Mastering C Programming

A Complete Course for Students

From fundamentals to advanced concepts — everything you need to write powerful, efficient C programs with confidence.

FULL COURSE BEGINNER TO ADVANCED

Introduction to C Programming

Why Learn C?

C is often called the **"mother of all languages"** — it powers operating systems (Linux, Windows), embedded systems, and compilers. Understanding C gives you deep insight into how computers actually work.

Your First Program: Hello World

```
#include <stdio.h>

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

#include <stdio.h> — Imports the standard I/O library

int main() — Entry point of every C program

return 0 — Signals successful program termination

Setup

Install **gcc** or **clang**. Use VS Code, Code::Blocks, or any terminal-based editor.

Compile & Run

`gcc hello.c -o hello` then `./hello`

Variables, Data Types & Input/Output

int

Whole numbers

`int age = 20;`

float / double

Decimal numbers

`float gpa = 3.8;`

char

Single character

`char grade = 'A';`

const

Immutable value

`const int MAX = 100;`

Simple Calculator — Input/Output Example

```c
#include <stdio.h>

int main() {
    float a, b, sum;
    printf("Enter two numbers: ");
    scanf("%f %f", &a, &b);
    sum = a + b;
    printf("Sum = %.2f\n", sum);
    return 0;
}
```

Key Concepts Explained

→ **scanf()** reads user input; **&** passes the memory address of the variable

→ **printf()** uses format specifiers: `%d` (int), `%f` (float), `%c` (char)

→ `%.2f` formats the float to **2 decimal places** for clean output

Operators and Control Flow

Operator Categories

Arithmetic

`+ - * / %`

Relational

`== != > < >= <=`

Logical

`&& || !`

Assignment

`= += -= *= /=`

Grade Evaluator: if-else + switch

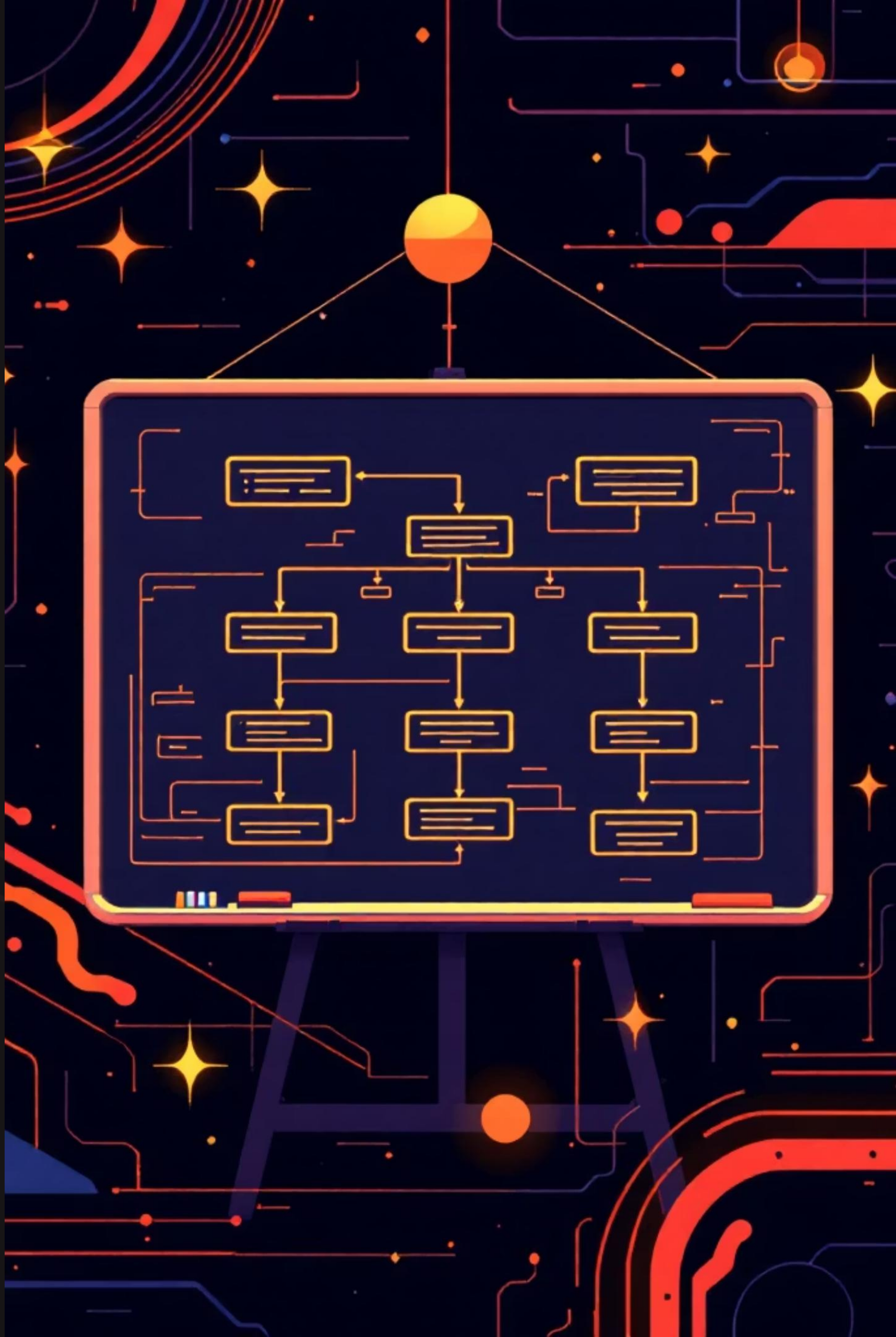
```
#include <stdio.h>

int main() {
    int marks;
    printf("Enter marks: ");
    scanf("%d", &marks);

    char grade;
    if (marks >= 90)      grade = 'A';
    else if (marks >= 75) grade = 'B';
    else if (marks >= 60) grade = 'C';
    else                   grade = 'F';

    switch (grade) {
        case 'A': printf("Excellent!\n"); break;
        case 'B': printf("Good Job!\n");  break;
        case 'C': printf("Keep Going!\n");break;
        default:  printf("Needs Work.\n");
    }
    return 0;
}
```

Tip: Always use `break` inside `switch` cases to prevent fall-through execution into the next case.



Loops and Iteration

for loop
Best when the **number of iterations is known**.
`for(i=0; i<n; i++)`

while loop
Runs as long as a **condition is true**; checks before executing.
`while(x > 0)`

do-while loop
Executes **at least once**, then checks the condition.
`do { } while(cond);`

Number Guessing Game

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

int main() {
    int secret = 42, guess, tries = 0;
    do {
        printf("Guess the number: ");
        scanf("%d", &guess);
        tries++;
        if (guess < secret) printf("Too low!\n");
        else if (guess > secret) printf("Too
high!\n");
    } while (guess != secret);
    printf("Correct in %d tries!\n", tries);
    return 0;
}
```

Loop Control Keywords

The `do-while` loop is perfect here — the game must run **at least once** before checking the exit condition.

break
Immediately **exits** the loop, skipping remaining iterations entirely

continue
Skips the current iteration and jumps to the next loop cycle

Functions and Variable Scope

Arithmetic Functions Program

```c
#include <stdio.h>

float add(float a, float b) { return a + b; }
float sub(float a, float b) { return a - b; }
float mul(float a, float b) { return a * b; }
float divd(float a, float b){
    if(b == 0) { printf("Error!\n"); return 0; }
    return a / b;
}

int main() {
    float x = 10, y = 5;
    printf("Add: %.1f\n", add(x, y));
    printf("Sub: %.1f\n", sub(x, y));
    printf("Mul: %.1f\n", mul(x, y));
    printf("Div: %.1f\n", divd(x, y));
    return 0;
}
```

Function Anatomy

01

Return Type

Specifies what data type the function returns (e.g., `int`, `float`, `void`)

02

Parameters

Inputs passed into the function; they are **local** to that function only

03

return Statement

Sends a value back to the caller and exits the function

Scope Rule: Local variables exist only inside their function. **Global variables** are accessible everywhere but should be used sparingly.

Arrays and Strings

1D Array

A linear list of elements of the same type.
`int nums[5] = {1,2,3,4,5};`

2D Array

A matrix/grid of elements.
`int mat[3][3];`

Strings

Character arrays ending with `'\0'`.
`char name[20] = "Alice";`

Text Processing Example

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

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

    printf("Length : %lu\n", strlen(str));
    printf("Upper  : ");
    for(int i = 0; str[i]; i++)
        printf("%c", str[i]-32*(str[i]>='a'));

    char rev[50];
    int n = strlen(str);
    for(int i=0;i<n;i++) rev[i]=str[n-1-i];
    rev[n]='\0';
    printf("\nReversed: %s\n", rev);
    return 0;
}
```

Common String Functions (string.h)

strlen(s)

Returns the length of the string

strcpy(d,s)

Copies string `s` into destination `d`

strcmp(a,b)

Compares two strings; returns 0 if equal

strcat(d,s)

Appends string `s` to end of `d`

Pointers and Dynamic Memory

Dynamic Array with Pointers

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

int main() {
    int n;
    printf("Array size: ");
    scanf("%d", &n);

    int *arr = (int*)malloc(n * sizeof(int));
    if(arr == NULL) {
        printf("Allocation failed!\n");
        return 1;
    }
    for(int i = 0; i < n; i++) arr[i] = i * 10;
    for(int i = 0; i < n; i++)
        printf("arr[%d] = %d\n", i, arr[i]);

    free(arr); // Always free memory!
    return 0;
}
```

Pointer Essentials

Declaration: `int *p;` — `p` holds a memory address

Address-of: `p = &x;` — assigns address of `x` to `p`

Dereference: `*p` — reads/writes the value at the address

malloc Allocates uninitialized memory block	**calloc** Allocates zero-initialized memory
realloc Resizes an existing allocation	**free** Releases memory back to OS

Golden Rule: Every `malloc`/`calloc` must have a matching `free()` to prevent memory leaks.

Structures, Unions & File I/O

Student Record System

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

typedef struct {
    int id;
    char name[50];
    float gpa;
} Student;

int main() {
    Student s = {1, "Alice", 3.9};

    // Write to file
    FILE *fp = fopen("students.dat", "w");
    fprintf(fp, "%d %s %.2f\n", s.id, s.name,
s.gpa);
    fclose(fp);

    // Read from file
    FILE *fr = fopen("students.dat", "r");
    Student r;
    fscanf(fr, "%d %s %f", &r.id, r.name, &r.gpa);
    printf("ID:%d Name:%s GPA:%.2f\n",
           r.id, r.name, r.gpa);
    fclose(fr);
    return 0;
}
```

Core Concepts

struct

Groups **different data types** under one name. Members have their own memory space.

union

Like struct but all members **share the same memory** — only one active at a time.

typedef

Creates a cleaner **alias** so you can write `Student` instead of `struct Student`

fopen / fclose

Open and close a file safely

fprintf / fscanf

Write and read formatted file data

Wrapping Up & Next Steps

9

Core Topics

Covered end-to-end

30+

Code Examples

Real, runnable programs

∞

Projects Ahead

Your journey starts now

Best Practices to Remember

→ Always initialize variables and **free allocated memory**

→ Use **meaningful variable names** and add comments for clarity

→ Compile with `-Wall -Wextra` to catch hidden warnings

→ Test edge cases: zero, negatives, empty strings, large inputs

What to Explore Next

Systems Programming

OS internals, Linux kernel, device drivers

Competitive Coding

LeetCode, Codeforces — sharpen your C skills

Data Structures

Linked lists, trees, graphs in pure C

> **"The only way to learn a new programming language is by writing programs in it."** — Dennis Ritchie, Creator of C