C Programming

1️⃣ C Basics (Foundation Level)

Learn these first:

Basic Syntax

- Keywords
- Variables
- Data types
- Constants
- Input/Output (printf, scanf)

Operators

- Arithmetic
- Relational
- Logical
- Bitwise
- Assignment
- Ternary operator

Control Flow

- if, else, switch
- for, while, do-while
- break, continue

Functions

- Function declaration & definition
- Function call

- Arguments & return values
- Scope: local, global

2️⃣ Intermediate C (Must Know for Jobs)

Arrays

- 1D, 2D arrays
- Character arrays
- Multi-dimensional arrays

Strings

- String handling functions (strlen, strcpy, strcmp, etc.)
- Manual string operations

Pointers

- Pointer basics
- Pointer arithmetic
- Pointer to pointer
- Pointer to array
- Function pointers
- void* pointer

Structures & Unions

- Defining and using struct
- Nested structures
- Structures with pointers
- Unions and memory sharing

Dynamic Memory

- malloc, calloc, realloc, free
- Memory leaks
- Dangling pointers

Storage Classes

- auto, static, extern, register

Preprocessor

- #define
- Macros
- Conditional compilation (#ifdef, #endif)
- Header files

3️⃣ Advanced C Concepts (High-Level Mastery)

File Handling

- Read/write
- Binary vs Text files
- fopen, fclose, fread, fwrite

Modular Programming

- Multiple .c/.h files
- Clean code structure

Command-line Arguments

- int main(int argc, char *argv[])

Memory Management & Debugging

- Heap/stack

- Memory alignment
- Segmentation faults
- gdb debugging basics

Bit Manipulation

- Bit masking
- Bit shifting
- Set/clear/toggle bits
- This is **critical for embedded systems**

Function Pointers

- Callbacks
- Passing functions as arguments

Recursion

- Stack usage
- When to avoid recursion in embedded systems

4️⃣ Embedded C (Professional Level)

These topics turn you into an embedded software developer.

volatile keyword

- Why it is used
- Preventing compiler optimization

Memory-Mapped Registers

- Accessing peripheral registers in C
- Using pointers to hardware addresses

ISR (Interrupt Service Routines)

- Writing interrupt handlers
- Re-entrancy issues

Embedded Timing Concepts

- Timers
- Delays
- Watchdog timers

Low-Level Drivers

- GPIO
- UART
- I2C
- SPI
- PWM

Real-Time Concepts

- Polling vs Interrupts
- Cooperative vs Preemptive scheduling

5️⃣ Data Structures in C (Important for Interviews)

Even for embedded developers:

Linked List

Stack

Queue

Trees (basic)

Sorting algorithms

Searching algorithms

6️⃣ Industry-Level Concepts

Writing Makefiles

Understanding compiler toolchain

- GCC
- Linking
- Assembly + C integration

MISRA C (Coding Standard for Safety-Critical Systems)

- Automotive (ISO 26262)
- Medical devices
- Aerospace

Code Optimization

- Speed vs Memory
- Removing dead code
- Loop unrolling
- Inline functions

1️⃣ C Basics (Foundation Level)

Learn these first:

Basic Syntax

- Keywords
- Variables
- Data types
- Constants
- Input/Output (printf, scanf)

1. Basic Syntax in C

C program follows a **fixed structure**.
Every program must have:

```
#include <stdio.h>     // Library include
int main() {           // Main function begins
    printf("Hello");   // Code to execute
    return 0;          // Program ends
}
```

Key syntax rules:

- Every statement ends with **;**
- Code inside functions is written inside **{ }**
- main() is the starting point of every C program
- C is **case-sensitive** → Printf ≠ printf

#include <stdio.h>

This is a preprocessor directive - it's processed before the actual compilation of your C program.

`#include <stdio.h>` is like telling the compiler: *"Hey, I'm going to use some input/output functions.

Breakdown:

1. `#` (Hash Symbol)

- Indicates this is a preprocessor directive

- Preprocessor commands start with **'#'**

- Processed before compilation by the C Preprocessor

2. include

- **Keyword** that means **insert the contents of another file here**

- Tells the preprocessor to **copy-paste** the specified file into your program

3. <stdio.h>

- < > (Angle brackets):

  - Means look in the **standard system directories**

What's inside stdio.h

stdio.h contains **function prototypes** (declarations) for standard input/output functions:

Common Functions Declared:

// Output functions

int printf(const char *format, ...);   // Print formatted output

int putchar(int c);   // Print single character

int puts(const char *s);   // Print string

```
// Input functions
int scanf(const char *format, ...);     // Read formatted input
int getchar(void);                      // Read single character
char *gets(char *s);                    // Read string (deprecated)

// File operations
FILE *fopen(const char *filename, const char *mode);
int fclose(FILE *stream);
```

Why do we need it?

Without #include <stdio.h>

```
// This will NOT compile
int main() {
    printf("Hello World!\n");  // ERROR: printf not declared
    return 0;
}
```

Compiler Error: implicit declaration of function 'printf'

With #include <stdio.h>

```
#include <stdio.h>  // Now compiler knows about printf()

int main() {
    printf("Hello World!\n");  // OK: printf is declared
    return 0;
}
```

How it Works - Step by Step:

1. Preprocessing Stage:

- It only provides **declarations** - actual code is in the standard library
- Without it, compiler doesn't know what **`printf()`, `scanf()`**, etc. are
- Always place it at the **top** of your C file

```
// Your code:
#include <stdio.h>
int main() {
    printf("Hello\n");
    return 0;
}
// After preprocessing:
// [Entire contents of stdio.h copied here - hundreds of lines]
int printf(const char *format, ...);  // Declaration from stdio.h
int scanf(const char *format, ...);   // Another declaration
// ... many more declarations ...
int main() {
    printf("Hello\n");
    return 0;
}
```

2. Compilation Stage:

- Compiler sees the declaration: `int printf(...)`

- Knows `printf` is a valid function

- Can check if you're using it correctly

3. Linking Stage:

- Links your program with the actual **printf** function implementation
- Implementation is in the **C Standard Library** (`libc.so` or `msvcrt.dll`)

Other Include Variations:

Using `" "` (Double Quotes):

```
#include "myheader.h"  // Looks in current directory first
                       // Then in system directories
```

Multiple Includes:

```
#include <stdio.h>     // For I/O functions
#include <math.h>      // For math functions (sqrt(), sin(), etc.)
#include <string.h>    // For string functions (strlen(), strcpy(), etc.)


int main() {
    printf("Square root of 16: %.2f\n", sqrt(16));
    return 0;
}
```

Common Mistakes:

1. Wrong Case:

```
#include <Stdio.h>   // WRONG - C is case-sensitive
#include <STDIO.H>   // WRONG
#include <stdio.h>   // CORRECT
```

2. Missing Angle Brackets:

```
include <stdio.h>    // WRONG - missing #
```

```
#include stdio.h     // WRONG - missing <>
```

3. Unnecessary Semicolon:

```
#include <stdio.h>;  // WRONG - no semicolon needed
```

Real Example:

```
#include <stdio.h>  // Tells compiler: "I'll use I/O functions"
int main() {
    int number;
    // printf() and scanf() come from stdio.h
    printf("Enter a number: ");
    scanf("%d", &number);
    printf("You entered: %d\n", number);
    return 0;
}
```

2. Keywords

Keywords are **reserved words** that have special, predefined meanings in C. You **cannot** use them as variable names, function names, or any other identifiers.

Complete List of 32 Keywords in C

Keyword	Category	Description
auto	Storage class	Declares an automatic (local) variable
break	Control flow	Exits loop or switch immediately
case	Control flow	Defines a case inside a switch
char	Data type	Character data type

Keyword	Category	Description
const	Type qualifier	Makes variable value constant
continue	Control flow	Skips to next loop iteration
default	Control flow	Default case in switch
do	Loop	Starts a do–while loop
double	Data type	Double-precision floating number
else	Control flow	Else part of if statement
enum	Data type	User-defined integer constants
extern	Storage class	Declares external/global variable
float	Data type	Single-precision floating number
for	Loop	For loop
goto	Control flow	Jumps to a labeled statement
if	Control flow	Conditional execution
int	Data type	Integer data type
long	Type modifier	Long integer type
register	Storage class	Requests variable to be stored in CPU register
return	Function	Returns value from function
short	Type modifier	Short integer type
signed	Type modifier	Signed integer type
sizeof	Operator	Gets memory size of variable/type
static	Storage class	Preserves value across function calls
struct	Data type	Structure — group of variables
switch	Control flow	Multi-way selection
typedef	Type definition	Creates new type alias
union	Data type	Shares memory among variables
unsigned	Type modifier	Unsigned integer type
void	Data type	No value / empty
volatile	Type qualifier	Prevents compiler optimization
while	Loop	While loop

Detailed Categories with Examples:

1. Data Type Keywords

Define the type of data a variable can hold.

```
char grade = 'A';                // Character (1 byte)
int age = 25;                // Integer (usually 4 bytes)
float price = 99.99;      // Floating point (4 bytes)
double pi = 3.14159265;  // Double precision (8 bytes)
void no_return();         // No return value
```

2. Type Modifiers

Modify the size or sign of basic data types.

```
short small_num = 100;      // Short integer (2 bytes)
long big_num = 100000L;            // Long integer (8 bytes)
signed int s = -10;                        // Can be negative (default)
unsigned int u = 100;          // Only positive (0 to 2^32-1)
long double ld = 3.14159;   // Extended precision
```

3. Control Flow Keywords

Control the flow of program execution.

a) // if-else

```
if (age >= 18) {
    printf("Adult\n");
} else {
    printf("Minor\n");
}
```

b) // switch-case

```
switch (grade) {
    case 'A':
        printf("Excellent\n");
        break;
    case 'B':
        printf("Good\n");
        break;
    default:
        printf("Average\n");
}
```

c) // goto (use sparingly!)

```
start:
    printf("Enter positive number: ");
    scanf("%d", &num);
    if (num <= 0) goto start;
```

4. Loop Keywords

Create repetitive execution.

a) // for loop

```
for (int i = 0; i < 5; i++) {
    printf("%d ", i);  // 0 1 2 3 4
}
```

b) // while loop

```
int count = 0;
```

```
while (count < 3) {
    printf("Hello\n");
    count++;
}
```

c) // do-while loop (executes at least once)

```
int x = 10;
do {
    printf("%d ", x);  // 10
    x++;
} while (x < 5);
```

d) // break and continue

```
for (int i = 0; i < 10; i++) {
    if (i == 5) break;     // Exit loop at 5
    if (i % 2 == 0) continue;  // Skip even numbers
    printf("%d ", i);      // 1 3
}
```

<u>5. Storage Class Keywords</u>

Determine scope, visibility, and lifetime of variables.

```
auto int a = 10;           // Default (automatic) - local scope
register int r = 20;              // Suggest storing in register (fast access)
static int counter = 0;    // Preserves value between function calls
extern int global_var;     // Declared elsewhere
void function() {
    static int calls = 0;  // Remembers value between calls
```

```
    calls++;
    printf("Called %d times\n", calls);
}
```

Keyword	Meaning	Scope	Lifetime	Where Stored	Example
auto	Default storage class for local variables	Local (inside function/block)	Until function ends	Stack	auto int a = 10;
register	Suggests storing variable in CPU register for fast access	Local	Until function ends	CPU Register (if available)	register int r = 20;
static	Preserves value between function calls	Local (if used inside function) or Global (if used outside)	Entire program execution	Static memory	static int count = 0;
extern	Refers to a global variable declared in another file or location	Global	Entire program execution	Global memory	extern int global_var;

Local variables (auto/register)

- Created **inside function**
- Destroyed when function ends
- Default is **auto**

Static variable inside function

- Created once
- Value is remembered across function calls

Extern variable

- Used to **access global variable** from another file
- Does **not** allocate memory, only references it

Example Explained

```
auto int a = 10;        // Local variable, exists only inside block
register int r = 20;    // Fast access variable, stored in CPU register
static int counter = 0; // Exists until program ends
extern int global_var;  // Refers to variable defined elsewhere
```

Static inside function:

```
void function() {
   static int calls = 0; // Created once and retains previous value
   calls++;
   printf("Called %d times\n", calls);
}
```

Output (if function is called 3 times):

```
Called 1 times
Called 2 times
Called 3 times
```

Because static preserves the value.

6. Type Qualifiers in C (const & volatile)

Type qualifiers give **special properties** to variables:

- **const** → value **cannot be changed**

- **volatile** → value **may change unexpectedly**, so compiler should not optimize it

1. const Keyword – Example Explanations

Example 1: Constant Variable

```
const int MAX_SPEED = 120;
printf("%d", MAX_SPEED);   // Allowed
MAX_SPEED = 150;           // ❌ ERROR: cannot modify const variable
```

Used when you want a **read-only** value

Prevents accidental modification

Example 2: Constant Pointer

```
int x = 10;
int y = 20;

int *const ptr = &x;   // Pointer is constant
*ptr = 15;             // Allowed: changing value of x
ptr = &y;              // ❌ ERROR: cannot make pointer point elsewhere
```

Pointer itself cannot change

But it can modify the value it points to

2. volatile Keyword – Example Explanations

Volatile tells the compiler:

> "This variable can change anytime.
> Don't optimize it or assume its value is fixed."

Used in:

- Hardware registers
- Sensors
- Interrupts
- Multi-threading

Example 1: Hardware Register (Sensor Input)

```
volatile int temperature_sensor = 0;

while (temperature_sensor < 50) {
    // Compiler must re-read sensor value each time
}
```

Sensor value can change at any time

volatile prevents compiler from caching the value

Example 2: Interrupt-Modified Variable

```
volatile int flag = 0;

void ISR() {   // Interrupt changes the variable
    flag = 1;
}

int main() {
    while (flag == 0) {
        // Without volatile, compiler may optimize this loop incorrectly
    }
}
```

- Interrupt updates flag
  Main program must always read fresh value
  volatile avoids dangerous optimizations

7. Structure/Union/Enum Keywords

Define custom data types.

Feature	Structure (struct)	Union (union)	Enum (enum)
Definition	Collection of variables of different types	Collection of variables sharing the same memory	Collection of named integer constants
Memory Usage	**Total = sum** of all member	**Total = size of largest member**	Uses size of an integer

Feature	Structure (struct)	Union (union)	Enum (enum)
	sizes		
Access	All members can be accessed at once	Only **one member valid at a time**	Represents one value from a set
Use Case	Grouping related data (e.g., student info)	Memory-efficient storage where only one data type is needed at a time	Creating meaningful names for constants (states, modes, options)
Value Stored	Each member has its own separate value	One shared storage → value of previous member is overwritten	Stores one constant from enum list
Example Use	Complex data structures	Embedded systems, device drivers	Modes, states, error codes
Keyword	struct	union	enum

1. STRUCTURE

A **structure groups multiple variables** (ints, floats, char arrays...) under one name.

Why use?

To create a complex data type that represents a real-world object.

Example

```
struct Student {
   int roll;
   float marks;
   char name[20];
};

struct Student s1 = {1, 88.5, "Arun"};
```

Memory Example

If roll=4 bytes, marks=4 bytes, name=20 bytes →
Total = 28 bytes

What is Structure Padding?

Structure padding is the process where the compiler inserts **extra unused bytes** in a struct to align data members in memory for faster CPU access.

Why does padding happen?

Different data types require different alignment (1, 2, 4, or 8 bytes).
The compiler adds gaps so each variable starts at an address the CPU can access efficiently.

Memory Alignment Rules (Simple)

Data Type	Typical Alignment (bytes)
char	1
short	2
int	4
float	4
double	8

(Varies by compiler/architecture, but this is standard.)

Example 1: Simple Structure Padding

```
struct A {
   char c;   // 1 byte
   int x;    // 4 bytes
};
```

Memory Layout (with padding)

Member	Size	Padding
char c	1 byte	**3 bytes padding**

Member	Size	Padding
int x	4 bytes	0
Total Size		**8 bytes**

Why?

- int must start at a 4-byte aligned address
- After char, CPU inserts 3 dummy bytes

Example 2: Structure Without Padding (If reordered)

```
struct B {
   int x;    // 4 bytes
   char c;   // 1 byte
};
```

Memory Layout

Member	Size	Padding
int x	4	0
char c	1	**3 bytes padding at end**
Total Size		**8 bytes**

Even after rearranging, end padding may still happen to align the structure itself.

Example 3: Big Padding Issue

```
struct C {
   char a;   // 1
   double b; // 8
   char c;   // 1
};
```

Memory Layout with Padding

Member	Size	Padding
char a	1	**7 padding bytes**

Member	Size	Padding
double b	8	0
char c	1	**7 padding bytes**
Total Size		**24 bytes**

How to Reduce Padding (Reordering Members)

Optimized version:

```
struct C2 {
    double b; // 8
    char a;   // 1
    char c;   // 1
};
```

Member	Size	Padding
double b	8	0
char a	1	0
char c	1	**6 bytes padding** at end
Total Size		**16 bytes**

🔷 Saved **8 bytes** by reordering!

🔷 Very important in **embedded systems**.

Structure Packing (Remove Padding)

We can force the compiler to **remove padding**:

```
#pragma pack(1)
struct Packed {
    char a;
    int b;
    char c;
};
#pragma pack()
```

Size becomes **1 + 4 + 1 = 6 bytes**

But slower access → CPU may do extra cycles.

When Padding Matters Most?

Field	Why Important
Embedded systems	Memory is limited
Network packets	Exact layout must match protocol
Hardware registers	Must match memory map
File I/O binary formats	Layout must not change

2. UNION

A **union stores different data types in the same memory location**.

Why use?

To **save memory**, when only **one variable will be used at a time**.

Example

```
union Data {
    int id;
    float temp;
};

union Data d;
d.id = 10;        // Setting integer
d.temp = 55.6f;   // Now "temp" overwrites "id"
```

Memory Example

int = 4 bytes, float = 4 bytes → union size = **4 bytes**, not 8.

Key Point:

Only **one member is valid at a time** because all share the same memory.

3. ENUM

enum creates a set of **named integer constants**.

Why use?

To improve **code readability** for states/modes.

Example

```
enum TrafficLight {
   RED,       // 0
   YELLOW,    // 1
   GREEN      // 2
};

enum TrafficLight signal = GREEN;
```

Custom values

```
enum ErrorCodes {
   OK = 200,
   NOT_FOUND = 404,
   SERVER_ERR = 500
};
```

8. typedef Keyword

The typedef keyword is used to **create a new name (alias) for an existing data type**.

It does **not** create a new data type —it **renames** a type to make code simpler and more readable.

Tabular

Feature	Description	Example
Purpose	Gives a new name to an existing type	typedef int myInt;
Original Type	Can be primitive or user-defined	struct, union, enum
Use Case	Simplify long declarations	typedef struct ...
Effect on Program	Improves readability	No memory/size change
Common Use	Structures, pointers, function	typedef void (*fp)();

Feature	Description	Example
	pointers	

Example 1: Typedef with Basic Types

Without typedef

```
unsigned long int distance;
```

With typedef

```
typedef unsigned long int ulong;
ulong distance;
```

Makes long declarations short and easy.

Example 2: Typedef with Structures

Without typedef

```
struct Student {
    int id;
    char name[20];
};

struct Student s1;
```

With typedef

```
typedef struct {
    int id;
    char name[20];
} Student;

Student s1;
```

No need to write the keyword **struct** every time.
This is the **most common usage** in embedded systems.

Example 3: Typedef with Pointers

Useful for clean code:

```
typedef int* IntPtr;
IntPtr p1, p2;
```

Better readability

Helpful in hardware register access

Example 4: Typedef for Function Pointer

Very important in **drivers, RTOS, callbacks**:

```
typedef void (*CallbackFunc)(int);

void display(int x) {
    printf("%d", x);
}

CallbackFunc cb = display;
```

Makes function pointers look cleaner.

When to use typedef?

Situation	Why typedef helps
Long or complex type names	Makes code shorter and cleaner
Structs/unions/enums	Avoid writing struct repeatedly
Hardware registers	Easy to create custom types
Function pointers	Avoid confusing pointer syntax
Embedded systems	Improves readability in drivers

9. return Keyword

The return keyword is used inside a function to:

Exit the function immediately

Send a value back to the caller (optional)

Tabular

Feature	Description	Example
Purpose	Ends function execution	return;
With Value	Returns a value to caller	return 10;
Without Value	Used in void functions	return;
Multiple Returns	A function may contain multiple return statements	Inside conditions
Return Type	Must match function’s declared type	int fun() → return int
Last Statement	Usually last, but can be anywhere	Mid-function allowed

Example 1: Returning a Value (Most Common)

```
int add(int a, int b) {
    return a + b;   // returns integer value
}
int result = add(5, 3);    // result = 8
```

Function returns an **int** because the return type is int.

Example 2: Using return in Conditions

```
int max(int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}
```

Function returns early depending on condition.

Example 3: void Function Using return (No Value)

```
void greet() {
```

```
    printf("Hello!");

    return; // optional
}
```

return; only exits — no value.

Example 4: Early Exit from a Function

```
int divide(int a, int b) {
    if (b == 0)
        return -1;   // exit early if invalid

    return a / b;
}
```

Useful for error checking.

Example 5: Returning Structures

```
struct Point {
    int x, y;
};

struct Point getPoint() {
    struct Point p = {10, 20};
    return p;
}
```

You can return entire structures too.Important Rules

Rule	Explanation
Must match return type	int function() must return an int
Every path must return a value	Otherwise compiler gives warning
main() must return an int	Usually return 0;
Can have more than one return	But too many returns reduce readability

10. sizeof Operator**

Get size of type/variable in bytes.

```c
printf("Size of int: %lu bytes\n", sizeof(int));
printf("Size of float: %lu bytes\n", sizeof(float));

int arr[10];
printf("Size of array: %lu bytes\n", sizeof(arr));
```

🚫 Invalid Usage - Common Errors:

```c
// ERROR: Keywords cannot be variable names
int int = 10;         // ❌ 'int' is a keyword
float float = 3.14;   // ❌ 'float' is a keyword
char char = 'A';      // ❌ 'char' is a keyword

// ERROR: Keywords cannot be function names
void if() { }         // ❌ 'if' is a keyword
int return() { }      // ❌ 'return' is a keyword

// CORRECT: Use meaningful names instead
int integer_value = 10;     // ✅
float price_value = 3.14;   // ✅
```

char character = 'A';   // ✅
```

## 📝 Special Notes:

### **1. Case Sensitivity:**
```c
Int num1;    // ✅ Valid (not a keyword - C is case-sensitive)
INT num2;    // ✅ Valid
int num3;    // ✅ Valid (keyword)
```

2. Compiler Variations:
- Some compilers may have additional keywords (like `asm`, `inline`)
- C99 added: `_Bool`, `_Complex`, `_Imaginary`, `restrict`
- C11 added: `_Alignas`, `_Alignof`, `_Atomic`, `_Generic`, `_Noreturn`, `_Static_assert`, `_Thread_local`

3. Underscore Keywords:
```c
// Some compilers support:
_Bool flag = 1;        // Boolean type (C99)
_Complex double z;     // Complex number
_Atomic int counter;   // Atomic type (C11)
```
```

💡 Memory Aid:

To remember all 32 keywords, here's a mnemonic:

"Auto Breaks Case Char Const Continue Default Do Double Else Enum Extern Float For Goto If Int Long Register Return Short Signed Sizeof Static Struct Switch Typedef Union Unsigned Void Volatile While"

Or group them:

- **Data types**: char, double, float, int, void
- **Modifiers**: long, short, signed, unsigned
- **Loops**: for, while, do
- **Conditionals**: if, else, switch, case, default
- **Jump**: break, continue, goto, return
- **Storage**: auto, extern, register, static
- **Others**: const, enum, sizeof, struct, typedef, union, volatile

🎯 Key Takeaways:

1. Keywords are **reserved words** with special meaning
2. There are **32 standard keywords** in ANSI C
3. **Cannot** be used as identifiers
4. All keywords are **lowercase**
5. Understanding keywords is essential for writing valid C programs