C Programming For Absolute Beginners & Question Set III

Files, Structs and the Standard Library Sahas Talasila

Files Introduction

Open a file: `fopen()`
Close a file: `fclose()`
Write to a file: `fprintf()`, `fputs()`, `fwrite()`
Read from a file: `fscanf()`, `fgets()`, `fread()`
Move the file pointer: `fseek()`, `ftell()`

Persistent data storage:
Files allow you to store data permanently on disk, even after the program terminates. This is useful for applications that need to retain data between runs, such as databases, configuration files, or log files.

Input/Output operations:
Files provide a way to read and write data to/from external storage devices, such as hard drives, solid-state drives, or flash drives. This enables your program to interact with the outside world, reading input from files and writing output to files.

Modularity and organisation:
By breaking down a large program into smaller, independent files, you can:
1. Organise code into logical modules or components.
2. Make it easier to maintain and update individual files without affecting the entire program.
3. Allow multiple developers to work on different files simultaneously.

Reusability:
Files can be designed to be reusable across multiple programs or projects, making it easier to share code and reduce duplication.

Portability:
Files can be used to store platform-independent data, allowing your program to run on different operating systems or architectures without modification.

Error handling and debugging:
Files provide a way to log errors, debug information, or other relevant data, making it easier to diagnose and fix issues in your program.

Configuration and settings:
Files can be used to store configuration settings, user preferences, or other application-specific data, allowing your program to adapt to different environments or user preferences.

```c
// Writing to a File
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "w"); // Open file in write mode
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fprintf(file, "This is a test file.\n");
    fprintf(file, "Writing to files in C is simple!\n");

    fclose(file); // Close the file
    printf("Data written successfully.\n");

    return 0;
}
```

What if we want to read from a file?

```c
// Reading from a File
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "r"); // Open file in read mode
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    char line[100];
    while (fgets(line, sizeof(line), file)) { // Read lines from file
        printf("%s", line);
    }

    fclose(file); // Close the file
    return 0;
}
```

Using `fwrite` and `fread` for Binary Files

```c
#include <stdio.h>
#include <stdlib.h>

struct Person {
    char name[50];
    int age;
    float height;
};

int main() {
    struct Person p = {"John Doe", 30, 5.9};
    FILE *file = fopen("person.dat", "wb"); // Open file in binary write mode
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fwrite(&p, sizeof(struct Person), 1, file); // Write struct to file
    fclose(file);

    // Reading back the binary data
    file = fopen("person.dat", "rb"); // Open file in binary read mode
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    struct Person readPerson;
    fread(&readPerson, sizeof(struct Person), 1, file); // Read struct from file
    fclose(file);

    printf("Name: %s, Age: %d, Height: %.2f\n", readPerson.name, readPerson.age,
   readPerson.height);

    return 0;
}
```

Structs

A struct (short for structure) in C is a user-defined data type that allows you to group variables of different data types under a single name. This grouping makes it easier to organize related data. Unlike arrays (which store multiple elements of the same type), a struct can store a mix of different data types (e.g., `int, float, char[]`, etc.).

Key Points

- Definition: You define a struct with the `struct` keyword, followed by a structure tag (optional) and curly braces containing members of the struct (or fields).
- Members: Each member has its own type and name.
- Instantiation: You can create variables of that struct type once it is defined.
- Accessing Members: Use the dot (.) operator to access members of a struct instance, e.g., `myStructVariabl`
- Pointers: When working with pointers to structs, you use the arrow operator (-¿) to access members, e.g., `myStructPointer->memberName`.

```c
#include <stdio.h>
#include <string.h>

// Define a struct to store person's information
struct Person {
    char name[50];
    int age;
};

int main(void) {
    // Create an instance of struct Person
    struct Person john;

    // Assign values to the struct members
    strcpy(john.name, "John Doe");
    john.age = 30;

    // Print out the member values
    printf("Name: %s\n", john.name);
    printf("Age : %d\n", john.age);

    return 0;
}
```

Listing 1: Defining and using a simple struct in C

Explanation:

- `struct Person` is declared with two members: a `char` array `name` (to hold a string) and an `int age`.
- In `main`, we create a variable `john` of type `struct Person`.
- We use `strcpy` to copy the string ``John Doe" into `john.name`.
- We directly assign an integer to `john.age`.
- We print the values to confirm that the `struct` was populated correctly.

Struct Initialisation and Functions
You can also initialise (creates) structs at the time of declaration or pass them to functions.

```c
#include <stdio.h>

// Define a struct to store 2D point coordinates
struct Point {
    float x;
    float y;
};

// Function to print the coordinates of a Point
void printPoint(struct Point p) {
    printf("Point: (%.2f, %.2f)\n", p.x, p.y);
}

int main(void) {
    // Initialize a struct Point instance
    struct Point p1 = {3.5f, 4.8f};

    // Pass the struct to a function
    printPoint(p1);

    return 0;
}
```

Listing 2: Initializing a struct and passing it to a function

Explanation:

- `struct Point` holds two floating-point members, `x` and `y`.
- We define `printPoint` that accepts a `struct Point` argument.
- In `main\, we initialise the \verb`struct p1— with the values `{3.5f, 4.8f}`.
- We pass `p1` to `printPoint`, which prints out its coordinates.

Structs with Nested Structs:
Structs can also contain other structs, which allows complex data structures:

Example 4: Nested Struct Example:

```c
#include <stdio.h>
#include <string.h>

struct Address {
    char street[50];
    char city[50];
    char country[50];
};

struct Person {
    char name[50];
    int age;
    struct Address address;  // nested struct
};

int main(void) {
    // Create a Person and fill in details
    struct Person alice;
    strcpy(alice.name, "Alice Wonderland");
    alice.age = 28;

    strcpy(alice.address.street, "123 Imaginary Rd");
    strcpy(alice.address.city, "Fictional City");
    strcpy(alice.address.country, "Neverland");

    // Print out the nested info
    printf("Name: %s\n", alice.name);
    printf("Age : %d\n", alice.age);
    printf("Address: %s, %s, %s\n",
           alice.address.street,
           alice.address.city,
           alice.address.country);

    return 0;
}
```

Listing 3: A struct that contains another struct

Explanation:

The `Person struct` has its own members (`name, age`) and a nested `struct Address`. We use the dot operator multiple times to access nested fields (e.g. `alice.address.street`).

Understanding Header Files in C

This document provides a comprehensive guide to header files in C, explaining their purpose, how the compiler processes them, and the role of preprocessor directives.

2.1 What is a Header File in C?

A header file in C is a file with a `.h` extension that contains declarations of functions, variables, macros, and other definitions shared across multiple source files (`.c` files). Header files promote modularity and avoid code duplication by keeping your declarations in one spot. Your life is much easier with stuff organised and without unnecessary clutter, etc. The same principle applies to computers as well. We will look at the requirements on the next page:

Key contents of a header file include:

- **Function prototypes**: Declare function signatures without implementation.
- **Macro definitions**: Define constants or inline code using `#define`.
- **Type definitions**: Declare structs, unions, enums, or typedefs.
- **Global variable declarations**: Use `extern` for variables defined elsewhere.
- **Include guards**: Prevent multiple inclusions of the same header.

Here is an example of a header file, `math_utils.h`:

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

// Function prototypes
int add(int a, int b);
double square(double x);

// Macro definition
#define PI 3.14159

// Struct definition
typedef struct {
    double x;
    double y;
} Point;

#endif
```

Listing 4: Example Header File

2.2 Purpose of Header Files

Header files serve several critical purposes:

- **Modularity**: Separate interface (declarations) from implementation (definitions).
- **Code Reusability**: Centralize declarations for use across multiple files.
- **Avoid Duplication**: Prevent repetitive declarations, reducing maintenance issues.
- **Encapsulation**: Expose interfaces while hiding implementation details.
- **Standardisation**: Provide consistent interfaces, as in standard library headers (e.g., `<stdio.h>`).

For example, in a program with `main.c` and `math_utils.c`, a header file like `math_utils.h` centralizes the `add` function declaration, simplifying maintenance.

2.3 How the Compiler Handles Header Files

The C compiler processes header files during the **preprocessing phase**, before compilation. This phase is managed by the preprocessor, a distinct component of the compiler that performs text-based transformations on the source code (check the first set of notes for this!).

The preprocessor reads the source file, processes all directives starting with `#`, and produces a single translation unit, which is then passed to the compiler for further processing. When handling header files, the preprocessor performs several key tasks. When it encounters a `#include` directive, such as `#include "math_utils.h"` or `#include <stdio.h>`, it replaces the directive with the entire content of the specified header file, effectively inlining the header's content into the source file at the point of inclusion.

For `#include "file"`, the preprocessor searches the current directory first, then system include paths; for `#include <file>`, it searches only system include paths. To prevent multiple inclusions, which could cause redefinition errors (e.g., duplicate function prototypes or macros), header files use include guards or `#pragma once`. An `include` guard employs conditional directives.

1. **Preprocessing**: The `#include "math_utils.h"` directive is replaced with the entire content of `math_utils.h`, a text substitution performed by the preprocessor.
2. **Include Guards**: Prevent multiple inclusions using:

```
#ifndef HEADER_NAME
#define HEADER_NAME
// Header content
#endif
```

Listing 5: Header guards for the header file

The preprocessor checks if `HEADER_NAME` is defined; if not, it defines it and processes the header content; if defined, it skips the content, ensuring the header is included only once per translation unit. The `#pragma once` directive, though non-standard and compiler-dependent, achieves the same effect more concisely. The preprocessor also expands all macros defined in the header or source file.

For example, a macro like `#define PI 3.14159` replaces every occurrence of `PI` with `3.14159`, and function-like macros, such as `#define SQUARE(x) ((x) * (x))`, are expanded with their arguments, with recursive resolution of nested macros. Conditional directives like `#ifdef`, `#ifndef`, `#if`, `#elif`, and `#else` are evaluated to include or exclude code blocks, enabling platform-specific or configuration-dependent code. Comments (both `/* */` and `//`) are removed, and line continuations (lines ending with \) are handled to ensure proper formatting. The preprocessor can also generate errors or warnings via `#error` or `#warning` directives, such as enforcing a standard C compiler requirement.

After processing all directives, the preprocessor produces a single translation unit, combining the source file's content with all included headers expanded, macros replaced, and conditional blocks resolved. Developers can inspect this output using compiler flags like `gcc -E source.c`, which is useful for debugging macro expansions or include issues. Recursive includes, where a header includes other headers, are processed in turn, but circular includes (e.g., header A includes B, and B includes A) require forward declarations or include guards to avoid errors.

Excessive or nested includes can slow preprocessing, so minimizing unnecessary includes or using precompiled headers (where supported) optimizes compilation. Common issues include missing include guards leading to redefinition errors, incorrect include paths causing "file not found" errors, and macro misuse (e.g., omitting parentheses in `#define SQUARE(x) x * x`) producing incorrect results due to operator precedence.

Example Workflow:

- Source files: `main.c`, `math_utils.c`.
- Header: `math_utils.h` with `add` prototype.
- `main.c` includes `math_utils.h`.
- Preprocessor replaces `#include` with header content.
- Compiler generates object code for each source file.
- Linker resolves `add` calls in `main.c` to definitions in `math_utils.c`.

Potential Issues:

- **Multiple inclusions**: Cause redefinition errors without include guards.
- **Missing definitions**: Lead to linker errors (e.g., "undefined reference").

Ignore after this page. still undergoing work. most likely will be moved to the advanced course.

Preprocessor Directives (Extra)

The C preprocessor modifies source code before compilation using directives starting with `#`. Below is a detailed explanation of each directive.

3.0.1 `#include`

- **Purpose**: Includes another file's content.
- **Syntax**:
  - `#include <file>`: For standard headers (e.g., `<stdio.h>`), searches system paths.
  - `#include "file"`: For user headers (e.g., `"math_utils.h"`), searches current directory first.
- **Example**:

```
#include <stdio.h>
#include "math_utils.h"
```

Listing 6: Header File referencing/usage

3.0.2 `#define` and `#undef`

- **Purpose**: `#define` creates macros (constants or function-like); `#undef` removes them.
- **Syntax**:

```
#define NAME value
#define NAME(param) value
#undef NAME

```

Listing 7: Preprocessor Define

- **Example**:

```
#define MAX_SIZE 100
#define SQUARE(x) ((x) * (x))
#undef MAX_SIZE
```

Listing 8: #define Example

- **Note**: Macros are text replacements; use parentheses to avoid precedence issues.

3.0.3 `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`

- **Purpose**: Conditional compilation based on macro definition.
- **Syntax**:

```
#ifdef MACRO
// Code if MACRO is defined
#else
// Code if MACRO is not defined
#endif

#ifndef MACRO
// Code if MACRO is not defined
#endif

```

Listing 9: Macro example

- **Example** (Include guard):

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H
int add(int a, int b);
#endif
```

Listing 10: Include guard

3.0.4 `#if`, `#else`, `#elif`, `#endif`

- **Purpose**: Conditional compilation based on an expression.
- **Syntax**:

```
#if expression
// Code if expression is non-zero
#elif expression
// Code if previous conditions false
#else
// Code if all conditions false
#endif
```

Listing 11: Conditional compilation

- **Example**:

```
#define VERSION 2
#if VERSION == 1
#define FEATURE "Basic"
#elif VERSION == 2
#define FEATURE "Advanced"
#else
#define FEATURE "Unknown"
#endif
```

Listing 12: USe case of conditional compilation

3.0.5 `#pragma`

- **Purpose**: Compiler-specific instructions. Pragma is the much newer method and is used to replace the `#define` directive.
- **Common Uses**: `#pragma once` (alternative to include guards), `#pragma pack` (struct alignment).
- **Example**:

```
#pragma once
struct Data {
    char c;
    int i;
};
```

Listing 13: Pragma usage

- **Note**: Non-portable, varies by compiler.

3.0.6 `#error`

- **Purpose**: Generates a compilation error with a message.
- **Example**:

```
#if !defined(__STDC__)
#error "This code requires a standard C compiler"
#endif
```

Listing 14: Error directive example

3.0.7 `#warning`

- **Purpose**: Issues a warning (non-standard, supported by GCC/Clang).
- **Example**:

```
#warning "This code is deprecated"
```

Listing 15: #warning directive

3.0.8 #line

- **Purpose**: Changes reported line number and filename.
- **Example**:

```
#line 100 "custom.c"
int x = 1 / 0; // Error reported at custom.c:100
```

Listing 16: #line directive

3.1 Predefined Macros

Common predefined macros include:

- __FILE__: Current file name.
- __LINE__: Current line number.
- __DATE__: Compilation date.
- __TIME__: Compilation time.
- __STDC__: Defined if standard C.

Example:

```
printf("File: %s, Line: %d\n", __FILE__, __LINE__);
```

Listing 17: Predefined Macros example

Stringification (#) and Token Pasting (##)

- **Stringification**: Converts a macro argument to a string.
- **Token Pasting**: Concatenates tokens.
- **Example**:

```
#define STR(x) #x
#define CONCAT(a, b) a##b
int main() {
    printf("%s\n", STR(hello)); // Prints "hello"
    int xy = 10;
    printf("%d\n", CONCAT(x, y)); // Prints 10
}
```

Listing 18: Macro argument to string conversion

Example: Putting It All Together

Below is a complete example with a header file, source file, and main program. The compiler will parse the `main.c` file and realise that the header file and utils file has been called upon, so it will then use them to complete the area in our case.

Header File (`math_utils.h`):

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

#define PI 3.14159
#define SQUARE(x) ((x) * (x))

typedef struct {
    double x;
    double y;
} Point;

int add(int a, int b);
```

```c
double circle_area(double radius);

#endif
```

Listing 19: Header file construction

Source File (`math_utils.c`):

```c
#include "math_utils.h"

int add(int a, int b) {
    return a + b;
}

double circle_area(double radius) {
    return PI * SQUARE(radius);
}
```

Listing 20: math_utils.c file

Main Program (`main.c`):

```c
#include <stdio.h>
#include "math_utils.h"

int main() {
    Point p = {1.0, 2.0};
    printf("Sum: %d\n", add(3, 4));
    printf("Circle area: %.2f\n", circle_area(5.0));
    printf("Point: (%.1f, %.1f)\n", p.x, p.y);
    #ifdef DEBUG
    printf("Debug mode enabled\n");
    #endif
    return 0;
}
```

Listing 21: main.c file

Compilation:

```
gcc -o program main.c math_utils.c
```

This example demonstrates how the preprocessor replaces `#include` with header content, include guards prevent duplication, and the linker resolves function calls.

Best Practices for Header Files

- Use include guards or `#pragma once`.
- Avoid definitions in headers; use `extern` for variables.
- Keep headers minimal to reduce compilation time.
- Match header and source file names (e.g., `math_utils.h`, `math_utils.c`).
- Document functions and macros in headers.

Common Pitfalls

- **Missing include guards**: Cause redefinition errors.
- **Unnecessary includes**: Increase compilation time.
- **Circular includes**: Use forward declarations to resolve.
- **Macros with side effects**: Avoid macros like `#define DOUBLE(x) x * 2` with expressions like `DOUBLE(i++)`.

To achieve a comprehensive understanding of the C standard library for an advanced C programming course, a wide range of topics must be covered, spanning fundamentals, implementation details, integration with the toolchain, and advanced considerations for expert programmers. The following list outlines all critical aspects, ensuring a thorough grasp of the library's structure, functionality, and nuances. Each aspect is designed to build foundational knowledge while addressing complex issues relevant to advanced applications.

- **Definition and Purpose**: Understand the C standard library as a standardized collection of headers, functions, macros, and types defined by the C standard (e.g., ISO/IEC 9899:2011, C11). Recognize its role in providing portable functionality for tasks like input/output, memory management, string manipulation, and mathematical computations, abstracting platform-specific details.
- **Standard Headers**: Study the 24 headers in C11, including their purposes and contents:
  - `<assert.h>`: Diagnostics (`assert`).
  - `<complex.h>`: Complex number arithmetic.
  - `<ctype.h>`: Character classification (`isalpha`, `tolower`).
  - `<errno.h>`: Error handling (`errno`).
  - `<fenv.h>`: Floating-point environment control.
  - `<float.h>`: Floating-point type limits.
  - `<inttypes.h>`: Integer type formatting.
  - `<iso646.h>`: Alternative operator spellings.
  - `<limits.h>`: Integer type limits.
  - `<locale.h>`: Localisation (`setlocale`).
  - `<math.h>`: Mathematical functions (`sin`, `pow`).
  - `<setjmp.h>`: Non-local jumps (`setjmp`, `longjmp`).
  - `<signal.h>`: Signal handling (`raise`, `signal`).
  - `<stdalign.h>`: Alignment control.
  - `<stdarg.h>`: Variable argument lists (`va_start`).
  - `<stdatomic.h>`: Atomic operations (C11).
  - `<stdbool.h>`: Boolean type (`bool`).
  - `<stddef.h>`: Standard types and macros (`size_t`, `NULL`).
  - `<stdint.h>`: Fixed-width integer types (`uint32_t`).
  - `<stdio.h>`: Input/output (`printf`, `fopen`).
  - `<stdlib.h>`: Utilities (`malloc`, `rand`).
  - `<string.h>`: String manipulation (`strlen`, `strcpy`).
  - `<threads.h>`: Thread support (C11, `thrd_create`).
  - `<time.h>`: Time and date (`time`, `strftime`).
- **Header Contents**: Examine the components of headers, including function prototypes (e.g., `int printf(const char *restrict format, ...)`), macros (e.g., `NULL`, `EOF`), and type definitions (e.g., `FILE`, `div_t`). Understand the use of `restrict` for pointer aliasing optimisation and `inline` functions (C99/C11).
- **Runtime Library**: Explore the runtime library, which provides implementations of declared functions as precompiled object files (`libc.a`) or shared libraries (`libc.so`, `msvcrt.dll`). Learn how these libraries interact with the operating system via system calls (e.g., `write`, `mmap`).
- **Implementation Variations**: Analyze major implementations (glibc, MSVCRT, newlib, musl), their design goals (performance, size, correctness), and differences in extensions, error handling, and performance characteristics. For example, glibc's `printf` supports POSIX `%m`, while MSVCRT does not.
- **Preprocessing Phase**: Understand how the preprocessor handles `#include <stdio.h>`, inlining header content, expanding macros (e.g., `NULL` to `((void *)0)`), and evaluating conditional directives (e.g., `#ifdef __STDC__`). Study include paths and potential errors (e.g., missing headers).
- **Compilation Phase**: Learn how the compiler verifies function calls against declarations, ensuring type safety and generating object code with unresolved symbols (e.g., `_printf`). Understand compiler optimisations like inlining `memcpy` or replacing functions with intrinsics.

- **Linking Phase**: Study how the linker resolves standard library references, using static (`libc.a`) or dynamic (`libc.so`) linking. Explore linker options (`-lc`, `-nostdlib`) and symbol versioning for compatibility (e.g., `memcpy@GLIBC_2.2.5`).
- **Hosted vs. Freestanding Environments**: Differentiate between hosted environments (full library, `main` entry point) and freestanding environments (minimal headers, custom entry points like `reset_handler`). Study the nine freestanding headers and the need for custom implementations.
- **Thread Safety**: Investigate thread safety of functions (e.g., `strtok`, `rand` are non-thread-safe due to static state). Learn about thread-safe alternatives (e.g., `strtok_r`, POSIX-specific) and synchronisation techniques (e.g., mutexes).
- **Reentrancy**: Understand reentrancy, where functions avoid shared state (e.g., `strlen` is reentrant, `localtime` is not). Explore C11's `_Thread_local` for thread-local storage and its impact on reentrancy.
- **Performance Considerations**: Analyze performance factors:
  - **I/O Buffering**: Study buffering modes (`_IOFBF`, `_IOLBF`, `_IONBF`) and control via `setvbuf`, `fflush`.
  - **Memory Allocation**: Examine allocators (ptmalloc, jemalloc) and issues like fragmentation, metadata overhead.
  - **Mathematical Functions**: Explore `<math.h>` optimisations (hardware instructions, lookup tables) and compiler flags (`-ffast-math`).
  - **Inline Optimisations**: Understand compiler inlining and intrinsics for functions like `memset`.
- **Portability Issues**: Address implementation-defined behaviors, including type sizes (`size_t`, `int`), `errno` values, locale-specific functions (`strftime`), and floating-point limits (`<float.h>`).
- **Error Handling**: Learn proper error checking for functions (e.g., `NULL` from `malloc`, `fopen`) and use of `errno`, `perror`, or `strerror`.
- **Undefined Behavior**: Study cases where standard library functions trigger undefined behavior (e.g., `NULL` to `strlen`, buffer overflows in `strcpy`).
- **Safe Functions**: Compare unsafe functions (`strcpy`, `sprintf`) with safer alternatives (`strncpy`, `snprintf`, `strlcpy` where available).
- **Extending the Library**: Explore creating custom functions that mimic standard library conventions, including header files, error handling, and const-correctness. Example:

```
// mystring.h
#ifndef MYSTRING_H
#define MYSTRING_H
char *strrev(const char *str);
#endif

```

Listing 1: Header for a custom `strrev` function.

- **Debugging Tools**: Master tools for debugging standard library usage:
  - `valgrind`: Detect memory leaks and invalid accesses.
  - `strace`: Trace system calls (e.g., `open`, `write`).
  - `gdb`: Step into library functions with debug symbols.
  - `nm`, `objdump`: Inspect symbols and disassemble code.
  - `perf`: Profile performance bottlenecks.
- **Profiling Techniques**: Learn to profile standard library functions (e.g., `printf`, `malloc`) to identify inefficiencies using `perf`, `valgrind`, or `gprof`.
- **Low-Level Interactions**: Understand how functions interact with the operating system (e.g., `fopen` uses `open`, `malloc` uses `sbrk`/`mmap`). Study system call overhead and optimisation strategies.
- **Custom Implementations**: Practice writing custom versions of standard library functions (e.g., `memcpy`, `strlen`) for freestanding environments or performance optimisation.
- **Inline Assembly**: Explore replacing standard library functions with inline assembly for performance (e.g., `memset` using `rep stosb` on x86).
- **Cross-Platform Development**: Use conditional compilation (`#ifdef __linux__`, `#ifdef _WIN32`) to

handle platform differences, ensuring portability.

- **Standard Compliance**: Study the C standard (C11) to understand guaranteed behaviors vs. implementation-defined or undefined behaviors. Use `<assert.h>` for runtime checks.
- **Extensions and Non-Standard Functions**: Recognize implementation-specific extensions (e.g., glibc's `strlcpy`, POSIX functions) and their impact on portability.
- **Memory Management Nuances**: Understand `malloc`, `calloc`, `realloc`, `free`, including alignment, padding, and fragmentation issues.
- **Floating-Point Behavior**: Explore `<math.h>` functions and `<fenv.h>` for floating-point control (e.g., rounding modes, exceptions).
- **Locale and Internationalisation**: Learn how `<locale.h>` affects functions like `printf`, `strftime`, and `tolower`, and how to manage locales.
- **Signal Handling**: Study `<signal.h>` for handling signals (`raise`, `signal`) and their interaction with library functions.
- **Non-Local Jumps**: Understand `<setjmp.h>` for non-local control flow (`setjmp`, `longjmp`) and their use in error handling.
- **Atomic Operations**: Explore C11's `<stdatomic.h>` for atomic operations in multithreaded programs, including memory ordering semantics.
- **Thread Support**: Study C11's `<threads.h>` for thread creation (`thrd_create`), synchronisation, and limitations compared to POSIX threads.
- **Best Practices**: Adopt practices like error checking, const-correctness, using `<stdint.h>` for portable types, and avoiding unsafe functions.
- **Pitfalls**: Identify common pitfalls, including undefined behavior, buffer overflows, resource leaks, and non-thread-safe functions in multithreaded contexts.
- **Documentation and Resources**: Consult the C standard, implementation documentation (e.g., glibc source, MSVC docs), and man pages for function details and platform-specific behavior.

4.1 Multiple Choice

4.2 Short Form Answers

4.3 Long Form/Open Ended Questions

4.4 Spot The Error Questions