UNIVERSITÀ DEGLI STUDI DI PADOVA

Introducing CMake

Stefano Ghidoni

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

INTELLIGENT AUTONOMOUS SYSTEMS LAB

Agenda

- Describing a software project
- Building systems
- CMake
  - Advantages
  - User's perspective
  - Developer's perspective

Describing a project

- A software project is usually composed of several source files (.h, .hpp, .cpp)
- Complex projects are composed of:
  - One or more executable(s)
  - One or more library/ies

Describing a project

- Source files alone do not provide a description of the project
- Need to know the structure of the SW project
  - Deeper understanding of the SW project
  - Info about how to compile

Describing a project

- A SW project can be described by means of its **targets**
- Compilation targets
  - Create an executable
  - Create a library
- In both cases
  - List of the .cpp files to be compiled together
  - List of the libraries to be linked

Describing a project

- How can we describe a project?
- Description based on **compile command**
  - Impractical
  - Hard to interpret
- Description based on **IDE project**
  - Convenient
  - Platform-dependent
- Description based on a **building system**

Building systems

- A building system handles the compilation process
- Several options
  - Make, Ninja, …
  - Sometimes depend on the platform

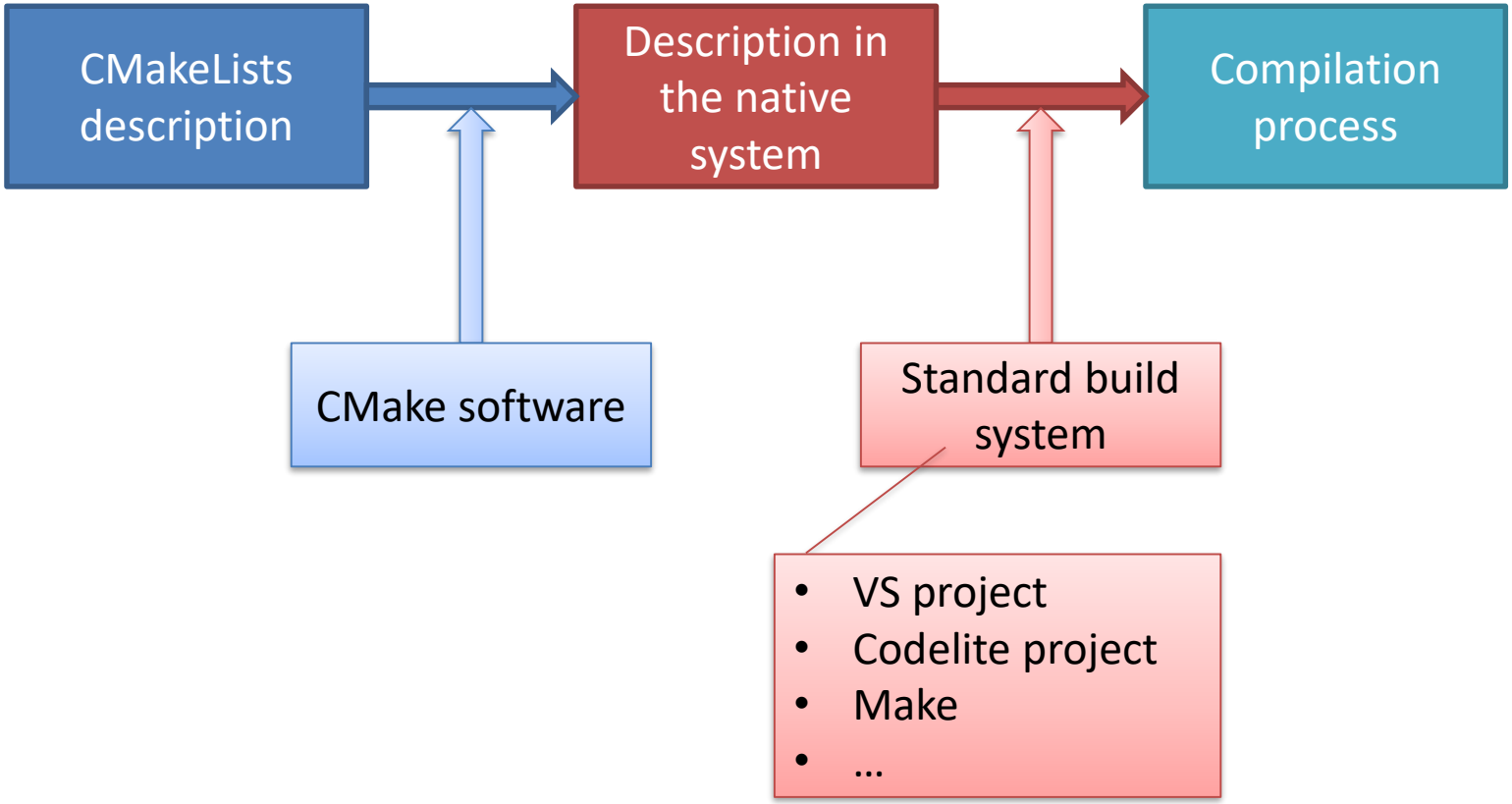
Introducing CMake

- Standard meta-building system
- Independent from:
  - Compiler
  - Platform
- Used in combination with the "local" building system
- Supported by many IDEs

Introducing CMake

- CMake can be seen as a way for describing projects
- It defines:
  - Compilation targets
    - Executables
    - Libraries
  - Libraries to be linked
- Based on text files named CMakeLists.txt

CMake: process



CMake homepage

CMake

About Resources Developer Resources Download

Build with CMake. Build with Confidence.

Learn How

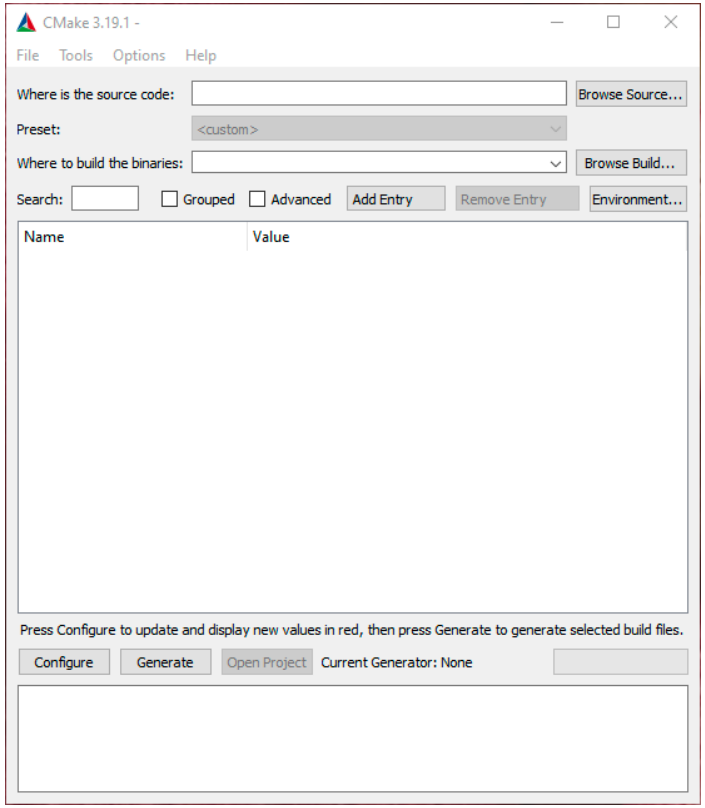
CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice. The suite of CMake tools were created by Kitware in response to the need for a powerful, cross-platform build environment for open-source projects such as ITK and VTK.

CMake is part of Kitware's collection of commercially supported **open-source platforms** for software development.

User's perspective

CMake software

- GUI or command-line
- Two activities:
  - Configure
  - Generate



User's perspective

- Typical workflow:
  - **Call CMake** (GUI/command line)
  - Select the directory containing the source
  - Select the directory where the compiler output shall go
    - Standard: [project root]/build
  - **Configure**
  - (possible) module enable/disable
  - **Generate**

Compiling the project

- The *generate* action creates the files to be provided to the build system
  - E.g.: VS solution
- From that point, the compilation process gets out of CMake's scope

Compiling the project

- Example under Linux

```
mkdir build
cd build
cmake ..
make
```

Compilation results in a separate folder (optional)

Compiling the project

- Example under Linux

```
mkdir build
cd build
cmake ..
make
```

- Generates compiler specific files
- **No compilation done yet!**

Compiling the project

- Example under Linux

```
mkdir build
cd build
cmake ..
make
```

- Compiles everything using a Makefile (generated by CMake)

Developer's perspective

Developer's perspective

- Describe the project using the CMake standard
  - Create CMakeLists.txt file manually
  - Use an IDE to generate CMakeLists.txt automatically
    - More complex
    - Do not modify them directly – only through the IDE

CMakeLists.txt

- Choose a minimum CMake version
  - 2.8 is usually enough

Sets the mimimum required version of CMake.

```
cmake_minimum_required(VERSION 3.2 FATAL_ERROR)
```

CMakeLists.txt

- Give a name to the project

Set the name and version; enable languages.

```
project(<name> VERSION <version> LANGUAGES CXX)
```

- CMake sets several variables based on project().
- Call to project() must be direct, not through a function/macro/include.
- CMake will add a call to project() if not found on the top level.

CMakeLists.txt

- List libraries needed by the project
  - E.g., OpenCV

Finds preinstalled dependencies

```
find_package(Qt5 REQUIRED COMPONENTS Widgets)
```

- Can set some variables and define imported targets.
- Supports components.

CMakeLists.txt

- Target: creating an executable
  - Corresponds to: g++ -o tool –c main.cpp another_file.cpp

Add an executable target.

```
add_executable(tool
  main.cpp
  another_file.cpp
  )
```

CMakeLists.txt

- Target: creating a library

Add a library target.

```
add_library(foo STATIC
  foo1.cpp
  foo2.cpp
  )
```

- Libraries can be STATIC, SHARED, MODULE, or INTERFACE.
- Default can be controlled with BUILD_SHARED_LIBS.

CMake – a simple example


```
cmake_minimum_required(VERSION 3.2 FATAL_ERROR)

project (tuple)

include_directories(include)

add_library(tuple
    include/tuple.h
    src/tuple.cpp
)

add_executable(add_tuple
    src/add_tuple.cpp
)
target_link_libraries(add_tuple
    tuple
)
```

CMake – working with OpenCV


```
cmake_minimum_required(VERSION 2.8)
project(test1)

find_package(OpenCV REQUIRED)

include_directories(${OpenCV_INCLUDE_DIRS})

add_executable(${PROJECT_NAME} src/test1.cpp)
target_link_libraries(${PROJECT_NAME} ${OpenCV_LIBS})
```

CMake – working with OpenCV

CMakeLists with a system installed library (taken from your basic_opencv_package on Moodle)

```
cmake_minimum_required(VERSION 2.8)
project(test1)

find_package(OpenCV REQUIRED)

include_directories(${OpenCV_INCLUDE_DIRS})

add_executable(${PROJECT_NAME} src/test1.cpp)
target_link_libraries(${PROJECT_NAME} ${OpenCV_LIBS})
```

Looks for OpenCVConfig.cmake (where a bunch of OpenCV related vars are defined)

CMake – working with OpenCV

CMakeLists with a system installed library (taken from your basic_opencv_package on Moodle)

```
cmake_minimum_required(VERSION 2.8)
project(test1)

find_package(OpenCV REQUIRED)

include_directories(${OpenCV_INCLUDE_DIRS})

add_executable(${PROJECT_NAME} src/test1.cpp)
target_link_libraries(${PROJECT_NAME} ${OpenCV_LIBS})
```

Tells the compiler to look for the headers also in the folders defined in this variable

CMake – working with OpenCV

CMakeLists with a system installed library (taken from your basic_opencv_package on Moodle)

```
cmake_minimum_required(VERSION 2.8)
project(test1)

find_package(OpenCV REQUIRED)

include_directories(${OpenCV_INCLUDE_DIRS})

add_executable(${PROJECT_NAME} src/test1.cpp)
target_link_libraries(${PROJECT_NAME} ${OpenCV_LIBS})
```

Tells the compiler to build an executable named test1, the source file for the compilation is just src/test1.cpp

CMake – working with OpenCV

CMakeLists with a system installed library (taken from your basic_opencv_package on Moodle)

```
1  cmake_minimum_required(VERSION 2.8)
2  project(test1)
3
4  find_package(OpenCV REQUIRED)
5
6  include_directories(${OpenCV_INCLUDE_DIRS})
7
8  add_executable(${PROJECT_NAME} src/test1.cpp)
9  target_link_libraries(${PROJECT_NAME} ${OpenCV_LIBS})
```

Tells the linker to to link the test1 executable with the libraries defined by the variable

UNIVERSITÀ DEGLI STUDI DI PADOVA

Introducing CMake

Stefano Ghidoni

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

INTELLIGENT AUTONOMOUS SYSTEMS LAB