Final Submission Report

A. Introduction

We implemented a custom version of the standard C library printf() function, as specified in the GNU/Linux man 3 printf manual page. Our implementation fully replicates the behavior of printf(), including parsing format strings, handling variable arguments, supporting all formatting options (flags, field width, precision, length modifiers, and conversion specifiers), and returning the correct number of characters that would have been written. The implementation, written in C, avoids reliance on standard formatting functions (except for debugging) and includes a comprehensive test program/cases to verify correctness. This report details our approach, implementation, testing strategy, and verification process.

B. Project Structure

myprintf.c

The core implementation resides in myprintf.c, which includes the main formatting functions and helper utilities. Key components are:

Core Functions

- my_vsnprintf(char *output, size_t max_size, const char *format, va_list arguments): The primary function, implementing snprintf-like behavior. It handles six key tasks:
    1. Parsing the format string character by character to identify specifiers (%).
    2. Extracting and applying flags (-, +, #, 0, space).
    3. Managing field width, either fixed (e.g., %5d) or dynamic via * (e.g., %*d).
    4. Handling precision for numeric (e.g., %.2f) and string outputs (e.g., %.3s).
    5. Supporting length modifiers (hh, h, l, ll, j, z, t, L).
    6. Converting arguments into formatted output for all standard specifiers.
- my_snprintf: A wrapper for my_vsnprintf, handling variable arguments via va_start and va_end.
- my_printf: Outputs to stdout using my_vsnprintf and fputs, returning -1 on output errors with errno set to EIO.

Helper Functions

- String Reversal: Reverses strings for number formatting (e.g., building digits in reverse order).
- Integer Conversion: Converts integers to strings in various bases (decimal, octal, hexadecimal, binary), handling signed/unsigned values, padding, and precision.
- Floating-Point Conversion: Formats floating-point numbers for %f, %e, %g, %a, handling special cases like NaN, ±Infinity, and rounding.
- Pointer Conversion: Formats pointers as 0x followed by hexadecimal.

The implementation ensures buffer safety by respecting max_size in my_snprintf, null-terminating output, and handling edge cases like max_size = 0 or 1.

Makefile

The Makefile automates building, running, and testing the project. It includes the following targets:

- make: Compiles code4.c into the test_printf executable and generates output.txt by running the test program.
- make run: Runs the test_printf executable, displaying output to the console.
- make test: Runs the test program and redirects output to output.txt.
- make clean: Removes compiled files (test_printf, output.txt, and .dSYM for macOS).
  The Makefile uses gcc with strict flags (-Wall, -Wextra, -std=c99) to ensure robust compilation. It builds and runs the test program in code4.c, meeting the project requirement to use a single Makefile for both tasks.

C. Implementation Details

Argument Handling

We used stdarg.h to process variable arguments. For each format specifier, the correct type is extracted based on the length modifier (e.g., int for %d, long for %ld, long double for %Lf) and passed to the appropriate conversion function. This ensures type safety and correct formatting.

Format String Parsing

The format string is parsed using a state-machine-like approach, following the order specified in man 3 printf:

1. Detects the start of a specifier (%).
2. Parse flags (+, -, #, 0, space), stored as a bitmask.
3. Parse field width (numeric, e.g., %5d, or dynamic via *).
4. Parse precision (starting with ., numeric, e.g., %.2f, or *).
5. Parse length modifiers (hh, h, l, ll, j, z, t, L).
6. Parse the conversion specifier (d, f, s, etc.).
   Invalid format strings (e.g., % alone) are handled gracefully, though error reporting could be enhanced (see limitations).

Step	What it Parses	Example
1	Start specifier	%
2	Flags	+, -, 0
3	Width	5 in %5d
4	Precision	.2 in %.2f
5	Length modifier	l in %ld
6	Conversion specifier	d, f, s, etc

Demo Table

Supported Specifiers

The implementation supports all standard specifiers:

- Integers: %d, %i (signed), %u (unsigned), %x, %X (hexadecimal, lower/uppercase), %o (octal).
- Floating-Point: %f, %F (fixed-point, lower/uppercase), %e, %E (scientific, lower/uppercase), %g, %G (shortest form), %a, %A (hexadecimal float).
- Character/String: %c (character), %s (string, outputs (null) for NULL).
- Pointer: %p (outputs 0x followed by hexadecimal, 0x0 for NULL).
- Special: %n (stores characters written so far), %% (literal %).

Special Cases

- Floating-Point: Handles NaN (nan/NAN), ±Infinity (inf/INF), with case sensitivity based on specifier (%f vs. %F).
- Strings/Pointers: NULL strings output (null); NULL pointers output 0x0.
- Precision: Zero precision (%.0d) outputs nothing for zero; negative precision is treated as absent for most specifiers.
- Width: Negative width is treated as left-justified with absolute value.
- Errors: Output errors in my_printf return -1 with errno = EIO. Invalid format strings are processed without crashing, though explicit error reporting is limited.

Return Value

The functions return the number of characters that would have been written (excluding the null terminator), as specified in man 3 printf. For my_snprintf, this includes characters truncated due to max_size. For my_printf, a negative value is returned on output errors.

D. Testing

The test program, embedded in code4.c's main function, exercises my_printf in all possible ways, covering:

- Specifiers:
    - Integers: %d (42, -42), %u (3000000000U), %x/%X (255), %o (255).
    - Floating-Point: %f (3.14159, INFINITY, NAN), %e (3.14159), %g (0.000123, 3141590000.0), %a (3.14159).
    - Strings: %s ("Hello, World!", NULL, ""), %c ('A').
    - Pointers: %p (0x12345678, NULL).
    - Special: %n (stores character count), %%.
- Flags: + (%+d), space (% d), - (%-5d), 0 (%05d), # (%#x, %#f).
- Width/Precision:
    - Fixed: %5d, %.2f, %.3s.
    - Dynamic: %*d, %.*f.
    - Edge Cases: Zero precision (%.0d), negative width (%*d with -5), negative precision (treated as absent).

- Length Modifiers: hh (%hhd), h (%hd), l (%ld), ll (%lld), j (%jd), z (%zu), t (%td), L (%Lf).
- Edge Cases:
    - Large numbers (INTMAX_MAX, 1e308).
    - Truncation (my_snprintf with small buffer).
    - Empty strings, zero values, extreme widths (%100d).

The test program runs automatically via make test, redirecting output to output.txt. We verified correctness by comparing output.txt with the output of standard printf() (using a modified code4.c with printf instead of my_printf) via manual inspection and diff output.txt expected.txt. All test cases matched standard printf() behavior, except for minor %g precision differences.

E. How to Run

To build and test the project:

1. Ensure code4.c and the Makefile are in the same directory.
2. Run:
    a. make
    b. make run
    c. make test
    d. make clean

F. Challenges Faced

- Floating-Point Formatting: Ensuring correct precision, rounding, and handling of NaN/±Infinity was complex, especially for %g and %a.
- Flag Combinations: Debugging interactions between flags, width, precision, and length modifiers (e.g., %+0#10.5x) required careful testing.
- Type Casting: Managing signed/unsigned conversions for length modifiers (hh, z, j) was tricky due to type promotion and platform differences.
- Testing: Creating comprehensive test cases to cover all edge cases and verifying against standard printf() was time-consuming.

G. Conclusion

This project provided deep insights into the inner workings of printf(), one of C's most widely used functions. By implementing it from scratch, we gained hands-on experience with:

- Variable argument handling using stdarg.h.
- Format string parsing and error handling.Numeric and string formatting logic for various bases and types.
- Modular code design with helper functions.
- Makefile automation for building and testing.

Our implementation is robust, buffer-safe, and highly compliant with man 3 printf, with minor limitations in %g precision and long double support that can be addressed in future iterations. The comprehensive test program and automated Makefile ensure the project is portable and verifiable.

H. Team Contributions

The project was done by *Pankaj*, *Nida*, and *Poorvith*, we worked together over two weeks to implement, test, and document a custom version of the printf() function. Each of us brought unique ideas and actively participated in all aspects of the development process of the Program. Here's a breakdown of our contributions

Understanding Requirements:
We began by thoroughly reading the *man 3 printf* manual and analyzing the specifications. Each of us researched about how the standard printf works internally, identifying key components, and dividing the responsibilities based on interest and expertise.

Implementation/Coding:
We together wrote the core formatting logic. Pankaj focused on handling numeric conversions and length modifiers; Nida implemented support for string and pointer formatting and helped integrate flag parsing; Poorvith worked on floating-point formatting and buffer management. We regularly peer-reviewed each other's code to ensure that we are on track and suggested any improvements.

Testing and Debugging:
All three of us contributed to writing comprehensive test cases. We debugged failed cases and refined our implementation to match the behavior of the standard *printf*. Each of us even compared the outputs, identifying errors, and making improvements.

Makefile Automation:
We together handled writing the Makefile with input from the others, ensuring smooth compilation, execution, and output comparison.

Apart from this ,We met regularly to discuss progress, troubleshoot bugs, and plan next steps. Open communication and sharing screens during debugging sessions helped us learn from one another and kept the project moving forward efficiently.