Practice Problem

This list is based on the recommendation by both standard C practice problem lists and the early chapters of Kernighan & Ritchie's **"The C Programming Language"**

Section 1: Print & Read

This section covers the fundamentals of input/output in C, variable operations, data types, assignments, and basic arithmetic.

Basic Input/Output and Variable Handling

Write a program to :

0. Print “Hello World”.
1. Print an integer constant.
2. Read an integer from the user and print it.
3. Read two integers from the user and display both.
4. Read two integers, to perform :
    a. addition, subtraction, and multiplication **without using a third variable**
    b. addition, subtraction, and multiplication using a third variable
5. Swap two numbers using a third variable.
6. Swap two numbers without using a third variable.
7. Get and display the size of `int`, `float`, `double`, and `char` using `sizeof`.
8. Assign a character constant to a variable and print it.
9. Read a character from the user, assign to a variable, and print it.
10. Read the string "Hello World" using `scanf` and print it. (Explore why this may fail.)

Arithmetic and Expression Evaluation

1. Find the roots of a quadratic equation given coefficients a, b and c.
2. Write a program that evaluates and prints arithmetic expressions with mixed data types (verify integer division vs floating-point).

String Input/Output

1. Read a string (single word) from the user and print it.
2. Read a string with spaces (e.g., `"Hello World"`) from the user and print it (using `gets` or `fgets`, explore limitations).

Type Conversions and Literals

1. Assign integer, floating, and character literals to variables and print them using relevant format specifiers.
2. Demonstrate the effect of type casting between integer and float types.
3. Use escape sequences (`\n`, `\t`, etc.) in string and character outputs.

Edge I/O Scenarios

1. Check what happens if non-numeric data is entered when an integer or float is expected (input validation).
2. Print the value returned by `EOF` constant.

Symbolic Constants

1. Use `#define` and `const` to declare constants and utilize them in expressions and outputs.

Section 2: Conditional Statements

This section provides practice programs focusing on conditional statements, thoroughly covering `if`, `if...else`, `else if` ladders, `switch`, and the conditional (ternary) operator

2.1 Simple If, If...Else, and Else If

1. Read an integer and determine whether it is positive, negative, or zero.
2. Read two integers and check if they are equal, or determine which is larger.
3. Read three integers and find the largest among them.
4. Check if a number is odd or even.
5. Determine if a given year is a leap year.
6. Check if a character entered by the user is an alphabet or not.
7. Check if a character is a vowel or consonant.
8. Check if a character is uppercase or lowercase.
9. Check if the input character is the alphabet, digit, or special character.

2.2 Nested If...Else and Logical Operators

1. Read marks for five subjects and assign a grade based on percentage ranges (A/B/C/D/E/F).
2. Compute profit or loss given cost price and selling price; display the result.
3. Read the angles of a triangle and check whether the triangle is valid.
4. Check if the sides entered can form a valid triangle and categorize it (equilateral, isosceles, scalene).

5. Input week number (1–7) and print corresponding weekday name.
6. Input month number and display the number of days in the month (consider leap years for February).

2.3 Switch-Case Statements

1. Read a character (`+`, `-`, `*`, `/`) and two integers, and perform corresponding arithmetic operations.
2. Take a number (1–7) and print the day of the week using a switch.
3. Input a number (1–12) and display the name of the month via switch.
4. Read a grade character and print remarks based on grade using switch-case.
5. Basic calculator: Input operands and operator, perform operation using switch-case and print result.
6. Given a number, output "small" if less than 10, "large" if more than 10, "equal" if exactly 10 (switch).

2.4 Ladder and Multiple Conditions

1. Print message based on multiple ranges:
   a. If number <100 print “small”,
   b. 100–200 “large”,
   c. 201–300 “bigger”,
   d. 301–400 “largest”,
   e. >400 “very large”.
2. Complex logic with nested and sequential checks (e.g., months, seasons, ticket prices, etc.).

2.5 Conditional (Ternary) Operator

1. Input two numbers, find the maximum using the conditional operator.
2. Use a ternary operator to find if a number is even or odd.
3. Assign remarks ("Pass"/"Fail") using conditional operators based on marks.
4. Use nested conditional operators to classify a number as positive/negative/zero.

2.6 Exercises for Error Checking and Input Validation

1. Read an integer and check for invalid (non-numeric) input.
2. If input is not within expected range (e.g., day of week, month), print an error message.
3. Demonstrate use of default case in switch for invalid input.

Example Challenge Programs

1. Program to read two integers, compute their quotient and remainder if the second is nonzero (handle divide-by-zero).

2. Take an integer and check for special values (e.g., if 10 divide by 2, 20 divide by 3 ... as in sample switch/case).
3. Read a number and, based on its specific or range value, perform cascaded actions (nested if, else-if, switch)

Section 3: Loops and Iteration

This section develops your mastery of loops and iterative constructs

3.1 While Loops

1. Print all numbers from 1 to n (user input), using a while loop.
2. Sum all integers from 1 to n using a while loop.
3. Print the multiplication table for a given number using a while loop.
4. Reverse a number entered by the user with a while loop.
5. Count and display the number of digits in a number using a while loop.
6. Calculate the factorial of a number using a while loop.

3.2 For Loops

1. Print all even numbers between 1 and 100 using a for loop.
2. Print the first n terms of the Fibonacci series using a for loop.
3. Find and display all prime numbers between 1 and 100 using a for loop.
4. Calculate and print the sum of the series 1 + 1/2 + 1/3 + ... + 1/n using a for loop.
5. Display all uppercase ASCII characters with their integer values using a for loop.
6. Print a square/triangle/star pattern based on user input (height) using a for loop.

3.3 Do-While Loops

1. Read numbers until the user enters 0; compute and print their sum using a do-while loop.
2. Ask the user repeatedly for a password until correct (demonstrating do-while).
3. Print a menu-driven calculator: repeat operations until the user chooses to exit (do-while).
4. Display digits of a number, one per line, using a do-while loop.

3.4 Nested Loops

1. Print multiplication tables from 1 to 10 using nested for loops.
2. Print various formatted patterns (pyramid, diamond, Pascal’s triangle) using nested loops.
3. Generate and print the prime-factored form of numbers 1 to n.
4. Check and print all Armstrong numbers in a range using nested loops.

3.5 Use of Break and Continue

1. Search for a number in a sequence; stop searching when found using break.
2. Skip printing odd numbers in a loop using continue.
3. From a list of numbers entered by the user, print only non-negative values, use break if a negative is entered.
4. Find the smallest divisor of a number greater than 1 using break.

3.6 Loops with Complex Conditions

1. Calculate the average of positive numbers entered by the user.
2. Test whether a given integer is a palindrome.
3. Count the number of vowels and consonants in an input string using a loop.
4. Count frequency of each digit in a long integer using loops and arrays.

3.7 Infinite and Sentinel-Controlled Loops

1. Implementation of an infinite loop using for(;;).
2. Read inputs until a specific sentinel value is entered (e.g., -99 to stop).
3. Implement a simple command processor loop that exits on a keyword ("quit"/"exit").

3.8 Practical Applications and Mini-Projects

1. Implement a basic text-based menu interface—repeat options until the user selects quit
2. Write a program that draws a horizontal/vertical bar chart based on user input.
3. Number guessing game: repeatedly prompt the user until the correct value is guessed (give hints after each attempt).

Section 4: Functions and Program Structure

This section focuses on writing and using functions, understanding argument passing, scope, and program modularity, based on Chapter 4 of *The C Programming Language* and practical programming exercises.

4.1 Writing Basic Functions

1. Write a function `int power(int base, int n)` that raises `base` to the power `n` (for non-negative integers).
2. Create a program that uses this `power` function to print powers of 2, 3, and -3 for exponents 0 to 9.
3. Write a function to swap two integer values by passing pointers (demonstrate call-by-reference using pointers).
4. Implement a function that returns the minimum and maximum among three integers using multiple functions.

4.2 Functions with Arguments and Return Values

1. Write a function `int factorial(int n)` that computes the factorial of `n` recursively and test it.
2. Write a function to check whether a number is prime, returning 1 or 0, and print all primes up to a given limit.
3. Implement a recursive function to compute Fibonacci numbers.
4. Write a function `int strlen(char s[])` that returns the length of a string.
5. Write a function `void reverse(char s[])` to reverse a string in place.

4.3 Understanding Call by Value and Call by Reference

1. Write a program with a function that attempts to modify an integer parameter passed by value and observe no change.
2. Rewrite the above using pointers to modify the caller's variable, demonstrating call by reference via pointers.
3. Pass arrays to functions and modify elements within the function to demonstrate array behavior with arguments.

4.4 Using Static and External Variables, Scope

1. Write a function with a static variable that counts the number of times the function is called.
2. Write a program demonstrating local vs global variable scope and external variables accessed via `extern`.
3. Rewrite the longest line program to use external variables for line buffers and length counters.
4. Write a function that maintains a running total using a static variable.

4.5 Header Files and Modularization

1. Split a program into multiple files: one containing function definitions, one with `main()`, and a header file declaring function prototypes.
2. Write and use a header file with symbolic constants (`#define`) and function prototypes.
3. Create a mini-library of string functions: copy, concatenate, compare, and length.

4.6 Recursive Functions

1. Write the `power` function recursively.
2. Write a recursive function to calculate the greatest common divisor (GCD) of two numbers.
3. Implement recursive algorithms for summations or factorials with explanation.

4.7 The C Preprocessor and Macros

1. Define simple macros for constants, such as `#define PI 3.14159`.
2. Define parameterized macros for common operations like
    a. `#define MAX(a,b) ((a) > (b) ? (a) : (b))`.
3. Write a program to demonstrate pitfalls of macros (e.g., side effects in macro arguments).
4. Use conditional compilation directives: `#ifdef`, `#ifndef`, `#else`, `#endif`.

4.8 Practice Exercises

1. Rewrite the Fahrenheit-Celsius table program to use a conversion function.
2. Write a function `int getline(char s[], int lim)` that reads a line from input into array `s`.
3. Write a function `void copy(char to[], char from[])` to copy strings, and use these in a program to print the longest line of input.
4. Write a function `void squeeze(char s[], int c)` to remove all occurrences of `c` from `s`.
5. Implement your own versions of standard string functions: `strcpy`, `strcat`, `strcmp`.

Section 5: Pointers and Arrays

This section covers pointers and arrays, their interplay.

5.1 Pointers and Addresses

1. Declare and initialize pointers to variables of different types.
2. Print the address of an integer variable using `%p` format specifier.
3. Use pointer variables to read and modify values indirectly.
4. Write a program to swap two numbers using pointers.
5. Perform pointer arithmetic: increment/decrement pointers pointing to array elements.
6. Access array elements via pointers and compare with array subscripting.

5.2 Pointers and Function Arguments

1. Write a function that swaps two integers using pointer arguments.
2. Write functions that modify an array’s elements via pointers.
3. Implement a function to calculate the length of a string using pointer arithmetic.
4. Write a function to reverse a string, passing a pointer to it.
5. Demonstrate the difference between passing an array and passing a pointer to a function.

5.3 Pointers and Arrays

1. Declare arrays and pointers to arrays, and manipulate elements via pointers.
2. Explore how arrays decay to pointers when passed to functions.
3. Create programs to copy one array to another using pointers.
4. Perform string manipulation by moving through character arrays with pointers.
5. Implement a function that returns a pointer to a static array.

5.4 Address Arithmetic

1. Perform arithmetic on pointers to traverse arrays.
2. Calculate the difference between two pointers to elements of the same array.
3. Understand pointer comparison and pointer increment ordering.
4. Write a program to find the largest/smallest element in an array using pointers.

5.5 Character Pointers and String Functions

1. Use character pointers to iterate through strings.
2. Write your own versions of `strlen`, `strcpy`, and `strcmp` using character pointers.
3. Demonstrate pointer-based string concatenation.
4. Implement a function to find the first occurrence of a character in a string using pointers.

5.6 Pointer Arrays and Pointers to Pointers

1. Declare and initialize arrays of pointers to strings.
2. Write a program to sort an array of string pointers alphabetically.
3. Use pointers to pointers to access a two-dimensional array.
4. Understand the difference between array of pointers and pointer to an array.

5.7 Multi-dimensional Arrays

1. Declare, initialize, and access elements of 2D arrays.
2. Use pointers to traverse multi-dimensional arrays.
3. Write functions to add and multiply matrices using pointers and/or arrays.
4. Understand row-major storage and how multi-dimensional arrays are stored in memory.

5.8 Initialization of Pointer Arrays

1. Initialize an array of pointers with strings.
2. Traverse and print strings using pointer arrays.
3. Modify string content through pointer arrays.

5.9 Pointers vs Multi-dimensional Arrays

1. Program to demonstrate differences in declaration and use between pointers and multidimensional arrays.
2. Use pointers to simulate multi-dimensional array access.

5.10 Command-line Arguments

1. Write a program that prints every command-line argument on a separate line.
2. Sum integer arguments passed via command line.
3. Modify an environment variable using command line arguments.

5.11 Pointers to Functions

1. Write a function pointer declaration and assign it to different functions.
2. Implement simple callback functions using function pointers.
3. Use function pointers to select a sorting or calculation algorithm dynamically.

5.12 Complicated Declarations

1. Parse and interpret complicated pointer and array declarations.
2. Write programs that declare and manipulate pointers to arrays, arrays of pointers, pointers to functions, etc.

Section 6: Structures

This section covers structures in C, their definition, usage, and related topics such as arrays of structures, pointers to structures, and advanced features like unions and bit-fields.

6.1 Basics of Structures

1. Define a simple struct `point { int x, y; }` and write a program to create and print points.
2. Read values into a structure and print its members.
3. Write a function to swap two structures of the same type.
4. Create a structure representing a complex number with real and imaginary parts, and write functions to add and multiply two complex numbers.

6.2 Structures and Functions

1. Write functions that take structures as arguments by value and by pointer; observe effects and efficiency.
2. Write a function that modifies a structure passed by pointer.

3. Implement a function to print the contents of a structure passed by pointer.

6.3 Arrays of Structures

1. Define an array of structures representing a list of students (name, roll number, marks).
2. Write a program to input and print the array.
3. Sort an array of structures by one field (e.g., marks).
4. Search an array of structures for a given key (e.g., student by roll number).

6.4 Pointers to Structures

1. Declare pointers to structures, assign structure addresses, and access members via the pointer using `->`.
2. Demonstrate the difference between `(*ptr).member` and `ptr->member`.
3. Modify structure members through pointers.

6.5 Self-Referential Structures

1. Define a self-referential structure for linked list nodes with a data field and pointer to the next node.
2. Write a program to create a simple linked list, traverse it, and print its elements.
3. Add functions to insert and delete nodes in the linked list.

6.6 Table Lookup Using Structures

1. Implement a lookup table with structures, for example to map month numbers to names and days.
2. Use structures to implement simple symbol tables or configuration parameter lookups.

6.7 Typedef

1. Use `typedef` to simplify structure definitions for easier readability, e.g., `typedef struct point Point;`.
2. Redefine linked list nodes using typedef and rewrite related code.

6.8 Unions

1. Define and initialize unions with multiple members sharing the same memory.
2. Write a program that uses a union to store different types of data and demonstrates the effect of overwriting members.
3. Use unions for memory-efficient data representation.

6.9 Bit-fields

1. Define a structure with bit-fields to store boolean flags or small integer ranges efficiently.

2. Write a program to set, clear, and print values of bit-fields.
3. Compare memory sizes of normal structure vs. bit-field structure.

Section 7: Input and Output

This section focuses on input/output operations in C, covering standard input and output, formatted I/O, variable-length argument lists, file access, error handling, line input/output, and miscellaneous utility functions.

7.1 Standard Input and Output Basics

1. Write programs using `getchar()` and `putchar()` for character-by-character input and output.
2. Implement a file copy program that copies input to output.
3. Create a program to count characters, lines, and words in input using standard character I/O functions.
4. Demonstrate EOF handling with input streams.

7.2 Formatted Output with `printf`

1. Use `printf` to output integers, floats, characters, and strings with different format specifiers (`%d`, `%f`, `%c`, `%s`).
2. Practice field width and precision specifiers for alignment and formatting precision.
3. Print tables with aligned columns using formatted output.
4. Format floating-point numbers with fixed decimal places.
5. Write custom messages combining text and variable values using formatted output.
6. Explore escape sequences (`\n`, `\t`, `\\`) in output formatting.

7.3 Variable-Length Argument Lists

1. Write simplified versions of variadic functions such as a custom `minprintf` supporting `%d`, `%f`, and `%s`.
2. Implement functions accepting variable numbers of arguments using `stdarg.h`.
3. Practice using `va_start`, `va_arg`, and `va_end` macros.

7.4 Formatted Input with `scanf`

1. Read integers, floats, characters, and strings using `scanf`.
2. Handle whitespace and newlines in formatted input.
3. Use scansets (`%[ ]`) to read input until a set of characters is encountered.
4. Manage input buffer to prevent overflow and unwanted leftover characters.
5. Write input validation programs checking valid integer or float input.
6. Compare reading input using `scanf` vs `getchar`.

7.5 File Access

1. Open and close files using `fopen()` and `fclose()`.
2. Read and write characters using `fgetc()` and `fputc()`.
3. Read and write formatted data using `fprintf()` and `fscanf()`.
4. Write a program to copy one file to another.
5. Count characters, lines, and words in a file.
6. Append data to an existing file.
7. Use `fseek()` and `ftell()` to perform random access in files.
8. Read and write binary data with `fread()` and `fwrite()`.

7.6 Error Handling: `stderr` and `exit()`

1. Print error messages on `stderr`.
2. Write programs that detect errors in file operations (e.g., file not found, access denied).
3. Use the `exit()` function to terminate a program with success or failure status.
4. Handle incorrect user inputs with error messages and clean termination.
5. Explore `perror()` for system error messages.

7.7 Line Input and Output

1. Implement line-oriented input using `fgets()`.
2. Write functions to read a line safely into a buffer.
3. Implement line output with `fputs()` and `puts()`.
4. Read lines from files and process line-by-line.
5. Write programs to remove trailing blanks and tabs from lines.
6. Implement line folding (breaking long lines into shorter lines).

7.8 Miscellaneous Standard I/O Functions

1. Use string operations (`strlen`, `strcpy`, `strcat`, `strcmp`) from `<string.h>` in conjunction with I/O.
2. Apply character classification and conversion functions (`isdigit()`, `isalpha()`, `tolower()`, `toupper()`) from `<ctype.h>` to validate and process input.
3. Experiment with `ungetc()` to push characters back onto the input stream.
4. Utilize `system()` to execute shell commands from C programs.
5. Manage dynamic storage allocation when reading input of unknown length.

7.9 Practical Programs and Exercises

1. Build a small text editor that reads a file, allows line deletion or insertion, and writes back.
2. Implement a word count program supporting files and standard input, printing counts of lines, words, and characters.

3. Create a formatted report generator that reads data values and formats them with specific layouts.
4. Validate and parse user input robustly to prevent crashes or unexpected behavior.
5. Write a program to search and replace text patterns in files.

Section 8: The UNIX System Interface

This section covers low-level UNIX system calls and file operations that go beyond the standard library I/O covered previously.

8.1 File Descriptors and Basic Operations

1. Write a program to copy a file using UNIX system calls `open`, `read`, `write`, and `close`.
2. Demonstrate error checking after each system call; print appropriate error messages if system calls fail.
3. Use the symbolic constants `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, and `O_TRUNC` to open files with different modes.

8.2 Low-Level I/O: `read` and `write`

1. Read input from the standard input (`stdin`) using `read` system call byte-by-byte and write it to standard output (`stdout`) using `write`.
2. Implement file copying with buffering using `read` and `write` in blocks rather than single bytes.
3. Compare performance and behavior of `read`/`write` versus standard library `fread`/`fwrite`.

8.3 Opening, Creating, Closing, and Removing Files

1. Use `open` with flags to create new files or open existing ones.
2. Close file descriptors with `close`.
3. Delete a file using an unlink system call.
4. Write programs that open a file, write data, close it, and then reopen and read the data back.

8.4 Random Access with `lseek`

1. Use `lseek` to move the file pointer within an open file for reading/writing at arbitrary offsets.
2. Write a program to read the middle portion of a file without reading the whole file.
3. Implement a program to append data at the end of a file using `lseek` and `write`.

8.5 Example: Implementing `fopen` and `getc`

1. Write your own versions of the standard `fopen` and `getc` functions using low-level UNIX system calls.
2. Manage buffering and file descriptor state to mimic standard I/O behavior.
3. Handle errors properly to illustrate the difference between system calls and buffered library calls.

8.6 Listing Directories

1. Write a program to list all files in a directory using UNIX system calls (`opendir`, `readdir`, `closedir`).
2. Retrieve and print file information such as name, size, and modification time.
3. Practice error handling for directory operations.

8.7 Example: A Storage Allocator

1. Implement a simple dynamic memory allocator (like a mini `malloc` and `free`).
2. Manage a free list and allocate blocks of memory on demand.
3. Practice pointer arithmetic and structure management.

Practice Exercises

1. Write a program to compare two files byte-by-byte and report the first difference.
2. Write a program to copy a file but replace all lowercase letters with uppercase.
3. Implement a command-line tool that reads from multiple files and concatenates their contents (similar to UNIX `cat`).
4. Write a log program that appends timestamped messages to a log file using system calls.

Section 9: Dynamic Memory Management, Advanced Data Structures, and Miscellaneous Topics

This section brings together late-chapter topics from the book ("The C Programming Language" Chapters 5, 6, 8, Appendix B) and the advanced, integrative problems from standard practice lists. The focus is on dynamic memory allocation, abstract data structures (arrays, linked lists, stacks, queues, trees), use of macros and typedef, bitwise and miscellaneous operators, and comprehensive projects.

9.1 Dynamic Memory Management

1. Allocate and free memory for a single integer using `malloc` and `free`.
2. Dynamically create a 1D integer array of user-defined size, read and process elements, then safely release memory.

3. Dynamically allocate a 2D matrix (MxN) using pointers to pointers, perform matrix operations (addition, multiplication), and free memory.
4. Implement a function to resize a dynamically allocated array using `realloc`.

9.2 Abstract Data Structures with Arrays and Memory

1. Create and manage a dynamic array with automatic resizing on insertion.
2. Write a program for stack operations (push, pop) using arrays and dynamic memory allocation.
3. Implement a circular queue using a dynamic array.
4. Simulate basic string operations (`strcpy`, `strcat`, etc.) using dynamically allocated character arrays.

9.3 Linked Lists

1. Create a singly linked list: insert, delete, traverse, and print elements.
2. Create a doubly linked list and perform standard operations (insertion at head/tail, deletion, traversal).
3. Reverse a singly linked list in-place.
4. Implement a sorted (ordered) singly linked list.
5. Remove duplicates from a linked list.
6. Copy a linked list.

9.4 Stacks and Queues

1. Implement stack operations (push, pop, peek, is-empty) using linked lists.
2. Implement queue operations (enqueue, dequeue, is-empty) using linked lists.
3. Convert infix expressions to postfix and evaluate them using stack data structures.
4. Simulate browser back/forward functionality with two stacks.

9.5 Trees (Binary Trees, Traversals)

1. Create a binary search tree (BST): insert, search, delete nodes.
2. Write functions for pre-order, in-order, and post-order tree traversal (recursive).
3. Calculate the height/depth of a binary tree.
4. Find the minimum and maximum element in a BST.
5. Count leaf and non-leaf nodes in a tree.

9.6 Graphs (Basic Operations)

1. Represent a graph via adjacency matrix and adjacency list.
2. Implement Depth-First Search (DFS) and Breadth-First Search (BFS) traversals on a graph.
3. Detects cycles in an undirected graph.
4. Find the degree of a given vertex.

9.7 Macros, Typedef, and Miscellaneous Operators

1. Create macros for common mathematical operations such as MAX and MIN; demonstrate pitfalls.
2. Use `typedef` to simplify pointer and structure declarations.
3. Demonstrate and document the use of `sizeof`, comma operator, bitwise (*, |, ^, ~, <<, >>), and ternary (`? :`) operators in small test programs.
4. Write a program to show the difference between macros and inline functions.

9.8 Projects and Integrative Challenges

1. Mini database using arrays of structures; perform CRUD operations.
2. File-based stack or queue that preserves state between runs.
3. Text file analysis (word, character, line count; most frequent word/character).
4. Develop a custom dynamic string library enabling resizing, appending, and string manipulation.
5. Implement a simple shell-like interface: accept, parse, and execute user commands.