CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)
(An Autonomous Institution, Affiliated to Anna University, Chennai)

NIRF 151 - 200 Band Engineering 2021 | NAAC A+

HAND WRITTEN MATERIAL

I Yr/I Sem **CS5102-C++ PROGRAMMING**

Course Objectives:

- To give a foundation in C programming.
- To provide comprehensive understanding of object-oriented principles.
- To gain advanced knowledge of the concepts such as inheritance and polymorphism in C++.
- To equip with advanced C++ skills in exception handling and generic programming.
- To handle the files using C++.

Course Outcomes:

At the end of the course the students will be able to

- CO1: Solve complex problems using modular and maintainable C code.
- CO2: Implement object-oriented features including classes, objects, pointers and encapsulation.
- CO3: Implement string handling, polymorphism and inheritance using C++
- CO4: Implement exception handling and generic programming with templates using C++
- CO5: Implement I/O streams using C ++ and develop simple applications.

UNIT I - OVERVIEW OF C

Introduction to C- C Program Structure- Basic Syntax and Structure, Variables and Data Types, Operators, Decision control and loops, Arrays, String, Functions and Structures.

CO-PO mapping Matrix:

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	2	3	-	-	-	-	-	-	-	2	2
CO2	3	2	3	-	-	-	-	-	-	-	2	2
CO3	3	2	3	-	-	-	-	-	-	-	2	2
CO4	3	2	3	-	-	-	-	-	-	-	2	2
CO5	3	2	3	-	-	-	-	-	-	-	2	2

Prepared By Verified By Approved By

UNIT-I : OVERVIEW OF C

Introduction to C- C Program Structure- Basic Syntax and Structure, Variables and Data Types, Operators, Decision control and loops, Arrays, String, Functions and Structures.

1. Introduction to C

C is a general-purpose, high-level programming language develop in the early 1970s by Dennis Ritchie at Bell Labs. It is widely used for system programming, embedded systems, and application software due to its efficiency and control over hardware.

History

- Developed in 1972 by Dennis Ritchie.
- Designed for writing the UNIX operating system.
- Influenced many later languages like C++, Java, and Python.

Features of C

- Simple and Efficient : Provides a clear and straightforward syntax.
- Portability : Programs written in C can be run on different Machines with Minimal changes.
- Low-level Access : Allows Manipulation of bits, bytes and addresses using pointers.
- Modularity : Supports functions to divide programs into smaller, Manageable pieces.
- Rich Library : Comes with a standard library offering many useful functions.
- Structured language : Supports structured programming techniques for Clarity and ease of Maintenance.

Uses of C

- System software development (e.g. operating systems, compilers).
- Embedded Systems (e.g., Microcontrollers, IoT Devices).
- Application software (e.g. games, GUI - based programs).
- Developing libraries and frameworks.

2. C Program Structure

C program is organized into six main sections:

Documentation

Contains comments describing the program, author, date and purpose. It does not affect the program execution but helps readers understand the code.

Preprocessor Section

Includes header files (#Include) needed for Input/output and other functionalities. These files are inserted before compilation.

Definition

Contains Preprocessor directives like #define to create constants or Macros used throughout the program

Global Declaration

Declares global variables and function prototypes accessible throughout the program.

Main() Function

The entry point of the program where execution begins. It contains the core logic and can return an integer (int main()) or no value (void main())

Sub Programs
User-defined functions that perform specific tasks and are called from the main function or other parts of the program.

Example Program

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

// Definition
#define X 20

// Global Declaration
int sum(int y);

// Main Function
int main(void) {
    int y = 55;
    printf("Sum: %d", sum(y));
    return 0;
}

// Sub Program
int sum(int y) {
    return y + X;
}
```

Output:
Sum: 75

Explanation of Example
- Documentation describes the program.
- Header file <stdio.h> allows use of printf.
- #define X 20 defines a constant.
- Function prototype int Sum(int y); declared globally.
- main() executes first, calling Sum(y) and printing the result.
- Sum() adds the argument y and constant X and returns the result.
- The program outputs the Sum of 55 + 20 = 75.

Compilation and Execution Steps
- Write the Program (Source code).
- Compile the program to convert it to Machine code.
- Execute the compiled program to see the output.

3. Basic Syntax and Structure of a C program

Case Sensitivity
C is case sensitive. for example, variable and variable are treated as different identifiers.

Semicolon (;)
Each statement must end with a semicolon to Mark the end of the instruction

Braces ({})
Curly braces define the beginning and end of function bodies and blocks of codes.

Comments
Comments are used to document the code and are ignored by the compiler.
- Single-line comment: // Comment here
- Multi-line comment:

```
/*
  Multiple
  line comments
*/
```

Main Function
Every C program starts execution from the main() function, which is mandatory

Function Structure
A typical function consists of a return type, function name, parameters and a body;

```
return_type function_name(parameter_list) {
    // statements
}
```

Variables and Data Types
Variables must be declared before use, specifying their data type, such as int, float, char.

Whitespace and Indentation

Spaces, tabs, and newlines improve readability but do not affect program execution.

Example of Basic C Program Structure

```
#include <stdio.h>  // Preprocessor directive

int main() {       // main function start
   int number = 10; // variable declaration and initialization
   printf("Number is %d\n", number); // output statement
   return 0;      // indicate successful program termination
}              // main function end
```

Key Points

- Program execution always starts from main()
- Each statement ends with a semicolon.
- Use braces {} to group statements
- Comments helps explain code but are ignored by the compiler.
- Proper declaration of variables and functions is necessary.

Variables and Data Types

In C programming, variables are named memory locations used to store data. Data types specify the type and size of data a variable can hold. Choosing the correct data type ensures proper use of memory and valid operations.

Variables in C

Definition

A variable is an name assigned to a memory location where data is stored and can be modified during program execution.

Syntax

```
data_type variable_name;
```

Example

```
int age;
float salary;
char grade;
```

Initialization

A variable can be assigned a value at the time of declaration:

```
int age = 25;
```

Rules for Naming Variables

- Must begin with a letter (A-Z, a-z) or underscore (_)
- followed by letters, digits (0-9), or underscores
- Case-sensitive (Num ≠ Num)
- No special characters or spaces.
- Cannot use reserved keywords (e.g. int, float, return)

Types of Variables

- **Local Variables** – Declared inside functions or blocks; accessible only within them.
- **Global Variables** – Declared outside all functions; accessible throughout the program
- **Static Variables** – Retain their value between function calls.
- **Extern Variables** – Declared using extern; defined elsewhere.

Data Types in C

C provides several built-in data types to handle different kinds of data.

Primary Data Types

Data Type	Description	Size (in bytes)	Format Specifier
int	Integer numbers	2 or 4	%d
float	Decimal numbers (single)	4	%f

double	Decimal numbers (double)	8	%lf
char	Single character	1	%c

Derived Data Types

- **Arrays** – Collection of similar data types
- **Pointers** – Variables that store addresses
- **Structures** – Group of different data types
- **Unions** – Like structures, but with shared memory

Void Type

void represents "no type". It is used:

- For functions that return nothing
- For generic pointers (void *)

Example Program

```
#include <stdio.h>

int main() {
    int age = 25;
    float salary = 45000.50;
    char grade = 'A';

    printf("Age: %d\n", age);
    printf("Salary: %.2f\n", salary);
    printf("Grade: %c\n", grade);

    return 0;
}
```

Output

Age: 25
Salary: 45000.50
Grade: A

Note:

- Variables must be declared before use.
- Data types determine the kind of data a variable can hold.
- Correct data types helps in efficient memory usage and avoid type errors
- Variables scope (local or global) defines where the variables is accessible.

5 Operators in C

Operators are symbols that perform operations on variables and values. In C, program operators are used to manipulate data and perform calculations, comparisons, and logical operations. Understanding operators is fundamental for writing effective C programs.

Types of Operators

1. **Arithmetic Operators**
   Used to perform basic mathematical operations.
2. **Relational Operators**
   Used to compare two values or expressions.
3. **Logical Operators**
   Used to combine multiple conditions or invert a condition.
4. **Assignment Operators**
   Used to assign values to variables.
5. **Increment and Decrement Operators**
   Used to increase or decrease a value by one.
6. **Bitwise Operators**
   Operate at the bit level on integers.
7. **Conditional (Ternary) Operator**
   A shorthand for if-else statements.
8. **Other Operators**
   Include sizeof, comma, and pointer operators.

1. Arithmetic Operators

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b

Example:

```
int a = 10, b = 3;
printf("%d\n", a + b);  // 13
printf("%d\n", a % b);  // 1
```

2. Relational Operators

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater or equal	a >= b
<=	Less or equal	a <= b

Example:

```
int a = 5, b = 10;
if (a < b) {
   printf("a is less than b\n");
}
```

3. Logical Operators

Operator	Description	Example
&&	Logical AND	(a > 0 && b > 0)
!	Logical NOT	!(a > b)

Example:

```
int a = 5, b = 10;
if (a > 0 && b > 0) {
   printf("Both are positive\n");
}
```

4. Assignment Operators

Operator	Description	Example
=	Simple assignment	a = b
+=	Add and assign	a += b (a = a + b)
-=	Subtract and assign	a -= b
*=	Multiply and assign	a *= b
/=	Divide and assign	a /= b
%=	Modulus and assign	a %= b

Example:

```
int a = 5;
a += 3;  // a is now 8
```

5. Increment and Decrement Operators

Operator	Description	Example
++	Increment by 1	a++ or ++a
--	Decrement by 1	a-- or --a

- **Prefix (++a)**: Increments the value before use.
- **Postfix (a++)**: Uses the value first, then increments.

Example:

```
int a = 5;
printf("%d\n", ++a);  // 6 (prefix)
printf("%d\n", a++);  // 6 (postfix, prints then increments)
printf("%d\n", a);    // 7
```

6. Bitwise Operators

Operator	Description	Example
&	Bitwise AND	a & b
|		Bitwise OR
^	Bitwise XOR	a ^ b
~	Bitwise NOT	~a
<<	Left shift	a << 1
>>	Right shift	a >> 1

7. Conditional (Ternary) Operator

Syntax:

```
condition ? expression1 : expression2;
```

Evaluates the condition; if true, returns expression1; else returns expression2.

Example:

```
int a = 5, b = 10;
int max = (a > b) ? a : b;
printf("Max is %d\n", max);
```

8. Other Operators

- sizeof - Returns the size of a data type or variable in bytes.
- , (comma) - Allows multiple expressions where only one is expected.
- & - Address-of operator (gets variable's memory address).
- * - Pointer dereference operator.

6. **Decision Control Structures in C Programming Language**

Decision control structures are used to direct the flow of execution in a program based on conditions. They allow programs to make choices and execute different code paths depending on whether certain conditions are true or false. In C, decision-making is primarily handled through if, if-else, if-else-ladder, nested if, and switch statements.

1. The if Statement

Definition

- executes a block of code only if a specified condition is true

The if statement executes a block of code only if a specified condition is true. If the condition evaluates to false, the code block is skipped.

Syntax

```
if (condition) {
   // statements to execute if condition is true
}
```

Description

- The condition is evaluated first.
- If true, the code inside the block executes.
- If false, the program skips the block and continues.

Example

```
#include <stdio.h>

int main() {
   int num = 10;
   if (num > 0) {
      printf("Number is positive.\n");
   }
   return 0;
}
```

Output

Number is positive.

---

2. The if-else Statement

Definition

Provides an alternative path of execution if the condition is false.

Syntax

```
if (condition) {
   // statements if condition is true
} else {
   // statements if condition is false
}
```

Example

```
#include <stdio.h>

int main() {
   int num = -5;
   if (num >= 0) {
      printf("Number is non-negative.\n");
   } else {
      printf("Number is negative.\n");
   }
   return 0;
}
```

Output

Number is negative.

Description

- If the condition is true, the first block executes.
- If false, the else block executes.

---

3. The if-else if-else Ladder

Definition

This structure allows multiple conditions to be checked sequentially. The first true condition executes its block, and the rest are skipped.

Syntax

```
if (condition1) {
   // statements if condition1 is true
} else if (condition2) {
   // statements if condition2 is true
} else {
   // statements if all above conditions are false
}
```

Example

```
#include <stdio.h>

int main() {
   int marks = 75;
   if (marks >= 90) {
      printf("Grade A\n");
   } else if (marks >= 75) {
      printf("Grade B\n");
   } else if (marks >= 60) {
      printf("Grade C\n");
   } else {
      printf("Grade F\n");
   }
   return 0;
}
```

Output

Grade B

Description

- Conditions are evaluated from top to bottom.
- Execution stops at the first true condition.
- If none is true, the else block runs.

---

4. Nested if Statement

Definition

- Statement placed inside another.

An if or if-else statement placed inside another if or if-else block. useful for testing multiple conditions within conditions.

Syntax

```
if (condition1) {
   if (condition2) {
      // statements if both conditions are true
   }
}
```

Example

```
#include <stdio.h>

int main() {
   int num = 20;
   if (num > 0) {
      if (num % 2 == 0) {
         printf("Positive even number.\n");
      }
   }
   return 0;
}
```

Output

```
Positive even number.
```

---

5. The switch Statement

Definition

The switch statement is a multi-way decision construct that tests a variable against multiple values (called cases) and executes the matching case block.

Syntax

```
switch (expression) {
   case constant1:
      // statements
      break;
   case constant2:
      // statements
      break;
   ...
   default:
      // statements if no case matches
}
```

Example

```
#include <stdio.h>

int main() {
   int day = 3;
   switch (day) {
      case 1:
         printf("Monday\n");
         break;
      case 2:
         printf("Tuesday\n");
         break;
      case 3:
         printf("Wednesday\n");
         break;
      default:
         printf("Invalid day\n");
   }
   return 0;
}
```

Output

```
Wednesday
```

Description

- The expression is evaluated once.
- Execution jumps to the matching case label.
- break exits the switch to prevent fall-through.
- default executes if no case matches (optional).

Comparision:

Decision Structure	Use Case	Condition Evaluation	Execution Flow
if	Single condition	True → executes block	Skips block if false
if-else	Two alternative blocks	True or False	Executes either if or else block
if-else-if-else ladder	Multiple conditions sequential	First true only	Executes first matching block
Nested if	Multiple dependent conditions	Inner condition inside outer	Executes nested blocks if true
switch	Multi-way selection	Matches case constant	Executes matched case block

---

Scenario-Based Question Examples

1. Write a C program to check whether a student has passed or failed based on their marks using if-else statements.
   The program should accept input marks and print "Pass" if marks are 40 or above; otherwise print "Fail".

2. *Develop a menu-driven C program using the switch statement that performs arithmetic operations (+, -, , /) based on user choice.
   The program should prompt for two numbers and the operation choice, then output the result.

Detailed Example Programs

Example 1: Check Positive, Negative, or Zero Using if-else

```
#include <stdio.h>

int main() {
   int num;
   printf("Enter an integer: ");
   scanf("%d", &num);

   if (num > 0) {
      printf("Number is positive.\n");
   } else if (num < 0) {
      printf("Number is negative.\n");
   } else {
      printf("Number is zero.\n");
   }
   return 0;
}
```

Sample Output:

```
Enter an integer: -12
Number is negative.
```

Example 2: Simple Calculator Using switch

```
#include <stdio.h>

int main() {
   char op;
   float num1, num2;

   printf("Enter operator (+, -, *, /): ");
   scanf(" %c", &op);

   printf("Enter two operands: ");
   scanf("%f %f", &num1, &num2);

   switch (op) {
      case '+':
         printf("%.2f + %.2f = %.2f\n", num1,
num2, num1 + num2);
         break;
      case '-':
         printf("%.2f - %.2f = %.2f\n", num1,
num2, num1 - num2);
         break;
      case '*':
         printf("%.2f * %.2f = %.2f\n", num1,
num2, num1 * num2);
         break;
      case '/':
         if (num2 != 0.0)
            printf("%.2f / %.2f = %.2f\n", num1,
num2, num1 / num2);
         else
            printf("Division by zero error.\n");
         break;
      default:
         printf("Invalid operator.\n");
   }
   return 0;
}
```

Sample Output:

```
Enter operator (+, -, *, /): *
Enter two operands: 10 5
10.00 * 5.00 = 50.00
```

Loops in C Programming Language

Introduction

Loops are fundamental programming constructs that allow us to execute a block of code repeatedly based on a condition. Instead of writing repetitive code manually, loops help automate tasks that require iteration. In the C programming language, loops are used extensively in algorithms, data processing and handling repetitive tasks efficiently.

There are three main types of loops in C:

- for loop
- while loop
- do-while loop

Each has its specific use case depending on when and how the loop condition is evaluated.

1. For Loop

Definition – block of statements executes repeatedly for a fixed number of times

A for loop is used to execute a block of statements repeatedly for a fixed number of times. It is particularly useful when the number of iterations is known beforehand. The for loop integrates initialization, condition checking and update expression in a compact syntax.

Syntax

```
for (initialization; condition; update) {
   // statements to be executed repeatedly
}
```

- Initialization: Executed once at the start of the loop. Used to initialize a counter variable.
- Condition: Evaluated before each iteration. The loop continues if true; otherwise, it stops.
- Update: Executed after each iteration to update the counter.

Description

The for loop starts by executing the initialization part, then checks the condition. If the condition is true, the loop body executes, followed by the update step. This process repeats until the condition becomes false.

Example Code

```
#include <stdio.h>

int main() {
   int i;
   printf("For loop output:\n");
   for (i = 1; i <= 10; i++) {
      printf("%d ", i);
   }
   printf("\n");
   return 0;
}
```

Output

For loop output:
1 2 3 4 5 6 7 8 9 10

Advantages

- Compact and easy to manage when number of iterations is known.
- All loop control expressions are in one line making it easier to read.
- Suitable for counting loops and iterating over arrays.

---

2. While Loop

Definition

A while loop is a control flow statement that repeatedly executes a block of statements as long as the specified condition evaluates to true. The condition is checked before each iteration. so if the condition is false initially the loop body may never execute.

Syntax

```
while (condition) {
   // statements to be executed repeatedly
}
```

- The loop continues executing as long as the condition is true.
- The condition is evaluated before the execution of the loop body.

Description

The while loop is ideal when the number of iterations is not known beforehand and depends on dynamic conditions within the program.

```
#include <stdio.h>

int main() {
   int i = 1;
   printf("While loop output:\n");
   while (i <= 10) {
      printf("%d ", i);
      i++;
   }
   printf("\n");
   return 0;
}
```

Output

While loop output:
1 2 3 4 5 6 7 8 9 10

Advantages

- Useful when number of iterations depends on a condition that is evaluated dynamically.
- More flexible than the for loop for some scenarios.
- Can be used for indefinite loops until condition is met.

3. Do-While Loop

Definition

The DO-while loop is similar to the while loop but guarantees that the loop body executes at least once, because the condition is checked after the execution of the loop body.

Syntax

```
do {
   // statements to be executed repeatedly
} while (condition);
```

- The loop executes the body first.
- Then the condition is evaluated to decide if the loop should continue.

Description

The do-while loop is useful when you want the loop body to execute at least once, such as when prompting a user for input that must be validated.

Example Code

```
#include <stdio.h>

int main() {
   int i = 1;
   printf("Do-while loop output:\n");
   do {
      printf("%d ", i);
      i++;
   } while (i <= 10);
   printf("\n");
   return 0;
}
```

Output

Do-while loop output:
1 2 3 4 5 6 7 8 9 10

Advantages

- Guarantees the loop body runs at least once regardless of condition.
- Useful for menu-driven programs or input validation.

Nested Loops

Nested loops occurs when one loop is placed inside another loop. The inner loop completes all its iterations for every single iteration of the outer loop. Nested loops are useful for working with multi-dimensional data structures like Matrices or performing repetitive tasks within repetitive tasks.

Example: Nesting of for loops

To implement a C program to print half pyramid using '*'.

```
#include <stdio.h>

int main() {
   int rows, i, j;
   printf("Enter number of rows: ");
   scanf("%d", &rows);

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

   return 0;
}
```

Output:

```
Enter number of rows: 5
*
* *
* * *
* * * *
* * * * *
```

Similarly, we can define nested while loops also. For loop can be nested inside while loop and vice versa.

Comparison:

Loop Type	Condition Check	Executes At Least Once?	Best Use Case
For Loop	Before	No	Known number of iterations
While Loop	Before	No	Condition-based iteration
Do-While	After	Yes	Loop body must execute once

Loop Control Statements

1. Break Statement

The break statement allows immediate exit from a loop, regardless of the loop's condition. to stop the loop when a certain condition inside the loop is met

Example:

```
for (int i = 1; i <= 10; i++) {
   if (i == 5) {
      break;  // exits loop when i is 5
   }
   printf("%d ", i);
}
```

Output:

1 2 3 4

2. Continue Statement

The continue statement skips the current iteration of the loop and moves control back to the loop condition check for the next iteration.

Example:

```
for (int i = 1; i <= 10; i++) {
   if (i == 5) {
      continue; // skip printing 5
   }
   printf("%d ", i);
}
```

Output:

1 2 3 4 6 7 8 9 10

---

7. Arrays

An Array in C is a collection of elements of the same data type stored in contiguous memory locations. Arrays enable storage of multiple values under a single variable name, accessed using indices starting from zero. They are widely for efficient data Management and processing.

Types of Arrays

- **One-Dimensional Array:** Linear list of elements accessed by one index.
- **Two-Dimensional Array:** Matrix format accessed by two indices (row and column).
- **Multi-Dimensional Array:** Arrays with three or more dimensions, used for complex data structures.

One-Dimensional Array

Definition

A sequence of elements stored linearly.

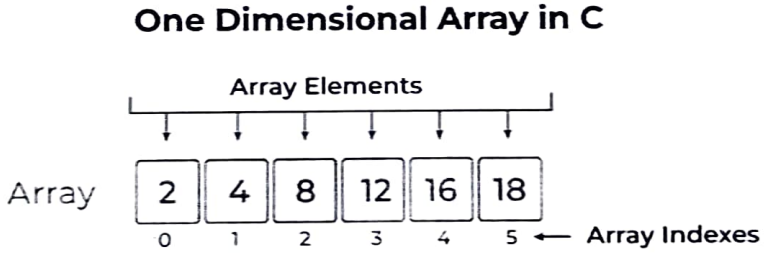

Fig:One Dimension array with 6 elements.

Syntax

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

```
#include <stdio.h>

int main() {
   int marks[3] = {85, 90, 78};

   // Accessing and printing each element using a loop
   for (int i = 0; i < 3; i++) {
      printf("%d ", marks[i]);
   }
   printf("\n"); // For new line after printing all elements

   return 0;
}
```

Output:

85 90 78

Two-Dimensional Array

Definition

A collection of elements arranged in rows and columns.

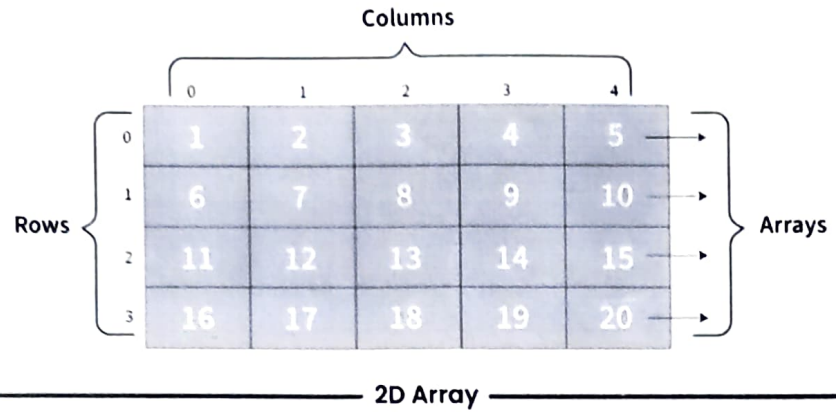

Fig:Two Dimension array with 4 rows and 5 columns.

Syntax

```
data_type array_name[rows][columns];
```

program for 2D array example including accessing and looping through the elements:

```
#include <stdio.h>

int main() {
   int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};

   // Loop to access and print elements of 2D array
   for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 3; j++) {
         printf("%d ", matrix[i][j]);
      }
      printf("\n"); // New line after each row
   }

   return 0;
}
```

Output:

```
1 2 3
4 5 6
```

Multi-Dimensional Array

Definition

An extension of 2D arrays with more than two indices, used for storing complex data like 3D Models or scientific data

Syntax

```
data_type array_name[d1][d2][d3]...;
```

Comparison of Array Types

Feature	One-Dimensional Array	Two-Dimensional Array	Multi-Dimensional Array
Structure	Linear list of elements	Matrix of rows and columns	Arrays with three or more dimensions
Access Method	Single index	Two indices (row and column)	Multiple indices corresponding to each dimension
Syntax Complexity	Simple	Moderate	More complex, multiple brackets
Use Cases	Lists, sequences	Tables, matrices	Scientific data, 3D graphics, simulations
Memory Layout	Contiguous block	Contiguous block in row-major order	Contiguous block in row-major order
Visualization	One-dimensional line	Two-dimensional grid	Multi-dimensional grid

Advantages of Arrays

- Storage Multiple values of the same type efficiently.
- Provide fast indexed access to elements.
- Useful in implementing algorithms and data structures.

Limitations of Arrays

- Fixed size, declared at compile time.
- All elements must be of the same data type.
- Lack of bounds checking may cause runtime errors.

---

8. **String**

A string in C is a sequence of characters stored in contiguous memory locations, terminated by a special null character '\0'. Strings are used to represent text data and are handled as arrays of characters.

String Declaration and Initialization

Syntax

```
char str[size];
```

Examples

```
char name[6] = {'H', 'e', 'l', 'l', 'o', '\0'};  // Explicit null character
char greeting[] = "Hello";  // Implicit null character added by compiler
```

Accessing and Modifying Strings

- Strings can be accessed using array indices, starting from zero.
- Each character in the string is stored in a separate array element.
- The null character '\0' indicates the end of the string.

Common Built-in String Functions in <string.h>

Function	Description
strlen()	Returns the length of the string (excluding '\0')
strcpy()	Copies one string into another
strcat()	Concatenates (joins) two strings
strcmp()	Compares two strings lexicographically
strncpy()	Copies a specified number of characters
strchr()	Finds first occurrence of a character
strstr()	Finds first occurrence of a substring
strrev()*	Reverses a string (non-standard, often custom implementation)

Example of String Functions

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

int main() {
    char str1[50] = "Hello";
    char str2[50] = "World";

    printf("Length of str1: %lu\n", strlen(str1)); // prints 5

    strcpy(str1, "Hi");
    printf("After strcpy, str1: %s\n", str1);  // prints Hi

    strcat(str1, str2);
    printf("After strcat, str1: %s\n", str1);  // prints HiWorld

    int cmp = strcmp(str1, str2);
    if (cmp == 0)
        printf("Strings are equal\n");
    else if (cmp > 0)
        printf("str1 is greater than str2\n");
    else
        printf("str1 is less than str2\n");

    return 0;
}
```

Output:

```
Length of str1: 5
After strcpy, str1: Hi
After strcat, str1: HiWorld
str1 is greater than str2
```

String Input and Output

- To input strings, scanf("%s", str); is commonly used but reads only until the first whitespace.
- To read strings with spaces, functions like fgets() are preferred.
- To print strings, use printf("%s", str);.

Example: Read and Print String

```
#include <stdio.h>

int main() {
   char name[50];
   printf("Enter your name: ");
   fgets(name, sizeof(name), stdin);   // reads a line
including spaces
   printf("Hello, %s", name);
   return 0;
}
```

Output: (Assuming input is *Alice Johnson*)

```
Enter your name: Alice Johnson
Hello, Alice Johnson
```

Additional Examples of Common String Functions

Using strncpy() — Copy Part of a String

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

int main() {
   char source[] = "Programming";
   char dest[20];

   strncpy(dest, source, 6);
   dest[6] = '\0';  // Manually add null terminator

   printf("Partial copy: %s\n", dest);
   return 0;
}
```

Output:

```
Partial copy: Progra
```

Using strchr() — Find First Occurrence of a Character

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

int main() {
   char str[] = "Hello World";
   char *ptr = strchr(str, 'o');

   if (ptr != NULL)
      printf("First 'o' found at position: %ld\n", ptr - str);
   else
      printf("'o' not found.\n");

   return 0;
}
```

Output:

```
First 'o' found at position: 4
```

Using strstr() — Find Substring

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

int main() {
   char str[] = "Hello World";
   char *substr = strstr(str, "World");

   if (substr != NULL)
      printf("Substring found: %s\n", substr);
   else
      printf("Substring not found.\n");

   return 0;
}
```

Output:

```
Substring found: World
```

Advantages of Strings

- Efficient way to store and manipulate textual data
- Supported by many standard library functions for ease of use.
- Flexible to represent any sequence of characters.

Limitations of Strings in C

- Strings are fixed-size arrays, buffer overflow can happen if not handled properly.
- Manual management of null character is necessary.
- No native string type; more prone to errors compared to higher-level languages.

Comparison: Strings vs Character Arrays

Feature	String	Character Array
Null Terminator	Ends with '\0'	May or may not be null-terminated
Usage	Represents text	Can store any characters
Library Support	Has many string handling functions	No direct support
Input/Output	Handled with %s, fgets()	Handled as array elements

String Operations Without Using Built-in Functions

1. String Length

```
#include <stdio.h>

int main() {
   char str[] = "Hello, World!";
   int length = 0;

   while (str[length] != '\0') {
      length++;
   }

   printf("Length of string: %d\n", length);
   return 0;
}
```

Output:

Length of string: 13

2. String Copy

```
#include <stdio.h>

int main() {
   char source[] = "Good morning";
   char destination[50];
   int i = 0;

   while (source[i] != '\0') {
      destination[i] = source[i];
      i++;
   }
   destination[i] = '\0';

   printf("Copied string: %s\n", destination);
   return 0;
}
```

Output:

Copied string: Good morning

3. String Comparison

```
#include <stdio.h>

int main() {
   char str1[] = "Apple";
   char str2[] = "Apples";
   int i = 0, result = 0;

   while (str1[i] != '\0' && str2[i] != '\0') {
      if (str1[i] != str2[i]) {
         result = str1[i] - str2[i];
         break;
      }
      i++;
   }

   if (result == 0) {
      result = str1[i] - str2[i];
   }

   if (result == 0)
      printf("Strings are equal.\n");
   else if (result < 0)
      printf("String 1 is less than String 2.\n");
   else
      printf("String 1 is greater than String 2.\n");

   return 0;
}
```

Output:

String 1 is less than String 2.

4. String Concatenation

```
#include <stdio.h>

int main() {
   char str1[50] = "Hello ";
   char str2[] = "World!";
   int i = 0, j = 0;

   while (str1[i] != '\0') {
      i++;
   }

   while (str2[j] != '\0') {
      str1[i] = str2[j];
      i++;
      j++;
   }
   str1[i] = '\0';

   printf("Concatenated string: %s\n", str1);
   return 0;
}
```

Output:

Concatenated string: Hello World!

Functions

A Function is a self-contained block of code designed to perform a specific task. Functions helps break down complex problems into smaller parts, Promote code reuse, improve readability, and make debugging easier. In C, every program has a main() function that acts as the entry point, and you can define additional functions as needed.

Structure of a Function

A function consists of:

- Function Declaration (prototype): Informs the compiler about the function names, return type, and parameters.
- Function Definition: Contains the code block that implements the function
- Function call: The place in the code where the function is invoked.

Syntax

```
return_type function_name(parameter_list) {
  // statements
}
```

Advantages of Using Functions

- Promote **code reusability**.
- Enhance **readability** and maintainability.
- Help with **modularity** and logical organization.
- Simplify **testing and debugging**.

Parameter Passing in C

Call by Value

- Passes a copy of the argument to the function.
- Changes inside the function do **not** affect the original variable.
- Safer for protecting data integrity.

Example:

```
#include <stdio.h>

void increment(int x) {
  x = x + 1;
  printf("Inside function: x = %d\n", x);
}

int main() {
  int num = 5;
  increment(num);
  printf("After function call: num = %d\n", num);
  return 0;
}
```

Output:

```
Inside function: x = 6
After function call: num = 5
```

Call by Reference (Using Pointers)

- Passes the address of the variable to the function.
- Changes inside the function affect the original variable.
- Useful when the function needs to modify the caller's data.

Example: Swapping two integers

```
#include <stdio.h>

void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  printf("Inside function: a = %d, b = %d\n", *a, *b);
}

int main() {
  int x = 10, y = 20;
  printf("Before swap: x = %d, y = %d\n", x, y);
  swap(&x, &y);
  printf("After swap: x = %d, y = %d\n", x, y);
  return 0;
}
```

Output:

```
Before swap: x = 10, y = 20
Inside function: a = 20, b = 10
After swap: x = 20, y = 10
```

Four Types of Functions Prototypes Based on Parameters and Return Type

Function Type	Description	Example Prototype
No parameters, no return value	Performs a task without input or output.	void greet(void);
With parameters, no return value	Takes inputs but returns nothing.	void printNumber(int num);
No parameters, with return value	Returns a value but takes no input.	int getRandomNumber(void);
With parameters, with return value	Takes inputs and returns a value.	int add(int a, int b);

Examples

Function without Parameters and without Return Value

```
#include <stdio.h>

void greet(void) {
   printf("Hello, welcome!\n");
}

int main() {
   greet();
   return 0;
}
```

Output:

Hello, welcome!

Function with Parameters and without Return Value

```
#include <stdio.h>

void printNumber(int num) {
   printf("Number: %d\n", num);
}

int main() {
   printNumber(100);
   return 0;
}
```

Output:

Number: 100

Function without Parameters and with Return Value

```
#include <stdio.h>

int getFive(void) {
   return 5;
}

int main() {
   int value = getFive();
   printf("Returned value: %d\n", value);
   return 0;
}
```

Output:

Returned value: 5

Function with Parameters and with Return Value

```
#include <stdio.h>

int add(int a, int b) {
   return a + b;
}

int main() {
   int sum = add(10, 20);
   printf("Sum = %d\n", sum);
   return 0;
}
```

Output:

Sum = 30

Recursive Functions in C Programming Language

A recursive function is a function that calls itself directly or indirectly to solve a problem by breaking it down into smaller, simpler subproblems. Recursion is useful for tasks that can be defined in terms of similar subtasks, such as mathematical computation, tree traversals, and searching algorithms. Every recursive function must have a base case to stop the recursion and avoid infinite calls.

Key Concepts

Base Case: The condition under which the recursion stops.

Recursive Case: The part where the function calls itself with modified parameters moving towards the base case.

Syntax

```
return_type function_name(parameters) {
   if (base_condition) {
      return base_value;  // Base case
   } else {
      return  function_name(modified_parameters);
// Recursive call
}
```

Advantages of Recursion

- Simplifies code for complex problems that involve repetitive subproblems.
- Easier to implement solutions for problems like factorial, Fibonacci series, tree traversals.
- Matches well with mathematical definitions.

Disadvantages of Recursion

- Can lead to high memory usage to call stack overhead.
- Less efficient than iterative solution for some problems.
- Risk of stack overflow if base case

is missing or not reached.

Example Programs

Factorial of a Number Using Recursion

Calculates the factorial of a positive integer n (n! = n * (n-1)!).

```
#include <stdio.h>

int factorial(int n) {
   if (n == 0)  // Base case
      return 1;
   else
      return n * factorial(n - 1);  // Recursive call
}

int main() {
   int num = 5;
   printf("Factorial of %d is %d\n", num, factorial(num));
   return 0;
}
```

Output:

Factorial of 5 is 120

Fibonacci Series Using Recursion

Computes the nth Fibonacci number where F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2).

```
#include <stdio.h>

int fibonacci(int n) {
   if (n == 0)  // Base case 1
      return 0;
   else if (n == 1)  // Base case 2
      return 1;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);  // Recursive calls
}

int main() {
   int terms = 7;
   printf("Fibonacci series up to %d terms:\n", terms);
   for (int i = 0; i < terms; i++) {
      printf("%d ", fibonacci(i));
   }
   printf("\n");
   return 0;
}
```

Output:

Fibonacci series up to 7 terms:
0 1 1 2 3 5 8

Sum of Natural Numbers Using Recursion

Finds the sum of first n natural numbers.

```
#include <stdio.h>

int sumNatural(int n) {
   if (n == 1)  // Base case
      return 1;
   else
      return n + sumNatural(n - 1);  // Recursive call
}

int main() {
   int num = 10;
   printf("Sum of first %d natural numbers is %d\n", num, sumNatural(num));
   return 0;
}
```

Output:

Sum of first 10 natural numbers is 55

When to Use Recursion

- Problems with **recursive structure**, e.g., divide and conquer algorithms.
- When a problem can be defined in terms of smaller instances of itself.
- Traversing hierarchical data like trees and graphs.

---

10. **Structures**

A structure in C is a user-defined data type that groups variables of different data types under one name. It allows organizing complex data, such as records for students, employees or products, into a single unit. Structures help in handling related information conveniently and logically.

Structure Definition and Declaration

Syntax to define a structure:

```
struct StructureName {
   data_type member1;
```

```
    data_type member2;
    // additional members
};
```

Declaring structure variables:

```
struct StructureName var1, var2;
```

Accessing Structure Members

Members of a structure variable are accessed using the **dot operator (.)**:

```
var1.member1 = value;
printf("%d", var1.member2);
```

Example: Employee structure defining and accessing

```
#include <stdio.h>

// Define Employee structure
struct Employee {
    int id;
    char name[50];
    float salary;
};

int main() {
    // Declare and initialize an Employee variable
    struct Employee emp1 = {201, "Vijay", 45000.50};

    // Access and print employee details
    printf("Employee ID: %d\n", emp1.id);
    printf("Employee Name: %s\n", emp1.name);
    printf("Employee Salary: %.2f\n", emp1.salary);

    return 0;
}
```

Output:

```
Employee ID: 201
Employee Name: Vijay
Employee Salary: 45000.50
```

Passing Structures to Functions

Structures can be passed to functions in two ways: Pass by value and Pass by reference

Pass by Value

- A **copy** of the structure is passed.
- Changes inside the function do not affect the original structure.
- Suitable when you only want to read data.

Syntax:

```
void functionName(struct StructureName param) {
    // access param.member1, param.member2
}
```

Example Program

Example 1: Pass Student Record by Value

```
#include <stdio.h>
// Define Student structure
struct Student {
    int id;
    char name[50];
    float marks;
};
// Function to display student information (pass by value)
void displayStudent(struct Student s) {
    printf("Students data is:\n");
    printf("ID: %d\n", s.id);
    printf("Name: %s\n", s.name);
    printf("Marks: %.2f\n", s.marks);
}
int main() {
    struct Student stu1;

    // Get data from user
    printf("Enter student ID: ");
    scanf("%d", &stu1.id);

    printf("Enter student name: ");
    scanf(" %[^\n]s", stu1.name);  // To read string with spaces

    printf("Enter student marks: ");
    scanf("%f", &stu1.marks);

    // Display student details
    displayStudent(stu1);

    return 0;
}
```

Output:

```
Enter student ID: 101
Enter student name: Aishwarya
Enter student marks: 87.5
Students data is:
ID: 101
Name: Aishwarya
Marks: 87.50
```

Note

* Use the dot operator (.) to access members of structure variables.
* Use the arrow Operator (->) to access members when using pointers
* Passing structures by reference is better for large data and when changes are needed.
* Structures can contain arrays and even other structures as members.