CSE306
Software Quality in Practice

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

End-of-Semester overview

M	T	W	Th
4/15 Callgrind	4/16 LEX22	4/17 TODAY Expectations: LEX23/24 LPR POST	4/18 LEX23 LPR practice part 1
4/22 Process review (interactive) TBD suggestions	4/23 LEX24 LPR practice part 2	4/24 TBD	4/25 LPR part 1 of 2
4/29 LPR Q&A	4/30 LEX make-up with prior approval only	5/01 Tools in other languages (interactive)	5/02 LPR part 2 of 2
5/06 no class	5/07 LEX make-up with prior approval only	5/08	5/09

Important instructions for the Lab Practical exam (LPR)

Timeframe

You may work on this part of the lab practical exam only for two hours and only during your scheduled lab time.

Please note: if you do not finish all the coding during part 1 (a.k.a LPR1), don't worry - you can finish up during part 2 (a.k.a. LPR2), though you may not modify your code/repo between LPR1 and LPR2. We give some implementation hints (see below). Remember to show proper use of tools and techniques.

Please note: In LPR2 you will receive buggy code so you can use gdb to track down a segfault and valgrind/memcheck to document and fix memory leaks in case you did not find opportunities to showcase that in LPR1.

Resources

You may use any prior work *you* have done for this course (any earlier LEXes, PRE, EXP01, EXP02, POST), any tool documentation, etc. You may ask TAs (but not other students) questions during the lab practical, though they cannot answer all questions: this is an exam after all. TAs will be available during your regular lab time. Between part 1 and part 2 you should may ask questions on Piazza (though (again) we cannot answer all questions: this is an exam after all).

In preparation for the last few assignments (LEX23 & LEX24 (which are practice for the LPR), the LPR itself, and POST) it is important that everyone is clear on what sort of evidence we are looking when assessing your work.

These are the things for which we are expecting to find evidence of use:

1. git (on GitHub)

- For full credit we expect regular commits, appropriate branching (e.g. feature branches, bug fix branches), and meaningful/descriptive commit comments.

- For full credit all branches must be pushed (use -u on first push of branch only): git push -u origin HEAD

- For full credit the commits must be of sufficient granularity to show evidence of following sound development/debugging practices (as discussed throughout the course).

- For full credit the commit messages must be descriptive enough to show evidence of following sound development/ debugging practices (as discussed throughout the course).

2. Planning tool (Trello) - use for planning rather than collaboration in LPR since this is an individual activity

- For full credit we expect that meaningful (GitHub) issues are created.

- For full credit we expect that Trello cards are linked to GitHub issues.

3. Build tools (make)

- The makefile itself will be used as evidence. The makefile must be functional for compiling the project code. If you use your 'makeMake' script the script itself must be committed to the repo as well.

- The 'makeMake' script may not create targets for all the tools you wish to use (e.g. 'gprof' and the 'valgrind' tools. You can either add this into your script or hand-edit the resulting makefile.

4. TDD/opaque testing (Criterion)

- We expect there to be a git branch for writing the tests, committed and pushed.

- We expect there to be a sequence of commits showing the opaque tests being written (including the stubbed out application code to allow the tests to compile/run).

- We expect there to be be a commit that includes the output from Criterion (e.g. in a text file) that shows the tests (mostly) failing due to functionality not being built yet.

- We expect there to be git branches for feature development with commits that show opaque tests failing prior to feature implementation, and opaque tests passing after feature implementation.

5. Software testing - transparent testing (Criterion); after a feature is developed and passes its opaque tests:

- We expect there to be commits on a transparent testing branch that show the output of a coverage tool (section 6, 'gcov') - e.g. the 'gcov' output committed to a text file.

- If coverage is not 100% additional tests must be written to achieve (close to) 100% coverage. Test development must be shown in a separate commit after the initial 'gcov' run.

- If the opaque tests you develop naturally have 100% coverage, try to engineer a feature development cycle for which the opaque tests won't have 100% coverage, so you can show (artificially) the process of improving the test coverage.

- We expect there to be a commit after the transparent tests are written that show improved code coverage - e.g. the 'gcov' output committed to a text file.

6. Code Coverage ('gcov')

- This is tied to section 5. Show coverage tool output before and after writing transparent tests, in separate commits, as coverage increases.

7. Compiler

- Provide evidence that the appropriate flags were passed to the compiler (the makefile can provide evidence).

- Provide evidence that compiler errors and compiler warnings were addressed on an ongoing basis (show compiler output in a text file, commit and push file, resolve issues, show compiler output in a text file, commit and push file again.)

- You should aim for clean compiles, with no errors or warnings using the -Wall flag.

8. Performance tools (gprof, valgrind/callgrind)

- Capture output from one or both of these profiling tools in a text file. Explain output (indicate where hotspots are) - this can be done in a commit message or the text file. If possible show code changes and then performance improvement. If this doesn't happen naturally you can engineer in poor performance and then show improvement by fixing.

9. Debugging ('gdb')

- We expect you to show adherence to process and productive use of gdb.

- When there's a bug, create a bug fix branch.

- Write tests that fail due to the bug (commit tests, show output of running tests in commit).

- Run 'gdb' to track down source of bug. Explain what you think the problem is, how you're going to investigate it, gather data from 'gdb' to support or refute your hypothesis, preserve data to file, commit to repo.

- Fix bug - show tests passing. Commit evidence.

10. Memory leaks (valgrind/memcheck)

- We expect you to verify presence/absence of memory problems with the 'memcheck' tool (from the 'valgrind' suite). Preserve output to file. Commit.

- We expect you to address memory issues, and show improvement (preserve 'memcheck' output to file. Commit.)

- If no memory issues are present you can (as in other cases) engineer a problem to demonstrate that you can use the tool to detect the problem, fix the problem, and demonstrate that it is resolved.

We recommend you look over the rubric in UBLearns prior to starting work.

Keep in mind too that you have limited time to do this. Be strategic: have a game plan going in, making sure you hit each of these items. You are not expected to complete all the requirements, or necessarily fix all bugs, or plug all memory leaks.

Use the code you write to demonstrate that you know when/how to use the tools effectively, while demonstrating adherence to the sound development/debugging processes we discussed in class.