# **≤Module 2 – Introduction to Programming≥**

# Overview of C Programming 

THEORY EXERCISE: o Write an essay covering the history and evolution of C programming. Explain its importance and why it is still used today.

**The Importance of C**

## ✅ **Step 1: Install a C Compiler (GCC)**

GCC (GNU Compiler Collection) is one of the most widely used C compilers.

### 🪟 **For Windows:**

You need to install a distribution of GCC like **MinGW** or **TDM-GCC**.

#### 🔹 Option 1: Install MinGW (Minimalist GNU for Windows)

1. Go to https://osdn.net/projects/mingw/releases/
2. Download mingw-get-setup.exe and run it.
3. In the installer:
   * Select the base packages: mingw32-gcc-g++, mingw32-gcc-objc, and msys-base.
   * Click **Install**.
4. After installation, add the path to bin directory to your **System Environment Variables**:
   * Example: C:\MinGW\bin
   * Go to **Control Panel > System > Advanced system settings > Environment Variables** → edit Path.

To check if it's installed:

bash

CopyEdit

gcc --version

You should see version info in the terminal/command prompt.

## ✅ **Step 2: Choose and Install an IDE**

You can write and compile C code using various IDEs. Here's how to set up three popular ones:

### 🔹 **A. Dev-C++**

1. Download from https://sourceforge.net/projects/orwelldevcpp/
2. Run the installer and follow the prompts.
3. GCC is bundled with Dev-C++, so it works right out of the box.
4. Open Dev-C++ → File > New > Source File → write your C code → Save it as .c.
5. Click **Compile & Run** to test your code.

### 🔹 **B. Visual Studio Code (VS Code)**

VS Code is a lightweight editor that requires some setup to work with C.

#### Step-by-step setup:

1. Download and install VS Code: <https://code.visualstudio.com/>
2. Install GCC (MinGW) as shown above.
3. Open VS Code → Go to Extensions (Ctrl+Shift+X) → Search and install:
   * **C/C++** by Microsoft
   * Optionally: **Code Runner** for easy code execution
4. Add a .c file, and save it.
5. Create a simple **tasks.json** file in .vscode folder to configure build settings.  
   Example setup available upon request.

Tip: Run the program in a terminal using:

bash

CopyEdit

gcc filename.c -o output.exe

./output.exe

### 🔹 **C. Code::Blocks**

1. Download from http://www.codeblocks.org/downloads
2. Choose the version with **mingw-setup** (e.g., codeblocks-XX.XXmingw-setup.exe)
3. Install it. It includes GCC, so no need for separate compiler installation.
4. Open Code::Blocks → File > New > Project > Console Application → Choose C.
5. Follow the wizard and start coding.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
| 3. Basic Structure of a C Program  THEORY EXERCISE: o Explain the basic structure of a C program, including headers, main function, comments, data types, and variables. Provide examples. |  |  |  |  |  |  |
|  |  |  | Dev-C++ |  |  |  |
|  |  |  | VS Code | ❌ No | 👌 Medium | ✅ High |

4o

Top of Form

Tools

Bottom of Form

C became the foundation for many important developments in computing:

* **Operating Systems**: Unix, Linux, and parts of Windows are written in C.
* **Embedded Systems**: Due to its low-level memory control and efficiency, C is widely used in microcontrollers and firmware.
* **Programming Languages**: Many popular languages such as C++, Java, and Python have roots in or are influenced by C.
* **Compilers and Tools**: Many compilers and development tools are themselves written in C.

Its syntax and structure have also influenced nearly all modern programming languages, making C a critical language to understand for computer science students and professionals.

**Why C Is Still Used Today**

Despite the rise of newer languages, C remains widely used for several reasons:

1. **Performance**: C programs are fast and efficient, making it ideal for resource-constrained environments.
2. **Portability**: Code written in C can run on many different types of systems with minimal changes.
3. **Low-Level Access**: C allows direct manipulation of hardware resources like memory, making it perfect for systems programming.
4. **Simplicity**: While C requires careful management of memory and pointers, its syntax is relatively simple and clean.
5. **Legacy Code**: Many critical systems have been written in C and continue to be maintained in it.

In education, C remains a popular first language because it teaches fundamental programming concepts like pointers, memory management, and data structures at a low level.

**Conclusion**

The C programming language has had a profound impact on the development of modern computing. From operating systems and embedded systems to the creation of other languages, its legacy is unmatched. Even today, C is valued for its speed, control, and portability. Understanding C is not just about learning a language—it's about understanding the foundation of how computers and software work.

Let me know if you'd like this in a more academic format (with citations), or need a version tailored for a school or college assignment.

4o

### **1. Header Files**

* These are included at the top of the program using the #include directive.
* Header files contain **functions and macros** used in the program (e.g., input/output functions).

### **2. Main Function**

* Every C program must have a main() function.
* This is the **starting point** of execution.

c

CopyEdit

### **3. Comments**

* Comments are used to explain the code.
* **Single-line comment:** // This is a comment
* **Multi-line comment:**

### **4. Data Types**

* Data types define the type of data a variable can hold.
* Common types:
  + int – integers
  + float – decimal numbers
  + char – single characters
  + double – double-precision decimal numbers

### **5. Variables**

* Variables are named memory locations used to store data.
* Declaration follows the format:  
  data\_type variable\_name = value;

c

CopyEdit

1. Operators in C
2.  THEORY EXERCISE: o Write notes explaining each type of operator in C: arithmetic, relational, logical, assignment, increment/decrement, bitwise, and conditional operators.
3. Top of Form

Bottom of Form

## **1. Arithmetic Operators**

Used to perform basic mathematical operations.

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| + | Addition | a + b | Adds a and b |
| - | Subtraction | a - b | Subtracts b from a |
| \* | Multiplication | a \* b | Multiplies a and b |
| / | Division | a / b | Divides a by b (integer division if both are integers) |
| % | Modulus (remainder) | a % b | Remainder after a / b |

## **2. Relational (Comparison) Operators**

Used to compare two values.

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | a == b | true if equal |
| != | Not equal to | a != b | true if not equal |
| > | Greater than | a > b | true if a > b |
| < | Less than | a < b | true if a < b |
| >= | Greater than or equal | a >= b | true if a ≥ b |
| <= | Less than or equal | a <= b | true if a ≤ b |

## ⚙️ **3. Logical Operators**

Used to combine multiple conditions (true/false values).

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| && | Logical AND | a > 0 && b > 0 | true if both are true |
| ` |  | ` | Logical OR |
| ! | Logical NOT | !a | true if a is false |

## 📝 **4. Assignment Operators**

Used to assign values to variables.

| **Operator** | **Description** | **Example** | **Equivalent To** |
| --- | --- | --- | --- |
| = | Simple assignment | a = 5 | - |
| += | Add and assign | a += 2 | a = a + 2 |
| -= | Subtract and assign | a -= 3 | a = a - 3 |
| \*= | Multiply and assign | a \*= 4 | a = a \* 4 |
| /= | Divide and assign | a /= 2 | a = a / 2 |
| %= | Modulus and assign | a %= 3 | a = a % 3 |

## **5. Increment and Decrement Operators**

Used to increase or decrease a variable’s value by 1.

| **Operator** | **Description** | **Example** | **Effect** |
| --- | --- | --- | --- |
| ++ | Increment by 1 | a++ or ++a | Adds 1 to a |
| -- | Decrement by 1 | a-- or --a | Subtracts 1 from a |

++a (pre-increment): Increments first, then uses value  
a++ (post-increment): Uses value first, then increments

## **6. Bitwise Operators**

Operate on individual bits of data (useful in low-level programming).

| **Operator** | **Description** | **Example** | **Meaning** |
| --- | --- | --- | --- |
| & | AND | a & b | 1 if both bits are 1 |
| ` | ` | OR | `a |
| ^ | XOR | a ^ b | 1 if bits are different |
| ~ | NOT (One's complement) | ~a | Inverts bits |
| << | Left shift | a << 1 | Shifts bits left |
| >> | Right shift | a >> 1 | Shifts bits right |

## **7. Conditional (Ternary) Operator**

Used to replace simple if-else statements in a single line.

### Syntax:

c

CopyEdit

condition ? expression\_if\_true : expression\_if\_false;

1. Control Flow Statements in C  THEORY EXERCISE: o Explain decision-making statements in C (if, else, nested if-else, switch). Provide examples of each.

## **1.** if **Statement**

The if statement checks a **condition**. If it's true, the code inside the block runs.

## **2.** if-else **Statement**

The if-else statement allows two possibilities: if the condition is true, do something; otherwise, do something else.

## **3. Nested** if-else **Statement**

Used when multiple conditions must be checked. An if or else if block can contain another if-else structure.

## **4.** switch **Statement**

The switch statement allows a variable to be tested for **multiple values**, each defined in a case.

1. Looping in C  THEORY EXERCISE: o Compare and contrast while loops, for loops, and do-while loops. Explain the scenarios in which each loop is most appropriate.

## **1.** while **Loop**

### 🔹 **Description:**

* Repeats a block of code **as long as a condition is true**.
* The condition is **checked before** each iteration.

## **2.** for **Loop**

### 🔹 **Description:**

* Best for looping **a known number of times**.
* Has **initialization**, **condition**, and **update** in one line.

## **3.** do-while **Loop**

### 🔹 **Description:**

* Executes the loop body **at least once**, then checks the condition.
* Condition is evaluated **after** the loop body.

1. Loop Control Statements  THEORY EXERCISE: o Explain the use of break, continue, and goto statements in C. Provide examples of each.

## **1.** break **Statement**

### 🔹 **Purpose:**

* Immediately **terminates a loop** (for, while, do-while) or exits a switch case.

## **2.** continue **Statement**

### 🔹 **Purpose:**

* Skips the **current iteration** of the loop and proceeds to the next one.

## **3.** goto **Statement**

### 🔹 **Purpose:**

* Transfers control to a **labeled statement** in the program.
* Generally **discouraged** due to the risk of creating unreadable "spaghetti code", but may be useful in certain low-level or error-handling situations

1. Functions in C  THEORY EXERCISE: o What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples

### ✅ What Are Functions in C?

**Functions** in C are blocks of code that perform a specific task. They help in **modular programming**, making code more readable, reusable, and easier to debug.

## 🔹 **Why Use Functions?**

* Avoid code repetition
* Break complex problems into simpler parts
* Improve code organization and maintenance

## 🔧 **1. Function Declaration (Prototype)**

* Tells the compiler about the **function name**, **return type**, and **parameters**.
* Placed **before main()**.

## **2. Function Definition**

* Actual code block that defines what the function does.

## **3. Calling a Function**

* Used to execute the function’s code from main() or another function.

1. Arrays in C  THEORY EXERCISE: o Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.

## ✅ What is an Array in C?

An **array** is a **collection of variables** of the same data type, stored in **contiguous memory locations**. It allows storing multiple values using a single variable name, accessed via indices.

## 🔹 Why Use Arrays?

* To store large amounts of data efficiently.
* To handle multiple values of the same type using loops.
* Useful for tasks like storing lists, matrices, tables, etc.

## **1. One-Dimensional Array**

### 🧾 Definition:

A linear list of elements stored in a single row.

## **2. Multi-Dimensional Array**

### 🧾 Definition:

An array of arrays. The most common form is the **2D array**, which is essentially a table (matrix).

## Differences Between 1D and Multi-Dimensional Arrays

| **Feature** | **One-Dimensional Array** | **Multi-Dimensional Array** |
| --- | --- | --- |
| Structure | Single row/list | Table-like (rows × columns) |
| Syntax Example | int a[5]; | int b[3][4]; |
| Accessing Elements | a[2] | b[1][3] |
| Use Case | Lists, scores, names | Matrices, tables, grids |

## 📝 Notes:

* **Indexing starts at 0** in C.
* Arrays in C have **fixed size**; dynamic resizing isn't built-in (use pointers/dynamic memory for that).
* For **3D arrays**, you can define them as int a[x][y][z];, and so on.