Intro to Efficient Programming

Mikolaj Krzewicki[1]   Patrick Huhn[3]   Sandro Wenzel[2]

[1]FIAS/University of Frankfurt

[2]CERN

[3]University of Frankfurt

HGS-HIRe power week

Limburg

June 2019

Lecture I

Introduction

Who we are

Mikolaj Krzewicki

- in ALICE since 2007

- ALICE High Level Trigger / ALICE O2

- Software validation

- TPC calibration

- Analysis (correlations and flow)

Patrick Huhn

- in ALICE since 2014

- PhD student since 2017

- Analysis (charged particle R_{AA})

Sandro Wenzel

- at CERN since 2012; in ALICE since 2015

- Detector Simulation

- Previous experience in various computing activities (PhD: Quantum Monte Carlo; The Blue Brain Project)

Credits

Material of this course building on previous courses given by Jens Wiechula, David Rohr, Matthias Richter and Jochen Klein.

Outline / Goals of the week

Software development / computing in high energy physics is ubigous and essential knowledge. This week should help you to ...

- ▶ Be able to come up with efficient algorithms in C++ implementing software solutions to problems in (particle) physics.
- ▶ Be able to use ROOT as a library and to look into data using ROOT
- ▶ Be able to decompose problem into small pieces, structure code and work on software projects incrementally
- ▶ Understand tools and practices in the software development process
- ▶ Know about modern C++11 features and few optimization strategies
- ▶ Know about easy parallelism options and SIMD

Introduction

There are many aspects of *efficiency*

- ▶ Coding concepts

- ▶ Tools

- ▶ Fast development (prototyping)

- ▶ Fast code execution (optimization)

- ▶ Small memory imprint

- ▶ Code design

- ▶ Code flexibility / configurability

- ▶ . . .

Programme

What we want to cover:

- ▶ Modern C++ features and concepts (C++11, C++14, ...)
- ▶ Tools

  - ▶ gcc
  - ▶ (c)make
  - ▶ git

  - ▶ doxygen
  - ▶ gdb
  - ▶ profilers: valgrind, perf, ...

- ▶ Methods
  - ▶ object orientation, templates
  - ▶ libraries
- ▶ Algorithms
- ▶ Parallelisation
- ▶ SIMD vectorisation

Outlook Programme 2

What we can't cover here but intend to do in part 2:

- ▶ GP-GPU
- ▶ Parallelization in more depth
- ▶ distributed computing and messaging
- ▶ ...

Course format

There will be some lectures but focus will be on practical side!

- ▶ Lot's of do-it yourself exercises/examples
- ▶ A real coding project touch typical high energy physics subject
- ▶ Possibility to do code reviews / interaction with lecturers

link to dynamic plan
https://tinyurl.com/hgspw1

A small project

- small groups (up to 5 people)

- work shall be carried out over the whole week,
  presentation of results on Friday (20+5),
  code reviews in between

- we want you to
  - use the tools
  - try the methods
  - test the algorithms

  which are discussed during the meeting

- you should learn something
  $\Rightarrow$ try and understand what you are doing

Computing I

▶ local servers (hostnames: power[1-4].power.week)
  personal user accounts:
  username: first letter of firstname $+$ lastname,
  initial password: pwLimburg
  (reference environment, you can compare to your machine)

▶ Every groups gets assigned one server (please use it exclusively)

▶ separate network with WLAN access (or cable)
  SSID: power-week
  pw: efficient

▶ passwordless login
  often it is convenient to login usind ssh keys

```
ssh-keygen
ssh-copy-id <you-user-id>@power[1-4]
```

Computing II

- examples and slides are provided via git

```
git clone https://github.com/hgspowerweek/powerweek1/
cd powerweek1
git pull
```

do the last step before every session and you will get the latest examples and slides

- Slides are directly in this folder as pdfs

- Examples are in the folder *examples*

```
examples/bla
```

Lecture II

Introduction to code / document management using git – absolute basics

Code repository

Why would you use a code repository?

- ▶ keep control over your changes
- ▶ keep a history of changes and go back to any previous state
- ▶ add logging messages to individual changes
- ▶ develop different topics in parallel
- ▶ keep a working version as a reference
- ▶ create releases for distribution of the code
- ▶ synchronize several developers

A code repository can serves as

- ▶ back-up solution
- ▶ communication medium
- ▶ team and product management tool

git - the stupid content tracker

- ▶ Developed by Linus Torvalds and others in 2005 for the Linux kernel community
- ▶ Nobody knows what git stands for at least one does not get a real answer.
- ▶ Instead of beeing stupid - see manpage - it's a extremely powerful scalable, distributed revision control system.

In contrast to other versioning systems (CVS, subversion),

- ▶ git allows to use the full functionality of a code repository locally
- ▶ can be distributed
- ▶ does not require a central server, but can be used with a server

git commands in a nutshell

Basic operations:

- `git init`
- `git clone`
- `git add` – add something to staging area
- `git commit` – commit staging area to local reposity
- `git checkout <commit>` – retrieve certain state

Getting information

- `git log` – show commit history
- `git status`
- `git diff` – show differences (between commits)

Advanced commands

- `git stash`
- `git push` – publish repo somewhere else (some URL)
- `git pull` – sync/get changes from somwhere else (some URL)
- `git rebase/merge` – integrate someone elses changes; change history

git - the stupid content tracker

How is a git repository structured:

- **local copy** (tree) - those are your files.
- **staging area** (intermediate store) - changes (diffs) staged for commit.
- **local repository** - full repository containing all changes to all branches. Everybody has a full copy, there is no concept of a central repository - you still may declare some repository the central one.
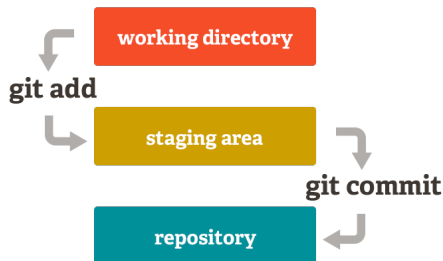
git - One time (identity) setup

- ▶ git config Configure git or query configuration
- ▶ Some essential setup: Give yourself a git identity:
  ```
  git config --global user.name "Foo Bar"
  git config --global user.email foo.bar@cern.ch
  ```
- ▶ git config -l will show you the whole configuration including your identity
- ▶ configuration is stored in a file ${HOME}/gitconfig which can also be edited

git - Creating and cloning repositories

```
mkdir -p ~/src/project
cd ~/src/project
git init
Initialized empty Git repository in ~/src/project/.git
```

```
git clone power1:/data/PowerWeek
Cloning into 'PowerWeek'...
Password:
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
Resolving deltas: 100% (1/1), done.
Checking connectivity... done
```

git - status of the local repository

Command: `git status`

```
richterm@power1 ~ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:    twoparticle.C
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       result.root
#       twoparticle_C.d
#       twoparticle_C.so
no changes added to commit (use "git add" and/or "git commit -a")
```

- information about the current branch
- files which are staged for commit
- tracked files whith local changes
- untracked files (can be masked by .gitignore)

git - committing
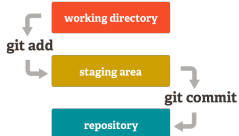
Step 1: `git add` mark changes to be committed

```
richterm@power1 ~/src/example_01 $ git add twoparticle.C
richterm@power1 ~/src/example_01 $ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:    twoparticle.C
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       result.root
#       twoparticle_C.d
#       twoparticle_C.so
```

git - committing
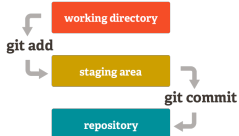
Step 2: `git commit` changes

```
richterm@power1 ~/src/example_01  git commit -m "initial version
of particle class"
[master a56e827] initial version of particle class
```

Now it is locally committed, check the log

```
richterm@power1 ~/src/example_01  git log
commit a56e8270fd6f3c99d4cdbcdb0e45f287e1c71711
Author: Matthias Richter <richterm@power1.power.week>
Date:   Tue Nov 26 11:51:09 2013 +0100

    initial version of particle class

commit ea2c328e7ac3e21c5546480dad1a55af4a0f5e35
Author: Jochen Klein <jochen.klein@cern.ch>
Date:   Mon Nov 25 13:51:12 2013 +0100

    - initial commit of example
```
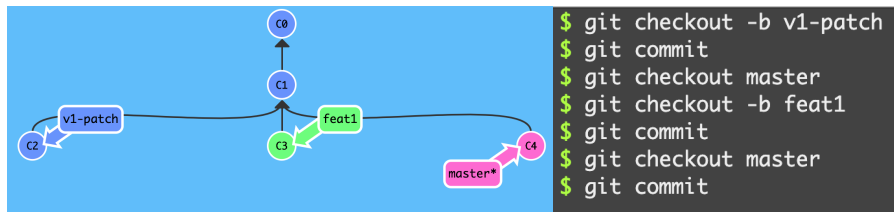
git commits; git checkout

- git organizes commits in a connected **tree** of nodes; A git commit adds a new node
- each node consists of
    - The actual changeset/diff to files
    - Metadata (author, ...)
    - A commit message
    - A SHA-256 hash digest - This hash uniquely identifies the precise node and all its history!
- one can checkout specific nodes by using `git checkout <commit-sha>`
- pointer to last node – of main development line – is typically called `master`
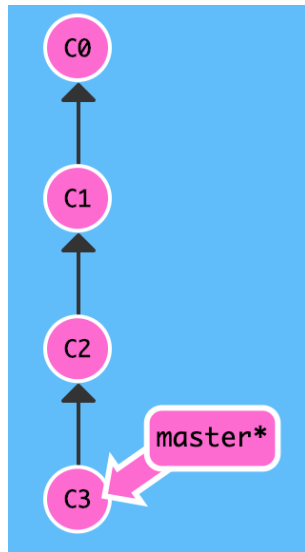
The commit tree and branches

The git commit structure can be a tree. Pointers to leave nodes are called branches.

- ▶ The master branch is the main development line
- ▶ Other branches typically used for feature development in isolation (`Feat1`) or for releasing a certain stable version and patches (`v1-patch`)
- ▶ branches are started with `git commit -b NewFeature` on the currently checked out commit
- ▶ one can switch between branches with `git checkout branchname`



```
$ git checkout -b v1-patch
$ git commit
$ git checkout master
$ git checkout -b feat1
$ git commit
$ git checkout master
$ git commit
```

nice online learning platform: `https://learngitbranching.js.org/?NODEMO`

Short intro to merging and rebasing

- **merging/rebasing** : operations on the tree to bring together 2 branches

- used to integrate commits from one branch into the other
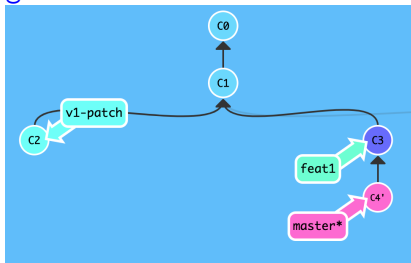
- for example when feature is fully developed
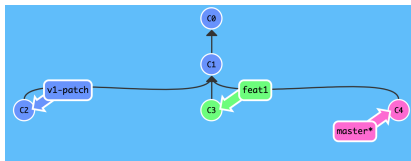
`git merge feat1`
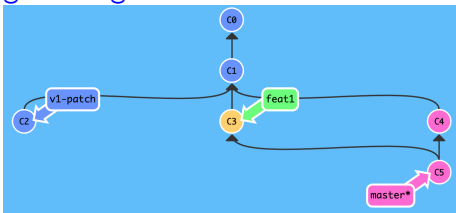
- tree (old commits) stays intact
- merge creates adds a special commit

`git rebase feat2`

- branches are linearized
- no new commit; but old commits rewritten

git - looking at the difference

Command: `git diff`

```
richterm@power1 ~/src/example_01 git diff twoparticle.C
diff --git a/twoparticle.C b/twoparticle.C
index fdca6ac..77d77c5 100644
--- a/twoparticle.C
+++ b/twoparticle.C
@@ -1,7 +1,7 @@
 // a simple macro with surprises for the purpose of training usage of valgrind a

 // include header files for the porpose of compilation
-#ifndef __CINT__
+#if !defined(__CINT__) || defined(__MAKECINT__)
 #include "TParticle.h"
 #include "TSystem.h"
 #include "TH1.h"
```

- ▶ shows local differences in tracked files
- ▶ without arguments: for all tracked files
- ▶ can be used to show differences between revisions

Distributed gits: push and pull

- A git repository can exist at various locations (or in multiple copies) at the same time (it's distributed); one speaks of local and remote repositories.
- the authorative version of a git repository is often hosted on some web server (remote)
  - if you cloned from the remote it is called 'origin'
  - otherwise you can declare a remote repo with `git remote add foo URL`
- Users syncronize local and remote repositories via git pull and git push commands.
  - git pull [–rebase] foo get all remote changes and apply locally
  - git push foo publish your own changes to the remote
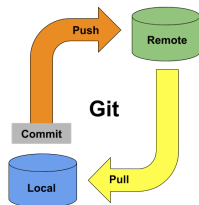
Github/Gitlab/Bitbucket

- many open source projects host git on platforms like Github, Gitlab or Bitbucket
- free git server and infrastructure
- collaborative code reviews
- integrations with other services
    - task tracking
    - continuous integration (CI) - automatic testing of new code and accepting only if good
    - documentation
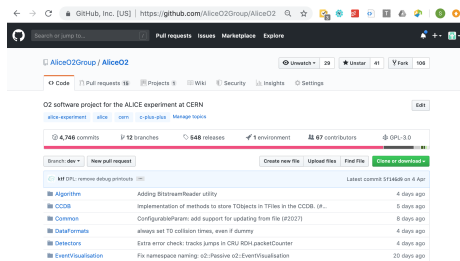
Github/Gitlab/Bitbucket - workflow

- A typical workflow on those platforms uses 3 repositories
  - the authorative repo (remote)
  - a fork (a user copy remote)
  - the local user repository
- Changes are integrated via push to fork and pull requests to real repo
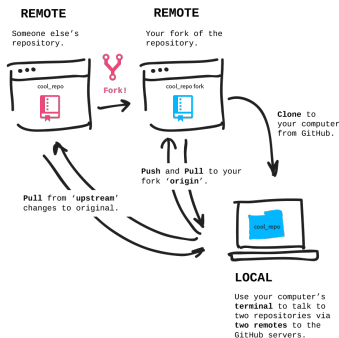
git - Further reading

We found the following nice looking pages ... but there are tons other really

- https://rogerdudler.github.io/git-guide/index.de.html
- https://try.github.io/ – resources to learn git;
- https://www.atlassian.com/git/tutorials
- https://www.edureka.co/blog/git-tutorial/
- `git help`

git - stash

You are in the middle of developing and get a request to fix something or
update your clone
$\Rightarrow$ stash allows to save your current status

```
> git pull
# ... pull fails due to merge conflicts ...
> git stash save
> git pull
> git stash pop
```

git - checking logs revisited

tig; gitk; etc

Exercises

- ▶ Create a git repo; Do some add-commit cycles
- ▶ Play with branching merging:
  - ▶ locally
  - ▶ as an interactive game `https://learngitbranching.js.org/`
- ▶ Get familiar with github by forking or cloning the PowerWeek github repo `https://github.com/hgspowerweek/powerweek1`.
  - ▶ Look around
  - ▶ Contribute to the documentation of F.A.Q. section
- ▶ In order to contribute to an github repo, you need to create a github account
- ▶ For the coding project, we suggest to use gitlab.cern.ch in a private repository
  - ▶ needs CERN lightweight account account.cern.ch
  - ▶ keeps code private (also for future power weeks participants)
  - ▶ enables code review features

Lecture III

Code compilation

What is GCC

- ▶ Originally 'only' GNU C Compiler

- ▶ Release in March 1987 as the first **free** ANSI C optimizing compiler

- ▶ C++ support was added in December of that year

- ▶ Now, many other languages are supported as well, e.g. Objective-C, Objective-C++, Fortran (gfortran), Java (gcj), ...

- ▶ In addition, many different CPU architectures supported, e.g. Intel, ARM, Alpha, PowerPC, ...

- ▶ Today GCC stands for GNU Compiler Collection

Compilers

A compiler translates the human readable code into machine executable code

The main compilers of the GCC suite we are interested in are the GNU C and C++ compilers: gcc and g++

Brian Gough
http://www.network-theory.co.uk/docs/gccintro/

examples/gcc

First steps with g++

```cpp
#include <iostream>
int main()
{
  std::cout << "hello world" << std::endl;
  return 0;
}
```

The hello world example above (hello_world.cpp) can be compiled using

```
g++ -Wall hello_world.cpp -o hello_world
```

-o specifies the name of the executable (default it is *a.out*)

-Wall turns on most commonly used compiler warning → **highly recommended to use**

To run the program simply type

```
./hello_world
```

⇒

Try it!

First steps with g++

```cpp
#include <iostream>
int main()
{
  std::cout << "hello world" << std::endl;
  return 0;
}
```

The hello world example above (hello_world.cpp) can be compiled using

```
g++ -Wall hello_world.cpp -o hello_world
```

-o specifies the name of the executable (default it is *a.out*)

-Wall turns on most commonly used compiler warning → **highly recommended to use**

To run the program simply type

```
./hello_world
```
⇒

Try it!

First steps with g++

```cpp
#include <iostream>
int main()
{
  std::cout << "hello world" << std::endl;
  return 0;
}
```

The hello world example above (hello_world.cpp) can be compiled using

```
g++ -Wall hello_world.cpp -o hello_world
```

-o specifies the name of the executable (default it is *a.out*)

-Wall turns on most commonly used compiler warning → **highly recommended to use**

To run the program simply type

```
./hello_world
```

Try it!

First steps with g++

```cpp
#include <iostream>
int main()
{
  std::cout << "hello world" << std::endl;
  return 0;
}
```

The hello world example above (hello_world.cpp) can be compiled using

```
g++ -Wall hello_world.cpp -o hello_world
```

-o specifies the name of the executable (default it is *a.out*)

-Wall turns on most commonly used compiler warning → **highly recommended to use**

To run the program simply type

```
./hello_world
```
⇒

Try it!

Splitting code

Often it is useful to split the code into separate logical files

- ▶ Enhances readability and maintenance
- ▶ Enables to compile code parts independently
    - ▶ Saves compilation time, not all code needs to be recompiled if somethings changes
- ▶ Allows to compile code using the functionality of other code without knowing the actual implementation

We split the hello_world example into three files:

- ▶ main.cpp
- ▶ hello_fn.h
- ▶ hello_fn.cpp

Splitting code

Often it is useful to split the code into separate logical files

- ▶ Enhances readability and maintenance
- ▶ Enables to compile code parts independently
  - ▶ Saves compilation time, not all code needs to be recompiled if somethings changes
- ▶ Allows to compile code using the functionality of other code without knowing the actual implementation

We split the hello_world example into three files:

- ▶ main.cpp
- ▶ hello_fn.h
- ▶ hello_fn.cpp

Splitting code - example

main.cpp

```cpp
#include "hello_fn.h"
int main()
{
  hello("world");
  return 0;
}
```

hello_fn.h

```cpp
void hello(const char* to);
```

hello_fn.cpp

```cpp
#include <iostream>
void hello(const char* to)
{
  // function to print hello to someone on the command line
  std::cout << "Hello " << to << std::endl;
}
```

Splitting code - header files

- Separate the *declaration* of classes / functions from the actual *implementation*
- The declaration is given in the *header file* ending on `.h`
- When using external code in an own class, during compilation only the declaration is needed
- A declaration should not be included several times (compilation time), this is handled by a pre-compiler directive (*header guard*)

```
#ifndef MYCODE_H
#define MYCODE_H
void myfunction(int x, float y);
#endif
```

Compile multiple source files

To compile the code run

```
g++ -Wall main.cpp hello_fn.cpp -o hello_world
```

⇒ Try it!

Not won too much, still all code is compiled all the time

- compile parts of the code into separate *object files*
- *link* the *object files* to the executable

Compile multiple source files

To compile the code run

```
g++ -Wall main.cpp hello_fn.cpp -o hello_world
```

⇒ Try it!

Not won too much, still all code is compiled all the time

- ▶ compile parts of the code into separate *object files*
- ▶ *link* the *object files* to the executable

Creating object files

We create one *object file* per input file:

```
g++ -Wall -c main.cpp hello_fn.cpp
```

- ▶ `-c` tells the compiler to create an object
- ▶ object files are machine code, but not yet executable

Produces the object files *main.o* and *hello_fn.o*

This can also be run separately for each file
```
g++ -Wall -c main.cpp
g++ -Wall -c hello_fn.cpp
```
⇒ Try it!

Creating object files

We create one *object file* per input file:
```
g++ -Wall -c main.cpp hello_fn.cpp
```

- ▶ -c tells the compiler to create an object
- ▶ object files are machine code, but not yet executable

Produces the object files *main.o* and *hello_fn.o*

This can also be run separately for each file
```
g++ -Wall -c main.cpp
g++ -Wall -c hello_fn.cpp
```
⇒ Try it!

Linking objects to an executable

Now the objects can be *linked* together to the executable:

```
g++ main.o hello_fn.o -o hello_world
```

NOTE:
The code is already compiled → You don't need warning options

⇒ Try it!

- ▶ Modify something in one of the files (e.g. world → moon in main.cpp)
- ▶ Recompile only main.cpp
- ▶ Link all files to one executable

⇒ Try it!

Linking objects to an executable

Now the objects can be *linked* together to the executable:
```
g++ main.o hello_fn.o -o hello_world
```
NOTE:
The code is already compiled → You don't need warning options
⇒ Try it!

- ▶ Modify something in one of the files (e.g. world → moon in main.cpp)
- ▶ Recompile only main.cpp
- ▶ Link all files to one executable

⇒ Try it!

Makefiles

- ▶ The steps mentioned above can be automatized using the *make system*
- ▶ Define dependencies (e.g. the executable can only be built if all objects files are available)
- ▶ Only compiles code which changed

Documentation:
https://www.gnu.org/software/make/manual/make.html

Makefiles – An example

```
CXX       = /usr/bin/g++
CXXFLAGS  = -Wall -Wextra -Wconversion -Wshadow -g
LDFLAGS   =

OBJ       = hello_fn.o main.o

split: $(OBJ)
        $(CXX) -o $@ $(OBJ) $(LDFLAGS)

main.o: main.cpp hello_fn.h
        $(CXX) -o $@ -c $< $(CXXFLAGS)

%.o: %.cpp %.h
        $(CXX) $(CXXFLAGS) -c $< -o $@


clean:
        @rm -f ${OBJ} split
```

Makefiles – Primer

Makefiles consist of rules that tell the make system what to do. A rule has the form:

```
target ... : prerequisites ...
<tab> recipe
<tab> ...
<tab> ...
```

`target` is usually an output file name, `prerequisites` can be other targets or e.g. file names

NOTE: A `recipe` MUST be started with a $< tab >$

When the `make` command is called, it looks for a file called *Makefile* or *makefile* in the present directory and processes the first target (*default target*).

Makefiles – Primer

A few important automatic variables are defined in the `make` system:

- `$@` The target name
- `$<` The first prerequisite
- `$^` The names of all the prerequisites, with spaces between them

For more see `https://www.gnu.org/software/make/manual/make.html#Automatic-Variables`

⇒ Try it!

CMake - introduction

- ▶ treatment of dependencies and automatic re-compilation covered by Makefiles and make

- ▶ manual maintenance of Makefiles can become tedious and error-prone

- ▶ configuration for specific setup of SDK and external dependencies covered by autotools suite or CMake

CMake - introduction

CMake is an open-source, cross-platform family of tools
designed to build, test and package software.

- ▶ universal (toolchain agnostic) description of build flow
  in CMakeLists.txt

- ▶ automated creation of Makefiles
  depending on configuration options environment found

- ▶ automatic re-evaluation when needed

- ▶ possibly add testing and packaging steps

here: CMake with Makefiles

typical build layout

- ▶ separate build into different directories:
    - ▶ source: no generated files
    - ▶ build: generated files, object files, libraries, executables
      possibly more than one with different build options
    - ▶ install: final files only

- ▶ test stage

- ▶ delivering/deploying stage

a minimal CMake project

```cmake
cmake_minimum_required(VERSION 3.9)
project(MyPowerProject CXX)

add_executable(testExe test.cc)
```

- ▶ always specify minimum version of CMake
  (depending on the features you use)

- ▶ declare project and language used

- ▶ add targets (here just one executable)

example project:

```
examples/cmake_simple
```

a CMake run

```
cd <build directory>
cmake <source directory>
make -j$(nproc)
```

happens in phases:

- ▶ configuration:
  evaluation of CMakeLists.txt files, additional options, toolchain
  probing

- ▶ generation:
  generation of build files (depending on selected generator)

- ▶ compilation:
  actual build using Makefiles

libraries

```
...
add_library(power power.cc)
add_executable(main main.cc)
target_link_libraries(main power)
```

- ▶ add target for library and source files needed to build it

- ▶ link executable against library

- ▶ include directories are propagated to the targets using the library

using lists

```
...
set(SOURCES s1.cc s2.cc s3.cc)
add_library(s ${SOURCES})

set(EXECUTABLES e1 e2 e3)
foreach(EXE ${EXECUTABLES})
  add_executable(${EXE} ${EXE}.cc)
  target_link_libraries(${EXE} power)
endforeach()
```

- ▶ define lists that can be reused

- ▶ avoid overly repetitive code

ROOT integration

```
...
find_package(ROOT)
include(${ROOT_USE_FILE})
if(ROOT_FOUND)
  message(STATUS "Using ROOT: ${ROOT_VERSION} <${ROOT_CONFIG}>")
  target_compile_definitions(power PUBLIC "-DUSE_ROOT")
  target_include_directories(power PRIVATE ${ROOT_INCLUDE_DIRS})
  target_include_directories(power PRIVATE .)
  ROOT_GENERATE_DICTIONARY(G__Power ${CMAKE_CURRENT_SOURCE_DIR}/power.h LINKDEF Lin
  target_sources(power PRIVATE power_rooted.cc G__Power)
  target_link_libraries(power ROOT::Core ROOT::Gui ROOT::Tree)
endif(ROOT_FOUND)
```

▶ ROOT comes with additional tools to build dictionaries

warnings and errors

```
message("something important")
message(STATUS "just a status message")
message(WARNING "something's fishy here")
message(ERROR "this is plain wrong")
message(FATAL_ERROR "this is too wrong, I rather die ...")
```

▶ use messages to check on your build,
  don't be blind on what is happening

language options

```
...
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
message(STATUS "Using C++${CMAKE_CXX_STANDARD}")

enable_language(CUDA)
```

- ▶ compiler-agnostic settings of language standard
  N.B.: you can also request specific language features

- ▶ enabling of additional programming languages

build types and compiler options

```
set(CMAKE_CXX_FLAGS_DEBUG "-O0 -ggdb -DDEBUG -D__DEBUG")
set(CMAKE_CXX_FLAGS_RELWITHDEBINFO "${CMAKE_CXX_FLAGS_RELEASE} -ggdb")
set(CMAKE_CXX_FLAGS_RELEASE "-O3 -march=native -ftree-vectorize -ffast-math -DNODEB
message(STATUS "Using CXX flags for ${CMAKE_BUILD_TYPE}: ${CMAKE_CXX_FLAGS_${CMAKE_
```

- ▶ compiler options/flags are controlled by build types

- ▶ can be changed separately for different build types

- ▶ don't put target-specific stuff here

```
cmake -DCMAKE_BUILD_TYPE=DEBUG <source dir>
```

default build type

```
# by default build optimized code with debug symbols
if(NOT CMAKE_BUILD_TYPE AND NOT CMAKE_CONFIGURATION_TYPES)
  set(CMAKE_BUILD_TYPE RELWITHDEBINFO)
endif()

set(CMAKE_ALLOWED_BUILD_TYPES DEBUG RELEASE RELWITHDEBINFO)
if(NOT CMAKE_BUILD_TYPE IN_LIST CMAKE_ALLOWED_BUILD_TYPES)
  message(FATAL_ERROR "Invalid build type ${CMAKE_BUILD_TYPE}. Use one of: ${CMAKE_
endif()
```

▶ explicit control over default build type

install

```
install(TARGETS main power
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin
)
```

- ► this should install only the final build products

- ► try and adhere to conventions about installation paths

some more involved stuff

- nested projects

- tests

nested projects

```
add_subdirectory(sub1)
add_subdirectory(sub2)
```

- ▶ include sub-projects in sub-directories,
  hierarchical layout

- ▶ sub-directories can contain project,
  this allows to build the sub-project independently

- ▶ different directories:
  - ▶ CMAKE_SOURCE_DIRECTORY: directory from which cmake was run
  - ▶ CMAKE_PROJECT_SOURCE_DIRECTORY: (sub-)project directory
  - ▶ CMAKE_CURRENT_SOURCE_DIRECTORY: current directory in
    source tree

CTest

▶ you can add tests which can be run programatically

```
enable_testing()

add_test(NAME MyPowerTest COMMAND echo done)

ctest
```

You can also run this using the *make* process by

```
make test
```

summary

- surely not an extensive coverage of CMake,
  but it should get you started

- extensive documentation on project pages
  but not always good explanation of the underlying concepts