CSC 412 Assignment 3: Return of C++

Out: Tuesday, October 3, 2023 03:15 PM
Due: Thursday, October 12, 2023 11:59 AM

Contents

1 Introduction

1.1 Objectives

This assignment will be similar to your work in Assignment 2. However, instead of writing C implementations of UNIX utilities you will be writing C++ implementations of utilities found in the GNU core utilities:

The objectives of this assignment are for you to

1. Refresh your knowledge of C++ programming
2. Learn some common system utilities and how they work
3. Learn how some common system calls work
4. Learn how to utilize C++ for systems programming
5. Expand your knowledge of interprocess communication in C++
6. Learn how to create a Makefile
7. Learn why a Makefile is useful

This is a solo assignment.

1.2 Preliminaries

1. A computer with Docker installed to run the CSC 412's development environment.
2. Familiarity with C++ programming.
3. Familiarity with stdin and stdout.

1.3 Resources

The links below are clickable.

GNU

1. Linux man pages
2. GNU Coreutils
3. Makefile Tutorial

C++

1. C++ Tutorial - Tutorialspoint

Makefile

1. Makefile Tutorial - Tutorialspoint
2. Makefile Manual - GNU
3. Makefile Quick Reference - GNU
4. Makefile Implicit Variables - GNU
5. Makefile Implicit Rules - GNU
6. Makefile Pattern Examples - GNU

2 Part I: GNU Coreutils

2.1 Overview

You will be writing basic C++ implementations of the following GNU core utilities:

1. **head** - output the first 10 lines of files OR stdin if no files are provided
2. **pwd** - print name of current/working directory
3. **tail** - output the last 10 lines of files OR stdin if no files are provided
4. **tee** - read from standard input and write to standard output and files (taken as arguments). If not files are given, just write to stdout.
5. **wc** - print line, word, and byte counts for each file OR stdin if no files are provided

2.2 Example Input and Output

Below are examples of valid and invalid input.

If an error occurs, print the error messages to stderr (do you remember how to use perror?). When the program exits, it should have a return code of 1. However, **this may not stop the program immediately**.

Remember: do not hardcode in program names.

2.2.1 pwd

```
user@system:/tmp/test$ /tmp/bin/pwd
/tmp/test
$ rm -rf /tmp/test
$ /tmp/bin/pwd
/tmp/bin/pwd: error: No such file or directory
```

2.2.2 wc

```
$ echo Hello World | ./wc
1 2 12 (stdin)
$ ./wc seq20.txt
20 20 51 seq20.txt
$ ./wc seq20.txt seq30.txt helloworld.txt
20 20 51 seq20.txt
30 30 81 seq30.txt
1 2 12 helloworld.txt
51 52 144 total

$ wc doesnotexist.txt
wc: doesnotexist.txt: No such file or directory
$ wc seq20.txt doesnotexist.txt
20 20 51 seq20.txt
wc: doesnotexist.txt: No such file or directory
20 20 51 total
$ wc dne1.txt dne2.txt
wc: dne1.txt: No such file or directory
wc: dne2.txt: No such file or directory
0 0 0 total
```

2.2.3 tee

```
$ cat test1.txt test2.txt
cat: test1.txt: No such file or directory
cat: test2.txt: No such file or directory
$ echo "Hello World" | ./tee test1.txt test2.txt
Hello World
$ cat test1.txt test2.txt
Hello World
Hello World
$ echo "Hello World" | ./tee
Hello World

$ echo "Hello World" | ./tee /tmp/doesnotexist/test.txt
Error: Could not open file /tmp/doesnotexist/test.txt.
```

2.2.4 head

```
$ seq 20 | ./head
1
2
3
4
5
6
7
8
9
10
$ seq 3 | ./head
1
2
3
$ ./head seq20.txt seq30.txt
==> seq20.txt <==
1
2
3
4
5
6
7
8
9
10

==> seq30.txt <==
1
2
3
4
5
6
7
8
9
10
$ ./head doesnotexist.txt
./head: cannot open 'doesnotexist.txt' for reading: No such file or directory
$ ./head doesnotexist.txt helloworld.txt
./head: cannot open 'doesnotexist.txt' for reading: No such file or directory
==> helloworld.txt <==
Hello World
$ ./head doesnotexist.txt helloworld.txt doesnotexist.txt
./head: cannot open 'doesnotexist.txt' for reading: No such file or directory
==> helloworld.txt <==
Hello World
./head: cannot open 'doesnotexist.txt' for reading: No such file or directory
```

2.2.5 tail

```
$ seq 20 | ./tail
11
12
13
14
15
16
17
18
19
20
$ seq 3 | ./tail
1
2
3
$ ./tail seq20.txt seq30.txt
==> seq20.txt <==
11
12
13
14
15
16
17
18
19
20

==> seq30.txt <==
21
22
23
24
25
26
27
28
29
30
$ ./tail doesnotexist.txt
./tail: cannot open 'doesnotexist.txt' for reading: No such file or directory
$ ./tail doesnotexist.txt helloworld.txt
./tail: cannot open 'doesnotexist.txt' for reading: No such file or directory
==> helloworld.txt <==
Hello World
$ ./tail doesnotexist.txt helloworld.txt doesnotexist.txt
./tail: cannot open 'doesnotexist.txt' for reading: No such file or directory
==> helloworld.txt <==
Hello World
./tail: cannot open 'doesnotexist.txt' for reading: No such file or directory
```

3 Part II: Make and Makefiles

A Makefile is a file that contains a set of instructions for the **make** utility. As you have seen in previous assignments, you can write bash scripts to automate compiling and running your programs. However, Makefiles have the following advantages over bash scripts:

- Makefiles remove the need to recompile all source files when only a few files have changed. This is especially important for large projects.
- Make separates the linking and compiling steps, which can save time if you only need to recompile a few files.
- Makefiles are much easier to read and write than bash scripts.
- Makefiles are more platform independent than bash scripts. Some systems may have different versions of bash or a different shell altogether.
- Makefiles are more consistent than bash scripts. They follow a standard sequence of steps, while bash scripts can get very complicated very quickly.

3.1 Overview

Your makefile should meet the following specifications:

1. The makefile should be named **Makefile**.
2. Explicitly specify the compiler as **g++**.
3. Explicitly specify the compiler flags as **-Wall -Wextra -pedantic -std=c++20 -g -O3**.
4. The object files and executables should be placed in the current directory
5. The **clean** target should remove all object files and executables.

4 Grading

4.1 Grade allocation

Assessment Component	Percentage
Design Document	15%
Post-implementation Review	15%
Autograder Unit Tests	50%
Structure and Organization	20%

The Design Documents and Post-implementation Review are to help you with your critical thinking and planning skills. In industry or research positions, you would be asked to describe your plan before starting and how it went after you finished. It is okay if you stumble or fail at certain parts; that is expected. Being able to communicate your successes and failures will help you become better.

4.2 Design Document

4.2.1 Phase 1 Description

This document describes your initial ideas and plans for finishing this assignment. You can attempt to prototype your ideas first. If you did, please add a section that describes your prototyping attempts. If not, just write your initial ideas.

You should start this process BEFORE you start programming anything. If you did not, please say you skipped this process.

It should contain, at a minimum, the following:

1. How you will approach the assignment? (see above Phase 1)
2. A rough outline of the functions, data structures, and/or comments describing what you think you will need.

Describe your approach to testing. (Optional)

4.2.2 Phase 2 Description

Things you are unsure of and will need to research to complete the assignment. You need to write at least one thing you are unsure of. For example:

1. Is this your first time writing a bash script?
2. Do you know how to handle command line arguments in Bash, C, C++?
3. Do you know how to redirect output in Bash, C, C++?
4. Or, is there any OS concept you are uncomfortable with?

4.3 Post-Implementation Review

This document describes what changed between your design document and the final implementation. What worked? What didn't work? What changed? If you had topics to research, what difficulties did you have?

5 What to submit

Your final submission should contain the following files:

1. **design.pdf** - Your design document.
2. **review.pdf** - Your post-implementation review.
3. **Makefile** - Your Makefile.
4. **head.cpp** - Your head implementation.
5. **pwd.cpp** - Your pwd implementation.
6. **tail.cpp** - Your tail implementation.
7. **tee.cpp** - Your tee implementation.
8. **wc.cpp** - Your wc implementation.

If your files are improperly named, you will fail the autograder.