Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM **(INDEPENDENT SKILL DEVELOPMENT MISSION)**

INTRODUCTION TO PROGRAMMING AND C LANGUAGE

INTRODUCTION TO PROGRAMMING

What is Programming?

Programming is the process of writing instructions that a computer can understand and execute to perform specific tasks. These instructions, known as **code**, are written in programming languages that define a structured way to interact with a computer. Programming enables automation, software development, and problem-solving in various domains such as web applications, artificial intelligence, data processing, and system software.

Programming languages can be categorized into different levels based on their proximity to machine instructions. **Low-level languages**, such as assembly and machine code, provide direct hardware control but are difficult to understand. **High-level languages**, like Python, Java, and C, are easier to write and read since they use syntax closer to human language. The **C language** bridges the gap between low-level and high-level programming, providing both hardware-level control and efficient software development capabilities.

In a programming environment, an essential concept is **compilation and execution**. A program written in a high-level language must be converted into machine code through a **compiler** before execution.

For example, in C programming, the **GCC (GNU Compiler Collection)** compiles the code into an executable file, which the operating system runs.

Example of a simple C program:

```
#include <stdio.h>


int main() {

   printf("Hello, World!\n");

   return 0;

}
```

This program prints "Hello, World!" to the screen, demonstrating a basic structure in C programming.

Why Learn Programming?

Programming is a critical skill for software developers, engineers, and data scientists. It enhances **logical thinking, problem-solving abilities, and career opportunities** in various industries. Some of the key benefits of learning programming include:

- **Automation**: Reducing manual effort by automating repetitive tasks.
- **Software Development**: Creating applications, websites, and systems software.
- **Problem Solving**: Developing algorithms to solve real-world challenges.
- **Career Growth**: High demand for programmers in software engineering, data science, and cybersecurity.

HISTORY AND EVOLUTION OF C LANGUAGE

Origins of C Language

The **C programming language** was developed by **Dennis Ritchie** at Bell Labs in 1972. It was designed as an improvement over the **B language**, which was influenced by BCPL (Basic Combined Programming Language). C introduced structured programming concepts and efficient memory management, making it one of the most widely used programming languages even today.

C was primarily used for developing the **UNIX operating system**, which made it popular in the software development industry. It influenced many later programming languages, including **C++, Java, Python, and JavaScript**.

Key Features of C Language

- **Procedural Language**: C follows a step-by-step approach to problem-solving.
- **Structured Programming**: Uses functions, loops, and conditional statements for better code organization.
- **Portability**: C programs can run on various platforms with minimal modifications.
- **Efficiency**: Direct memory access and optimized execution make C highly efficient.
- **Flexibility**: Suitable for system programming, embedded systems, and application development.

Example: Basic C Program Structure

A simple C program consists of:

1. **Preprocessor Directives** (#include <stdio.h> – includes standard libraries)
2. **Main Function** (int main() {} – entry point of execution)
3. **Code Statements** (printf() – used for displaying output)

```
#include <stdio.h>


int main() {

   printf("Welcome to C Programming!\n");

   return 0;

}
```

This program prints a welcome message and demonstrates the fundamental syntax of C.

FUNDAMENTAL CONCEPTS OF C PROGRAMMING

1. Variables and Data Types

Variables store data values that can be manipulated during program execution. In C, every variable must be declared with a **data type**, which defines the kind of value it holds.

Data Type	Description	Example
int	Integer values	int age = 25;
float	Decimal values	float price = 99.99;
char	Single character	char grade = 'A';

double	Double-precision decimal values	double pi = 3.14159;

2. Operators and Expressions

C provides various operators to perform arithmetic, logical, and relational operations.

- **Arithmetic Operators**: +, -, *, /, %
- **Relational Operators**: ==, !=, >, <, >=, <=
- **Logical Operators**: &&, ||, !

Example of arithmetic operations:

```
#include <stdio.h>


int main() {
  int a = 10, b = 5;
  printf("Sum: %d\n", a + b);
  return 0;
}
```

3. Control Flow Statements

C supports decision-making and loop structures for controlling program execution.

- **Conditional Statements**: if, if-else, switch
- **Looping Statements**: for, while, do-while

Example of a loop:

```
#include <stdio.h>


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

This loop executes **five times**, printing iteration numbers.

Case Study: Building a Simple Student Record System

Problem Statement

A school wants to develop a **basic student record management system** using C. The program should store student names, grades, and roll numbers, and allow retrieval of records.

Solution Approach

1. **Use arrays** to store student details.
2. **Implement file handling** to save and retrieve records.
3. **Use functions** for modular programming.

Implementation

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

```
struct Student {

   int roll;

   char name[50];

   float grade;

};


int main() {

   struct Student s1 = {101, "John Doe", 89.5};

   printf("Student: %s\nRoll: %d\nGrade: %.2f\n", s1.name, s1.roll,
s1.grade);

   return 0;

}
```

This program **stores student details using structures** and prints them.

Exercise

1. Write a C program to **calculate the factorial of a number** using a loop.
2. Implement a **simple grade calculator** that takes user input for marks and assigns grades.
3. Modify the **student record program** to allow user input instead of predefined values.

4. Write a program to **check whether a number is even or odd** using an if-else statement.
5. Develop a program that **counts the number of vowels in a given string**.

CONCLUSION

C programming is a **powerful, efficient, and widely-used language** that provides deep control over hardware and software execution. Understanding **basic programming concepts, C syntax, control flow, and data handling** is crucial for mastering C. This foundational knowledge **prepares learners** for **system programming, software development, and embedded applications**.

SETTING UP A C PROGRAMMING ENVIRONMENT

UNDERSTANDING THE C PROGRAMMING ENVIRONMENT

What is a Programming Environment?

A **programming environment** is a set of tools, software, and configurations that allow developers to write, compile, and debug programs efficiently. For C programming, the environment includes:

- **A text editor** to write code
- **A compiler** to convert code into machine-executable format
- **A debugger** to identify and fix errors
- **A terminal or command prompt** to execute the compiled programs

Setting up an appropriate **C programming environment** ensures a smooth development experience, enabling programmers to focus on writing efficient code without unnecessary technical hurdles.

Why is Environment Setup Important?

A properly configured environment:
✓ Ensures compatibility across different operating systems
✓ Allows error-free compilation and debugging
✓ Provides tools for effective code writing and execution
✓ Enhances productivity with automation and optimization features

This chapter will guide you through the **step-by-step setup** of a C programming environment across **Windows, macOS, and Linux** systems.

CHOOSING THE RIGHT C COMPILER

What is a Compiler?

A **compiler** translates human-readable C code into machine code that the operating system can execute. Different compilers are available for various platforms, such as:

Compiler	**Platform**	**Features**
GCC (GNU Compiler Collection)	Windows, macOS, Linux	Open-source, widely used
Clang	macOS, Linux	High performance, used in macOS development
Turbo C++	Windows	Old but still used for educational purposes
Microsoft C Compiler (MSVC)	Windows	Integrated with Visual Studio

For beginners and professionals, **GCC (GNU Compiler Collection)** is recommended due to its **cross-platform support, efficiency, and extensive community support**.

Checking if GCC is Installed

Before installing GCC, check if it is already available on your system:

```
gcc --version
```

If the command returns a version number, GCC is installed. Otherwise, follow the installation steps for your operating system.

INSTALLING C PROGRAMMING ENVIRONMENT ON DIFFERENT PLATFORMS

Windows Setup

1. **Install MinGW (Minimalist GNU for Windows)**
   - Download MinGW from mingw-w64.org
   - Install and add MinGW's bin directory to the system PATH
2. **Verify Installation**
   Open **Command Prompt** and type:
3. gcc --version

If it displays the version, the installation is successful.

4. **Install Code Editor (Optional)**
   - Recommended: **VS Code, Code::Blocks, Dev-C++**
   - Configure the editor to use GCC for compiling C programs

Linux Setup

1. **Install GCC Compiler**
2. sudo apt update
3. sudo apt install gcc -y
4. **Verify Installation**
5. gcc --version

6. **Install a Text Editor**
   Options include **Vim, Nano, VS Code, or Code::Blocks**
7. sudo apt install vim -y

macOS Setup

1. **Install Xcode Command Line Tools**
2. xcode-select --install
3. **Verify GCC Installation**
4. gcc --version
5. **Install a Text Editor**
   Recommended: **VS Code, Sublime Text, or Vim**

Writing and Compiling a C Program

Step 1: Writing the Code

Create a file called hello.c:

```
#include <stdio.h>


int main() {
  printf("Hello, C Programming!\n");
  return 0;
}
```

Step 2: Compiling the Code

Use GCC to compile the program:

```
gcc hello.c -o hello
```

Step 3: Running the Program

Execute the compiled file:

```
./hello
```

Expected Output:

```
Hello, C Programming!
```

Debugging and Error Handling

Understanding Compilation Errors

Common errors encountered during compilation include:

- **Syntax Errors**: Incorrect code syntax
- **Missing Semicolons**: Every statement should end with ;
- **Undefined Variables**: Ensure all variables are declared
- **Linker Errors**: Caused by missing libraries

Example of a syntax error:

```
#include <stdio.h>

int main() {
   printf("Hello, World!")
   return 0;
}
```

Fix: Add a semicolon after printf("Hello, World!");

Using a Debugger

A debugger helps identify logical errors. Use gdb (GNU Debugger):

```
gcc -g hello.c -o hello
gdb hello
```

Commands in gdb:

- **break main** – Set a breakpoint at the main function
- **run** – Run the program
- **step** – Execute line-by-line

CASE STUDY: SETTING UP A C DEVELOPMENT ENVIRONMENT FOR A SOFTWARE FIRM

Scenario:

A **software development firm** needs to configure a **standardized C development environment** for its engineers across multiple operating systems. The firm has **20 developers**, half using **Windows** and half using **Linux/macOS**.

Solution Implementation:

1. **Windows Setup**
   - Install MinGW and configure PATH
   - Use VS Code with the C/C++ extension
2. **Linux/macOS Setup**

  - Install GCC, Vim, and VS Code
  - Configure Makefiles for automated builds

3. **Testing and Deployment**
  - Run sample programs on each system
  - Debug compilation issues
  - Document setup for new developers

Outcome:

- Successfully standardized **C development tools**
- Reduced **setup time** for new developers
- Improved **code consistency across platforms**

EXERCISE

1. **Set up the C development environment** on your system and verify GCC installation.
2. **Write a basic C program** to display your name and compile it using GCC.
3. **Experiment with different text editors** (Nano, Vim, VS Code) to write and compile C programs.
4. **Modify the 'Hello, World' program** to accept user input and display a personalized message.
5. **Practice debugging**: Introduce errors in your C program and fix them using a debugger.

CONCLUSION

Setting up a **C programming environment** is the first step toward becoming a **proficient C developer**. Choosing the right **compiler, editor, and debugging tools** ensures a smooth coding experience. Mastering **compilation, debugging, and environment optimization** prepares developers for **high-performance programming in software development, embedded systems, and operating system development**.

Understanding C Program Structure (Header Files, Functions, and Execution Flow)

Introduction to C Program Structure

What is a C Program?

A **C program** is a collection of **instructions** that tell the computer how to perform a specific task. Unlike interpreted languages like Python, **C programs are compiled** into machine code before execution. A typical **C program structure** consists of various components, including **header files, functions, variables, and execution flow control**.

Why Understanding C Program Structure is Important?

- **Code Organization**: Helps in writing clean and modular code.
- **Efficiency**: Reduces execution time by following a structured approach.
- **Scalability**: Allows easy modification and extension of programs.
- **Debugging & Maintenance**: Makes debugging and troubleshooting easier.

This chapter covers the **core structure of a C program**, including **header files, functions, and execution flow**, with practical examples and exercises.

Components of a C Program

A **basic C program** consists of the following parts:

1. **Preprocessor Directives (Header Files)**
2. **Global Declarations**
3. **Main Function (Entry Point)**
4. **User-Defined Functions**
5. **Execution Flow (Statements, Loops, and Function Calls)**

Here's a simple **C program structure**:

```
#include <stdio.h>  // Header file


// Function prototype

void greet();


// Main function

int main() {

   printf("Hello, C Programming!\n");

   greet();  // Function call

   return 0;

}


// Function definition

void greet() {
    printf("Welcome to C Language!\n");
}
```

UNDERSTANDING HEADER FILES IN C

What Are Header Files?

Header files in C contain **function declarations, macros, and type definitions** that can be included in a program using the #include directive. They allow **code reuse and modular programming**.

Common Header Files in C

Header File	Purpose
<stdio.h>	Standard Input/Output functions (printf, scanf)
<stdlib.h>	Memory management and type conversions
<math.h>	Mathematical functions (sqrt, pow)
<string.h>	String handling functions (strlen, strcpy)
<time.h>	Time-related functions

Example: Using Multiple Header Files

```
#include <stdio.h>  // For input/output
#include <math.h>   // For mathematical functions

int main() {
    double num = 25.0;
    printf("Square root of %.2f is %.2f\n", num, sqrt(num));

    return 0;

}
```

Here, <stdio.h> is used for printing output, and <math.h> provides the sqrt() function to calculate the square root of a number.

UNDERSTANDING FUNCTIONS IN C

What is a Function?

A **function** in C is a block of code designed to perform a specific task. Functions allow:
✓ **Code Reusability** – Avoid writing the same code multiple times.
✓ **Modular Programming** – Organize the program into smaller, manageable parts.
✓ **Easy Debugging** – Identify and fix errors quickly.

Types of Functions in C

1. **Library Functions** – Built-in functions like printf(), scanf(), sqrt().
2. **User-Defined Functions** – Custom functions created by programmers.

Function Structure in C

A function in C consists of **three parts**:

1. **Function Declaration (Prototype)** – Tells the compiler about the function.
2. **Function Definition** – Contains the actual code.

3. **Function Call** – Executes the function in the program.

Example: Function Declaration, Definition, and Call

```
#include <stdio.h>


// Function declaration

void greet();


// Main function

int main() {

   greet();  // Function call

   return 0;

}


// Function definition

void greet() {

   printf("Hello from a function!\n");

}
```

Output:

```
Hello from a function!
```

Function Parameters and Return Values

A function can accept **parameters (arguments)** and return values.

```
#include <stdio.h>


// Function declaration

int add(int, int);


int main() {

   int sum = add(5, 10);

   printf("Sum: %d\n", sum);

   return 0;

}


// Function definition

int add(int a, int b) {

   return a + b;

}
```

Output:

```
Sum: 15
```

Understanding Execution Flow in C

How Does C Execute a Program?

A C program follows a structured **execution flow**:

1. **Compilation** – Converts source code into machine code.
2. **Linking** – Resolves dependencies and creates an executable file.
3. **Execution** – Runs the compiled program line by line.

Flow of Execution in a C Program

1. The **main function** is executed first.
2. Statements within main() are executed **line by line**.
3. Function calls execute their respective function blocks.
4. The program **terminates** when the return statement is reached.

Example of Execution Flow with Multiple Functions

```
#include <stdio.h>


// Function prototypes

void step1();

void step2();


int main() {

   printf("Program Started\n");

   step1();

   step2();

   printf("Program Ended\n");
```

```
    return 0;
}


// Function definitions
void step1() {
    printf("Step 1 Executed\n");
}
void step2() {
    printf("Step 2 Executed\n");
}
```

Execution Order (Output):

```
Program Started
Step 1 Executed
Step 2 Executed
Program Ended
```

Case Study: Modular Programming in C for a Banking System

Problem Statement:

A bank requires a **C program** that performs **account creation, deposits, and withdrawals**. Instead of writing all logic in main(), the **program is modularized** using **functions**.

Solution Approach:

1. **Create separate functions** for create_account(), deposit(), and withdraw().
2. **Use header files** for function declarations.
3. **Organize execution flow** to allow user input and operations.

Implementation:

```
#include <stdio.h>


// Function prototypes

void create_account();

void deposit();

void withdraw();


int main() {

   create_account();

   deposit();

   withdraw();

   return 0;

}


// Function definitions

void create_account() {
  printf("Account Created Successfully!\n");
}
void deposit() {
  printf("Amount Deposited Successfully!\n");
}
void withdraw() {
  printf("Amount Withdrawn Successfully!\n");
}
```

Outcome:

✅ The program follows **modular programming** principles.
✅ Functions **enhance code readability and maintainability**.
✅ Execution flow remains **structured and user-friendly**.

Exercise

1. **Modify the function example** to calculate the **product of two numbers**.
2. **Create a function** that checks if a number is even or odd.
3. **Write a C program** that takes a user's name as input and prints a greeting.
4. **Experiment with function parameters and return types** by implementing an area calculator.
5. **Create a function-based program** that simulates a simple **login system**.

CONCLUSION

Understanding **C program structure** is **fundamental** for writing **efficient and modular code**. By mastering **header files, functions, and execution flow**, programmers can create **scalable and maintainable** applications. Functions enable **reusability**, execution flow ensures **structured program execution**, and **header files** provide essential tools for development.

DATA TYPES, VARIABLES, CONSTANTS, AND KEYWORDS IN C

INTRODUCTION TO DATA REPRESENTATION IN C

What is Data Representation?

Data representation refers to the way information is stored, processed, and manipulated in a program. In C programming, data is represented using different **data types**, stored in **variables**, and can be defined as **constants** when immutable. C also uses **keywords** that have predefined meanings for the compiler.

Why Understanding Data Types and Variables is Important?

- **Memory Efficiency**: Helps in selecting appropriate data types to optimize memory usage.
- **Program Accuracy**: Prevents data loss or errors due to incorrect type usage.
- **Code Readability**: Improves clarity by defining meaningful variable names and data types.
- **Efficient Computations**: Ensures correct mathematical and logical operations in programs.

This chapter explores **data types, variables, constants, and keywords** with detailed explanations, examples, exercises, and a case study.

DATA TYPES IN C

What are Data Types?

A **data type** defines the type of value a variable can store and the amount of memory allocated to it.

Primary Data Types in C

C has several fundamental data types categorized into **basic, derived, and user-defined types**.

Data Type	Description	Memory Size (in bytes)	Example
int	Integer values (whole numbers)	4	int age = 25;
float	Decimal numbers with single precision	4	float price = 99.99;
double	Decimal numbers with double precision	8	double pi = 3.141592;
char	Stores single character	1	char grade = 'A';
void	Represents empty or no data	0	Used in function return types

Example: Declaring and Using Data Types

```
#include <stdio.h>


int main() {

   int number = 10;

   float percentage = 89.5;

   char grade = 'A';
```

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

    printf("Percentage: %.2f\n", percentage);

    printf("Grade: %c\n", grade);


    return 0;
}
```

Output:

```
Number: 10

Percentage: 89.50

Grade: A
```

VARIABLES IN C

What is a Variable?

A **variable** is a named storage location in memory that holds a value, which can be modified during program execution.

Rules for Naming Variables

1. Must **start with a letter (A-Z or a-z) or an underscore (_)**.
2. Cannot use **C keywords** as variable names.
3. Should not contain spaces or special characters (@, $, %).
4. Can contain **letters, digits, and underscores (_)**.

Declaring and Initializing Variables

```
#include <stdio.h>


int main() {
    int age = 25;  // Declaring and initializing an integer variable
    float salary = 50000.75;  // Float variable
    char gender = 'M';  // Character variable


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


    return 0;
}
```

Scope of Variables

- **Local Variables**: Declared inside a function and accessible only within that function.
- **Global Variables**: Declared outside all functions and accessible throughout the program.
- **Static Variables**: Retain their value even after function exits.

Example: Global vs Local Variables

```
#include <stdio.h>


int globalVar = 10;  // Global variable
```

```
void display() {
    int localVar = 5;  // Local variable
    printf("Local Variable: %d\n", localVar);
    printf("Global Variable: %d\n", globalVar);
}

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

CONSTANTS IN C

What is a Constant?

A **constant** is a variable whose value remains fixed throughout program execution. Constants can be:

1. **Literal Constants** (directly written values like 10, 'A', 3.14)
2. **Symbolic Constants** (defined using #define)
3. **Constant Variables** (declared with const keyword)

Declaring Constants

```
#include <stdio.h>


#define PI 3.14159  // Symbolic constant


int main() {
   const int DAYS_IN_WEEK = 7;  // Constant variable


   printf("Value of PI: %.5f\n", PI);
   printf("Days in a week: %d\n", DAYS_IN_WEEK);


   return 0;
}
```

Types of Constants in C

- **Integer Constants**: Whole numbers (10, 200, -45)
- **Floating Point Constants**: Decimal numbers (3.14, -0.99)
- **Character Constants**: Single characters enclosed in single quotes ('A', 'b')
- **String Constants**: Sequences of characters enclosed in double quotes ("Hello")

KEYWORDS IN C

What are Keywords?

Keywords are reserved words in C with predefined meanings that cannot be used as variable names. C has **32 standard keywords**.

List of C Keywords

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

Example: Using Keywords in a Program

```
#include <stdio.h>


void display();  // Function prototype using void keyword


int main() {
   int age = 30;  // Using int keyword
   display();
   return 0;
}


void display() {
   printf("Inside the function display()\n");
}
```

Case Study: Student Marks Calculation System

Problem Statement:

A school wants to **store student marks** for three subjects and calculate the **total and average** using C programming.

Solution Approach:

1. **Use variables** to store marks and student name.
2. **Use constants** to define the number of subjects.
3. **Use data types appropriately** for different values.

Implementation:

```
#include <stdio.h>


#define SUBJECTS 3  // Constant value


int main() {
   char name[50];
   int marks1, marks2, marks3;


   printf("Enter Student Name: ");
   scanf("%s", name);
```

```
    printf("Enter marks for three subjects: ");
    scanf("%d %d %d", &marks1, &marks2, &marks3);

    int total = marks1 + marks2 + marks3;
    float average = total / (float)SUBJECTS;  // Typecasting for accuracy

    printf("Student: %s\n", name);
    printf("Total Marks: %d\n", total);
    printf("Average Marks: %.2f\n", average);

    return 0;
}
```

Outcome:

✔ Demonstrates **use of variables, constants, and data types**.
✔ Handles **user input and mathematical operations**.
✔ Provides **accurate calculations using typecasting**.

Exercise

1. **Write a program** that accepts two numbers from the user and prints their sum.

2. **Modify the student marks program** to calculate the highest and lowest marks.
3. **Use a constant** to define a tax percentage and calculate salary after tax deduction.
4. **Write a program** to swap two numbers using a third variable.
5. **Print the ASCII value of a character** entered by the user.

CONCLUSION

Understanding **data types, variables, constants, and keywords** is essential for writing **efficient C programs**. Selecting the right **data type** optimizes memory, using **variables** properly improves readability, and **constants** ensure reliability. Mastering these fundamentals is key to building **robust applications** in C programming.

OPERATORS AND EXPRESSIONS

CHAPTER 1: INTRODUCTION TO OPERATORS AND EXPRESSIONS

Operators and expressions form the backbone of any programming language. They allow programmers to perform computations, manipulate data, and control the flow of execution. An **operator** is a symbol that performs a specific operation on one or more operands, while an **expression** is a combination of variables, constants, operators, and functions that evaluate to a value.

Programming languages provide various types of operators, including arithmetic, relational, logical, bitwise, assignment, and special operators. These operators help execute mathematical calculations, make decisions, and process data efficiently. Expressions are the building blocks of programming logic and enable code to function dynamically.

For example:

```
x = 5 + 3 * 2

print(x)  # Output: 11
```

Here, multiplication has a higher precedence than addition, so 3 * 2 is calculated first, then added to 5. Understanding operators and expressions is crucial to writing efficient and correct code.

CHAPTER 2: TYPES OF OPERATORS

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, division, and modulus.

These operators work with numerical values and are commonly used in calculations.

Examples:

```
x = 10

y = 3

print(x + y)  # Output: 13

print(x - y)  # Output: 7

print(x * y)  # Output: 30

print(x / y)  # Output: 3.33

print(x % y)  # Output: 1
```

In expressions, arithmetic operators follow the precedence order: **Parentheses, Exponents, Multiplication/Division, Addition/Subtraction (PEMDAS).** Mastering arithmetic operators enables programmers to efficiently manipulate numerical data.

Relational (Comparison) Operators

Relational operators compare values and return Boolean results (True or False). These are widely used in decision-making and loops.

Examples:

```
x = 10

y = 5

print(x > y)  # Output: True

print(x < y)  # Output: False
```

print(x == y) # Output: False

print(x != y) # Output: True

Understanding relational operators is fundamental in programming as they help implement logic conditions and filtering mechanisms.

CHAPTER 3: LOGICAL OPERATORS

Logical operators are used to combine multiple conditions and return a Boolean value. These operators include **AND (and), OR (or), and NOT (not)**. Logical operators are commonly used in conditional statements and loops.

Examples:

x = True

y = False

print(x and y) # Output: False

print(x or y)   # Output: True

print(not x)   # Output: False

Logical operators are essential for designing complex decision structures in programs. They help developers create robust and efficient code for various real-world applications.

CHAPTER 4: ASSIGNMENT AND BITWISE OPERATORS

Assignment Operators

Assignment operators are used to assign values to variables. They include =, +=, -=, *=, /=, and more.

Examples:

```
x = 10

x += 5  # Equivalent to x = x + 5

print(x)  # Output: 15
```

Bitwise Operators

Bitwise operators work at the binary level to manipulate bits within integers. They include **AND (&), OR (|), XOR (^), NOT (~), left shift (<<), and right shift (>>).**

Examples:

```
x = 5  # Binary: 0101

y = 3  # Binary: 0011

print(x & y)  # Output: 1 (Binary: 0001)

print(x | y)  # Output: 7 (Binary: 0111)
```

Bitwise operations are useful in low-level programming and cryptographic applications.

CHAPTER 5: EXPRESSIONS IN PROGRAMMING

Expressions are crucial in programming as they define computations and logic. They include **constant expressions, arithmetic expressions, logical expressions, and conditional expressions.**

Examples of Expressions:

```
# Arithmetic Expression
result = (10 + 5) * 3


# Logical Expression
is_valid = (10 > 5) and (5 < 8)


# Conditional Expression
max_value = 10 if (10 > 5) else 5
```

Expressions make code dynamic and help in the efficient execution of operations based on conditions.

CHAPTER 6: CASE STUDY - EVALUATING STUDENT GRADES

A school wants to compute final grades for students based on their scores. The grading system follows these conditions:

- If score >= 90, grade = "A"
- If score >= 80, grade = "B"
- If score >= 70, grade = "C"
- If score >= 60, grade = "D"
- Else, grade = "F"

Implementation:

```
score = int(input("Enter student score: "))

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"
print("Student Grade:", grade)
```

This case study demonstrates the use of **relational and logical operators** to evaluate student performance.

CHAPTER 7: EXERCISES

Exercise 1: Arithmetic Operations

Write a Python program to:

1. Accept two numbers as input.
2. Perform addition, subtraction, multiplication, and division.
3. Display the results.

Exercise 2: Logical Operators

Create a program that checks whether a given number is within a specified range (10 to 50) using logical operators.

Exercise 3: Expression Evaluation

Write a program to evaluate the expression (a + b) * (c - d) / e, where the user provides values for a, b, c, d, and e.

INPUT AND OUTPUT OPERATIONS IN C (PRINTF, SCANF, GETS, PUTS)

INTRODUCTION TO INPUT AND OUTPUT OPERATIONS IN C

What is Input and Output (I/O) in C?

Input and Output (I/O) operations allow programs to **communicate with users** by accepting data (input) and displaying results (output). The standard **input device** is the **keyboard**, and the standard **output device** is the **screen (monitor)**.

Why is I/O Important?

- **Interactivity**: Users can enter data dynamically.
- **Flexibility**: Programs can handle various inputs at runtime.
- **Efficiency**: Well-structured I/O improves user experience and program functionality.

C provides several **library functions** for handling I/O, with the most commonly used functions being:
✓ printf() – Outputs formatted text to the screen.
✓ scanf() – Accepts user input from the keyboard.
✓ gets() – Reads an entire line of input.
✓ puts() – Displays a string with automatic newline.

This chapter explores **how to use these functions effectively**, with practical examples, exercises, and a case study.

USING PRINTF() FOR OUTPUT

What is printf()?

The printf() function is used to **display text and variables** on the screen. It is defined in the <stdio.h> library and supports **formatted output**.

Syntax:

printf("format_string", variable1, variable2, ...);

- **format_string** – The text to display, with placeholders for variables.
- **variable1, variable2** – Variables whose values will replace placeholders.

Format Specifiers in printf()

Specifier	**Description**	**Example**
%d	Integer	printf("%d", 25);
%f	Floating-point	printf("%f", 3.14);
%c	Character	printf("%c", 'A');
%s	String	printf("%s", "Hello");
%lf	Double	printf("%lf", 3.141592);

Example: Using printf()

```
#include <stdio.h>


int main() {

   int age = 25;

   float price = 99.99;
```

```
    char grade = 'A';


    printf("Age: %d\n", age);

    printf("Price: %.2f\n", price);

    printf("Grade: %c\n", grade);


    return 0;

}
```

Output:

```
Age: 25

Price: 99.99

Grade: A
```

*Note: %.2f formats the floating-point number to **2 decimal places**.*

USING SCANF() FOR INPUT

What is scanf()?

The scanf() function allows users to enter values from the keyboard and store them in variables. It also belongs to <stdio.h>.

Syntax:

```
scanf("format_specifier", &variable);
```

- **format_specifier** – Defines the data type of input.

- **&variable** – Address of the variable where input is stored (& is required for most types).

Example: Using scanf() to Accept User Input

```
#include <stdio.h>


int main() {
    int age;
    float salary;

    printf("Enter your age: ");
    scanf("%d", &age);  // Accept integer input

    printf("Enter your salary: ");
    scanf("%f", &salary);  // Accept float input

    printf("You are %d years old and earn %.2f\n", age, salary);

    return 0;
}
```

Sample Input:

```
Enter your age: 30
```

Enter your salary: 55000.75

Output:

You are 30 years old and earn 55000.75

Common Mistakes with scanf()

❌ **Forgetting & (Address-of Operator)**

scanf("%d", age); // Wrong (missing &)

scanf("%d", &age); // Correct

❌ **Not Handling Spaces Properly**

char name[20];

scanf("%s", name); // Only reads the first word (stops at space)

💡 Use gets() or fgets() instead for **multi-word input**.

USING GETS() FOR STRING INPUT

What is gets()?

The gets() function allows users to input a **whole line of text (including spaces)**. It stops reading input when the user presses **Enter**.

Syntax:

gets(variable);

- variable must be a **character array (string)**.

Example: Accepting Full Name Input

```
#include <stdio.h>


int main() {
    char name[50];


    printf("Enter your full name: ");
    gets(name);  // Reads full name including spaces


    printf("Hello, %s!\n", name);
    return 0;
}
```

Sample Input:

```
Enter your full name: John Doe
```

Output:

```
Hello, John Doe!
```

Warning: gets() is Unsafe

The gets() function does **not limit input size**, which may cause buffer overflow. **Use fgets() instead**.

```
fgets(name, 50, stdin);
```

💡 fgets() ensures safe input by limiting the number of characters read.

Using puts() for String Output

What is puts()?

The puts() function is used to **display a string** and **automatically moves to the next line**.

Syntax:

```
puts(variable);
```

- It is simpler than printf() for string output.

Example: Displaying a String

```
#include <stdio.h>


int main() {
   char message[] = "Welcome to C Programming!";
   puts(message);  // Prints message with a newline


   return 0;
}
```

Output:

```
Welcome to C Programming!
```

Difference Between printf() and puts()

Function	Purpose	Moves to New Line?

printf("%s", str);	Prints string without automatic newline	No
puts(str);	Prints string and moves to a new line	Yes

Case Study: Creating a Student Registration System

Problem Statement:

A university needs a **simple student registration system** where students can enter their **name, age, and department**, and the system stores and displays the information.

Solution Approach:

1. Use gets() or fgets() to accept **full name input**.
2. Use scanf() for **numerical values** (age).
3. Use puts() for **output messages**.

Implementation:

```
#include <stdio.h>


int main() {

   char name[50], department[30];

   int age;


   printf("Enter your full name: ");
```

```
    fgets(name, 50, stdin);


    printf("Enter your age: ");

    scanf("%d", &age);


    printf("Enter your department: ");

    scanf("%s", department);  // Accepts single-word input


    printf("\n--- Student Registration Complete ---\n");

    puts("Student Details:");

    printf("Name: %s", name);  // fgets includes newline, so no extra \n
needed

    printf("Age: %d\n", age);

    printf("Department: %s\n", department);


    return 0;

}
```

Sample Input:

```
Enter your full name: Alice Johnson

Enter your age: 20

Enter your department: ComputerScience
```

Output:

```
--- Student Registration Complete ---

Student Details:

Name: Alice Johnson

Age: 20

Department: ComputerScience
```

Exercise

1. **Write a program** that asks the user for their name, age, and city, then prints a greeting message.
2. **Modify the student registration program** to allow multi-word department input.
3. **Write a program** that accepts two numbers from the user and prints their sum.
4. **Use gets() and puts()** to input and display a user's favorite quote.
5. **Write a program** that prompts the user to enter their favorite book title and author, then displays them.

CONCLUSION

Mastering **input and output functions (printf, scanf, gets, puts)** is crucial for writing interactive C programs.

✓ printf() – Displays formatted output.

✓ scanf() – Accepts user input.

✓ gets() – Reads multi-word input (use fgets() for safety).

✓ puts() – Displays strings with automatic newlines.

CONTROL FLOW STATEMENTS (IF-ELSE, SWITCH-CASE, LOOPS)

CHAPTER 1: INTRODUCTION TO CONTROL FLOW STATEMENTS

Control flow statements dictate the execution sequence of statements in a program. Without control flow, a program would execute linearly, executing one statement after another. However, real-world programming requires decision-making and looping mechanisms to handle different scenarios dynamically.

Control flow statements include:

1. **Conditional Statements**: These include if-else and switch-case, allowing programs to make decisions based on conditions.
2. **Looping Statements**: These include for, while, and do-while loops, enabling repetition of code blocks.
3. **Jump Statements**: These include break, continue, and return, controlling loop execution.

By effectively using control flow statements, programmers can create flexible and efficient applications.

CHAPTER 2: CONDITIONAL STATEMENTS

if-else Statement

The if-else statement is used to execute code based on a condition. If the condition is True, the if block executes; otherwise, the else block executes.

Syntax:

```
if condition:
    # Execute if condition is true
else:
    # Execute if condition is false
```

Example:

```
age = int(input("Enter your age: "))
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

Nested if-else

A nested if-else structure contains one or more if-else statements inside another.

Example:

```
num = int(input("Enter a number: "))
if num > 0:
    print("Positive number")
    if num % 2 == 0:
        print("Even number")
    else:
        print("Odd number")
else:
    print("Negative number or zero")
```

Nested structures allow complex decision-making by evaluating multiple conditions sequentially.

Switch-Case Statement

Python lacks a built-in switch-case statement, but it can be implemented using dictionaries.

Example:

```
def switch_case(choice):
    switch = {
        1: "Option 1 selected",
        2: "Option 2 selected",
        3: "Option 3 selected"
    }
    return switch.get(choice, "Invalid option")


choice = int(input("Enter your choice (1-3): "))
print(switch_case(choice))
```

In languages like C, C++, and Java, switch-case is used for handling multiple conditions efficiently.

CHAPTER 3: LOOPING STATEMENTS

Loops allow executing a block of code multiple times until a condition is met.

for Loop

A for loop iterates over a sequence (list, tuple, string, or range).

Syntax:

```
for variable in sequence:

   # Loop body
```

Example:

```
for i in range(1, 6):

   print("Iteration:", i)
```

while Loop

A while loop executes as long as the condition remains True.

Syntax:

```
while condition:

   # Loop body
```

Example:

```
num = 5

while num > 0:

   print("Countdown:", num)
```

```
    num -= 1
```

do-while Loop (Python Equivalent)

Python lacks do-while, but it can be simulated using while True and break.

Example:

```
while True:
    num = int(input("Enter a positive number: "))
    if num > 0:
        break
print("You entered:", num)
```

CHAPTER 4: JUMP STATEMENTS

Jump statements alter the flow of loops.

break Statement

Exits the loop immediately.

Example:

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)
```

continue Statement

Skips the current iteration and moves to the next.

Example:

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

return Statement

Used to return a value from a function.

Example:

```
def square(num):
    return num * num


print(square(4))
```

Chapter 5: Case Study - Student Grading System

A university grading system categorizes students based on their scores.

Problem Statement:

A student's grade is determined as follows:

- **A**: 90-100
- **B**: 80-89

- **C**: 70-79
- **D**: 60-69
- **F**: Below 60

Implementation:

```
score = int(input("Enter student score: "))


if score >= 90:
   grade = "A"
elif score >= 80:
   grade = "B"
elif score >= 70:
   grade = "C"
elif score >= 60:
   grade = "D"
else:
   grade = "F"


print("Student Grade:", grade)
```

This program demonstrates if-else control flow to determine student performance.

CHAPTER 6: EXERCISES

Exercise 1: if-else Condition

Write a Python program that checks whether a given number is positive, negative, or zero using if-else.

Exercise 2: Looping Mechanisms

Write a program that prints the multiplication table of a user-input number using for and while loops.

Exercise 3: Implementing Switch-Case

Implement a simple calculator using a dictionary-based switch-case mechanism to perform addition, subtraction, multiplication, and division.

HANDS-ON CODING PRACTICE IN C

INTRODUCTION TO HANDS-ON CODING IN C

What is Hands-on Coding?

Hands-on coding involves actively writing, compiling, and debugging C programs to develop a **deep understanding of programming concepts**. Unlike passive learning, hands-on coding helps in:
✓ **Reinforcing theoretical knowledge**
✓ **Enhancing problem-solving skills**
✓ **Improving debugging and troubleshooting ability**
✓ **Building confidence in programming**

Why Hands-on Practice is Important?

- **Better Retention**: Writing code helps in long-term memory retention.
- **Problem-Solving Ability**: Encourages logical thinking and creativity.
- **Career Readiness**: Helps in preparing for technical interviews and real-world projects.

This chapter covers **a structured set of coding exercises**, starting from **basic syntax** to **advanced problem-solving**, with examples, exercises, and case studies.

Basic Hands-on Coding Exercises

1. PRINTING "HELLO, WORLD" (YOUR FIRST C PROGRAM)

One of the most fundamental programs in any language is printing text to the screen.

Example: Hello, World Program

```
#include <stdio.h>


int main() {
   printf("Hello, World!\n");
   return 0;
}
```

✓ **Concepts Covered**: Syntax, printf(), main() function

---

2. Using Variables and Data Types

Example: Declaring and Printing Variables

```
#include <stdio.h>


int main() {
   int age = 25;
   float price = 99.99;
   char grade = 'A';


   printf("Age: %d\n", age);
   printf("Price: %.2f\n", price);
   printf("Grade: %c\n", grade);
```

```
    return 0;
}
```

✓ **Concepts Covered**: Variables, Data Types, printf()

---

3. Taking User Input (scanf) and Performing Arithmetic Operations

Example: Add Two Numbers

```
#include <stdio.h>

int main() {
    int num1, num2, sum;

    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);

    sum = num1 + num2;
    printf("Sum: %d\n", sum);

    return 0;
}
```

✓ **Concepts Covered**: scanf(), Arithmetic Operations

---

4. Using Conditional Statements (if-else)

Example: Check if a Number is Even or Odd

```
#include <stdio.h>

int main() {
  int num;

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

  if (num % 2 == 0)
    printf("The number is Even\n");
  else
    printf("The number is Odd\n");

  return 0;
}
```

✓ **Concepts Covered**: If-else statements, Modulus Operator %

---

5. Using Loops (for, while, do-while)

Example: Print Numbers from 1 to 10 Using a for Loop

```
#include <stdio.h>


int main() {
    for (int i = 1; i <= 10; i++) {
        printf("%d ", i);
    }
    return 0;
}
```

✓ **Concepts Covered**: for loop, Iteration

Intermediate Hands-on Coding Exercises

6. Using Functions for Modular Code

Example: Function to Find the Square of a Number

```
#include <stdio.h>


// Function declaration
int square(int);


int main() {
    int num;

    printf("Enter a number: ");

    scanf("%d", &num);


    printf("Square: %d\n", square(num));

    return 0;

}


// Function definition

int square(int x) {

    return x * x;

}
```

✓ **Concepts Covered**: Functions, Function Parameters, Return Values

---

7. Using Arrays

Example: Find the Largest Element in an Array

```
#include <stdio.h>


int main() {

    int arr[5] = {10, 20, 5, 30, 25};
    int max = arr[0];


    for (int i = 1; i < 5; i++) {
        if (arr[i] > max)
            max = arr[i];
    }


    printf("Largest number: %d\n", max);
    return 0;
}
```

✓ **Concepts Covered**: Arrays, Looping through Arrays

---

8. String Handling

Example: Reverse a String

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


int main() {
    char str[50], temp;
    int i, j;
    printf("Enter a string: ");
    scanf("%s", str);

    j = strlen(str) - 1;
    for (i = 0; i < j; i++, j--) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }

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

✓ **Concepts Covered**: Strings, strlen(), Swapping

Advanced Hands-on Coding Challenges

9. File Handling in C

Example: Writing and Reading from a File

```
#include <stdio.h>

int main() {
    FILE *fptr;
    char data[50];

    // Writing to file
    fptr = fopen("example.txt", "w");
    fprintf(fptr, "C Programming is powerful!");
    fclose(fptr);

    // Reading from file
    fptr = fopen("example.txt", "r");
    fgets(data, 50, fptr);
    printf("File Content: %s\n", data);
    fclose(fptr);

    return 0;
}
```

✓ **Concepts Covered**: File Handling, fopen(), fprintf(), fgets(), fclose()

---

10. Data Structures: Linked List

Example: Singly Linked List (Insert & Display)

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

   int data;

   struct Node* next;

};


void display(struct Node* head) {

   struct Node* temp = head;

   while (temp != NULL) {

      printf("%d -> ", temp->data);

      temp = temp->next;

   }

   printf("NULL\n");

}


int main() {

   struct Node* head = (struct Node*)malloc(sizeof(struct Node));

   struct Node* second = (struct Node*)malloc(sizeof(struct Node));
```

```
  head->data = 10;

  head->next = second;


  second->data = 20;

  second->next = NULL;


  display(head);

  return 0;

}
```

✓ **Concepts Covered**: Dynamic Memory Allocation, Linked List

---

Case Study: Building a Simple Contact Management System

Problem Statement:

A company wants a **Contact Management System** to store names and phone numbers, allowing users to add and view contacts.

Solution Approach:

1. **Use a structure** to store name and phone number.
2. **Use file handling** to save and retrieve contacts.
3. **Use loops and conditionals** for user interaction.

Implementation:

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

```
struct Contact {
    char name[50];
    char phone[15];
};

int main() {
    struct Contact c1;

    printf("Enter Name: ");
    scanf("%s", c1.name);

    printf("Enter Phone Number: ");
    scanf("%s", c1.phone);

    printf("\n--- Contact Saved ---\n");
    printf("Name: %s\nPhone: %s\n", c1.name, c1.phone);

    return 0;
}
```

Outcome:

✓ Practical use of structures and user input
✓ Enhances problem-solving skills

CONCLUSION

Hands-on coding in C is **essential** for mastering programming skills. Practicing **basic to advanced programs** builds confidence and prepares for **real-world projects and technical interviews**.

Assignment Solution: Calculating the Area of a Rectangle, Circle, and Triangle

Objective

The goal of this assignment is to write a Python program that calculates the area of three different geometric shapes:

1. **Rectangle** – using length and width.
2. **Circle** – using radius.
3. **Triangle** – using base and height.

The program should take user inputs and display the calculated areas.

Step-by-Step Guide

Step 1: Understand the Formulas

Before writing the program, we need to understand the mathematical formulas for each shape.

1. **Rectangle**

Area=Length×Width\text{Area} = \text{Length} \times \text{Width}

2. **Circle**

Area=π×Radius2\text{Area} = \pi \times \text{Radius}^2

Where π≈3.1416\pi \approx 3.1416.

3. **Triangle**

Area=12×Base×Height\text{Area} = \frac{1}{2} \times \text{Base} \times \text{Height}

Step 2: Define a Python Program

We will use the input() function to take user inputs, perform the calculations, and display the results.

```
import math  # Importing math module for using pi


def calculate_rectangle_area(length, width):

   return length * width


def calculate_circle_area(radius):

   return math.pi * radius ** 2


def calculate_triangle_area(base, height):

   return 0.5 * base * height


# Taking user input

print("Choose the shape to calculate the area:")

print("1. Rectangle")

print("2. Circle")

print("3. Triangle")
```

```
choice = int(input("Enter the number corresponding to your choice: "))


if choice == 1:

   length = float(input("Enter the length of the rectangle: "))

   width = float(input("Enter the width of the rectangle: "))

   area = calculate_rectangle_area(length, width)

   print(f"The area of the rectangle is: {area:.2f}")


elif choice == 2:

   radius = float(input("Enter the radius of the circle: "))

   area = calculate_circle_area(radius)

   print(f"The area of the circle is: {area:.2f}")


elif choice == 3:

   base = float(input("Enter the base of the triangle: "))

   height = float(input("Enter the height of the triangle: "))

   area = calculate_triangle_area(base, height)

   print(f"The area of the triangle is: {area:.2f}")
```

```
else:
    print("Invalid choice! Please enter 1, 2, or 3.")
```

STEP 3: EXPLANATION OF THE CODE

1. Importing Required Libraries

```
import math
```

We import the math module to use math.pi for circle area calculation.

2. Defining Functions

Each shape's area is calculated using a separate function:

- calculate_rectangle_area(length, width)
- calculate_circle_area(radius)
- calculate_triangle_area(base, height)

This makes the program modular and easy to understand.

3. Taking User Input

The program presents a menu for users to choose a shape:

```
print("Choose the shape to calculate the area:")

print("1. Rectangle")

print("2. Circle")

print("3. Triangle")
```

The user inputs their choice, and based on it, the program prompts for the required values.

4. Using Conditional Statements

```
if choice == 1:
    length = float(input("Enter the length of the rectangle: "))
    width = float(input("Enter the width of the rectangle: "))
    area = calculate_rectangle_area(length, width)
    print(f"The area of the rectangle is: {area:.2f}")
```

Each shape's area is calculated using the corresponding function, and the result is displayed.

5. Handling Invalid Choices

If the user enters an invalid choice, the program prints an error message:

```
else:
    print("Invalid choice! Please enter 1, 2, or 3.")
```

STEP 4: EXAMPLE OUTPUTS

Case 1: Rectangle Calculation

```
Choose the shape to calculate the area:

1. Rectangle

2. Circle

3. Triangle

Enter the number corresponding to your choice: 1

Enter the length of the rectangle: 10
```

Enter the width of the rectangle: 5

The area of the rectangle is: 50.00

Case 2: Circle Calculation

Choose the shape to calculate the area:

1. Rectangle

2. Circle

3. Triangle

Enter the number corresponding to your choice: 2

Enter the radius of the circle: 7

The area of the circle is: 153.94

Case 3: Triangle Calculation

Choose the shape to calculate the area:

1. Rectangle

2. Circle

3. Triangle

Enter the number corresponding to your choice: 3

Enter the base of the triangle: 6

Enter the height of the triangle: 4

The area of the triangle is: 12.00

STEP 5: ADDITIONAL EXERCISE

Modify the program to:

- Allow the user to calculate areas for multiple shapes in one execution.
- Include input validation (ensure values are positive numbers).
- Extend the program to include additional shapes like square and trapezium.

Assignment Solution: Develop a Number Guessing Game Using Loops and Conditional Statements

Objective

The objective of this assignment is to **develop a simple number guessing game** using **loops and conditional statements** in **C programming**. The program will generate a **random number**, and the user will have multiple attempts to guess it correctly. After each guess, the program will provide hints to guide the user.

Step 1: Understanding the Game Logic

1. The program generates a **random number** within a specified range (e.g., 1-100).
2. The user inputs a guess.
3. The program checks if the guess is **correct, too high, or too low**.
4. If the guess is incorrect, the user is prompted to try again.
5. The loop continues until the user guesses the number correctly.
6. Once the correct number is guessed, the program displays the number of attempts taken.

Step 2: Setting Up the C Programming Environment

Before starting, ensure you have a **C compiler installed**. You can use:

- **Windows**: Install **MinGW** and use gcc compiler.
- **Linux/macOS**: Install GCC using:
- sudo apt install gcc   # Ubuntu/Debian
- sudo yum install gcc   # CentOS/RHEL

Step 3: Writing the Number Guessing Game Code

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>


int main() {

   int number, guess, attempts = 0;


   // Seed the random number generator

   srand(time(0));

   number = rand() % 100 + 1;  // Generates a random number between 1 and 100


   printf("Welcome to the Number Guessing Game!\n");

   printf("Try to guess the number between 1 and 100.\n");
```

```
    // Loop until the user guesses correctly
    do {
        printf("Enter your guess: ");
        scanf("%d", &guess);
        attempts++;  // Increment attempt count

        // Check if guess is correct
        if (guess > number) {
            printf("Too high! Try again.\n");
        } else if (guess < number) {
            printf("Too low! Try again.\n");
        } else {
            printf("Congratulations! You guessed the number %d in %d attempts.\n", number, attempts);
        }

    } while (guess != number);  // Repeat until correct guess

    return 0;
}
```

Step 4: Explanation of the Code

1. Random Number Generation

```
srand(time(0));

number = rand() % 100 + 1;
```

- The srand(time(0)) function ensures a different random number is generated each time the program runs.
- rand() % 100 + 1 generates a number between **1 and 100**.

2. Taking User Input with scanf()

```
printf("Enter your guess: ");

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

- The program **prompts the user** for a guess and stores it in the guess variable.

3. Implementing the Loop (do-while loop)

```
do {

  // Code for checking guess

} while (guess != number);
```

- The do-while loop **ensures at least one attempt** before checking the condition.
- It **keeps looping** until the user guesses the correct number.

4. Conditional Statements to Give Hints

```
if (guess > number) {
    printf("Too high! Try again.\n");
} else if (guess < number) {
    printf("Too low! Try again.\n");
} else {
    printf("Congratulations! You guessed the number %d in %d
attempts.\n", number, attempts);
}
```

- If the guess is **greater than** the number, the program displays "Too high!".
- If the guess is **less than** the number, the program displays "Too low!".
- If the guess **matches**, the program **ends the loop** and displays the success message.

STEP 5: TESTING THE PROGRAM

Test Case 1: Correct Guess in 3 Attempts

Input:

```
Enter your guess: 50
Too high! Try again.
Enter your guess: 30
Too low! Try again.
Enter your guess: 40
```

Congratulations! You guessed the number 40 in 3 attempts.

Test Case 2: Correct Guess in 1 Attempt

Input:

Enter your guess: 72

Congratulations! You guessed the number 72 in 1 attempt.

Test Case 3: Multiple Incorrect Guesses

Input:

Enter your guess: 20

Too low! Try again.

Enter your guess: 80

Too high! Try again.

Enter your guess: 60

Too high! Try again.

Enter your guess: 45

Too low! Try again.

Enter your guess: 55

Congratulations! You guessed the number 55 in 5 attempts.

STEP 6: ENHANCING THE GAME (OPTIONAL FEATURES)

To improve the user experience, you can:

1. Add a Limited Number of Attempts

Modify the do-while loop to **restrict the number of guesses** (e.g., max 5 attempts).

```
if (attempts >= 5) {
    printf("Game Over! The correct number was %d\n", number);
    break;
}
```

2. Ask the User If They Want to Play Again

After finishing the game, allow the user to **restart** instead of exiting.

```
char playAgain;
printf("Do you want to play again? (y/n): ");
scanf(" %c", &playAgain);

if (playAgain == 'y' || playAgain == 'Y') {
    main();  // Restart game
}
```

3. Provide Difficulty Levels

Allow users to **choose a range** for the random number (e.g., 1-50 for easy, 1-500 for hard).

```
printf("Choose difficulty (1: Easy, 2: Hard): ");
int difficulty;
scanf("%d", &difficulty);
```

```
if (difficulty == 1) {
   number = rand() % 50 + 1;  // Easy Mode
} else {
   number = rand() % 500 + 1;  // Hard Mode
}
```

STEP 7: EXERCISE QUESTIONS FOR PRACTICE

1. **Modify the program** to keep track of the **best score (fewest attempts)**.
2. **Add a timer** to measure how long the user takes to guess the number.
3. **Modify the game** to accept a **range input** from the user instead of a fixed range (1-100).
4. **Enhance user experience** by adding colors and animations using system("cls") and sleep() functions.
5. **Convert the game** into a multiplayer version where **two players compete** to guess numbers in the fewest attempts.

STEP 8: CONCLUSION

By following this guide, we successfully developed a **Number Guessing Game** using:

✓ **Loops (do-while)** for repeated guessing
✓ **Conditional Statements (if-else)** for hints

✓ **Random Number Generation (rand())** for unpredictability
✓ **User Input (scanf())** for interactive gameplay

This project **reinforces core C programming concepts** and serves as a great foundation for building **game logic, interactive programs, and problem-solving skills**.

ASSIGNMENT SOLUTION: IMPLEMENTING A SIMPLE CALCULATOR IN C

Objective

The objective of this assignment is to implement a simple calculator in C that can perform basic arithmetic operations such as:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)

The program should prompt the user to input two numbers and an operator, perform the calculation, and display the result.

Step-by-Step Guide

STEP 1: UNDERSTAND THE REQUIREMENTS

The calculator should:

1. Take two numbers as input.
2. Ask the user for the operation (+, -, *, /).
3. Perform the calculation based on the chosen operator.
4. Display the result.

STEP 2: WRITE THE C PROGRAM

Complete Code for the Simple Calculator

```
#include <stdio.h>


int main() {
   double num1, num2, result;
   char operator;


   // Display options to the user
   printf("Simple Calculator in C\n");
   printf("Available operations: + (Addition), - (Subtraction), * (Multiplication), / (Division)\n");


   // Taking user input
   printf("Enter first number: ");
   scanf("%lf", &num1);


   printf("Enter an operator (+, -, *, /): ");
   scanf(" %c", &operator);  // Space before %c to ignore any newline character from previous input


   printf("Enter second number: ");
   scanf("%lf", &num2);
```

```
// Performing the calculation based on the chosen operator
switch (operator) {
  case '+':
    result = num1 + num2;
    printf("Result: %.2lf + %.2lf = %.2lf\n", num1, num2, result);
    break;
  case '-':
    result = num1 - num2;
    printf("Result: %.2lf - %.2lf = %.2lf\n", num1, num2, result);
    break;
  case '*':
    result = num1 * num2;
    printf("Result: %.2lf * %.2lf = %.2lf\n", num1, num2, result);
    break;
  case '/':
    if (num2 != 0) {  // Prevent division by zero
      result = num1 / num2;
      printf("Result: %.2lf / %.2lf = %.2lf\n", num1, num2, result);
    } else {
      printf("Error: Division by zero is not allowed.\n");
```

```
            }
            break;
        default:
            printf("Invalid operator! Please enter +, -, *, or /.\n");
    }

    return 0;
}
```

STEP 3: EXPLANATION OF THE CODE

1. Include Necessary Header Files

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

The stdio.h library is included to enable input (scanf) and output (printf) operations.

2. Declare Variables

```
double num1, num2, result;
char operator;
```

- num1 and num2 store user input numbers.
- result stores the computed value.
- operator stores the arithmetic operation (+, -, *, /).

3. Take User Input

```
printf("Enter first number: ");

scanf("%lf", &num1);
```

- %lf is used to read a double (floating-point number).
- Similarly, input is taken for operator and num2.

4. Use a switch-case for Operation Handling

```
switch (operator) {

  case '+':

    result = num1 + num2;

    printf("Result: %.2lf + %.2lf = %.2lf\n", num1, num2, result);

    break;
```

- Each case performs the respective operation.
- %.2lf ensures the result is displayed with two decimal places.

5. Handle Division by Zero

```
case '/':

  if (num2 != 0) {

    result = num1 / num2;

    printf("Result: %.2lf / %.2lf = %.2lf\n", num1, num2, result);

  } else {

    printf("Error: Division by zero is not allowed.\n");

  }

  break;
```

- Before performing division, the program checks if num2 is zero to prevent errors.

6. Handle Invalid Input

```
default:

   printf("Invalid operator! Please enter +, -, *, or /.\n");
```

- If the user enters an incorrect operator, an error message is displayed.

STEP 4: EXAMPLE OUTPUTS

Case 1: Addition

```
Simple Calculator in C

Available operations: + (Addition), - (Subtraction), * (Multiplication),
/ (Division)

Enter first number: 12

Enter an operator (+, -, *, /): +

Enter second number: 5

Result: 12.00 + 5.00 = 17.00
```

Case 2: Division

```
Enter first number: 15

Enter an operator (+, -, *, /): /

Enter second number: 3

Result: 15.00 / 3.00 = 5.00
```

Case 3: Division by Zero Error Handling

```
Enter first number: 10

Enter an operator (+, -, *, /): /

Enter second number: 0

Error: Division by zero is not allowed.
```

Case 4: Invalid Operator

```
Enter first number: 8

Enter an operator (+, -, *, /): ^

Enter second number: 2

Invalid operator! Please enter +, -, *, or /.
```

Step 5: Additional Enhancements

1. **Looping for Multiple Calculations**
    - Modify the program to allow multiple calculations without restarting.
    - Use a while or do-while loop to ask the user if they want to continue.
2. **Error Handling**
    - Ensure the user inputs valid numbers.
    - Handle invalid characters gracefully.
3. **Extend Functionality**

   - Add more mathematical operations like modulus (%), exponentiation (pow()), and square root (sqrt()).
   - Implement an **Advanced Calculator** with trigonometric functions.

CONCLUSION

This assignment successfully demonstrates:

- **Taking user input** using scanf().
- **Processing arithmetic operations** using a switch-case structure.
- **Handling errors like division by zero**.
- **Displaying results with appropriate formatting**.