Introduction to the C++ Ecosystem

Hello World in C++

myprogram.cpp

```cpp
#include <iostream>
int main(int argc, char** argv) {
    std::cout << "Hello " << argv[1] << '!' << std::endl;
    return 0;
}
```

```
$ c++ -std=c++20 -Wall -Werror -o myprogram ./myprogram.cpp
$ ./myprogram World
Hello World!
```

Generating an Executable Program

- Programs that transform C++ files into executables are called *compilers*
- Popular compilers: gcc (GNU), clang (llvm)
- Minimal example to compile the hello world program with gcc:

```
$ g++ -o myprogram ./myprogram.cpp
```

- Internally, the compiler is divided into:
  - Preprocessor
  - Compiler
  - Linker

Compiler Flags

General syntax to run a compiler: `c++ [flags] -o output inputs...`
Most common flags:

`-std=c++20`	Set C++ standard version
`-O0`	no optimization
`-O1`	optimize a bit, assembly mostly readable
`-O2`	optimize more, assembly not readable
`-O3`	optimize most, assembly not readable
`-Os`	optimize for size, similar to `-O3`
`-Wall`	Enable most warnings
`-Wextra`	Enable warnings not covered by `-Wall`
`-Werror`	Treat all warnings as errors
`-march=native`	Enable optimizations supported by your CPU
`-g`	Enable debug symbols

make

- C++ projects usually consist of many `.cpp` (*implementation files*) and `.hpp` (*header files*) files
- Each implementation file needs to be compiled into an object file first, then all object files must be linked
- Very repetitive to do this by hand
- When one `.cpp` file changes, only the corresponding object file should be recompiled, not all
- When one `.hpp` file changes, only implementation files that use it should be recompiled
- make is a program that can automate this
- Requires a `Makefile`
- GNU make manual: `https://www.gnu.org/software/make/manual/make.html`

Basic Makefile

- Makefiles consist of *rules* and contain *variables*
- Each rule has a *target*, *prerequisites*, and a *recipe*
- Recipes are only executed when the prerequisites are newer than the target or when the target does not exist
- Note: The indentation in Makefiles must be exactly one tab character, no spaces!

Makefile

```
CONTENT="test 123"  # set the variable CONTENT
# rule and recipe to generate the target file foo
foo:
	echo $(CONTENT) > foo
# $^ always contains all prerequisites ("foo baz" here)
# $< contains only the first prerequisite ("foo" here)
# $@ contains the target ("bar" here)
bar: foo baz
	cat $^ > $@
```

make and Timestamps

- make uses timestamps of files to decide when to execute recipes
- When any prerequisite file is newer than the target → execute recipe

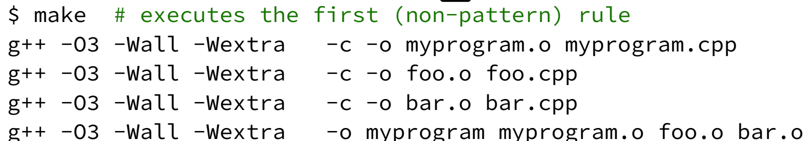
```
$ make foo  # the file foo does not exist yet
echo "test 123"   > foo
$ make foo  # now foo exists
make: 'foo' is up to date.
$ make bar  # bar requires baz which doesn't exist
make: *** No rule to make target 'baz', needed by 'bar'.  Stop.
$ touch baz # create the file baz
$ make bar
cat foo baz > bar
$ make bar  # bar exists, nothing to do
make: 'bar' is up to date.
$ touch baz # update timestamp of file baz
$ make bar  # now the recipe for bar is executed again
cat foo baz > bar
```

Advanced Makefile

- Recipes are usually the same for most files
- *Pattern rules* can be used to reuse a recipe for multiple files

Makefile

```
CXX?=g++  # set CXX variable only if it's not set
CXXFLAGS+= -O3 -Wall -Wextra  # append to CXXFLAGS
SOURCES=foo.cpp bar.cpp
%.o: %.cpp  # pattern rule to make .o files out of .cpp files
	$(CXX) $(CXXFLAGS) -c -o $@ $<
# use a substitution reference to get .o file names
myprogram: myprogram.o $(SOURCES:.cpp=.o)
	$(CXX) $(CXXFLAGS) -o $@ $^
```



```
$ make  # executes the first (non-pattern) rule
g++ -O3 -Wall -Wextra   -c -o myprogram.o myprogram.cpp
g++ -O3 -Wall -Wextra   -c -o foo.o foo.cpp
g++ -O3 -Wall -Wextra   -c -o bar.o bar.cpp
g++ -O3 -Wall -Wextra   -o myprogram myprogram.o foo.o bar.o
```

CMake

- `make` prevents writing many repetitive compiler commands
- Still, extra flags must be specified manually (e.g. `-l` to link an external library)
- On different systems the same library may require different flags
- CMake is a tool specialized for C and C++ projects that uses a `CMakeLists.txt` to generate `Makefiles` or files for other build systems (e.g. ninja, Visual Studio)
- Also, the C++ IDE CLion uses CMake internally
- `CMakeLists.txt` consists of a series of *commands*
- CMake Reference Documentation: `https://cmake.org/cmake/help/latest/`

Basic CMakeLists.txt

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(myprogram)
set(MYPROGRAM_FILES sayhello.cpp saybye.cpp)
add_executable(myprogram myprogram.cpp ${MYPROGRAM_FILES})
```

```
$ mkdir build; cd build  # create a separate build directory
$ cmake ..  # generate Makefile from CMakeLists.txt
-- The C compiler identification is GNU 8.2.1
-- The CXX compiler identification is GNU 8.2.1
[...]
-- Configuring done
-- Generating done
-- Build files have been written to: /home/X/myproject/build
$ make
Scanning dependencies of target myprogram
[ 25%] Building CXX object CMakeFiles/myprogram.dir/myprogram.cpp.o
[ 50%] Building CXX object CMakeFiles/myprogram.dir/sayhello.cpp.o
[ 75%] Building CXX object CMakeFiles/myprogram.dir/saybye.cpp.o
[100%] Linking CXX executable myprogram
```

CMake Commands

`cmake_minimum_required(VERSION 3.10)`
: Require a specific cmake version.

`project(myproject)`
: Define a C/C++ project with the name “myproject”, required for every project.

`set(FOO a b c)`
: Set the variable `${FOO}` to be equal to `a b c`.

`add_executable(myprogram a.cpp b.cpp)`
: Define an executable to be built that consists of the source files `a.cpp` and `b.cpp`.

`add_library(mylib a.cpp b.cpp)`
: Similar to `add_executable()` but build a library.

`add_compile_options(-Wall -Wextra)`
: Add `-Wall -Wextra` to all invocations of the compiler.

`target_link_library(myprogram mylib)`
: Link the executable or library `myprogram` with the library `mylib`.

CMake Variables

CMake has many variables that influence how the executables and libraries are built. They can be set in the `CMakeLists.txt` with `set()`, on the command line with `cmake -D FOO=bar`, or with the program `ccmake`.

`CMAKE_CXX_STANDARD=20`
: Set the C++ to standard to C++20, effectively adds `-std=c++20` to the compiler flags.

`CMAKE_CXX_COMPILER=clang++`
: Set the C++ compiler to `clang++`.

`CMAKE_BUILD_TYPE=Debug`
: Set the "build type" to `Debug`. Other possible values: `Release`, `RelWithDebInfo`. This mainly affects the optimization compiler flags.

`CMAKE_CXX_FLAGS(_DEBUG/_RELEASE)=-march=native`
: Add `-march=native` to all compiler invocations (or only for the `Debug` or `Release` build types).

Subdirectories with CMake

- Larger C++ projects are usually divided into subdirectories
- CMake allows the `CMakeLists.txt` to also be divided into the subdirectories
- A subdirectory can have its own `CMakeLists.txt` (without the `project()` command)
- The "main" `CMakeListst.txt` can then include the subdirectory with `add_subdirectory(subdir)`

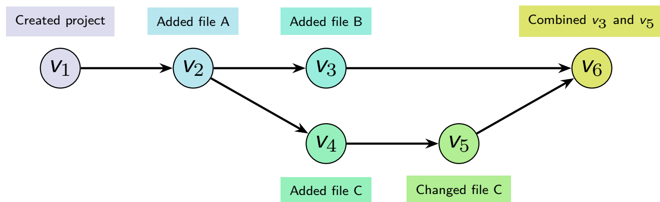
Complete CMake Example

```
cmake_example_project
├── CMakeLists.txt
├── lib
│   ├── CMakeLists.txt
│   ├── saybye.cpp
│   ├── saybye.hpp
│   ├── sayhello.cpp
│   └── sayhello.hpp
└── src
    ├── CMakeLists.txt
    └── print_greetings.cpp
```

- This project contains the library `greetings` and the executable `print_greetings`
- The library consists of the files `sayhello.cpp` and `saybye.cpp`
- You can find this project in our Gitlab

Version Control Systems (VCS)

- Code projects evolve gradually
- Incremental changes, also called *versions*, should be tracked to allow:
  - Documentation of the project history
  - Selective inspection/modification of specific versions
  - Efficient collaboration when working in a team
- Version Control Systems (VCS) manage versions, usually represent them in a directed acyclic graph



Git

- Many VCS exist, Git is a very popular one: Used by Linux, GCC, LLVM, etc.
- Git in particular has the following advantages compared to other version control systems (VCS):
  - Open source (LGPLv2.1)
  - Decentralized, i.e., no server required
  - Efficient management of *branches* and *tags*
- All Git commands are document with man-pages (e.g. type `man git-commit` to see documentation for the command `git commit`)
- Pro Git book: `https://git-scm.com/book`
- Git Reference Manual: `https://git-scm.com/docs`

Git Concepts

Repository: A collection of Git objects (*commits* and *trees*) and references (*branches* and *tags*).

Branch: A named reference to a *commit*. Every repository usually has at least the `master` branch and contains several more branches, like `fix-xyz` or `feature-abc`.

Tag: A named reference to a *commit*. In contrast to a branch a tag is usually set once and not changed. A branch regularly gets new commits.

Commit: A snapshot of a *tree*. Identified by a SHA1 hash. Each commit can have multiple parent commits. The commits form a directed acyclic graph.

Tree: A collection of files (not directories!) with their path and other metadata. This means that Git does *not* track empty directories.

Creating a Git Repository

`git init`
: Initialize a Git repository

`git config --global user.name <name>`
: Sets the name that will be used in commits

`git config --global user.email <email>`
: Sets the e-mail address that will be used in commits

`git status`
: Shows information about the repository

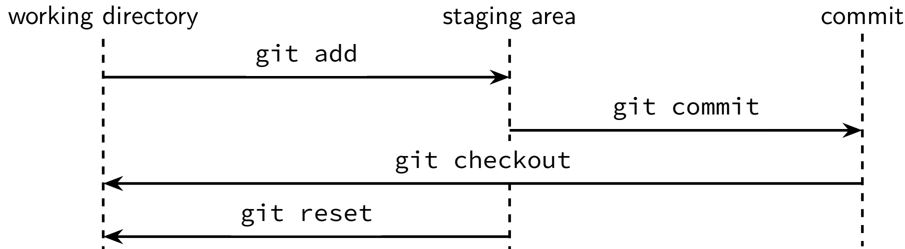
```
$ mkdir myrepo && cd myrepo
$ git init
Initialized empty Git repository in /home/X/myrepo/.git/
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Git Working Directory and Staging Area

When working with a Git repository, changes can live in any of the following places:

- In the working directory (when you edit a file)
- In the staging area (when you use `git add`)
- In a commit (after a `git commit`)

Once a change is in a commit and it is referenced by at least one branch or tag, you can always restore it even if you remove the file.



Committing Changes

`git add [-p] <path>...`
: Add changes to the staging area. Changes can be selected interactively when the -p option is used.

`git reset [-p] <path>...`
: Remove changes from the staging area without directly modifying the files. Can also be done interactively with -p.

`git commit`
: Take all changes from the staging area and turn them into a commit. Includes a commit message and author and date information. The parent of the new commit is set to the newest commit of the current branch. Then, the current branch is updated to point to the new commit.

`git checkout -- <path>...`
: Remove changes from the working directory by overwriting the given files or directories with their committed versions.

Inspecting the Commit History (1)

`git log [<branch>]`
View the commit history of the current (or another) branch.

`git show [<commit>]`
Show the changes introduced by the last (or the given) commit.

- “Browsing” the commit history with Git alone usually requires you to know the commands that list commits, show changes, etc., and execute several of them.
- There is a program called `tig` that provides a text-based interface where you can scroll through branches, commits, and changes.
- Running `tig` without arguments shows an overview of the current branch.
- `tig` also understands the subcommands `tig status`, `tig log`, and `tig show`, which take the same arguments as the `git` variants

Inspecting the Commit History (2)

`git diff`
: View the changes in the working directory (without the staging area).

`git diff --staged`
: View the changes in the staging area (without the working directory).

`git diff HEAD`
: View the changes in the working directory and the staging area.

`git diff branch1..branch2`
: View the changes between two branches (or tags, commits).

Example output of `git diff`

```
diff --git a/foo b/foo
index e965047..980a0d5 100644
--- a/foo
+++ b/foo
@@ -1 +1 @@
-Hello
+Hello World!
```

Working with Branches and Tags

`git branch`
: Show all branches and which one is active.

`git branch <name>`
: Create a new branch that points to the current commit (`HEAD`).

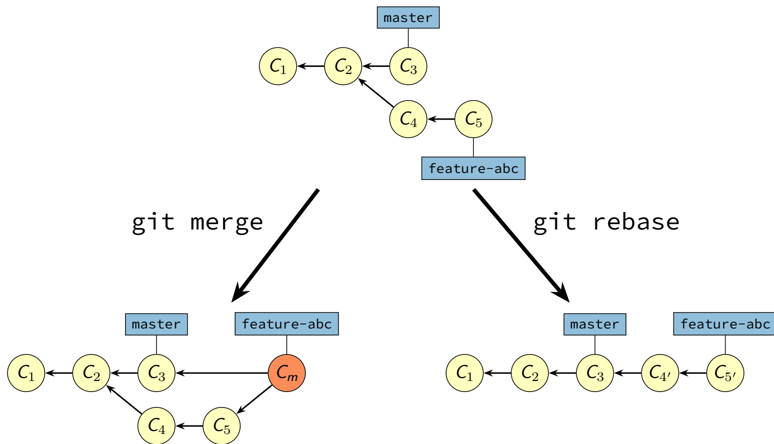
`git checkout <name>`
: Switch to another branch, i.e. change all files in the working directory so that they are equal to the tree of the other branch.

`git checkout -b <name>`
: Create a branch and switch to it.

`git tag`
: Show all tags.

`git tag [-s] <name>`
: Create a new tag that points to the current commit. Is signed with PGP when `-s` is given.

Modifying the Commit History (overview)



Modifying the Commit History

`git merge <branch>...`

- Combines the current branch and one or more other branches with a special *merge commit*
- The merge commit has the latest commit of all merged branches as parent
- No commit is modified

`git rebase <branch>`

- Start from the given branch and reapply all diverging commits from the current branch one by one
- All diverging commits are changed (they get a new parent) so their SHA1 hash changes as well

Dealing with Merge Conflicts

- Using `merge` or `rebase` may cause *merge conflicts*
- This happens when two commits are merged that contain changes to the same file
- When a merge conflict happens, Git usually tells you:

```
$ git merge branch2
Auto-merging foo
CONFLICT (content): Merge conflict in foo
Automatic merge failed; fix conflicts and then commit the result.
```

- `git status` also shows additional information on how to proceed
- To fix the conflict you have to manually fix all conflicting files. Git inserts markers in the files to show where the conflicts arose:

foo
```
<<<<<<< HEAD
Hello World!
=======
Hello You!
>>>>>>> branch2
```

Undoing Committed Changes

: This operation may potentially irrevocably remove data

git revert <commit>
Create a new commit that is the "inverse" of the specified commit.

git reset <commit>
Reset the current branch to point to the given commit. No files are changed.

git reset --hard <commit>
Reset the current branch to point to the given commit. All files in the working directory are overwritten.

git rebase -i <commit>
Show all commits from the given one up to the current one and potentially remove individual commits.

git reflog
Shows a history of SHA1 commit hashes that were added or removed. Allows to restore removed commits if they were not garbage collected yet.

Working with Remote Git Repositories

git clone <url>
Download the repository with all its commits, tags, and branches from the url.

git push
Upload the current branch to a remote repository.

git push --force-with-lease

Override the current branch on the remote repository. This is necessary when the local and remote branches have diverging histories, e.g., after using git rebase or git reset --hard.

git fetch
Download new commits, tags, and branches from a remote repository into an existing repository.

git pull
Run git fetch and then update (i.e. git merge) the current branch to match the branch on the remote repository.

Finding out Who Wrote the Code

- Sometimes, especially when reading a new code base, you want to know which commit changed a certain line
- Also, sometimes you want to know *who* wrote a certain line

`git blame <filename>`

- Shows the given file with commit annotations
- Each line starts with the commit hash, the name of the author, and the commit date

`tig blame <filename>`

- Like `git blame` but with a nicer interface
- Allows to "re-blame" at a given line, i.e. showing the selected line in the version just before it was last modified
- `tig` can also be used with other git commands: `tig log`, `tig diff`, etc.

Special Files in Git

`.gitignore`

- `git status`, `git diff`, etc. usually look at all files in all subdirectories of the repository
- If files or directories should always be excluded (e.g. build or cache directories), they can be added to the `.gitignore` file
- This file contains one entry per line, lines starting with # are skipped:

`foo.txt`	Ignores all files named `foo.txt`
`/foo.txt`	Ignores only the file `foo.txt` in the top-level directory
`foo/`	Ignores all directories named foo and their contents
`*f*`	Ignores all files and directories that contain the letter f

`.git`

- This directory contains all commits, branches, etc.
- E.g., `.git/refs/heads` contains one file per branch
- If you remove this directory, all data is lost!