CMake Training
Session 2
More Advanced CMake

JAMIE SMITH

Intro: A Short History of CMake

- 2000: Kitware contracted to develop a new build system for a bioinformatics program called ITK: a Cross-platform Make
- 2006: Massive KDE project (Linux desktop environment) switches its entire build system to CMake and gives it rave reviews.
- 2014: CMake 3.0 is released, introducing many of the features we use as standard today.
- More good info: https://www.aosabook.org/en/cmake.html

Part 1

Targets and Properties

What is a target?

- A target is anything that CMake can build:
  - Executable
  - Static library
  - Shared library
  - Object library
  - Custom target (we’ll talk about these in session 4)
- The entire purpose of a build system, its reason for existence, is to create targets.

```
add_executable(test_regression
    ${TEST_QUADREG_SOURCES})
```

CMake build systems are structured around targets

- All code files are compiled as part of one target or another
- Dependencies are expressed in terms of targets
  - “Dependency” in CMake means “build X before Y”
- Dependencies are automatically created between an executable and its libraries

Executable targets

- Created by add_executable()
- Create programs that can be run (.exe on Windows, no suffix on Mac)
- On microcontrollers, e.g. MBed, these create images that can be programmed to the chip

Static library targets

- Created by add_library(STATIC)
- Create static libraries of code that can be linked into executables (.lib on Windows MSVC, .a elsewhere)

Object library targets

- Created by add_library(OBJECT)
- Create object libraries: code compiled into .o files but not combined into a single library
- Function very similar to STATIC libraries, used in certain specific cases
  - Most common use: improving performance by only building certain code files once that are needed for multiple targets
- You likely won’t need to use these except for very complicated build systems!

Properties can be used to configure the build further

- What you can’t accomplish through functions in CMake, you accomplish through properties.
- Properties determine the specific details of how CMake builds a target, such as compile flags and link libraries
- Properties can be set on a number of different levels:
  - Global (affects the entire project)
  - Directory (affects the current directory and all subdirectories)
  - Target (affects a specific target)
  - Source (affects a specific source file in *all* targets it’s present in)
- List of all properties (you will use this doc page a LOT): https://cmake.org/cmake/help/latest/manual/cmake-properties.7.html

Properties can be used to add compile flags.

Property	Scope	Function
`INCLUDE_DIRECTORIES`	Directory, Target, Source (since CMake 3.11)	List of directories to add to the include path
`COMPILE_DEFINITIONS`	Directory, Target, Source	List of preprocessor macros to define when compiling the code
`COMPILE_OPTIONS`	Directory, Target, Source (since CMake 3.11)	List of compiler flags to use when compiling the code.
`POSITION_INDEPENDENT_CODE`	Target	Controls whether the code will be built as position independent, which is required when compiling shared libraries.

Setting properties the standard way

- CMake has a number of different functions to set properties, but some of them have hidden gotchas!
  - e.g. not accepting a list as a property value
- Instead, use the one function that can do it all:

set_property

Set a named property in a given scope.

```
set_property(<GLOBAL                      |
              DIRECTORY [<dir>]           |
              TARGET    [<target1> ...]   |
              SOURCE    [<src1> ...]
                        [<TARGET_DIRECTORY ... | DIRECTORY ...>]    |
              INSTALL   [<file1> ...]     |
              TEST      [<test1> ...]     |
              CACHE     [<entry1> ...]    >
             [APPEND] [APPEND_STRING]
             PROPERTY <name> [value1 ...])
```

Properties can also be used to configure targets.

Property	Scope	Function
OUTPUT_NAME	Target	Name of the library or executable file that is built. Used when you need a different name than its CMake target.
INSTALL_RPATH	Target	List of directories that will be searched for shared libraries when the program runs.
SOURCES	Target	Used to view or change the source files configured for a target.
LINK_LIBRARIES	Target	List of other libraries (targets or file paths) that the target should be linked to.

How to link targets

- Libraries can be *linked* to a target using:

  ```
  target_link_libraries(<target name> <library targets…>)
  ```

- This allows the target to reference code stored in the given libraries.
- Link dependencies are multi-level: if you link libA to libB, and libB to libC, then CMake will automatically link libA to libC as well.
- Linking will also pull in the *interface options* of the libraries being linked. Let’s discuss what that means.

Interface properties are used to carry dependencies between targets.

- Many target properties come in two types: a private version (e.g. `COMPILE_DEFINITIONS`) and an interface version (e.g. `INTERFACE_COMPILE_DEFINITIONS`)
- The private version affects the target it is set on
- The interface version affects every other target that links to this target.

Why are interface properties useful?

foo.h

```c
#include <readline.h>

// do stuff with readline
```

bar.h

```c
#include <foo.h>
```

foo.h requires the flag

`-I/usr/include/readline`

to compile.

bar.h (in another target) includes foo.h

Solution:

```cmake
set_property(TARGET foo PROPERTY
INTERFACE_INCLUDE_DIRECTORIES /usr/include/readline)
```

Exercise 1: Adding a Static Library

- Now we will apply what we’ve learned!
- We will take the project from session 1 and convert it into contain a C++ library.
- We will do a static library for now (shared libraries will be covered in session 3), and it will contain the quad reg code.
- It will also set interface properties to make using the code easier.

Exercise 1 Setup

- Open a terminal in CMakeTraining/exercises/session2
- Run these commands to set up the cmake project:
  - `mkdir build`
  - `cd build`
  - `cmake ..`

Exercise 1 Background

- You'll recall our top-level CMakeLists from last time.
- We now want to separate MovingQuadReg into its own library.
- There's another annoyance: having to include the quad_reg directory in the top-level CMakeLists.
- We're going to fix that too.

CMakeLists.txt

```cmake
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})
```

Creating the Library

- Create a new CMakeLists.txt in the quad_reg folder.
- Add a source list and create the library.

CMakeLists.txt — session2 | CMakeLists.txt — session2\quad_reg

```cmake
set(MOVING_QUAD_REG_SOURCES
    MovingQuadReg.cpp
    MovingQuadReg.h)

add_library(moving_quad_reg STATIC
    ${MOVING_QUAD_REG_SOURCES})

```

Interface Include Directories

- Now we're going to take our first step toward fixing the include file problem.
- Start by adding the call to `target_include_directories()`.
- This function is shorthand for setting the INCLUDE_DIRECTORIES target property.
- With the PUBLIC keyword it sets both the interface and private versions of the property.
- Note: `target_compile_options()` and `target_compile_definitions()` also exist and have the same behavior for their respective properties.

CMakeLists.txt — session2 ×   CMakeLists.txt — session2\quad_reg ×

```
set(MOVING_QUAD_REG_SOURCES
    MovingQuadReg.cpp
    MovingQuadReg.h)

add_library(moving_quad_reg STATIC
    ${MOVING_QUAD_REG_SOURCES})

target_include_directories(moving_quad_reg
    PUBLIC .)

```

Using the library

- Back to the top-level CMakeLists.txt.
- Remove MovingQuadReg from the source list.
- Add the `add_subdirectory()` call. This calls the CMakeLists.txt that we created in quad_reg.
- Remove the `include_directories()` call – it's not needed any more.
- Add the `target_link_libraries()` call. This links our executable to the library that we just created.

CMakeLists.txt — session2 ×   CMakeLists.txt — session2\quad_reg ×

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

set(TEST_QUADREG_SOURCES
    test_regression.cpp)

add_compile_options(--std=c++11)

add_subdirectory(quad_reg)

add_executable(test_regression
    ${TEST_QUADREG_SOURCES})

target_link_libraries(test_regression
    moving_quad_reg)
```

Exercise 1: Results

- Run your code:
  - `cmake ..`
  - `make`
- You should see things build just like before, only now the quadratic regression code will get built into a library first.

```
→  build cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/i/RPL/CMakeTraining/exercises/session2
/build
→  build make
[ 25%] Building CXX object quad_reg/CMakeFiles/moving_quad_reg.dir/MovingQuadReg
.cpp.o
[ 50%] Linking CXX static library libmoving_quad_reg.a
[ 50%] Built target moving_quad_reg
[ 75%] Building CXX object CMakeFiles/test_regression.dir/test_regression.cpp.o
[100%] Linking CXX executable test_regression
[100%] Built target test_regression
→  build
```

Part 2

Build Types and Compile Flags

The Weirdness that is Cache Variables

- Like all build systems, CMake provides a way for you to pass command line options that affect the build system’s behavior
- It’s also desirable to have a way to store the results of different tests so the cmake script runs faster when you run it subsequent times
- For CMake, both of these are achieved through the same mechanism: cache variables
- Cache variables are special variables that keep their values between invocations of CMake

Creating Cache Variables

- Cache variables are created using an alternate signature for set().
- Signature: set(<variable> <value>... CACHE <type> <docstring> [FORCE])
- CACHE keyword argument specifies cache signature
- Type specifies the type of data that will be stored in the variable.
  - Possible values: BOOL, FILEPATH, PATH, STRING, INTERNAL
- docstring is the comment that will be attached to the variable in the cache.
- If FORCE is given, the variable’s value will be overwritten. Otherwise it will not be modified if it already exists
  - Why is this?

Command Line Options

- Cache variables can also be set on the command line using –DMY_CACHE_VAR=<value>
- Intention is for set(CACHE) to be used to initialize an option with a default value, and then the user can override it on the command line
- However, sometimes scripts need to store data even though it isn't a settable option. This is what the INTERNAL type, which hides the variable from the user, is for

Viewing Cache Variables

- Cache variables can be viewed and edited by:
  - Editing CMakeCache.txt
  - Using the ccmake tool
  - Using the cmake-gui tool
- The intention was for these to provide a user-friendly interface for configuring CMake projects
- However, 95% of users don’t know these exist, so developers don’t usually spend time making them clean and usable, so people rarely use them
- Bit of a chicken-and-egg problem…

Cache Variable Example: CMakeCache.txt

```
set(MY_BOOL_VAR TRUE CACHE BOOL "My custom Boolean")
set(MY_PATH_VAR "/usr/local" CACHE PATH "Path to my thing")
set(MY_INTERNAL_VAR "don't touch me" CACHE INTERNAL "Internal data")
```

```
//My custom Boolean
MY_BOOL_VAR:BOOL=TRUE

//Path to my thing
MY_PATH_VAR:PATH=/usr/local
```

<snip>

```
//Internal data
MY_INTERNAL_VAR:INTERNAL=don't touch me
```

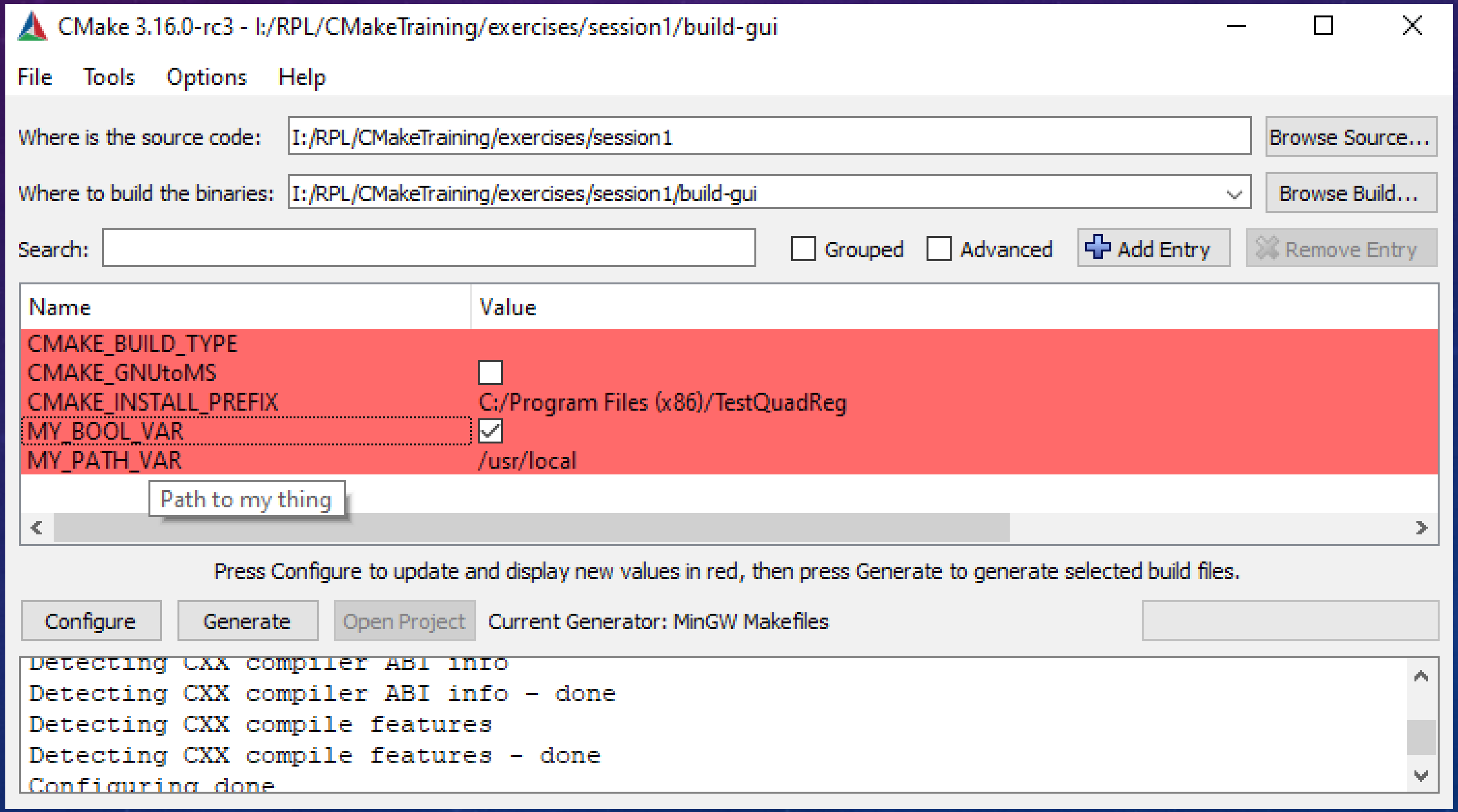
Cache Variable Example: ccmake

```
set(MY_BOOL_VAR TRUE CACHE BOOL "My custom Boolean")
set(MY_PATH_VAR "/usr/local" CACHE PATH "Path to my thing")
set(MY_INTERNAL_VAR "don't touch me" CACHE INTERNAL "Internal data")
```

```
ccmake ..
                                                     Page 1 of 1
 CMAKE_BUILD_TYPE
 CMAKE_INSTALL_PREFIX            /usr/local
 MY_BOOL_VAR                     ON
 MY_PATH_VAR                     /usr/local





MY_BOOL_VAR: My custom Boolean
Press [enter] to edit option Press [d] to delete an entry                            CMake Version 3.14.0-rc2
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Cache Variable Example: cmake-gui

```
set(MY_BOOL_VAR TRUE CACHE BOOL "My custom Boolean")
set(MY_PATH_VAR "/usr/local" CACHE PATH "Path to my thing")
set(MY_INTERNAL_VAR "don't touch me" CACHE INTERNAL "Internal data")
```



Cache Variable Overriding: A Cheat Sheet

This is one of the most confusing things in CMake by a large margin. Even I still mess it up.

If a _____ is defined, and you set a ____ with the same name, then…

local variable	local variable	The local variable's value is changed.
cache variable	cache variable (without FORCE)	The cache variable's original value is not changed
cache variable	cache variable (with FORCE)	The cache variable's original value is overwritten
cache variable	local variable	The cache variable is shadowed while the local variable is in scope
local variable	cache variable	If the cache variable was already in the cache, then the cache variable stays out of scope. However, if the cache variable did not exist, the local variable is deleted and the new cache variable is put into scope

Cache Variables Practice 1

What will it print?

```
set(INSTALL_LOCATION "/usr" CACHE STRING "Install location for stuff")
set(INSTALL_LOCATION "/opt" CACHE STRING "")
message(STATUS "Stuff will be installed to: ${INSTALL_LOCATION}")
```

```
-- Stuff will be installed to: /usr
```

Cache Variables Practice 2

What will it print?

```
set(INSTALL_LOCATION "/usr" CACHE STRING "Install location for stuff")
set(INSTALL_LOCATION "/opt" CACHE STRING "Install location for stuff" FORCE)
message(STATUS "Stuff will be installed to: ${INSTALL_LOCATION}")
```

```
-- Stuff will be installed to: /opt
```

Cache Variables Practice 3

What will it print?

```
set(INSTALL_LOCATION "/usr")
set(INSTALL_LOCATION "/opt" CACHE STRING "Install location for stuff")
message(STATUS "Stuff will be installed to: ${INSTALL_LOCATION}")
```

The first time CMake is run:

```
-- Stuff will be installed to: /opt
```

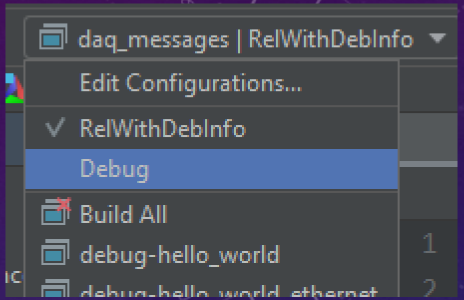
Subsequent runs:

```
-- Stuff will be installed to: /usr
```

Build configurations are used to adapt your build for different situations.

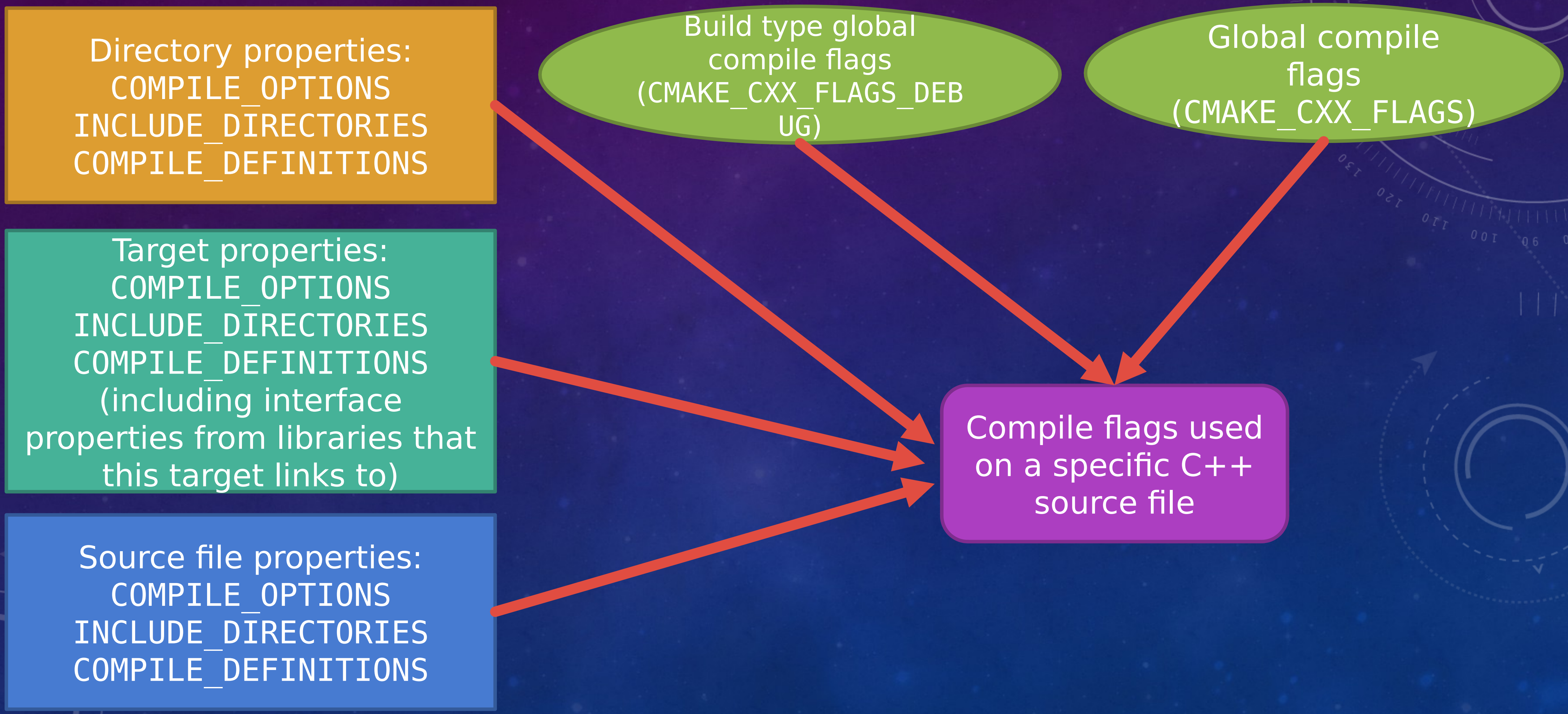
- Most build systems have the concept of Debug vs Release configurations.
  - Debug configuration: Optimization turned down to a low level or disabled, debug information generated.
  - Release configuration: Optimization at full, no debug information
- In CMake, this is achieved through the `CMAKE_BUILD_TYPE` cache variable.
- This variable, by default, has four standard build types: Debug, Release, RelWithDebInfo, and MinSizeRel.

Using CMAKE_BUILD_TYPE



- `CMAKE_BUILD_TYPE` is often set on the command line when building, e.g. `-DCMAKE_BUILD_TYPE=Release`. It is also set through the menu in IDEs.
- `CMAKE_BUILD_TYPE`'s main purpose is to control which compile flags are used.
- You can also use generator expressions to make it control other things, such as using a different version of a system library in debug vs release mode.

Compile flags come from many different sources.



Global compile flags

- CMake reads the global flags for each language from the cache variable “`CMAKE_<language>_FLAGS`”.
- This is “old”, but not “legacy” – it’s so baked in to the language that there is no way they will ever change it.
- As an old-style variable, it is a *space*-separated string, NOT a list.
- `CMAKE_<language>_FLAGS` is vital for ONE specific situation: when certain flags are required to make your compiler work.
- Example: when compiling for embedded ARM, you must pass –mcpu=<your CPU>.
- This is because `CMAKE_<language>_FLAGS` is used in CMake’s testing of the compiler itself, and you need to make sure this testing goes smoothly.

How to live with CMAKE_<language>_FLAGS

- Option one: ignore it completely.
  - Instead, apply all flags through directory and target properties.
  - This is nice because users can set it to add flags without doing any harm
- Option two: initialize it by setting `CMAKE_<language>_FLAGS_INIT` before the `project()` call.
  - The value will be copied from this variable when your project is configured.
  - This is required if you need specific flags to make your compiler work properly
  - However, you must tell your users to not set the variable manually, and give them some other option that adds compile flags if needed.

Build type global compile flags

- CMake also reads the flags for each configuration from the variable “CMAKE_<language>_FLAGS_<configuration>” (e.g. CMAKE_C_FLAGS_DEBUG).
- These are initialized to sensible defaults by CMake.
  - e.g. for Release, CMAKE_C_FLAGS_RELEASE is initialized to “-O3 –DNDEBUG”
- You can set these to your own values *after* the project() command if you want to change them.

Exercise 2: Adding Compile Flags

- This short exercise will apply some of what we just learned.
- We’re going to add multiple configuration types to our build system

Adding a build type

- First we need to initialize the build type.
- It defaults to an empty cache variable on the first run, so we need to define it.
- This code defaults it to Release, but you can use whatever works best for your project.
- NOTE: If supporting multi-configuration generators such as Visual Studio, a more complicated version of this code is needed. See here for details.

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

set(TEST_QUADREG_SOURCES
    test_regression.cpp)

if("${CMAKE_BUILD_TYPE}" STREQUAL "")
    set(CMAKE_BUILD_TYPE Release CACHE STRING
        "Type of build, options are: Debug, Release" FORCE)
endif()

add_compile_options(--std=c++11)

add_subdirectory(quad_reg)

add_executable(test_regression
    ${TEST_QUADREG_SOURCES})

target_link_libraries(test_regression
    moving_quad_reg)
```

Default compile options

- Now we’ll set up standard compile flags to enable warnings. Note that we are setting directory properties and ignoring CMAKE_CXX_FLAGS completely.
- We can also replace the --std=c++11 flag with something more portable. Setting CMAKE_CXX_STANDARD will cause CMake to automatically apply the correct flag for the current compiler.

CMakeLists.txt — session2 × CMakeLists.txt — session2\quad_reg ×

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

set(TEST_QUADREG_SOURCES
    test_regression.cpp)

set(CMAKE_BUILD_TYPE Release CACHE STRING
    "Type of build, options are: Debug, Release")

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

add_subdirectory(quad_reg)

add_executable(test_regression
    ${TEST_QUADREG_SOURCES})

target_link_libraries(test_regression
    moving_quad_reg)

```

Build type compile flags.

- Now we will set the build type compile flags.
- These are sensible defaults.
  - The Debug flags allow debugging and disable optimization
  - The Release flags enable optimization and disable asserts (that’s what -DNDEBUG does).

CMakeLists.txt — session2 | CMakeLists.txt — session2\quad_reg | untitled

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

set(TEST_QUADREG_SOURCES
    test_regression.cpp)

if("${CMAKE_BUILD_TYPE}" STREQUAL "")
    set(CMAKE_BUILD_TYPE Release CACHE STRING
        "Type of build, options are: Debug, Release" FORCE)
endif()

# set compile options
add_compile_options(-Wall -Wextra)
set(CMAKE_CXX_STANDARD 11)

set(CMAKE_CXX_FLAGS_DEBUG "-g2 -O0")
set(CMAKE_CXX_FLAGS_RELEASE "-O2 -DNDEBUG")

add_subdirectory(quad_reg)

add_executable(test_regression
    ${TEST_QUADREG_SOURCES})

target_link_libraries(test_regression
    moving_quad_reg)
```

Printing Compile Flags

- But with multiple configurations in the mix, how are users to know what the current compile flags are?
- For this, we will print a simple build report showing the current build type compile flags.
- We can get these by using a nested variable evaluation with CMAKE_BUILD_TYPE – but we have to convert it to uppercase first.
- Then we can print it out using CMake's message(STATUS) command.

CMakeLists.txt — session2 | CMakeLists.txt — session2\quad_reg

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

set(TEST_QUADREG_SOURCES
    test_regression.cpp)

if("${CMAKE_BUILD_TYPE}" STREQUAL "")
    set(CMAKE_BUILD_TYPE Release CACHE STRING
        "Type of build, options are: Debug, Release" FORCE)
endif()

# set compile options
add_compile_options(-Wall -Wextra)
set(CMAKE_CXX_STANDARD 11)

set(CMAKE_CXX_FLAGS_DEBUG "-g2 -O0")
set(CMAKE_CXX_FLAGS_RELEASE "-O2 -DNDEBUG")

add_subdirectory(quad_reg)

add_executable(test_regression
    ${TEST_QUADREG_SOURCES})

target_link_libraries(test_regression
    moving_quad_reg)

string(TOUPPER "${CMAKE_BUILD_TYPE}" CMAKE_BUILD_TYPE_UCASE)
message(STATUS ">> CXX Compile Flags (For ${CMAKE_BUILD_TYPE}):\
 ${CMAKE_CXX_FLAGS_${CMAKE_BUILD_TYPE_UCASE}}")

```

Exercise 2: Results

- First run CMake leaving the build type at its default: `cmake ..`
- Now change the build type: `cmake .. –DCMAKE_BUILD_TYPE=Debug`
- The compile flags will change like magic!

```
→  build cmake ..
-- >> CXX Compile Flags (For Release): -O2 -DNDEBUG
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/i/RPL/CMakeTraining/exercises/session2
/build
→  build cmake .. -DCMAKE_BUILD_TYPE=Debug
-- >> CXX Compile Flags (For Debug): -g2 -O0
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/i/RPL/CMakeTraining/exercises/session2
/build
→  build |
```

Session 2 Review

- Targets
- Properties
- Linking and interface properties
- Cache variables
- Build type
- Global compile flags