Systems Programming Home Work Assignment-1

Part - I

I.1

The provided program is intended to perform operations based on user input. Multiplication, division, absolute value calculation, and string concatenation can be done based on the User's input. But the program has several issues that can lead to compilation errors, runtime errors, and logical flaws.

The issues include invalid use of switch with string literals, missing function declarations, potential division by zero, incorrect handling of command-line arguments, inadequate memory allocation for string concatenation, and the absence of proper error handling and messaging. Correcting these issues will involve changing the switch statement to if-else, adding necessary function prototypes, including the required header files, adding proper checks for valid inputs, and ensuring clear and precise user instructions. By addressing these problems, the program can achieve its intended functionality in a reliable manner.

Potential Compilation Errors and Warnings

1. The switch statement in C is designed to work with integral types such as int or char. In the provided code, the switch statement is used with a char * (string) type and compares it with string literals like "1", "2". This is not valid in C and will cause a compilation error. This can be solved by using if-else statements combined with strcmp for string comparisons. This can also be solved by type casting input arguments and converting into integers. The problem can also be solved by type casting arguments into characters such that it is compatible with switch case statements unlike untraditional string literal.
2. The functions **mymul**, **mydiv**, **myabs**, and **myconcat** are called in the main function before their definitions appear in the code. In C, functions should either be defined before they are used or declared with prototypes at the beginning of the file. This oversight will lead to compilation warnings about these declarations.
3. The program uses standard library functions like strlen and strcpy, which are declared in the string.h header file. This header file is not included in the program, leading to compilation warnings or errors.

Potential Runtime Errors

1. The division operation in the **mydiv** function does not check if the divisor b is zero. Dividing by zero will result in undefined behavior can lead to potential run time error.
2. In the line "op2 = argv[3];" If the program is run with only three arguments (argc == 3), accessing argv[3] will result in undefined behavior because argv[3] does not exist. This can lead to runtime crashes.
3. The **myabs** function is intended to compute the absolute value of an integer but it is modifying the input argument in place rather than returning the result. This does not align with typical usage in C, where functions generally return the computed result.

Failures to Handle Unexpected Inputs

1. The atoi function is used to convert the input strings to integers without checking if the conversion is successful. If the input strings are not valid integers, atoi will return 0, which might not be the intended behavior. A better approach would be to use strtol and validate the conversion.
2. The myconcat function allocates memory for the concatenated string but does not account for the null terminator. The correct allocation can be like "malloc(strlen(a) + strlen(b) + 1)" to ensure there is enough space for the concatenated result plus the null terminator.
3. The program does not handle the case where an operand is missing for operations that require two operands (like multiplication, division, and concatenation). This can lead to accessing invalid memory or producing incorrect results. The program does not check if Op2 is provided or not. If Op2 is missing, the program might use an uninitialized pointer, leading to invalid memory access or incorrect results.

Inadequacies of the Intention

1. The switch cases lack break statements, which will cause fall-through behavior. For example, if type is "1", the program will execute the multiplication case and then proceed to execute the division, absolute value, and concatenation cases as well. This is likely not the intended behavior.
2. The program does not clearly separate the logic for each operation. Using if-else statements would be more appropriate to ensure that only the relevant code for the selected operation is executed.
3. The print_usage function does not provide clear guidance on how to use the program with different types of operations, particularly for string concatenation. Users may not know that they need to provide string operands in quotes.

I.2

The code:

while (count-- > 0) { *dest = *src++; }

is correct if dest is a pointer to a volatile memory location, such as a hardware register. This ensures that each assignment to *dest triggers an action (e.g., sending data to a peripheral device), making each write operation meaningful and necessary. The **volatile** keyword prevents the compiler from optimizing out these repeated writes.

The loop initially seems ineffective as it assigns all elements of src to a single memory location pointed to by dest, overwriting previous values. However, if dest points to a memory-mapped I/O location, this loop can be used to send a series of values to a device. In this scenario, dest being a volatile pointer, indicating to the compiler that dest can change in ways not specified by the program, ensuring each write operation occurs as intended for correct device operation.

I.3

(a) int a, *b;

a is an integer variable. b is a pointer to an integer.

Both “a” and “b” are Valid and are properly declared.

(b) double (fs[])(int);

fs is an array of functions, where each function takes an int argument and returns a double.

This is Valid. This declares an array of function pointers.

(c) double *(*f)(double d);

f is a pointer to a function that takes a double argument and returns a pointer to a double.

This is a correct declaration of a function pointer.

(d) int *const p;

p is a constant pointer to an integer. The pointer itself cannot change to point to another integer, but the integer value it points to can be modified.

This correctly declares a constant pointer to an integer.

(e) int (*cmps[10])(const void *, const void *);

cmps is an array of 10 pointers to functions. Each function takes two const void * arguments and returns an int. This correctly declares an array of function pointers.

I.4

(a)

typedef double (*func_ptr)(int);

typedef func_ptr* arg_type;

typedef void *(*return_func)();

return_func f(arg_type arg);

(b)

void *(*f())();

void *(*f(double (**arg)(int)))();

I.5

(a) If a C function is declared with prototype void f(), but then defined with prototype void f(int a), then the compiler will signal an error about the inconsistency in the prototypes.

This is not a Valid statement. In C, declaring a function with void f() means the function takes an unspecified number of arguments, not necessarily that it takes no arguments. This is different from void f(void), which explicitly states that the function takes no arguments. If you declare void f() and define void f(int a), most standard C compilers **will not signal an error** due to the unspecified parameters in the declaration. However, this is generally considered poor programming practice and can lead to undefined behavior if the function is called incorrectly.

(b) Adding a positive number to a numeric variable will always increase its value.

This is not always correct. This is not always true due to the potential for overflow. For example, if you have an integer variable at its maximum value (`INT_MAX` in C), adding any positive number will cause an overflow, resulting in a wrap-around to negative values (in the case of signed integers), or cycling back through zero (in the case of unsigned integers). Thus, the variable's value does not increase.

(c) Assigning an unsigned char to a int will not lead to arithmetic overflow.

Yes, This is a valid statement. In C, unsigned char values are in the range of 0 to 255. When you assign an unsigned char to an int (which typically has a much larger range, like -2,147,483,648 to 2,147,483,647 for a 4 byte int), the value of the unsigned char fits comfortably within the range of the int. This is a safe operation with no risk of overflow or data loss due to implicit type promotion rules in C.

(d) Calling malloc multiple times for the same pointer will not lead to memory leaks because malloc overwrites the old address with the new one.

This statement is invalid. Each call to malloc allocates a separate block of memory on the heap. If malloc is called multiple times and is assigned a result to the same pointer without freeing the previously allocated memory, one could lose the reference to the previously allocated memory. This results in a memory leak, as the lost memory block cannot be reclaimed during the runtime of the program. malloc does not automatically free the old memory or overwrite it; it merely returns a new memory address.

(e) In C, a typedef defines a new type.

This is not a valid statement. A typedef in C does not create a new data type, instead it provides a new name (alias) for an existing type. This is often used for convenience and to enhance code readability, especially with complex structured data types like structs, unions or function pointers. A typedef merely creates a synonym for another type but does not create a distinct new type in the type system.