1. What is a macro in C, and how is it defined?

A **macro** in C is a preprocessor directive used to define constants, functions, or code snippets that are replaced by the preprocessor before the compilation process begins. Macros are typically used for code simplification, debugging, and performance optimization.

Definition: Macros are defined using the #define preprocessor directive.

Ex:

```
#define PI 3.14    // Constant macro

#define SQUARE(x) ((x) * (x))  // Function-like macro
```

What is the difference between macros and functions?

Macros are expanded by the preprocessor before the actual compilation begins. They are essentially text replacements and are not part of the compiled code itself.

Advantages: Faster execution (no function call overhead) and flexibility with parameter types.

Disadvantages: Can lead to code bloat, harder debugging, and unexpected behavior due to lack of type checking.

Functions are actual code entities compiled into the final executable. They allow for type checking and can be used for more complex operations.

Advantages: Type safety, easier debugging, code reuse.

Disadvantages: Function call overhead, can be slower than macros.

What do #ifdef, #ifndef, and #endif do?

These are conditional preprocessor directives used to control the compilation process based on whether certain conditions are met.

- #ifdef <macro>: Checks if a macro is defined. If defined, the code between #ifdef and #endif is included in the compilation.
- #ifndef <macro>: Checks if a macro is **not** defined. If not defined, the code between #ifndef and #endif is included in the compilation.
- #endif: Marks the end of a preprocessor conditional block

What does malloc() do, and what type does it return?

The malloc() function in C dynamically allocates memory on the heap and returns a pointer to the allocated memory.

- **Return type**: void * (pointer to the allocated memory block)
- **Syntax**: void* malloc(size_t size);

What is the difference between malloc(), calloc(), and realloc()?

- **malloc(size_t size)**: Allocates a block of memory of the specified size. The contents of the allocated memory are uninitialized.
- **calloc(size_t num, size_t size)**: Allocates memory for an array of num elements, each of size size, and initializes all the memory to zero.
- **realloc(void *ptr, size_t size)**: Resizes a previously allocated memory block pointed to by ptr. It can either expand or shrink the block. If the new block is larger, it may move to a new memory location.
- **Why must we always call `free()` after dynamic allocation?**
- The `free()` function releases dynamically allocated memory back to the heap. Failing to call `free()` after using functions like `malloc()`, `calloc()`, or `realloc()` leads to **memory leaks**, where the memory is not returned to the system, potentially exhausting available memory and causing performance degradation.

What is a header guard, and what problem does it solve?

A **header guard** prevents the multiple inclusion of the same header file, which can lead to errors like redefinition of types, functions, and variables. This typically happens when a header file is included in multiple files, and each of those files includes it again indirectly.

How does the preprocessor handle nested includes?

When the preprocessor encounters a #include directive, it opens the included file and processes it as if its contents were directly included in the original file. In the case of **nested includes** (when one file includes another file, and that file includes yet another), the preprocessor recursively processes all the files.

Header guards are essential here, as they prevent the same file from being processed multiple times, which could otherwise cause redefinition errors.