# CS246E—Assignment 1 (Fall 2021)

B. Lushman

Due: Friday, September 24, 5pm

**All program code is to be written in C++14. You may include only the headers `<cstdlib>`, `<cstddef>`, `<iostream>`, `<iomanip>`, `<fstream>`, `<sstream>`, and `<string>`.**

1. Implement the Linux command `wc`. Your implementation should be able to take input from either one or more files specified on the command line, or from stdin. You are to support the flags `-c`, `-l`, and `-w`. Your output is allowed to differ from that of `wc` with respect to whitespace usage.

   **To submit:** a1q1.cc

2. A *regular expression* is an expression used to specify search patterns in text documents. In its simplest form, a regular expression is a string of text. For example, the expression `needle` indicates that you wish to search for the exact string *needle* within a text document. However, several operators are available that permit you to specify more complex patterns. For example, the expression `needle|pin` indicates a search for either the string *needle* or the string *pin*.

   Learn about the meanings of the following operators within regular expressions:

   ```
   () | * + ? . \ ^ $ [] [^]
   ```

   The tool `egrep` (equivalently, `grep -E`) takes a regular expression as its first argument, and then searches one or more files (if specified as additional arguments) or stdin for lines that **contain a match** to the pattern. Spend some time familiarizing yourself with this tool, and try out a variety of regular expressions in your searches.

   Using only the operators shown above, construct regular expressions such that the command `egrep your-pattern some-file` (with `your-pattern` and `some-file` replaced as appropriate) produces lines matching the specifications given below. Submit only the regular expression to Marmoset; our testing will supply the rest. **In particular, this means that you must assume that `egrep` is being called without options.** If your pattern contains special characters, enclose it in quotes.

   (a) Lines that contain cs246e.
   Place your answer in the file `a1q2a.txt`.

   (b) Lines that contain both cs246 (not followed by e) and cs246e.
   Place your answer in the file `a1q2b.txt`.

   (c) Lines whose length is divisible by 2, but not by 4.
   Place your answer in the file `a1q2c.txt`.

   (d) Lines that do not contain three digits (they may have fewer or more, but they may not have three). The digits do not have to be next to each other.
   Place your answer in the file `a1q2d.txt`.

(e) Lines that contain a C preprocessor include directive. You may need to do some reading, or play with the compiler, to determine what is allowed here. You do not need to worry about what characters make up valid file names or paths. We will test with only letters and dot (`.`). You may assume that all whitespace characters are spaces; we will not test your regular expression with tabs or other whitespace characters.
Place your answer in the file `a1q2e.txt`.

3. Write an implementation of the Linux tool `egrep`. For this simplified implementation, you must support all of the operators shown above (except `()`), but you may assume that the pattern string contains at most one of them, and no more than one occurrence. You may assume that the pattern is well-formed. For the `[]` operator, you do not need to handle ranges, and you do not need to handle the `[]abc]` case. You must support the command-line options `-n`, `-i`, and `-v`. Your program must take filenames specified on the command-line, and search those files, in order, for matches, and print the matching lines, just like `egrep` does (note: do **not** try to produce colour output). If no files are specified, your program must act on stdin instead. Your program must produce an exit status of 0 if at least one matching line is found, 1 if no matching lines are found.

   **To submit:** a1q3.cc

4. **Note: the following program will be useful to you in upcoming assignments. Be sure to complete it!**

   For this problem, in addition to the headers listed at the top of this assignment, you may include `<cstdio>` and `<sys/wait.h>`. Create a C++ program called `runSuite` that is invoked as follows (after compilation):

   ```
   ./runSuite suite-file program
   ```

   The argument `suite-file` is the name of a file containing a list of filename stems (more details below), and the argument `program` is the name of the program to be run.

   In summary, the `runSuite` program runs `program` on each test in the test suite (as specified by `suite-file`) and reports on any tests whose output does not match the expected output.

   The file `suite-file` contains a list of stems, from which we construct the names of files containing the input, command-line argumments, and expected output of each test. Stems will not contain spaces. For example, suppose our suite file is called `suite.txt` and contains the following entries:

   ```
   test1 test2
   reallyBigTest
   ```

   Then our test suite consists of three tests. The first one (`test1`) will use the file `test1.args` to hold its command-line arguments (if any), `test1.in` to hold its input (if any), and `test1.out` to hold its expected output. The second one (`test2`) will use the file `test2.args` to hold its command-line arguments (if any), `test2.in` to hold its input (if any), and `test2.out` to hold its expected output. The last one (`reallyBigTest`) will use the file `reallyBigTest.args` to hold its command-line arguments (if any), `reallyBigTest.in` to hold its input (if any), and `reallyBigTest.out` to hold its expected output.

   A sample run of `runSuite` would be as follows:

   ```
   ./runSuite suite.txt ./myprogram
   ```

   The program will then run `myprogram` three times, once for each test specified in `suite.txt`:

- The first time, it will run `myprogram` with arguments from `test1.args` (if this file exists) and standard input redirected to come from `test1.in` (if this file exists). The results, captured from standard output, will be compared with `test1.out`.
- The second time, it will run `myprogram` with arguments from `test2.args`, (if this file exists) standard input redirected to come from `test2.in` (if this file exists). The results, captured from standard output, will be compared with `test2.out`.
- The third time will be like the first two, but using the `reallyBigTest` file stem.

At least one of `x.args` and `x.in` should exist for each test `x`. The file `x.out` should always exist (even if it is empty).

If the output of a given test case differs from the expected output, print the following to standard output (assuming test `test2` failed):

```
Test failed: test2
Args:
(contents of test2.args, if it exists)
Input:
(contents of test2.in, if it exists)
Expected:
(contents of test2.out)
Actual:
(contents of the actual program output)
```

with the `(contents ...)` lines replaced with actual file contents, as described. The literal output `Args:` and `Input:` should appear, even if the corresponding files do not exist. **Follow these output specifications *very carefully*. You will lose a lot of marks if your output does not match them.** If you need to create temporary files, create them in `/tmp`, and use the `mktemp` command to prevent name duplications. Also be sure to delete any temporary files you create in `/tmp`.

Your program must also check for the following error conditions:

- incorrect number of command line arguments
- missing or unreadable `.out` files (for example, the suite file contains an entry `xxx`, but `xxx.out` doesn't exist or is unreadable).

If such an error condition arises, print an informative error message to standard error and abort the program with a nonzero exit status.

You are allowed to check for additional error conditions, but you will only be graded on these.

To solve this problem, your C++ program will need to interface with the Linux shell. You will issue shell commands from within your C++ program, and capture results and exit statuses. You will learn the necessary pieces in Tutorial 1.

**To submit:** a1q4.cc