Summary

Problem 1 – Sorting with Macros and Conditional Compilation

You learned how to use two different sorting functions—one for ascending and one for descending order—and control which one is used by defining a macro. The #ifdef directive checks if a macro is defined and runs specific code based on that. This shows how macros and conditional compilation can control program behavior.

Problem 2 – Arithmetic Using Macros Instead of Functions

You created macro-based arithmetic operations like addition and subtraction. Instead of writing actual functions, you used macro-like expressions that act like shortcuts. Macros are handled by the preprocessor, not at runtime like functions. They are faster but don't do type checking or error handling like real functions.

Problem 3 – Dynamic Memory Allocation with malloc, calloc, realloc

You used malloc to allocate memory during runtime, calloc to allocate and initialize memory to zero, and realloc to resize a memory block. You learned that malloc returns a pointer to the start of the allocated memory. Always check if the allocation was successful, and always free the memory after using it to avoid memory leaks.

Problem 4 – Free and Allocate a Larger Memory Block

Instead of resizing with realloc, you freed a memory block and then allocated a new one with more space. This showed how freeing memory makes space available again, but not necessarily large enough for a bigger block. If system memory is limited or fragmented, the new allocation might still fail.

Problem 5 – User-Input Array and Sum Calculation

You asked the user how many numbers they wanted to enter, allocated memory dynamically using malloc, allowed input into the array, calculated the sum, and then freed the memory. This reinforced the use of pointers, loops, and memory management in a practical way.

Problem 6 – Using a Struct and Header File for Math Operations

You created a header file that defined a struct to hold two numbers and declared functions for math operations like addition and exponentiation. These functions were implemented in a separate source file and used in the main program. You also used a header guard to prevent multiple inclusions of the same file.

Key Concept Answers

- **Macro in C:** A macro is a preprocessor directive that defines a constant or expression before the code is compiled. It is written using #define.
- **Difference Between Macros and Functions:** Macros are replaced directly into the code and don’t use memory or function calls. Functions are type-checked and safer but run at runtime and use stack memory.
- **#ifdef, #ifndef, #endif:** These are conditional compilation directives. They allow you to include or exclude code depending on whether a macro is defined.
- **malloc():** Allocates a block of memory at runtime and returns a pointer to it. The type returned is a void*, which is usually cast to the appropriate type.
- **malloc vs calloc vs realloc:**
  - malloc allocates uninitialized memory.
  - calloc allocates and zero-initializes memory.
  - realloc resizes an already allocated block.
- **Why Use free():** If you don’t free dynamically allocated memory, it remains reserved and causes memory leaks, which waste system resources.
- **Header Guard:** A header guard prevents the same header file from being included more than once. Without it, the compiler might see duplicate definitions and throw errors.
- **Header Guard Format:** Usually defined using #ifndef, #define, and #endif with a unique name to protect the contents of the header.
- **Preprocessor and Nested Includes:** The preprocessor processes include in order. If header guards are present, it avoids including the same file more than once, even if it's nested through other includes.