

QUESTION BANK

Programming for Problem Solving (ICT-101T)

UNIT I – INTRODUCTION TO PROGRAMMING AND FUNDAMENTALS OF C

A. Direct / Basic Questions

1. Define an algorithm. List the essential characteristics of a good algorithm.
2. Differentiate source code, object code and executable code.
3. Explain the roles of preprocessor, compiler and linker in creating a C program.
4. Define flowchart and pseudocode. State two advantages of each.
5. Explain C tokens with suitable examples.
6. Differentiate keywords, identifiers, constants and variables in C.
7. List the basic C data types and explain their typical use.
8. Differentiate declaration, initialization and assignment with examples.
9. Explain arithmetic, relational, logical and assignment operators.
10. Explain operator precedence and associativity with an example.
11. Explain implicit and explicit type conversion in C.
12. Write the basic structure of a C program and identify its major components.

B. Moderate / Application Questions

1. Write a C program to determine whether an integer is positive, negative or zero.
2. Write a C program to determine whether a given year is a leap year.
3. Write a program to find the largest of three numbers using decision making.
4. Write a C program to calculate an electricity bill using slab-wise charges.
5. Write a program to print the first N natural numbers and their sum using a loop.
6. Write a C program to check whether a number is prime.
7. Write a program to find the factorial of a number using iteration.
8. Write a C program to reverse an integer and check whether it is a palindrome.
9. Write a program to print a multiplication table using a loop.
10. Develop a menu-driven calculator using switch-case.

C. Difficult / Advanced Programming & Logic

1. Design an algorithm, flowchart and C program to generate all Armstrong numbers within a given range.
2. Design an algorithm and program to classify a triangle as equilateral, isosceles or scalene after validating the input.
3. Develop a program to find the second-largest among N numbers without sorting the array.
4. Design a program to calculate a student's grade with input validation and clearly defined grade boundaries.
5. Write a program to determine whether the sum of digits of a number is divisible by a specified value.
6. Develop a program using nested decision statements to solve a multi-condition eligibility problem.

D. Error Finding / Debugging

1. Find and correct the errors in a C program containing a missing semicolon and an incorrect return statement.
2. Debug a leap-year program whose condition uses incorrect logical precedence.
3. Identify and correct an infinite loop caused by an incorrect loop-control statement.
4. Debug a percentage calculation that produces an incorrect result because of integer division.
5. Find the logical errors in a switch-case calculator, including missing break statements and incorrect input handling.

E. Critical Thinking / Competency / HOTS

1. Compare nested if-else and switch-case for a menu-driven application. Justify which is more maintainable.
2. Explain how selecting an inappropriate data type can cause overflow or loss of precision in a C program.
3. Design an efficient approach for checking primality and justify why it is better than testing every integer up to $n-1$.
4. Explain when fixed-width integer types are preferable to ordinary integer types in portable C programs.
5. Given a real-world problem, identify the inputs, outputs, constraints, abstraction level and most suitable algorithm before writing code.

UNIT II – ARRAYS, FUNCTIONS, SEARCHING, SORTING AND STRINGS

A. Direct / Basic Questions

1. Define an array. Explain declaration, initialization and indexing of a one-dimensional array.
2. Explain two-dimensional and multidimensional arrays with examples.
3. Differentiate user-defined functions and built-in/library functions.
4. Explain function declaration, definition and function call.
5. Differentiate actual and formal parameters.
6. Explain call by value and the idea of call by reference in C.
7. Explain storage classes, scope and lifetime of variables.
8. Define a string in C and explain its relationship with a character array.
9. List common character and string handling functions and state their purposes.
10. Explain recursion and identify the essential components of a recursive function.

B. Moderate / Application Programming

1. Write a C program to find the minimum and maximum elements of an array.
2. Write a program to calculate the sum and average of array elements.
3. Write a program to add two matrices.
4. Write a program to find the transpose of a matrix.
5. Write a function that receives an array and returns its average.
6. Write recursive functions to calculate factorial and Fibonacci numbers.
7. Implement linear search for an element in an array.
8. Implement binary search on a sorted array.
9. Write separate programs for bubble sort, insertion sort and selection sort.
10. Write a program to reverse the elements of an array.
11. Write a program to reverse a string without using a library reverse function.
12. Write a program to check whether a string is a palindrome.

13. Write a program to count vowels, consonants, digits and spaces in a string.
14. Write a program to determine the frequency of each character in a string.
15. Write a function-based program to remove duplicate elements from an integer array.

C. Difficult / Flowchart / Logical Questions

1. Draw a flowchart and write a program to find the second-largest element in an array.
2. Develop a flowchart and C program for binary search, including the unsuccessful-search condition.
3. Design an iterative square-root approximation algorithm and represent it using a flowchart.
4. Design a flowchart and program to process marks of students stored in a two-dimensional array and generate subject-wise averages.
5. Develop an algorithm, flowchart and program to remove duplicate values from an array while preserving the first occurrence.

D. Error Finding / Debugging

1. Debug an array program containing an off-by-one error in the loop boundary.
2. Find and correct the incorrect midpoint calculation or boundary update in a binary-search implementation.
3. Identify the missing base case in a recursive function and explain the resulting problem.
4. Debug a function intended to swap two variables but whose parameter passing prevents the original values from changing.
5. Find the error in a string comparison using == and replace it with an appropriate approach.
6. Debug a sorting program that fails when duplicate elements are present.

E. Critical Thinking / Competency / HOTS

1. Compare linear and binary search in terms of prerequisites, complexity and practical use.
2. Compare iterative and recursive implementations of Fibonacci. Recommend one for large inputs and justify your answer.
3. Explain how row-major storage affects the performance of nested loops over a two-dimensional array.
4. Which sorting technique is more suitable for an almost-sorted array? Justify your choice.
5. Design a modular program using functions for input, processing, searching and reporting. Explain how modularity improves testing.
6. Design a robust string-palindrome solution that handles spaces, punctuation and case differences.

UNIT III – POINTERS, STRUCTURES, UNIONS, DYNAMIC MEMORY AND FILE HANDLING

A. Direct / Basic Questions

1. Define a pointer and explain the address-of (&) and dereference (*) operators.
2. Explain pointer arithmetic and its relationship with array elements.
3. Explain pointer to pointer with a suitable example.
4. What is a void pointer? State its use and limitation.
5. Differentiate an array of pointers from a pointer to an array.
6. Explain the relationship between pointers and strings.
7. Explain function pointers and one practical use.
8. Why is returning the address of a local variable unsafe?

9. Define a structure and explain declaration, initialization and member access.
10. Explain nested structures and arrays of structures.
11. Explain self-referential structures and their use in linked data structures.
12. Differentiate structures and unions.
13. Explain typedef, enumerations and bit fields.
14. Differentiate malloc(), calloc(), realloc() and free().
15. Explain memory leaks, dangling pointers and NULL-pointer safety.
16. Explain text and binary file modes and basic file operations.
17. Explain command-line arguments argc and argv and the role of errno in error handling.

B. Moderate / Application Programming

1. Write a C program to swap two variables using pointers.
2. Write a function that finds the maximum element of an array using pointers.
3. Demonstrate pointer arithmetic for traversing an array.
4. Write a program using a pointer to pointer to modify a dynamically allocated integer.
5. Create a structure for employee records and write functions to display and search records.
6. Design a nested structure for student information containing address details.
7. Define a self-referential structure for a singly linked-list node and create a small list.
8. Use dynamic memory allocation to store N numbers and calculate their average.
9. Write a program that increases a dynamically allocated array using realloc().
10. Write a program to count characters, words and lines in a text file.
11. Write a program to copy one file into another with appropriate error handling.
12. Write a program to append records to an existing file.
13. Develop a command-line program that accepts a filename and reports whether it can be opened.

C. Difficult / Case Study / Flowchart / Logic

1. Design an employee database using structures and dynamic memory. Provide the algorithm, flowchart and C implementation for adding and searching records.
2. Design a student-result file system that stores records and calculates results while handling file-open and read/write errors.
3. Develop a flowchart for dynamic allocation, validation, processing and deallocation of an array.
4. Represent polynomials using suitable C structures/arrays and design algorithms for polynomial addition and multiplication.
5. Design the flow of a command-line utility that accepts options and filenames, validates arguments, opens files safely and reports errors.

D. Error Finding / Debugging

1. Debug a program that dereferences an uninitialized pointer.
2. Identify why a program containing free(p); free(p); can fail and correct it.
3. Find the error in a function that returns the address of a local variable.
4. Debug code that overwrites the original pointer with realloc() without safely handling allocation failure.
5. Find and correct the use of an uninitialized pointer to a structure.
6. Debug a file program that uses the FILE pointer without checking whether fopen() returned NULL.
7. Identify why binary data becomes corrupted when processed using inappropriate text-mode operations.

E. Critical Thinking / Competency / HOTS

1. Compare an array of structures with a structure containing multiple arrays for an employee database. Justify the better representation for different access patterns.
2. Explain when a union is preferable to a structure and identify the risks of reading a member other than the one most recently stored.
3. Design a memory-safe dynamically growing collection and explain how it handles allocation failure and deallocation.
4. Design a file-processing solution for a very large file that avoids loading the complete file into memory.
5. Explain how pointer arithmetic, alignment and memory layout affect low-level C programming.
6. Design a modular command-line application using function pointers and separate header/source files. Explain how header hygiene and forward declarations improve maintainability.

UNIT IV – C STANDARD LIBRARIES, ALGORITHMS AND DATA STRUCTURES

A. Direct / Basic Questions

1. Define an Abstract Data Type (ADT) and explain its importance.
2. State the purposes of `stdio.h`, `stdlib.h`, `math.h`, `time.h`, `ctype.h` and `string.h`.
3. Define a linked list node and explain the concept of a singly linked list.
4. Differentiate singly, doubly and circular linked lists.
5. Define a stack and explain LIFO with applications.
6. Define a queue and distinguish linear, circular and priority queues.
7. Define a binary tree and explain preorder, inorder and postorder traversal.
8. Define a binary search tree and state its ordering property.
9. Explain the use of function pointers in implementing flexible operations.
10. Explain `_Static_assert`, `_Generic` and assertions/diagnostics in modern C.
11. Explain program segmentation, forward declarations and header-file hygiene.
12. Explain why unsafe function-like macros can produce unexpected results and how to design safer macros.

B. Moderate / Application Programming

1. Write programs to calculate factorial and generate Fibonacci series.
2. Implement linear search and binary search.
3. Implement bubble sort, insertion sort and selection sort.
4. Create and traverse a singly linked list.
5. Insert a node at the beginning, end and a specified position of a singly linked list.
6. Delete a node from a singly linked list and handle an empty list.
7. Implement a doubly linked list with forward and backward traversal.
8. Implement a circular linked list and demonstrate traversal.
9. Implement a stack using an array.
10. Implement a stack using a linked list.
11. Convert an infix expression to postfix using a stack.
12. Check balanced parentheses using a stack.
13. Implement a linear queue using an array.
14. Implement a circular queue using an array.

15. Implement a priority queue using a suitable representation.
16. Implement a binary tree and perform preorder, inorder and postorder traversals.
17. Create a BST and implement insertion.
18. Implement searching in a BST.
19. Implement deletion of a node from a BST.
20. Design a menu-driven data-structure program using separate functions for each operation.

C. Difficult / Case Study / Flowchart / Logic

1. Design a browser-history system using a stack. Provide the data representation, operations and flowchart.
2. Design a hospital emergency-management queue and justify the choice of a priority queue.
3. Design a library-record system using a linked list. Include insertion, deletion, searching and traversal.
4. Design a student-search application using a BST and explain how ordering affects search performance.
5. Draw a flowchart for infix-to-postfix conversion and identify the role of operator precedence.
6. Compare linear, circular and priority queues for a real-world service system and recommend the most suitable one.

D. Error Finding / Debugging

1. Debug a linked-list program that loses the head pointer after insertion.
2. Find and correct a linked-list deletion bug caused by freeing a node before saving its next pointer.
3. Debug a stack implementation that allows overflow because of an incorrect boundary condition.
4. Find the front/rear update errors in a circular-queue implementation.
5. Debug a BST insertion/search program that follows the wrong subtree for a comparison.
6. Identify why an infix-to-postfix implementation mishandles parentheses or operator precedence.
7. Find the problem in an unsafe SQUARE(x) macro when it is called with an expression such as a+1. Rewrite it safely.

E. Critical Thinking / Competency / HOTS

1. Compare arrays and linked lists for insertion, deletion, random access and memory usage. Recommend one for a frequently changing dataset.
2. Compare singly, doubly and circular linked lists for navigation, memory overhead and application suitability.
3. Compare array-based and linked-list-based stacks and justify the better choice for an application with unpredictable size.
4. Explain why a circular queue avoids the wasted-space problem of a simple linear queue.
5. Analyze the performance of an unbalanced BST and explain how insertion order can affect its efficiency.
6. Design an ADT for hospital emergency management and specify its operations, data representation and error conditions.
7. Design a modular multi-file C project for a data-structure application, including headers, source files, forward declarations and interface separation.
8. Explain how `_Generic` can support type-generic interfaces while retaining compile-time type selection.
9. Explain the difference between `assert()`, `_Static_assert` and runtime error handling, and identify an appropriate use case for each.

End of Question Bank