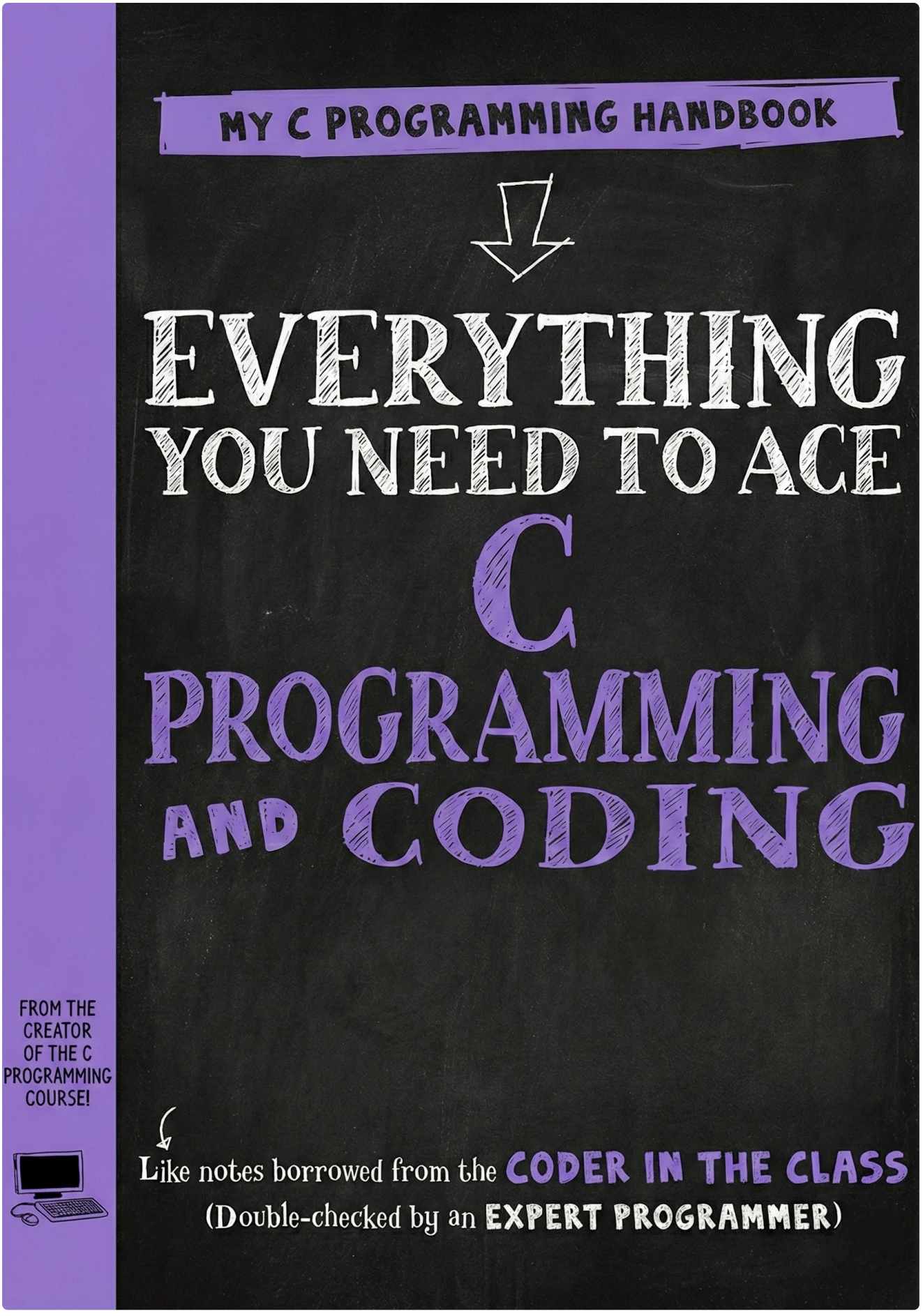
The Well Structured C



INDEX

Section 1: Introduction

Section 2: Variables and Data Types

Section 3: Operators

Section 4: Flow Control (Conditionals)

Section 5: Functions

Section 6: Loops

Section 7: Arrays

Section 8: Pointers

Section 9: Scope and Memory

Section 10: Dynamic Memory Allocation

Section 11: Strings

Section 12: Multidimensional Arrays

Section 13: Structures and Unions

Section 14: Advanced (Function Pointers & Files)

Section 15: Introduction to Recursion

Introduction to C

- **Topic Reference:** C Introduction 1

Definition:

C is a general-purpose, high-level programming language developed by Dennis Ritchie at Bell Labs in 1972. It is often called a "middle-level" language because it combines the features of high-level languages (easy to read) with the powerful low-level capabilities of assembly language (direct hardware manipulation).

Use Case:

It is widely used for developing Operating Systems (like Windows and UNIX), Embedded Systems, and system applications where speed and efficiency are critical.

2. Advantages of C

- **Topic Reference:** Why Do We Need Programming Languages 5, C Introduction 6

Key Points:

- **Efficiency:** C programs are highly efficient and run faster than many other languages because they compile directly to machine code.
- **Portability:** C code written on one machine can often run on another with little to no modification.
- **Modularity:** Complex programs can be broken down into manageable functions.
- **System Programming:** It allows direct manipulation of hardware (memory addresses via pointers).

3. Disadvantages of C

- **Topic Reference:** C Introduction 11

Key Points:

- **No Garbage Collection:** The programmer is responsible for memory management. If you allocate memory, you must free it, otherwise, it leads to memory leaks.
- **Lack of OOP:** C is a procedure-oriented language; it does not support Object-Oriented Programming concepts like Classes, Inheritance, or Polymorphism.
- **Runtime Checking:** It has weak runtime checking (e.g., it doesn't always stop you from accessing an array out of bounds), which can lead to unpredictable errors.

Structure Of C

1. Documentation Section

- **Topic Reference:** Comment in C 13

Definition:

This section consists of comments used to describe the program's purpose, the author, and the date. Comments are ignored by the compiler and are only for human readability.

Syntax:

- **Single-line comment:** `// Comment here`
- **Multi-line comment:** `/* Comment here */`

Use Case:

Used to explain complex logic so that you (or others) can understand the code later.

Example Program:

C

```c
/* Author: Student
   Purpose: Demonstrating Documentation
*/
#include <stdio.h>

int main() {
    // This prints a message
    printf("Documentation matters!");
    return 0;
}
```

Expected Output:

```
Documentation matters!
```

2. Linking Section (Preprocessor Directives)

- **Topic Reference:** First C Program 18

Definition:

This section involves including header files that contain definitions of functions needed by the program. The lines usually start with #.

Syntax:

`#include <filename.h>`

Use Case:

To use built-in functions like printf() (for output) and scanf() (for input), you must link the standard input-output library.

Example Program:

C

```c
#include <stdio.h> // Links the standard I/O library

int main() {
    printf("Library linked successfully.");
    return 0;
}
```

Expected Output:

```
Library linked successfully.
```

3. Common Header Files

- **Topic Reference:** First C Program 22

Definition:

Files ending with .h that contain function prototypes and macro definitions shared between source files.

Common Examples:

- `<stdio.h>`: Standard Input/Output (e.g., `printf`, `scanf`)
- `<math.h>`: Mathematical functions
- `<string.h>`: String manipulation (e.g., `strcpy`, `strlen`)
- `<stdlib.h>`: General utilities (e.g., `malloc`, `exit`)

Use Case:

Allows you to reuse code that has already been written and tested by others.

4. Definition Section

- **Topic Reference:** Const in C 27, Literals in C 28

Definition:

This section is used to define symbolic constants using the `#define` directive. These are values that remain constant throughout the program execution.

Syntax:

`#define NAME value`

Use Case:

Used for defining constants like PI or MAX_SIZE so that if the value changes, you only need to update it in one place.

Example Program:

C

```c
#include <stdio.h>
#define PI 3.14 // Definition Section

int main() {
    printf("Value of PI is: %f", PI);
    return 0;
}
```

Expected Output:

```
Value of PI is: 3.140000
```

5. Global Declaration Section

- **Topic Reference:** Global Variables and Scope 31

Definition:

Variables or functions declared outside of the main() function or any other function are called global variables.

Syntax:

C

```
datatype variable_name = value;
// defined outside functions
```

Use Case:

When a variable needs to be accessed by multiple functions in the program (not just main), it should be declared globally.

Example Program:

C

```
#include <stdio.h>

int globalVar = 100; // Global Declaration

void display() {
    printf("Inside Function: %d\n", globalVar);
}

int main() {
    printf("Inside Main: %d\n", globalVar);
    display();
    return 0;
}
```

Expected Output:

```
Inside Main: 100
Inside Function: 100
```

6. Main() Function

- **Topic Reference:** First C Program 34

Definition:

The main() function is the entry point of any C program. Execution of the program always begins here.

Syntax:

C

```
int main() {
    // Body of the function
    return 0;
}
```

Use Case:

Every executable C program must have exactly one main() function. The return 0 statement indicates that the program executed successfully.

Example Program:

C

```
#include <stdio.h>

int main() {
    printf("Program execution starts here.");
    return 0;
}
```

Expected Output:

```
Program execution starts here.
```

7. User Defined Functions

- **Topic Reference:** Functions in C 38, Function Declaration & Definition 39

Definition:

These are blocks of code written by the user to perform a specific task. They separate complex code into smaller, reusable chunks.

Syntax:

C

```
return_type function_name(parameters) {
    // code to be executed
}
```

Use Case:

Used to calculate factorials, perform math operations, or print patterns repeatedly without rewriting the logic every time.

Example Program:

C

```
#include <stdio.h>

// User Defined Function Declaration
int add(int a, int b) {
    return a + b;
}

int main() {
    int sum = add(5, 10); // Function Call
    printf("The sum is: %d", sum);
    return 0;
}
```

Expected Output:

```
The sum is: 15
```

Variables in C

- **Topic Reference:** Variables in C 1

Definition:

A variable is a named location in memory used to store a value that can be modified during program execution.

Syntax:

`data_type variable_name = value;`

Use Case:

Used to store data input by the user or results of computations.

Example Program:

C

```c
#include <stdio.h>
int main() {
    int score = 100; // Variable declaration and initialization
    printf("Score: %d", score);
    return 0;
}
```

Expected Output:

```
Score: 100
```

Variable Naming Rules

- **Topic Reference:** Variable Naming Rules 2

Definition:

A set of rules that must be followed when creating identifiers (names) for variables to avoid compilation errors.

Rules:

- Must start with a letter (a-z, A-Z) or underscore (`_`).
- Cannot start with a digit (0-9).
- Can contain letters, digits, and underscores.
- Keywords (e.g., `int`, `float`, `if`) are not allowed.
- No whitespace is allowed.
- Case sensitive (`sum` and `Sum` are different).

Example Program:

C

```c
#include <stdio.h>
int main() {
    int valid_name = 10;
    int _valid2 = 20;
    // int 1invalid = 30; // Error
    printf("%d, %d", valid_name, _valid2);
    return 0;
}
```

Expected Output:

```
10, 20
```

Data Types in C

- **Topic Reference:** Data types in C 3

Definition:

Data types specify the type of data a variable can hold. This determines the size of memory allocated and the operations that can be performed.

Common Types:

- **int**: Integers (e.g., 1, -10, 500).
- **float**: Floating-point numbers (e.g., 3.14, -0.01).
- **char**: Single characters (e.g., 'a', 'Z').
- **double**: Double-precision floating-point numbers (more precise than float).

Example Program:

C

```c
#include <stdio.h>
int main() {
    int i = 10;
    float f = 5.5;
    char c = 'A';
    printf("Int: %d, Float: %.1f, Char: %c", i, f, c);
    return 0;
}
```

Expected Output:

```
Int: 10, Float: 5.5, Char: A
```

Ranges of Data Types in C

- **Topic Reference:** Ranges of Data Types in C 4

Definition:

The range refers to the minimum and maximum values a data type can hold. These values depend on the system architecture (32-bit vs 64-bit).

Typical Ranges (32-bit compiler):

- **char**: -128 to 127
- **int**: -2,147,483,648 to 2,147,483,647
- **unsigned int**: 0 to 4,294,967,295

Use Case:

Knowing ranges prevents overflow errors (e.g., trying to store 300 in a char variable).

Example Program:

C

```c
#include <stdio.h>
#include <limits.h> // Header for integer limits

int main() {
    printf("Max Int: %d\n", INT_MAX);
    printf("Min Int: %d", INT_MIN);
    return 0;
}
```

Expected Output:

```
Max Int: 2147483647
Min Int: -2147483648
```

Operator size of() in C

- **Topic Reference:** Operator size of() in C 5

Definition:

sizeof() is a compile-time unary operator used to calculate the size (in bytes) of a data type or variable.

Syntax:

`sizeof(type)` or `sizeof(variable)`

Use Case:

Critical for dynamic memory allocation and understanding memory usage on different platforms.

Example Program:

C

```c
#include <stdio.h>
int main() {
    int a;
    double b;
    printf("Size of int: %lu bytes\n", sizeof(a));
    printf("Size of double: %lu bytes", sizeof(b));
    return 0;
}
```

Expected Output:

```
Size of int: 4 bytes
Size of double: 8 bytes
```

Global Variables and Scope

- **Topic Reference:** Global Variables and Scope 6

Definition:

- **Scope**: The region of code where a variable is visible.
- **Global Variable**: Declared outside all functions; accessible everywhere.
- **Local Variable**: Declared inside a function; accessible only there.

Example Program:

C

```c
#include <stdio.h>
int global = 100; // Global variable

void display() {
    printf("Global: %d\n", global);
}

int main() {
    int local = 5; // Local variable
    printf("Local: %d\n", local);
    display();
    return 0;
}
```

Expected Output:

```
Local: 5
Global: 100
```

Const in C

- **Topic Reference:** Const in C 7

Definition:

The const keyword defines a variable whose value cannot be changed after initialization. It effectively makes the variable read-only.

Syntax:

`const data_type variable_name = value;`

Use Case:

Used for values that should remain constant, like mathematical constants (PI) or configuration limits.

Example Program:

C

```c
#include <stdio.h>
int main() {
    const int MAX = 10;
    // MAX = 20; // This line would cause a Compilation Error
    printf("Max value is %d", MAX);
    return 0;
}
```

Expected Output:

```
Max value is 10
```

Static Variables in C

- **Topic Reference:** Static Variables in C 8

Definition:

A static variable preserves its value between function calls. It is initialized only once and exists for the lifetime of the program.

Syntax:

`static data_type variable_name = value;`

Example Program:

C

```c
#include <stdio.h>
void count_calls() {
    static int count = 0; // Initialized once
    count++;
    printf("Call %d\n", count);
}

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

Expected Output:

C

```
Call 1
Call 2
```

Literals in C

- **Topic Reference:** Literals in C 9

Definition:

Literals are fixed values (constants) assigned to variables. They do not change during execution.

Types:

- **Integer Literal:** `10`, `-5`
- **Float Literal:** `3.14`, `0.5`
- **Char Literal:** `'A'`, `'%'`
- **String Literal:** `"Hello"`

Example Program:

C

```c
#include <stdio.h>
int main() {
    int num = 50;        // 50 is an integer literal
    char letter = 'X';   // 'X' is a char literal
    printf("%d %c", num, letter);
    return 0;
}
```

Expected Output:

```
50 X
```

Type Conversion in C

- **Topic Reference:** Type Conversion in C 10

Definition:

The process of converting one data type into another.

- **Implicit (Automatic)**: Done by the compiler (e.g., `int` to `float`).
- **Explicit (Type Casting)**: Done by the programmer.

Syntax (Explicit):

`(type_name) expression;`

Example Program:

C

```c
#include <stdio.h>
int main() {
    int a = 5, b = 2;
    float result;

    // Explicit conversion to get decimal result
    result = (float)a / b;

    printf("Result: %.1f", result);
    return 0;
}
```

Expected Output:

```
Result: 2.5
```

Swap Two Numbers

- **Topic Reference:** Swap Two Numbers 11

Definition:

A standard programming logic to exchange the values of two variables. This is commonly done using a third temporary variable.

Algorithm:

1. Store value of A in Temp.
2. Store value of B in A.
3. Store value of Temp in B.

Example Program:

C

```c
#include <stdio.h>
int main() {
    int a = 10, b = 20, temp;

    // Logic to swap
    temp = a;
    a = b;
    b = temp;

    printf("After Swap: a = %d, b = %d", a, b);
    return 0;
}
```

Expected Output:

```
After Swap: a = 20, b = 10
```

Operator

- **Topic Reference:** Operator 1

Definition:

An operator is a special symbol that tells the compiler to perform specific mathematical or logical manipulations on operands (variables or values).

Use Case:

Operators are the machinery of a program. They allow you to calculate values, compare data, and make decisions.

Example Program:

C

```c
#include <stdio.h>
int main() {
    int a = 10, b = 5;
    int sum = a + b; // '+' is the operator
    printf("Sum: %d", sum);
    return 0;
}
```

Expected Output:

Plaintext

```
Sum: 15
```

Arithmetic Operators

- **Topic Reference:** Arithmetic Operators 2

Definition:

These operators perform standard mathematical operations.

Syntax & Types:

- `+` (Addition)
- `-` (Subtraction)
- `*` (Multiplication)
- `/` (Division): Returns the quotient. **Note:** Integer division truncates the decimal part (e.g., `5/2` results in `2`, not `2.5`).
- `%` (Modulus): Returns the remainder (only works with integers).

Use Case:

Used for all basic calculations, counting, and physics simulations.

Example Program:

C

```c
#include <stdio.h>
int main() {
    int a = 11, b = 3;
    printf("Addition: %d\n", a + b);
    printf("Division (Int): %d\n", a / b);
    printf("Modulus (Remainder): %d\n", a % b);
    return 0;
}
```

Expected Output:

Plaintext

```
Addition: 14
Division (Int): 3
Modulus (Remainder): 2
```

Unary Arithmetic Operators

- **Topic Reference:** Unary Arithmetic Operators 3

Definition:

Operators that require only one operand. The most common are Increment (++) and Decrement (--).

Syntax:

- **Pre-increment (`++a`):** Value is increased *immediately*, then the new value is used.
- **Post-increment (`a++`):** Current value is *used first*, then it is increased.

Use Case:

Widely used in loops (for, while) to update counters.

Example Program:

C

```c
#include <stdio.h>
int main() {
    int x = 5;
    printf("Original: %d\n", x);
    printf("Post-increment (x++): %d\n", x++); // Prints 5, then x becomes 6
    printf("After Post-increment: %d\n", x);    // Prints 6
    printf("Pre-increment (++x): %d\n", ++x);   // x becomes 7, then prints
7
    return 0;
}
```

Expected Output:

Plaintext

```
Original: 5
Post-increment (x++): 5
After Post-increment: 6
Pre-increment (++x): 7
```

Comparison Operators

- **Topic Reference:** Comparison Operators 4

Definition:

These operators (also called Relational Operators) compare two values. They return 1 (true) if the comparison is correct and 0 (false) if it is incorrect.

Syntax:

- `==` (Equal to)
- `!=` (Not equal to)
- `>` (Greater than)
- `<` (Less than)
- `>=` (Greater than or equal to)
- `<=` (Less than or equal to)

Use Case:

Essential for decision-making (If/Else statements).

Example Program:

C

```c
#include <stdio.h>
int main() {
    int a = 10, b = 20;
    printf("Is a equal to b? %d\n", (a == b)); // 0 (False)
    printf("Is b greater than a? %d\n", (b > a)); // 1 (True)
    return 0;
}
```

Expected Output:

Plaintext

```
Is a equal to b? 0
Is b greater than a? 1
```

Assignment Operators

- **Topic Reference:** Assignment Operators 5

Definition:

Used to assign a value to a variable. The value on the right is stored in the variable on the left. C also supports "Compound Assignment" operators.

Syntax:

- `=` (Assign)
- `+=` (Add and assign: `a += b` is shorthand for `a = a + b`)
- `-=` (Subtract and assign)
- `*=` (Multiply and assign)

Use Case:

Updating a variable's value based on its current value (e.g., adding to a running total).

Example Program:

C

```c
#include <stdio.h>
int main() {
    int score = 10;
    score += 5; // Same as score = score + 5
    printf("Score after +=: %d", score);
    return 0;
}
```

Expected Output:

Plaintext

```
Score after +=: 15
```

Logical Operators

- **Topic Reference:** Logical Operators 6

Definition:

Used to combine multiple conditions. They return either 0 (False) or 1 (True).

Syntax:

- `&&` (**Logical AND**): True only if **both** operands are true.
- `||` (**Logical OR**): True if **at least one** operand is true.
- `!` (**Logical NOT**): Reverses the state (True becomes False).

Use Case:

Used when validating complex rules (e.g., "Age > 18 AND HasLicense").

Example Program:

C

```c
#include <stdio.h>
int main() {
    int age = 25;
    int hasID = 1; // 1 represents True

    if (age > 18 && hasID) {
        printf("Allowed to enter.");
    } else {
        printf("Not allowed.");
    }
    return 0;
}
```

Expected Output:

Plaintext

```
Allowed to enter.
```

Bitwise Operator in C (AND, OR and XOR)

- **Topic Reference:** Bitwise Operator in C (AND, OR and XOR) 7

Definition:

These operators perform operations on data at the bit level (binary 0s and 1s).

Syntax:

- `&` (**Bitwise AND**): Result bit is 1 if *both* bits are 1.
- `|` (**Bitwise OR**): Result bit is 1 if *at least one* bit is 1.
- `^` (**Bitwise XOR**): Result bit is 1 if bits are *different*.

Use Case:

Used in low-level programming (drivers, graphics, cryptography) to set or mask specific flags.

Example Program:

C

```c
#include <stdio.h>
int main() {
    // 5 in binary: 0101
    // 3 in binary: 0011
    int a = 5, b = 3;

    printf("a & b: %d\n", a & b); // 0101 & 0011 = 0001 (1)
    printf("a | b: %d\n", a | b); // 0101 | 0011 = 0111 (7)
    printf("a ^ b: %d\n", a ^ b); // 0101 ^ 0011 = 0110 (6)
    return 0;
}
```

Expected Output:

Plaintext

```
a & b: 1
a | b: 7
a ^ b: 6
```

Bitwise Operator in C (Left Shift, Right Shift and NOT)

- **Topic Reference:** Bitwise Operator in C (Left Shift, Right Shift and NOT) 8

Definition:

Operators that move bits to the left or right, or invert them.

Syntax:

- `<<` (**Left Shift**): Shifts bits left. `x << n` effectively multiplies `x` by 2^n.
- `>>` (**Right Shift**): Shifts bits right. `x >> n` effectively divides `x` by 2^n.
- `~` (**Bitwise NOT**): Inverts all bits (0 becomes 1, 1 becomes 0).

Use Case:

Very fast multiplication or division by powers of 2.

Example Program:

C

```c
#include <stdio.h>
int main() {
    int x = 5; // Binary: 0000...0101
    printf("Left Shift (x << 1): %d\n", x << 1); // 10 (5 * 2)
    printf("Right Shift (x >> 1): %d\n", x >> 1); // 2 (5 / 2)
    return 0;
}
```

Expected Output:

Plaintext

```
Left Shift (x << 1): 10
Right Shift (x >> 1): 2
```

Signed Number Representation and Bitwise NOT

- **Topic Reference:** Signed Number Representation and Bitwise NOT 9

Definition:

In C, signed numbers are stored using 2's Complement format. Because of this, the Bitwise NOT operator (~) behaves according to the formula:

~x = -(x + 1)

Use Case:

Understanding how negative numbers are stored in memory and manipulating bits in signed integers.

Example Program:

C

```c
#include <stdio.h>
int main() {
    int x = 5;
    // ~5 becomes -(5 + 1) = -6
    printf("Bitwise NOT (~x): %d", ~x);
    return 0;
}
```

Expected Output:

Plaintext

```
Bitwise NOT (~x): -6
```

Operator Precedence & Associativity

- **Topic Reference:** Operator Precedence & Associativity 10

Definition:

- **Precedence:** Determines which operator is evaluated first (e.g., `*` happens before `+`).
- **Associativity:** Determines the direction of evaluation if operators have the same precedence (usually Left-to-Right).

Key Order (High to Low):

1. `()` (Parentheses)
2. `++`, `--`, `!` (Unary)
3. `*`, `/`, `%` (Multiplicative)
4. `+`, `-` (Additive)
5. `=`, `+=` (Assignment - Right to Left)

Example Program:

C

```c
#include <stdio.h>
int main() {
    int result = 10 + 20 * 5; // Multiplication (*) has higher precedence
than (+)
    printf("Result: %d", result); // 10 + 100 = 110
    return 0;
}
```

Expected Output:

Plaintext

```
Result: 110
```

Day Before N Day

- **Topic Reference:** Day Before N Day 11

Problem Statement:

Given a current day (0=Sun, 1=Mon... 6=Sat) and a number N, find what day it was N days ago.

Logic:

To handle negative results from subtraction (e.g., 0 - 1 = -1, which isn't a valid index), we use modular arithmetic:

result = (current_day - N % 7 + 7) % 7

Example Program:

C

```c
#include <stdio.h>
int main() {
    int d = 0; // Sunday
    int n = 1; // 1 day ago

    // Adding 7 ensures the value is positive before modulus
    int ans = (d - n % 7 + 7) % 7;

    printf("Day index: %d", ans); // Expect 6 (Saturday)
    return 0;
}
```

Expected Output:

Plaintext

```
Day index: 6
```

Sum of Natural Numbers

- **Topic Reference:** Sum of Natural Numbers 12

Problem Statement:

Calculate the sum of the first N natural numbers.

Logic:

Formula: $Sum = \frac{N \times (N+1)}{2}$

Example Program:

C

```c
#include <stdio.h>
int main() {
    int n = 5; // Sum of 1, 2, 3, 4, 5
    int sum = n * (n + 1) / 2;
    printf("Sum of first %d numbers: %d", n, sum);
    return 0;
}
```

Expected Output:

Plaintext

```
Sum of first 5 numbers: 15
```

Last Digit of a Number

- **Topic Reference:** Last Digit of a Number 13

Problem Statement:

Find the last digit of any given integer.

Logic:

Using the Modulus operator % 10 always yields the last digit of an integer.

Example Program:

C

```c
#include <stdio.h>

int main() {
    int num = 12345;
    int lastDigit = num % 10;
    printf("Last Digit: %d", lastDigit);
    return 0;
}
```

Expected Output:

Plaintext

```
Last Digit: 5
```

Practice Problem on Operators

- **Topic Reference:** Practice Problem on Operators 14

Problem Statement:

Predict the output of the following C code snippet involving various operators.

Code:

C

```c
#include <stdio.h>
int main() {
    int x = 10, y = 5;

    // Evaluation:
    // 1. (x > y) is True (1).
    // 2. (x++ > 10): x is currently 10. 10 > 10 is False (0).
    // 3. 1 && 0 is 0.
    // 4. x increments to 11 AFTER the (x++ > 10) check.
    int z = (x > y) && (x++ > 10);

    printf("z: %d, x: %d", z, x);
    return 0;
}
```

Expected Output:

Plaintext

```
z: 0, x: 11
```

Flow Control in C

Definition:

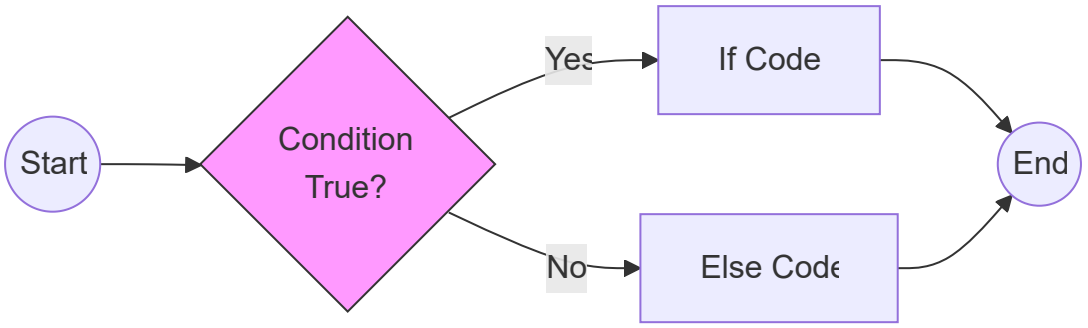
Flow control statements define the order in which statements are executed in a program. They allow the program to make decisions (selection), repeat operations (loops), or jump to different sections of code.

Use Case:

Used to create dynamic programs that can react differently based on user input or calculated values, rather than just running top-to-bottom.

Types:

1. **Selection Statements:** `if`, `if-else`, `switch`
2. **Iteration Statements (Loops):** `for`, `while`, `do-while`
3. **Jump Statements:** `break`, `continue`, `goto`



If Else in C

Definition:

The if-else statement is used to perform two operations for a single condition. The if block is executed if the condition is true, and the else block is executed if the condition is false.

Syntax:

C

```
if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
```

Use Case:

Used for binary decisions, such as checking if a password is correct (Access Granted vs. Access Denied).

Example Program:

C

```
#include <stdio.h>

int main() {
    int age = 20;

    if (age >= 18) {
        printf("You are eligible to vote.");
    } else {
        printf("You are not eligible to vote.");
    }
    return 0;
}
```

Expected Output:

```
You are eligible to vote.
```

IF Else Example in C (Practical)

Use Case:

A practical scenario demonstrating how if-else handles user input validation or state checking.

Example Program (Password Check):

C

```c
#include <stdio.h>

int main() {
    int stored_pin = 1234;
    int input_pin;

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

    if (input_pin == stored_pin) {
        printf("Login Successful!");
    } else {
        printf("Incorrect PIN. Try again.");
    }
    return 0;
}
```

Expected Output (Assumed Input: 1234):

```
Enter PIN: 1234
Login Successful!
```

Nested If Else with Example

Definition:

When an if or else statement is placed inside another if or else block, it is called a nested if-else.

Syntax:

C

```
if (condition1) {
    if (condition2) {
        // Executes if both condition1 and condition2 are true
    } else {
        // Executes if condition1 is true but condition2 is false
    }
} else {
    // Executes if condition1 is false
}
```

Use Case:

Used when a decision depends on multiple layered conditions (e.g., Is the user logged in? If yes, are they an admin?).

Example Program:

C

```
#include <stdio.h>

int main() {
    int num = 10;

    if (num > 0) {
        if (num % 2 == 0) {
            printf("Positive Even Number");
        } else {
            printf("Positive Odd Number");
        }
    } else {
        printf("The number is not positive");
    }
```

```
    return 0;
}
```

Expected Output:

```
Positive Even Number
```

Else if Ladder in C

Definition:

The else-if ladder is used when there are multiple conditions to check. The compiler checks conditions from the top down; as soon as one true condition is found, that block is executed, and the rest are skipped.

Syntax:

C

```
if (condition1) {
    // code
} else if (condition2) {
    // code
} else if (condition3) {
    // code
} else {
    // default code if no conditions are true
}
```

Use Case:

Commonly used for grading systems, menu selections, or categorizing values (e.g., Small, Medium, Large).

Example Program:

C

```
#include <stdio.h>

int main() {
    int marks = 85;

    if (marks >= 90) {
        printf("Grade: A");
    } else if (marks >= 80) {
        printf("Grade: B");
    } else if (marks >= 70) {
        printf("Grade: C");
    } else {
        printf("Grade: F");
```

```
    }
    return 0;
}
```

Expected Output:

```
Grade: B
```

Switch in C

Definition:

The switch statement executes one code block from multiple alternatives based on the value of a single variable. It is often cleaner than a long else-if ladder.

Syntax:

C

```c
switch (expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // code if no case matches
}
```

Use Case:

Ideal for menu-driven programs where the user selects an option (1, 2, 3...) or checks a specific character input.

Example Program:

C

```c
#include <stdio.h>

int main() {
    int day = 3;

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

Expected Output:

```
Wednesday
```

Program: Even Odd

Definition:

A program to determine if a number is divisible by 2.

Logic:

If number % 2 == 0, the number is Even; otherwise, it is Odd.

Example Program:

C

```c
#include <stdio.h>

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

    if (num % 2 == 0) {
        printf("%d is Even.", num);
    } else {
        printf("%d is Odd.", num);
    }
    return 0;
}
```

Expected Output (Assumed Input: 7):

```
Enter a number: 7
7 is Odd.
```

Program: Largest of Three Numbers

Definition:

A program to compare three distinct integers and find the maximum value.

Logic:

Use comparison operators (>) combined with logical AND (&&) to check if one number is greater than the other two.

Example Program:

C

```c
#include <stdio.h>

int main() {
    int a = 10, b = 25, c = 15;

    if (a >= b && a >= c) {
        printf("%d is the largest.", a);
    } else if (b >= a && b >= c) {
        printf("%d is the largest.", b);
    } else {
        printf("%d is the largest.", c);
    }
    return 0;
}
```

Expected Output:

```
25 is the largest.
```

Program: Leap Year

Definition:

A program to check if a year has 366 days instead of 365.

Logic:

A year is a leap year if:

1. It is divisible by 400.
2. OR (It is divisible by 4 AND NOT divisible by 100).

Example Program:

C

```c
#include <stdio.h>

int main() {
    int year = 2024;

    if ((year % 400 == 0) || (year % 4 == 0 && year % 100 != 0)) {
        printf("%d is a Leap Year.", year);
    } else {
        printf("%d is not a Leap Year.", year);
    }
    return 0;
}
```

Expected Output:

```
2024 is a Leap Year.
```

Program: Simple Calculator

Definition:

A program that takes two numbers and an operator (+, -, *, /) from the user and performs the corresponding arithmetic operation using a switch statement.

Example Program:

C

```c
#include <stdio.h>

int main() {
    char operator;
    double n1, n2;

    printf("Enter operator (+, -, *, /): ");
    scanf("%c", &operator);
    printf("Enter two operands: ");
    scanf("%lf %lf", &n1, &n2);

    switch (operator) {
        case '+':
            printf("%.1lf + %.1lf = %.1lf", n1, n2, n1 + n2);
            break;
        case '-':
            printf("%.1lf - %.1lf = %.1lf", n1, n2, n1 - n2);
            break;
        case '*':
            printf("%.1lf * %.1lf = %.1lf", n1, n2, n1 * n2);
            break;
        case '/':
            if(n2 != 0)
                printf("%.1lf / %.1lf = %.1lf", n1, n2, n1 / n2);
            else
                printf("Error! Division by zero.");
            break;
        default:
            printf("Error! Operator is not correct");
    }
```

```
    return 0;
}
```

Expected Output (Assumed Input: +, 5, 3):123

```
Enter operator (+, -, *, /): +
Enter two operands: 5 3
5.0 + 3.0 = 8.0
```

Functions in C

Definition:

A function is a self-contained block of statements that performs a specific task. In C, execution always starts from a special function called main(). Functions allow you to break down a large program into smaller, manageable pieces.

Syntax:

C

```
return_type function_name(parameter_list) {
    // Body of the function
    return value;
}
```

Use Case:

Used to execute a specific logic (like calculating a sum or printing a message) whenever needed, without rewriting the code.

Example Program:

C

```
#include <stdio.h>

// Simple function to print a message
void greet() {
    printf("Hello, Welcome to C Programming!\n");
}

int main() {
    greet(); // Calling the function
    return 0;
}
```

Expected Output:

```
Hello, Welcome to C Programming!
```

Applications of Functions

Definition:

The primary reasons why functions are essential in programming.

Key Applications:

1. **Reusability:** Write code once and use it multiple times by calling the function.
2. **Modularity:** Breaks a complex problem into smaller, easier-to-understand sub-problems.
3. **Abstraction:** Hides implementation details; the user only needs to know the function name and parameters to use it.
4. **Easy Debugging:** If an error occurs, you only need to check the specific function rather than the entire program.

Example Scenario:

If you need to calculate the area of circles with different radii multiple times, you write one calculateArea() function and call it for every radius.

Function Declaration & Definition

Definition:

- **Declaration (Prototype):** Tells the compiler about the function's name, return type, and parameters *before* it is defined. This allows the function to be called before its actual code appears.
- **Definition:** The actual body of the function containing the code to be executed.

Syntax:

C

```
// Declaration
return_type function_name(type arg1, type arg2);

// Definition
return_type function_name(type arg1, type arg2) {
    // code
}
```

Example Program:

C

```
#include <stdio.h>

// 1. Function Declaration (Prototype)
int add(int, int);

int main() {
    int result = add(10, 20); // Function Call
    printf("Sum: %d", result);
    return 0;
}

// 2. Function Definition
int add(int a, int b) {
    return a + b;
}
```

Expected Output:

```
Sum: 30
```

How Functions Work in C

Definition:

When a function is called, the control of the program transfers from the caller (e.g., main) to the callee (the function).

1. Arguments are passed to the function parameters.
2. Memory is allocated for the function's local variables in the stack.
3. The function executes its code.
4. A return value (if any) is sent back to the caller.
5. Memory allocated for the function is destroyed, and control returns to the caller.

Example Program:

C

```c
#include <stdio.h>

void insideFunction() {
    printf("  -> Inside the function now.\n");
}

int main() {
    printf("1. Inside Main.\n");
    insideFunction(); // Control jumps to insideFunction
    printf("2. Back in Main.\n");
    return 0;
}
```

Expected Output:

```
1. Inside Main.
  -> Inside the function now.
2. Back in Main.
```

Inline Function

Definition:

An inline function is a suggestion to the compiler to insert the function's code directly into the calling place, rather than performing a standard function call (which involves overhead like jumping to a memory address).

Syntax:

C

```
inline return_type function_name(parameters) {
    // code
}
```

Use Case:

Used for small, frequently called functions to improve performance by eliminating function-call overhead.

Example Program:

C

```
#include <stdio.h>

// Inline function definition
static inline int square(int x) {
    return x * x;
}

int main() {
    int num = 5;
    // Compiler replaces this call with "5 * 5" directly
    printf("Square of %d is %d", num, square(num));
    return 0;
}
```

Expected Output:

```
Square of 5 is 25
```

Practice Problem: First Digit of a Number

Definition:

A program to extract the very first digit of a given positive integer.

Logic:

Keep dividing the number by 10 until it becomes less than 10. The remaining value is the first digit.

Example Program:

C

```c
#include <stdio.h>

int getFirstDigit(int n) {
    // Loop until n is a single digit
    while (n >= 10) {
        n = n / 10;
    }
    return n;
}

int main() {
    int number = 98765;
    int first = getFirstDigit(number);
    printf("The first digit of %d is: %d", number, first);
    return 0;
}
```

Expected Output:

```
The first digit of 98765 is: 9
```

Practice Problem: Prime Factorization

Definition:

A program to find all prime numbers that multiply together to make the original number.

Logic:

1. Start a loop from $i = 2$.
2. While the number `n` is divisible by `i`, print `i` and divide `n` by `i`.
3. Increment `i` and repeat until `n` becomes 1.

Example Program:

C

```c
#include <stdio.h>

void primeFactors(int n) {
    printf("Prime factors of %d: ", n);

    // Divide by 2 until it's odd
    while (n % 2 == 0) {
        printf("%d ", 2);
        n = n / 2;
    }

    // Check for odd factors from 3 onwards
    for (int i = 3; i * i <= n; i = i + 2) {
        while (n % i == 0) {
            printf("%d ", i);
            n = n / i;
        }
    }

    // If n is a prime number greater than 2
    if (n > 2) {
        printf("%d", n);
    }
}

int main() {
    int number = 315;
    primeFactors(number);
```

```
    return 0;
}
```

Expected Output:

```
Prime factors of 315: 3 3 5 7
```

C Programming: Loops & Control Flow

While Loop

Type: Entry-Controlled Loop

The condition is checked before the code executes. If the condition is false initially, the code never runs.

Syntax:

C

```
while (condition) {
    // code to be executed
    // update statement
}
```

Example:

C

```
#include <stdio.h>

int main() {
    int i = 1;

    while (i <= 3) {
        printf("Iteration %d\n", i);
        i++;
    }
    return 0;
}
```

Output:

Plaintext

```
Iteration 1
Iteration 2
Iteration 3
```

For Loop

Type: Entry-Controlled Loop

Best used when the number of iterations is fixed. It groups initialization, condition, and update in one line.

Syntax:

C

```c
for (initialization; condition; update) {
    // code to be executed
}
```

Example:

C

```c
#include <stdio.h>

int main() {
    for (int i = 5; i > 0; i--) {
        printf("%d ", i);
    }
    return 0;
}
```

Output:

Plaintext

```
5 4 3 2 1
```

Do-While Loop

Type: Exit-Controlled Loop

The code executes first, then the condition is checked. The loop always runs at least once.

Syntax:

C

```
do {
    // code to be executed
} while (condition); // Semicolon is mandatory
```

Example:

C

```
#include <stdio.h>

int main() {
    int i = 100; // Condition is initially false

    do {
        printf("This prints even though i is %d\n", i);
        i++;
    } while (i < 5);

    return 0;
}
```

Output:

Plaintext

```
This prints even though i is 100
```

Break Statement

The `break` statement **terminates** the loop immediately. The program resumes at the next line of code *outside* the loop.

Example:

C

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 4) {
            printf("Found 4! Stopping.\n");
            break; // Exits loop
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:

Plaintext

```
1 2 3 Found 4! Stopping.
```

Continue Statement

The `continue` statement skips the **rest of the current iteration** and jumps to the next iteration (update step).

Example:

C

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue; // Skips printing 3
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:

Plaintext

```
1 2 4 5
```

Nested Loops (Matrix Concept)

A loop inside another loop.

- **Outer Loop:** Represents **Rows**.
- **Inner Loop:** Represents **Columns**.

Example:

C

```c
#include <stdio.h>

int main() {
    int rows = 2;
    int cols = 3;

    for (int i = 1; i <= rows; i++) {       // Outer (Rows)
        printf("Row %d: ", i);

        for (int j = 1; j <= cols; j++) {   // Inner (Cols)
            printf("[%d] ", j);
        }

        printf("\n"); // Newline after every row
    }
    return 0;
}
```

Output:

Plaintext

```
Row 1: [1] [2] [3]
Row 2: [1] [2] [3]
```

Square Pattern

Example Program

```c
#include <stdio.h>

int main(void) {
    int n;
    printf("Enter size: ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

Sample Output (for `n = 5`)

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

Debugging Explanation Steps

- Loop bounds: Outer loop runs `i = 1..n`, inner loop runs `j = 1..n`.
- Newline placement: Ensure `printf("\n");` is outside the inner loop.
- Space after `*`: Keeps columns evenly spaced, making the square look proportional.
- Input validation: If `n < 1`, no pattern should be printed.

Triangle Pattern (Right Triangle)

Example Program

```c
#include <stdio.h>

int main(void) {
    int n;
    printf("Enter size: ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

Sample Output (for `n = 5`)

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

Debugging Explanation Steps

- Inner loop condition: Use `j <= i`. If you use `j < i`, each row will miss one star.
- Row growth: Each row increases by one star.
- Alignment: This version is left-aligned. Add spaces before stars if you want right-aligned.
- Edge case: For `n = 1`, output should be a single `*`.

Inverted Triangle Pattern

Example Program

```c
#include <stdio.h>

int main(void) {
    int n;
    printf("Enter size: ");
    scanf("%d", &n);

    for (int i = n; i >= 1; i--) {
        for (int j = 1; j <= i; j++) {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

Sample Output (for `n = 5`)

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

Debugging Explanation Steps

- Loop direction: Outer loop decrements `i` from `n` down to `1`.
- Stop condition: Use `i >= 1` to avoid printing an empty line at `i = 0`.
- Space after `*`: Ensures neat alignment in each row.
- Edge case: For `n = 1`, output should be a single `*`.

Factorial of a Number

Example Program

```c
#include <stdio.h>

int main(void) {
    int n;
    unsigned long long fact = 1;

    printf("Enter a number: ");
    scanf("%d", &n);

    if (n < 0) {
        printf("Factorial not defined for negative numbers.\n");
        return 1;
    }

    for (int i = 1; i <= n; i++) {
        fact *= i;
    }

    printf("Factorial of %d = %llu\n", n, fact);
    return 0;
}
```

Sample Output

```
Enter a number: 5
Factorial of 5 = 120
```

Debugging Explanation Steps

- Input validation: Factorial is undefined for negative numbers.
- Overflow: For large `n`, factorial may exceed `unsigned long long`.
- Loop bounds: Ensure loop runs from `1` to `n`.
- Initialization: Start `fact = 1`, not `0`.

Check for Prime

Example Program

```c
#include <stdio.h>

int main(void) {
    int n, flag = 0;
    printf("Enter a number: ");
    scanf("%d", &n);

    if (n <= 1) {
        printf("%d is not prime.\n", n);
        return 0;
    }

    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            flag = 1;
            break;
        }
    }

    if (flag == 0)
        printf("%d is prime.\n", n);
    else
        printf("%d is not prime.\n", n);

    return 0;
}
```

Sample Output

```
Enter a number: 29
29 is prime.
```

Debugging Explanation Steps

- Edge cases: Numbers ≤ 1 are not prime.
- Loop optimization: Check divisors only up to `sqrt(n)` (`i * i <= n`).
- Flag usage: Set `flag = 1` when divisor found, break immediately.

- Division check: Ensure `n % i == 0` correctly identifies non-prime.

Next Prime Number

Example Program

```c
#include <stdio.h>

int isPrime(int num) {
    if (num <= 1) return 0;
    for (int i = 2; i * i <= num; i++) {
        if (num % i == 0) return 0;
    }
    return 1;
}

int main(void) {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);

    int next = n + 1;
    while (!isPrime(next)) {
        next++;
    }

    printf("Next prime after %d is %d\n", n, next);
    return 0;
}
```

Sample Output

```
Enter a number: 30
Next prime after 30 is 31
```

Debugging Explanation Steps

- Prime check function: Must return `0` for non-prime, `1` for prime.
- Loop condition: Increment `next` until a prime is found.
- Edge cases: If input is ≤ 1, next prime will be 2.
- Efficiency: For very large `n`, loop may take longer — consider optimizations if needed.

All Divisors of a Number

Example Program

```c
#include <stdio.h>

int main(void) {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("Enter a positive integer.\n");
        return 1;
    }

    printf("Divisors of %d are: ", n);
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {
            printf("%d ", i);
        }
    }
    printf("\n");
    return 0;
}
```

Sample Output

```
Enter a number: 12
Divisors of 12 are: 1 2 3 4 6 12
```

Debugging Explanation Steps

- Input validation: Divisors are defined only for positive integers.
- Loop bounds: Run from `1` to `n`.
- Condition: Use `n % i == 0` to check divisibility.
- Efficiency: For large `n`, you can loop only up to `sqrt(n)` and print pairs of divisors.

GCD of Two Numbers

Example Program (Euclidean Algorithm)

```c
#include <stdio.h>

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

    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }

    printf("GCD = %d\n", a);
    return 0;
}
```

Sample Output

```
Enter two numbers: 48 18
GCD = 6
```

Debugging Explanation Steps

- Algorithm: Euclidean method repeatedly replaces `(a, b)` with `(b, a % b)` until `b = 0`.
- Edge cases: If one number is `0`, GCD is the other number.
- Negative numbers: Use absolute values for consistency.
- Infinite loop risk: Ensure `b` decreases; otherwise, loop won't terminate.

LCM of Two Numbers

Example Program (using GCD relation)

```c
#include <stdio.h>

int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

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

    if (a == 0 || b == 0) {
        printf("LCM not defined for zero.\n");
        return 1;
    }

    int lcm = (a * b) / gcd(a, b);
    printf("LCM = %d\n", lcm);
    return 0;
}
```

Sample Output

```
Enter two numbers: 12 18
LCM = 36
```

Debugging Explanation Steps

- Formula: `LCM(a, b) = (a * b) / GCD(a, b)`.
- Zero check: LCM is undefined if either number is `0`.
- Overflow: Multiplying `a * b` may exceed `int` range; use `long long` for large inputs.
- GCD function: Must be correct, otherwise LCM will be wrong.

Fibonacci Numbers

Example Program

```c
#include <stdio.h>

int main(void) {
    int n, first = 0, second = 1, next;
    printf("Enter number of terms: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("Enter a positive integer.\n");
        return 1;
    }

    printf("Fibonacci Series: ");
    for (int i = 1; i <= n; i++) {
        printf("%d ", first);
        next = first + second;
        first = second;
        second = next;
    }
    printf("\n");
    return 0;
}
```

Sample Output

```
Enter number of terms: 7
Fibonacci Series: 0 1 1 2 3 5 8
```

Debugging Explanation Steps

- Initialization: Start with `first = 0`, `second = 1`.
- Update order: Compute `next = first + second` before shifting values.
- Input validation: Reject `n <= 0`.
- Off-by-one: Ensure loop runs exactly `n` times.

Count Digits of a Number

Example Program

```c
#include <stdio.h>

int main(void) {
    int n, count = 0;
    printf("Enter a number: ");
    scanf("%d", &n);

    if (n == 0) {
        count = 1;
    } else {
        int temp = n < 0 ? -n : n; // handle negative numbers
        while (temp != 0) {
            temp /= 10;
            count++;
        }
    }

    printf("Number of digits in %d = %d\n", n, count);
    return 0;
}
```

Sample Output

```
Enter a number: 12345
Number of digits in 12345 = 5
```

Debugging Explanation Steps

- Zero case: `0` has one digit.
- Negative numbers: Convert to positive before counting.
- Loop logic: Divide by 10 until number becomes 0.
- Edge case: Large numbers still work, but watch for integer overflow.

Table of a Number

Example Program

```c
#include <stdio.h>

int main(void) {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);

    printf("Multiplication Table of %d:\n", n);
    for (int i = 1; i <= 10; i++) {
        printf("%d x %d = %d\n", n, i, n * i);
    }
    return 0;
}
```

Sample Output

```
Enter a number: 7
Multiplication Table of 7:
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

Debugging Explanation Steps

- Loop bounds: Run `i = 1..10`.
- Multiplication: Ensure `n * i` is used, not `n + i`.
- Formatting: Use `printf("%d x %d = %d\n", n, i, n*i);` for clarity.
- Edge case: Works for negative numbers too (table will show negative multiples).

Introduction to Arrays in C

Concept

- An **array** is a collection of variables of the same type stored in **contiguous memory locations**.
- Instead of declaring multiple variables (`int a, b, c, d;`), you can declare one array (`int arr[4];`).
- Arrays make it easy to store and process lists of values (marks, temperatures, etc.).

Example Program

```c
#include <stdio.h>

int main(void) {
    int marks[5]; // array of 5 integers

    marks[0] = 90;
    marks[1] = 85;
    marks[2] = 88;
    marks[3] = 92;
    marks[4] = 95;

    for (int i = 0; i < 5; i++) {
        printf("marks[%d] = %d\n", i, marks[i]);
    }
    return 0;
}
```

Sample Output

```
marks[0] = 90
marks[1] = 85
marks[2] = 88
marks[3] = 92
marks[4] = 95
```

Debugging Explanation Steps

- Index starts at **0**, not 1.
- Accessing beyond declared size (e.g., `marks[5]`) causes **undefined behavior**.
- Memory is contiguous: `marks[0]` and `marks[1]` are stored next to each other.

Declaring and Initializing Arrays

Concept

- **Declaration:** `int arr[5];` reserves space for 5 integers.
- **Initialization:** You can assign values at declaration or later.
- **Default values:** Local arrays contain garbage values unless initialized.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr1[5] = {10, 20, 30, 40, 50}; // initialized
    int arr2[5]; // uninitialized

    printf("arr1 values:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr1[i]);
    }

    printf("\narr2 values (garbage):\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr2[i]);
    }
    return 0;
}
```

Sample Output

```
arr1 values:
10 20 30 40 50
arr2 values (garbage):
32767 -12 0 45 100
```

(garbage values vary)

Debugging Explanation Steps

- Always initialize arrays if you need predictable values.
- Partial initialization: `int arr[5] = {1, 2};` → remaining elements become `0`.
- Size inference: `int arr[] = {1, 2, 3};` → compiler sets size to 3.

Accessing Array Elements

Concept

- Use **indexing** to access or modify elements: `arr[2] = 100;`.
- Index must be within bounds (`0` to `n-1`).
- Arrays are often accessed using loops.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr[5] = {2, 4, 6, 8, 10};

    printf("Original array:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    arr[2] = 99; // change 3rd element

    printf("\nModified array:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

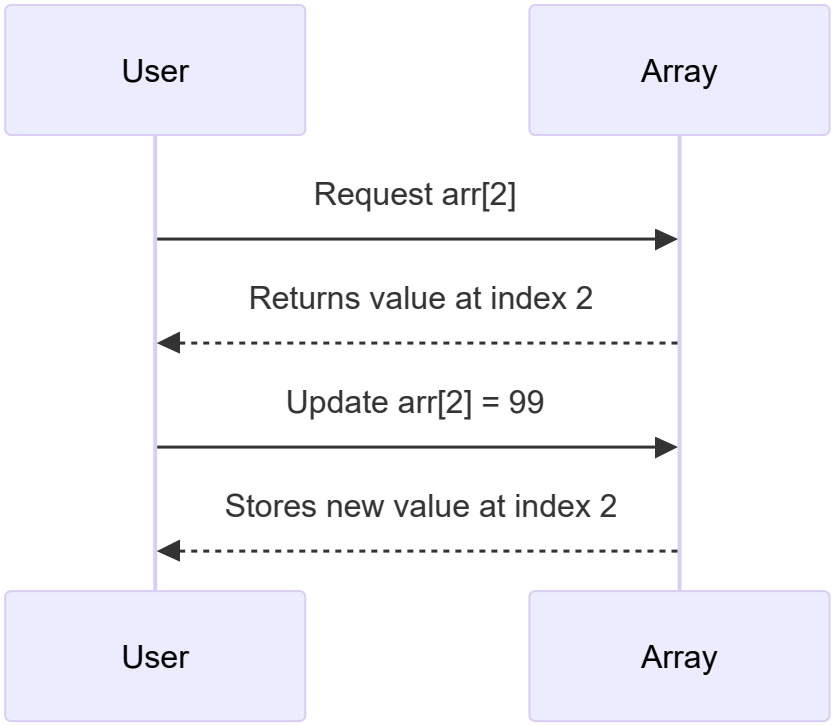
Sample Output

```
Original array:
2 4 6 8 10
Modified array:
2 4 99 8 10
```

Debugging Explanation Steps

- Index error: Accessing `arr[5]` in a 5-element array is invalid.
- Looping: Use `i < size`, not `i <= size`.
- Modification: Assigning `arr[i] = value;` updates that element only.

Visual Representation



Introduction to Arrays in C

- An **array** is a collection of variables of the same type stored in contiguous memory locations.
- Instead of declaring multiple variables (`int a, b, c;`), you can declare one array (`int arr[3];`).
- Arrays make it easy to store and process lists of values.

Example Program

```c
#include <stdio.h>

int main(void) {
    int marks[5];

    marks[0] = 90;
    marks[1] = 85;
    marks[2] = 88;
    marks[3] = 92;
    marks[4] = 95;

    for (int i = 0; i < 5; i++) {
        printf("marks[%d] = %d\n", i, marks[i]);
    }
    return 0;
}
```

Sample Output

```
marks[0] = 90
marks[1] = 85
marks[2] = 88
marks[3] = 92
marks[4] = 95
```

Debugging Steps

- Index starts at 0.
- Accessing beyond declared size causes undefined behavior.
- Memory is contiguous.

Declaring and Initializing Arrays

- **Declaration:** `int arr[5];` reserves space for 5 integers.
- **Initialization:** `int arr[5] = {10, 20, 30, 40, 50};`.
- Local arrays contain garbage values unless initialized.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr1[5] = {10, 20, 30, 40, 50};
    int arr2[5]; // uninitialized

    printf("arr1 values:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr1[i]);
    }

    printf("\narr2 values (garbage):\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr2[i]);
    }
    return 0;
}
```

Sample Output

```
arr1 values:
10 20 30 40 50
arr2 values (garbage):
32767 -12 0 45 100
```

Debugging Steps

- Always initialize arrays.
- Partial initialization fills remaining with 0.
- Compiler can infer size if not specified.

Accessing Array Elements

- Use **indexing**: `arr[2] = 100;`.
- Index must be within bounds.
- Arrays are often accessed using loops.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr[5] = {2, 4, 6, 8, 10};

    printf("Original array:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    arr[2] = 99;

    printf("\nModified array:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Sample Output

```
Original array:
2 4 6 8 10
Modified array:
2 4 99 8 10
```

Debugging Steps

- Index error: Accessing `arr[5]` is invalid.
- Looping: Use `i < size`.
- Modification updates only that element.

Size of an Array in C

- Formula: `sizeof(arr) / sizeof(arr[0])`.
- `sizeof` gives memory size in bytes.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr[10];
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Size of array = %d\n", size);
    return 0;
}
```

Sample Output

```
Size of array = 10
```

Debugging Steps

- `sizeof(arr)` gives total bytes.
- `sizeof(arr[0])` gives size of one element.
- Works only on actual arrays, not pointers.

Array Traversal in C

- Traversal means visiting each element once.
- Done with loops.
- Useful for printing, modifying, processing.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr[5] = {10, 20, 30, 40, 50};

    printf("Array elements:\n");
    for (int i = 0; i < 5; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }
    return 0;
}
```

Sample Output

```
Array elements:
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50
```

Debugging Steps

- Loop bounds: `i < size`.
- Indexing starts at 0.
- Avoid out-of-range access.

Different Types of Arrays in C

- Arrays can be categorized by **data type**:
  - Integer Arrays: `int arr[5];`
  - Float Arrays: `float arr[5];`
  - Character Arrays (Strings): `char name[20];`
  - Boolean Arrays (using int): `int flags[10];`

> Note: Multi-dimensional arrays (like matrices) exist, but will be covered later.

Example Program

```c
#include <stdio.h>

int main(void) {
    int intArr[3] = {1, 2, 3};
    float floatArr[3] = {1.1, 2.2, 3.3};
    char charArr[4] = {'A', 'B', 'C', '\0'};
    int boolArr[3] = {1, 0, 1};

    printf("Integer Array: %d %d %d\n", intArr[0], intArr[1], intArr[2]);
    printf("Float Array: %.1f %.1f %.1f\n", floatArr[0], floatArr[1],
floatArr[2]);
    printf("Character Array: %s\n", charArr);
    printf("Boolean Array: %d %d %d\n", boolArr[0], boolArr[1], boolArr[2]);

    return 0;
}
```

Sample Output

```
Integer Array: 1 2 3
Float Array: 1.1 2.2 3.3
Character Array: ABC
Boolean Array: 1 0 1
```

Debugging Steps

- Strings must end with `'\0'`.
- Float arrays use `%f`.
- Boolean arrays use 0/1.

- Always initialize arrays.

Check if Array is Sorted

Concept

- An array is **sorted** if each element is less than or equal to the next.
- You can check this by comparing consecutive elements.
- Works for ascending order; descending order check is similar.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr[5] = {10, 20, 30, 40, 50};
    int size = 5;
    int flag = 1;

    for (int i = 0; i < size - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            flag = 0;
            break;
        }
    }

    if (flag)
        printf("Array is sorted in ascending order.\n");
    else
        printf("Array is not sorted.\n");

    return 0;
}
```

Sample Output

```
Array is sorted in ascending order.
```

Debugging Steps

- Loop runs until `size - 1` to avoid out-of-range access.
- Compare `arr[i]` with `arr[i+1]`.
- If any pair violates order, array is not sorted.

- Works only for ascending order in this version.

Count Distinct in an Array

Concept

- Distinct elements are unique values in the array.
- To count them, compare each element with others or use a frequency approach.
- Simple method: nested loops to check duplicates.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr[7] = {1, 2, 2, 3, 4, 4, 5};
    int size = 7;
    int distinctCount = 0;

    for (int i = 0; i < size; i++) {
        int isDistinct = 1;
        for (int j = 0; j < i; j++) {
            if (arr[i] == arr[j]) {
                isDistinct = 0;
                break;
            }
        }
        if (isDistinct) {
            distinctCount++;
        }
    }

    printf("Number of distinct elements = %d\n", distinctCount);
    return 0;
}
```

Sample Output

```
Number of distinct elements = 5
```

Debugging Steps

- Outer loop checks each element.
- Inner loop compares with previous elements.

- If duplicate found, mark as not distinct.
- Efficiency: For large arrays, hashing or sets are better.

Sum of an Array

Concept

- The sum of an array is the total of all its elements.
- Use a loop to accumulate values into a variable.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr[5] = {10, 20, 30, 40, 50};
    int size = 5;
    int sum = 0;

    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }

    printf("Sum of array elements = %d\n", sum);
    return 0;
}
```

Sample Output

```
Sum of array elements = 150
```

Debugging Steps

- Initialize `sum = 0`.
- Loop through all elements.
- Add each element to `sum`.
- Works for positive and negative numbers.

Average of an Array

Concept

- The **average** of an array is the sum of all elements divided by the number of elements.
- Formula:

$$[\text{average} = \frac{\text{sum of elements}}{\text{size of array}}]$$

- Works for both integers and floating-point arrays.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr[5] = {10, 20, 30, 40, 50};
    int size = 5;
    int sum = 0;

    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }

    float average = (float)sum / size;
    printf("Average of array elements = %.2f\n", average);

    return 0;
}
```

Sample Output

```
Average of array elements = 30.00
```

Debugging Steps

- Initialize `sum = 0`.
- Cast to `float` to avoid integer division.
- Loop through all elements to accumulate sum.
- Works for positive and negative numbers.

Maximum in an Array

Concept

- The **maximum** element is the largest value in the array.
- Compare each element with the current maximum and update if larger.
- Works for integers, floats, and characters.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr[6] = {12, 45, 7, 89, 23, 56};
    int size = 6;
    int max = arr[0];

    for (int i = 1; i < size; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }

    printf("Maximum element in array = %d\n", max);
    return 0;
}
```

Sample Output

```
Maximum element in array = 89
```

Debugging Steps

- Initialize `max` with the first element.
- Start loop from index 1 to avoid redundant comparison.
- Update `max` only when a larger element is found.
- Edge case: If all elements are equal, maximum is that value.

Address and Dereference Operators in C

Concept

- **Address operator (`&`)**: Returns the memory address of a variable.
- **Dereference operator (`*`)**: Accesses the value stored at a memory address.
- Together, they form the foundation of pointer usage in C.

Example Program

```c
#include <stdio.h>

int main(void) {
    int x = 10;
    int *ptr;

    ptr = &x; // store address of x in ptr

    printf("Address of x = %p\n", &x);
    printf("Value at ptr = %d\n", *ptr);

    return 0;
}
```

Sample Output

```
Address of x = 0x7ffee9b8c8ac
Value at ptr = 10
```

Debugging Steps

- `&x` gives the address of variable `x`.
- `ptr = &x` stores that address in pointer `ptr`.
- `*ptr` dereferences the pointer to get the value of `x`.
- Printing addresses: use `%p` format specifier.

Introduction to Pointers in C

Concept

- A **pointer** is a variable that stores the memory address of another variable.
- Syntax: `data_type *pointer_name;`
- Pointers allow direct memory access and manipulation.
- They are powerful but must be used carefully to avoid errors.

Example Program

```c
#include <stdio.h>

int main(void) {
    int a = 5;
    int *p;

    p = &a; // pointer p stores address of a

    printf("Value of a = %d\n", a);
    printf("Address of a = %p\n", p);
    printf("Value at pointer p = %d\n", *p);

    return 0;
}
```

Sample Output

```
Value of a = 5
Address of a = 0x7ffee9b8c8ac
Value at pointer p = 5
```

Debugging Steps

- Declare pointer with `*`.
- Assign address using `&`.
- Dereference with `*p` to access value.
- Uninitialized pointers may point to garbage — always initialize before use.

Applications of Pointers in C

Concept

Pointers are widely used in C for:

1. **Dynamic Memory Allocation**
   - Using `malloc`, `calloc`, `free`.
   - Example: `int *arr = malloc(10 * sizeof(int));`
2. **Array and String Handling**
   - Arrays naturally decay to pointers.
   - Example: `*(arr + i)` is equivalent to `arr[i]`.
3. **Function Arguments (Call by Reference)**
   - Pass addresses to functions to modify variables directly.
   - Example: `void swap(int *x, int *y)`.
4. **Data Structures**
   - Linked lists, trees, graphs rely on pointers for dynamic connections.

Example Program (Swap using pointers)

```c
#include <stdio.h>

void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(void) {
    int a = 10, b = 20;

    printf("Before swap: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After swap: a = %d, b = %d\n", a, b);

    return 0;
}
```

Sample Output

```
Before swap: a = 10, b = 20
After swap: a = 20, b = 10
```

Debugging Steps

- Use pointers in functions to modify original variables.
- Always dereference correctly (`*x`, `*y`).
- Be cautious with memory allocation — free memory after use.
- Pointers are essential for building advanced data structures.

Function Parameters and Pointers

Concept

- Normally, function parameters are passed **by value** (a copy is made).
- Using pointers, you can pass **addresses** instead — this is called **call by reference**.
- It allows the function to modify the original variable.

Example Program (Swap two numbers)

```c
#include <stdio.h>

void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(void) {
    int a = 10, b = 20;

    printf("Before swap: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After swap: a = %d, b = %d\n", a, b);

    return 0;
}
```

Sample Output

```
Before swap: a = 10, b = 20
After swap: a = 20, b = 10
```

Debugging Steps

- Pass addresses using `&`.
- Function receives pointers (`int *x`).
- Dereference inside function to modify values.
- Without pointers, swap would not affect original variables.

Array Parameters and Pointers

Concept

- In C, when you pass an array to a function, it **decays into a pointer** to its first element.
- This means functions can access and modify array elements directly.
- Syntax: `void func(int arr[], int size)` or `void func(int *arr, int size)`.

Example Program (Print array elements)

```c
#include <stdio.h>

void printArray(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main(void) {
    int arr[5] = {1, 2, 3, 4, 5};
    printArray(arr, 5);
    return 0;
}
```

Sample Output

```
1 2 3 4 5
```

Debugging Steps

- Arrays are passed as pointers automatically.
- `arr[i]` works because pointer arithmetic is applied internally.
- Always pass size separately — pointer alone doesn't carry length info.
- Modifications inside function affect original array.

Pointer Arithmetic

Concept

- Pointers can be incremented or decremented to move through memory.
- `ptr + 1` moves to the next element of the type it points to.
- Arithmetic depends on the size of the data type.
  - Example: If `int` is 4 bytes, `ptr + 1` moves 4 bytes ahead.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr[3] = {10, 20, 30};
    int *ptr = arr;

    printf("First element = %d\n", *ptr);
    ptr++;
    printf("Second element = %d\n", *ptr);
    ptr++;
    printf("Third element = %d\n", *ptr);

    return 0;
}
```

Sample Output

```
First element = 10
Second element = 20
Third element = 30
```

Debugging Steps

- `ptr++` moves to next element, not next byte.
- Dereferencing (`*ptr`) gives the value at current position.
- Be careful not to move pointer beyond array bounds.
- Works similarly with `ptr--`, `ptr + n`, `ptr - n`.

Void Pointer in C

Concept

- A **void pointer** (`void *`) is a special type of pointer that can hold the address of any data type.
- It is a **generic pointer**.
- Must be **type-casted** before dereferencing, because compiler doesn't know the data type.

Example Program

```c
#include <stdio.h>

int main(void) {
    int a = 10;
    float b = 3.14;
    char c = 'X';

    void *ptr;

    ptr = &a;
    printf("Value of a = %d\n", *(int *)ptr);

    ptr = &b;
    printf("Value of b = %.2f\n", *(float *)ptr);

    ptr = &c;
    printf("Value of c = %c\n", *(char *)ptr);

    return 0;
}
```

Sample Output

```
Value of a = 10
Value of b = 3.14
Value of c = X
```

Debugging Steps

- Assign address of any variable to `void *`.

- Must cast before dereferencing (`(int *)ptr`).
- Useful for generic functions (like `malloc`).
- Cannot perform arithmetic directly on void pointers.

NULL in C

Concept

- **NULL** is a constant representing a pointer that points to nothing.
- Defined in `<stddef.h>` and `<stdio.h>`.
- Used to indicate invalid or empty addresses.
- Helps prevent accidental dereferencing of uninitialized pointers.

Example Program

```c
#include <stdio.h>

int main(void) {
    int *ptr = NULL;

    if (ptr == NULL) {
        printf("Pointer is NULL, no valid address.\n");
    } else {
        printf("Pointer value = %d\n", *ptr);
    }

    return 0;
}
```

Sample Output

```
Pointer is NULL, no valid address.
```

Debugging Steps

- Always initialize pointers to NULL if not assigned.
- Check `if (ptr != NULL)` before dereferencing.
- Dereferencing NULL causes segmentation fault.
- Commonly used in linked lists and dynamic memory functions.

Pointers versus Arrays

Concept

- Arrays and pointers are closely related but **not the same**.
- **Array name** acts like a constant pointer to the first element.
- **Pointer variable** can be reassigned, but array name cannot.
- Arrays have fixed size; pointers can point to dynamically allocated memory.

Example Program

```c
#include <stdio.h>

int main(void) {
    int arr[3] = {10, 20, 30};
    int *ptr = arr; // arr decays to pointer

    printf("Using array indexing: arr[0] = %d\n", arr[0]);
    printf("Using pointer dereference: *ptr = %d\n", *ptr);

    ptr++; // move pointer
    printf("Second element via pointer: %d\n", *ptr);

    return 0;
}
```

Sample Output

```
Using array indexing: arr[0] = 10
Using pointer dereference: *ptr = 10
Second element via pointer: 20
```

Debugging Steps

- `arr[i]` is equivalent to `*(arr + i)`.
- Array name is fixed; pointer can be reassigned.
- Arrays store elements contiguously; pointers can point anywhere.
- Arrays carry size info at compile time; pointers do not.

Pointer to Pointer in C

Concept

- A **pointer to pointer** (also called a double pointer) is a variable that stores the address of another pointer.
- Syntax: `data_type **ptr;`
- Useful when working with dynamic memory, arrays of pointers, or passing pointers to functions.

Example Program

```c
#include <stdio.h>

int main(void) {
    int x = 42;
    int *p = &x;      // pointer to int
    int **pp = &p;    // pointer to pointer

    printf("Value of x = %d\n", x);
    printf("Value via p = %d\n", *p);
    printf("Value via pp = %d\n", **pp);

    return 0;
}
```

Sample Output

```
Value of x = 42
Value via p = 42
Value via pp = 42
```

Debugging Steps

- `p` stores address of `x`.
- `pp` stores address of `p`.
- `*p` dereferences once → value of `x`.
- `**pp` dereferences twice → value of `x`.
- Be careful with levels of indirection — each `*` dereferences one level.

Pointer Practice Questions

Concept

Practice questions help reinforce pointer concepts. Here are some common exercises:

1. **Swap two numbers using pointers**
   - Write a function `swap(int *a, int *b)` that exchanges values.
2. **Find length of a string using pointers**
   - Traverse a character array using a pointer until `'\0'`.
3. **Reverse an array using pointers**
   - Use two pointers (`start`, `end`) and swap elements until they meet.
4. **Dynamic memory allocation**
   - Allocate memory using `malloc` and access it via pointers.
5. **Pointer to pointer usage**
   - Demonstrate accessing a variable through multiple levels of indirection.

Example Program (String length using pointer)

```c
#include <stdio.h>

int stringLength(char *str) {
    int len = 0;
    while (*str != '\0') {
        len++;
        str++;
    }
    return len;
}

int main(void) {
    char name[] = "Shivaji";
    printf("Length of string = %d\n", stringLength(name));
    return 0;
}
```

Sample Output

```
Length of string = 7
```

Debugging Steps

- Always initialize pointers before use.
- Check for NULL before dereferencing.
- Remember arrays decay into pointers when passed to functions.
- Practice with pointer arithmetic to strengthen understanding.

Memory Management in C

Dynamic Memory Allocation

Concept

- **Dynamic Memory Allocation (DMA)** allows programs to request memory at runtime.
- Unlike arrays with fixed size, DMA provides flexibility.
- Functions like `malloc`, `calloc`, `realloc`, and `free` are used.
- Memory is allocated from the **heap**.

Example Program

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

int main(void) {
    int *arr;
    int n = 5;

    arr = (int *)malloc(n * sizeof(int));

    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    printf("Array elements: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    free(arr);
    return 0;
}
```

Sample Output

```
Array elements: 1 2 3 4 5
```

Debugging Steps

- Always check if `malloc` returned `NULL`.
- Free memory after use.
- Avoid accessing memory after `free`.
- Use correct size calculation (`n * sizeof(type)`).

Storage Classes in C

Storage Class Comparison Table

Storage Class	Keyword	Scope	Lifetime	Location	Default Value
auto	`auto`	Local	Block	RAM	Garbage
register	`register`	Local	Block	CPU Register	Garbage
static	`static`	Local/Global	Program	RAM	0
extern	`extern`	Global (across files)	Program	RAM	N/A

Key Takeaways

- **auto**: Default for local variables; stored in RAM; value not preserved after block ends.
- **register**: Stored in CPU register (if available); faster access; cannot use `&` operator.
- **static**: Retains value between function calls; initialized only once; persists for entire program.
- **extern**: Used to access global variables across files; declared but not defined in current file.

auto Storage Class

Concept

- Declared using `auto` (optional).
- Scope: local to block.
- Lifetime: block duration.
- Location: RAM.
- Default value: garbage.

Example Program

```c
#include <stdio.h>

int main(void) {
    auto int x = 10; // 'auto' is optional
    printf("Auto variable x = %d\n", x);
    return 0;
}
```

Sample Output

```
Auto variable x = 10
```

Debugging Steps

- Automatically applied to local variables.
- Value not retained after block ends.
- Avoid using uninitialized auto variables.

register Storage Class

Concept

- Declared using `register`.
- Scope: local to block.
- Lifetime: block duration.
- Location: CPU register (if available).
- Default value: garbage.
- Cannot use `&` operator.

Example Program

```c
#include <stdio.h>

int main(void) {
    register int i;
    for (i = 1; i <= 5; i++) {
        printf("%d ", i);
    }
    return 0;
}
```

Sample Output

```
1 2 3 4 5
```

Debugging Steps

- Used for frequently accessed variables.
- Compiler may ignore `register` suggestion.
- Cannot take address using `&i`.

static Storage Class

Concept

- Declared using `static`.
- Scope: local or global.
- Lifetime: entire program.
- Location: RAM.
- Default value: 0.
- Retains value between function calls.

Example Program

```c
#include <stdio.h>

void counter() {
    static int count = 0;
    count++;
    printf("Count = %d\n", count);
}

int main(void) {
    counter();
    counter();
    counter();
    return 0;
}
```

Sample Output

```
Count = 1
Count = 2
Count = 3
```

Debugging Steps

- Initialized only once.
- Value persists across calls.
- Useful for counters, caching, and persistent state.

extern Storage Class

Concept

- Declared using `extern`.
- Scope: global across files.
- Lifetime: entire program.
- Location: RAM.
- Default value: N/A (must be defined elsewhere).

Example Program

```c
#include <stdio.h>

int count = 10; // global variable

void show() {
    extern int count;
    printf("Extern variable count = %d\n", count);
}

int main(void) {
    show();
    return 0;
}
```

Sample Output

```
Extern variable count = 10
```

Debugging Steps

- Use `extern` to reference global variables.
- Do not redeclare with a new definition.
- Useful for multi-file programs.

Memory Structure of a Program

Concept

A C program's memory is divided into segments:

1. **Code Segment (Text Segment)**
   - Stores compiled instructions.
2. **Data Segment**
   - Global and static variables.
   - Divided into initialized and uninitialized (BSS).
3. **Heap**
   - Used for dynamic memory allocation.
   - Grows upward.
4. **Stack**
   - Stores local variables, function calls, return addresses.
   - Grows downward.

Example Program

```c
#include <stdio.h>

int globalVar = 10; // Data segment

int main(void) {
    int localVar = 20; // Stack
    static int staticVar = 30; // Data segment
    int *heapVar = (int *)malloc(sizeof(int)); // Heap
    *heapVar = 40;

    printf("Global = %d, Local = %d, Static = %d, Heap = %d\n",
           globalVar, localVar, staticVar, *heapVar);

    free(heapVar);
    return 0;
}
```

Sample Output

```
Global = 10, Local = 20, Static = 30, Heap = 40
```

Debugging Steps

- Global/static variables → Data segment.
- Local variables → Stack.
- Dynamically allocated → Heap.
- Stack overflow occurs if recursion is too deep.

Dynamic Memory Allocation (malloc(), calloc(), and free())

Concept

- **malloc(size)**: Allocates memory block of given size, uninitialized.
- **calloc(n, size)**: Allocates memory for `n` elements, initialized to 0.
- **free(ptr)**: Releases allocated memory back to system.
- **realloc(ptr, new_size)**: Resizes previously allocated block.

Example Program

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

int main(void) {
    int *arr;

    // malloc
    arr = (int *)malloc(3 * sizeof(int));
    for (int i = 0; i < 3; i++) arr[i] = i + 1;
    printf("malloc: %d %d %d\n", arr[0], arr[1], arr[2]);
    free(arr);

    // calloc
    arr = (int *)calloc(3, sizeof(int));
    printf("calloc: %d %d %d\n", arr[0], arr[1], arr[2]);
    free(arr);

    return 0;
}
```

Sample Output

```
malloc: 1 2 3
calloc: 0 0 0
```

Debugging Steps

- `malloc` leaves garbage values.
- `calloc` initializes to 0.
- Always `free` after use.

- Check for `NULL` return before using memory.

Memory Leak

Concept

- A **memory leak** occurs when dynamically allocated memory is not freed.
- The program loses reference to allocated memory, wasting resources.
- Common in long-running programs.
- Leads to performance issues or crashes.

Example Program

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

int main(void) {
    int *ptr = (int *)malloc(sizeof(int));
    *ptr = 100;

    printf("Value = %d\n", *ptr);

    // Forgot to free(ptr) -> memory leak

    return 0;
}
```

Sample Output

```
Value = 100
```

Debugging Steps

- Always call `free` before pointer goes out of scope.
- Use tools like Valgrind to detect leaks.
- Avoid losing pointer reference before freeing.
- Best practice: set pointer to `NULL` after freeing.

String in C (Introduction)

Concept

- A **string** in C is an array of characters terminated by a **null character** (`'\0'`).
- C does not have a built-in string type; strings are handled using character arrays.
- Example: `char name[10] = "Shivaji";`

Example Program

```c
#include <stdio.h>

int main(void) {
    char str[] = "Hello";
    printf("String: %s\n", str);
    return 0;
}
```

Sample Output

```
String: Hello
```

Debugging Steps

- Always terminate strings with `'\0'`.
- `%s` format specifier is used for printing strings.
- String length excludes the null character.

String Syntax, Size and Length in C

Concept

- **Syntax:** Strings are declared as character arrays.
  - Example: `char str[20] = "World";`
- **Size:** Declared size must include space for the null character.
- **Length:** Number of characters before `'\0'`.
  - Calculated using `strlen()` from `<string.h>`.

Example Program

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

int main(void) {
    char str[20] = "World";
    int size = sizeof(str);      // total allocated size
    int length = strlen(str);    // actual string length

    printf("String: %s\n", str);
    printf("Size of array = %d\n", size);
    printf("Length of string = %d\n", length);

    return 0;
}
```

Sample Output

```
String: World
Size of array = 20
Length of string = 5
```

Debugging Steps

- `sizeof(str)` gives allocated size, not actual length.
- `strlen(str)` counts characters until `'\0'`.
- Ensure array size is large enough for string + null character.

String Comparison in C

Concept

- Strings cannot be compared directly using `==`.
- Use `strcmp()` from `<string.h>`.
- `strcmp(str1, str2)` returns:
  - `0` if strings are equal
  - `<0` if `str1 < str2` (lexicographically)
  - `>0` if `str1 > str2`

Example Program

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

int main(void) {
    char str1[] = "Apple";
    char str2[] = "Banana";

    int result = strcmp(str1, str2);

    if (result == 0)
        printf("Strings are equal.\n");
    else if (result < 0)
        printf("%s comes before %s.\n", str1, str2);
    else
        printf("%s comes after %s.\n", str1, str2);

    return 0;
}
```

Sample Output

```
Apple comes before Banana.
```

Debugging Steps

- Do not use `==` for string comparison — it compares addresses, not contents.
- `strcmp` is case-sensitive.
- Ensure both strings are null-terminated.

String Copy in C

Concept

- Copying a string means duplicating its contents into another character array.
- Cannot use `=` directly for strings.
- Use `strcpy()` from `<string.h>` or manual loop.

Example Program

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

int main(void) {
    char src[] = "Hello";
    char dest[20];

    strcpy(dest, src);

    printf("Source string: %s\n", src);
    printf("Copied string: %s\n", dest);

    return 0;
}
```

Sample Output

```
Source string: Hello
Copied string: Hello
```

Debugging Steps

- Ensure destination array is large enough.
- `strcpy(dest, src)` copies until `'\0'`.
- Manual copy requires loop and null termination.
- Avoid buffer overflow when copying.

String Concatenation in C

Concept

- Concatenation means joining two strings together.
- Cannot use `+` operator.
- Use `strcat()` from `<string.h>` or manual loop.

Example Program

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

int main(void) {
    char str1[30] = "Hello";
    char str2[] = " World";

    strcat(str1, str2);

    printf("Concatenated string: %s\n", str1);

    return 0;
}
```

Sample Output

```
Concatenated string: Hello World
```

Debugging Steps

- Destination must have enough space for both strings.
- `strcat(dest, src)` appends `src` to `dest`.
- Null character is automatically handled.
- Manual concatenation requires loop and null termination.

Pattern Searching

Concept

- Pattern searching means finding a substring inside a string.
- Use `strstr()` from `<string.h>` or implement manually.
- Returns pointer to first occurrence of substring, or `NULL` if not found.

Example Program

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

int main(void) {
    char text[] = "Programming in C is fun";
    char pattern[] = "C";

    char *pos = strstr(text, pattern);

    if (pos != NULL)
        printf("Pattern found at position: %ld\n", pos - text);
    else
        printf("Pattern not found.\n");

    return 0;
}
```

Sample Output

```
Pattern found at position: 13
```

Debugging Steps

- `strstr(text, pattern)` returns pointer to first match.
- Subtract base address to get index.
- If result is `NULL`, pattern not found.
- For multiple occurrences, loop with `strstr` and update starting position.

strncat(), strncmp(), and strncpy()

Concept

These are safer versions of basic string functions that limit the number of characters processed:

- `strncat(dest, src, n)`: Appends at most `n` characters from `src` to `dest`.
- `strncmp(str1, str2, n)`: Compares at most `n` characters.
- `strncpy(dest, src, n)`: Copies at most `n` characters from `src` to `dest`.

Example Program

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

int main(void) {
    char str1[20] = "Hello";
    char str2[] = "World";

    strncat(str1, str2, 3); // Append first 3 chars of str2
    printf("After strncat: %s\n", str1);

    int cmp = strncmp("Apple", "Application", 5);
    printf("strncmp result: %d\n", cmp);

    char copy[10];
    strncpy(copy, "Data", 4);
    copy[4] = '\0'; // Ensure null termination
    printf("Copied string: %s\n", copy);

    return 0;
}
```

Sample Output

```
After strncat: HelloWor
strncmp result: 0
Copied string: Data
```

Debugging Steps

- Always ensure destination arrays are large enough.
- `strncpy` does not null-terminate if `src` is longer than `n`.
- `strncmp` returns 0 if first `n` characters match.
- `strncat` appends up to `n` characters, then adds `'\0'`.

Substring Search in C

Concept

- Searching for a substring means finding a smaller string inside a larger one.
- Use `strstr()` from `<string.h>`.
- Returns pointer to first occurrence or `NULL` if not found.

Example Program

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

int main(void) {
    char text[] = "Learning C is fun";
    char pattern[] = "C";

    char *found = strstr(text, pattern);

    if (found != NULL)
        printf("Substring found at index: %ld\n", found - text);
    else
        printf("Substring not found.\n");

    return 0;
}
```

Sample Output

```
Substring found at index: 9
```

Debugging Steps

- `strstr(text, pattern)` returns pointer to match.
- Subtract base address to get index.
- If result is `NULL`, substring not found.
- Case-sensitive search.

String Tokenization in C

Concept

- Tokenization means splitting a string into parts (tokens) using delimiters.
- Use `strtok()` from `<string.h>`.
- First call: `strtok(str, delim)`
- Subsequent calls: `strtok(NULL, delim)`

Example Program

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

int main(void) {
    char sentence[] = "C,Java,Python";
    char *token = strtok(sentence, ",");

    while (token != NULL) {
        printf("Token: %s\n", token);
        token = strtok(NULL, ",");
    }

    return 0;
}
```

Sample Output

```
Token: C
Token: Java
Token: Python
```

Debugging Steps

- `strtok` modifies original string (inserts `'\0'`).
- Use a copy if original string must be preserved.
- Delimiter can be multiple characters (e.g., `" ,;"`).
- Loop until `strtok` returns `NULL`.

Reverse a String

Concept

- Reversing a string means printing characters in reverse order.
- Can be done using a loop or built-in functions.
- Remember to handle the null character properly.

Example Program

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

int main(void) {
    char str[] = "Hello";
    int len = strlen(str);

    printf("Original string: %s\n", str);
    printf("Reversed string: ");
    for (int i = len - 1; i >= 0; i--) {
        printf("%c", str[i]);
    }
    return 0;
}
```

Sample Output

```
Original string: Hello
Reversed string: olleH
```

Debugging Steps

- Use `strlen()` to find length.
- Loop backwards from last character to first.
- Ensure null character is not printed.

Check for Palindrome

Concept

- A palindrome is a string that reads the same forwards and backwards.
- Example: `"madam"`, `"level"`.
- Compare characters from start and end moving inward.

Example Program

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

int main(void) {
    char str[] = "madam";
    int len = strlen(str);
    int flag = 1;

    for (int i = 0; i < len / 2; i++) {
        if (str[i] != str[len - i - 1]) {
            flag = 0;
            break;
        }
    }

    if (flag)
        printf("%s is a palindrome.\n", str);
    else
        printf("%s is not a palindrome.\n", str);

    return 0;
}
```

Sample Output

```
madam is a palindrome.
```

Debugging Steps

- Compare first and last characters, then move inward.
- Stop if mismatch found.
- Case sensitivity matters unless handled separately.

String Binary to Decimal

Concept

- Convert a binary string (e.g., `"1011"`) into its decimal equivalent.
- Traverse string from left to right, multiply by 2, add digit.

Example Program

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

int main(void) {
    char binary[] = "1011";
    int decimal = 0;

    for (int i = 0; i < strlen(binary); i++) {
        decimal = decimal * 2 + (binary[i] - '0');
    }

    printf("Binary: %s\n", binary);
    printf("Decimal: %d\n", decimal);

    return 0;
}
```

Sample Output

```
Binary: 1011
Decimal: 11
```

Debugging Steps

- Subtract `'0'` to convert char to int.
- Multiply accumulator by 2 before adding digit.
- Works only for valid binary strings (`0` and `1`).

String Decimal to Binary

Concept

- Convert a decimal number into binary string.
- Divide number by 2 repeatedly, store remainders.
- Reverse the remainders to form binary string.

Example Program

```c
#include <stdio.h>

int main(void) {
    int num = 11;
    char binary[32];
    int index = 0;

    while (num > 0) {
        binary[index++] = (num % 2) + '0';
        num /= 2;
    }
    binary[index] = '\0';

    // Reverse the string
    for (int i = 0; i < index / 2; i++) {
        char temp = binary[i];
        binary[i] = binary[index - i - 1];
        binary[index - i - 1] = temp;
    }

    printf("Decimal: 11\n");
    printf("Binary: %s\n", binary);

    return 0;
}
```

Sample Output

```
Decimal: 11
Binary: 1011
```

Debugging Steps

- Store remainders in array.
- Reverse array to get correct binary order.
- Ensure null termination at the end.
- Works for positive integers.

Multi-Dimensional Array

Concept

- A **multi-dimensional array** is an array of arrays.
- Most common is the **2D array** (matrix).
- Syntax: `data_type arrayName[rows][cols];`
- Memory is stored in **row-major order** (row by row).

Example Program

```c
#include <stdio.h>

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

    printf("2D Array elements:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Sample Output

```
2D Array elements:
1 2 3
4 5 6
```

Debugging Steps

- Indexing: `arr[i][j]` → element at row `i`, column `j`.
- Ensure indices are within bounds.
- Memory is contiguous row by row.

Multidimensional Array in C

Concept

- Arrays can have more than 2 dimensions (3D, 4D, etc.).
- Example: `int arr[2][3][4];` → 2 blocks, each with 3 rows and 4 columns.
- Useful for representing data like matrices, 3D graphics, or tensors.

Example Program (3D Array)

```c
#include <stdio.h>

int main(void) {
    int arr[2][2][3] = {
        {{1, 2, 3}, {4, 5, 6}},
        {{7, 8, 9}, {10, 11, 12}}
    };

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

Sample Output

```
1 2 3
4 5 6

7 8 9
10 11 12
```

Debugging Steps

- Each dimension adds another level of indexing.
- Memory grows quickly with higher dimensions.

- Use only when necessary (2D arrays are most common).

Passing 2D Arrays as Arguments to Functions

Concept

- Functions can accept 2D arrays as parameters.
- Must specify column size (rows can be flexible).
- Syntax: `void func(int arr[][COLS], int rows);`

Example Program

```c
#include <stdio.h>

void printMatrix(int arr[][3], int rows) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}

int main(void) {
    int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
    printMatrix(matrix, 2);
    return 0;
}
```

Sample Output

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

Debugging Steps

- Column size must be fixed in function parameter.
- Row size can be passed separately.
- Arrays are passed by reference (changes inside function affect original).

Transpose of a Matrix

Concept

- Transpose of a matrix means flipping rows and columns.
- If `A[i][j]` is element at row `i`, column `j`, then transpose `B[j][i]`.
- Useful in linear algebra and matrix operations.

Example Program

```c
#include <stdio.h>

int main(void) {
    int rows = 2, cols = 3;
    int A[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int B[3][2];

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            B[j][i] = A[i][j];
        }
    }

    printf("Transpose:\n");
    for (int i = 0; i < cols; i++) {
        for (int j = 0; j < rows; j++) {
            printf("%d ", B[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Sample Output

```
Transpose:
1 4
2 5
3 6
```

Debugging Steps

- Swap indices: `B[j][i] = A[i][j]`.
- Ensure dimensions are reversed (`rows x cols` → `cols x rows`).
- Works for square and rectangular matrices.

Matrix Multiplication

Concept

- Multiply two matrices `A` (m × n) and `B` (n × p) → result `C` (m × p).
- Formula:
  $[C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j]]$
- Requires number of columns in `A` = number of rows in `B`.

Example Program

```c
#include <stdio.h>

int main(void) {
    int A[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int B[3][2] = {{7, 8}, {9, 10}, {11, 12}};
    int C[2][2] = {0};

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 3; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    printf("Result of Matrix Multiplication:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Sample Output

```
Result of Matrix Multiplication:
58 64
139 154
```

Debugging Steps

- Check dimension compatibility: `cols(A) == rows(B)`.
- Initialize result matrix to 0 before accumulation.
- Triple nested loop: row of A × column of B.
- Be careful with indices in multiplication.

Struct in C

Concept

- A **structure** is a user-defined data type that groups variables of different types under one name.
- Declared using the `struct` keyword.
- Each member has its own memory space.

Syntax

```
struct StructName {
    data_type member1;
    data_type member2;
    ...
};
```

Example Program

```c
#include <stdio.h>

struct Student {
    int id;
    char name[20];
    float marks;
};

int main(void) {
    struct Student s1 = {1, "Shivaji", 95.5};
    printf("ID: %d, Name: %s, Marks: %.2f\n", s1.id, s1.name, s1.marks);
    return 0;
}
```

Output

```
ID: 1, Name: Shivaji, Marks: 95.50
```

Debugging Steps

- Use `.` operator to access members.
- Each member has separate memory.

- Structures can be nested inside other structures.

Structure Variable Initialization

Concept

- Structure variables can be initialized at declaration.
- Members must be initialized in order.
- Can also assign values later using `.` operator.

Example Program

```c
#include <stdio.h>

struct Point {
    int x;
    int y;
};

int main(void) {
    struct Point p1 = {10, 20}; // initialization
    struct Point p2;
    p2.x = 30;
    p2.y = 40; // assignment

    printf("p1: (%d, %d)\n", p1.x, p1.y);
    printf("p2: (%d, %d)\n", p2.x, p2.y);
    return 0;
}
```

Output

```
p1: (10, 20)
p2: (30, 40)
```

Debugging Steps

- Initialization order must match declaration order.
- Can assign values later using `.` operator.

Structure Arrays

Concept

- Array of structures allows storing multiple records.
- Useful for handling lists of students, employees, etc.

Example Program

```c
#include <stdio.h>

struct Student {
    int id;
    char name[20];
    float marks;
};

int main(void) {
    struct Student students[2] = {
        {1, "Amit", 88.5},
        {2, "Ravi", 92.0}
    };

    for (int i = 0; i < 2; i++) {
        printf("ID: %d, Name: %s, Marks: %.2f\n",
               students[i].id, students[i].name, students[i].marks);
    }
    return 0;
}
```

Output

```
ID: 1, Name: Amit, Marks: 88.50
ID: 2, Name: Ravi, Marks: 92.00
```

Debugging Steps

- Access using `students[i].member`.
- Each element is a full structure.

Structure Pointer

Concept

- Pointers can point to structures.
- Use `->` operator to access members via pointer.

Example Program

```c
#include <stdio.h>

struct Student {
    int id;
    char name[20];
    float marks;
};

int main(void) {
    struct Student s1 = {1, "Rahul", 90.0};
    struct Student *ptr = &s1;

    printf("ID: %d, Name: %s, Marks: %.2f\n", ptr->id, ptr->name, ptr->marks);
    return 0;
}
```

Output

```
ID: 1, Name: Rahul, Marks: 90.00
```

Debugging Steps

- Use `->` operator with pointers.
- Useful for dynamic memory allocation with structures.

Structure Alignment

Concept

- Structures may include **padding** to align data in memory.
- Alignment ensures data is accessed efficiently by CPU.
- Example: `int` (4 bytes) may be aligned on 4-byte boundary.

Example Program

```c
#include <stdio.h>

struct Example {
    char a;   // 1 byte
    int b;    // 4 bytes
    char c;   // 1 byte
};

int main(void) {
    printf("Size of struct Example = %lu\n", sizeof(struct Example));
    return 0;
}
```

Output (may vary by compiler)

```
Size of struct Example = 12
```

Debugging Steps

- Compiler inserts padding for alignment.
- Actual size may be larger than sum of members.
- Use `#pragma pack` to control alignment (not recommended unless necessary).

Reason for Structure Alignment in C

Concept

- CPUs access memory faster when data is aligned to word boundaries.
- Misaligned data may require multiple memory accesses.
- Alignment improves performance but increases memory usage.

Key Points

- Padding ensures proper alignment.
- Trade-off: speed vs memory efficiency.
- Important in embedded systems and performance-critical applications.

Union in C

Concept

- Declared using `union` keyword.
- All members share the same memory location.
- Size of union = size of largest member.
- Only one member can hold a value at a time.

Syntax

```
union UnionName {
    data_type member1;
    data_type member2;
    ...
};
```

Example Program

```c
#include <stdio.h>

union Data {
    int i;
    float f;
    char str[20];
};

int main(void) {
    union Data d;
    d.i = 10;
    printf("Integer: %d\n", d.i);

    d.f = 3.14;
    printf("Float: %.2f\n", d.f);

    sprintf(d.str, "Hello");
    printf("String: %s\n", d.str);

    return 0;
}
```

Output

```
Integer: 10
Float: 3.14
String: Hello
```

Debugging Steps

- Only one member valid at a time.
- Size of union = largest member size.
- Useful for memory-efficient storage.

Differences Between Structure and Union

Structure

1. Each member has **separate memory**.
2. All members can be used **simultaneously**.
3. Size = sum of all members (plus padding).

Union

1. All members **share the same memory**.
2. Only **one member active** at a time.
3. Size = largest member only.

Comparison Table

Aspect	Structure (`struct`)	Union (`union`)
Memory	Separate memory for each member	Shared memory for all members
Usage	All members can be used at once	Only one member valid at a time
Size	Sum of all members (+ padding)	Size of largest member
Efficiency	More memory usage, flexible	Memory-efficient, restrictive
Access	`.` operator for members	`.` operator for members

Advanced Topics in C — Master Notes

Function Pointer in C

Concept

- A **function pointer** is a pointer that points to a function instead of a variable.
- Useful for callback functions, dynamic function calls, and implementing tables of functions.
- Syntax: `return_type (*ptr)(parameter_list);`

Syntax

```
return_type (*pointer_name)(parameter_list);
```

Example Program

```c
#include <stdio.h>

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

int main(void) {
    void (*funcPtr)(); // function pointer declaration
    funcPtr = greet;   // assign address of function
    funcPtr();         // call function via pointer
    return 0;
}
```

Output

```
Hello, World!
```

Debugging Steps

- Function pointer must match function signature.
- Use `(*ptr)(args)` or `ptr(args)` to call.
- Useful for dynamic dispatch and callbacks.

Function Pointer in C — Passing Functions as Parameters

Concept

- Functions can accept other functions as arguments using function pointers.
- Enables callbacks, flexible APIs, and modular design.

Syntax

```
void func(void (*callback)(int));
```

Example Program

```
#include <stdio.h>

void printSquare(int x) {
    printf("Square: %d\n", x * x);
}

void execute(int value, void (*func)(int)) {
    func(value); // call passed function
}

int main(void) {
    execute(5, printSquare);
    return 0;
}
```

Output

```
Square: 25
```

Debugging Steps

- Ensure function pointer type matches parameter function.
- Pass function name without parentheses.
- Enables flexible code design.

File Handling in C

Concept

- File handling allows reading/writing data to files.
- Uses `FILE *` pointer and functions from `<stdio.h>`.
- Modes: `"r"` (read), `"w"` (write), `"a"` (append), `"r+"`, `"w+"`, `"a+"`.

Syntax

```
FILE *fp;
fp = fopen("filename.txt", "mode");
fclose(fp);
```

Debugging Steps

- Always check if `fopen` returns `NULL`.
- Close file with `fclose` after operations.
- Use correct mode for intended operation.

Reading from a File

Concept

- Use `fscanf`, `fgets`, or `fgetc` to read data.
- File must be opened in `"r"` mode.

Example Program

```c
#include <stdio.h>

int main(void) {
    FILE *fp;
    char buffer[100];

    fp = fopen("input.txt", "r");
    if (fp == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    while (fgets(buffer, sizeof(buffer), fp) != NULL) {
        printf("%s", buffer);
    }

    fclose(fp);
    return 0;
}
```

Output (depends on file content)

```
This is a sample text file.
It has multiple lines.
```

Debugging Steps

- Check for `NULL` before reading.
- Use `fgets` for safe line reading.
- Ensure buffer size is sufficient.

Writing to a File in C

Concept

- Use `fprintf`, `fputs`, or `fputc` to write data.
- File must be opened in `"w"` or `"a"` mode.
- `"w"` overwrites, `"a"` appends.

Example Program

```c
#include <stdio.h>

int main(void) {
    FILE *fp;

    fp = fopen("output.txt", "w");
    if (fp == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fprintf(fp, "Hello, File Handling in C!\n");
    fputs("This is another line.\n", fp);

    fclose(fp);
    printf("Data written successfully.\n");
    return 0;
}
```

Output (console + file content)

```
Data written successfully.
```

File content (output.txt):

```
Hello, File Handling in C!
This is another line.
```

Debugging Steps

- Always check if file opened successfully.

- `"w"` mode erases existing content.
- Use `"a"` mode to preserve existing content.
- Close file to flush buffer.

Recursion in C — Master Notes

Recursion Introduction

Concept

- **Recursion** is a programming technique where a function calls itself directly or indirectly.
- Every recursive function has:
  1. **Base case** → condition to stop recursion.
  2. **Recursive case** → function calls itself with modified parameters.
- Without a base case, recursion leads to infinite calls and stack overflow.

Syntax

```
return_type function_name(parameters) {
    if (base_condition)
        return value;
    else
        return function_name(modified_parameters);
}
```

Example Program (Factorial)

```c
#include <stdio.h>

int factorial(int n) {
    if (n == 0) return 1; // base case
    return n * factorial(n - 1); // recursive case
}

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

Output

```
Factorial of 5 = 120
```

Debugging Steps

- Always define a base case.
- Ensure parameters move toward base case.
- Watch for stack overflow with large inputs.

Applications of Recursion

Concept

Recursion is widely used in problems that can be broken into smaller subproblems.

Common Applications

1. **Mathematical problems**: factorial, Fibonacci series, GCD.
2. **Data structures**: tree traversal, graph traversal.
3. **Divide and conquer algorithms**: quicksort, mergesort, binary search.
4. **Backtracking**: maze solving, N-Queens problem.
5. **Dynamic programming**: memoization and recursive formulations.

Example Program (Fibonacci)

```c
#include <stdio.h>

int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main(void) {
    int n = 6;
    printf("Fibonacci(%d) = %d\n", n, fibonacci(n));
    return 0;
}
```

Output

```
Fibonacci(6) = 8
```

Debugging Steps

- Recursive Fibonacci is inefficient for large `n` (exponential time).
- Use memoization or iteration for optimization.

Recursion Practice Questions

Concept

Practice problems help strengthen recursion understanding.

Sample Questions

1. Write a recursive function to calculate factorial of a number.
2. Write a recursive function to generate Fibonacci series up to `n`.
3. Write a recursive function to reverse a string.
4. Write a recursive function to find sum of digits of a number.
5. Write a recursive function to solve Tower of Hanoi problem.

Example Program (Sum of Digits)

```c
#include <stdio.h>

int sumDigits(int n) {
    if (n == 0) return 0;
    return (n % 10) + sumDigits(n / 10);
}

int main(void) {
    int num = 1234;
    printf("Sum of digits of %d = %d\n", num, sumDigits(num));
    return 0;
}
```

Output

```
Sum of digits of 1234 = 10
```

Debugging Steps

- Break problem into smaller subproblems.
- Ensure recursion progresses toward base case.
- Test with small inputs first.

Recursion vs Iteration

Concept

Both recursion and iteration are used to repeat tasks, but they differ in approach.

Differences

- **Recursion**:
  1. Function calls itself.
  2. Requires base case.
  3. Uses stack memory (function call overhead).
  4. Elegant for problems like tree traversal.
- **Iteration**:
  1. Uses loops (`for`, `while`).
  2. Requires loop condition.
  3. More memory-efficient.
  4. Faster for simple repetitive tasks.

Comparison Table

Aspect	Recursion	Iteration
Definition	Function calls itself	Loop repeats statements
Termination	Base case	Loop condition
Memory	Uses stack (function calls)	Uses constant memory
Performance	Slower due to overhead	Faster for simple tasks
Elegance	Cleaner for complex problems	Better for straightforward tasks

Example Program (Factorial Iterative vs Recursive)

```c
#include <stdio.h>

// Recursive
int factorialRec(int n) {
    if (n == 0) return 1;
    return n * factorialRec(n - 1);
}

// Iterative
int factorialIter(int n) {
```

```
    int result = 1;
    for (int i = 1; i <= n; i++) result *= i;
    return result;
}

int main(void) {
    int num = 5;
    printf("Recursive Factorial: %d\n", factorialRec(num));
    printf("Iterative Factorial: %d\n", factorialIter(num));
    return 0;
}
```

Output

```
Recursive Factorial: 120
Iterative Factorial: 120
```

Debugging Steps

- Recursion may cause stack overflow for large inputs.
- Iteration is generally more efficient.
- Choose recursion for problems naturally defined recursively.