Modern CMake

Dennis Klein

Software Development for Experiments Group
Central IT Department, GSI

C++ User Group Meeting
3rd November 2021, GSI Darmstadt

Notes

Disclaimer

I do not necessarily endorse CMake. My CMake experience is purely coincidental.

Notes

Disclaimer

I do not necessarily endorse CMake. My CMake experience is purely coincidental.

- However, it is worth talking and learning about CMake, because it is a popular choice among C++ projects.
- We are planning a more basic intro to buildsystems/CMake in a separate talk in the future.

Outline

Definitions

What is Modern CMake?

Modern CMake refers to a buildsystem written using CMake language idioms and CMake library features available and preferred since version 3 (roughly).

Definitions

What is Modern CMake?

Modern CMake refers to a buildsystem written using CMake language idioms and CMake library features available and preferred since version 3 (roughly).

What is a buildsystem?

A *buildsystem* is a graph of high-level logical targets used to automate (incremental) building, installing (packaging), and testing software from source.

Definitions

What is Modern CMake?

Modern CMake refers to a buildsystem written using CMake language idioms and CMake library features available and preferred since version 3 (roughly).

What is a buildsystem?

A *buildsystem* is a graph of high-level logical targets used to automate (incremental) building, installing (packaging), and testing software from source.

What is a target?

A *target* represents an executable, a library or a custom artifact (usually produced by a user-defined command/script).

Properties

Properties

Properties

Definition

CMake properties are key/value pairs defined on various CMake objects or `scopes`:

- Global Scope,
- Directories, (avoid using these in modern CMake)
- Targets,
- Tests,
- Source Files,
- Cache Entries, and
- Installed Files.

https://cmake.org/cmake/help/latest/manual/cmake-properties.7.html

Scopes - Examples (1)

Directory Scope

```
project
├── CMakeLists.txt
├── libA
│   └── CMakeLists.txt
└── libB
    └── CMakeLists.txt
```

```
include_directories([AFTER|BEFORE] [SYSTEM] dir1 [dir2 ...])
```

⇒ Sets the `INCLUDE_DIRECTORIES` property on a directory, which is inherited by all targets in this directory.

Scopes - Examples (2)

Target Scope

```
1 target_include_directories(<target> [SYSTEM] [BEFORE]
2   <INTERFACE|PUBLIC|PRIVATE> [items1...]
3   [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

⇒ Sets the `*INCLUDE_DIRECTORIES` properties on a target.

Propagation

Some CMake commands offer a compact syntax to modify multiple target properties at once:

```
target_link_libraries(<target>
  <PRIVATE|PUBLIC|INTERFACE> <lib1> ...
  [<PRIVATE|PUBLIC|INTERFACE> <lib2> ...] ...)
```

populates	INTERFACE_LINK_LIBRARIES	LINK_LIBRARIES
PRIVATE		X
PUBLIC	X	X
INTERFACE	X	

Generic Accessors

Retrieving properties:

```
get_property(...)
get_directory_property(...)
get_target_property(...)
```

Setting properties (if supported):

```
set_property(...)
set_directory_properties(...)
set_target_properties(...)
```

Target Relationships

Target Relationships

Expressing Dependencies

Idiom

Modern CMake reuses the command `target_link_libraries` to declare dependencies between targets.
We also use this command, even if we do not want to *link* with the dependencies, e.g. for header-only libraries.

Expressing Dependencies

Idiom

Modern CMake reuses the command `target_link_libraries` to declare dependencies between targets.
We also use this command, even if we do not want to *link* with the dependencies, e.g. for header-only libraries.

Example: Shared library B depends on header-only library A

project/libA/CMakeLists.txt

```
1 add_library(A INTERFACE)
2 target_include_directories(A INTERFACE ${CMAKE_CURRENT_SOURCE_DIR})
3 target_compile_definitions(A INTERFACE "DEBUG=1")
```

project/libB/CMakeLists.txt

```
1 add_library(B SHARED source.cpp)
2 target_link_libraries(B PRIVATE A)
```

Exported Targets

Idiom

Declare properties that are relevant for consuming a library/executable at target scope.

Exported Targets

Idiom

Declare properties that are relevant for consuming a library/executable at target scope.

In addition to the installed library files, a CMake script can be generated and installed along, that contains a description of all installed targets and their properties.

Exported Targets

Idiom

Declare properties that are relevant for consuming a library/executable at target scope.

In addition to the installed library files, a CMake script can be generated and installed along, that contains a description of all installed targets and their properties.

Add targets to an export set:

```
1 install(TARGETS <target> ... [EXPORT <export-set>] ...)
```

Exported Targets

Idiom

Declare properties that are relevant for consuming a library/executable at target scope.

In addition to the installed library files, a CMake script can be generated and installed along, that contains a description of all installed targets and their properties.

Add targets to an export set:

```
install(TARGETS <target> ... [EXPORT <export-set>] ...)
```

Install the export set:

```
install(EXPORT <export-set> DESTINATION <dir>
  [NAMESPACE <namespace>] [FILE <name>.cmake] ...)
```

This installs the exported targets to a file `<install-dir>/<dir>/<name>.cmake`.

CMake package

Definition

A CMake package consists of a top level CMake script file, which includes a version CMake script and exported target set files.

If the top level CMake package file is installed in the search path, `find_package()` can find and include the external CMake package without a dedicated `Find*.cmake` module.

CMake package

Definition

A CMake package consists of a top level CMake script file, which includes a version CMake script and exported target set files.

If the top level CMake package file is installed in the search path, `find_package()` can find and include the external CMake package without a dedicated `Find*.cmake` module.

Example

```
1 list(PREPEND CMAKE_PREFIX_PATH $ENV{ROOTSYS})
2 find_package(ROOT REQUIRED COMPONENTS RIO Net)
```

https://cmake.org/cmake/help/latest/manual/cmake-packages.7.html

https://cmake.org/cmake/help/latest/command/find_package.html

Imported Targets

Include exported target set or find CMake package

```
include(${FAIRROOTPATH}/include/cmake/FairMQ.cmake)
# or
find_package(FairRoot)
# ...
target_link_libraries(A PUBLIC FairRoot::FairMQ)
```

Imported Targets

Include exported target set or find CMake package

```
include(${FAIRROOTPATH}/include/cmake/FairMQ.cmake)
# or
find_package(FairRoot)
# ...
target_link_libraries(A PUBLIC FairRoot::FairMQ)
```

Define imported targets in a find module, e.g. Findnanomsg.cmake

```
find_path(NANOMSG_INCLUDE_DIR NAMES nanomsg/nn.h)
find_library(NANOMSG_LIBRARY_SHARED NAMES nanomsg)
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(nanomsg
  REQUIRED_VARS NANOMSG_LIBRARY_SHARED NANOMSG_INCLUDE_DIR)
if(NOT TARGET nanomsg)
  add_library(nanomsg SHARED IMPORTED)
  set_target_properties(nanomsg PROPERTIES
    IMPORTED_LOCATION ${NANOMSG_LIBRARY_SHARED}
    INTERFACE_INCLUDE_DIRECTORIES ${NANOMSG_INCLUDE_DIR})
endif()
```

Generator Expressions

Generator Expressions

Two-pass Configure

```
cd <build-dir>
cmake -DCMAKE_INSTALL_PREFIX=<install-dir> <source-dir>
make
make install
```

Two-pass Configure

```
cd <build-dir>
cmake -DCMAKE_INSTALL_PREFIX=<install-dir> <source-dir>
make
make install
```

or

```
cmake -S <source-dir> -B <build-dir> -DCMAKE_INSTALL_PREFIX=<install-dir>
cmake --build <build-dir>
cmake --build <build-dir> --target install
```

Two-pass Configure

```
cd <build-dir>
cmake -DCMAKE_INSTALL_PREFIX=<install-dir> <source-dir>
make
make install
```

or

```
cmake -S <source-dir> -B <build-dir> -DCMAKE_INSTALL_PREFIX=<install-dir>
cmake --build <build-dir>
cmake --build <build-dir> --target install
```

The configure step is implemented with a two-pass logic:

1. Configuration pass - evaluates your `CMakeLists.txt` scripts
2. Generation pass

Usually printed at the end of a CMake configure:

```
-- Configuring done
-- Generating done
-- Build files have been written to: <build-dir>
```

Generator Expressions

```
$<condition:true_string>
$<KEYWORD:list/string/expr>
$<KEYWORD:arg1,arg2[,arg3]>
```

Generator Expressions

```
$<condition:true_string>
$<KEYWORD:list/string/expr>
$<KEYWORD:arg1,arg2[,arg3]>
```

Examples

Build and installation locations of header files might differ:

```
1 target_include_directories(A PUBLIC
2   $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}>
3   $<INSTALL_INTERFACE:include>)
```

Generator Expressions

```
$<condition:true_string>
$<KEYWORD:list/string/expr>
$<KEYWORD:arg1,arg2[,arg3]>
```

Examples

Build and installation locations of header files might differ:

```
target_include_directories(A PUBLIC
  $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}>
  $<INSTALL_INTERFACE:include>)
```

Invoke an executable as custom command:

```
add_custom_command(TARGET A PRE_BUILD
  COMMAND $<TARGET_FILE:ROOT::cling> ...)
```

https://cmake.org/cmake/help/latest/manual/cmake-generator-expressions.7.html

Functions

Functions

Functions

- Use *functions* for lexical variable scope
- Use *macros* for dynamic variable scope (prefer *functions* if possible)
- Return values from *functions* via: `set(result varname PARENT_SCOPE)`.

Functions

- Use *functions* for lexical variable scope
- Use *macros* for dynamic variable scope (prefer *functions* if possible)
- Return values from *functions* via: `set(result varname PARENT_SCOPE)`.

Pass positional and optional arguments explicitely

Implement optional arguments with the `CMakeParseArguments` module.

```
1  # add_fairroot_library(name [SOURCES source1 source2 ...]
2  #   [HEADERS header1 header2 ...] [NO_DICT_SRCS source1 source2 ...]
3  #   [DEPENDENCIES dep1 dep2 ...] [LINKDEF linkdef1 linkdef2 ...]
4  #   [INCLUDE_DIRS incdir1 incdir2 ...] [DEFINITIONS def1 def2 ...])
5  #
6  function(add_fairroot_library lib_NAME)
7    cmake_parse_arguments(lib "" ""
8        "SOURCES;HEADERS;NO_DICT_SRCS;DEPENDENCIES;LINKDEF;INCLUDE_DIRS;DEFINITIONS"
9        ${ARGN})
10   # access optional args via ${lib_SOURCES} or ${lib_LINKDEF}
11 endfunction()
```

Style - Wrapper

Provide custom function with optional arguments:

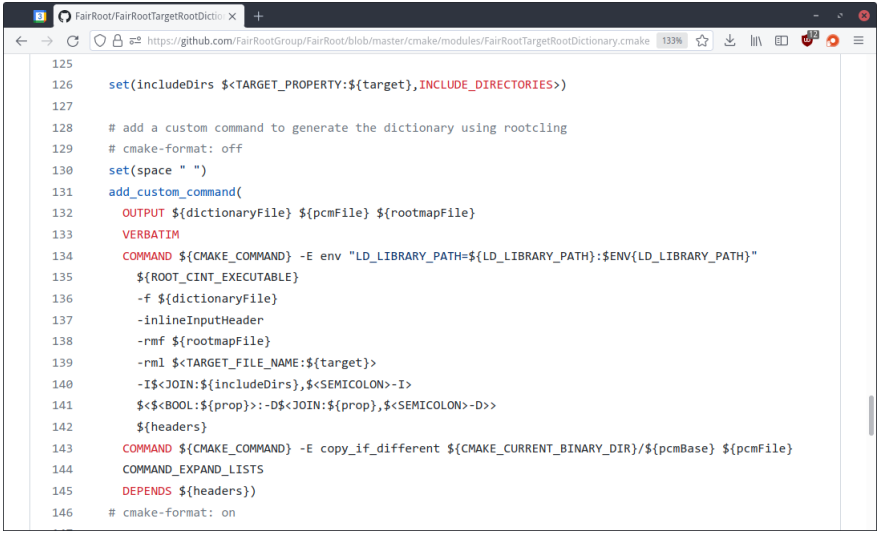
```
add_fairroot_library(ParBase
  SOURCES
  FairContFact.cxx
  FairDetParAsciiFileIo.cxx
  (...)
  FairRtdbRun.cxx
  FairRuntimeDb.cxx

  INCLUDE_DIRS $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}>
  DEPENDENCIES FairRoot::FairTools ROOT::RIO ROOT::Core
  LINKDEF ParBaseLinkDef.h
)
```

Style - plain CMake with opt-in

Plain CMake with opt-in custom function for root dictionary:

```
1  add_library(ParBase SHARED
2    FairContFact.cxx
3    FairDetParAsciiFileIo.cxx
4    (...)
5    FairRtdbRun.cxx
6    FairRuntimeDb.cxx)
7  add_library(FairRoot::ParBase ALIAS ParBase)
8  target_include_directories(ParBase PUBLIC $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}>)
9  target_link_libraries(ParBase PUBLIC FairRoot::FairTools ROOT::RIO ROOT::Core)
10 fairroot_target_root_dictionary(ParBase LINKDEF ParBaseLinkDef.h)
11 install(TARGETS ParBase EXPORT FairRoot ...)
```

ROOT Dictionary Generation




```
125
126   set(includeDirs $<TARGET_PROPERTY:${target},INCLUDE_DIRECTORIES>)
127
128   # add a custom command to generate the dictionary using rootcling
129   # cmake-format: off
130   set(space " ")
131   add_custom_command(
132     OUTPUT ${dictionaryFile} ${pcmFile} ${rootmapFile}
133     VERBATIM
134     COMMAND ${CMAKE_COMMAND} -E env "LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:$ENV{LD_LIBRARY_PATH}"
135       ${ROOT_CINT_EXECUTABLE}
136       -f ${dictionaryFile}
137       -inlineInputHeader
138       -rmf ${rootmapFile}
139       -rml $<TARGET_FILE_NAME:${target}>
140       -I$<JOIN:${includeDirs},$<SEMICOLON>-I>
141       $<$<BOOL:${prop}>:-D$<JOIN:${prop},$<SEMICOLON>-D>>
142       ${headers}
143     COMMAND ${CMAKE_COMMAND} -E copy_if_different ${CMAKE_CURRENT_BINARY_DIR}/${pcmBase} ${pcmFile}
144     COMMAND_EXPAND_LISTS
145     DEPENDS ${headers})
146   # cmake-format: on
```

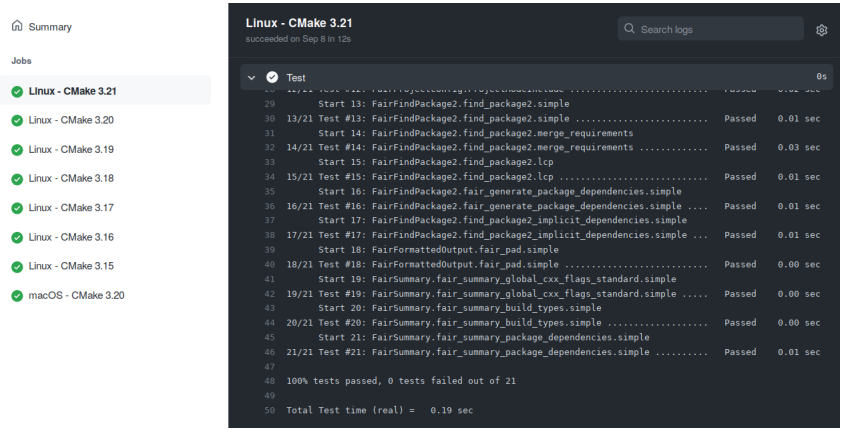
https://github.com/FairRootGroup/FairRoot/blob/master/cmake/modules/FairRootTargetRootDictionary.cmake

Testing CMake

Testing CMake

CMake is totally testable

There is no excuse to not test CMake functions/macros! ;-P



https://github.com/FairRootGroup/FairCMakeModules/tree/main/tests

FairCMakeModules

- Tested and documented CMake Module library
- Deduplicate CMake code otherwise copied literally in multiple of our repos
- Docs: https://fairrootgroup.github.io/FairCMakeModules/latest/
- Sources: https://github.com/FairRootGroup/FairCMakeModules
- Open source and open development, you are welcome to use/fork/contribute to it!

FairCMakeModules

- Tested and documented CMake Module library
- Deduplicate CMake code otherwise copied literally in multiple of our repos
- Docs: https://fairrootgroup.github.io/FairCMakeModules/latest/
- Sources: https://github.com/FairRootGroup/FairCMakeModules
- Open source and open development, you are welcome to use/fork/contribute to it!

Thank you for your attention!