Programming in C

Short notes for B.Tech CSE / IT · BCA · MCA · GATE CS

Uncodemy Handwritten Notes · free for students

1. Introduction

- **C** = a general-purpose, procedural, middle-level language made by Dennis Ritchie at Bell Labs (1972).
- Middle-level means it has high-level features (functions, loops) and low-level access (pointers, bit operations).
- Used for operating systems, compilers, embedded systems and drivers.

➤ Structure of a C program

```
#include <stdio.h>      // preprocessor directive
int main(void)          // execution starts here
{
    printf("Hello\n");  // statement
    return 0;           // 0 = success
}
```

➤ From source to output

1. **Preprocessor**: handles `#include`, `#define`, removes comments.
2. **Compiler**: converts C code to assembly / object code.
3. **Linker**: joins object files and libraries into one executable.
4. **Loader**: puts the executable in RAM to run.

EXAM POINT Compiler translates the whole program at once. Interpreter translates and runs line by line. C uses a compiler.

2. Data Types, Variables and Operators

➤ Basic data types (typical 32/64-bit system)

Type	Size	Format specifier
char	1 byte	%c
int	4 bytes (at least 2)	%d
float	4 bytes	%f
double	8 bytes	%lf
long long	8 bytes	%lld

- Only `sizeof(char) = 1` is fixed by the standard. Other sizes depend on the compiler.
- **Identifier rules**: letters, digits and `_`; cannot start with a digit; no keywords.
- **Storage of signed int**: 2's complement. For n bits, range = `-2^(n-1) to 2^(n-1) - 1`.

FORMULA Range of n-bit unsigned = 0 to 2^n - 1. Signed (2's complement) = -2^(n-1) to 2^(n-1) - 1.

➤ Operators

Group	Operators
Arithmetic	`+ - * / %`
Relational	`< > <= >= == !=`
Logical	`&&` (AND), OR (two pipe signs), `!` (NOT)
Bitwise	`&` AND, single pipe OR, `^` XOR, `~` NOT, `<<` `>>` shifts
Assignment	`= += -= *= /=`
Others	`?:` `sizeof` `,` `&` `*`

➤ Precedence (high to low, short form)

- `() [] → .` > unary (`! ~ ++ -- sizeof`) > `* / %` > `+ -` > `<< >>` > relational > `== !=` > `&` > `^` > `|` > `&&` > `||` > `?:` > assignment > `,`
- Unary, `?:` and assignment are **right to left**. Most others are left to right.

REMEMBER `&&` and `||` use short-circuit evaluation. In `a && b`, if a is false, b is never evaluated.

EXAMPLE `int x = 7 / 2;` gives 3 (integer division). `7 % 2` = 1. `7.0 / 2` = 3.5.

3. Control Statements

- **Decision**: `if`, `if-else`, `else if` ladder, `switch`.
- **Loops**: `for`, `while` (entry-controlled), `do-while` (exit-controlled, runs at least once).
- **Jumps**: `break` (leave loop/switch), `continue` (skip to next iteration), `goto`, `return`.

```
switch (ch) {
  case 'a': printf("A"); break;
  case 'b': printf("B");      // no break: falls through
  default : printf("D");
}
```

- If `ch = 'b'`, output is `BD` because of fall-through.
- `switch` works only with integer / char constants, not float or strings.

TIP In GATE output questions, count loop iterations carefully and watch for `;` right after `for(...)` which makes an empty loop body.

4. Functions and Storage Classes

- **Function** = a named block of code that does one job. Parts: declaration (prototype), definition, call.
- **Call by value**: a copy is passed, changes do not affect the caller. C uses only call by value.
- **Call by reference** is simulated by passing a pointer (address).

```
void swap(int *a, int *b) {
    int t = *a; *a = *b; *b = t;
}
// call: swap(&x, &y);
```

➤ Recursion

- A function that calls itself. Needs a **base case** to stop.
- Each call uses a new stack frame, so deep recursion can cause stack overflow.

```c
int fact(int n) {
    if (n <= 1) return 1;     // base case
    return n * fact(n - 1);
}
```

Storage classes

Class	Storage	Default value	Scope	Lifetime
auto	Stack	Garbage	Block	Till block ends
register	CPU register (request)	Garbage	Block	Till block ends
static	Data segment	0	Block / file	Whole program
extern	Data segment	0	Global	Whole program

EXAM POINT A `static` local variable keeps its value between function calls. It is initialised only once.

EXAMPLE void f(){ static int c = 0; c++; printf("%d ", c); } called 3 times prints 1 2 3.

5. Arrays and Strings

- **Array** = collection of same-type elements in contiguous memory. Index starts at 0.
- `int a[5];` takes 5 x 4 = 20 bytes (if int is 4 bytes).
- Array name gives the address of the first element: `a` `&a[0]`, and `a[i]` `*(a + i)`.

FORMULA 1-D: address of a[i] = Base + i x w. 2-D row-major (m x n): a[i][j] = Base + (i x n + j) x w. Column-major: Base + (j x m + i) x w.

EXAMPLE int a[3][4], base 1000, w = 4. Row-major a[2][1] = 1000 + (2x4 + 1) x 4 = 1000 + 36 = 1036.

Strings

- A string is a char array ending with the null character `'\0'`.
- `char s[] = "abc";` has size 4 (3 chars + `'\0'`). `strlen(s)` = 3, `sizeof(s)` = 4.

Function	Job
`strlen(s)`	Length without `'\0'`
`strcpy(d, s)`	Copy s into d
`strcat(d, s)`	Add s at end of d
`strcmp(a, b)`	0 if equal, negative if a < b, positive if a > b

6. Pointers

- **Pointer** = a variable that stores the address of another variable.
- `&` gives the address, `*` gives the value at an address (dereference).

```
int x = 10;
int *p = &x;    // p holds address of x
*p = 20;        // x becomes 20
int **q = &p;   // pointer to pointer, **q == 20
```

➤ Pointer arithmetic

- `p + 1` moves by `sizeof(*p)` bytes, not 1 byte. For `int *p`, p + 1 adds 4.
- Allowed: pointer + int, pointer − int, pointer − pointer (same array), comparison.
- Not allowed: pointer + pointer, multiply or divide pointers.

Term	Meaning
NULL pointer	Points to nothing (`NULL`, value 0)
Dangling pointer	Points to memory that is already freed
Wild pointer	Not initialised, holds garbage address
Void pointer	`void *`, generic, must be cast before dereference

➤ Reading declarations

- `int *p[5];` = array of 5 pointers to int.
- `int (*p)[5];` = pointer to an array of 5 ints.
- `int (*f)(int);` = pointer to a function taking int and returning int.

REMEMBER Read from the name outward: `()` and `[]` bind tighter than `*`.

➤ Dynamic memory (stdlib.h)

Function	Job
`malloc(n)`	Allocate n bytes, garbage values
`calloc(k, n)`	Allocate k blocks of n bytes, all set to 0
`realloc(p, n)`	Resize old block to n bytes
`free(p)`	Release memory

- Memory from these functions comes from the **heap**. Forgetting `free` causes a `memory leak`.

7. Structures, Unions and Enums

- **Structure** = groups different types under one name. Each member has its own memory.
- **Union** = all members share the `same memory`. Size = size of the largest member (plus padding).

```
struct S { char c; int i; };   // size often 8 (3 bytes padding)
union  U { char c; int i; };   // size 4
```

- Access members with `.` (variable) or `→` (pointer). `p→x` is the same as `(*p).x`.
- **Padding**: the compiler adds empty bytes so members sit on aligned addresses.
- **enum** gives names to integer constants: `enum color {RED, GREEN, BLUE};` gives 0, 1, 2.
- **typedef** gives a new name to a type: `typedef struct S Node;`.

Structure	Union
Separate memory for each member	Shared memory
Size >= sum of members	Size = largest member (+ padding)
All members usable together	Only one member valid at a time

8. File Handling and Preprocessor

- Use `FILE *fp = fopen("a.txt", "r");`. Always check if `fp == NULL`.
- Modes: `"r"` read, `"w"` write (creates / erases), `"a"` append, `"r+"` read and write, add `b` for binary.

- Functions: `fprintf`, `fscanf`, `fgets`, `fputs`, `fgetc`, `fputc`, `fread`, `fwrite`, `fseek`, `ftell`, `rewind`, `fclose`.

➤ Preprocessor

- `#define PI 3.14` = macro, text replaced before compiling.
- `#define SQ(x) ((x)*(x))` = macro with argument. Always use brackets.
- `#ifdef`, `#ifndef`, `#endif` = conditional compilation.

EXAMPLE `#define SQ(x) x*x` then `SQ(2+3)` becomes `2+3*2+3 = 11`, not 25. With brackets `((x)*(x))` it gives 25.

➤ Command-line arguments

- `int main(int argc, char *argv[])`: `argc` = number of arguments (includes program name), `argv[0]` = program name.

9. Most Asked Questions (practice)

1. Difference between call by value and call by reference with a swap example.
2. Explain all storage classes with scope, lifetime and default value.
3. Structure vs union. Find the size of a given structure with padding.
4. What is a pointer? Explain pointer arithmetic, NULL and dangling pointers.
5. malloc vs calloc. What is a memory leak?
6. Find the address of an element in a 2-D array (row-major and column-major).
7. Write a recursive program for factorial and Fibonacci.
8. Explain string functions and write your own `strlen` without library.
9. Difference between `while` and `do-while`. Explain `break` vs `continue`.
10. What are macros? Why can a macro give a wrong result without brackets?

TIP In GATE and university exams, most C questions are "find the output". Practise pointers, static variables, recursion and operator precedence by tracing code on paper.