Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 14 – Build Automation

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 23.05.2019
FS2019

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

Motivation

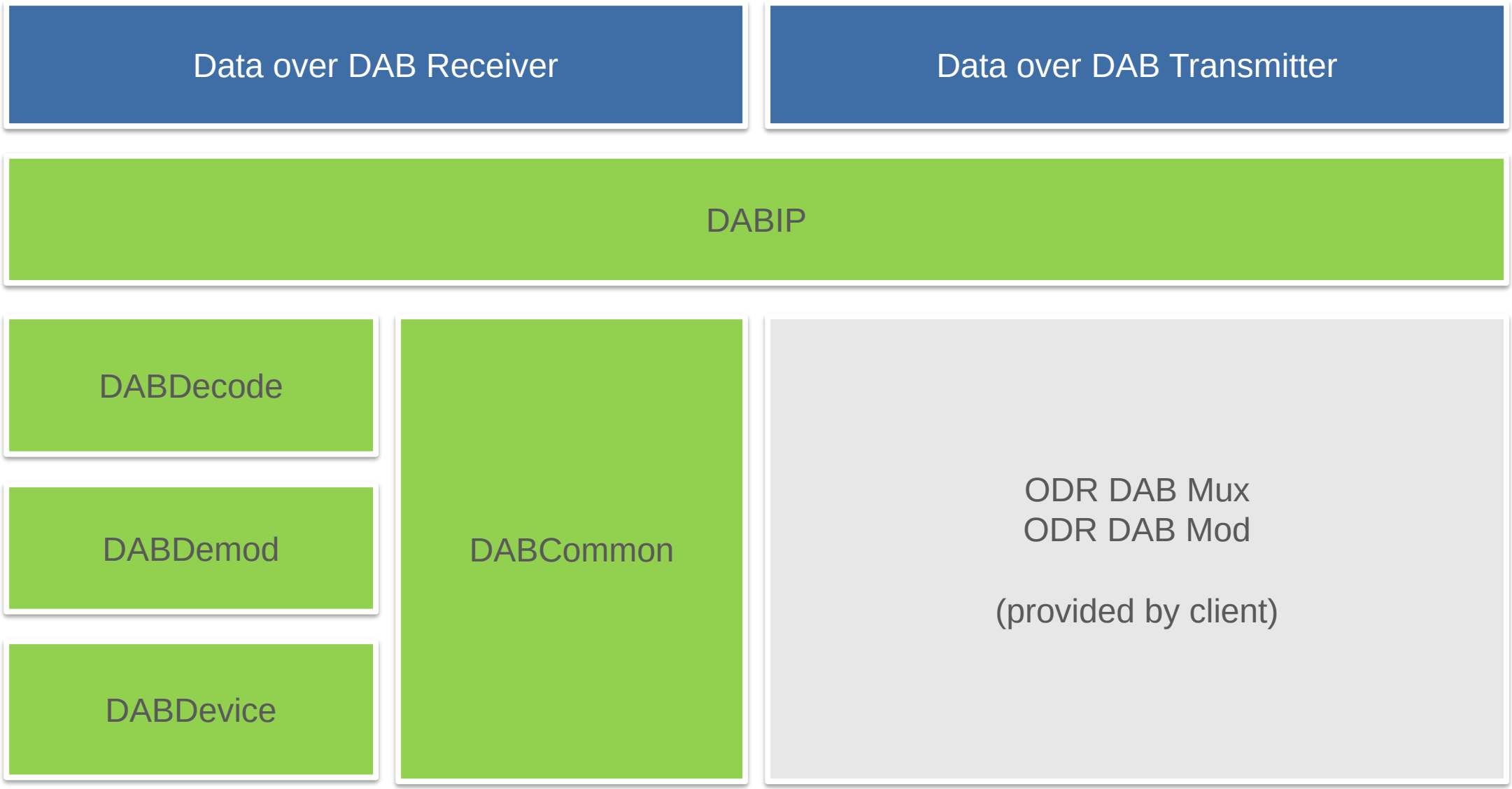
Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

- **You know how to set up build automation for your own projects**
- **You can explain why you should have build automation in your projects**
- **You know how to structure non-trivial projects**

- **Imagine that…**
  - ... you plan on building a large product (maybe your Thesis/Term Project?)
  - … your product consists of multiple parts
  - … you need to have build products at any moment (ship early, ship often)
  - … you need to target multiple platforms
  - … others need to build your code (maybe on different platforms)
  - … you work in a team
  - … everyone uses their favorite IDE or editor
- **With what you know now, does that sound like fun?**
- **Sounds made up or too theoretical?**


Data over DAB Receiver
Data over DAB Transmitter
DABIP
DABDecode
DABDemod
DABDevice
DABCommon
ODR DAB Mux
ODR DAB Mod
(provided by client)

- **Five libraries**
  - All depending on a common infrastructure library
- **Two executables**
  - Depend on some or all of the libraries
- **Two target-platforms**
  - Linux on x86_64 and armv7
  - OS X
- **Code will change owners**
- **4 months time-frame**

Build Automation

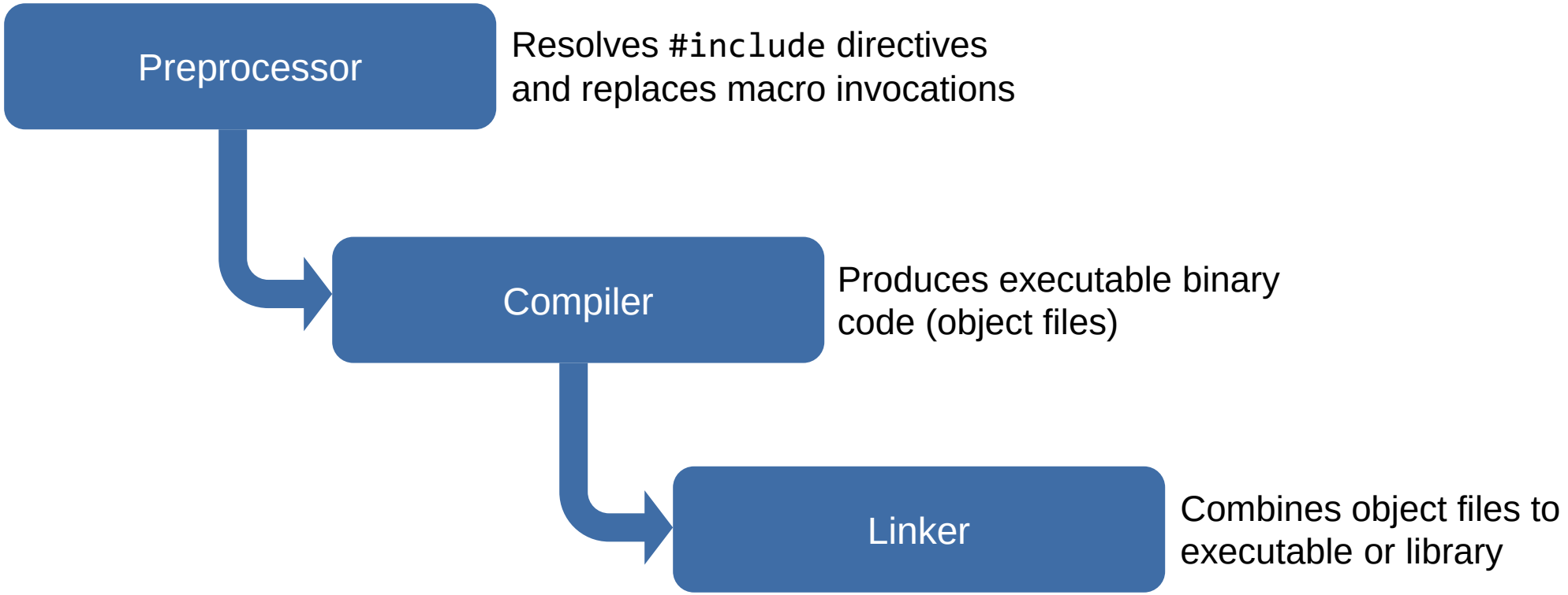
- **Build automation and Reproducibility**
  - No “Wait for <insert name here> to build the package”!
  - No “Builds on my machine”!
- **Productivity**
- **Project Layout / Maintainability**
  - Independent code should live in a separate project
  - Link- and compile-time dependencies must be easy to resolve
- **Shareability**

Visual Studio Code

Visual Studio®

Code::Blocks

- **There are many IDEs and Editors available**
  - An you should make use of them!
- **Most IDEs have a concept of “Projects” (Project Layout / Maintainability)**
  - Click “Build” to get your program/library (Productivity)
- **But…**
  - … do we want to run an IDE on our build server? (Build automation)
  - … does the IDE run on other Platforms? (Shareability)
  - … how are compiler/linker flags stored and shared? (Reproducibility)
  - … are project files of X compatible with Y? (Shareability)


Preprocessor
Resolves #include directives
and replaces macro invocations
Compiler
Produces executable binary
code (object files)
Linker
Combines object files to
executable or library

- **The compiler generates object files**
  - `gcc -c packet_parser.cpp`
    - Output: “packet_parser.o”
  - Could specify multiple at a time
    - `gcc -c packet_parser.cpp packet_generator.cpp -o parser_and_generator.o`
- **Object files get linked together**
  - `gcc -shared -l libdabdemod.so packet_parser.o packet_generator.o …`
  - `gcc my_awesome_function.o main.o –o my_awesome_app`

Idea: Scripts!

- **Write a script that…**
  - Compiles each source file
  - Links all object files together
  - Repeats that for every target
- **Profit!**

```
#!/bin/bash

# Build libdabip
gcc -c package_parser.cpp
gcc -c package_generator.cpp
…
gcc –shared -o libdabip package_parser.o
…

# Build libdabdecode
gcc -c ensemble.cpp
gcc -c subchannel.cpp
…
gcc -shared -o libdabdecode ensemble.o …

# Build <you get the idea>
```

- **Write a script that…**
  - Compiles each source file
  - Links all object files together
  - Repeats that for every target
- **DON'T! Because…**
  - … every source file get built every time!
  - … the commands tend to be platform specific
  - … build order must be managed manually
  - … scripts tend to become messy over time

```
#!/bin/bash

# Build li dabip
gcc -c p      parser.cpp
gcc -c        nerat
…
gcc –shar                  e_parser.o
…

# Build l
gcc -c
gcc -c        1.cpp
…
gcc -share -o libdabdecode nsemble.o …

# Build <you get the idea>
```

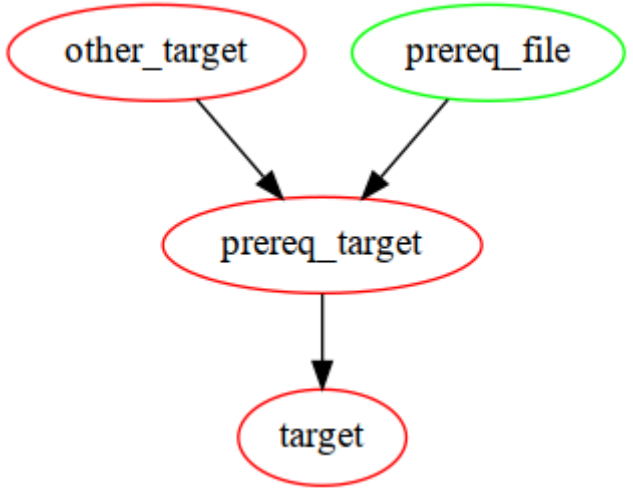
- **Building non-trivial projects is an old problem**
- **There are plenty of existing tools:**
  - GNU make
  - Scons
  - Ninja
  - CMake
  - autotools
  - …
- **Don’t reinvent the wheel! Other people are doing that for you…**

- **Incremental builds**
- **Parallel builds**
- **Automatic (intra-project) dependency resolution**
- **Package management**
- **Automatic test execution**
- **Platform independence**
- **Additional processing of build products**
  - E.g. code signing, minification, …

- **Make-style Build Tools**
  - Run build scripts
  - Produce your final products
  - Often verbose
  - Use a language agnostic configuration language
- **Build Script Generators**
  - Generate configurations for Make-style Build Systems or Build Scripts
  - Configuration independent of actual build tool
  - Advanced features (download dependencies, etc.)

- **Well-known tool to build all kinds of projects**
  - Many IDEs “understand” make projects
- **Workflow description in Makefile via “Target” rules**
  - Each target may have one or more prerequisites…
  - …and execute one or more commands to…
  - …generate one or more results
- **Targets are then executed “top-down”**
- **A Target is only executed if required**

```
target: prereq_target

prereq_target: prereq_file other_target
    command_to_generate_output

other_target:
```



```cpp
// main.cpp

#include <iostream>

int main() {
  std::cout << "This is my awesome app!\n";
}
```

```
# Makefile

all: my_app

my_app: main.o
    g++ -o my_app main.o

main.o: main.cpp
    g++ -c main.cpp
```

```
$ make
```

- **Consider a project with multiple files**
  - Requires a target for every object file
  - Not a lot better than shell script
  - Lots of duplication
    - Need to specify compiler flags for each target
- **Idea:**
  - Define how to create an object file
  - Let make generate “implicit” targets

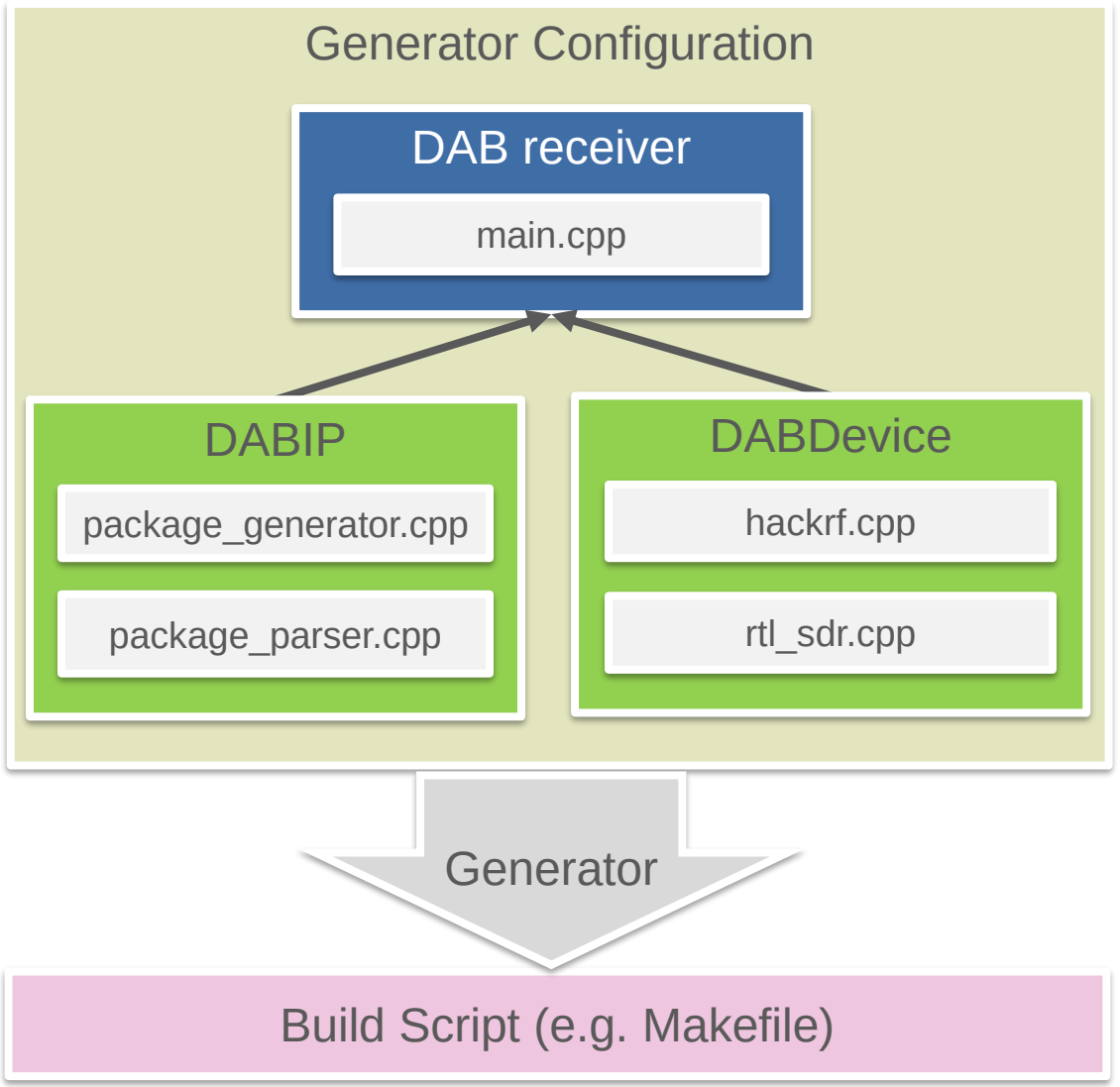
```
all: frobnibulator

frobnibulator: main.o frobnify.o discombobulate.o
    g++ -o frobnibulator $^

main.o: main.cpp
    g++ -c main.cpp

frobnify.o: frobnify.cpp
    g++ -c frobnify.cpp

discombobulate.o: discombobulate.cpp
    g++ -c discombobulate.cpp
```

- **Pattern Rules**
  - Use % as placeholder
  - Use $^ to refer to all prerequisites
  - Use $< to refer to the first prerequisite
  - Value derived from usage in target name
  - Can be accessed in prerequisites
  - Can be overridden
    - Best match wins
  - Only one place to specify compiler flags and one place for linker flags

```
OBJECTS = main.o frobnify.o discombobulate.o

all: frobnibulator

frobnibulator: $(OBJECTS)
    g++ -o frobnibulator  $^

%.o: %.cpp
    g++ -c $<
```

- **Pros:**
  - Very generic automation tool
  - Powerful pattern matching mechanism
  - Builds only what is needed, when its needed
- **Cons:**
  - Often platform-specific commands
  - Need to specify how to do things

- **Idea: Take a step back**
  - Define what we want to achieve, not how to do it
  - Work on a higher level
  - Let the create the actual build configurations
- **Platform independent build specification**
- **Tool Independent**
  - Often can generate IDE projects
  - Support multiple build tools



- **Widely used. e.g.:**
  - Netflix
  - LLVM
  - vcpkg (Microsoft)
- **Built-in support for many languages**
  - C, C++, Java, C#, Swift, …
  - Can be extended if needed
- **Custom configuration language**
- **Platform independent**

```cpp
// main.cpp

#include <iostream>

int main() {
  std::cout << "This is my awesome app!\n";
}
```

```
# CMakeLists.txt

project(my_app LANGUAGES CXX)

add_executable(my_app main.cpp)
```

```
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
```

Hint: **Always** use "`cmake --build .`" NOT "`make`" to build your CMake project! Why?

- **`project(…)` command defines …**
  - … the name of our project
  - … which languages we use
- **Common compiler flags can be set using built-in variables**
- **`include_directories(…)` defines include search paths**
- **`add_library(…)` defines libraries**
- **`add_executable(…)` defines binaries**
- **`target_link_libraries(…)` specifies libraries to link against**

```
project(my_app LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

include_directories(some_dir)

add_library(awesome awesome.cpp)
add_executable(my_app main.cpp)
target_link_libraries(my_app awesome)
```

```
# Makefile

OBJECTS = main.o frobnify.o discombobulate.o

all: frobnibulator

frobnibulator: $(OBJECTS)
    g++ -o frobnibulator $^

%.o: %.cpp
    g++ -c $<
```

```
# CMakeLists.txt

project(frobnibulator LANGUAGES CXX)

add_executable(frobnibulator
    discombobulate.cpp
    frobnify.cpp
    main.cpp)
```

- **CMake includes CTest**
  - Enable CTest using `enable_testing()`
  - Create a “Test Runner” executable
    - Make sure to include your suite sources!
  - Configure build environment:

```
$ cmake ..
```

  - Build the project:

```
$ cmake --build .
```

  - Run `ctest`

```
$ ctest --output-on-failure
```

```
# CMakeLists.txt

project(answer LANGUAGES CXX)

enable_testing()

include_directories(cute)

add_library(answer answer.cpp)

add_executable(test_runner Test.cpp)

target_link_libraries(test_runner answer)

add_test(tests test_runner)
```

- **CMake makes building project easy**
  - Platform specifics are handled behind the scenes
  - Declare **what** you want, not **how** to create it
  - CTest allows you to run your unit tests
- **You can choose what kind of build scripts you want:**

```
$ cmake .. -G"Eclipse CDT4 - Unix Makefiles"
```

```
$ cmake .. -G"MinGW Makefiles"
```

- **Declare the version of CMake required for your project**

```
cmake_minimum_required(VERSION 3.14.0)
```

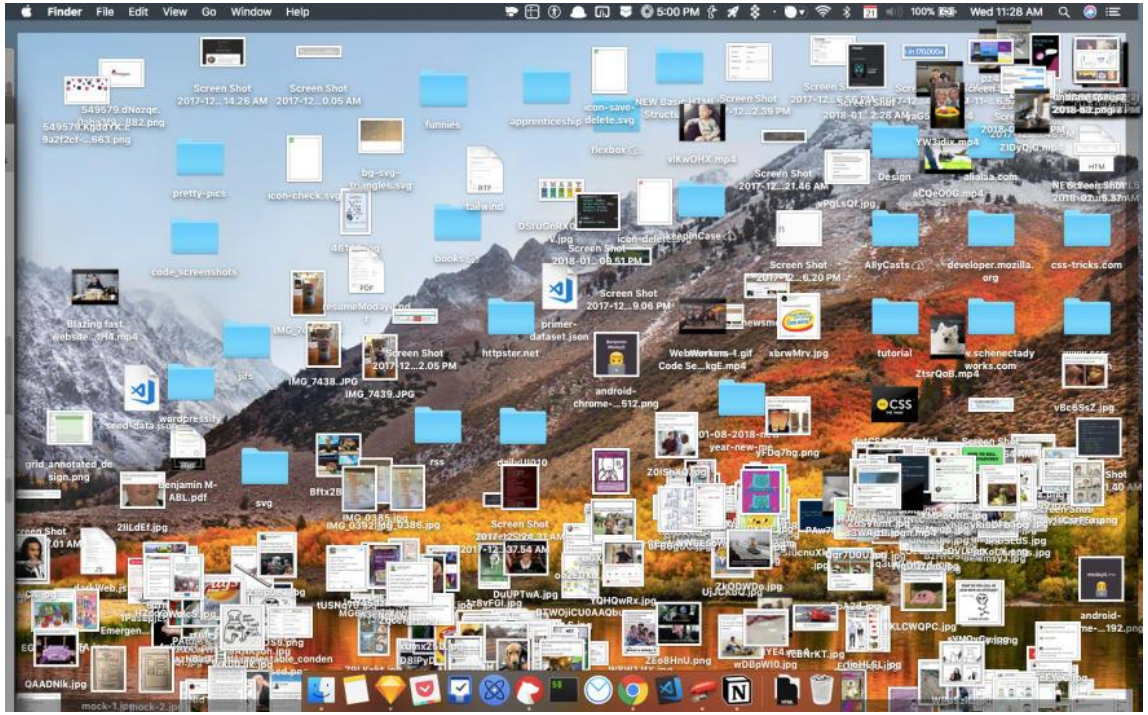
Project Layout

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

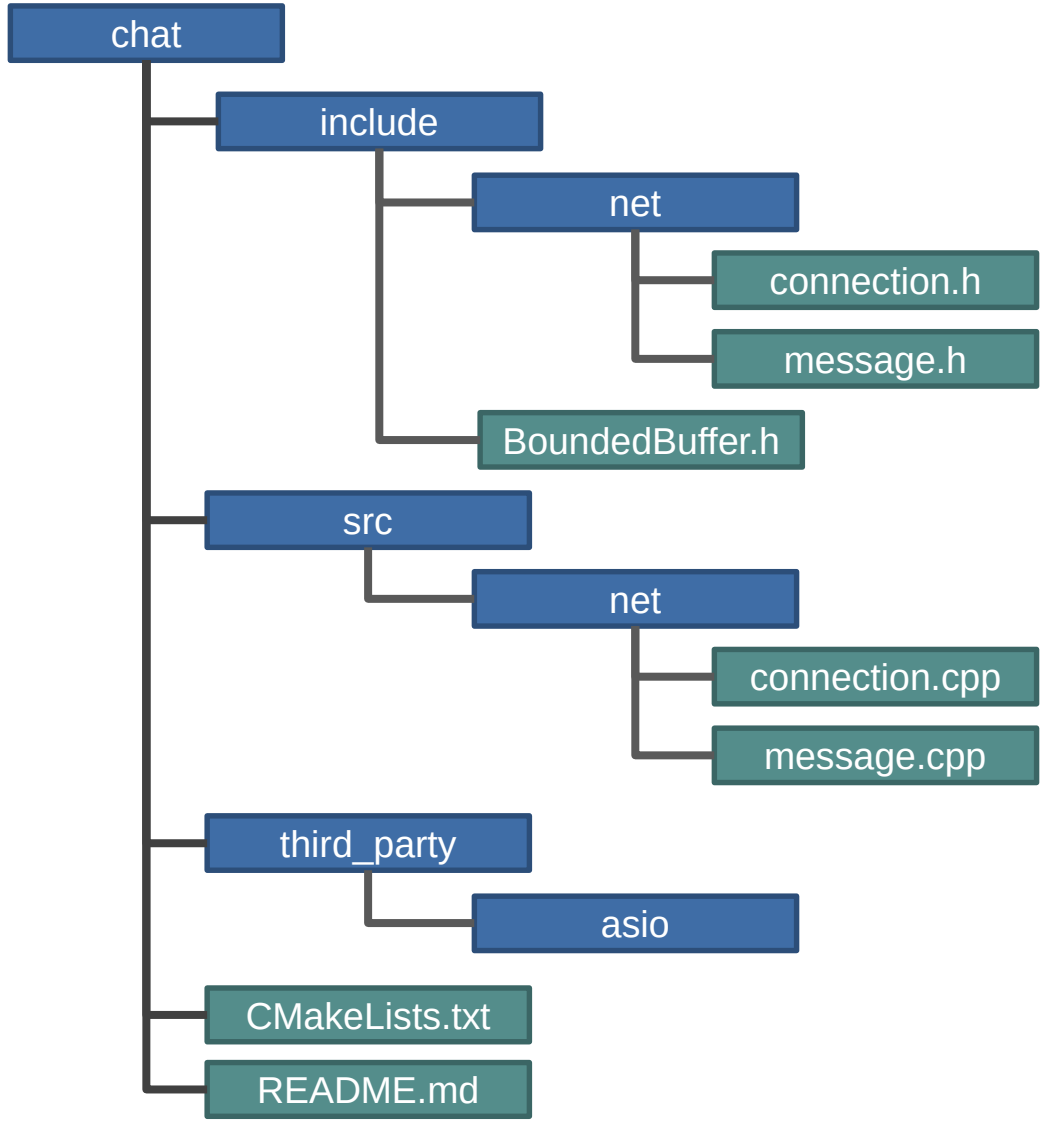
IFS
INSTITUTE FOR
SOFTWARE

- **C++ does not enforce any Layout**
  - Can have all files in one directory…
  - … or each file in a separate directory …
  - … and anything in between
- **Best practices**
  - Separate headers from implementation files
  - Group files by submodule / functionality
  - Be **consistent!**



Don’t let your projects look like this!

- **Headers live in the “include” folder**
  - Add subfolders for separate subsystems if needed
- **Implementation files live in the “src” folder**
  - Make sure that subfolder layout matches the “include” folder (**consistency**)
- **Put third-party projects/sources in a “third_party” or “lib” folder**
- **Test resource live in the “test” folder**
  - The test folder will have src, include, and third_party subfolders if required
- **Build configuration files should live in the root of your project**



- **Libraries may benefit from a slightly different layout**
  - You will need to ship your headers
  - Your headers might have very generic names
- **Idea: Introduce another nesting level for your headers**
  - Use the name of your project

```
#include “message.h”
```

… becomes …

```
#include “chat/message.h”
```

  - Helps avoid filename clashes