
```
int getValue();
```

4. Arguments, Return Value
Function takes arguments and returns a value.

Example:
```
int add(int a, int b);
```

C. BASED ON USAGE

1. Recursive Functions

A function that calls itself to solve a problem by breaking it into smaller subproblems.

Example:

```
int factorial(int n) {
    if(n == 0) return 1;
    else return n * factorial(n-1);
}
```

2. Non-Recursive Functions

A function that does not call itself.

Example:

```
int sum(int n) {
    int s = 0;
    for(int i = 1; i <= n; i++)
        s += i;
    return s;
}
```

KEY TAKEAWAY

- Functions make programs modular, reusable and easy to maintain.
- Use function prototype to inform the compiler before use.
- Functions can be categorized based on origin, parameters and usage.

ACTUAL ARGUMENTS AND FORMAL ARGUMENTS

They are used in functions to pass data between the calling function and the called function.

1. ACTUAL ARGUMENTS (Actual Parameters)

Actual arguments are the values or variables that are passed to a function when it is called.

- They are written in the function call.
- They send values to the called function.
- They can be constants, variables or expressions.

In Function Call

```
- add( 10, 20 );
```

Actual argument 1 | Actual argument 2

Actual arguments appear in the calling function.

2. FORMAL ARGUMENTS (Formal Parameters)

Formal arguments are the variables listed inside the function definition.

- They receive values from the actual arguments.
- They act as local variables inside the function.
- They exist only while the function is executing.

In Function Definition

```
int add( int a, int b )
{
    // function body
}
```

Formal parameter 1 | Formal parameter 2

Formal arguments appear in the called function.

3. EXAMPLE

Program

```
#include <stdio.h>

void add(int a, int b) {  // formal arguments
    int sum = a + b;
    printf("Sum = %d\n", sum);
}

int main() {
   int x = 10, y = 20;
   add(x, y);   // actual arguments
   return 0;
}
```

In main()

```
int x = 10;
int y = 20;

add(x, y);
```

(Actual Arguments)

Values of x and y are passed to function

In add() function

```
int a, int b
(Formal Arguments)
a = 10;   b = 20;
int sum = a + b;
printf("Sum = %d", sum);
```

Output: Sum = 30

4. KEY DIFFERENCES

Feature	Actual Arguments	Formal Arguments
Also called	Actual Parameters	Formal Parameters
Where they appear	In the function call	In the function definition
Purpose	Pass values to the function	Receive values inside the function
Example	`add(10, 20)`	`int add(int a, int b)`
Lifetime	Exist in the calling function	Exist only during function execution
Can be	Constants, variables or expressions	Only variables (parameters)
Number	Must match in type, order and number	Must match in type, order and number

IMPORTANT NOTES

- The number, type and order of actual arguments must match with formal arguments.
- Actual arguments are passed by value in C (default).
- Changes made to formal arguments do not affect actual arguments.

QUICK TIP

Think of it like a post office:

- Actual arguments are the parcels you send.
- Formal arguments are the boxes that receive those parcels.

SUMMARY — Actual arguments send data to the function. | Formal arguments receive that data and work in the function.

FUNCTION CALLING IN C

CALL BY VALUE, CALL BY REFERENCE & FUNCTION PARAMETERS

A function is executed when it is called. Data can be passed to a function through parameters.

1. FUNCTION CALLING

When a function is called, the control jumps to the function, statements inside the function are executed and then control returns to the calling function.

How it works?

Function Call
(from main/other function)
↓
Function Definition
(code executes)
↓
Return to Calling Function
(after completion)

2. FUNCTION PARAMETERS

Parameters are variables listed in the function definition.

- They act as placeholders.
- They receive values when the function is called.
- Types and number must match with arguments.

```
return_type function_name(type p1, type p2, ...)
{
    // function body
}
```

Example

```
int add(int a, int b)   // a and b are parameters
```

3. CALL BY VALUE

In call by value, a copy of the actual value is passed to the function.

- Changes made to the parameter inside the function do not affect the original value.
- Memory is separate for actual argument and parameter.

Main Function

```
int x = 10;
add(x);
  // x is 10
  (no change)
```

Function

```
void add(int a)
{
  a = a + 5;
    // only 'a'
    // changes
}
```

Example

```
#include <stdio.h>
void add(int a)
{
  a = a + 5;
  printf("Inside: %d", a);
}
```

```
int main()
{
  int x = 10;
  add(x);
  printf("\nOutside: %d", x);
  return 0;
}
```

Output:

```
Inside: 15
Outside: 10
```

4. CALL BY REFERENCE (USING POINTERS)

In call by reference, the address of the actual variable is passed to the function.

- The function works on the original variable.
- Changes made inside the function affect the original value.
- Pointers are used to implement this in C.

Main Function

```
int x = 10;
add(&x);
// address of x
  is passed
```

Address of x (&x) →

Function

```
void add(int *p)
{
  *p = *p + 5;
    // original x
    // is changed
}
```

Memory

Address	Value
&x	15 (Changed)

Example

```
#include <stdio.h>
void add(int *p)
{
  *p = *p + 5;
  printf("Inside: %d", *p);
}
```

```
int main()
{
  int x = 10;
  add(&x);
  printf("\nOutside: %d", x);
  return 0;
}
```

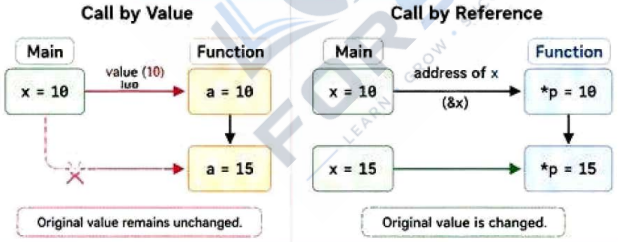
Output:

```
Inside: 15
Outside: 15
```

KEY DIFFERENCE

Feature	Call by Value	Call by Reference
What is passed?	Actual value (copy)	Address of actual variable
Parameter type	Normal variables	Pointers
Effect of changes inside function	Does NOT affect original value	Affects original value
Memory	Separate memory for parameter	Same memory (original variable)
Efficiency	Less efficient	More efficient
Use	Quando we don't want to modify original data	Quando we want to modify original data

SUMMARY DIAGRAM



USE WHEN?

Use Call by Value when:

- You only need to read data.
- You don't want to change the original data.

Use Call by Reference when:

- You want to modify original data.
- You are working with large data (arrays, structures, etc.) for better performance.

QUICK NOTES

- In C, call by reference is achieved using pointers.
- Arrays are naturally passed by reference (array name is the address of first element).
- Always pass valid addresses to functions.

REMEMBER

Functions make code modular and reusable. Choose the right way of passing data to make your program efficient, safe and easy to understand.

LOCAL AND GLOBAL VARIABLES

Variables in C can be declared inside or outside a function.
They differ in scope, lifetime and accessibility.

1. GLOBAL VARIABLES

A global variable is declared outside all functions.
It can be accessed (used) by any function in the program.

- **Declaration:** Outside all functions
- **Scope:** Whole program
- **Lifetime:** Entire program execution
- **Default value:** 0 (zero) for int, 0.0 for float, NULL for pointers
- **Access:** Can be accessed by any function

Example:

```c
#include <stdio.h>
int g = 10;     // global variable
void display() {
    printf("Global variable g = %d\n", g);  // accessible
}

int main() {
    printf("In main, g = %d\n", g);       // accessible
    display();
    return 0;
}
```

2. LOCAL VARIABLES

A local variable is declared inside a function or a block.
It can be accessed only within that function or block.

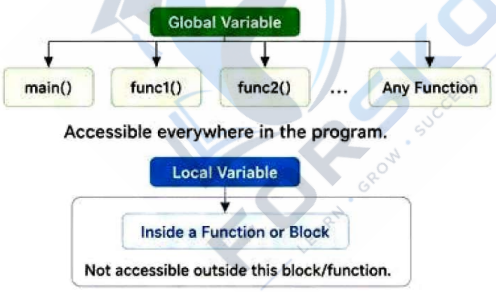
- **Declaration:** Inside a function or block
- **Scope:** Only within that function/block
- **Lifetime:** Only during execution of that function/block
- **Default value:** Garbage value (not initialized)
- **Access:** Cannot be accessed outside the function/block

Example:

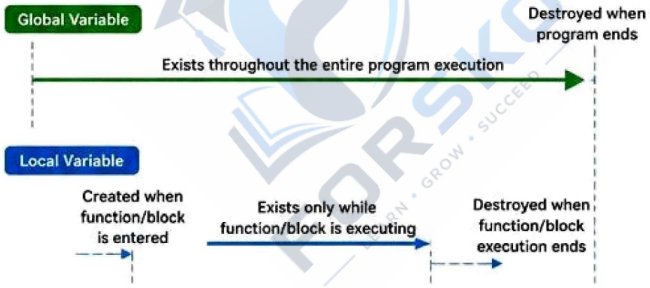
```c
#include <stdio.h>
void display() {
    int x = 5;    // local variable
    printf("Local variable x = %d\n", x);  // accessible
}

int main() {
    // printf("x = %d", x);   // Error: x is local to display()
    display();
    return 0;
}
```

3. SCOPE AND LIFETIME COMPARISON

Scope (Where it can be Accessed)



Lifetime (How Long it Exists)



4. EXAMPLE PROGRAM

```c
#include <stdio.h>
int g = 100;     // global variable
void change() {
    int x = 10;               // local to change()
    g = g + 1;                // can access global variable
    printf("Inside change(): x = %d, g = %d\n", x, g);
}
int main() {
    int y = 20;               // local to main()
    printf("In main(): y = %d, g = %d \n", y, g);
    change();                 // call function
    printf("In main() after call: g = %d \n", g);
    // printf("x = %d", x);   // Error: x is local to change()
    return 0;
}
```

5. OUTPUT

```
In main(): y = 20, g = 100
Inside change(): x = 10, g = 101
In main() after call: g = 101
```

Explanation

- y is local to main(), so it cannot be used in change().
- x is local to change(), so it cannot be used in main().
- g is global, so both functions can access it.
- Value of g is changed inside change(), and the change is reflected in main().

6. KEY DIFFERENCES

Feature	Global Variable	Local Variable
Declaration	Outside all functions	Inside a function or block
Scope	Whole program	Only within that function/block
Lifetime	Entire program execution	Only during function/block execution
Default value	0 (zero)	Garbage value
Accessibility	Accessible by any function	Accessible only inside the function/block
Memory	Allocated once	Allocated each time function is called
Example	`int g;`	`int x;` (inside function)

7. BEST PRACTICES

- Use local variables whenever possible.
- Use global variables only when necessary.
- Excessive use of global variables can make code difficult to debug and maintain.
- Prefer passing data through function parameters.

SUMMARY

Global variables are accessible throughout the program and exist for the entire execution.
Local variables are limited to the function or block and exist only during its execution.

FUNCTION WITH ARRAY IN C

An array can be passed to a function so that the function can process all its elements.

1 WHY USE FUNCTIONS WITH ARRAY?

- Improve modularity (break code into reusable parts)
- Reuse the same code for different arrays
- Make programs easier to read, test and maintain

2 HOW ARRAY IS PASSED TO A FUNCTION?

- In C, an array name represents the base address of its first element.
- When an array is passed to a function, its base address is passed.
- The function receives the address and uses it to access all elements.

Array in main()	Passed to function	In function
`int arr[5]`	`arr`	`int arr[]` or `int *arr`
(Base address)	(Address is passed)	(Address received)

3 SYNTAX

Without Size (1D Array)

```
return_type function_name(data_type arr[]) {
    // function body
}
```

Function Call: `function_name(array_name);`

With Pointer (Recommended)

```
return_type function_name(data_type *arr, int size) {
    // function body
}
```

Function Call: `function_name(array_name, size);`

4 EXAMPLES

a) Print Array Elements

```
#include <stdio.h>

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

Output: `10 20 30 40 50`

b) Find Sum of Array Elements

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

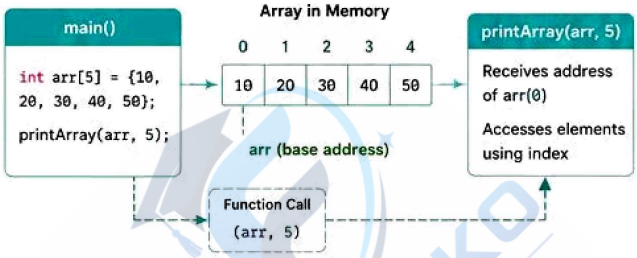
Output: `Sum = 15`

c) Modify Array Elements

```
#include <stdio.h>
void doubleArray(int arr[], int n) {
    for(int i = 0; i < n; i++)
        arr[i] = 2 * arr[i];
}
int main() {
    int a[4] = {1, 2, 3, 4};
    doubleArray(a, 4);   // array modified
    for(int i = 0; i < 4; i++)
        printf("%d ", a[i]);
    return 0;
}
```

Output: `2 4 6 8`

5 HOW IT WORKS (DIAGRAM)



6 IMPORTANT NOTES

- Always pass array size to the function.
- Array name in function acts like a pointer.
- Changes made to array elements inside the function reflect in the original array.
- You can also use pointer notation to receive array.

Using Array Notation		Using Pointer Notation
`void func(int arr[], int n) { ... }`	OR	`void func(int *arr, int n) { ... }`

7 ARRAY PASSING METHODS COMPARISON

Method	Declaration in Function	Function Call	Can Modify Original Array?	Needs Size?	Efficiency
Using Array ([])	`void f(int arr[], int n)`	`f(arr, n);`	Yes	Yes	High
Using Pointer (*)	`void f(int *arr, int n)`	`f(arr, n);`	Yes	Yes	High
Without Size	`void f(int arr[])`	`f(arr);`	Yes	No (Not Recommended)	High

SUMMARY

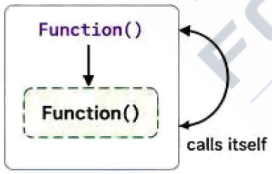
Functions with arrays make programs modular and reusable. Always pass the array and its size. The array is passed by address, so changes inside the function affect the original array.

FUNCTION RECURSION IN C

When a function calls itself directly or indirectly to solve a problem.

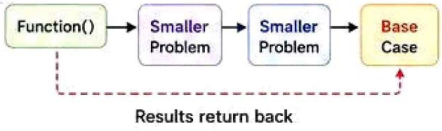
1. WHAT IS RECURSION?

- A technique where a function calls itself.
- It breaks a large problem into smaller subproblems.
- Recursion must have a base case to stop.



2. HOW RECURSION WORKS?

- Function is called with some input.
- It processes the input.
- It calls itself with a smaller (or simpler) input.
- When the base case is reached, it stops calling itself.



3. COMPONENTS OF RECURSION

1. **Base Case (Termination Condition)**
   Condition under which function stops calling itself.
2. **Recursive Case**
   Condition under which function calls itself with modified input.
3. **Progress Towards Base Case**
   Each recursive call should bring the solution closer to the base case.

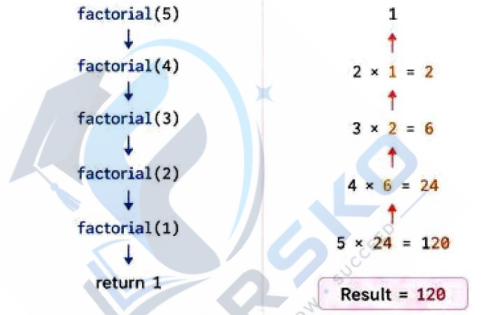
4. EXAMPLE PROGRAM : FACTORIAL OF A NUMBER

```c
#include <stdio.h>
int factorial(int n) {
    if (n == 0 || n == 1)
        return 1;              // Base case
    else
        return n * factorial(n - 1); // Recursive case
}
int main() {
   int num = 5;
   int fact = factorial(num);
   printf("Factorial of %d is %d\n", num, fact);
   return 0;
}
```

Logic:

- 5! = 5 × 4!
- 4! = 4 × 3!
- 3! = 3 × 2!
- 2! = 2 × 1!
- 1! = 1 (Base case)
- 0! = 1 (Base case)

5. CALL STACK TRACE (factorial(5))



6. ADVANTAGES

- ✓ Makes code shorter and easier to write for problems that have repetitive nature.
- ✓ Useful for problems like tree traversal, sorting, backtracking, divide and conquer.

7. DISADVANTAGES

- ✗ Uses more memory (function call stack).
- ✗ May be slower than iteration.
- ✗ Too many recursive calls can cause stack overflow.

8. TYPES OF RECURSION

1. Direct Recursion

A function calls itself directly.

```c
void fun() {
    fun();
}
```

2. Indirect Recursion

A function calls another function, which calls the first function.

```c
void A() { B(); }
void B() { A(); }
```

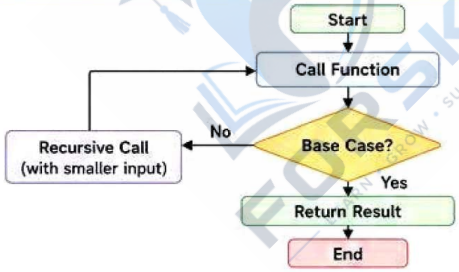
9. ANOTHER EXAMPLE : FIBONACCI SERIES

```c
#include <stdio.h>
int fib(int n) {
    if (n <= 1)
        return n;          // Base case
    else
        return fib(n-1) + fib(n-2); // Recursive case
}
int main() {
   int n = 6;
   printf("Fibonacci series: ");
   for (int i = 0; i < n; i++)
       printf("%d ", fib(i));
   return 0;
}
```

Output:

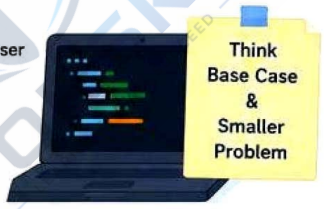
```
Fibonacci series: 0 1 1 2 3 5
```

10. RECURSION FLOW



11. TIPS FOR WRITING RECURSIVE FUNCTIONS

1. Identify the base case clearly.
2. Make sure each recursive call moves closer to the base case.
3. Test with small inputs first.
4. Visualize the call stack.
5. Avoid unnecessary recursion.



SUMMARY

- ★ Recursion is a powerful technique where a function calls itself.
- ★ Always define a base case to stop the recursion.
- ★ Useful in many real-world problems like factorial, fibonacci, tree traversal, sorting, etc.

IMPORTANT NOTE Infinite recursion (no base case) leads to stack overflow and program crash.

STRING FUNCTIONS IN C

String functions are used to perform common operations on strings. These functions are defined in <string.h> header file.

1. strlen() – String Length

Returns the length of the string (number of characters before the null character '\0').

Syntax

```
size_t strlen(const char *str);
```

Example

```
char str[] = "Hello, World!";
printf("Length = %zu", strlen(str));
```

```
Output: Length = 13
```

How it works?

Counts characters from the beginning until it reaches the null character '\0'.

2. strcpy() – String Copy

Copies the source string (src) into destination string (dest). The null character is also copied.

Syntax

```
char *strcpy(char *dest, const char *src);
```

Example

```
char src[] = "Hello";
char dest[20];
strcpy(dest, src);
printf("Copied string: %s", dest);
```

```
Output: Copied string: Hello
```

Notes

- dest must have enough space to hold src.

3. strcmp() – String Compare

Compares two strings lexicographically.

Returns:

- 0 → if both strings are equal
- < 0 → if str1 is less than str2
- > 0 → if str1 is greater than str2

Syntax

```
int strcmp(const char *str1, const char *str2);
```

Example

```
char s1[] = "apple";
char s2[] = "apricot";
int result = strcmp(s1, s2);
printf("Result = %d", result);  // Output: negative value
```

How it works?

Compares characters one by one (based on ASCII values) until a difference is found or both strings end.

4. strcat() – String Concatenation

Appends the source string (src) at the end of destination string (dest). The null character of dest is overwritten.

Syntax

```
char *strcat(char *dest, const char *src);
```

Example

```
char dest[20] = "Good ";
char srt[] = "Morning!";
strcat(dest, src);
printf("After concatenation: %s", dest);
```

```
Output: After concatenation: Good Morning!
```

Notes

- dest must have enough space to hold the result.

5. QUICK COMPARISON

Function	Purpose	Syntax	Returns	Modifies Strings?	Example Use
strlen()	Finds length of a string	size_t strlen(const char *str);	Length (size_t)	No	strlen("C Language") → 10
strcpy()	Copies src to dest	char *strcpy(char *dest, const char *src);	Pointer to dest	Yes (dest)	strcpy(dest, src);
strcmp()	Compares two strings	int strcmp(const char *s1, const char *s2);	0, < 0 or > 0	No	strcmp(s1, s2);
strcat()	Appends src to dest	char *strcat(char *dest, const char *src);	Pointer to dest	Yes (dest)	strcat(dest, src);

6. COMPLETE EXAMPLE

```
#include <stdio.h>
#include <string.h>
int main() {
  char str1[20] = "Hello";
  char str2[20];
  char str3[20] = "Hello";
  char str4[] = " World!";
  size_t len = strlen(str1);
  strcpy(str2, str1);
  int cmp = strcmp(str1, str3);
  strcat(str1, str4);
  printf("Length: %zu\n", len);
  printf("Copied String: %s\n", str2);
  printf("Compare Result: %d\n", cmp);
  printf("Concatenated String: %s\n", str1);
  return 0;
}
```

Output

```
Length: 5
Copied String: Hello
Compare Result: 0
Concatenated String:
Hello World!
```

Explanation

1. strlen() → 5
2. strcpy() → str2 = "Hello"
3. strcmp() → 0 (strings are equal)
4. strcat() → str1 = "Hello World!"

7. IMPORTANT NOTES

- All these functions are in <string.h>.
- Strings in C are arrays of characters ending with the null character '\0'.
- Always ensure the destination array has enough space.
- The strcmp() is case-sensitive.
- These functions do not check for buffer overflow.

REMEMBER

strlen → length
strcpy → copy
strcmp → compare
strcat → concatenate

KEY TAKEAWAY

Master these string functions – they are the building blocks for handling strings in C programming!

Always allocate enough memory and avoid buffer overflow.