Comprehensive Notes on C Programming (GATE Perspective)

Your Name

March 19, 2025

Contents

1 Introduction to C

1.1 Key Features

- Developed by **Dennis Ritchie** in 1972 at Bell Labs as a successor to B.
- **Structure**: Every program requires a `main()` function as the entry point; execution begins here.
- **Portability**: Standardized libraries (e.g., `stdio.h`) allow code to run across platforms with minimal changes.
- **Low-Level Access**: Provides direct memory manipulation via pointers, making it ideal for system programming (e.g., OS, embedded systems).

1.2 Underlying System: Compilation Process

C is a compiled language, and understanding its compilation process is key for GATE:

1. **Preprocessing**: Expands macros (`#define`), includes header files (`#include`), and removes comments.
2. **Compilation**: Translates preprocessed code into assembly language specific to the target architecture.
3. **Assembly**: Converts assembly code into machine code, producing an object file (e.g., `.o`).
4. **Linking**: Combines object files with libraries (e.g., `libc`) to create an executable.

GATE Note: Be familiar with errors like undefined references (linking) or macro redefinition (preprocessing).

2 Identifiers and Keywords

2.1 Rules for Identifiers

- Must begin with a letter (A-Z, a-z) or underscore (_); digits (0-9) allowed after the first character.
- Cannot start with a digit or use special symbols (e.g., @, #).

- Case-sensitive: `SUM` and `sum` are distinct variables.
- Length limit: Typically 31 characters (compiler-dependent).

2.2 Keywords

Category	**Examples**
Data Types	`int, char, float, double, void`
Control Flow	`if, else, switch, while, for, return`
Storage	`auto, static, extern, register`

GATE Note: Questions may ask to spot invalid identifiers (e.g., `int` as a variable name).

3 Data Types

3.1 Basic Data Types

Type	**Size (Bytes)**	**Range**	**Use**
`char`	1	-128 to 127 (signed)	ASCII characters
`unsigned char`	1	0 to 255	Extended characters
`int`	2 or 4	-32,768 to 32,767 (2B) or -2^{31} to 2^{31}-1 (4B)	Integers
`unsigned int`	2 or 4	0 to 65,535 (2B) or 0 to 2^{32}-1 (4B)	Positive integers
`float`	4	3.4E-38 to 3.4E+38	Single-precision floating-point
`double`	8	1.7E-308 to 1.7E+308	Double-precision floating-point

Underlying System: Size depends on the architecture (e.g., 32-bit vs 64-bit systems).

3.2 Modifiers

- `signed`: Default; includes negative values.
- `unsigned`: Positive values only, doubles the positive range.
- `short`: Reduces size (e.g., `short int`: 2 bytes).
- `long`: Increases size (e.g., `long int`: 4 or 8 bytes).

GATE Note: Questions may test ranges with modifiers (e.g., `unsigned short int`).

3.3 Type Conversion

- **Implicit**: Automatic (e.g., `int` to `float` in `3 + 2.5`).
- **Explicit**: Cast using `(type)` (e.g., `(int)3.14` = 3).

Underlying System: Type promotion follows a hierarchy (e.g., `char` → `int` → `float`).

4 Operators

4.1 Operator Precedence (Top 5)

1. Parentheses `()`, `[]`, `.`, `->`
2. Unary `++`, `--`, `!`, `~`, `sizeof`
3. Multiplicative `*`, `/`, `%`
4. Additive `+`, `-`
5. Relational `<`, `>`, `<=`, `>=`

GATE Note: Evaluate expressions like `a+++b` (post-increment vs addition).

4.2 Bitwise Operators

- `&` (AND), `|` (OR), `^` (XOR), `~` (NOT).
- `<<` (Left Shift): Shifts bits left, multiplies by 2 per shift.
- `>>` (Right Shift): Shifts bits right, divides by 2 (signed vs unsigned differs).

Example: `5 & 3 = 1` (Binary: `101 & 011 = 001`).

4.3 Logical vs Bitwise

- **Logical** (`&&`, `||`, `!`): Evaluates to 0 or 1; short-circuits.
- **Bitwise**: Operates on each bit; no short-circuiting.

GATE Note: Compare `if (a & b)` vs `if (a && b)`.

5 Control Statements

5.1 Decision Control

- `if-else`: Supports nesting; evaluates conditions sequentially.
- `switch-case`: Integer-based; `break` prevents fall-through.

Underlying System: `switch` compiles to jump tables for efficiency.

5.2 Loops

Loop	Use Case
`while`	Pre-test; condition checked first
`do-while`	Post-test; runs at least once
`for`	Counter-controlled; compact syntax

GATE Note: Analyze loop termination (e.g., infinite loops).

5.3 Jump Statements

- `break`: Exits innermost loop or `switch`.
- `continue`: Skips to next iteration.
- `return`: Exits function with a value.

6 Functions

6.1 Parameter Passing

- **Call by Value**: Copies arguments; original variables unchanged.
- **Call by Reference**: Uses pointers; modifies original data.

Underlying System: Stack frame created for each call; parameters pushed onto stack.

6.2 Function Prototype

- **Declaration**: Specifies return type and parameters.
- **Definition**: Implements the logic.

Example:

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

6.3 Recursion

- Function calls itself with a base case.
- **Stack Usage**: Each call adds a frame to the call stack.

GATE Note: Calculate recursion depth or spot stack overflow.

7 Pointers

7.1 Basics

- **Declaration**: `int *ptr;` (points to an integer).
- **Address**: `&x` gets memory address; `ptr = &x`.
- **Dereference**: `*ptr` accesses value at address.
- **Null Pointer**: `int *ptr = NULL;` (no valid memory).

7.2 Pointer Arithmetic

- Increments by data type size (e.g., `int *ptr; ptr++` adds 4 bytes on 32-bit systems).
- **Example**: `int arr[3]; int *p = arr; p+1` points to `arr[1]`.

Underlying System: Memory is byte-addressable; pointer arithmetic scales by type size.

7.3 Common Errors

- **Dangling Pointers**: Point to freed memory (e.g., after `free()`).
- **Uninitialized Pointers**: Random address access causes crashes.
- **Memory Leaks**: Forgetting to free dynamically allocated memory.

GATE Note: Predict output involving pointer misuse.

7.4 Dynamic Memory Allocation

- `malloc()`: Allocates uninitialized memory (e.g., `int *p = (int*)malloc(4);`).
- `calloc()`: Allocates and zeros memory.
- `free()`: Releases memory back to the heap.

Underlying System: Heap-managed; OS handles memory requests.

8 Important Code Examples

8.1 Prime Number Check

```c
#include <stdio.h>
int main() {
    int n, flag = 0;
    scanf("%d", &n);
    if (n <= 1) flag = 1;
    for(int i = 2; i <= n/2; i++) {
        if (n % i == 0) { flag = 1; break; }
    }
    printf(flag ? "Composite" : "Prime");
    return 0;
}
```

8.2 Swapping using Pointers

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

8.3 Factorial using Recursion

```c
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

9 GATE Focus Areas

- **Operator Precedence**: Solve nested expressions (e.g., `*p++`).
- **Pointer Arithmetic**: Compute addresses in arrays or structures.
- **Memory Allocation**: Static (stack) vs dynamic (heap) differences.
- **Type Conversion**: Effects on arithmetic operations.
- **Recursion**: Stack overflow and time complexity.
- **Bitwise Operations**: Efficient manipulation (e.g., checking odd/even).

10 Common Pitfalls

- **Assignment vs Comparison**: `if (x = 5)` vs `if (x == 5)`.
- **Missing `break` in `switch`**: Causes unintended fall-through.
- **Forgetting `&` in `scanf`**: Leads to runtime errors.
- **Buffer Overflow**: Writing beyond array bounds.
- **Unfreed Memory**: Causes leaks in long-running programs.