C Practical Viva Questions & Answers

Here are some common Viva questions and answers for a C practical exam, categorized for better understanding.

I. Basic C Concepts & Syntax

Q1. What is C? Why is it called a procedural language?

Answer: C is a powerful and versatile general-purpose programming language. It's called a procedural language because it emphasizes breaking down a program into smaller, manageable procedures or functions. These functions are executed sequentially to solve a problem.

Q2. What are the basic data types in C?

Answer: The basic data types in C are:
* int: Integer (whole numbers)
* float: Floating-point (single-precision decimal numbers)
* double: Double-precision floating-point (more precise decimal numbers)
* char: Character (single characters)
* void: Represents the absence of a data type.

Q3. Explain the difference between int, float, and char data types.

Answer:
* int: Stores whole numbers without decimal points (e.g., 10, -5, 0).
* float: Stores single-precision floating-point numbers (decimal numbers with limited precision, e.g., 3.14, -2.5).
* char: Stores single characters enclosed in single quotes (e.g., 'a', '?', '$'). It actually stores the ASCII value of the character as an integer.

Q4. What is a variable? How do you declare and initialize a variable in C?

Answer: A variable is a named storage location in memory used to hold data.
* **Declaration:** Specifies the data type and name of the variable. Syntax: data_type variable_name; (e.g., int age;)
* **Initialization:** Assigning an initial value to a variable during declaration or later. Syntax: variable_name = value; (e.g., int age = 25; or age = 30;)

Q5. What are operators in C? Give examples of arithmetic, relational, and logical operators.

Answer: Operators are symbols that perform operations on operands (variables or values).
* **Arithmetic Operators:** + (addition), - (subtraction), * (multiplication), / (division), % (modulo - remainder)
* **Relational Operators:** == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to)
* **Logical Operators:** && (logical AND), || (logical OR), ! (logical NOT)

Q6. What is the difference between = and == operators?

Answer:
* = (Assignment operator): Assigns the value on the right-hand side to the variable on the left-hand side. (e.g., x = 5; assigns 5 to variable x)
* == (Equality operator): Compares if the values on both sides are equal. It returns 1 (true) if they are equal, and 0 (false) otherwise. (e.g., if (x == 5) checks if x is equal to 5)

Q7. What is printf() and scanf()? What are format specifiers?

Answer:
* printf(): A standard library function used to display output on the console.
* scanf(): A standard library function used to read input from the user.
* **Format Specifiers:** Used within printf() and scanf() to specify the data type of the variable being printed or read. Examples:
* %d: Integer
* %f: Float
* %lf: Double
* %c: Character
* %s: String

Q8. Explain the structure of a basic C program.

Answer: A basic C program generally has the following structure:
```c
#include <stdio.h> // Header file inclusion (for standard input/output functions)

    int main() { // Main function - execution starts here
   // Statements (code to be executed)
   printf("Hello, World!\n"); // Example statement
   return 0; // Indicates successful program execution
}
```

Q9. What is a header file? Why do we use #include?

Answer: A header file is a file containing function declarations, macro definitions, and other preprocessor directives. We use #include to include header files in our C program. This makes the functions and definitions declared in the header file available for use in our program. stdio.h is a common header file for standard input/output functions like printf() and scanf().

Q10. What is the main() function? Why is it important?

Answer: The main() function is the entry point of every C program. Program execution begins from the main() function. It is essential because without it, the compiler doesn't know where to start executing the code.

II. Control Flow (Conditional Statements & Loops)

Q11. What are conditional statements in C? Explain if, else if, and else statements.

Answer: Conditional statements allow you to execute different blocks of code based on whether a condition is true or false.
* if: Executes a block of code if the condition is true.
* else if: Used after an if statement to check another condition if the previous if condition was false. You can have multiple else if blocks.
* else: Executes a block of code if none of the preceding if or else if conditions are true.

Q12. Explain the switch statement. When is it useful?

Answer: The switch statement is a multi-way branching statement. It allows you to select one block of code to execute from multiple choices based on the value of an expression. It's useful when you need to compare a variable against multiple constant values.

Q13. What are loops in C? Explain for, while, and do-while loops.

Answer: Loops are used to repeatedly execute a block of code until a certain condition is met.
* for loop: Used when you know the number of iterations in advance. It has three parts: initialization, condition, and increment/decrement.
* while loop: Executes a block of code as long as the condition is true. The condition is checked

before each iteration.
* do-while loop: Similar to while, but it executes the block of code at least once before checking the condition. The condition is checked after each iteration.

Q14. What is the difference between while and do-while loops?

Answer: The main difference is that in a while loop, the condition is checked *before* the loop body is executed, so the loop body might not execute at all if the condition is initially false. In a do-while loop, the loop body is executed *at least once* before the condition is checked.

Q15. What are break and continue statements? How are they used in loops?

Answer:
* break: Terminates the loop execution immediately and jumps to the statement after the loop.
* continue: Skips the current iteration of the loop and jumps to the beginning of the next iteration (condition check).

III. Functions

Q16. What is a function in C? Why are functions important?

Answer: A function is a block of code that performs a specific task. Functions are important for:
* **Modularity:** Breaking down a large program into smaller, manageable units.
* **Reusability:** Writing code once and using it multiple times in different parts of the program.
* **Readability:** Making code easier to understand and maintain.
* **Abstraction:** Hiding complex implementation details and providing a simple interface.

Q17. Explain function declaration, function definition, and function call.

Answer:
* **Function Declaration (Prototype):** Declares the function's name, return type, and parameters *before* it's used in the program. Syntax: return_type function_name(parameter_list); (e.g., int add(int a, int b);)
* **Function Definition:** Contains the actual code (body) of the function that performs the task. Syntax:
c return_type function_name(parameter_list) { // Function body (statements) return return_value; // Optional return statement }
 * **Function Call:** Invokes or executes the function from another part of the program. Syntax: function_name(argument_list); (e.g., int sum = add(5, 3);)

Q18. What are arguments and parameters in functions?

Answer:
* **Parameters (Formal Parameters):** Variables listed in the function declaration and definition. They act as placeholders for the values that will be passed to the function.
* **Arguments (Actual Parameters):** The actual values passed to the function when it is called. Arguments are assigned to the corresponding parameters.

Q19. What is the return type of a function? Can a function return multiple values?

Answer: The return type specifies the data type of the value that a function sends back to the calling function. A function can have any valid data type as its return type, including void if it doesn't return any value. In standard C, a function can directly return only **one** value. To return multiple values, you can use pointers, structures, or arrays.

Q20. What is recursion? Give an example.

Answer: Recursion is a programming technique where a function calls itself directly or indirectly. It's useful for solving problems that can be broken down into smaller, self-similar subproblems.

Example (Factorial using recursion):

```
 int factorial(int n) {
    if (n == 0) {
        return 1; // Base case: factorial of 0 is 1
    } else {
        return n * factorial(n - 1); // Recursive call
    }
}
```

IV. Arrays

Q21. What is an array? How do you declare and initialize an array in C?

Answer: An array is a collection of elements of the same data type stored in contiguous memory locations.
* **Declaration:** data_type array_name[array_size]; (e.g., int numbers[5];)
* **Initialization:**
* During declaration: int numbers[5] = {1, 2, 3, 4, 5};
* Element-wise: numbers[0] = 10; numbers[1] = 20; ...

Q22. How do you access elements of an array? What is the index of the first element in an array?

Answer: Array elements are accessed using their index (position). The index starts from **0** for the first element, 1 for the second, and so on. Syntax: array_name[index]. (e.g., numbers[0] accesses the first element of the numbers array).

Q23. What is the difference between a 1D array and a 2D array?

Answer:
* **1D Array (One-dimensional array):** A linear array that stores elements in a single row or column. Think of it as a list of elements.
* **2D Array (Two-dimensional array):** An array of arrays. It stores elements in rows and columns, like a table or matrix.

Q24. How do you pass an array to a function in C?

Answer: When passing an array to a function, you actually pass the address of the first element of the array (a pointer). You don't need to specify the size of the array in the function declaration, but it's often good practice to pass the size as a separate parameter.

Example:

```
 void printArray(int arr[], int size) { // arr[] is treated as int *arr
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int myArray[] = {10, 20, 30, 40, 50};
    int arraySize = sizeof(myArray) / sizeof(myArray[0]);
    printArray(myArray, arraySize); // Pass array name and size
    return 0;
}
```

Q25. What are the advantages and disadvantages of using arrays?

Answer:
* **Advantages:**
* Efficient storage of multiple elements of the same type.
* Easy access to elements using indices.
* Useful for implementing data structures like lists, tables, etc.
* **Disadvantages:**
* Fixed size: Once declared, the size of an array cannot be easily changed (in standard C).
* Contiguous memory allocation: Memory must be available in a continuous block.
* Insertion and deletion of elements can be inefficient (especially in the middle of the array).

V. Pointers

Q26. What is a pointer? Why are pointers used in C?

Answer: A pointer is a variable that stores the memory address of another variable. Pointers are used in C for:
* **Direct Memory Access:** Manipulating data directly in memory.
* **Dynamic Memory Allocation:** Allocating memory during program execution.
* **Passing Arguments by Reference:** Modifying the original value of a variable passed to a function.
* **Implementing Data Structures:** Creating linked lists, trees, and other dynamic data structures.
* **Improving Performance:** In some cases, pointer operations can be faster than direct variable access.

Q27. How do you declare a pointer variable? What is the & (address-of) and * (dereference) operator?

Answer:
* **Pointer Declaration:** data_type *pointer_name; (e.g., int *ptr; - declares a pointer ptr that can store the address of an integer variable)
* & (Address-of operator): Returns the memory address of a variable. (e.g., ptr = &variable; assigns the address of variable to ptr)
* * (Dereference operator): Accesses the value stored at the memory address pointed to by a pointer. (e.g., *ptr gives the value stored at the address in ptr)

Q28. Explain pointer arithmetic. Give examples of pointer increment and decrement.

Answer: Pointer arithmetic involves performing arithmetic operations on pointer variables. When you increment or decrement a pointer, it moves to the next or previous memory location of the data type it points to.

Examples:

```
    int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr; // ptr points to the first element (arr[0])

printf("%d\n", *ptr);  // Output: 10
ptr++; // Increment ptr (moves to the next integer location)
printf("%d\n", *ptr);  // Output: 20
ptr--; // Decrement ptr (moves back to the previous integer location)
printf("%d\n", *ptr);  // Output: 10
```

Q29. What is a null pointer? Why is it important?

Answer: A null pointer is a pointer that does not point to any valid memory location. It's usually represented by NULL (defined in <stdio.h>, <stddef.h>, etc.). Null pointers are important for:
* **Initializing pointers:** To indicate that a pointer is not yet pointing to anything.
* **Error checking:** To check if a pointer is valid before dereferencing it.
* **Signaling the end of data structures:** In linked lists, the last node's next pointer is often set to NULL.

Q30. What is the difference between call by value and call by reference? How are pointers used in call by reference?

Answer:
* **Call by Value:** When you pass arguments by value, a copy of the argument's value is passed to the function. Changes made to the parameters inside the function do not affect the original arguments in the calling function.
* **Call by Reference:** When you pass arguments by reference (using pointers in C), you pass the memory address of the argument to the function. Changes made to the parameters inside the function *do* affect the original arguments in the calling function because you are directly working with the original memory locations.

Example (Call by Reference using pointers to swap two numbers):

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

int main() {
    int x = 10, y = 20;
    printf("Before swap: x = %d, y = %d\n", x, y);
    swap(&x, &y); // Pass addresses of x and y
    printf("After swap: x = %d, y = %d\n", x, y);
    return 0;
}
```

VI. Strings

Q31. What is a string in C? How are strings represented?

Answer: In C, a string is an array of characters terminated by a null character \0. Strings are represented as char arrays.

Q32. How do you declare and initialize a string in C?

Answer:
* **Declaration:** char string_name[size]; (e.g., char name[20];)
* **Initialization:**
* Using string literal: char name[] = "John"; (null character \0 is automatically added)
* Character by character: char name[5] = {'J', 'o', 'h', 'n', '\0'};
* Using strcpy() function (from <string.h>): char name[20]; strcpy(name, "Jane");

Q33. What are some common string manipulation functions in C? (from <string.h>)

Answer: Some common string functions include:
* strlen(str): Returns the length of the string str (excluding the null terminator).
* strcpy(dest, src): Copies the string src to the string dest.
* strcat(dest, src): Concatenates (appends) the string src to the end of the string dest.
* strcmp(str1, str2): Compares two strings str1 and str2. Returns 0 if they are equal, a negative value if str1 is lexicographically less than str2, and a positive value otherwise.
* strncpy(dest, src, n): Copies at most n characters from src to dest.
* strncat(dest, src, n): Appends at most n characters from src to dest.
* strncmp(str1, str2, n): Compares at most n characters of str1 and str2.

Q34. How do you read a string from the user using scanf() and gets()? What are the differences and potential issues?

Answer:
* scanf("%s", string_variable);: Reads a string from input until whitespace (space, tab, newline) is encountered. **Issue:** Vulnerable to buffer overflow if the input string is longer than the allocated size of string_variable.
* gets(string_variable);: Reads an entire line of input (including spaces) until a newline character is encountered. **Issue:** Highly vulnerable to buffer overflow because it doesn't check the input length. **Avoid using gets() as it is deprecated and unsafe.**
* **fgets(string_variable, size, stdin);:** A safer alternative. Reads at most size-1 characters from input (or until a newline or EOF is encountered) into string_variable. It includes the newline character in the string if read. It prevents buffer overflow.

Q35. How do you compare two strings in C? Can you use == operator to compare strings?

Answer: You should use the strcmp() function (from <string.h>) to compare strings in C. **You cannot directly use the == operator to compare strings.** The == operator compares the memory addresses of the string variables, not the content of the strings themselves. strcmp() compares the content of the strings character by character.

VII. Structures & Unions

Q36. What is a structure in C? How do you define and declare a structure?

Answer: A structure is a user-defined data type that can group together variables of different data types under a single name. It allows you to create complex data records.

Definition:

```
 struct structure_name {
   data_type member1;
   data_type member2;
   // ... more members
};
```

Declaration of structure variables:

```
 struct structure_name variable_name;
struct structure_name variable_name_array[array_size];
```

Example:

```
 struct Student {
   char name[50];
   int rollNo;
   float marks;
};

struct Student student1; // Declare a structure variable
struct Student class[30]; // Declare an array of structures
```

Q37. How do you access members of a structure?

Answer: Structure members are accessed using the dot (.) operator for structure variables and the arrow (->) operator for structure pointers.

Example:

```
 struct Student student1;
strcpy(student1.name, "Alice"); // Access name member using dot operator
student1.rollNo = 101;

struct Student *ptrStudent = &student1;
printf("Name: %s\n", ptrStudent->name); // Access name member using arrow operator
printf("Roll No: %d\n", ptrStudent->rollNo);
```

Q38. What is the difference between a structure and a union?

Answer:
* **Structure:** Allocates memory for each member separately. Members can be accessed independently and simultaneously. The total size of a structure is typically the sum of the sizes of its members (plus potential padding).
* **Union:** Allocates memory to hold only the largest member among its members. All members share the same memory location. Only one member can be actively used at a time. The size of a union is equal to the size of its largest member.

Q39. When would you use a structure and when would you use a union?

Answer:
* **Structure:** Use structures when you need to group together related data of different types and access them independently. For example, representing a student record, a point in 2D space, etc.
* **Union:** Use unions when you want to save memory by storing different types of data in the same memory location, but you will only use one type at a time. For example, representing a value that can be an integer, float, or character, but only one of these at any given moment. Unions are also used in system programming and low-level programming for memory management.

VIII. File Handling (Basic)

Q40. What is file handling in C? What are different file modes?

Answer: File handling in C refers to the process of reading data from and writing data to files stored on a storage device (like a hard drive). File modes specify how a file should be opened and accessed. Common file modes are:
* "r" (Read): Opens a file for reading. File must exist.
* "w" (Write): Opens a file for writing. Creates a new file if it doesn't exist, or overwrites an existing file.
* "a" (Append): Opens a file for appending. Creates a new file if it doesn't exist. Data is written at the end of the file.
* "rb", "wb", "ab": Binary modes for reading, writing, and appending binary files.
* "r+", "w+", "a+", "rb+", "wb+", "ab+": Modes for both reading and writing.

Q41. How do you open and close a file in C? What is fopen() and fclose()?

Answer:
* **fopen() (File Open):** Opens a file and returns a file pointer (of type FILE*) that can be used to access the file. If the file cannot be opened, it returns NULL.
Syntax: FILE *fopen(const char *filename, const char *mode);
* **fclose() (File Close):** Closes an opened file. It's important to close files after you are done with them to release system resources and ensure data is properly written to the file.
Syntax: int fclose(FILE *file_pointer); (Returns 0 on success, EOF on error)

Q42. How do you read data from a file in C? Explain fscanf() and fgets().

Answer:
* **fscanf() (Formatted File Input):** Reads formatted input from a file, similar to scanf() for console input.
Syntax: int fscanf(FILE *file_pointer, const char *format, ...);
* **fgets() (File Get String):** Reads a line of text from a file, similar to gets() but safer.
Syntax: char *fgets(char *str, int size, FILE *file_pointer); (Reads at most size-1 characters or until newline or EOF)

Q43. How do you write data to a file in C? Explain fprintf() and fputs().

Answer:
* **fprintf() (Formatted File Output):** Writes formatted output to a file, similar to printf() for console output.
Syntax: int fprintf(FILE *file_pointer, const char *format, ...);
* **fputs() (File Put String):** Writes a string to a file.
Syntax: int fputs(const char *str, FILE *file_pointer);

Q44. What is error handling in file operations? How do you check if fopen() was successful?

Answer: Error handling in file operations is crucial because file operations can fail (e.g., file not found, permission issues). You should always check if fopen() was successful by verifying if it returned a non-NULL file pointer. If it returns NULL, it indicates an error. You can use perror() (from <stdio.h>) to print a system error message related to the last error that occurred.

Example error checking after fopen():

```
    FILE *fp = fopen("myfile.txt", "r");
if (fp == NULL) {
   perror("Error opening file");
   // Handle the error (e.g., exit program, display message)
   return 1; // Indicate error to calling function
}
// ... proceed with file operations if fopen was successful ...
```

```
fclose(fp);
```

IX. Preprocessor Directives (Basic)

Q45. What are preprocessor directives in C? Give examples of #include and #define.

Answer: Preprocessor directives are instructions that are processed by the C preprocessor *before* the actual compilation of the code. They start with a # symbol.
* #include: Includes the contents of a header file into the current source file. (e.g., #include <stdio.h>)
* #define: Defines a macro. It can be used to define symbolic constants or simple function-like macros.
* Symbolic constant: #define PI 3.14159 (replaces PI with 3.14159 throughout the code before compilation)
* Function-like macro: #define SQUARE(x) ((x) * (x)) (replaces SQUARE(5) with ((5) * (5)) before compilation - be careful with parentheses to avoid unexpected behavior).

Q46. What is the difference between #define and const for defining constants?

Answer:
* #define: Preprocessor directive. Performs text substitution before compilation. No type checking is done. Constants defined with #define are not variables; they are just text replacements.
* const: Keyword that declares a constant variable with a specific data type. Type checking is performed by the compiler. const variables are actual variables stored in memory, but their values cannot be changed after initialization.

Recommendation: For defining named constants, const is generally preferred because it provides type safety and better debugging support compared to #define. Use #define for macros and conditional compilation.

X. Debugging & Common Errors

Q47. What are some common types of errors you might encounter in C programs?

Answer: Common errors include:
* **Syntax Errors:** Errors in the grammar of the C language (e.g., missing semicolon, incorrect spelling). Detected by the compiler.
* **Semantic Errors:** Errors in the meaning or logic of the code (e.g., using a variable before initialization, incorrect loop condition). Compiler may or may not detect these.
* **Runtime Errors:** Errors that occur during program execution (e.g., division by zero, accessing memory outside array bounds, segmentation fault).
* **Logical Errors:** Errors in the program's algorithm or logic, leading to incorrect output even if the program compiles and runs without crashing. These are the hardest to find and require careful debugging.
* **Linker Errors:** Errors that occur during the linking phase (e.g., undefined functions, missing libraries).

Q48. How do you debug a C program? What tools can you use?

Answer: Debugging involves finding and fixing errors in a program. Common debugging techniques and tools:
* **Print statements (printf()):** Insert printf() statements at strategic points in your code to print the values of variables and track the program's execution flow.
* **Debugger (e.g., gdb):** A powerful tool that allows you to:
* Step through code line by line.

* Set breakpoints to pause execution at specific locations.
* Inspect the values of variables at runtime.
* Examine the call stack.
* **Compiler Warnings:** Pay attention to compiler warnings. They often indicate potential issues in your code even if they are not syntax errors. Compile with flags like -Wall (all warnings) and -Werror (treat warnings as errors) during development.
* **Code Review:** Ask someone else to review your code. Fresh eyes can often spot errors that you might miss.
* **Testing:** Write test cases to systematically test different parts of your program and identify bugs.

Q49. What is a segmentation fault? What are common causes of segmentation faults in C?

Answer: A segmentation fault (often "segfault") is a runtime error that occurs when a program tries to access memory that it is not allowed to access. This usually happens when a program tries to:
* **Dereference a null pointer:** Trying to access memory through a pointer that is NULL.
* **Access memory outside array bounds:** Trying to read or write to an array index that is invalid (e.g., index < 0 or index >= array size).
* **Write to read-only memory:** Trying to modify memory that is protected as read-only.
* **Stack overflow:** Occurs when the program uses too much stack memory (often due to deep recursion without a proper base case).
* **Memory corruption:** Overwriting memory unintentionally, leading to unpredictable behavior and potential segfaults later.

Q50. What are some best practices for writing clean and maintainable C code?

Answer: Best practices include:
* **Meaningful variable and function names:** Use descriptive names that clearly indicate the purpose of variables and functions.
* **Comments:** Add comments to explain complex code sections, function functionality, and logic.
* **Indentation and formatting:** Use consistent indentation and formatting to make code readable and visually structured.
* **Modular code (functions):** Break down large programs into smaller, well-defined functions.
* **Error handling:** Implement proper error checking and handling, especially for file operations and user input.
* **Avoid global variables (when possible):** Minimize the use of global variables to reduce complexity and potential side effects.
* **Use const where appropriate:** Declare variables as const if their values should not be changed after initialization.
* **Code review:** Have your code reviewed by others to catch errors and improve code quality.
* **Test your code thoroughly:** Write test cases to ensure your code works correctly under various conditions.

This list covers a wide range of common Viva questions for C practicals. Be prepared to explain concepts clearly, give examples, and demonstrate your understanding of the practical aspects of C programming. Good luck!