Concepts of C++ Programming (Exercises)

winter semester 2023

CIIP Team: David Frank, Jonas Jelten

Computational Imaging and Inverse Problems (CIIP)
Technical University of Munich

Contents

Who?

- Lecture:
  - PD Dr. Tobias Lasser
    - Computational Imaging and Inverse Problems (CIIP)
- Exercises:
  - First half: Jonas Jelten
  - Second half: David Frank

What?

Aim of the exercises:

- study the concepts of modern C++ programming
- learn to apply those concepts in elegant and efficient solutions

Target audience:

- Bachelor students of Informatics (3rd term onwards)
- Master students of Informatics, BMC, CSE, Robotics
- anyone who is interested!

When and where?

Dates, places, and links

- Tuesday, 14:15 - 15:45, Interims II Hörsaal 2 (003)
  - Live Stream, recording: `https://live.rbg.tum.de/`
- Thursday, 17:00 - 18:30, Galileo Hörsaal (8120.EG.001) (you are here)
  - Video conference: `https://bbb.rbg.tum.de/tob-w3o-4un-gp6` (no recording!)
  - Tweedback: `https://tweedback.de/`

Generally:

- Tuesday: lecture
- Thursday: exercises

Tweedback Session-ID today: zy98

- URL: `https://tweedback.de/zy98`

Online resources

Chat platform

`https://zulip.cit.tum.de`, Streams #CPP and #CPP Homeworks

- discussions, answering questions
- automatic announcements of GitLab changes in the *Tasks* repository

Moodle course

`https://www.moodle.tum.de/course/view.php?id=90655`

- news and announcements
- materials (slides, exercise sheets etc.)

Website

`https://ciip.cit.tum.de/teaching/cpp_ws23.html`

Homework assignments: brief details

- One homework assignment every Thursday via Moodle
- Deadline for automated tests to pass: +1 week (Thursday 17:00)
- Your solutions have to be uploaded to a GitLab repository
- If there are any issues: use our chat at `https://zulip.cit.tum.de`, stream #CPP Homeworks
- How to get access to your GitLab repository:
  - log in to `https://gitlab.lrz.de`
  - edit your name (Profile, Account, Username) if you wish
  - in Moodle: go to "Add your GitLab username" activity
    - add your GitLab profile username (without @ or *https://gitlab.lrz.de/*) (check `https://gitlab.lrz.de/-/profile/account` if unsure)
  - request access for the Waiting Room at `https://gitlab.lrz.de/cppcourse/ws2023/waiting-room`
  - your personal repository will be setup automatically

Exercises

Exercise format

- Homework:
  - weekly programming assignments
  - exercises are checked automatically
  - submission possible until 1 week after the session (until Thursday 17:00)
- Exercise Session:
  - discuss the previous week's assignments
  - reinforce topics from the lecture
  - small in-class programming exercises
  - give some pointers to the current exercise sheet
  - allow you to work on & ask about the current exercise sheet

- all materials will be available in Moodle
- the level of interaction is up to you!

Homework Points

Homework $\neq$ Bonus

Homework points are **part** of the exam **not a bonus**.

- Doing the homework makes it easier to pass the exam
- You can pass the exam without doing the homework

Please note

The exercises are quite new.

→ any feedback from you is welcome!

Contents

Required tools

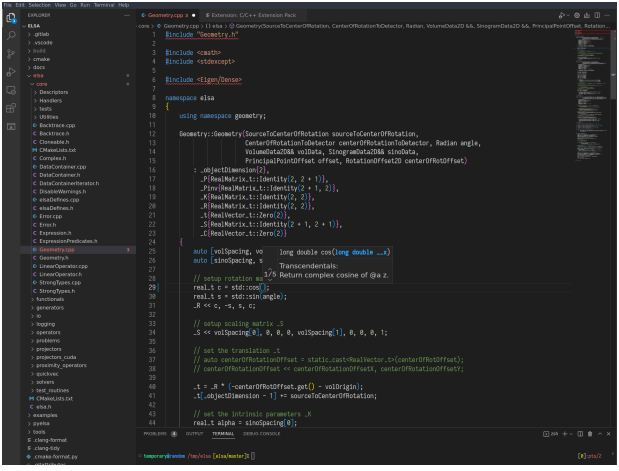
"Programming is learned by writing programs."

–Brian Kernighan

- Start by getting familiar with important tools, such as
  - your operating system (Linux, ...) + text editor
  - a C++ compiler (typically gcc and clang)
  - a text editor (IDE)
  - git for source code management
  - CMake for build automation and dependency management (next week)
  - a debugger (such as lldb or gdb) for debugging (next week)

IDEs

- Why an IDE?
  - syntax coloring
  - auto completion
  - static code analysis
  - facilitates debugging
  - interfaces with other tools



IDE choices

- All have pros/cons, just choose by preference
- Free open-source software:
  - Emacs ($\leftarrow$ editor for a lifetime)
    - Doom + lsp + clangd + projectile + magit + ...
  - Vim
    - neovim + YouCompleteMe + clangd
  - Eclipse
  - QtCreator
  - Code::Blocks
  - Visual Studio Code (currently most popular)
- Proprietary:
  - Sublime
  - Visual Studio
  - CLion

Contents

Survey

- How familiar are you with git? → Tweedback
- How familiar are you with GitLab? → Tweedback

Git

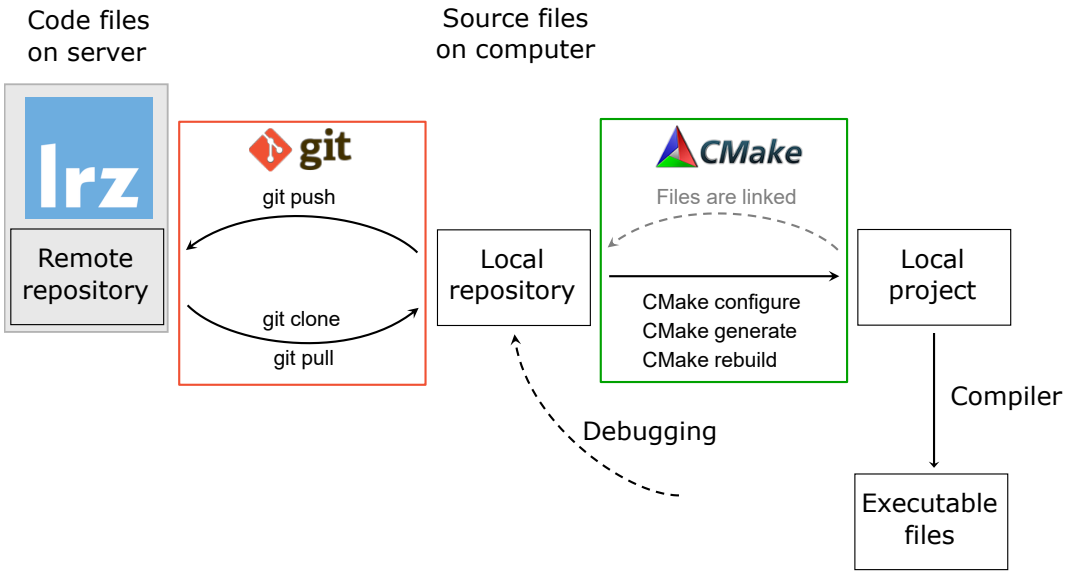
- Version control system to
  - track file history
  - facilitate collaboration across contributors
  - provide a central remote repository
- Can be used from either the command line, a GUI, or through the IDE
- Other alternatives include SVN, CVS, Mercurial, ...

Git Principle

What does `git` do?

Store a named snapshot history of files and distribute them across various locations (local folders, computers, GitLab, ...)

Homework repositories



Git terminology

- repository: a place to store your code
- clone: creates a local copy of a (remote) repository
- fork: creates an *independent* server-side copy of a repository
- pull: updates the local repository with new remote changes
- checkout: change to/creates a branch (version) of the project
- commit: track the specified changes of the local repository
  - the modified files to update need to be staged
  - provides a snapshot of the project as possible reference/fallback point
- push: updates the repository with the local changes
  - ensures that the local version is compatible with the remote one
  - might involve merging local and remote changes
- learn more about git: `https://try.github.io/`

Git workflows

- first time setup

```
git clone <repository>
```

- getting new changes (origin or upstream)

```
git pull <source>
```

- doing work

```
# make some changes
git add <changed files>
git commit
git push
# continue working ...
```

- try it at `https://git-school.github.io/visualizing-git/#upstream-changes`
- check for changes

```
git pull
```

- create an empty commit

```
git commit --allow-empty -m "Hello!"
```

- push the changes to the server

```
git push
```

GitLab

- open source web-based git repository manager
  - server is mainly written in Ruby, PostgreSQL, Redis
  - Vue.js webui
- provides additional features to work with
  - graphical visualization of development history
  - issue tracking
  - project wikis
- basically the free-software GitHub alternative
- the LRZ hosts gitlab at `gitlab.lrz.de`

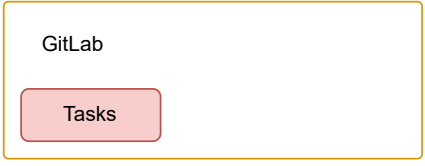
In-class Exercise

Let's simulate the homework git workflow with local git repositories.

- Create a tasks repository and a fork username_tasks.
- Submit changes to our repository, i.e., push to origin
- Get the new homework, i.e., pull from upstream
- Overwrite our local changes, i.e., restore

Creating a new GitLab Repository

- A repository is just a directory + git metadata
- These directories have a `.git` suffix by convention
- We use the `--bare` flag to not check out a branch

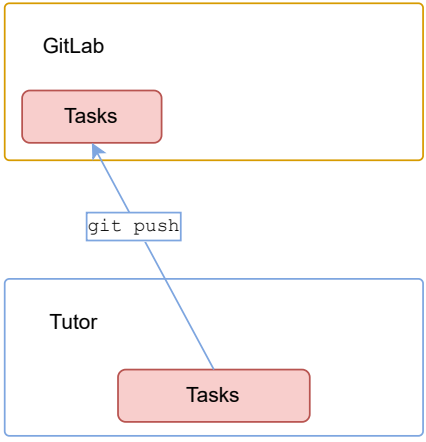



```
# Creating a git repository
mkdir /tmp/git-example/gitlab/tasks.git
cd /tmp/git-example/gitlab/tasks.git
git init --bare
```

Adding a README

```
# Adding a README to the Tasks repository
mkdir /tmp/git-example/tutor
cd /tmp/git-example/tutor
git clone /tmp/git-example/gitlab/tasks.git
cd tasks
echo 'CIIP C++ Course' > README.md
git add README.md
git commit -m 'init'
git push
```

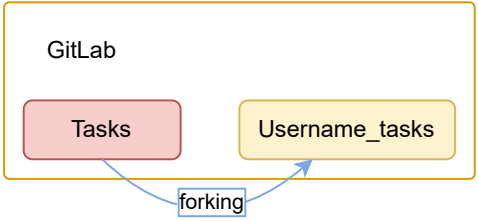
- We clone the Tasks repository to the Tutor directory
- Add a new `README.md` file
- Push the changes (`init`) to remote



Forking (Requesting Access to the Waiting Room)

- A fork is just a `--bare` clone.
- GitLab does the same thing when you request access to the Waiting Room

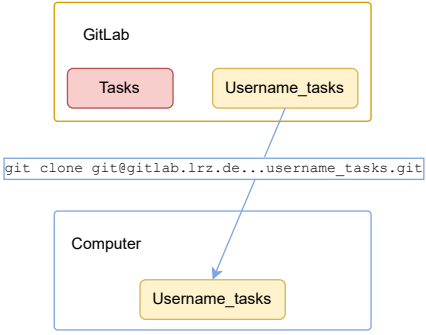
```
# Forking
cd /tmp/git-example/gitlab
git clone --bare tasks.git username_tasks.git
```



Cloning (Downloading your repository)

A clone is just a copy.

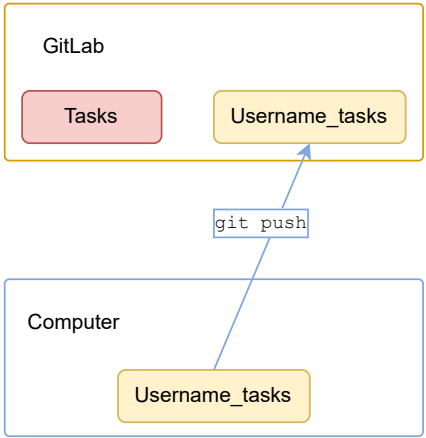
```
# Cloning
mkdir /tmp/git-example/computer
cd /tmp/git-example/computer
git clone /tmp/git-example/gitlab/username_tasks.git
cd username_tasks
```



Pushing (Uploading to GitLab)

- We write “Hello!” into `submission.txt`
- We push our changes to Username_tasks (origin).

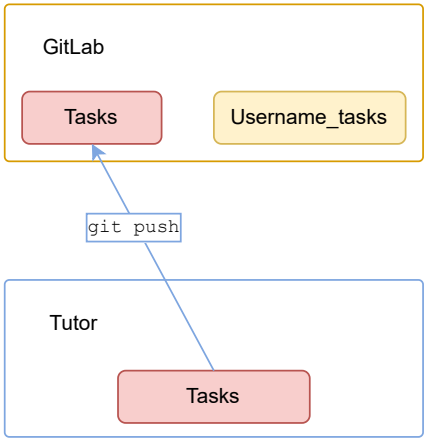
```
# Working
echo 'Hello!' > submission.txt
cat submission.txt

# Tracking changes
git add submission.txt
git commit -m 'Hello World!'

# Pushing
git push
```



Publishing a new homework sheet

- We add a new `submission.txt` file to Tasks
- Push the changes (`Homework 0`) to remote

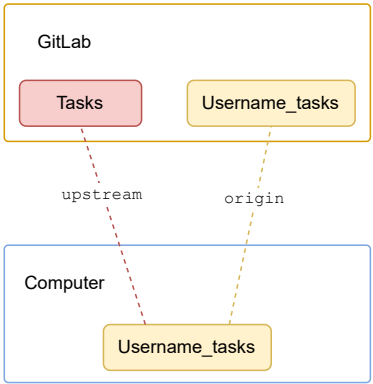
```
# Publishing the first exercise
cd /tmp/git-example/tutor/tasks
echo 'Greetings!' > submission.txt
git add submission.txt
git commit -m 'Homework 0'
git push
```



Getting the homework (Setting upstream)

We define how to access the remote Tasks repository (`upstream`)

```
# Adding upstream
cd /tmp/git-example/computer/username_tasks
git remote add upstream /tmp/git-example/gitlab/tasks.git
```

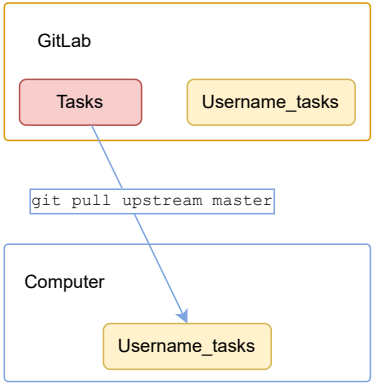


Getting the homework (Pulling from upstream)

We pull `Homework 0` from Tasks (`upstream`)

```
# Getting Homework 0
git config pull.rebase false
git pull upstream master
```

```
From /tmp/git-example/gitlab/tasks
* branch            master     -> FETCH_HEAD
Auto-merging submission.txt
CONFLICT (add/add): Merge conflict in submission.txt
Automatic merge failed;
fix conflicts and then commit the result.
```



Upstream and merge conflicts

- Inspecting the merge conflict

```
cat submission.txt
<<<<<<< HEAD
Hello!
=======
Greetings!
>>>>>>> bc8fe100...
```

- Overwriting our changes

```
git restore -s upstream/master submission.txt
```

- Committing the changes

```
git add submission.txt
git commit -m 'restoring tasks submission.txt'
```

```
diff --git a/submission.txt b/submission.txt
index 10ddd6d..ae4bcb0 100644
--- a/submission.txt
+++ b/submission.txt
@@ -1 +1,5 @@
+<<<<<<< HEAD
 Hello!
+=======
+Greetings!
+>>>>>>> bc8fe100b94adbb2f4ad1dc23ac0ab385a069800
```

Contents

Declaration, definition, and initialization

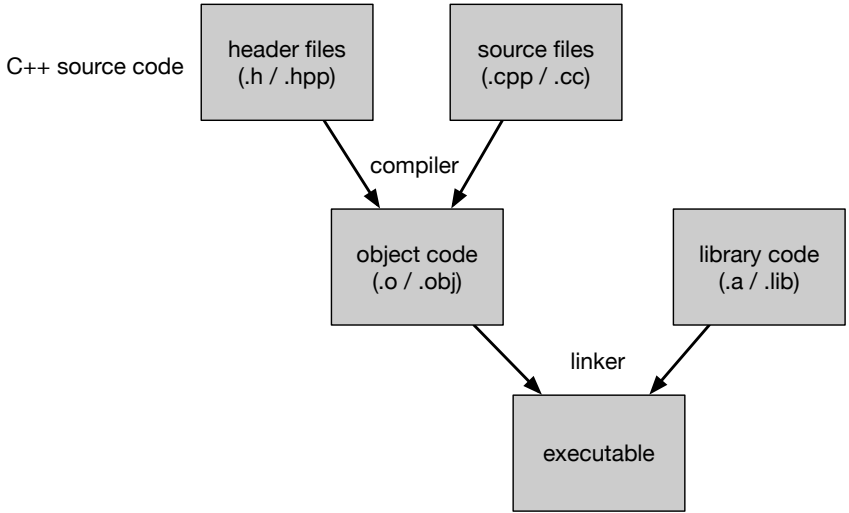
- A `declaration` specifies all that is needed to use a function or type.
  ```
  int f(int); // declares, but doesn't define function f
  ```
- A `definition` is a declaration that fully defines the entity.
  ```
  int i; // declares and defines variable i
  ```
- The `initialization` gives an object a value.
  ```
  int i{1}; // declares, defines, and initializes variable i
  ```

Headers and Sources

- .cpp files denote source files, which contain definitions.
- .h files denote header files, which contain declarations and define interfaces
- Therefore, header files are not explicitly passed as filename, but are included from source files.

- Each compiler invocation is a translation unit!

Compilation and linking



A first C++ program

File `first.cpp` (old style):

```cpp
#include <iostream>

int main() {
  std::cout << "Hello world!\n";
}
```

File `first.cpp` (C++ 23):

```cpp
#include <print>

int main() {
  std::println("Hello world!");
}
```

Preprocessor

- preprocess first.cpp

```
g++ -E -P first.cpp -o first_preprocessed.cpp
```

- resulting first_preprocessed.cpp includes the code from iostream
- file has now ~30k lines instead of 5

Compile

- compile translation unit: first_preprocessed.cpp

```
g++ -c first_preprocessed.cpp
```

- resulting first_preprocessed.o is the compiled source code without the definition of `std::cout`
- The member function std::cout is undefined (U) [1]

```
nm -C first_preprocessed.o
...
0000000000000000 T main
                 U std::ios_base_library_init()
                 U std::cout
                 U std::ostream& std::operator<< (std::ostream &, const char *)
...
```

[1] For more information watch Ben Eater's videos, e.g `https://youtu.be/y0yaJXpAYZQ` or compile the code on `https://godbolt.org/`

Link and Execute

- link first_preprocessed.o

```
g++ first_preprocessed.o -o first
```

- and execture first

```
./first
Hello world!
```

Contents

Preliminaries

Exercise 0:

Gain access to the course group on GitLab and install the required tools

- request access for the Waiting Room at `https://gitlab.lrz.de/cppcourse/ws2023/waiting-room`
- make sure you have all the necessary tools (git, compiler)
- set up your preferred development environment

Setup

Exercise 1:

Set up the assignments repository

- enter (and generate, if necessary) your SSH key in GitLab
- clone the repository to your machine

```
git clone git@gitlab.lrz.de:cppcourse/ws2023/your-username_tasks.git
```

- configure pulling from upstream (for future assignments)
- verify your setup: display remote git servers

```
git remote -v
```

Compiling & Linking

Exercise 2:

Build the provided project manually - and fix the missing parts. Invoke the compiler directly in a shell, like you saw in this exercise session.

- build a shared library from library.cpp
- create a header file for the library and include it in hw01.cpp
- build, link, and run hw01.cpp

Testing

Exercise 3:

Test your library using `doctest`. (We use `doctest` to validate the solutions of upcoming homeworks.)

- Build `test.cpp` while including `doctest` and link it to `library`
- run `./hw01test` to test your library