Department of Informatics, Technical University of Munich
Scientific Computing in Computer Science
G. Chourdakis, J.-M. Gallard, F. Simonis

Winter term 2019/20

Advanced Programming Tutorial
Worksheet 1: Calculator

Keywords: data types & casting, assignments, functions, branches, version control

Version Control with Git

Remember the last time you worked on a project, of which the submitted version was called `Project_FINAL2-new-thisTimeForReal.cpp`? There is an easier way to track versions and revert to previous states at any time! This is especially important when collaborating on a larger project.

In the Git version control system, every state is called a "commit", which describes the changes from the previous version. Every file in the project, together with the history, is stored in a "repository" (a directory). You can manage Git repositories either from your IDE, or from the command line (here examples in Linux).

To start a repository in the current directory, use `git init`. To clone a remote repository, use `git clone <URL>` (ask us for any issues). Then, the usual development cycle is:

1. Check that you don't have any changed files: `git status`.
2. Get the latest changes from the remote repository: `git pull` or `git pull origin master`.
3. Work on your project until you have a small, self-contained improvement (that compiles).
4. See your changed files: `git status` or `git diff`.
5. Collect the files you want to save with `git add file1.cpp file2.cpp`.
6. Create a new saved state: `git commit -m "Add a short and descriptive message"`. The same `git commit` with a flag will start your editor, with which you can write a longer message body in a second paragraph. It is common in this to also refer to "issues".
7. Publish the changes to the remote repository: `git push` or `git push origin master`. If, in the meantime, someone else has published changes, your "push" may be rejected. In that case, a "rebase" usually helps: `git pull --rebase` and then `git push`. This is a potentially dangerous operation (as it rewrites the history), so read first about it.

When working with other people, it makes a lot of sense to work on different "branches", which start from a common base, but try to implement different features of a project, often in the same files. The main branch has usually the name "master" and one can see the name of the current branch using `git branch`. To create a new branch, use `git checkout -b my-branch-name`. To change to an existing branch (or any other previous state), use `git checkout my-branch-name`.

To merge another branch into your branch, use `git merge other-branch-name`. If both branches changed the same line of a file, you will get a "merge conflict". Open the file in your editor, adapt the affected line and then use the usual add-commit procedure.

Exercise 1: Simple Calculator

In this exercise, we want to implement a very simple calculator. However, you are not going to do it alone! Talk to your neighbor and decide who is going to be "A" and who "B". It would be simpler to use the command line interface of Git in this exercise.

Student A

1. Create a project e.g. on `https://gitlab.lrz.de` and clone the repository.
2. Create a file `simpleCalculator.cpp`, add the main function, and publish your changes.
3. Invite "B" as a developer to your project.
4. In a branch `sum-subtract`, implement the following functions for `a` and `b` of type `double`:
   - `sum(a,b)`, which returns $a + b$
   - `subtract(a,b)`, which returns $a - b$

   All functions should return their results as doubles.
5. Synchronize with "B" and explain your implementations to each other. Merge both branches to "master" and push (only "A" does this now).
6. In a branch named `mean`, implement an additional function, `mean(a,b)`, calculating the average value of two `double` arguments. Reuse the previously defined functions.
7. In the branch `multiply-divide`, add another variant of `divide(a,b)`, which divides two `int` arguments. Use type casting and return a `double`. Can you use the same name?

Student B

1. Clone the repository that "A" invited you to.
2. In a branch named `multiply-divide`, implement the following functions for `a` and `b` of type `double`:
   - `multiply(a,b)`, which returns $a * b$.
   - `divide(a,b)`, which checks if b is non-zero and returns a/b.
     Hint: `if (condition) {...} [else if (condition) {...}] else {...}`.

   All functions should return their results as doubles.
3. Synchronize with "A" and explain your implementations to each other. Pull the changes.
4. In a branch named `menu`, implement a menu in which the user is prompted to specify an operation $(+, -, /, *, m)$ and to give 2 values. Print the result and finish the run. Use an `if(){...}` structure to call the respective function.
5. This time, take care of merging the two branches into `master` (only "B").

Discussion

Did you check for $b \neq 0$ in the main function or in the `divide(a,b)`? Where could an `assert(condition)` be useful?

Exercise 2: A more sophisticated Calculator

Extend your simple calculator from the previous exercise with methods to support:

1. the square operator (x^2).
2. the trigonometric operations ($sin(\pi x), cos(\pi x), tan(\pi x)$).

Make sure to also add these in your menu and check them for correctness.

Discussion

- Which library files do you need to include in `simpleCalculator.cpp`?
- How did you define / get the value of π?
- What would you have to do if you wanted to implement the main function in another file?

Idea: Base Conversion

Extend your Calculator with an operation `base(a, b)` to convert an integer (or other type) `a` from the decimal system to another base, `b` and print the result (no return value).

Note: Integer division returns *truncated* integers, which is useful here. You may also use the *modulo* operator `%`.

Homework 1: Your first game in C++

This is a project that will spread over the whole semester and it gives you the opportunity to work on a more complex code. This project is optional but highly recommended as it will apply many concepts taugh in this lecture.

In some of the next worksheets (approximately every two weeks), we will give you directions for the next steps, using the latest features we discussed. You will need to refactor your code multiple times. No precise instruction will be given. Instead, you will be given a lists of "User Stories" describing what a user should be able to do, which may come with hints on how to implement the respective features. On Moodle you can find a file `README.md` describing a reference code. Your implementation should approximately match it (you can of course do more). Creativity is encouraged!

You can develop this alone or with a partner and you can showcase your game at the end of the semester, but this optional homework is not graded. If you have any difficulties during the project, you can ask on the Moodle forum.

For the first version of your game:

1. Pick a name and create a project on the LRZ GitLab.
2. Familiarize yourself with the "User Stories"
3. Start implementing