CS 130 Hub

Assignment 1

In this assignment you will write tests and fix bugs in an nginx-format configuration file parser.

Each student should submit this assignment by 11:59PM on April 8, 2025 into the submission form.

Note: `${USER}` should be your user name @g.ucla.edu

TABLE OF CONTENTS

Environment Setup

The assignment instructions assume you are running in our Development Environment, which is a Docker container based on the latest LTS version of Ubuntu, 24.04 (Noble Numbat), with several tools and libraries pre-installed. While the development environment can be run on Linux or MacOS (see the instructions in the Guides) **we highly recommend Cloud Shell**.

To start, first choose or create a directory you will use to store all your projects for this class. Let's assume this is `cs130`. Everyone should download the CS130 tools into your project directory:

```
$ mkdir cs130
$ cd cs130
$ git clone https://code.cs130.org/tools
```

Then, install Docker (if not already installed) and start the development environment. Remember to replace `${USER}` with your UCLA username (without @g.ucla.edu):

```
$ tools/env/start.sh -u ${USER}
```

Note, if you're using an IDE like Visual Studio Code, you may want to enable IDE support by adding the `-r` flag.

Set up a git repository

You will set up your first git repository in the cs130.org instance of Gerrit (code.cs130.org) for this assignment. Make sure you've set up SSH Access, then create an individual repository for yourself named `${USER}-config-parser`.

Download and submit initial skeleton code

We've provided some skeleton code to start you off. Clone your new repository locally, create a branch, and then unpack the skeleton code, which will create `src/`, `include/` and `tests/` directories.

```
$ git clone ssh://${USER}@code.cs130.org:29418/${USER}-config-parser
$ cd ${USER}-config-parser
$ git review -s
$ git checkout -b skeleton
$ curl http://static.cs130.org/src/config-parser.tar.gz | tar -zxv
```

Next, you should run the project templates script to add some useful configuration files to your project, as detailed here:

```
$ ../tools/templates/init.sh
```

This code will not build without some changes, but before fixing that you should practice submitting just what you've downloaded and generated above, without any other changes. First

add all the files and see the list of everything you will be committing:

```
$ git add .
$ git status
```

We've used `.gitignore` to omit some files that should *not* be checked into source control. Take a look at the contents of `.gitignore` so you know:

```
$ cat .gitignore
```

If you're comfortable with the list of (hopefully 12) files to commit, then commit them to your local git repository:

```
$ git commit -m "Adding initial skeleton code"
```

Now that your change is committed locally, you'll want to push them to the remote repository. If you try to push your code back to the remote `main` branch with `git push`, you'll notice that Gerrit refuses the push. This is intentional! All your code must be reviewed, so everything will be submitted through the Gerrit server. Your repository has been set up to only allow pushes through the special Gerrit review branches. You *could* create a change by running some obscure command like `git push origin HEAD:refs/for/main`, but you'll probably want to just use `git review` instead (as installed in the development environment).

Create a review for your initial commit, and use `-f` to delete the skeleton branch since we're done with it:

```
$ git review -f
```

For this first change, you can act as your own reviewer. In the future you should avoid this, since you won't get necessary feedback on how to improve your code.

Navigate to the review page, either from the URL shown by `git review`, or through the Gerrit dashboard. Load the change, take a look around, and when you're done click *Code-Review+1* to give it +1 score. You can then click *Submit* to have the code submitted to the `main` branch.

You can verify that the code was submitted after pulling the latest version from the server:

```
$ git pull
$ git log -2
```

Get this code to build!

The `init.sh` script created a base CMake configuration for you in `CMakeLists.txt`. This will specify which source files define libraries and binaries, and all the dependencies between them. The initial version references non-existant example files, so you will need to update it to successfully build your project. Start by creating a new branch for the work in this section:

```
$ git checkout -b fix_build
```

Now you should edit `CMakeLists.txt` to refer to actual source files, following the `# TODO(!)` instructions. Make sure to uncomment (remove `#`) any targets you want to create, and remove any unnecessary targets. By the end you should address or reassign all `# TODO(!)` lines and remove those comments. Some hints:

- Learn what `add_library`, `add_executable`, and `target_link_libraries` do, and anything else you see in there that's relevant.
- Remove the references to Boost since the config parser shouldn't need Boost (we'll use Boost in the next assignment).
- Remove the test coverage targets for now, but those will be useful in future assignments.
- You will want two executables, one for your `config_parser` and one for your tests.
- You can either compile all your non-test sources in a single `add_executable` rule with multiple source files, or create a library with your `config_parser.cc` code and link it to your executable that includes `config_parser_main.cc`. Experiment with both, think about the pros and cons of each, and choose one. Either solution is fine, but they do have distinct pros and cons.

Once you've correctly updated `CMakeLists.txt`, you're ready to run `cmake` to generate `Makefile`s, and build the code. Note, do not run `cmake` from your repository root directory. Instead, create a `build` directory and perform an out-of-source build (What does that mean?).

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Run the existing tests

If everything builds correctly, you're ready to run the tests. You can run tests via `make` or `cmake`, or you can run them directly from the command line to see slightly different formatted output. Try both! Note that this `config_parser_test` looks for an example config in its current running directory, so it has to be run from the directory where the example config lives (`tests/`).

```
$ make test
$ cd ../tests && ../build/bin/config_parser_test && cd ..
```

Push your code to Gerrit

Assuming the tests pass, you're ready to submit your fixes. First re-examine your changes so you know what you're submitting:

```
$ git status
$ git diff
```

The commit your changes locally and submit for review:

```
$ git add .
$ git status
$ git commit -m "Fixed broken skeleton code"
$ git review -f
```

You can *also* self-approve this one and submit it. But this is the last time!

Write unit tests

Next you'll expand on the simple single unit test with new tests. But before getting too deep in writing new tests, take a look at the basic structure of the configurations here and here so you are familiar with the syntax.

Once you're familiar with the config format, you should write more unit tests for the config parser, especially the complex logic in the giant `Parse` method. There is already one very simple test case in `config_parser_test.cc`, and that's where you'll be adding additional test cases. Remember to create a new branch (`unit_tests` perhaps?) before editing `config_parser_test.cc`.

Take a look at the API of `config_parser.cc`'s large `Parse` method. Think about what possible inputs it could reasonably take, and what outputs you would expect it to generate for these inputs. You should write test cases for each of these, passing various input data to `Parse`, and verifying that the result of the function is what you expect in each case. Eventually, you should find at least one bug in the parsing code that will cause your test to fail. When this happens, you can then fix the bug in the parsing code until your test passes. Note, you probably shouldn't modify the config parser until you identify bugs, otherwise you may end up introducing new bugs.

In order to make the test case code less repetitive, use a test fixture. In gtest, this is a class that holds functionality, data, or setup/teardown code applicable to multiple test cases. For a suite of test cases about a parser, it would probably make sense to create an instance of a parser and possibly do some setup in the fixture, and give each test case access to the fixture's parser instance (or the name of a file to parse).

Each time you fix a bug (or maybe a number of closely related bugs), you should send your code out for review in Gerrit. Read through the following section to familiarize yourself with the process. Make sure you give your reviewer(s) access to your repository so you can assign them as reviewers. You can assign them as reviewers when running `git review` for new changes, or from the web UI for existing changes.

Engage in code reviews

As hinted above, you will be responsible for reviewing someone's code in this assignment. We will go in-depth on code review practices later in this class, so for now you will engage in a very basic review.

For this assignment we'll be reviewing in pairs of 2, so you should pair up with another student to review each other's code. Use this spreadsheet to find or record your partner.

To give access to your reviewer to your repository, go to your repository admin page (e.g. `https://code.cs130.org/admin/repos/joebruin-config-parser,access`), click **Edit**, Add permission *Read* for `Reference: refs/*`, and add your reviewer in the *Add group* field. Finally, click **Save for review** and you should see a change on Gerrit called "Review access change." Verify the diff in the change looks good, click **Code-Review+1**, and then **Submit** the access change.

You should receive an e-mail notification from *noreply@cs130.org* when someone sends you a review. Open the link in the e-mail, or go to https://code.cs130.org/ and find the review on your dashboard. Once you have the review open, look at the listed files to see the proposed changes in the current patch set. (A patch set is a snapshot of a group of files.) If you see any errors you'd like to comment on, do so by clicking the line number in the right pane of the file change view and

entering your comment. Another good comment would be to ask for more documentation/code comments for a line/section of code. Send your comments by navigating to the main change page, and clicking **Reply**.

When replying you have the chance to score the change. A **+1** score is required before the code can be submitted. If you're satisfied with the code, comment somewhere that they've done a good job (perhaps point out a good test case), and give them a **+1** score. If not, leave the score at **0** and wait until issues are resolved with updated patch sets before giving a **+1** score. If for some reason you really want to object to a change, you can give a score of **-1**.

You can iterate multiple times on a review, and may receive replies to your comments from the code author. Remember to reply to replies quickly! Follow the golden rule: respond to reviews as promptly as you would want someone else to respond to your reviews (which is, *quickly*).

As a code author, to reply to a review comment and make code updates you will have to make changes on your computer and upload a new patch set to Gerrit. See how to respond to reviews for more instructions.

You should use this assignment to practice with `git review` and the Gerrit site. As such, we expect you to submit **at least 3 changes** for review (not including your initial change that you reviewed yourself), and to have **multiple patch sets** (i.e. respond to a comment with new changes) on **at least 1 review**. Ordinarily you might want to send reviewers perfectly working code (get over that feeling, because such a thing does not exist), but this exercise works better if you purposefully include a mistake or two for your reviewer to catch (keep them paying attention).

If you find yourself short on the requirements above, add a temporary file or code comment, then delete it, creating two separate changes. That should give you at least 3 changes.

Of course, remember to submit your code back to the `main` branch by clicking the **Submit** button on your changes in the Gerrit UI.

Grading criteria

To meet the minimum for this assignment you must:

- Have a correctly named repository:
  - with all files checked into the `main` branch on Gerrit
  - and correct targets in `CMakeLists.txt`.
- Resolve, remove, or reassign TODOs.
- Create a text fixture and use it to discover and fix at least 1 bug.

- Produce and review at least 3 different changes
- Iterate with feedback on at least one change
- Provide feedback for someone else

Additional criteria may be considered at the discretion of graders or instructors.

Submit your assignment

Submit your final code on Gerrit, then use the submission form to turn it in. We will only review code that was added to the `main` branch of the repository before the last commit referenced in the submission form. Note you may update your submission by re-submitting the form.

"You may think using Google's great, but I still think it's terrible." —Larry Page

Page last modified: April 3, 2025.