Exercises for the Lecture Series

"Object-Oriented Programming for Scientific Computing"

Dr. Ole Klein
ole.klein@iwr.uni-heidelberg.de

To be handed in on 09. 05. 2017 before the lecture

EXERCISE 1 GIT SURVIVAL SKILLS

This exercise serves as an introduction to Git and GitLab. Please perform the following tasks in a *private* repository, and put only your observations and answers into the official repository of your group. Below the lists of tasks is a list of Git commands that might be useful.

Perform the following tasks, and document which Git command you have used, briefly describing its purpose:

- Create a new folder on your computer, and *initialize* a new Git repository inside.
- Put a text file into the repository (its contents don't matter, you could take one of the source files of the lecture), *add* it to the files tracked by Git, and *commit* your changes to the repository, using a meaningful commit message.
- Check that your commit was successful by reading the *log* file, and take a look at the *status* of your working directory.
- Change something in the text file, *add* and *commit* those changes, assume you had made a mistake, and *revert* your commit (note that `git revert` is not the right command in this situation).
- Edit the file a second time, but *stash* your changes away to keep them save for using them at some later point in time.

Create a GitLab account if you have not done so, find the "Projects" tab, and press the "New Project" button. Enter a project name and set visibility to *private*. Follow the instructions under "Git global setup" and "Existing Git repository" to get your private project repository on GitLab. Note that there are also options for completely new projects or existing (non-Git) folders.

- Use the URL on the project page to create *another copy* of your repository on your computer (you can delete it afterwards).
- Browse your repository content under the "Files" tab, and perfom a quick edit to your text file, *commiting* the change directly from GitLab.
- Change *a different* line on your computer. *Pulling* the newest version from GitLab should silently incorporate both your changes.
- Edit the line you changed locally also on GitLab, but in a different way. This time, *pulling* from GitLab should cause a *merge conflict*. Check the *difference* to your last clean version, and try to *resolve* the conflict by either accepting one of the versions ("theirs"/"ours") or using an interactive *merge tool*. Resolving a conflict includes *commiting* the merged files and explaining your reasoning and choices.

- Reapply the change you had *stashed* away, and check that the *difference* to your previous repository state is what you would expect.

This exercise has left out several important Git concepts, especially *branches* and *tags*, but should suffice as a starting point for those who have not used Git before. There are also several tabs in GitLab that might be useful, e.g. the "Commits" and "Graph" tabs under "Repository", the "Issues" and "Todos" tabs, and the "Wiki" tab. If and how you use these features is up to you.

List of Git commands:

`git add <file/folder>`, `git checkout --ours/--theirs <file>`, `git clone <url>`, `git commit`, `git init`, `git log`, `git mergetool`, `git pull`, `git push`, `git reset HEAD~`, `git stash`, `git status`, `git stash pop`

10 Points

EXERCISE 2 BASIC DEBUGGING

You can find three C++ files for download on the lecture website, namely `matrix_broken.h`, `matrix_broken.cc` and `testmatrix.cc`. These are modified versions of the files presented in the lecture, containing several bugs. Of course you could simply diff the files with the originals, but that would defeat the purpose of this exercise. Instead, you should try to systematically find those bugs using GDB.

Download the three files and compile them with debug information:

```
g++ -std=c++11 -Og -g -o testmatrix matrix_broken.cc testmatrix.cc
```

You can now start your program with GDB in TUI mode:

```
gdb -tui ./testmatrix
```

Entering `layout split` at the prompt (or `la sp`, almost all commands can be abbreviated) displays the assembly code equivalent of the program, and `layout src` (or `la sr`) removes the assembly window if you don't need / want it.

The most important GDB commands, with their abbreviation and possible arguments, are probably **`break`** `[b] <file:line, file:function>` (enable breakpoint at specified location, file may be omitted), `backtrace [bt]` (show hierarchy of called functions), **`continue`** `[c]` (continue running after break), `next [n]` (execute marked line), `print [p] <expression>` (print content of variable / object), `step [s]` (enter first function on marked line), `run [r] <arguments>` (start program with given arguments, if any), and `watch <expression>` (break if value of expression changes).

These commands are sufficient for this exercise, but you can find additional information at `https://beej.us/guide/bggdb/#qref` or any other GDB reference card on the internet. Note that `print` can be used to access members of objects, e.g. `p a.b` or `p a->b`, no need to step into some method for that — even if the member `b` is **`private`**.

Use GDB to find and correct the bugs in the provided source code, and document which bugs you found and how. There is a bug that is not covered by the tests, search for it (with a diff, if necessary). How would a test for this bug look like? What is problematic about the specific choice of test matrices in `testmatrix.cc`, what kind of bug are they unable to detect?

Note: the file `testmatrix.cc` *does not contain bugs, and neither does the most complicated of the methods,* `Matrix::solve`.

10 Points