# Introduction to C Programming & Basic Data Types

Dive into the structure of C programs, understand the compilation process, and master fundamental data types like int, float, and char.

Instructor
Vipin Kumar Soni

Duration
100 Min

TeachToTech

Session 1 • Fundamentals

# My Academic Journey & Research — Overview

**B.Tech in Artificial Intelligence & Machine Learning**

**AI-based research in pest and disease detection**

THE ROYAL SOCIETY

**IBM Software Developer Intern**

IBM

**Python Trainer**

LEADING TOWARDS FUTURE
TEACHTOTECH

SESSION 1

# Overview & Objectives



## Key Topics

01 Features & Structure of C

02 Hello World Program Analysis

03 Compilation Process Stages

04 Program vs. Process

05 Data Types & Keywords

## Learning Outcomes

### Write & Compile

Create, compile, and execute your first C program understanding each code component.

### Understand Internals

Explain the 4-stage compilation process and distinguish between a static program and a dynamic process.

### Master Basics

Correctly choose and use fundamental data types (int, float, char) and reserved keywords.

# What is C Programming?



C is a powerful general-purpose programming language. It is fast, portable, and available on all platforms.

### Procedural Language

Follows a step-by-step approach using functions and structured code blocks to solve problems.

### Fast & Efficient

As a compiled language, C converts directly to machine code, making it extremely fast for system tasks.

### Middle-Level Language

Combines the ease of high-level languages with the hardware control capabilities of low-level assembly.

## The Origin Story

**1972**

Bell Laboratories

Created by **Dennis Ritchie** to develop the UNIX operating system. It evolved from 'B' language.

## Where is C Used?

- **Operating Systems**
- **Databases**
- **Embedded Systems**
- **Compilers**

# Key Features of C



C remains a dominant language due to its efficiency and control. Here are the core characteristics that define its power and versatility:

### Procedural & Modular

Breaks problems into functions and modules, promoting code reusability and structured programming.

### Fast & Efficient

Statically typed and compiled directly to machine code, offering high performance with minimal overhead.

### Middle-Level Language

Bridges the gap between low-level assembly (hardware access) and high-level abstractions.

### Portability

"Write Once, Compile Anywhere." C programs can run on different machines with minimal modification.

### Manual Memory Management

Offers direct control over memory allocation and deallocation (DMA) using pointers.

### Rich Libraries

Extensive set of built-in functions and libraries simplify complex programming tasks.

# Structure of a C Program



1. **Documentation**
   Comments describing the program's purpose, author, and date.

2. **Preprocessor Section**
   Includes header files (e.g., `stdio.h`) before compilation.

3. **Definition Section**
   Defines symbolic constants using `#define`.

4. **Global Declaration**
   Variables and functions accessible throughout the program.

5. **Main Function**
   The entry point where program execution begins.

6. **Sub Programs**
   User-defined functions performing specific tasks.

sum.c

```c
/* Documentation: Program to add number to constant */

#include <stdio.h>
#define X 20

int sum(int y);

int main(void) {
    int y = 55;
    printf("Sum: %d\n", sum(y));
    return 0;
}

int sum(int y) {
    return y + X;
}
```

# Your First C Program



### Preprocessor Directive

`#include` tells the compiler to insert the standard input/output library `stdio.h` before compilation.

### Entry Point

`main()` is where program execution begins. The `int` indicates it returns an integer value to the OS.

hello.c

```c
#include <stdio.h>

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

### Output Function

`printf()` displays text to the console. `\n` creates a new line.

### Exit Status

`return 0;` indicates the program finished successfully. Non-zero values imply errors.

▶ Compile & Run

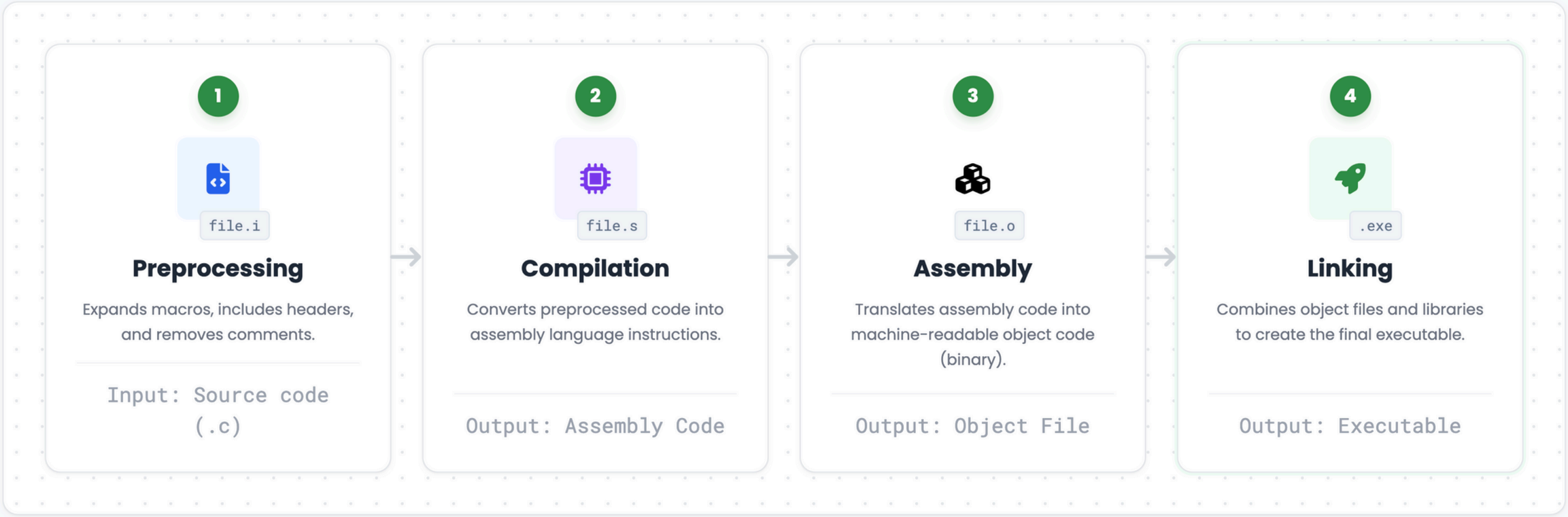
Terminal

```
user@pc:~$ gcc hello.c -o hello
user@pc:~$ ./hello
Hello, World!
```

# The Compilation Process



The journey from human-readable code to machine-executable binary involves four distinct stages. Understanding this pipeline helps in debugging and optimization.




```
$ gcc -save-temps hello.c -o hello
```

Use -save-temps to see intermediate files (.i, .s, .o)

# Program vs. Process



### Program

A passive entity stored on the disk. It is a collection of instructions (code) designed to perform a specific task.

### Process

An active entity. It is the actual execution of a program, loaded into memory with resources allocated.

| FEATURE / ASPECT | PROGRAM | | PROCESS |
|---|---|---|---|
| Nature | **Passive Entity** (Stored on disk) | VS | **Active Entity** (Running in RAM) |
| Lifetime | Permanent until deleted manually | VS | Temporary (exists only during execution) |
| Resources | Requires memory storage (disk sp | VS | Needs CPU, RAM, I/O devices |
| Analogy | A recipe written in a book | VS | The act of cooking the meal |

A single program can be executed as multiple separate processes simultaneously.

# Fundamental Data Types



C provides primitive data types to handle different kinds of values, from simple numbers to characters.

### Integer (`int`)

`Whole Numbers`

Stores counting numbers (positive or negative) without decimals.

Ex: 10, -5, 2025

### Floating Point (`float, double`)

`Decimals`

Stores numbers with fractional parts. `double` offers higher precision than `float`.

Ex: 3.14, -0.05, 99.9

### Character (`char`)

`Single Text`

Stores a single letter, digit, or symbol. Internally stored as ASCII integer.

Ex: 'A', 'z', '$'

### The Void Type

Represents the **absence of type**. It specifies that there is no value available.

```
void main() {
  // Returns nothing
}
```

### The `sizeof()` Operator

A unary operator used to compute the size of its operand in bytes. Essential for portability.

| sizeof(int) | sizeof(char) |
| --- | --- |
| **~4 bytes** | **1 byte** |

# Integer Data Type (int)



### 1 Definition

Stores whole numbers (positive, negative, or zero) without any fractional part. Declared using the `int` keyword.

### 2 Memory Size

Typically occupies **4 bytes** (32 bits) of memory on modern compilers. This supports values roughly between -2.1 billion to +2.1 billion.

### 3 Format Specifier

Uses the `%d` (decimal) or `%i` placeholder in `printf` and `scanf` functions.

### 4 Usage

Ideal for counting items, array indices, ages, or any mathematical operation requiring whole numbers.

integer_example.c

```c
#include <stdio.h>

int main(void) {
    // 1. Declare and initialize integer variables
    int age = 20;
    int year = 2025;
    int balance = -500; // Can hold negative values

    // 2. Print integer values using %d
    printf("Age: %d years\n", age);
    printf("Current Year: %d\n", year);
    printf("Account Balance: %d\n", balance);

    // 3. Size check
    printf("Storage size: %zu bytes\n", sizeof(int));

    return 0;
}
```

Terminal Output:

```
Age: 20 years
Current Year: 2025
Account Balance: -500
Storage size: 4 bytes
```

# Character Data Type (char)



### 1 Definition

The `char` keyword is used to declare character variables. It stores a single character value.

### 2 Memory Size & Range

Typically occupies **1 byte** of memory. It holds ASCII values ranging from -128 to 127 (signed) or 0 to 255 (unsigned).

### 3 Format Specifier

Use `%c` in `printf` and `scanf` functions to handle character I/O.

### 4 Single Quotes

Character literals must be enclosed in single quotes (e.g., `'A'`), unlike strings which use double quotes.

char_example.c

```c
#include <stdio.h>

int main(void) {
    // Declare and initialize a character variable
    1 char grade = 'A';

    // Printing the character
    printf("Student Grade: %c\n", grade);

    // Characters are stored as ASCII integers internally
    printf("ASCII value of %c is %d\n", grade, grade);

    // Char size in memory
    printf("Size of char: %lu byte\n", sizeof(char));

    return 0;
}
```

OUTPUT

```
Student Grade: A
ASCII value of A is 65
Size of char: 1 byte
```

# Float Data Type



### 1 Definition

Used to store single-precision floating-point numbers (decimal values). Ideal for numbers requiring fractional parts.

### 2 Storage Size

Typically occupies **4 bytes** (32 bits) of memory.

### 3 Format Specifier

Uses `%f` for printing and scanning values.

### 4 Precision & Range

Precision of 6 decimal places. Range approx 1.2E-38 to 3.4E+38.

float_example.c

```c
#include <stdio.h>

int main(void) {
    // Declaration of float variable
    float height = 5.75f;
    float price = 19.99f;

    // Printing float values using %f
    printf("Height: %f feet\n", height);

    // Controlling precision (2 decimal places)
    printf("Price: $%.2f\n", price);

    return 0;
}
```

```
$ gcc float_example.c -o float_ex && ./float_ex
Height: 5.750000 feet
Price: $19.99
```

# Double Data Type



### 1 Double Precision

Stores floating-point numbers with high precision (approx. 15-16 decimal digits).

### 2 Size & Memory

Typically occupies `8 bytes` (64 bits) of memory, offering greater range than float.

### 3 Format Specifier

Use `%lf` in `printf` and `scanf` for input/output operations.

### 4 Scientific Usage

Ideal for scientific calculations requiring high accuracy, like physics simulations or financial apps.

double_example.c

```c
/* Example: Using double for high precision */
#include <stdio.h>

int main(void) {
    /* Declaring double variables */
    double pi = 3.141592653589793;
    double scientific = 1.23e-5;  /* Scientific notation */

    /* Printing with %lf format specifier */
    printf("Value of Pi: %lf\n", pi);

    /* Controlling precision: %.15lf for 15 decimals */
    printf("Precise Pi: %.15lf\n", pi);

    printf("Scientific: %lf\n", scientific);

    return 0;
}
```

# Data Type Comparison



### Integral Types

Stores whole numbers and characters. Exact precision.

### Floating-Point Types

Stores real numbers with fractional parts. Varying precision.

| DATA TYPE | SIZE (Bytes) | TYPICAL RANGE | FORMAT / PRECISION |
|---|---|---|---|
| A `char` | 1 Byte | -128 to 127 | `%c` |
| # `int` | 4 Bytes (usually) | -2,147,483,648 to 2,147,483,647 | `%d` |
| % `float` | 4 Bytes | 1.2E-38 to 3.4E+38 | `%f` 6 decimal places |
| ∞ `double` | 8 Bytes | 2.3E-308 to 1.7E+308 | `%lf` 15 decimal places |

Sizes may vary depending on the compiler and architecture (32-bit vs 64-bit). Use `sizeof()` to verify.

# C Keywords

32 Reserved Words



## Data Types & Modifiers

12 keywords

Define variable types and their properties (size, sign, range).

`char` `double` `float` `int` `void` `long` `short` `signed` `unsigned` `enum` `struct` `union`

## Control Flow

12 keywords

Manage program execution path, loops, and decision making.

`if` `else` `switch` `case` `default` `for` `do` `while` `break` `continue` `goto` `return`

## Storage Classes

4 keywords

Define scope, visibility, and lifetime of variables.

`auto` `extern` `register` `static`

## Others & Qualifiers

4 keywords

Type qualifiers, size determination, and type aliasing.

`const` `volatile` `sizeof` `typedef`

Keywords are case-sensitive and cannot be used as variable names.

ANSI C Standard (C89/C90)

# Variables and Constants



### 1 Variables

Named storage locations in memory whose value **can change** during program execution. They act as containers for data.

### 2 Declaration Syntax

Defined by specifying a data type followed by a name: `type variable_name = value;`

### 3 Constants (const)

Fixed values that **cannot be altered** once defined. The `const` keyword enforces read-only status.

### 4 Best Practices

Use **camelCase** for variables (e.g., `userAge`) and **UPPERCASE** for constants (e.g., `MAX_SIZE`) to improve readability.

vars_consts.c

```c
#include <stdio.h>

int main() {
    // Variable: Value can change
    int count = 10;
    count = 20;  // This is allowed

    // Constant: Value is fixed
    const double PI = 3.14159;

    // PI = 3.14; // Error: assignment of read-only variable

    printf(\"Count: %d\n\", count);
    printf(\"PI: %.5f\n\", PI);

    return 0;
}
```

# Hands-on Activities



## 01 Hello World

Write, compile, and execute your first C program. Verify the output matches exactly.

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

→ **Goal: Successful Compilation**

## 02 Data Types & Specifiers

Declare variables of different types (`int`, `float`, `char`) and print them.

```
printf("%d, %f, %c", age, pi, grade);
```

→ **Goal: Correct Format Specifiers**

## 03 Memory Inspector

Use the **`sizeof()`** operator to check memory allocation for different types.

```
printf("Int size: %lu", sizeof(int));
```

→ **Goal: Understand Architecture**

## 04 Experimentation

Observe what happens when you exceed the range of an integer or mix data types.

```
short x = 32768; // What happens?
```

→ **Goal: Observe Overflow**

*Open your terminal or IDE to complete these tasks*

# Key Takeaways



## C Language Features

- **Middle-Level:** Combines high-level constructs with low-level memory access.
- **Portable & Fast:** Compiles to efficient machine code for various architectures.
- **Procedural:** Structured, modular approach using functions.

## Program Structure

- **6 Key Sections:** Documentation, Preprocessor, Definition, Global Declaration, Main, Subprograms.
- **Entry Point:** Execution always starts at `main()`.

## Compilation Process

- **Pre-processing:** Expands macros (`.i`).
- **Compiling:** C to Assembly (`.s`).
- **Assembling:** Assembly to Object (`.o`).
- **Linking:** Creates Executable (`.exe`).

## Data Types

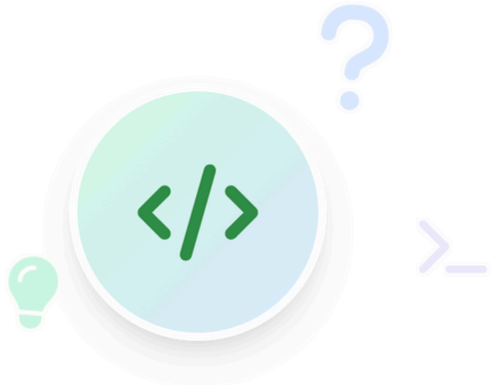
- **Primary Types:** `int`, `char`, `float`, `double`, `void`.
- **Sizing:** Architecture-dependent (e.g., int is typically 4 bytes).
- **Specifiers:** Use `%d`, `%c`, `%f` for I/O.

## Keywords

- **32 Reserved Words:** Cannot be used as variable names.
- **Categories:** Data types, control flow, storage classes.
- **Case Sensitive:** Always lowercase (e.g., `if`, `while`).

## Vars vs. Constants

- **Variable:** Named storage whose value can change.
- **Constant:** Fixed value, declared with `const` or `#define`.
- **Process vs Program:** Dynamic execution vs. Static file.



# Thank You!

We've covered the basics of C programming.
Now it's time to turn theory into practice.

### ⚡ Next Challenge

Complete the **Hands-on Activities** on Page 17. Try writing a program that calculates the area of a circle using `float` and `const`.

GOT QUESTIONS?

**Start Q&A**