Unit 1

Programming Foundations and C Basics

Learning Objectives

- To understand the role of programming in systems and embedded development
- To explore the history, features, and applications of the C programming language
- To identify the basic syntax, data types, operators, and I/O operations in C
- To implement conditional and looping constructs in C
- To practice structured programming, code commenting, and documentation

Structure

1.1 Role of Programming in Systems and Embedded Development

Programming is the process of writing instructions that a computer can understand and execute. These instructions are written in a specific language, such as C, to control how the computer behaves. In systems and embedded development, programming plays a central role in building the software that runs on devices, from personal computers to small microcontrollers in everyday electronics.

Programming in this field helps in making systems faster, smaller, more reliable, and better suited for real-time tasks. Whether it's an operating system, a car's engine control unit, or a microwave oven, programming defines the logic behind how these systems function.

1.1.1 Definition and Purpose of Programming

Definition:

Programming is the act of writing code that a machine can interpret and execute to perform specific tasks. It involves giving step-by-step instructions using a programming language.

Purpose:

- To solve problems using logic and automation
- To control hardware and devices
- To create software applications, from simple calculators to complex operating systems
- To manage data, perform calculations, and interact with users or other systems

In systems development, programming is used to create the software layer that interacts with the computer hardware. In embedded development, it controls small devices like sensors, appliances, or controllers.

1.1.2 Programming in System Software

System software is the foundation on which application software runs. It includes components like:

- Operating systems (e.g., Windows, Linux)
- Device drivers (used to control hardware)
- System utilities (like disk formatters or memory testers)
- Compilers and interpreters (used to process programming languages)

Programming in this area involves writing code that can:

- Manage hardware resources such as CPU, memory, and I/O devices
- Schedule tasks and run applications
- Ensure secure and efficient functioning of the computer system

C is one of the most commonly used languages for system software because it provides low-level access to memory and hardware while still being easier to use than pure assembly language.

1.1.3 Programming for Embedded Systems

An **embedded system** is a special-purpose computer built into a larger system. It is designed to perform a dedicated task, often in real-time.
Examples include:

- Microwave ovens
- Washing machines
- Smartwatches
- Automotive control units
- Industrial robots

Programming for embedded systems requires:

- Writing compact and efficient code, often in C
- Directly interacting with sensors, actuators, and other hardware components
- Meeting timing and performance constraints (e.g., responding to a button press instantly)
- Managing limited resources (like memory or power)

Unlike general-purpose computers, embedded systems usually do not have a full operating system, so the programmer must handle many low-level operations manually.

1.1.4 Importance of Efficiency and Portability

In systems and embedded programming, **efficiency** and **portability** are critical.

- **Efficiency** means that the code runs fast and uses minimal resources. This is important because embedded devices often have limited memory, CPU power, or battery life. For example, a smartwatch needs efficient code to run smoothly without draining the battery quickly.
- **Portability** means that code can be reused across different platforms with little modification. C is often chosen for its portability because:
  - It works on many hardware platforms
  - It has a standard structure and syntax
  - It allows access to low-level hardware when needed

By writing portable and efficient code, developers can build applications that are easier to maintain, update, and move to new devices.

1.1.5 Real-World Use Cases in Embedded Development

Here are some real-world examples of programming in embedded systems:

- **Automotive Industry**: Programming is used to control engine functions, airbag deployment, and navigation systems.
- **Medical Devices**: Programmers write software for heart monitors, infusion pumps, and diagnostic equipment.
- **Consumer Electronics**: Devices like smart TVs, fitness bands, and Bluetooth speakers all run embedded code.
- **Home Automation**: Thermostats, smart locks, and lighting systems are programmed for remote control and automation.
- **Industrial Automation**: Robotics and control systems in factories are driven by embedded software that must respond in real-time to changes in the environment.

In all these cases, programming ensures the device behaves as expected, responds to inputs, and interacts with other systems or users.

1.2 History and Importance of C Language

The C programming language is one of the most influential and widely used programming languages in the world. It was created to provide powerful, flexible, and efficient programming capabilities while still being relatively easy to use. It has become the foundation for many other programming languages and is still commonly used today in system software, embedded systems, and performance-critical applications.

1.2.1 Evolution of Programming Languages

Programming languages have evolved over time to make software development more efficient and understandable. This evolution can be broadly divided into generations:

- **First Generation (1GL)** – Machine Language: Uses binary code (0s and 1s). Hard to write and understand.
- **Second Generation (2GL)** – Assembly Language: Uses symbolic instructions (e.g., MOV, ADD) instead of binary. Still hardware-specific.
- **Third Generation (3GL)** – High-Level Languages: Easier to write and understand, portable across machines. Examples include FORTRAN, COBOL, BASIC, and C.
- **Fourth Generation (4GL)** – Very High-Level Languages: Designed for specific applications like databases (e.g., SQL).

- **Fifth Generation (5GL)** – Based on logic and constraints, used in AI and machine learning applications.

C emerged in the third generation and brought the power of low-level programming with the ease of high-level syntax, making it very popular.

1.2.2 Development of the C Language

C was developed in the early 1970s by **Dennis Ritchie** at **Bell Labs** in the United States. It was created as a successor to the B language, which itself was derived from BCPL (Basic Combined Programming Language).

Key milestones in C's development:

- **1972**: Dennis Ritchie created C to develop the UNIX operating system.
- **1978**: The first edition of the book *"The C Programming Language"* by Brian Kernighan and Dennis Ritchie was published. This version is often referred to as **K&R C**.
- **1989**: ANSI (American National Standards Institute) established a standard version known as **ANSI C** or **C89**.
- **1990s and beyond**: Later standards like C99 and C11 added more features like inline functions, new data types, and improved library support.

C quickly became popular for system programming because it was simple, close to hardware, and portable across systems.

1.2.3 Key Features and Strengths of C

C offers a unique combination of features that make it both powerful and flexible:

- **Simplicity**: The language has a small set of keywords and simple syntax.
- **Portability**: Programs written in C can run on different machines with little modification.
- **Efficiency**: C provides low-level access to memory and system resources, making programs run faster.
- **Structured Programming**: Supports functions and modular programming, which helps in managing large codebases.
- **Rich Library Support**: Standard libraries provide commonly used functions (I/O, math, strings, etc.).
- **Extensibility**: Programmers can build new operations and data structures easily.

- **Direct Memory Access**: Through pointers, C gives control over memory, which is crucial in systems and embedded development.

These features make C suitable for both application-level and system-level programming.

1.2.4 Role of C in Modern Systems and Devices

Even after decades, C remains highly relevant in today's software and hardware development. It is used in:

- **Operating Systems**: Most parts of UNIX, Linux, and even Windows kernels are written in C.
- **Embedded Systems**: Microcontrollers and firmware rely on C for direct hardware control.
- **Compilers and Interpreters**: Many language compilers (e.g., Python, Java) are implemented in C.
- **Device Drivers**: C is used to write drivers that communicate between the OS and hardware.
- **Networking Tools**: Many network applications and protocols are developed in C.
- **Gaming and Graphics Engines**: C is used for performance-critical parts of game engines.

Because of its speed and control over system resources, C is ideal for building foundational software components.

1.2.5 Comparison with Other Languages

Here's how C compares with some other common programming languages:

Language	Comparison with C
C++	Adds object-oriented features to C. More complex, but better for large-scale software.
Java	Platform-independent, runs on JVM. Easier memory management but slower than C.
Python	Very easy to write and read. Interpreted, so slower than C. Often uses C modules for performance.
Assembly	More low-level than C, faster, but harder to write and maintain.
Rust	Safer memory management and modern features, considered a modern alternative to C in some areas.

C strikes a balance between performance and usability. While newer languages offer more safety or features, C still remains unmatched in situations that require close-to-hardware control and high efficiency.

1.3 C Language Syntax and Structure

C programs follow a specific structure and syntax to ensure that the code is readable by both the compiler and the programmer. Understanding the building blocks of a C program is the first step toward writing correct and efficient code.

1.3.1 Structure of a C Program

A basic C program is organized into different sections, usually in the following order:

1. **Preprocessor Directives**: These are lines starting with #, used to include libraries or define constants.
   Example: #include <stdio.h>
2. **Global Declarations**: Variables or functions that are declared outside any function and accessible throughout the program.
3. **main() Function**: Every C program must have a main() function. It is the starting point of execution.
4. **Function Definitions**: If the program has other functions, they are defined either before or after main().

Example Structure:

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

int add(int a, int b);    // Function Declaration

int main() {                    // main() Function
      int result = add(5, 3);
      printf("Result: %d\n", result);
      return 0;
}

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

1.3.2 Header Files and Preprocessor Directives

- **Header Files** are files that contain predefined functions, macros, or declarations.
  Common headers:
  - <stdio.h> – for input/output functions (printf, scanf)
  - <math.h> – for mathematical functions (sqrt, pow)
  - <string.h> – for string manipulation functions (strlen, strcpy)
- **Preprocessor Directives** begin with # and are executed before the actual compilation.

- #include – includes a header file
- #define – defines constants or macros
- #ifdef, #ifndef – used for conditional compilation

Example:
#include <stdio.h>
#define PI 3.14
Preprocessor directives do not end with semicolons.

1.3.3 The main() Function and Execution Flow

- The **main() function** is the entry point of a C program.
  When a C program runs, execution starts from the first line inside main().

Basic Syntax:

```
int main() {
    // code here
    return 0;
}
```

- The int before main indicates that the function returns an integer value.
- The return 0; statement at the end tells the operating system that the program has ended successfully.

Execution Flow:

1. The compiler starts with the first statement inside main()
2. Statements are executed one by one in order
3. Function calls are made if any, and control is transferred accordingly
4. Execution ends when the last statement is reached or return is called

1.3.4 Compilation and Linking

C is a compiled language, meaning the source code must be converted into machine code before execution. This happens in several steps:

1. **Preprocessing** – Handles #include, #define, etc.
2. **Compilation** – Converts the C code (.c files) into object code (.o or .obj files)
3. **Linking** – Combines object code with libraries to produce the final executable (.exe or a.out)

Tools Used:

- gcc (GNU Compiler Collection) is commonly used to compile C programs
- make can automate the build process for larger projects

Example Compilation:

gcc program.c -o program
This command compiles program.c and creates an executable named program.

1.3.5 Syntax Rules and Semicolon Terminators

C is a language with **strict syntax rules**. Common rules include:

- **Each statement must end with a semicolon (;)**
  Example: int x = 10;
- **Braces {} are used to group multiple statements** into blocks, such as in if statements or loops.
- **C is case-sensitive**, so Variable and variable are different identifiers.
- **Correct use of parentheses ()** is necessary in function calls and control structures.
- **Indentation is not required by the compiler**, but it's important for readability.

Incorrect Syntax Example:

```
int x = 10
printf("x = %d", x)
```

The above code will generate errors because it is missing semicolons.

Corrected Version:

```
int x = 10;
printf("x = %d", x);
```

1.4 Constants, Variables, Keywords, and Data Types

In C programming, understanding how to define and use **variables**, **constants**, **keywords**, and **data types** is essential. These are the basic building blocks for writing any C program. They help in storing, processing, and managing data efficiently.

1.4.1 Declaring and Initializing Variables

A **variable** is a named storage location in memory used to hold a value that may change during program execution.
Declaration means telling the compiler the name and type of a variable.
Initialization means assigning a value to the variable at the time of declaration.
Syntax:

```
int age;                // Declaration
age = 25;               // Initialization

int marks = 90;     // Declaration with Initialization
```

Rules for declaring variables:

- The variable name must begin with a letter or underscore (_).
- It can contain letters, digits, and underscores.
- It should not be a **keyword** (like int, return).

Variables must be declared before they are used.

1.4.2 Understanding Constants and #define

A **constant** is a value that does not change during program execution.
There are two main ways to define constants in C:

1. **Using const keyword:**

const float PI = 3.14;
This tells the compiler that PI cannot be changed later in the program.

2. **Using #define preprocessor directive:**

#define PI 3.14
This replaces every occurrence of PI in the code with 3.14 before compilation. This is handled during the **preprocessing** phase.
Difference between const and #define:

- const is type-checked by the compiler, #define is not.
- const uses memory space; #define does not (it's just a substitution).

1.4.3 Keywords and Identifiers

- **Keywords** are reserved words in C that have special meaning and cannot be used for other purposes (like variable names).
- **Identifiers** are names given to variables, functions, arrays, etc., defined by the programmer.

Examples of Keywords in C:
int, float, return, if, else, while, for, void, break, continue, switch
There are 32 standard keywords in ANSI C.
Examples of Identifiers:
studentName, totalMarks, sum, average
Rules for Identifiers:

- Cannot be a keyword
- Cannot start with a digit
- No special characters except underscore (_)
- Case-sensitive (e.g., Marks and marks are different)

1.4.4 Basic and Derived Data Types

Data types define the type of data a variable can hold.
Basic Data Types:

Data Type	Description	Size (Typical)	Example
int	Integer	4 bytes	int a = 10;
float	Decimal	4 bytes	float pi = 3.14;
char	Character	1 byte	char ch = 'A';
double	Double-precision float	8 bytes	double x = 2.71828;

Derived Data Types:
These are based on basic types:

- **Arrays**: Collection of elements of the same type
  Example: int arr[5];
- **Pointers**: Stores address of another variable
  Example: int *ptr;
- **Structures**: Group of different data types
  Example: struct Student { int id; char name[20]; };
- **Unions**: Similar to structures but share memory
- **Functions**: Blocks of code to perform specific tasks

1.4.5 Type Modifiers and Storage Classes

Type Modifiers are used to change the size or sign of data types:

- signed (default): Allows positive and negative values
- unsigned: Only positive values
- short: Reduces the size of an int
- long: Increases the size of an int or double

Examples:
unsigned int a = 100;
short int b = 10;
long double c = 3.141592653;
Storage Classes define:

- **Scope** (where the variable is visible)
- **Lifetime** (how long the variable exists)
- **Default initial value**

Storage Class	Scope	Lifetime	Default Value	Keyword
auto	Local	Until function ends	Garbage	auto
static	Local/Global	Till program ends	0	static
register	Local	Until function ends	Garbage	register
extern	Global	Till program ends	0	extern

Example:
static int counter = 0;
This variable retains its value between function calls.

1.5 Operators and Expressions

In C programming, **operators** are symbols that perform operations on variables and values. An **expression** is a combination of variables, constants, and operators that produces a value.
For example, in a + b * 2, the + and * are operators, and the whole statement is an expression.

1.5.1 Arithmetic Operators

These operators are used to perform basic mathematical operations.

Operator	Meaning	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
%	Modulus	a % b	Remainder of a / b

Example:

```
int a = 10, b = 3;
int sum = a + b;          // sum = 13
int rem = a % b;          // rem = 1
```

Note: % can only be used with integers, not floating-point numbers.

1.5.2 Relational and Logical Operators

Relational Operators are used to compare values.

Operator	Meaning	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 is greater
<	Less than	a < b	True if a is smaller
>=	Greater or equal	a >= b	True if a is greater or equal
<=	Less or equal	a <= b	True if a is smaller or equal

These return 1 for true and 0 for false.

Logical Operators are used to combine multiple conditions.

Operator	Meaning	Example	Result
&&	Logical AND	a > 0 && b < 5	True if both are true
`		`	Logical OR
!	Logical NOT	!a	True if a is false (zero)

Example:

```
if (a > 0 && b < 10) {
    // this block runs if both conditions are true
}
```

1.5.3 Assignment, Compound, and Bitwise Operators

Assignment Operators assign values to variables.

Operator	Meaning	Example

Operator	Meaning	Example
=	Assign value	a = 5

Compound Assignment Operators combine arithmetic and assignment.

Operator	Meaning	Example	Equivalent To
+=	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 %= 5;	a = a % 5;

Bitwise Operators work at the binary level.

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

Bitwise operators are mostly used in embedded systems and hardware-level programming.

1.5.4 Increment and Decrement Operators

These are used to increase or decrease a variable's value by 1.

Operator	Meaning	Example
++	Increment (add 1)	a++ or ++a
--	Decrement (subtract 1)	a-- or --a

There are two types:

- **Post-Increment**: a++ → First use a, then increment
- **Pre-Increment**: ++a → First increment, then use a

Example:

```
int a = 5;
int b = a++;    // b = 5, a = 6
int c = ++a;    // a = 7, c = 7
```

1.5.5 Operator Precedence and Associativity

Operator Precedence determines which operator is evaluated first when multiple operators are in an expression.

Associativity decides the direction of execution (left to right or right to left) when operators have the same precedence.

Common Operator Precedence (High to Low)

Precedence Level	Operators	Associativity
Highest	(), [], ., ->	Left to right

Precedence Level	Operators	Associativity
	++, -- (prefix), +, - (unary)	Right to left
Medium	*, /, %	Left to right
	+, -	Left to right
	<, >, <=, >=	Left to right
	==, !=	Left to right
Lower	&&, `	
	=, +=, -=, *=	Right to left

Example:

```
int a = 10, b = 5, c = 2;
int result = a + b * c;    // result = 10 + (5 * 2) = 20
```

Multiplication has higher precedence than addition, so it's done first.
To change the order, use parentheses:

```
int result = (a + b) * c;    // result = (10 + 5) * 2 = 30
```

1.6 Input and Output in C

In C programming, **input** refers to data taken from the user or an external source, and **output** is the data shown to the user or sent to a device. C provides built-in functions to perform input and output operations through the standard I/O library: **stdio.h**.

1.6.1 Using printf() and Format Specifiers

printf() is a standard function used to display output on the screen.
Syntax:

```
printf("format string", variables);
```

Format specifiers are placeholders inside the string that define the type of data being printed.
Common Format Specifiers:

Format Specifier	Description	Example Output
%d	Integer (decimal)	printf("%d", 10); → 10
%f	Floating point number	printf("%f", 3.14); → 3.140000
%c	Single character	printf("%c", 'A'); → A
%s	String	printf("%s", "C"); → C
%.2f	Floating point with 2 decimals	printf("%.2f", 3.14); → 3.14

Example:

```
int age = 20;
printf("Age is: %d\n", age);
```

The \n is an **escape sequence** that moves the cursor to the next line.

1.6.2 Reading Input with scanf()

scanf() is used to take input from the user during program execution.
Syntax:

```
scanf("format string", &variables);
```

Note that **ampersand (&)** is used before variable names to pass their **addresses**.
Examples:

```
int age;
scanf("%d", &age);                // Reading an integer

float price;
scanf("%f", &price);              // Reading a float

char name[20];
scanf("%s", name);                // Reading a string (no & for arrays)
```

Limitations of scanf():

- It stops reading a string when it encounters a space.
- For reading full lines or strings with spaces, use gets() or fgets() instead.

1.6.3 Character I/O: getchar() and putchar()

These functions are used to read or print a **single character**.
getchar()
Reads one character from input (keyboard).
Example:

```
char ch;
ch = getchar();      // User enters a character
```

putchar()
Prints a single character to output.
Example:

```
putchar(ch);          // Displays the character
```

They are simple and useful in character-based programs.

1.6.4 String I/O: gets() and puts()

These are used for reading and writing **strings**.
gets() – Reads an entire line including spaces until Enter is pressed.

```
char name[50];
gets(name);
```

puts() – Prints a string followed by a newline.

```
puts(name);
```

Important Note:
gets() is **unsafe** and **not recommended** in modern C because it does not check for buffer overflow, which may lead to security issues. Use fgets() instead:

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

1.6.5 Common Input/Output Errors

1. **Missing & in scanf()**
2. scanf("%d", age);   // Incorrect
3. scanf("%d", &age);   // Correct
4. **Using scanf() for strings with spaces**

- Only the first word is read.
- Use gets() or fgets() instead.

5. **Buffer overflow with gets()**
    - Reading more characters than the array can hold.
    - Use fgets() for safety.
6. **Incorrect format specifiers**
7. int a;
8. scanf("%f", &a); // Wrong: %f is for float, not int
9. **Mismatched data types**
    - Reading into the wrong variable type causes undefined behavior.
    - Always match the format specifier with the correct data type.

1.7 Conditional Statements

Conditional statements allow a program to make decisions and execute different blocks of code based on whether a condition is **true** or **false**. These conditions are usually based on **relational** or **logical expressions**. They help control the flow of the program and make it interactive and intelligent.

1.7.1 if Statement

The if statement is the most basic form of a conditional. It executes a block of code **only when** the specified condition is true.

Syntax:

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

Example:

```
int age = 20;

if (age >= 18) {
    printf("You are eligible to vote.");
}
```

If the condition is **false**, the code block is **skipped**.

1.7.2 if-else and Nested if-else

if-else Statement

The if-else statement allows choosing between two options:

- One block runs if the condition is **true**
- The other runs if the condition is **false**

Syntax:

```
if (condition) {
        // true block
} else {
        // false block
}
```
Example:
```
int number = 5;

if (number % 2 == 0) {
        printf("Even");
} else {
        printf("Odd");
}
```
Nested if-else

Nested if-else refers to placing one if-else block **inside another** to check multiple conditions in sequence.

Example:
```
int marks = 85;

if (marks >= 90) {
        printf("Grade A");
} else if (marks >= 75) {
        printf("Grade B");
} else if (marks >= 60) {
        printf("Grade C");
} else {
        printf("Fail");
}
```
The program evaluates each condition **from top to bottom** and executes the block of the **first true condition**.

1.7.3 switch-case Statement

The switch-case statement is used when a variable needs to be compared with **multiple constant values**. It improves **readability** compared to many if-else statements.

Syntax:
```
switch (expression) {
        case value1:
                // code block
                break;
        case value2:
                // code block
                break;
        ...
        default:
                // default code block
}
```

Example:

```
int day = 2;

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

Key Points:

- break ends the case and prevents the code from **falling through** to the next case.
- default is optional and is executed when **no case matches** the expression.

1.7.4 Conditional (Ternary) Operator

The **ternary operator** is a shorthand way of writing a simple if-else statement. It evaluates a condition and returns one of two values depending on the result.

Syntax:

```
condition ? expression_if_true : expression_if_false;
```

Example:

```
int a = 5, b = 10;
int max = (a > b) ? a : b;

printf("Maximum = %d", max);
```

This checks whether a > b and assigns the greater value to max.

1.7.5 Best Practices in Conditional Logic

- **Keep conditions simple and readable**
  Avoid overly complex conditions in one line. Break them into smaller conditions if needed.
- if ((a > b) && (c < d)) {
- 	// clear and readable
- }
- **Use switch for fixed constant comparisons**
  Ideal for menu-driven programs and cases where one variable is compared against many constant values.

- **Avoid deep nesting**
  Multiple nested if-else blocks can make the code hard to read. Use functions or early returns to simplify.
- **Always use break in switch cases**
  Avoid logical errors caused by fall-through behavior unless you intentionally want to execute multiple cases.
- **Validate all user input**
  Ensure values fall within valid ranges before making decisions in the program.
- **Use ternary operators only for simple decisions**
  Great for reducing code size, but avoid using them with complex logic as it can reduce readability.
- **Always use braces {}**
  Even for single-line statements to prevent future bugs and improve clarity.

Bad Practice:

```
if (x > 0)
    printf("Positive");
```

Recommended:

```
if (x > 0) {
    printf("Positive");
}
```

1.8 Looping Constructs

Loops are used in programming to execute a block of code **multiple times** based on a condition. Instead of writing the same code repeatedly, loops help in repeating tasks efficiently. C provides three types of loops: for, while, and do-while.

1.8.1 for Loop and Variations

The for loop is used when the number of iterations is **known** in advance.

Syntax:

```
for (initialization; condition; increment/decrement) {
    // code to be repeated
}
```

- **Initialization**: Sets the starting point (e.g., int i = 0)
- **Condition**: Checked before each iteration
- **Increment/Decrement**: Updates the loop variable after each iteration

Example:

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

Output:

1
2
3
4
5

Variations of for loop:

- Decreasing loop:
- for (int i = 10; i >= 1; i--) { ... }
- Skipping values:
- for (int i = 0; i <= 10; i += 2) { ... }
- Infinite loop (use with caution):
- for(;;) { ... }

1.8.2 while Loop

The while loop is used when the number of iterations is **not known in advance**. The loop continues **as long as the condition is true**.

Syntax:

```
while (condition) {
    // code to repeat
}
```

Example:

```
int i = 1;
while (i <= 5) {
    printf("%d\n", i);
    i++;
}
```

If the condition is initially false, the loop body **does not run at all**.

1.8.3 do-while Loop

The do-while loop is similar to the while loop, but it **always executes at least once**, because the condition is checked **after** the loop body.

Syntax:

```
do {
    // code to execute
} while (condition);
```

Example:

```
int i = 1;
do {
    printf("%d\n", i);
    i++;
} while (i <= 5);
```

Even if i was initialized to a value greater than 5, the loop would still run once.

1.8.4 Nested Loops

A **nested loop** means placing one loop inside another. It is commonly used for working with **matrices**, **patterns**, or **tables**.

Syntax:

```
for (int i = 1; i <= 3; i++) {
      for (int j = 1; j <= 2; j++) {
            printf("i = %d, j = %d\n", i, j);
      }
}
```

Output:

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
i = 3, j = 1
i = 3, j = 2
```

Nested loops can also be used with while or do-while.

1.8.5 Loop Performance Considerations

Using loops efficiently is important for writing **fast and optimized programs**.

Best Practices:

1. **Avoid unnecessary calculations inside the loop**
2. // Inefficient
3. for (int i = 0; i < strlen(str); i++) { ... }
4.
5. // Efficient
6. int len = strlen(str);
7. for (int i = 0; i < len; i++) { ... }
8. **Choose the right loop type**
    - Use for when number of iterations is known
    - Use while when waiting for a condition
    - Use do-while for menu-driven programs (run at least once)
9. **Minimize nested loops**
    - Too many nested loops increase time complexity
10. **Use break and continue wisely**
    - break exits the loop

- continue skips the current iteration

11. **Avoid infinite loops**

12. // Potential infinite loop

13. while (1) {

14. // make sure to include a break condition

15. }

16. **Watch for off-by-one errors**

    - Make sure loop conditions are correctly written to avoid skipping or overstepping iterations

1.9 Jump and Branching Statements

Jump and branching statements are used to **alter the normal flow of execution** in a program. These statements allow the program to:

- Skip certain parts of code,
- Exit loops early, or
- Jump to a specific labeled location in the code.

These include break, continue, and goto.

1.9.1 break Statement

The break statement is used to **terminate a loop or switch-case** block **immediately**, regardless of the condition.

Usage:

- Inside for, while, or do-while loops
- Inside switch statements

Syntax:

```
break;
```

Example in loop:

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

Output:

```
1
2
3
4
```

Example in switch:

```
int choice = 2;

switch (choice) {
	case 1:
		printf("Option 1");
		break;
	case 2:
		printf("Option 2");
		break;
	default:
		printf("Invalid");
}
```
Without break, the control would "fall through" to the next case.

1.9.2 continue Statement

The continue statement is used to **skip the remaining code** in the current iteration of a loop and **jump to the next iteration**.

Syntax:

```
continue;
```

Example:

```
for (int i = 1; i <= 5; i++) {
	if (i == 3) {
		continue;    // skip when i is 3
	}
	printf("%d\n", i);
}
```

Output:

```
1
2
4
5
```

Here, when i == 3, continue causes the loop to skip the printf() and move to the next value of i.

1.9.3 goto Statement and Labels

The goto statement allows an **unconditional jump** to a labeled part of the code.

Syntax:

```
goto label_name;

// later in the code
label_name:
	// code to execute
```

Example:

```
int num = 5;

if (num > 0) {
	goto skip;
```

```
}

printf("This will be skipped.\n");

skip:
printf("Jumped using goto.\n");
```

Output:

Jumped using goto.

The goto statement is **rarely used** in modern programming because it can make code **difficult to read and maintain**, especially in large programs.

1.9.4 Use Cases and Cautions

Use Cases:

- break is commonly used to exit from loops or switch statements.
- continue is useful when you want to skip some part of the loop body for certain conditions.
- goto is sometimes used:
    - For **early exit** from deeply nested loops
    - For **error handling** in low-level system code (though modern practices avoid this)

Cautions:

- Overusing break and continue can make loops hard to follow.
- Using goto can lead to **spaghetti code**—code that is tangled and hard to debug.
- Using goto breaks the **structured programming model** (which promotes clarity and block-based control flow).

1.9.5 Structured Alternatives to goto

Structured programming recommends using **functions, loops, and conditionals** instead of goto.

Alternative using loop + break:

Instead of:

```
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        if (i + j == 10) {
            goto exit;
        }
    }
}
exit:
printf("Exited loop.\n");
```

Use:

```
int found = 0;
for (int i = 0; i < 10 && !found; i++) {
	for (int j = 0; j < 10; j++) {
		if (i + j == 10) {
			found = 1;
			break;
		}
	}
}
printf("Exited loop.\n");
```

Alternative using functions:

Instead of jumping around the code with goto, break the program into **functions** with proper return statements.

Example:

```
void checkNumber(int num) {
	if (num < 0) return;
	printf("Number is positive.\n");
}
```

Using structured control flow makes the code easier to understand, test, and maintain.

1.10 Structured Programming and Code Documentation

Structured programming is a programming method that emphasizes breaking a program into small, manageable, and logical parts. It promotes clarity, simplicity, and maintainability by using standard control structures such as sequence, selection, and iteration.

In addition to writing structured code, documenting the code through meaningful comments is essential for readability and collaboration.

1.10.1 Principles of Structured Programming

Structured programming is based on **three main principles**:

1. **Sequence** – Code is executed line by line in the order written.
2. int a = 5;
3. int b = 10;
4. int sum = a + b;
5. **Selection** – Uses conditional statements like if, if-else, and switch to make decisions.
6. if (a > b) {
7. printf("A is greater.");
8. }

9. **Iteration** – Repeats a block of code using loops like for, while, and do-while.

```
10. for (int i = 0; i < 5; i++) {
11.         printf("%d\n", i);
12. }
```

Structured programming avoids the use of goto and encourages **clear, logical control flow**.

1.10.2 Modular Programming Concepts

Modular programming is a technique where a program is divided into **independent modules or blocks**, each handling a specific task. Each module can be developed, tested, and debugged separately.

Benefits of Modular Programming:

- **Reusability**: Modules can be reused in other programs.
- **Maintainability**: Easier to update or fix parts of the code.
- **Readability**: Code is easier to understand when split into meaningful sections.
- **Teamwork**: Different team members can work on different modules.

Example:

```
void getInput();
void calculateResult();
void displayResult();

int main() {
    getInput();
    calculateResult();
    displayResult();
    return 0;
}
```

Each function here performs a specific role and can be modified without affecting others.

1.10.3 Use of Functions for Structure

Functions are the main tool for implementing structured and modular programming in C. A **function** is a block of code that performs a specific task and can be called when needed.

Syntax:

```
return_type function_name(parameters) {
    // function body
}
```

Example:

```
int add(int a, int b) {
```

```
    return a + b;
}

int main() {
    int result = add(5, 3);
    printf("Sum = %d", result);
    return 0;
}
```

Functions:

- **Reduce code duplication**
- **Improve readability**
- **Help in debugging and testing**
- **Enable modular design**

1.10.4 Importance of Comments and Documentation

Comments are **non-executable** parts of the code that help others (or your future self) understand what the code is doing. Documentation is important in every stage of development for:

- **Explaining logic and structure**
- **Making code maintainable**
- **Helping with debugging**
- **Supporting team collaboration**

Without comments, even the original author may forget the logic after some time.

1.10.5 Commenting Styles and Best Practices

Types of Comments in C:

1. **Single-line comment**:
2. // This is a single-line comment
3. **Multi-line comment**:
4. /* This is a
5. multi-line comment */

Best Practices:

- **Write comments that explain "why", not just "what"**
- // Checking if user input is valid
- if (input >= 0 && input <= 100) { ... }

- **Comment complex logic, not obvious code**
  Avoid this:
- i++; // increment i by 1
- **Update comments if code changes**
  Outdated comments are worse than no comments.
- **Use consistent formatting**
  Example:
- // Function: calculateAverage
- // Purpose: Calculates average of two numbers
- **Avoid over-commenting**
  Write only where necessary, to explain intent or tricky logic.

1.11 Summary

Unit 1 introduces students to the foundational concepts of C programming, beginning with an understanding of programming in the context of systems and embedded development. It explores the evolution of programming languages and the historical importance of C, emphasizing its power, portability, and influence on modern computing. The unit explains the basic structure of a C program, covering syntax, data types, variables, constants, and the usage of operators and expressions. Input and output functions are introduced, followed by decision-making structures such as if-else, switch-case, and conditional operators. The unit also explains looping mechanisms for repeated execution of code and control statements like break, continue, and goto. Finally, the unit discusses the principles of structured programming, modular design using functions, and the importance of code documentation and commenting. Altogether, this unit lays a solid foundation for building well-organized and efficient C programs.

1.12 Keywords

Term	Definition
Programming	Writing instructions for computers to execute tasks
Embedded System	A system with dedicated functionality, often with real-time constraints
C Language	A high-level, structured programming language developed in the 1970s
Syntax	Set of rules defining valid statements in a language
Variable	Named memory location for storing data
Constant	A value that does not change during program execution
Keyword	Reserved word in C with predefined meaning
Data Type	Defines the type of value a variable can hold

Term	Definition
Operator	Symbol that performs operations on variables or values
Expression	A combination of variables, constants, and operators
Function	A block of code designed to perform a specific task
Conditional Statement	Allows execution based on a condition (e.g., if, switch)
Loop	Repeats a block of code multiple times
break	Terminates loop or switch prematurely
continue	Skips the current iteration of a loop
goto	Unconditional jump to a labeled statement (discouraged in modern code)
Modularity	Dividing a program into smaller, manageable parts
Comment	Non-executable text that explains code
Preprocessor Directive	Instruction to the compiler to process before compilation (e.g., #include)
Header File	A file containing declarations, included using #include

1.13 Self-Assessment Questions (Subjective & Case-Based)

Short Answer Questions:

1. Define structured programming. How does it improve program design?
2. What is the difference between a variable and a constant in C?
3. Explain the difference between if, if-else, and switch statements.
4. What is the role of a main() function in a C program?
5. Describe the purpose and syntax of the for loop in C.

Long Answer Questions:

6. Discuss the history of C language. Why is it still used in modern programming?
7. Compare and contrast while and do-while loops with examples.
8. Explain the use of different data types in C with relevant examples.
9. Write a program to read a number from the user and check whether it is even or odd using conditional statements.
10. What are storage classes in C? Explain the scope and lifetime of variables using examples.

Case-Based Question:

11. A developer is building a menu-driven ATM interface using C. The program needs to:

- Display options (withdraw, deposit, check balance)
- Use conditional statements for each option
- Validate input and loop back if the input is invalid
- Use file I/O to log transactions (covered in future modules)

Question:
Based on your understanding from Unit 1, how would you structure the logic of such a program using control statements, loops, and modular design?

1.14 Case Study
Case Study: Embedded Traffic Light Controller
A traffic light system is controlled by a microcontroller that switches between red, yellow, and green lights using a time-based algorithm. The system needs to:

- Run in a continuous loop
- Change light states based on a timer
- Use structured programming for clarity and maintainability
- Handle user input for manual override (e.g., pedestrian button)

Analysis Based on Unit 1:

- Use a while loop or for loop to run the main light cycle.
- Implement if-else or switch-case to change between light states.
- Use functions to separate timing logic, light control, and input handling.
- Use comments and meaningful variable names to make the code maintainable.
- Use constants to define time durations for each light (e.g., #define RED_TIME 30)

This application demonstrates how Unit 1 concepts like loops, conditions, modularity, and documentation are used in real-world embedded systems.

1.15 References

1. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 2nd Edition.
2. E. Balagurusamy, *Programming in ANSI C*, McGraw Hill Education.
3. Yashavant Kanetkar, *Let Us C*, BPB Publications.
4. Reema Thareja, *Programming in C*, Oxford University Press.
5. ANSI C Standards Documentation – American National Standards Institute.

6. TutorialsPoint C Language Documentation – https://www.tutorialspoint.com/cprogramming
7. GeeksforGeeks C Tutorials – https://www.geeksforgeeks.org/c-programming-language/