25 Advanced C Function Challenges - Explained

1. What happens when a C function returns a local array? Why?

- Local arrays are stored in stack memory. When a function returns, its stack frame is destroyed. Returning a pointer to that local array leads to undefined behavior.
- Fix: Use static storage or dynamically allocated memory (malloc).

2. How would you pass a function as a parameter to another function in C?

- Use function pointers. Example:

```
    int add(int a, int b) { return a + b; }
    void operate(int x, int y, int (*func)(int, int)) {
        printf("%d", func(x, y));
    }
```

3. Difference between inline and static in function declarations:

- inline: Suggests the compiler to insert the function code at the call site.
- static: Limits the visibility of the function or variable to the current file.

4. Function declaration, definition, and prototype:

- Declaration: Introduces the function (e.g., int func();).
- Definition: Provides the body of the function.
- Prototype: Declaration with parameter types.

5. Effect of using static:

- Inside a function: variable retains value across calls.
- Outside a function: limits visibility to the file (internal linkage).

6. What is tail-call optimization and does C support it?

- Tail recursion: last action of a function is a recursive call.
- C supports TCO depending on compiler (e.g., GCC with -O2).

7. What happens if you call a function before its prototype is defined?

- In C89: assumes int return type.
- In C99+: compiler error due to implicit declaration.

8. Variadic function to return max of integers:

```
    int maxOf(int count, ...) {
        va_list args;
        va_start(args, count);
        int max = va_arg(args, int);
        for (int i = 1; i < count; i++) {
            int val = va_arg(args, int);
            if (val > max) max = val;
        }
        va_end(args);
        return max;
    }
```

9. Compiler can't enforce type safety in variadic functions. Arguments are read by va_arg without type info.

10. va_list, va_start, va_arg:

```
- va_list: holds the list.
- va_start: initializes it.
- va_arg: extracts the next argument.
```

11. Calling variadic function with fewer arguments:

```
- Causes undefined behavior: function reads more arguments than passed.
```

12. Passing float to variadic function:

```
- float is promoted to double. Always use va_arg(args, double).
```

13. Simulate anonymous functions:

```
- Use function pointers or GCC's lambda macro:
  #define LAMBDA(return_type, body) ({ return_type __fn__ body __fn__; })
```

14. Function pointer array for arithmetic:

```
    int (*ops[4])(int, int) = {add, sub, mul, divide};
    ops[0](a, b); // calls add
```

15. Function pointer from different files:

```
- Must not be static.
- Use extern in header file.
```

16. Function pointer to void returning func with (int, char*):

```
    void (*fp)(int, char*) = myFunc;
```

17. Dynamically switch behavior at runtime:

```
    void (*handler)();
    if (mode == 1) handler = printA;
    else handler = printB;
```

18. Maximum depth of recursion:

```
- Based on stack size. Use `ulimit -s` and frame size estimate.
```

25 Advanced C Function Challenges - Explained

19. Convert recursion to iteration:

```
- Use explicit stack data structure to mimic call stack.
```

20. Mutual recursion:

```
    void A(); void B();
    void A() { B(); }
    void B() { A(); }
```

21. Tail recursion:

```
    int tailFact(int n, int acc) {
        if (n == 0) return acc;
        return tailFact(n-1, n*acc);
    }
```

22. Real-world recursive problems:

```
- Tree traversal, graph DFS, backtracking, divide & conquer algorithms.
```

23. Recursive variadic function parsing (simulate):

```
- Use structs to nest calls:
    sum(1, sum(2, 3), 4) => represented with node trees.
```

24. Command parser with function pointers:

```
    if (strcmp(cmd, "add") == 0) fn = add;
```

25. Function pointers and recursion can cause stack overflows:

```
- Especially with deep/indirect recursion. Always include base cases.
```