Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 13 – Build Automation

Thomas Corbat / Felix Morgner

Rapperswil, 31.05.2022

FS2022

Agenda

- **Recap Week 12**
- **Build Automation**
- **Structuring Projects**

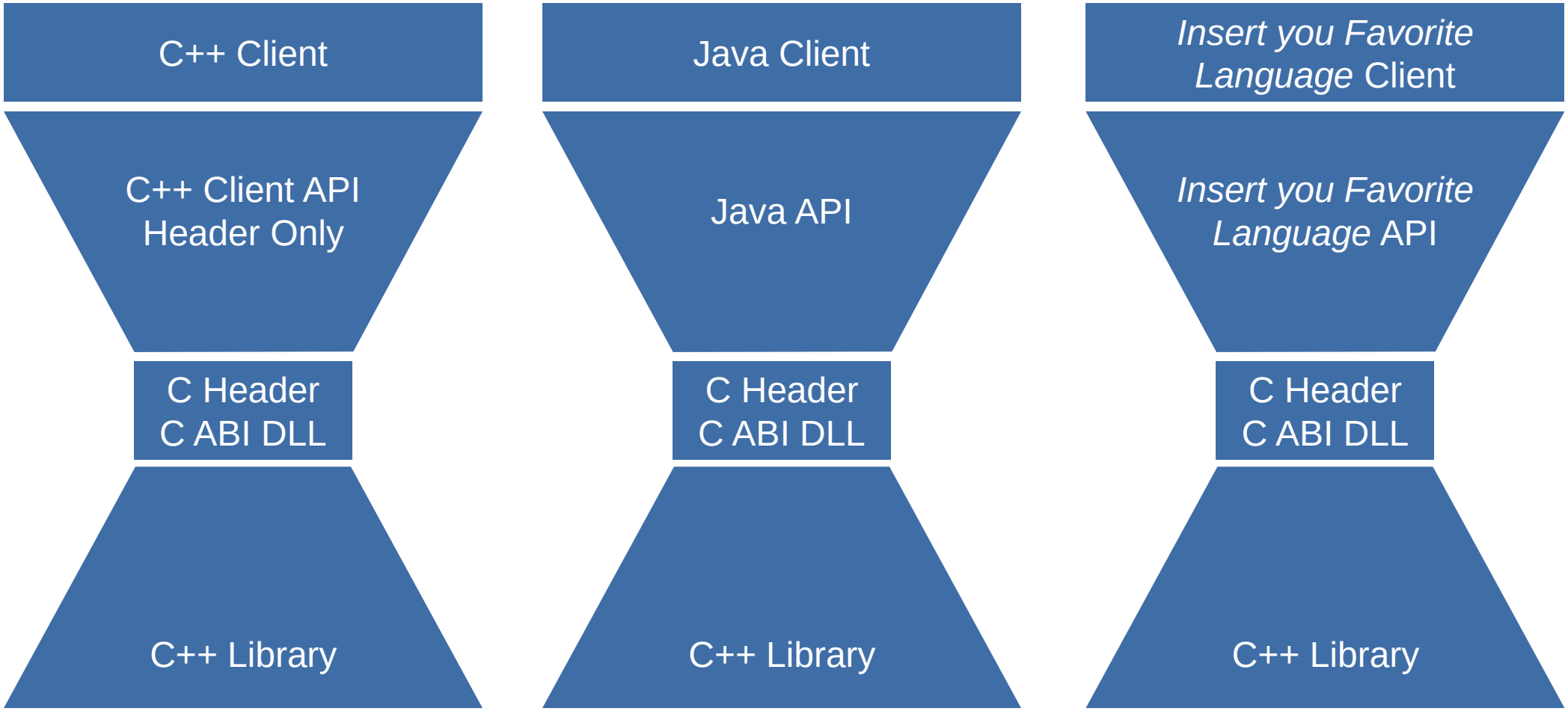
Goals

- **Participants should …**
  - know how to set up build automation for their own projects
  - explain why projects should have build-automation in place
  - know how to structure non-trivial projects

Recap Week 12

- **ABIs define how programs interact on a binary level**
  - Names of structures and functions
  - Calling conventions
  - Instruction sets
- **C++ does not define any specific ABI**
  - Because they are tightly coupled to the platform
- **ABIs change between OSes, compiler versions, library versions, etc.**
- **Case in point:**
  - GCC changed its ABI from:
    - Version 2.95 to 3.0
    - 3.0 to 3.1
    - 3.1 to 3.2
    - 3.3 to 3.4
    - 5.0 to 5.1
    - …
  - Different standard library implementations are usually incompatible

Hourglass Interface



What Parts of C++ Can Be Used in an `extern "C"` Interface?

- **Functions, but not templates or variadic**
  - No overloading in C!
- **C primitive types (`char`, `int`, `double`, `void`)**
- **Pointers, including function pointers**
- **Forward-declared structs**
  - Pointers to those are opaque types!
  - Are used for abstract data types
- **Enums (unscoped - without class or base type!)**
- **If using from C must embrace it with `extern "C"` when compiling it with C++**
  - Otherwise names do not match, because of mangling

Wizard.h

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct Wizard * wizard;
typedef struct Wizard const * cwizard;
wizard createWizard(char const * name,
                    error_t * out_error);
void disposeWizard(wizard toDispose);

// ...
// Comments are ok too, as the preprocessor
// eliminates them anyway

#ifdef __cplusplus
}
#endif
```

```
extern "C" {
void printInt(int number);
}
```

```
public interface CplaLib extends Library {
    CplaLib INSTANCE = (CplaLib) Native.load("cpla", CplaLib.class);

    void printInt(int number);
}
```

- **Function names and parameter types must match**
  - However: The types are not validated! Even at runtime! (They are not part of the signature in C!)
- **Parameter names don't matter**

- **Plain (non-opaque) struct types must inherit from `Structure`**
  - You must override `getFieldOrder()`
  - Can use the tag-interface `Structure.ByValue`
- **Opaque struct types should inherit from `Pointer`**
  - Provide a constructor using the `create…()` function
- **Managing lifetime is not trivial**
  - Using `dispose…()` API functions in finalizers is not recommended
  - Either provide a dispose method on you Java type
  - Or implement `AutoClosable` and use you objects with try-with-resources

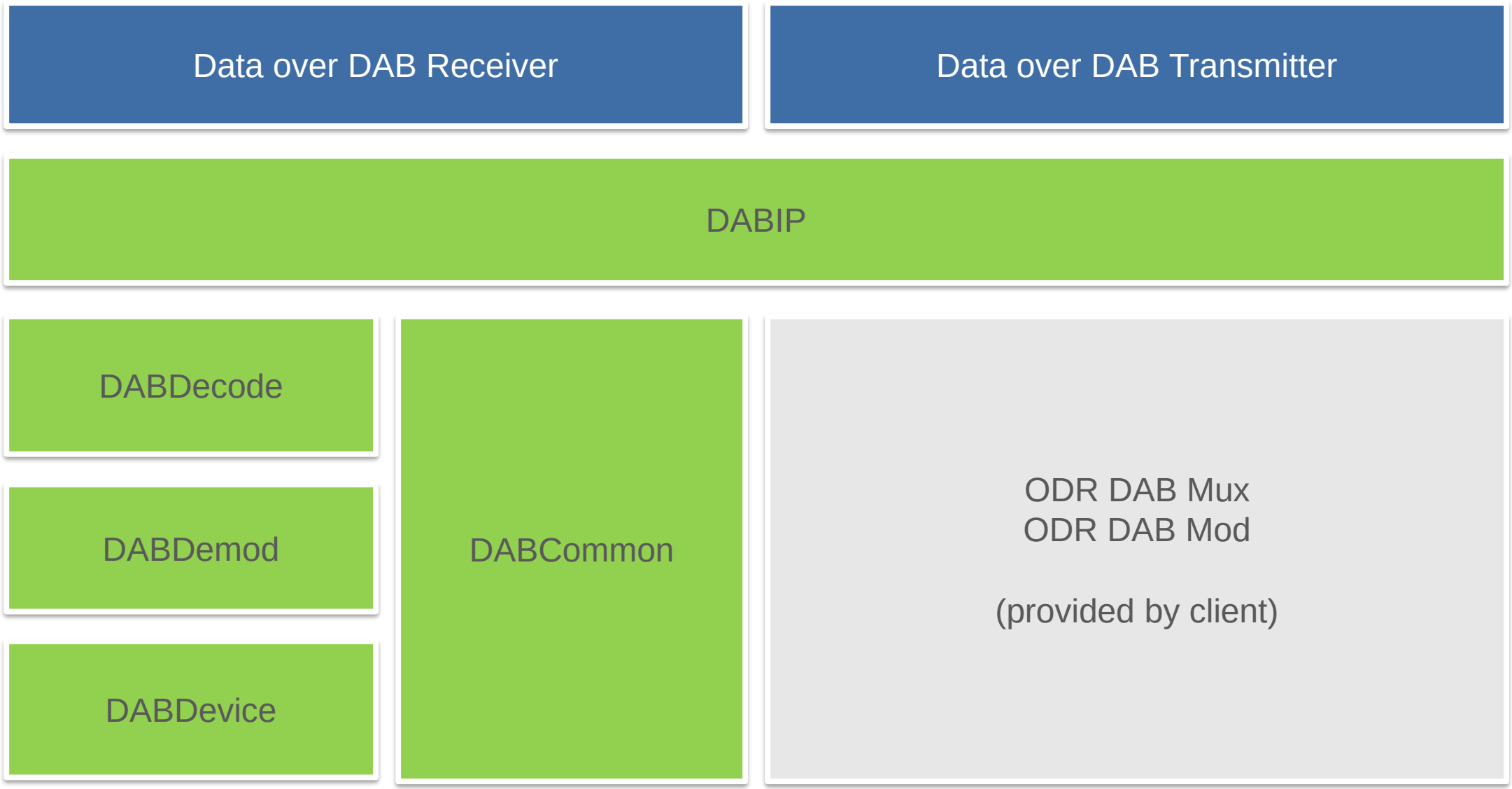
Motivation


Cevelop
++
Your C++ deserves it


OST
Ostschweizer
Fachhochschule

- **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?**

Motivating Example



- **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

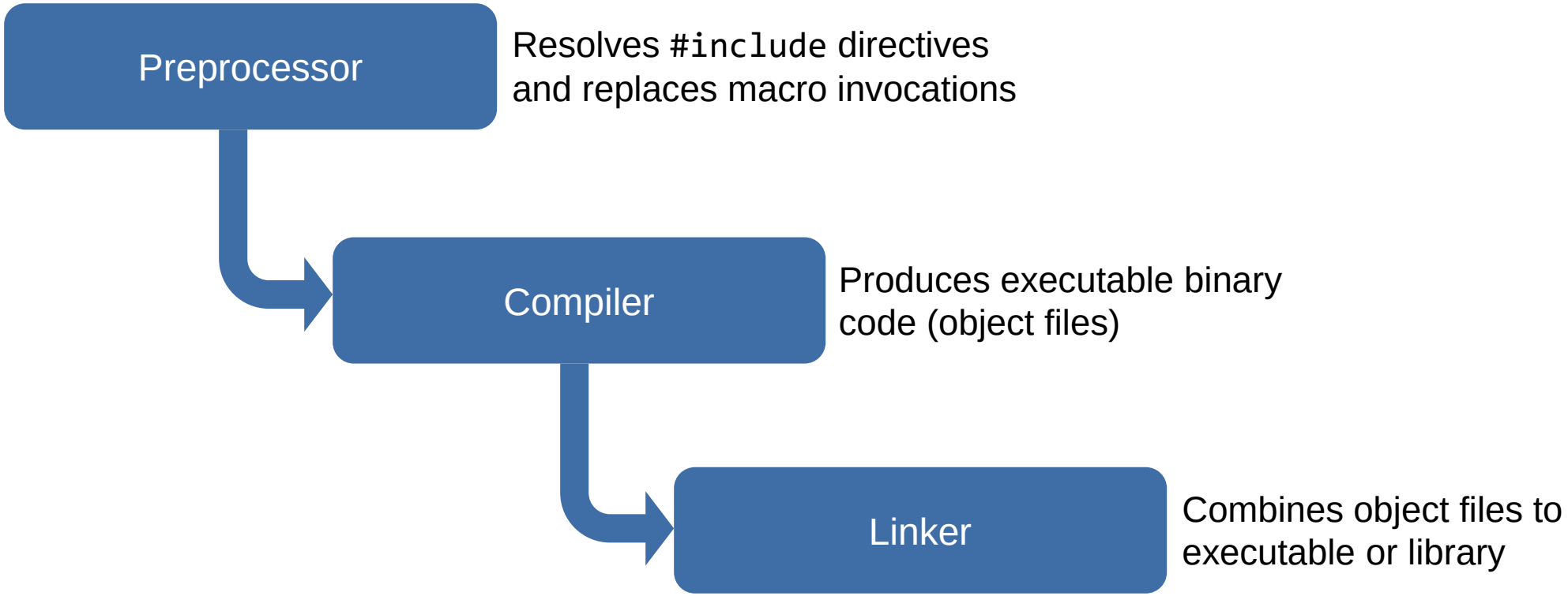
- **Build automation and Reproducibility**
  - No “Wait for <insert name here> to build the package”!
  - No “Builds on my machine”!
- **Productivity**
- **Project Layout / Maintainability**
  - Independent code should live in a separate project
  - Link- and compile-time dependencies must be easy to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

- **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>
```

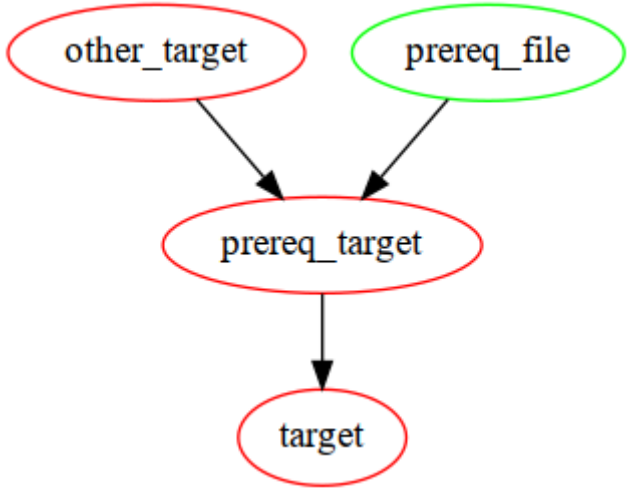
- **Building non-trivial projects is an old problem**
- **There are plenty of existing tools:**
  - GNU make
  - Scons
  - Ninja
  - CMake
  - autotools
  - …
- **Don’t reinvent the wheel! Other people are doing that for you…**

- **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.)

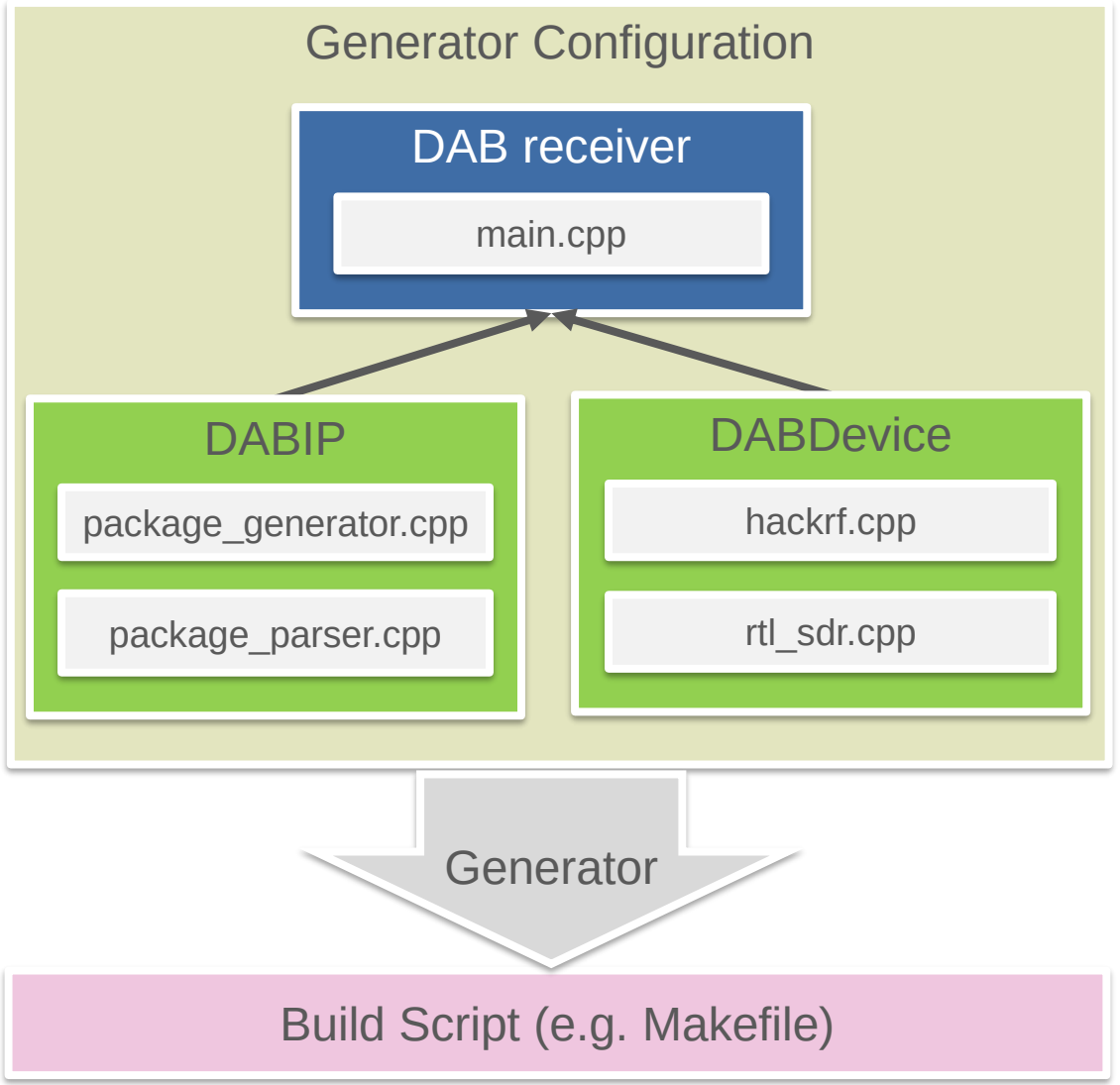
- **Well-known tool to build all kinds of projects**
  - Many IDEs “understand” make projects
- **Workflow description in Makefile via “Target” rules**
  - Each target may have one or more prerequisites…
  - …and execute one or more commands to…
  - …generate one or more results
- **Targets are then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



- **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?

- **`cmake_minimum_required(…)` sets the minimum required CMake version**
  - Implicitly defines available feature set
- **`project(…)` command defines …**
  - … the name of our project
  - … which languages we use
- **`add_executable(…)` defines binaries**
- **`target_compile_features(…)` defines which language features are used by the target**
- **`set_target_properties(…)` defines additional target properties**

```
cmake_minimum_required(VERSION "3.12.0")

project("my_app" LANGUAGES CXX)

add_executable("my_app"
  "main.cpp"
)

target_compile_features("my_app" PRIVATE
  "cxx_std_17"
)

set_target_properties("my_app" PROPERTIES
  CXX_STANDARD_REQUIRED YES
  CXX_EXTENSIONS NO
)
```

- **`add_library(…)` defines libraries**
  - Defaults to static libraries
  - Can be overridden at configuration time
    - `cmake -D BUILD_SHARED_LIBS=YES`
- **All features, include paths, dependencies should be `PUBLIC`**

```
cmake_minimum_required(VERSION "3.12.0")

project("my_lib“ LANGUAGES CXX)

add_library("my_lib"
  "lib.cpp"
)

target_compile_features("my_lib" PUBLIC
  "cxx_std_17"
)

set_target_properties("my_lib" PROPERTIES
  CXX_STANDARD_REQUIRED YES
  CXX_EXTENSIONS NO
)
```

```
cmake_minimum_required(VERSION "3.12.0")

project("my_lib" LANGUAGES CXX)

add_library("my_lib" "lib.cpp")

target_compile_features("my_lib" PUBLIC
  "cxx_std_17"
)

set_target_properties("my_lib" PROPERTIES
  CXX_STANDARD_REQUIRED YES
  CXX_EXTENSIONS NO
)

add_executable("my_app" "app.cpp")

target_link_libraries("my_app" PRIVATE "my_lib")
```

Defining Required Features (`target_compile_features`)

- **`target_compile_features` is used to define the language features required by the target**
  - C++ Standard:
    - `cxx_std_14`
    - `cxx_std_17`
    - …
  - C++ Features:
    - `cxx_range_for`
    - …
  - Can be `PUBLIC` or `PRIVATE`
  - Prefer requiring standards rather than specific features!

- **`target_include_directories` is used to define the include search path of the target**
  - Can define paths as being system includes or not
    - Default is non-system include path
    - Specify SYSTEM to define path as being a system include path (includes using <…>)
  - Can be `PUBLIC` or `PRIVATE`

Defining Library Dependencies (`target_link_libraries`)

- **`target_link_libraries` is used to define libraries required by a target**
  - Can be …
    - a target built by the current project
    - a target built by a sub-project
    - a system library (e.g. “stdc++fs”)
  - Can be `PUBLIC` or `PRIVATE`
  - Applies `PUBLIC` features/dependencies/includes of the library

- **Variables can be defined using `set(VAR_NAME VALUE)`**
- **Variables are referenced using `${VAR_NAME}`**
- **CMake defines certain global variables. E.g.:**
  - `PROJECT_NAME` – The name of the current project
  - `PROJECT_SOURCE_DIR` – The “root” source directory of the current project
  - `PROJECT_BINARY_DIR` – The “root” output directory of the current project
- **Can be used in place of concrete values. E.g.:**
  - `add_executable(${PROJECT_NAME} “source1.cpp” “source2.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
```

```
cmake_minimum_required(VERSION "3.12.0")

project("answer" LANGUAGES CXX)

enable_testing()

add_library("${PROJECT_NAME}"
  "answer.cpp"
)

add_executable("test_runner"
  "Test.cpp"
)

target_link_libraries("test_runner" PRIVATE
  "answer“
)

target_include_directories("test_runner" SYSTEM PRIVATE
  "cute“
)

add_test("tests" "test_runner")
```

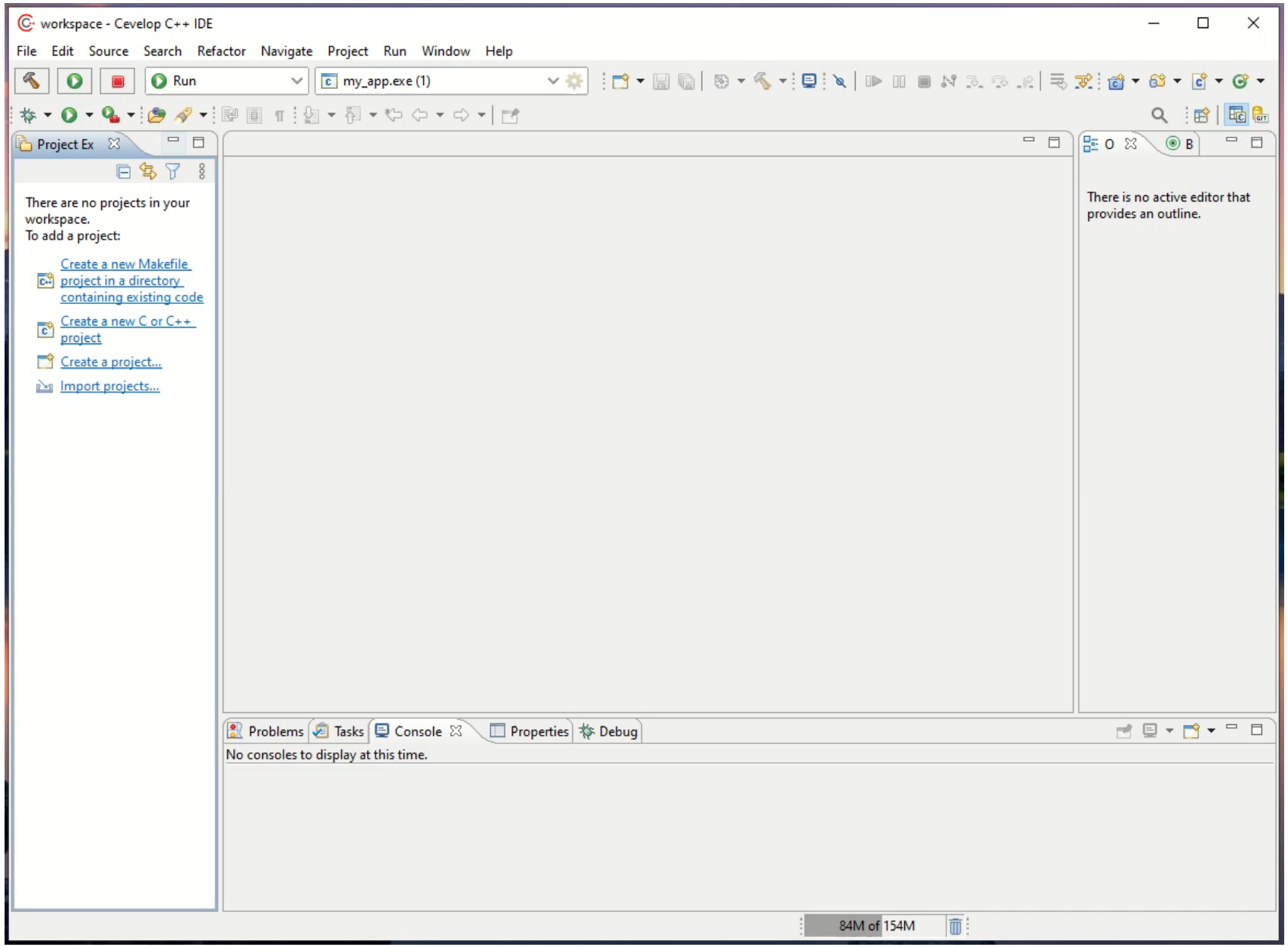
- **CMake makes building project easy**
  - Platform specifics are handled behind the scenes
  - Declare **what** you want, not **how** to create it
  - CTest allows you to run your unit tests
- **You can choose what kind of build scripts you want:**

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

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

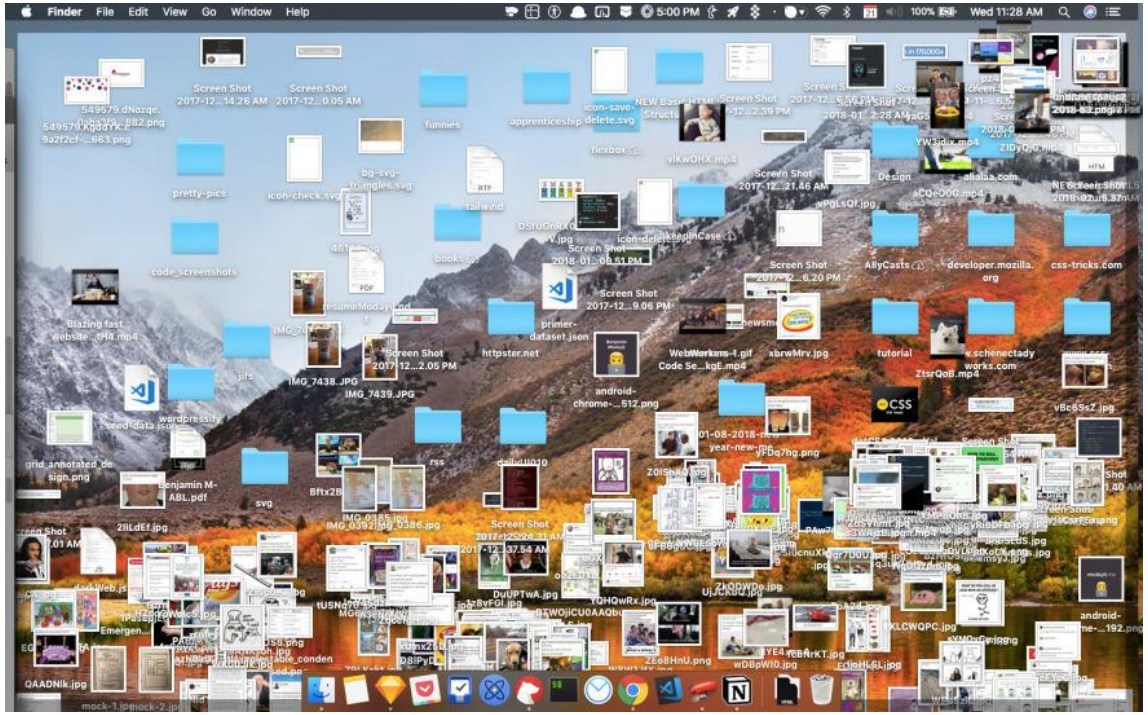
- **Declare the version of CMake required for your project**

```
cmake_minimum_required(VERSION 3.14.0)
```


workspace - Cevelop C++ IDE
File Edit Source Search Refactor Navigate Project Run Window Help
Run
my_app.exe (1)
Project Ex
There are no projects in your workspace.
To add a project:
Create a new Makefile project in a directory containing existing code
Create a new C or C++ project
Create a project...
Import projects...
There is no active editor that provides an outline.
Problems Tasks Console Properties Debug
No consoles to display at this time.
84M of 154M

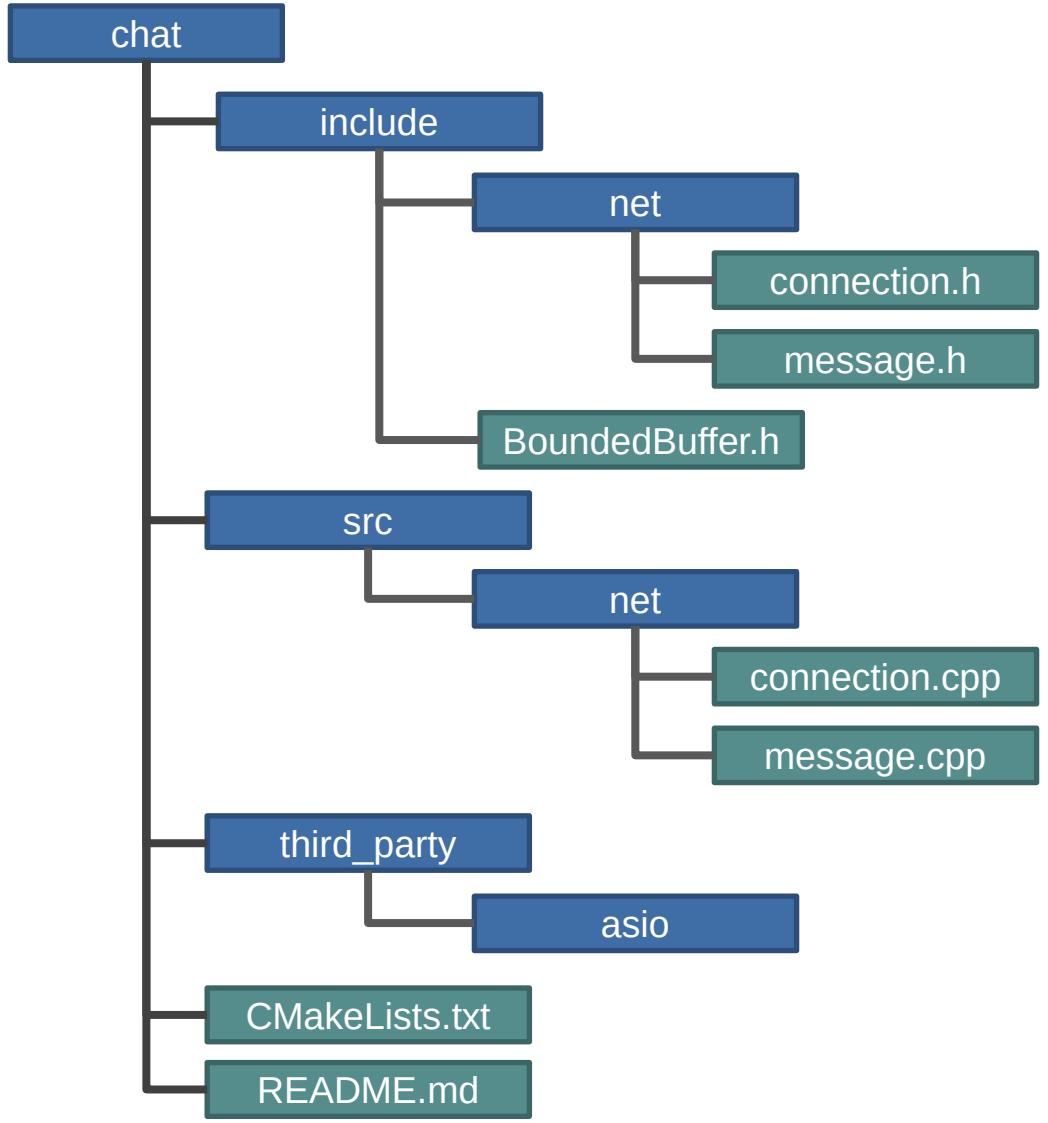
Project Layout

- **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