Lab FAQs

General

1. **How will implementation soundness and style be graded?**
   Here are some guidelines:

   - Complex pieces of code should be documented in comments so that they become easier for the reader to understand—but don't overdo it with comments for trivial things.
   - Common, general-purpose functionality should be decomposed into helper functions to avoid repeated code—don't overdo this either, e.g. by making it difficult to understand the flow of execution.
   - Code should use consistent naming conventions and not contain commented-out code or vestiges of debugging.
   - Please check the return values of all functions in which an error could occur, and handle these errors gracefully. Throwing an exception is one graceful way to handle an unrecoverable error.
   - Avoid memory errors and leaks, egregious performance issues (e.g. allocating large amounts of memory for no reason) or unnecessarily complex approaches (e.g. a complicated data structure that is only marginally more efficient than something much simpler, or something available in the C++ standard library).
   - Before submitting each assignment, please run `cmake --build build --target format` (to standardize the formatting), and make sure to remove any dummy code or comments that were included with the starter code.
   - Also (optionally) run `cmake --build build --target tidy` to receive suggestions for improvements related to C++ programming practices.

Your style grade may also include components of functionality not covered by the tests.

2. **How do I run an individual test?**

   ```
   ctest -R '^test_name$'
   ```

3. **Can I add include lines in the header files?**

   Sure.

4. **How do I use `git`?**

   Here are some links to resources to learn Git (https://try.github.io/).

   Brief summary: Git commits store a snapshot of the current state of your code. For instance, say you break a feature that you know used to work, but aren't sure which code caused it to break. If you made a commit after getting that feature working, you can checkout that commit to see what things looked like in the good old days when it worked. So, make a commit every time you get something working! To make a commit:
   First, git add the files you've changed - you must add before every commit. e.g. `git add ./src/byte_stream.*`
   Then, commit with a message describing the state of the code. e.g. `git commit -m "All tests pass except many_writes times out."`
   You can see which files you modified by running `git status`. To really understand the state of a Git repository, the command-line `tig` tool, or the graphical `gitk` tool, can be very helpful.
   A handy shortcut to commit all modified files is `git commit -am "[message]"`.

   Reality check: If you want to verify that you have submitted exactly the files you intended to, you may find the following useful.
   First, once you've committed, run `git status`. You should see a message saying `nothing to commit, working tree clean`. If you have uncommitted changes, you'll see this instead: `Changes not staged for commit:`.
   Second, after you've copied your git bundle to rice, and logged into rice:

   1. Run `$ git clone [bundle name] [unbundled name]`.

2. `cd` into the unbundled directory.
3. Run `$ git log` and verify that you see the commits you expect to.
4. Copy the commit hash of the starter code commit.
5. Run `$ git diff [hash]` to make sure they see all changes you want to submit.

5. **How do I debug?**

`gdb` is a great tool for debugging the labs! Check out the CS107 guide (https://web.stanford.edu/class/archive/cs/cs107/cs107.1202/resources/gdb) if you need a refresher.

To use `gdb` to debug a test:

1. Install `gdb` (on the VM, `sudo apt-get install gdb`).
2. Start `gdb` on the executable corresponding to the test you want to debug (from the `build` directory, `ls tests` to see the executables). e.g. from the build directory: `gdb tests/byte_stream_one_write`.
3. The output of the test failure will show you the part of the test that failed. To set a breakpoint at that part of the test, break on the line in the test file where the test harness for that part is created. The test source code is in `minnow/tests` (not `minnow/build/tests`). e.g. if you're failing the `write-pop2-end` test in `byte_stream_one_write`, `break 83` (i.e. where the `write-pop2-end` test harness is created).
4. You can set breakpoints on your functions using the function names, as usual.

Other notes:

- We don't recommend modifying any files in the `util` directory, since messing up the support library will make debugging difficult.
- If a test is timing out, but you want to check if it passes without the timeout, run the test executable individually, which won't enforce the timeout. e.g. from the build directory: `./tests/byte_stream_one_write`.
- `gdb` may help debug timeouts. While running the test in `gdb`, if it appears to hang (meaning it may be executing a slow portion of code), ctrl-C and backtrace to pause and see which code was executing.

Lab 0

1. **What should the behaviour of my program be if the caller tries to pop with a `len` greater than what is available?**
   We don't have any preference on the behavior if the caller tries to pop more than is buffered in the stream, as long as you behave reasonably and don't crash in that situation. If you want to pop the maximum available, that is fine with us. If you want to throw an exception, that is also fine with us.
2. **How fast should my code be?**
   Here's a reasonable rule of thumb: if any of your tests are taking longer than 2 seconds, it's probably good to keep exploring other ways of representing the ByteStream. And if your benchmark results at the end of the test are worse than 0.1 Gbit/s (100 million bits per second), that's another sign the design may need to be rethought. This lab doesn't require any sophisticated data structures, but it might still take some trial-and-error to find a reasonable approach. Sometimes "big-O" analysis can be misleading about the performance of real programs—the constant factors can matter a lot!