CMake Training

Session 1

CMake and Building C++

JAMIE SMITH

Intro: Why we Care about Build Systems

- Build systems are not exactly a glamorous topic.
- They are not going to win you any job offers or grab people's attention on your resume.
- I can understand if you want to leave this long and complex training and go learn about something fun and flashy like machine learning or React or Swift
- HOWEVER: build systems are one of the most key parts of *any* software project. You cannot write any serious code without a good build system supporting you.
- If you can stick it out through this series of 5 lectures, you will know more about C++ build systems than 75% of professional software engineers (if Qualcomm is any judge)
    - Great way to earn points at any new workplace!

Qualcomm Experience

- When I was at Qualcomm, the build system we had to work with took almost an hour to build an entire OS image and flash it onto the target device
- Instead of using that, I spent my first week on the project writing a new build system and incremental upload tool in CMake
- This made incremental builds take 12 seconds instead of >30 minutes!
- This let me iterate quickly instead of taking ages to create and test new builds
- People were always like “how are you making progress so fast on this project?”
- Answer: build systems!

Part 1

How C++ Code is Built

Exercise 1: Manually Building C++

- Before we can start learning about CMake, we must first review how C++ code is built
- We’ll practice by compiling a simple program ourselves on the command line
- This program has two parts:
  - MovingQuadReg.h/cpp: implements a quadratic regression that converts a set of data points into a parabolic equation
  - test_regression.cpp: test program for MovingQuadReg

Building C++ Code

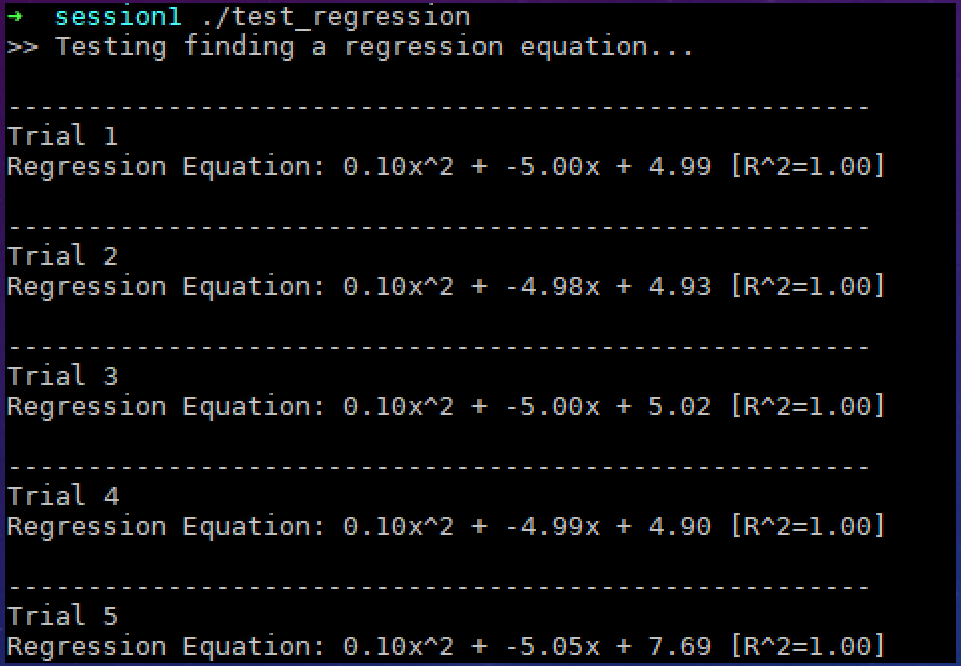
- Two basic steps in building C++ programs: *compilation* and *linking*.
- Compilation: Functions in a source file are turned into machine code.
  - Machine code stored in an *object file* (.o)
  - Errors in the *source code* are detected at this time.
- Linking: Files containing object code are merged together and a single program is created.
  - This generates an executable you can run
  - Errors in the *complete program* are detected at this time.
  - In particular, missing functions will be detected with an “undefined symbol” error.

Exercise 1 Steps

1. Open a terminal to CMakeTraining/exercises/section1
2. Type and run the following commands:
3. `g++ --std=c++11 -c quad_reg/MovingQuadReg.cpp -o MovingQuadReg.o`
4. `g++ --std=c++11 -I quad_reg -c test_regression.cpp -o test_regression.o`
5. `g++ test_regression.o MovingQuadReg.o -o test_regression`
6. `./test_regression`

Exercise 1 Results

Should see program run:



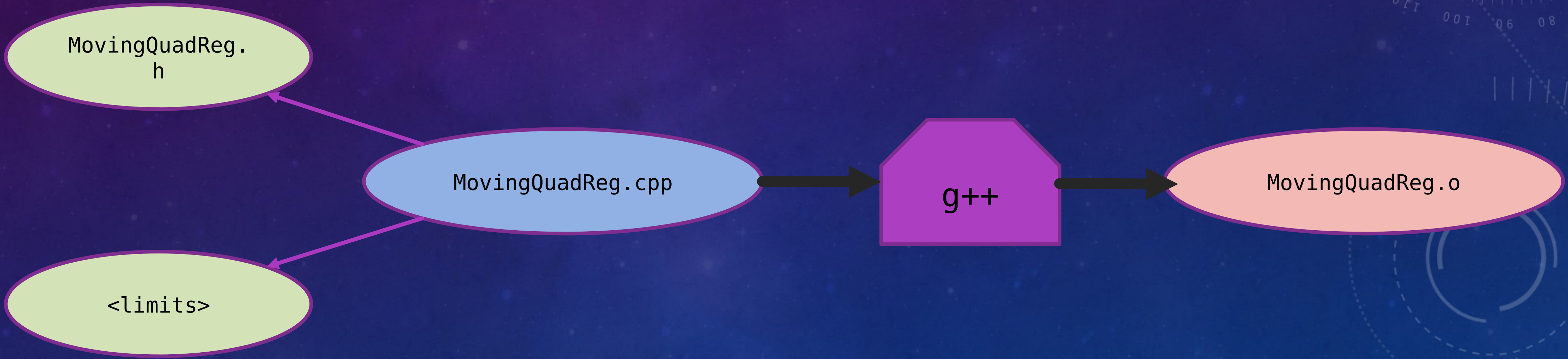
```
➜  session1 ./test_regression
>> Testing finding a regression equation...

-------------------------------------------------------
Trial 1
Regression Equation: 0.10x^2 + -5.00x + 4.99 [R^2=1.00]

-------------------------------------------------------
Trial 2
Regression Equation: 0.10x^2 + -4.98x + 4.93 [R^2=1.00]

-------------------------------------------------------
Trial 3
Regression Equation: 0.10x^2 + -5.00x + 5.02 [R^2=1.00]

-------------------------------------------------------
Trial 4
Regression Equation: 0.10x^2 + -4.99x + 4.90 [R^2=1.00]

-------------------------------------------------------
Trial 5
Regression Equation: 0.10x^2 + -5.05x + 7.69 [R^2=1.00]
```

Exercise 1: What did we do?

- First we compiled each cpp into an object (.o) file
- Then we linked the object files together into an executable

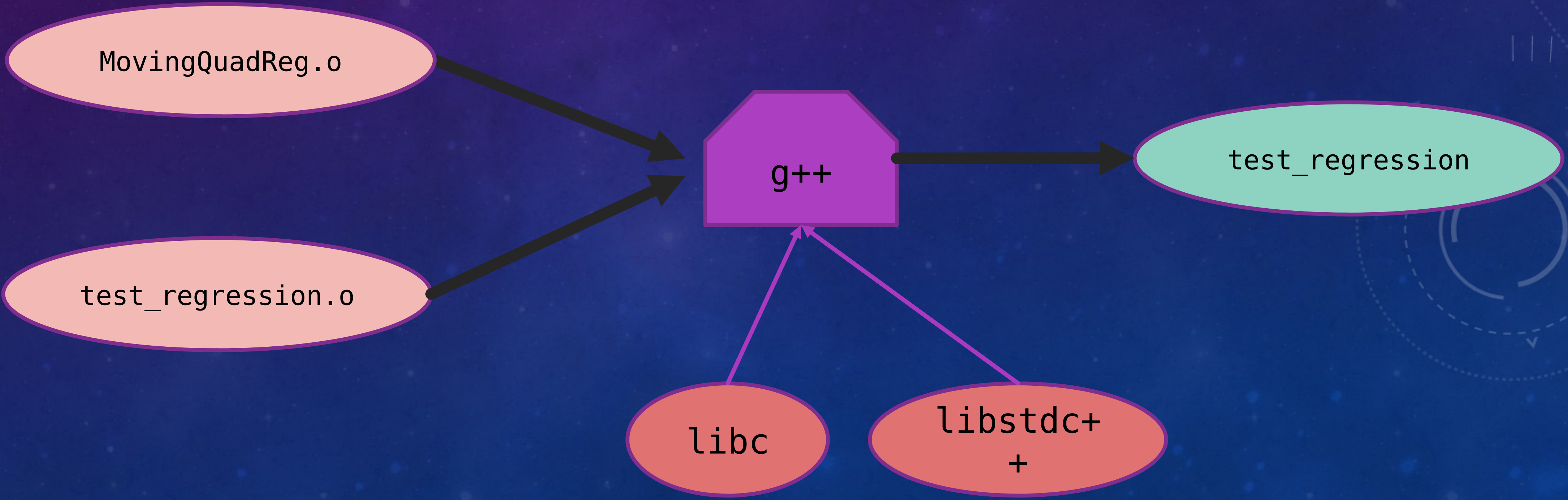
Compile Step

The compiler processes the entire source code for a file (including any headers you include) and generates the equivalent machine code.



Link Step

The compiler combines multiple .o files together and adds additional libraries to create the final executable.



Incremental Builds

- Why does each cpp get compiled into a separate object file?
- Build speed, that’s why.
- C++ files can take a significant amount of time to compile, especially once you have many files that contain lots of code and templates.
- Compiling each cpp file into its own .o file means that instead of rebuilding all sources when you change something, you just rebuild that one .o file, and then redo the final linking.
- This is significantly faster!

Common Compile Options

Option	Function	Variants
-Wall	Enables all standard warnings, so the compiler will tell you about code it thinks is suspicious	-W<name> and –Wno-<name> allow you to enable or disable specific warnings
--std=c++11	Tells the compiler to use C++11	Can also use newer or older C++ standards such as C++98 and C++17. C++11 is the most recent standard available across almost all systems.
-g	Generates debugging information so that you can view stacktraces in your code with GDB and Valgrind	-g3 generates even more debugging info for more detailed views, at the cost of making your code larger.
-O0	Disables all optimizations, which allows you to step through code normally in the debugger.	-Og only enables optimizations which do not affect debugging. I have had mixed results with it though.
-O2	Enables most optimizations, makes code run faster.	-O3: optimize for speed at the cost of size. -Os: optimize for size at the cost of speed.

Common Compile Options (continued)

Option	Function	Variants
`-I<folder>`	Adds a directory to the header search path	-include includes a specific header file at the top of each cpp file
`-o <name>`	Specifies the name of the output file.	If not given, by convention the result is named “a.out”
`-D<var>`	Defines a preprocessor definition with the given name	-D<var>=<value> gives the definition a specific value.

To –c or not to -c

- How do you control whether the compiler does compilation or linking? The much-rumored -c flag.
- If given -c, the compiler will always create a single .o file from a single source file.
- If not given –c, then the compiler will always create an executable using all of the files it's given.
    - This always involves linking, and may involve compilation too if you pass in .cpp files
- Note: there are other types things you can link besides executables, such as static libraries (.a) and shared libraries (.so). We'll cover those in session 2!

Part 1 Review

- Building C++ code on the command line is possible, but tedious.
- Two major steps, compilation and linking.
- Compile options used to adjust compiler behavior
  - Optimization
  - Code standard
  - Include directories
  - Input and output files

Part 2

My First CMakeLists.txt

Why use CMake?

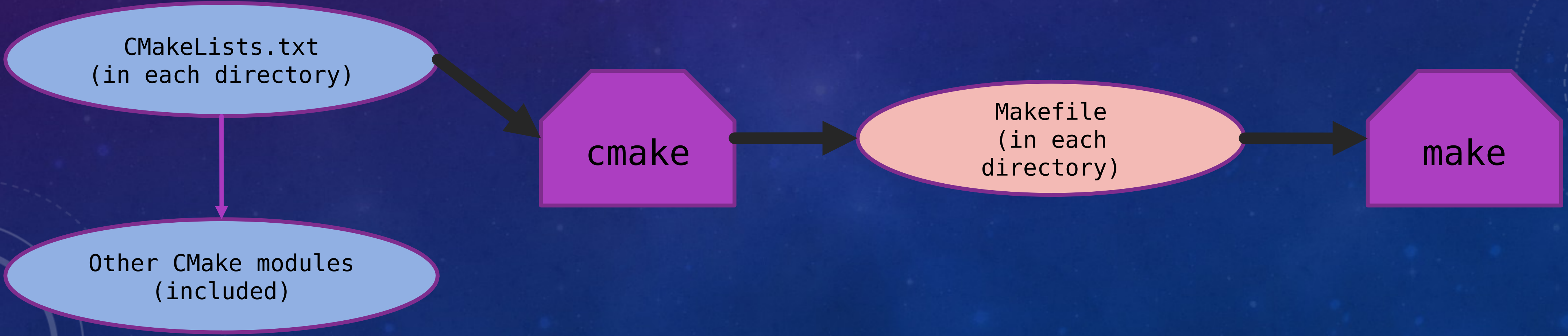
- There are many build systems available. Why am I teaching CMake?
- *Cross-platform.* Generates buildfiles for multiple different build tools on all major platforms
- *Convenient for developers*. CMake projects can be loaded into may different IDEs, such as CLion, Xcode, Visual Studio, and VS Code.
- *Capable*. Can use custom functions and scripts to complete almost any arbitrarily complicated build task. Can do just about anything that you can do with Makefiles, unlike many other build systems that are more restrictive.
- *Broad community support*. Currently the single most popular build system for up-to-date C++ projects. Lots of help and useful libraries available on the Internet. Prewritten modules for loading many libraries.

Why not use CMake?

- Sadly, there are some CMake haters out there. Haters gonna hate, but they do have some valid points to keep in mind.
- *The syntax is weird*. CMake’s syntax is somewhat similar to bash scripts, but is also its own thing. Escaping special characters is often a big issue.
- *Lots of history*. Often the CMake developers deprecated an old way of doing something and introduced a new way. Many projects are still using the old ways.
- *Ugly*. CMake code to do simple things can sometimes be verbose, hard to maintain, and bloated, especially if older functions are used.
    - Can be prevented with modern coding practices.
- *Esoteric*. While common functionality is generally well documented, docs for more advanced topics like find modules, system introspection, and build rules are… spotty.
    - Many of these things covered in this course!

What CMake Does

- We are now going to create our very first buildfile for test_regression with CMake.
- First, we need to understand how CMake processes this file.
- CMake is a *build system generator*. It reads files in its own language and outputs buildfiles in one of several other languages. We will use make for this course.
- These buildfiles then take care of building the actual code.



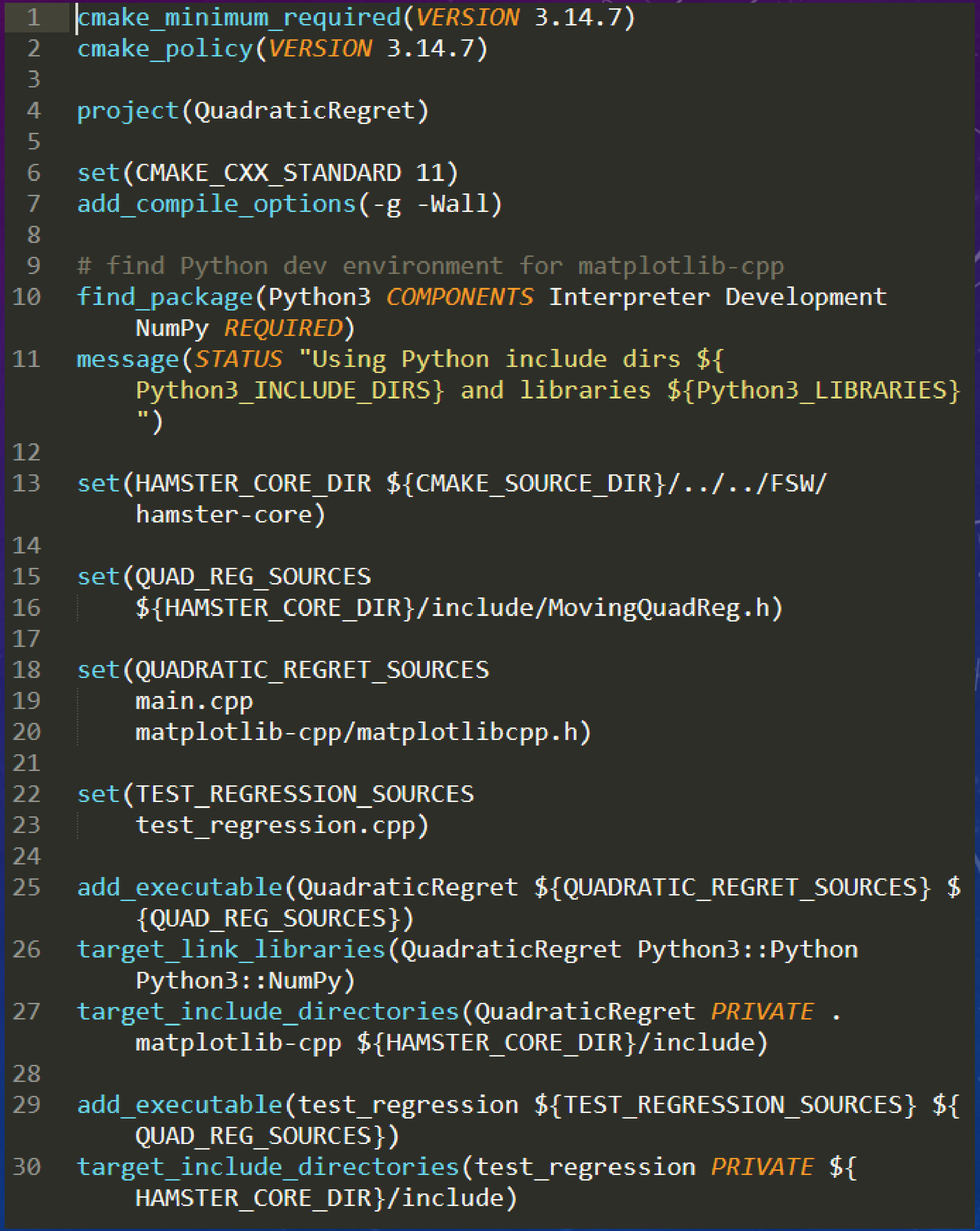
Functions of CMakeLists.txt

- The top-level buildfile performs some important tasks:
  - Initializing CMake (this session)
  - Setting compilation options (this session)
  - Inspecting the system and finding libraries (session 3)
- The buildfiles in each directory take care of building the code there:
  - Setting local compile options (session 2)
  - Building code into various targets (this session)
  - Linking targets to the libraries they depend on (sessions 2 and 3)
- This is just an overview! We will go into each of these in detail

CMake Syntax

A random example ->

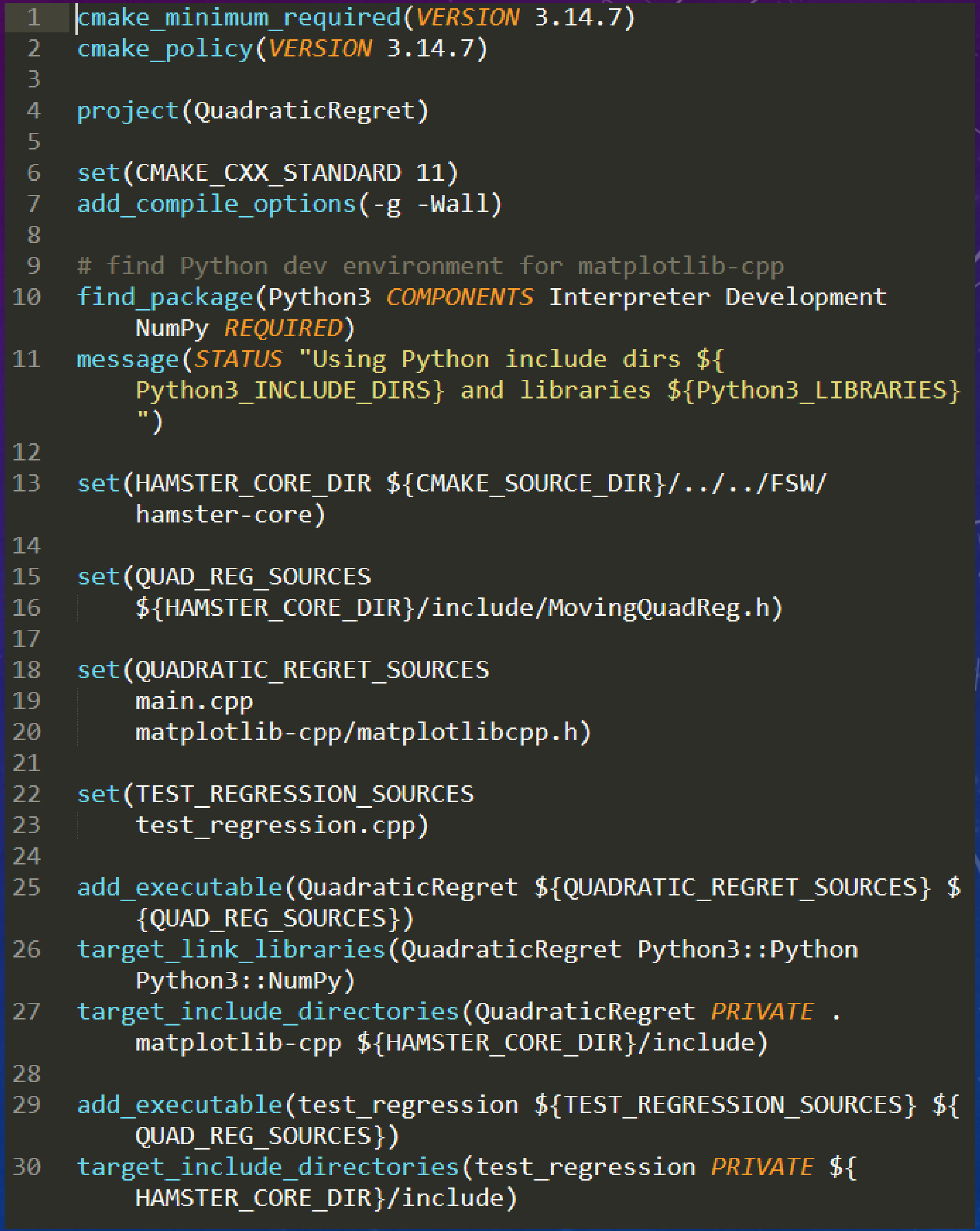
- There’s a lot to go over!
- For now we will start with the basics.
- Each line contains one command.
- Command calls continue across lines until the next closing parenthesis
- Variables can be evaluated like “${VAR}”
- Whitespace around commands is not significant, but it is used to separate command arguments
- All command arguments are string literals. Quotes used when literals contain whitespace.



```
cmake_minimum_required(VERSION 3.14.7)
cmake_policy(VERSION 3.14.7)

project(QuadraticRegret)

set(CMAKE_CXX_STANDARD 11)
add_compile_options(-g -Wall)

# find Python dev environment for matplotlib-cpp
find_package(Python3 COMPONENTS Interpreter Development
    NumPy REQUIRED)
message(STATUS "Using Python include dirs ${
    Python3_INCLUDE_DIRS} and libraries ${Python3_LIBRARIES}
    ")

set(HAMSTER_CORE_DIR ${CMAKE_SOURCE_DIR}/../../FSW/
    hamster-core)

set(QUAD_REG_SOURCES
    ${HAMSTER_CORE_DIR}/include/MovingQuadReg.h)

set(QUADRATIC_REGRET_SOURCES
    main.cpp
    matplotlib-cpp/matplotlibcpp.h)

set(TEST_REGRESSION_SOURCES
    test_regression.cpp)

add_executable(QuadraticRegret ${QUADRATIC_REGRET_SOURCES} $
    {QUAD_REG_SOURCES})
target_link_libraries(QuadraticRegret Python3::Python
    Python3::NumPy)
target_include_directories(QuadraticRegret PRIVATE .
    matplotlib-cpp ${HAMSTER_CORE_DIR}/include)

add_executable(test_regression ${TEST_REGRESSION_SOURCES} ${
    QUAD_REG_SOURCES})
target_include_directories(test_regression PRIVATE ${
    HAMSTER_CORE_DIR}/include)
```

CMake Variables

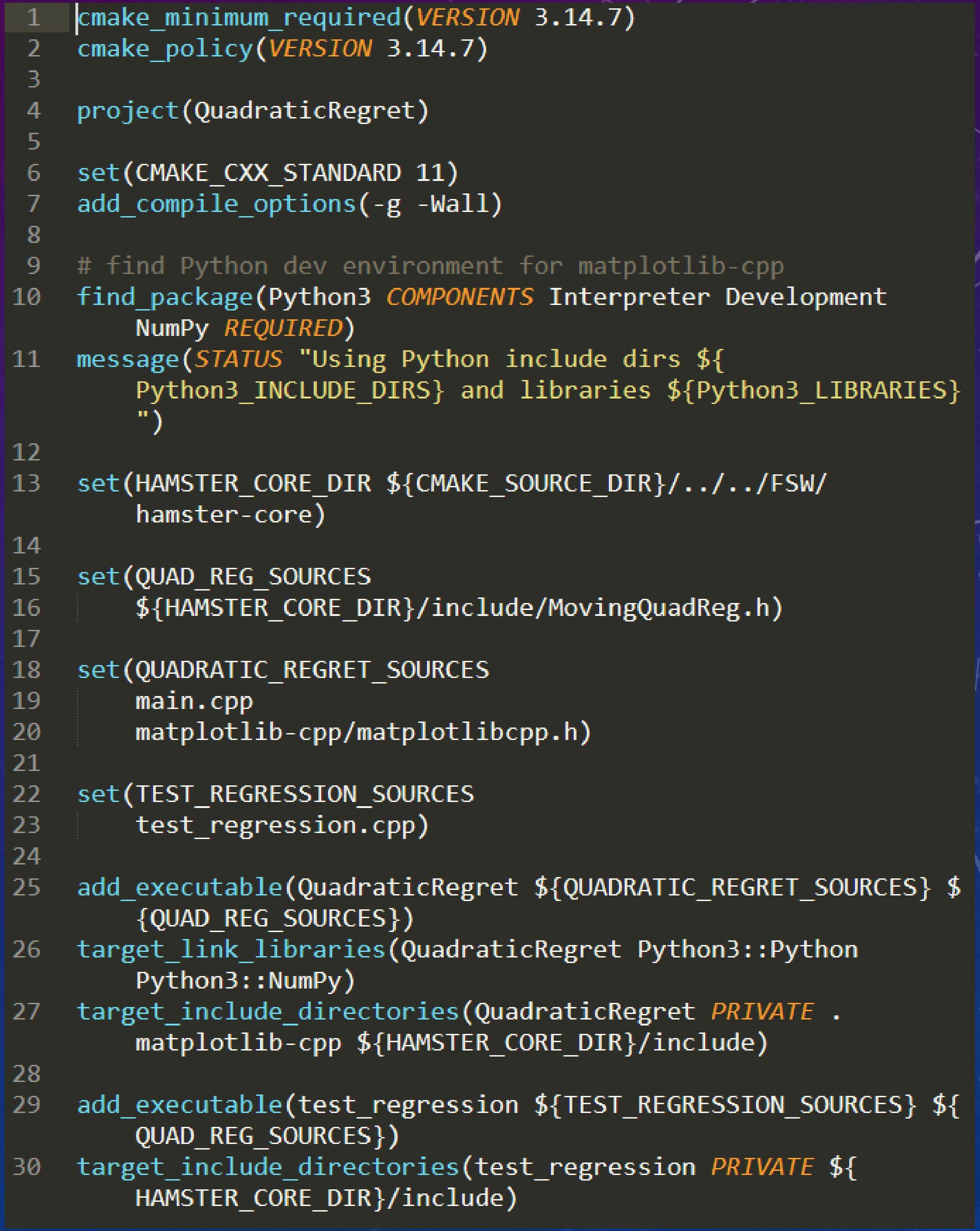
- All data is strings
- Lists stored as strings separated by semicolons
    - God forbid if you need to have semicolons in a string!
- Data is converted from strings to other types, e.g. Boolean and integer, where needed
- set() command used to assign a value to a variable
    - Multiple arguments passed to set() are converted into a list
- By convention, variable names are all caps. Some people like to use lowercase variables for local variables.



```cmake
cmake_minimum_required(VERSION 3.14.7)
cmake_policy(VERSION 3.14.7)

project(QuadraticRegret)

set(CMAKE_CXX_STANDARD 11)
add_compile_options(-g -Wall)

# find Python dev environment for matplotlib-cpp
find_package(Python3 COMPONENTS Interpreter Development NumPy REQUIRED)
message(STATUS "Using Python include dirs ${Python3_INCLUDE_DIRS} and libraries ${Python3_LIBRARIES}")

set(HAMSTER_CORE_DIR ${CMAKE_SOURCE_DIR}/../../FSW/hamster-core)

set(QUAD_REG_SOURCES
    ${HAMSTER_CORE_DIR}/include/MovingQuadReg.h)

set(QUADRATIC_REGRET_SOURCES
    main.cpp
    matplotlib-cpp/matplotlibcpp.h)

set(TEST_REGRESSION_SOURCES
    test_regression.cpp)

add_executable(QuadraticRegret ${QUADRATIC_REGRET_SOURCES} ${QUAD_REG_SOURCES})
target_link_libraries(QuadraticRegret Python3::Python Python3::NumPy)
target_include_directories(QuadraticRegret PRIVATE . matplotlib-cpp ${HAMSTER_CORE_DIR}/include)

add_executable(test_regression ${TEST_REGRESSION_SOURCES} ${QUAD_REG_SOURCES})
target_include_directories(test_regression PRIVATE ${HAMSTER_CORE_DIR}/include)
```

CMake Commands

- Two different types of commands, keyword arguments and positional.
- Keyword arguments: Use specific all-caps keywords to separate arguments.
  - Keywords can take no arguments, one argument, or a list.
- Positional arguments: Specific arguments are passed in a specific order without keywords
- *Look at documentation!* Arguments are hard to remember but docs are generally very helpful.
- Unlike variables, function names are not case-sensitive. Lowercase by convention.
- It is possible to write your own custom functions, we will go into that later.



```cmake
cmake_minimum_required(VERSION 3.14.7)
cmake_policy(VERSION 3.14.7)

project(QuadraticRegret)

set(CMAKE_CXX_STANDARD 11)
add_compile_options(-g -Wall)

# find Python dev environment for matplotlib-cpp
find_package(Python3 COMPONENTS Interpreter Development NumPy REQUIRED)
message(STATUS "Using Python include dirs ${Python3_INCLUDE_DIRS} and libraries ${Python3_LIBRARIES}")

set(HAMSTER_CORE_DIR ${CMAKE_SOURCE_DIR}/../../FSW/hamster-core)

set(QUAD_REG_SOURCES
    ${HAMSTER_CORE_DIR}/include/MovingQuadReg.h)

set(QUADRATIC_REGRET_SOURCES
    main.cpp
    matplotlib-cpp/matplotlibcpp.h)

set(TEST_REGRESSION_SOURCES
    test_regression.cpp)

add_executable(QuadraticRegret ${QUADRATIC_REGRET_SOURCES} ${QUAD_REG_SOURCES})
target_link_libraries(QuadraticRegret Python3::Python Python3::NumPy)
target_include_directories(QuadraticRegret PRIVATE . matplotlib-cpp ${HAMSTER_CORE_DIR}/include)

add_executable(test_regression ${TEST_REGRESSION_SOURCES} ${QUAD_REG_SOURCES})
target_include_directories(test_regression PRIVATE ${HAMSTER_CORE_DIR}/include)
```

CMake Directory Structure

- Unlike Make, CMake makes it very straightforward to keep your compiled files separate from your source directory
- Helps keep your Git repositories nice and clean!
- Three directories to remember:
  - Source dir: where your source code is
  - Binary dir: where the binaries are being stored
  - Install dir: where the binaries will be installed when you run “make install”
- When running cmake, you always run in the binary dir and pass CMake the path to the source dir

Directory Variables

These directories are accessed via special predefined variables

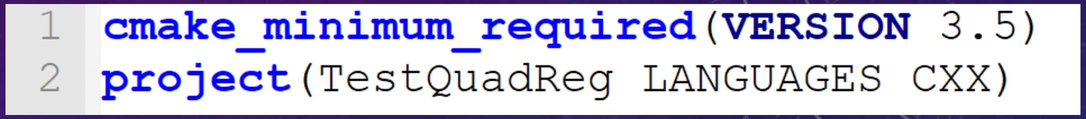
Variable	Contents	Example Value (for session1/quad_reg/CMakeLists.txt)
`CMAKE_SOURCE_DIR`	Top-level source folder of your project	`<…>/session1`
`CMAKE_CURRENT_SOURCE_DIR`	Source folder of the current CMakeLists.txt	`<…>/session1/quad_reg`
`CMAKE_BINARY_DIR`	Top-level binary dir of your project	`<…>/session1/build/`
`CMAKE_CURRENT_BINARY_DIR`	Binary dir for the current CMakeLists.txt	`<…>/session1/build/quad_reg`
`CMAKE_INSTALL_PREFIX`	Directory that build products will be installed to	`/usr/local`

Exercise 2: QuadRegTest

- We are now going to create a simple CMakeLists.txt
- Create a new one in the session1 folder and open it in your text editor

Initialization code

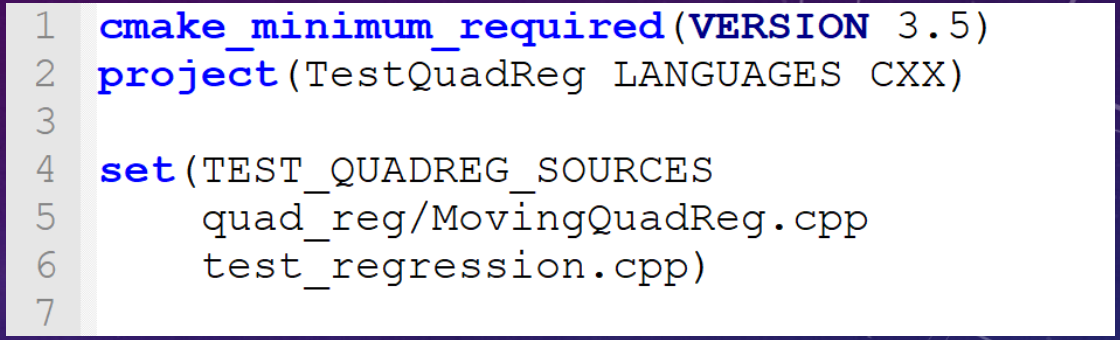
- These two statements are needed as the first lines of all CMake build systems
- cmake_minimum_required()
  - Sets minimum version needed to run the buildfiles.
  - Disables backwards compatibility for versions older than the given one.
- project()
  - Sets the name of the project (e.g. for display in IDEs).
  - Tells CMake which compilers to search for.
- NOTE: C++ is called "CXX" in CMake, probably so it can be used as a valid filesystem path



```cmake
cmake_minimum_required(VERSION 3.5)
project(TestQuadReg LANGUAGES CXX)
```

Source list

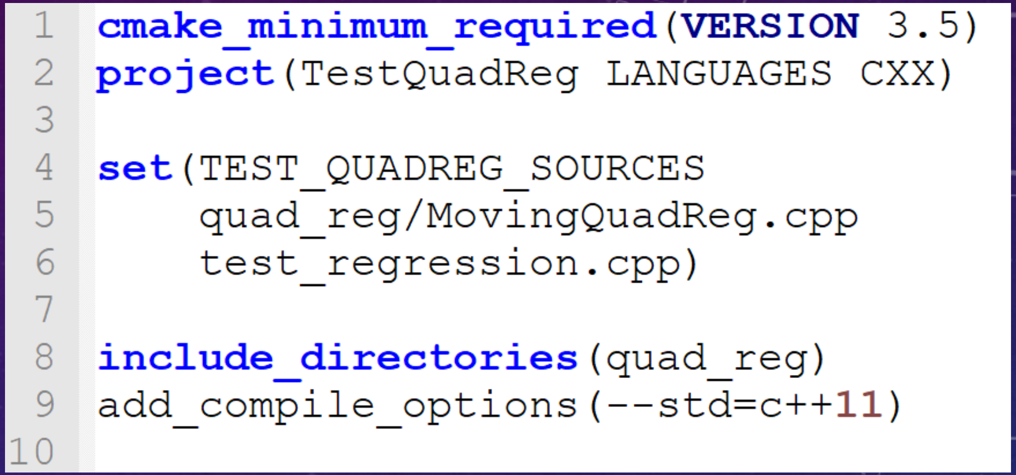
- Creates a variable (TEST_QUADREG_SOURCES) containing the paths to both files
    - Paths in CMake are almost always interpreted relative to the source directory
- I recommend always making lists of source files at the top of the file.
- This is dead simple to the reader and makes it easy for the non-CMake-literate to add source files



```cmake
cmake_minimum_required(VERSION 3.5)
project(TestQuadReg LANGUAGES CXX)

set(TEST_QUADREG_SOURCES
    quad_reg/MovingQuadReg.cpp
    test_regression.cpp)

```

Basic compile flags

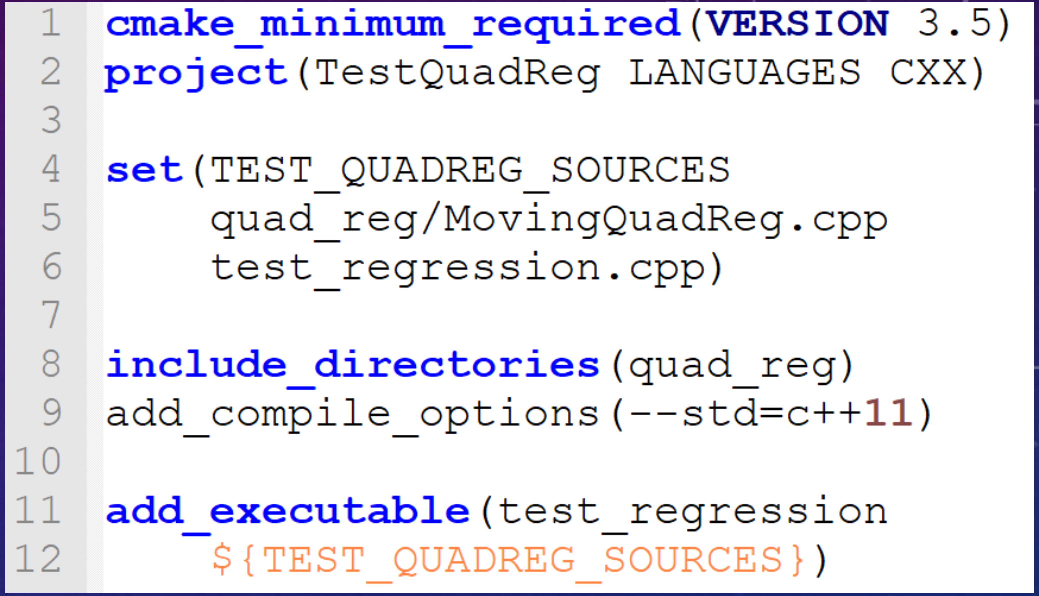
- include_directories()
  - Adds directories to the include path
  - Will be passed to the compiler using the –I option
- add_compile_options()
  - Adds the given options to the compiler flags
- For now we are using functions that operate at the global scope
  - Other ways to do this will be discussed in session 2



```
cmake_minimum_required(VERSION 3.5)
project(TestQuadReg LANGUAGES CXX)

set(TEST_QUADREG_SOURCES
    quad_reg/MovingQuadReg.cpp
    test_regression.cpp)

include_directories(quad_reg)
add_compile_options(--std=c++11)
```

Adding the Executable

- add_executable()
  - Tells CMake to build an executable target with the given name from the given sources
- We use the TEST_QUADREG_SOURCES variable we created earlier

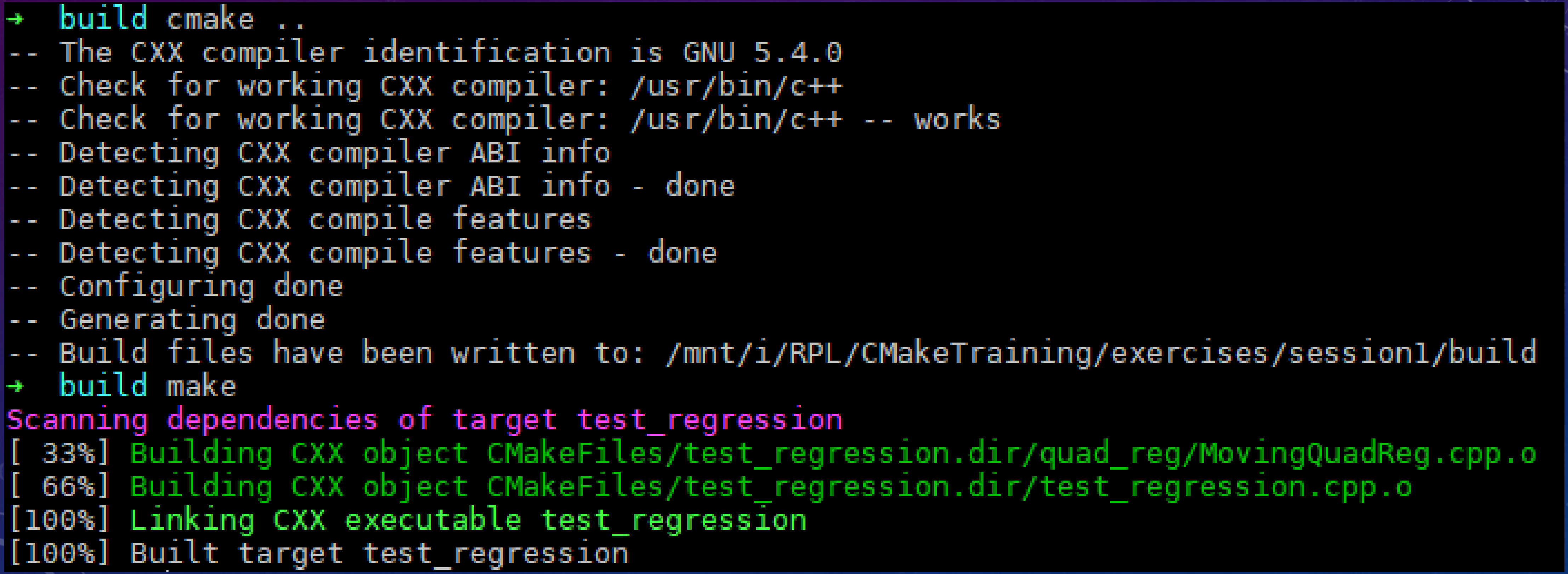


```
cmake_minimum_required(VERSION 3.5)
project(TestQuadReg LANGUAGES CXX)

set(TEST_QUADREG_SOURCES
    quad_reg/MovingQuadReg.cpp
    test_regression.cpp)

include_directories(quad_reg)
add_compile_options(--std=c++11)

add_executable(test_regression
    ${TEST_QUADREG_SOURCES})
```

Running Exercise 2

- `cd session2`
- `mkdir build`
- `cd build`
- `cmake ..`
- `make`

Exercise 2 Output



```
➜  build cmake ..
-- The CXX compiler identification is GNU 5.4.0
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/i/RPL/CMakeTraining/exercises/session1/build
➜  build make
Scanning dependencies of target test_regression
[ 33%] Building CXX object CMakeFiles/test_regression.dir/quad_reg/MovingQuadReg.cpp.o
[ 66%] Building CXX object CMakeFiles/test_regression.dir/test_regression.cpp.o
[100%] Linking CXX executable test_regression
[100%] Built target test_regression
```

Part 2 Review

- Basic CMake syntax
  - Variables, commands, whitespace
- Directory structure
  - source, binary, install
- My First CMakeLists.txt

Session 1 Complete!

- Nice work!
- Any questions?