2025 Edition

**Kushal Prasad Joshi**

kushalprasadjoshi@gmail.com

**Programming in C**

Course by Pokhara University

# Preface

The **Programming in C** handbook is designed to provide a comprehensive guide to the fundamentals of C programming, based on the **CMP 124** course offered by **Pokhara University**. This handbook serves as a valuable resource for students, beginners, and enthusiasts who wish to gain a solid understanding of C programming concepts and techniques.

This handbook covers key topics such as programming fundamentals, control structures, functions, arrays, pointers, and file handling, along with real-world examples and exercises. It aims to bridge the gap between theoretical knowledge and practical application, making the learning process engaging and effective.

The handbook is structured to facilitate self-paced learning, with each chapter presenting concepts in a clear and concise manner. The examples and exercises are designed to enhance understanding and encourage hands-on practice. Additionally, flowcharts, diagrams, and well-documented code snippets are included to aid visual learners and improve comprehension.

This work is an independent effort by **Kushal Prasad Joshi**, inspired by the curriculum of **Pokhara University**, and is intended to support students in their academic journey. While based on the university's course structure, this handbook is not officially affiliated with or endorsed by **Pokhara University**.

I hope this handbook serves as a helpful guide in your programming journey, equipping you with the skills and confidence to tackle challenges in C programming. Your feedback and suggestions are always welcome to improve future editions of this handbook.

Happy Learning!

**Kushal Prasad Joshi**

kushalprasadjoshi@gmail.com

*Author*

Table of Contents

[Preface 1](#_Toc190868688)

[Unit 1 - Programming Languages and Problem Solving 4](#_Toc190868689)

[Computer Architecture and Hardware 4](#_Toc190868690)

[Computer Software 5](#_Toc190868691)

[1. System Software 5](#_Toc190868692)

[2. Application Software 5](#_Toc190868693)

[Computer Programming 6](#_Toc190868694)

[Importance of Programming 6](#_Toc190868695)

[Software Development 6](#_Toc190868696)

[Programming Languages 6](#_Toc190868697)

[Types of Programming Languages 7](#_Toc190868698)

[Generation of Programming Languages 7](#_Toc190868699)

[Programming Approaches 8](#_Toc190868700)

[Code Conversion 8](#_Toc190868701)

[Algorithms 9](#_Toc190868702)

[Characteristics of a Good Algorithm 10](#_Toc190868703)

[Flowcharts 10](#_Toc190868704)

[Flowcharts Symbols 10](#_Toc190868705)

[Exercises 12](#_Toc190868706)

[Unit 2 – Introduction to c 27](#_Toc190868707)

[Introduction 27](#_Toc190868708)

[Key features 27](#_Toc190868709)

[Importance 27](#_Toc190868710)

[Writing first program 27](#_Toc190868711)

[flow of a c program 28](#_Toc190868712)

[Components of c program 28](#_Toc190868713)

[keywords 28](#_Toc190868714)

[Identifiers 29](#_Toc190868715)

[constants 29](#_Toc190868716)

[string literals 30](#_Toc190868717)

# Unit 1 - Programming Languages and Problem Solving

## Computer Architecture and Hardware

A diagram of a computer hardware system

AI-generated content may be incorrect.

Computer Architecture refers to the design and organization of a computer's core components, including the central processing unit (CPU), memory, input/output (I/O) devices, and storage. Understanding computer architecture is crucial for optimizing software performance and making informed decisions about hardware requirements.

**Central Processing Unit (CPU):** The CPU is the brain of the computer, responsible for executing instructions and performing calculations. It consists of the arithmetic logic unit (ALU), control unit (CU), and registers. The ALU handles arithmetic and logical operations, while the CU directs the flow of data and instructions within the system.

**Memory (Primary Memory):** Memory is used to store data and instructions that the CPU needs to access quickly. There are two main types of memory: volatile memory (RAM) and non-volatile memory (ROM). RAM is used for temporary storage while the computer is running, whereas ROM retains data even when the computer is turned off.

**Input/Output (I/O) Devices:** I/O devices allow the computer to interact with the external world. Input devices, such as keyboards and mice, enable users to provide data to the computer. Output devices, such as monitors and printers, allow the computer to present information to users.

**Storage (Secondary Memory):** Storage devices, such as hard drives and solid-state drives (SSDs), provide long-term data storage. They retain data even when the computer is powered off and are essential for storing the operating system, applications, and user data.

Understanding these components and their interactions is fundamental to both hardware and software development, as it impacts system performance, efficiency, and capabilities.

## Computer Software

Computer software refers to the collection of programs, procedures, and data that instruct a computer on how to perform specific tasks. Software can be broadly categorized into two types: system software and application software.

### System Software

System software serves as the foundation for application software, managing the hardware and basic system operations. It includes:

* **Operating System (OS):** These are essential for managing hardware resources and providing a user interface. Examples include Windows, macOS, Linux, and Android.
* **Device Drivers:** These programs allow the operating system to communicate with hardware devices like printers, graphics cards, and network adapters.
* **Utility Programs:** These perform maintenance tasks such as disk cleanup, antivirus scanning, and file management.

### Application Software

Application software is designed to help users perform specific tasks or applications. It includes:

* **Productivity Software:** Tools like word processors, spreadsheets, and presentation software (e.g., Microsoft Office, Google Workspace).
* **Media Software:** Programs for creating and editing multimedia content, such as photo editors, video editors, and music production software (e.g., Adobe Photoshop, Final Cut Pro).
* **Web Browsers:** Applications for accessing and navigating the internet (e.g., Google Chrome, Mozilla Firefox).
* **Games:** Software designed for entertainment and recreation.

## Computer Programming

Process of giving commands and instructions to a computer to perform some tasks. It involves writing code in a programming language, which is then translated into machine language so that the computer can understand and execute. Programming is essential for creating software applications, automating tasks, and solving complex problems efficiently.

### Importance of Programming

Programming is a fundamental skill in the modern world. It enables the creation of software that powers everything from simple applications to complex systems. Understanding programming concepts helps with logical thinking, problem-solving, and the ability to automate repetitive tasks. It also opens numerous career opportunities in various fields such as software development, data science, artificial intelligence, and more.

### Software Development

The process of creating software involves several stages, including requirements analysis, design, coding, testing, and maintenance. Software development methodologies, such as Agile and Waterfall, provide frameworks for managing these stages effectively.

Understanding computer software is essential for leveraging technology to solve problems, enhance productivity, and create innovative solutions in various domains.

## Programming Languages

Programming languages are formal languages comprising a set of instructions that produce various kinds of output. They are used to implement algorithms and control the behavior of machines. Each programming language has its own syntax and semantics, which define the structure and meaning of the code written in that language.

### Types of Programming Languages

* **Low Level Languages:** These include machine language and assembly language, which are closer to the hardware and provide little or no abstraction from the computer's instruction set architecture.
* **High Level Languages:** These are more abstract and closer to human languages, making them easier to read, write, and maintain. Examples include Python, Java, C++, and JavaScript.
* **Scripting Languages:** These are often used for automating tasks and writing small programs. Examples include Python, Ruby, and Perl.
* **Domain Specific Languages:** These are specialized languages designed for specific tasks. Examples include SQL for database queries and HTML for web page structure.

### Generation of Programming Languages

A diagram of a diagram with Great Pyramid of Giza in the background

AI-generated content may be incorrect.

Programming languages have evolved over time, and they are often categorized into different generations based on their level of abstraction and features.

1. **First Generation (1GL): Machine Language:** The most basic level of programming language, consisting of binary code (0s and 1s) that the computer's CPU can execute directly. It is highly efficient but extremely difficult for humans to read and write.
2. **Second Generation (2GL) / Assembly Language:** A low-level programming language that uses symbolic names (mnemonics) to represent machine-level instructions. It is slightly easier to read and write than machine language but still requires detailed knowledge of the computer's architecture.
3. **Third Generation (3GL) / High-Level Languages:** These languages are more abstract and closer to human languages, making them easier to read, write, and maintain. They include languages like C, C++, Java, and Python. They require a compiler or interpreter to translate the high-level code into machine code.
4. **Fourth Generation (4GL) / Very High-Level Languages:** These languages are designed to be even more user-friendly and efficient for specific tasks. They often include database query languages (e.g., SQL), report generators, and application generators. They allow developers to specify what they want to achieve without detailing how to do it.
5. **Fifth Generation (5GL) / Natural Language Processing Languages:** These languages are based on solving problems using constraints given to the program, rather than using an algorithm written by a programmer. They are often used in artificial intelligence and expert systems. Examples include Prolog and Mercury.

Understanding the evolution of programming languages helps in appreciating the advancements in software development and the increasing ease with which complex problems can be solved.

### Programming Approaches

1. **Procedural Oriented Programming (POP):** Procedural Oriented Programming is a programming paradigm based on the concept of procedure calls, where statements are structured into procedures (also known as routines or functions). It emphasizes a linear top-down approach and the use of procedures to operate on data. POP is characterized by its simplicity and ease of understanding, making it suitable for beginners. Examples of procedural programming languages include C, Pascal, and Fortran.
2. **Object Oriented Programming (OOP):** Object Oriented Programming is a programming paradigm based on the concept of "objects", which can contain data and code to manipulate that data. OOP focuses on the principles of encapsulation, inheritance, and polymorphism. It allows for the creation of modular, reusable, and maintainable code. OOP is widely used in software development for its ability to model real-world entities and relationships. Examples of object-oriented programming languages include Java, C++, Python, and Ruby.

### Code Conversion

Code conversion is the process of translating code written in one form into another form that a computer can execute. This process is essential for running programs written in high-level languages on a computer's hardware. There are three main types of code conversion: assembly, compilation, and interpretation.

* **Assembler:** An assembler is a tool that converts assembly language code into machine code. Assembly language is a low-level programming language that uses symbolic names to represent machine-level instructions. The assembler translates these symbolic names into binary code that the computer's CPU can execute directly. This process is crucial for creating efficient and optimized machine code from human-readable assembly instructions.
* **Compiler:** A compiler is a program that translates code written in a high-level programming language into machine code. The compilation process involves several stages, including lexical analysis, syntax analysis, semantic analysis, optimization, and code generation. The resulting machine code can be executed directly by the computer's CPU. Compilers are used for languages like C, C++, and Java, and they produce highly optimized code that can run efficiently on the target hardware.
* **Interpreter:** An interpreter is a tool that executes code written in a high-level programming language directly, without converting it into machine code beforehand. Instead, the interpreter reads and executes the code line by line at runtime. This approach allows for more flexibility and easier debugging, as changes to the code can be tested immediately. However, interpreted code is generally slower than compiled code because the translation happens at runtime. Examples of interpreted languages include Python, Ruby, and JavaScript.

Understanding these code conversion methods is essential for software development, as it impacts the performance, efficiency, and portability of the resulting programs.

## Algorithms

Algorithms are a sequential approach to performing a task.

**Syntax:**

Step 1: Start

Step 2: Input

Step 3: Processing

Step 4: Output

Step 5: End

**Example:** Algorithm to add two numbers.

Step 1: Start

Step 2: Read a, b

Step 3: c = a + b

Step 4: Write c

Step 5: End

### Characteristics of a Good Algorithm

A good algorithm should possess the following characteristics:

1. **Correctness:** The algorithm should produce the correct output for all possible valid inputs.
2. **Efficiency:** The algorithm should make efficient use of resources, such as time and memory.
3. **Finiteness:** The algorithm should terminate after a finite number of steps.
4. **Definiteness:** Each step of the algorithm should be precisely defined and unambiguous.
5. **Input:** The algorithm should have zero or more inputs.
6. **Output:** The algorithm should have one or more outputs.

Understanding these characteristics helps in designing effective algorithms that can solve problems efficiently and correctly.

## Flowcharts

The diagrammatic representation of algorithm is called flow chart.

### Flowcharts Symbols

Flowcharts use specific symbols to represent different types of actions or steps in a process. Here are some common flowchart symbols:

* **Oval:** Represents the start or end of a process.
* **Rectangle:** Represents a process or operation.
* **Diamond:** Represents a decision point.
* **Parallelogram:** Represents input or output.
* **Arrow:** Indicates the flow of the process.

Symbols of symbols used in flow chart

AI-generated content may be incorrect.

Understanding flowcharts helps in visualizing the steps involved in a process, making it easier to design and communicate algorithms effectively.

**Syntax and Example:**

**A diagram of a function

AI-generated content may be incorrect.**

## Exercises

1. **Write an algorithm and draw a flowchart to calculate the area and perimeter of Rectangle.**

Algorithm:

S1: Start

S2: Read l, b

S3: P = 2 \* (l + b)

S4: A = l \* based

S5: Write P, A

S6: End

Flowchart:

**A diagram of a mathematical equation

AI-generated content may be incorrect.**

1. **Write an algorithm to find number is even or odd and draw a flowchart.**

Algorithm:

S1: Start

S2: Read num

S3: Is num mod 2 == 0?

    3.1: Yes

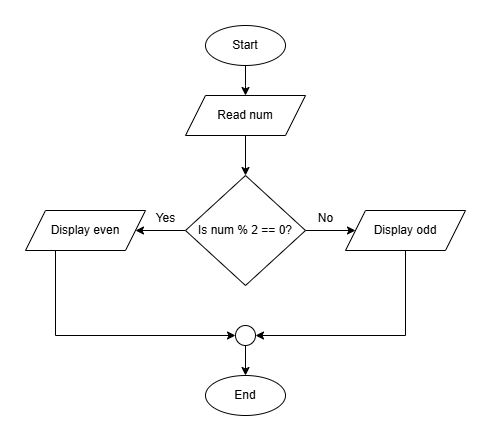
        3.1.1: Write even

    3.2: No

        3.2.2: Write odd

S4: End

Flowchart:

****

1. **Write an algorithm and draw a flowchart to find the greatest number among a and b.**

Algorithm:

1: Start

2: Read a, b

3: Is a > b?

    3.1: Yes

        3.1.1: Write a

    3.2: No

        3.2.1: Write b

4: End

Flowchart:

A diagram of a flowchart

AI-generated content may be incorrect.

1. **Write an algorithm and draw a flowchart to convert Celsius to Fahrenheit.**

Algorithm:

1. Start

2. Read C

3. F = 9 \* C / 5 + 32

4. Write F

5. End

Flowchart:

**A diagram of a diagram

AI-generated content may be incorrect.**

1. **Write an algorithm and draw a flowchart to find roots of quadratic equations.**

Algorithm:

1. Start

2. Read a, b, c

3. d = b\*b - 4\*a\*c

4. Is d > 0?

    4.1. Yes

        4.1.1. r1 = (-b + sqrt(d)) / (2 \* a)

        4.1.2. r1 = (-b - sqrt(d)) / (2 \* a)

        4.1.3. Write r1, r2

    4.2. No

        4.2.1. Is d = 0?

            4.2.1.1. Yes

                4.2.1.1.1. r = -b / (2 \* a)

                4.2.1.1.1. Write r

            4.2.1.2. No

                4.2.1.2.1. rp = -b / (2 \* a)

                4.2.1.2.2. ip = sqrt(abs(d)) / (2 \* a)

                4.2.1.2.3. Write rp + ip, rp - ip

5. End

Flowchart:

**A diagram of a flowchart

AI-generated content may be incorrect.**

1. **Write an algorithm and draw a flowchart to find if a year is leaping year.**

 HINT:

    If year % 4 = 0 --> a leap year

    If year % 100 = 0 --> not a leap year

    If year % 400 = 0 --> a leap year

Algorithm:

1. Start

2. Read year

3. Is year % 4 == 0?

    3.1. Yes

        3.1.1. Is year % 100 == 0?

            3.1.1.1. Yes

                3.1.1.1.1. Is year % 400 == 0?

                    3.1.1.1.1.1. Yes

                        3.1.1.1.1.1.1. Write a leap year.

                    3.1.1.1.1.2. No

                        3.1.1.1.1.2.1. Write not a leap year.

            3.1.1.2. No

                3.1.1.2.1. Write a leap year

    3.2. No

        3.2.1. Write not a leap year.

4. End

Flowchart:

**A diagram of a flowchart

AI-generated content may be incorrect.**

1. **Write an algorithm and flowchart to print from 1 to 10.**

Algorithm (Pre-Test Model):

1. Start

2. Calculate a = 1

3. Is a <= 10?

    3.1. Yes

        3.1.1. Write a

        3.1.2. a = a + 1

        3.1.3. Go to 4

4. End

Algorithm (Post-Test Model):

1. Start

2. Calculate a = 1

3. Write a

4. Is a <= 10?

    4.1. Yes

        4.1.1. a = a + 1

        4.1.2. Go to 4

5. End

Flowchart:

**A diagram of a method

AI-generated content may be incorrect.**

1. **Write an algorithm and flowchart to print from 1 to n.**

Algorithm (Pre-Test Model):

1. Start

2. Read n

3. Calculate a = 1

4. Is a <= n?

    4.1. Yes

        4.1.1. Write a

        4.1.2. a = a + 1

        4.1.3. Go to 4

5. End

Algorithm (Post-Test Model):

1. Start

2. Read n

3. Calculate a = 1

4. Write a

5. Is a <= 10?

    5.1. Yes

        5.1.1. a = a + 1

        5.1.2. Go to 4

6. End

Flowchart:

**A diagram of a method

AI-generated content may be incorrect.**

1. **Write an algorithm and draw a flowchart to find palindrome number.**

HINT:

    The number should be same from both LHS and RHS.

    e.g. 12321 is a palindrome number.

Algorithm:

1. Start

2. Read num

3. Calculate rev = 0, m = num

4. Calculate r = num % 10

5. Calculate num = num /10

6. Calculate rev = rev \* 10 + r

7. Is num != 0?

    7.1. Yes

        7.1.1. Go to 4

8. Is rev = m?

    8.1. Yes

        8.1.1. Write a palindrome number.

    8.2. No

        8.2.1. Write not a palindrome number.

9. End

Flowchart:

**A diagram of a flowchart

AI-generated content may be incorrect.**

Alternative Algorithm:

1. Start

2. Read num

3. Calculate rev = 0, m = num

4. Is num != 0?

    4.1. Yes

        4.1.1. Calculate r = num % 10

        4.1.2. Calculate num = num / 10

        4.1.3. Calculate rev = rev \* 10 + r

        4.1.4. Go to 4

5. Is rev = m?

    5.1. Yes

        5.1.1. Write a palindrome number.

    5.2. No

        5.2.1. Write not a palindrome number.

6. End

# Unit 2 – Introduction to c

## Introduction

C is a general-purpose programming language created by Dennis Ritchie at Bell Laboratories in 1972. It is a very popular language despite its age and is a fundamental language in the field of computer science.

### Key features

* General Purpose and Portable: C can be used to develop software that can be run on different hardware platforms.
* Low-level Memory Access: C provides low-level access to memory, which is crucial for system programming.
* Fast Speed: Compared to other programming languages like Java and Python, C is very fast.
* Clean Syntax: The syntax of C is straightforward and easy to understand.

### Importance

C is strongly associated with UNIX, as it was developed to write the UNIX operating system. Many later languages have borrowed syntax/features directly or indirectly from C. For example, the syntax Java, PHP, JavaScript, and many other languages is mainly based on C.

C++ was developed as an extension of C and both languages have almost the same syntax. The main difference between C and C++ is that C++ supports classes and objects while C doesn’t.

## Writing first program

**Program:**

/\*

Author: Kushal Prasad Joshi

Purpose: Introduction to C Programming

\*/

#include <stdio.h>  // Preprocessor directive --> Link Sectoin

int main()  // Start of main program --> main() section

{

    printf("Hello World");  // Our code

    return 0;   // Returning value  -->  Each function returns value after its completion

}

**Output:**

Hello World

### flow of a c program

**Preprocessing 🡪 Compilation 🡪 Assembly 🡪 Linking 🡪 Loading**

1. **Preprocessing:** Comments are removed; macros are expanded; #include files are resolved; converts .c file to .i file.
2. **Compilation:** Converts preprocessed file (.i file) to assembly file (.s file); Our program is converted to assembly language.
3. **Assembly:** Converts assembly file (.s file) to machine code instructions (.o file)
4. **Linking:** Connects all the machine code instructions (.o files) and makes an executable file (.exe file).
5. **Loading:** The executable file runs in your system.

**NOTE:** The command to get all these files is:

gcc -save-temps <|file\_name.c|> -o <|executable\_name.exe|>

## Components of c program

The following are the components of a C program:

* **Keywords**: Predefined words that have special meaning to the compiler.
* **Identifiers**: Names given to different entities such as variables, functions, etc.
* **Constants**: Values that remain fixed and cannot be changed.
* **String literals**: Sequence of characters enclosed in double quote.
* **Symbols**: abbreviations of operators.
* **Comments**: Portion of code that is not executed by compiler.

### keywords

The C programming language has 32 reserved keywords:

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

These keywords are reserved C for specific tasks so they can't be used for other tasks.

### Identifiers

Identifiers are names given to various program elements such as variables, functions, arrays, etc. They are used to identify these elements uniquely within the program.

#### rules for defining identifiers

1. Identifiers can consist of letters (both uppercase and lowercase), digits, and underscores (\_).
2. The first character of an identifier must be a letter or an underscore (\_). It cannot be a digit.
3. Identifiers are case-sensitive, meaning variable and Variable would be considered different identifiers.
4. Keywords cannot be used as identifiers.
5. There is no limit to the length of an identifier, but only the first 31 characters are typically significant.

**Example:**

int number;      // Valid identifier

float \_value;    // Valid identifier

char name1;      // Valid identifier

double 2ndValue; // Invalid identifier (cannot start with a digit)

int return;      // Invalid identifier (cannot use keyword)

### constants

Constants in C are fixed values that do not change during the execution of a program. They can be of any data type, such as integer, float, character, or string. Constants are defined using the const keyword or the #define preprocessor directive.

#### types of constants

1. **Integer Constants:** These are whole numbers without any fractional part. They can be written in decimal, octal, or hexadecimal format.
   * Decimal: 10, -20
   * Octal: 012 (equivalent to 10 in decimal)
   * Hexadecimal: 0xA (equivalent to 10 in decimal)
2. **Floating-point Constants:** These are numbers with a fractional part. They can be written in decimal or exponential notation.
   * Decimal: 3.14, -0.001
   * Exponential: 1.5e3 (equivalent to 1500)
3. **Character Constants:** These are single characters enclosed in single quotes.
   * Example: 'a', '1', '\n'
4. **String Constants:** **String Constants**: These are sequences of characters enclosed in double quotes.
   * Example: "Hello, World!", "C Programming"

#### Defining Constants

1. **Using #define Preprocessor Directive:** This method defines a constant value that is replaced by the preprocessor before compilation.

#define PI 3.14

#define MAX\_SIZE 100

1. **Using const Keyword:** This method defines a constant variable whose value cannot be changed after initialization.

const int maxAge = 100;

const float pi = 3.14;

**Example:**

#include <stdio.h>

#define PI 3.14  // Define constant using #define

int main() {

    const int maxAge = 100;  // Define constant using const keyword

    printf("Value of PI: %f\n", PI);

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

    return 0;

**Output:**

Value of PI: 3.140000

Maximum Age: 100

### string literals

String literals in C are sequences of characters enclosed in double quotes. They are used to represent text in a program. Each string literal ends with a special character \0 that indicates the end of the string.

#### characteristics of string literals

1. **Enclosed in Double Quotes:** String literals must be enclosed in double quotes. For example, "Hello, World!".
2. **Null-Terminated:** The C compiler automatically adds a null character \0 at the end of the string to mark its termination.
3. **Immutable:** String literals are read-only, meaning their content cannot be modified after they are defined.

#### FOrmat specifiers

Format specifiers in C are used in input and output functions to specify the type of data being read or written. They are used with functions like printf() and scanf() to format the data correctly.

Some common format specifiers are: