Embracing Modern CMake

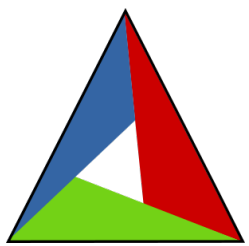
How to recognize and use modern CMake interfaces

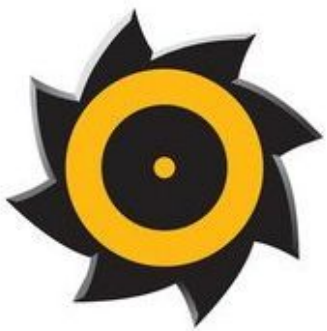
Stephen Kelly

Dublin C++ Meetup

September 11, 2017

Background

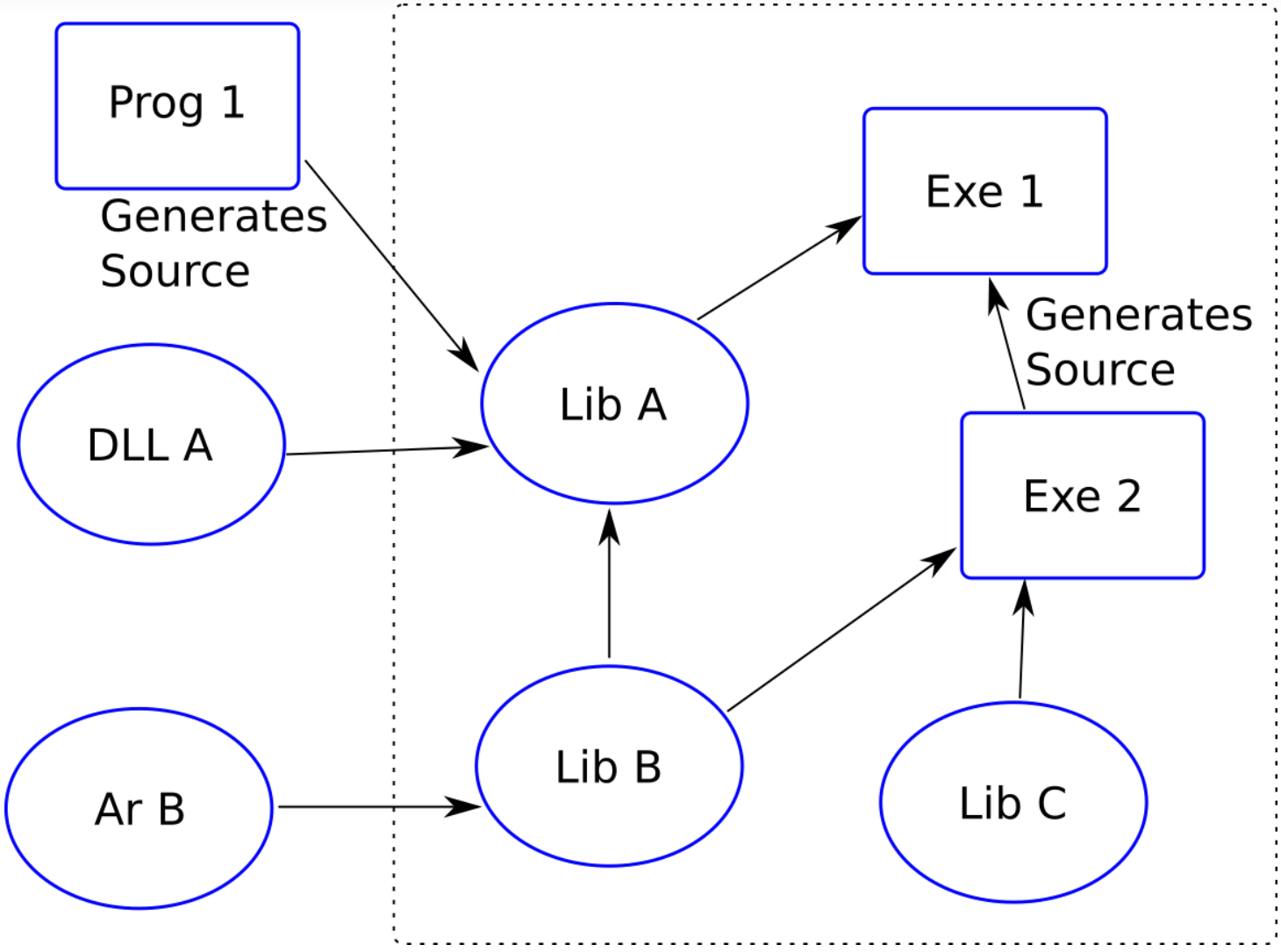




CMake - What, Why, Who

- Buildsystem Generator
- 'Cross-platform Make'
- Part of suite of productivity and quality tools
- Started by Kitware in 2000

CMake - What, Why, Who


Prog 1
Generates
Source
Exe 1
Generates
Source
DLL A
Lib A
Exe 2
Ar B
Lib B
Lib C

Makefiles

Visual Studio/Xcode Project

CMake

Where CMake shines

- Finding dependencies
- Portability
- Code generation
- Multi-language support

What is Modern CMake?

- New(er) APIs and mindset of writing CMake code
- Less code
- Cleaner code
- More target-focussed

The Good News

Mostly everything available to you already

	2012	2013		2014		2015			2016			2017
	October	May	October	June	December	March	July	November	March	July	November	April
CMake	<= 2.8.10	2.8.11	2.8.12	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
Ubuntu	12.04		12.04						16.04		17.04	
Debian		Wheezy		Jessie							Sid	
Fedora							23		24	25		
Travis					Yes							
RHEL	5		6				7					

Defining a Buildsystem


```
1  cmake_minimum_required(VERSION 3.5)
2  project(myproject)
3
4  add_library(libsalutation STATIC salutation.cpp)
5
6  add_executable(hello hello.cpp)
7  target_link_libraries(hello
8    libsalutation
9  )
10
11 add_executable(goodbye goodbye.cpp)
12 target_link_libraries(goodbye
13   libsalutation
14 )
```

Policies

Policies

```
1 cmake_minimum_required(VERSION 2.8)
```

Policies

```
1 cmake_minimum_required(VERSION 2.8)
```

- Fail at runtime if version is too low
- Populate variable `CMAKE_MINIMUM_REQUIRED_VERSION`
- (Re)set runtime behavior of CMake with policies
- Should be first line of your CMake buildsystem (before `project`)

Policies

- Behavior deprecation mechanism
- In WARN state by default
- Set individually for fine control

	...
CMake 3.3	CMP0057
	CMP0056
CMake 3.2	CMP0055
CMake 3.1	CMP0054
	...

Policies


```
1 cmake_minimum_required(VERSION 3.0)
2 if (POLICY CMP0053)
3   cmake_policy(SET CMP0053 NEW)
4 endif()
```


```
1 cmake_policy(SET CMP0053 OLD)
```

When to Set a Policy to OLD

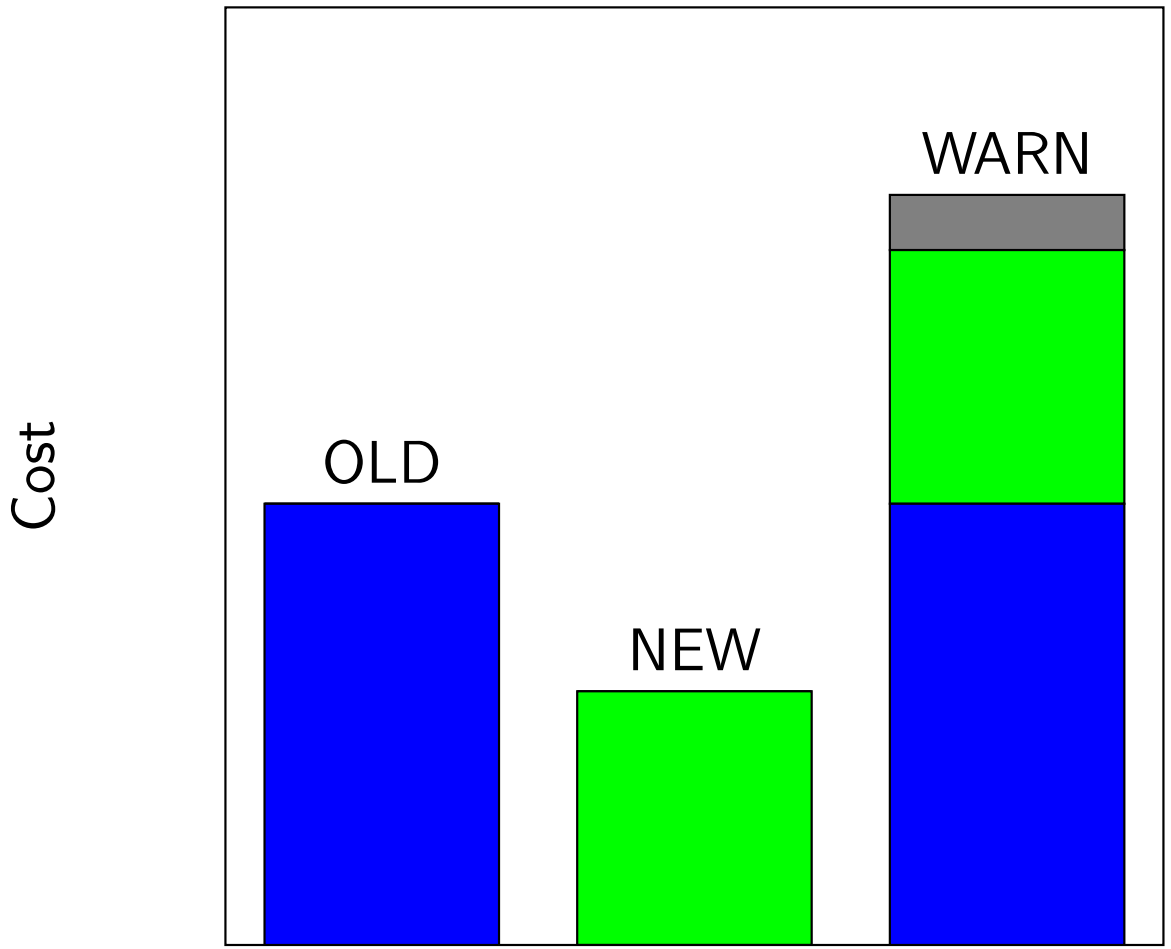
`std::allOf(`

- You are close to your own release
- A new release of CMake just happened
- A policy warning is triggered for your code
- Your code relies on the OLD behavior
  - Or you suspect it might
- You don't pass the setting to dependents

`);`

Create a plan to migrate to the `NEW` behavior!

Policy warning is slower



Policies

UseCase: Require new features (higher version) but rely on OLD behavior


```
1 cmake_minimum_required(VERSION 3.3)
2 cmake_policy(SET CMP0003 OLD)
3 target_include_directories(...)
```

Policies

UseCase: Allow old CMake but use `NEW` behavior where possible


```
1 cmake_minimum_required(VERSION 3.0)
2 foreach (pol CMP0053
3              CMP0063
4              CMP0065
5     )
6     if (POLICY ${pol})
7         cmake_policy(SET ${pol} NEW)
8     endif()
9 endforeach()
```

Some Policies do not issue warnings

as of 3.8

CMP0025	Compiler id for Apple Clang is now AppleClang
CMP0047	Use QCC compiler id for the qcc drivers on QNX
CMP0056	Honor link flags in try_compile()
CMP0060	Link libraries by full path even in implicit dirs
CMP0061	CTest does not by default tell make to ignore errors
CMP0065	ENABLE_EXPORTS target property flags
CMP0066	Honor per-config flags in try_compile()
CMP0067	Honor language standard in try_compile()

- Use `-DCMAKE_POLICY_WARNING_CMP<NNNN>=ON` to enable it

Modern CMake Guidelines

- Maintain up-to-date policy settings

Usage Requirements

Defining a Buildsystem


```
1  cmake_minimum_required(VERSION 3.5)
2  project(myproject)
3
4  add_library(libsalutation STATIC salutation.cpp)
5
6  add_executable(hello hello.cpp)
7  target_link_libraries(hello
8    libsalutation
9  )
10
11 add_executable(goodbye goodbye.cpp)
12 target_link_libraries(goodbye
13   libsalutation
14 )
```

Defining a Buildsystem


```
1 cmake_minimum_required(VERSION 3.5)
2 project(myproject)
3
4 add_subdirectory(libraries)
5
6 add_subdirectory(executables)
```

Defining a Buildsystem


```
1  include_directories(${salutation_INCLUDES})
2
3  add_executable(hello hello.cpp)
4  target_link_libraries(hello
5  libsalutation
6  )
7
8  add_executable(goodbye goodbye.cpp)
9  target_link_libraries(goodbye
10 libsalutation
11 )
```

Defining a Buildsystem


```
1  add_executable(hello hello.cpp)
2  target_link_libraries(hello
3    libsalutation)
4  target_include_directories(hello
5    PRIVATE ${salutation_INCLUDES})
6
7  add_executable(goodbye goodbye.cpp)
8  target_link_libraries(goodbye
9    libsalutation)
10 target_include_directories(goodbye
11   PRIVATE ${salutation_INCLUDES})
```

Defining a Buildsystem

2.8.11

```
1 add_executable(hello hello.cpp)
2 target_link_libraries(hello
3   libsalutation
4 )
5
6 add_executable(goodbye goodbye.cpp)
7 target_link_libraries(goodbye
8   libsalutation
9 )
```

Build properties

- Target-based buildsystem definition
- Single point of dependency specification
- Targets provide information to dependers
  - Requirements to compile
  - Requirements to link

Defining a Buildsystem

2.8.11

```
1 add_library(salutation salutation.cpp)
2 target_include_directories(salutation
3   PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include
4 )
```

Transitive compile dependency

2.8.11

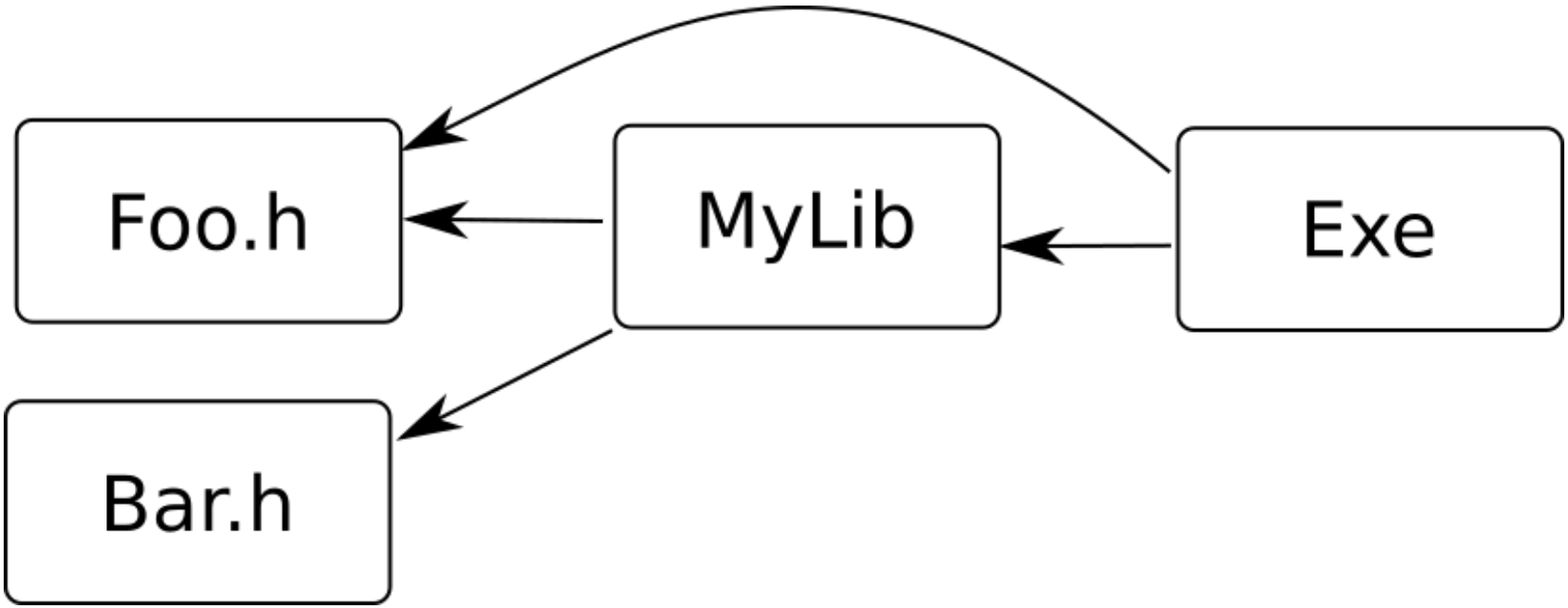
```
1 target_include_directories(<target>
2    <PUBLIC|PRIVATE|INTERFACE>
3    [items...]
4 )
```

Transitive compile dependency

```
1  #include <FooDependency>
2
3  class MyClass : public FooDependency
4  {
5        MyClass();
6  };
```

```
1  #include <BarDependency>
2
3  MyClass::MyClass()
4  {
5      doInitialize(BarDependency{});
6  }
```

Transitive compile dependency



Transitive compile dependency

PRIVATE	Needed by me, but **not** my dependers
PUBLIC	Needed by me **and** my dependers
INTERFACE	Needed **not** by me, **but** by my dependers

Defining a Buildsystem

2.8.11

```
1 add_library(salutation salutation.cpp)
2 target_include_directories(salutation
3   PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include
4 )
```

Transitive Scope

2.8.11

```
1 target_include_directories(myTarget
2   PUBLIC     "/something/public"
3   PRIVATE    "/something/private"
4   INTERFACE  "/something/interface"
5   PUBLIC     "/another/public"
6   PRIVATE    "/another/private"
7 )
```

include_directories command


```
1 include_directories(some_dir)
2
3 add_library(libA)
4
5 add_library(libB)
```

include_directories command


```
1 add_library(libA)
2
3 include_directories(some_dir)
4
5 add_library(libB)
```

include_directories command


```
1 add_library(libA)
2
3 add_library(libB)
4
5 include_directories(some_dir)
```

include_directories command


```
1  add_library(libA)
2
3  add_subdirectory(dir1)
4
5  include_directories(some_dir)
6
7  add_library(libB)
8
9  add_subdirectory(dir2)
```

include_directories command


```
1 add_library(libA)
2
3 subdirs(dir1)
4
5 include_directories(some_dir)
6
7 add_library(libB)
8
9 add_subdirectory(dir2)
```

Lack of transitivity


```
1  set(app_INCLUDES
2    ${lib1_INCLUDES}
3    ${lib2_INCLUDES}
4    ${lib3_INCLUDES})
5
6  set(app_LIBRARIES ...)
7
8  include_directories(${app_INCLUDES})
9
10 add_subdirectory(dir1)
11 add_subdirectory(dir2)
12 add_subdirectory(dir3)
13
14 add_subdirectory(app)
```

Compile Definitions

`add_definitions` command has all the same problems as `include_directories` command.


```
1 add_library(salutation salutation.cpp)
2 add_definitions(
3   -DUSE_INTERNAL_SIMD
4   -DUSE_MULTITHREADING)
```

Compile Definitions

2.8.11

```
1 add_library(salutation salutation.cpp)
2 target_include_directories(salutation
3   PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include
4 )
5 target_compile_definitions(salutation
6   PRIVATE USE_INTERNAL_SIMD
7   PUBLIC  USE_MULTITHREADING
8 )
```

Build properties

Support <PRIVATE|PUBLIC|INTERFACE> and transitivity

Include Directories (`-I/foo/bar`)	`target_include_directories`
Compile Definitions (`-DSOMEDEF`)	`target_compile_definitions`
Compile Options (`-fPIC`)	`target_compile_options`
Link Libraries (`-l/path/to/lib`)	`target_link_libraries`
Sources	`target_sources`

Modern CMake Guidelines

- Maintain up-to-date policy settings
- Write target-centric code
  - Use `target_` command variants
  - Specify usage requirements for targets

Lack of transitivity


```
1  set(app_INCLUDES
2    ${lib1_INCLUDES}
3    ${lib2_INCLUDES}
4    ${lib3_INCLUDES})
5
6  set(app_LIBRARIES ...)
7
8  include_directories(${app_INCLUDES})
9
10 add_subdirectory(dir1)
11 add_subdirectory(dir2)
12 add_subdirectory(dir3)
13
14 add_subdirectory(app)
```

Reliance on variables


```
1  set(main_SRCS
2    main.cpp)
3
4  add_executable(app ${main_SRCS})
5  target_include_directories(app
6    PRIVATE ${app_INCLUDES})
7  target_compile_definitions(app
8    PRIVATE ${app_DEFINES})
9  target_link_libraries(app
10   ${app_LIBRARIES})
```

Reliance on variables


```
1  set(main_SRCS
2    main.cpp)
3
4  add_executable(app ${main_SRCS})
5  target_include_directories(app
6    PRIVATE                    )
7  target_compile_definitions(app
8    PRIVATE                  )
9  target_link_libraries(app
10                       )
```

Reliance on variables

Eschew obfuscation; Espouse elucidation


```
1 add_executable(app main.cpp)
2 target_link_libraries(app
3     lib2 lib3
4 )
```

Problems with Variables

- Variables are fragile
- Variables leak to other contexts
- Variables don't express scope of dependencies
- Variables are not checked for correctness or content

Modern CMake Guidelines

- Maintain up-to-date policy settings
- Write target-centric code
  - Use `target_` command variants
  - Specify usage requirements for targets
- Avoid unnecessary variables

Generator Expressions

Conditions


```
1  set(main_SRCS
2    main.cpp
3  )
4  if (WIN32)
5    list(APPEND main_SRCS helper_win.cpp)
6  else()
7    list(APPEND main_SRCS helper_posix.cpp)
8  endif()
9
10 add_executable(hello ${main_SRCS})
```

Conditions

3.1

```
1  add_executable(hello main.cpp)
2  if (WIN32)
3    target_sources(hello PRIVATE
4      helper_win.cpp
5    )
6  else()
7    target_sources(hello PRIVATE
8      helper_posix.cpp
9    )
10 endif()
```

Conditions

3.1

```
1 add_executable(hello
2   main.cpp
3   $<$<BOOL:${WIN32}>:helper_win.cpp>
4   $<$<NOT:$<BOOL:${WIN32}>>:helper_posix.cpp>
5 )
```

Conditions

Warning: this code is buggy and non-portable!

```
1  set(main_SRCS
2    main.cpp
3  )
4  if (CMAKE_BUILD_TYPE STREQUAL DEBUG)
5    list(APPEND main_SRCS helper_debug.cpp)
6  else()
7    list(APPEND main_SRCS helper_rel.cpp)
8  endif()
9
10 add_executable(hello ${main_SRCS})
```

Conditions

3.1

```
1 add_executable(hello
2   main.cpp
3   $<$<CONFIG:Debug>:helper_debug.cpp>
4   $<$<NOT:$<CONFIG:Debug>>:helper_rel.cpp>
5 )
```

Generator Expressions

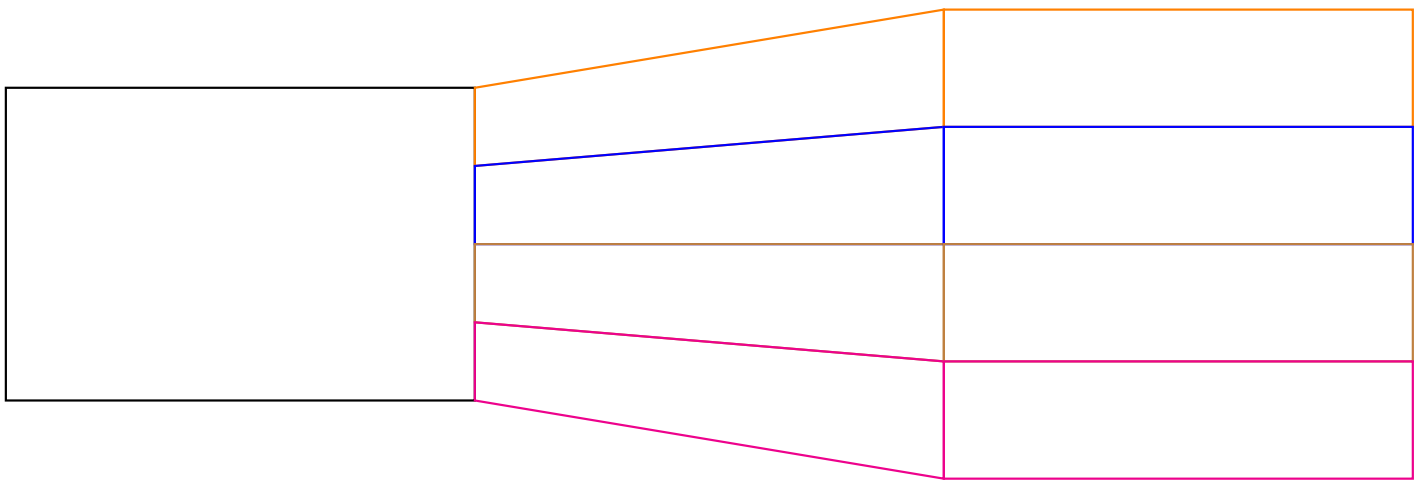
Configure

if()/else()/endif()

Compute

Generate

$<1:...>



Generator Expressions basics

$<1:...>	...
$<0:...>	
$<Config:Debug>	1 (in Debug config)
$<Config:Debug>	0 (in Debug config)
$<$<Config:Debug>:...>	... (in Debug config)
$<$<Config:Debug>:...>	(in Debug config)

Truthiness conversion

`$<$<BOOL:${WIN32}>:...>` at configure time produces `$<$<BOOL:1>:...>` or `$<$<BOOL:>:...>` at generate-time!

```
1 add_executable(hello
2   main.cpp
3   $<$<BOOL:${WIN32}>:helper_win.cpp>
4   $<$<NOT:$<BOOL:${WIN32}>>:helper_posix.cpp>
5 )
```

Support for Generator Expressions

- `target_` commands
- `file(GENERATE)` command
- `add_executable`/`add_library` commands
- `install` command (partial)
- `add_custom_target` command (partial)
- `add_custom_command` command (partial)

There are others, but these are the most important

Imperative versus Transitive

2.8.11

```
1  # Compile with USE_THREADS if the
2  # WITH_THREADS property is ON
3  get_property(buildWithThreads TARGET hello
4    PROPERTY WITH_THREADS)
5  if (buildWithThreads)
6    target_compile_definitions(hello PRIVATE
7      USE_THREADS)
8  endif()
9
10 set_property(TARGET hello
11   PROPERTY WITH_THREADS ON)
```

Imperative versus Transitive

2.8.11

```
1
2  # Compile with USE_THREADS if the
3  # WITH_THREADS property is ON
4  target_compile_definitions(hello PRIVATE
5    $<$<TARGET_PROPERTY:WITH_THREADS>:USE_THREADS>)
6
7  set_property(TARGET hello
8    PROPERTY WITH_THREADS ON)
```

Conditions

- "Generator Expression" conditions at Generate-time
- Test content after configure-time
  - Configuration
  - TARGET_PROPERTY
  - TARGET_POLICY
  - COMPILE_FEATURES
  - LOCATION
- Never use `CMAKE_BUILD_TYPE` in `if()`

Modern CMake Guidelines

- Maintain up-to-date policy settings
- Write target-centric code
  - Use `target_` command variants
  - Specify usage requirements for targets
- Avoid unnecessary variables
- Use generate-time conditions correctly

target_link_libraries

Defining a Buildsystem

2.8.11

```
1 add_executable(hello hello.cpp)
2 target_link_libraries(hello
3   libsalutation
4 )
5
6 add_executable(goodbye goodbye.cpp)
7 target_link_libraries(goodbye
8   libsalutation
9 )
```

target_link_libraries

```
1 target_link_libraries(someTarget <item>)
```

`<item>` can be:

- A CMake target
- A library name on disk
- A full library path
- A linker flag

target_link_libraries

```
1 target_link_libraries(someTarget aTargetName)
```

- Link to aTargetName
- Determine build order
- Consume usage requirements
  - Compiling
  - Linking
- Determine compatibility

target_link_libraries

```
1 target_link_libraries(someTarget oopsItsATypo)
```

- Check if it is a CMake target name
- Check if it is a link flag (starts with '-')
- Check if it is a path
- Assume it is a libraryname (add -loopsItsATypo)

CMake Target types

Executables	`add_executable`
Shared libraries	`add_library(SHARED)`
Static libraries	`add_library(STATIC)`
Object libraries	`add_library(OBJECT)`
Interface libraries	`add_library(INTERFACE)`
Alias libraries	`add_library(ALIAS)`

Interface targets

Suitable for header-only libraries

```
1 add_library(boost_mpl INTERFACE)
2 target_compile_definitions(boost_mpl
3 INTERFACE BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS)
4 target_include_directories(boost_mpl
5 INTERFACE "3rdparty/boost/mpl")
```

```
1 add_executable(my_exe)
2 target_link_libraries(my_exe boost_mpl)
```

Interface targets

Suitable for header-only libraries

```
1  add_library(boost_mpl INTERFACE)
2  target_compile_definitions(boost_mpl
3  INTERFACE BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS)
4  target_include_directories(boost_mpl
5  INTERFACE "3rdparty/boost/mpl")
6
7  add_library(boost_icl INTERFACE)
8  target_link_libraries(boost_icl
9  INTERFACE boost_mpl)
10 target_include_directories(boost_icl
11 INTERFACE "3rdparty/boost/icl")
```

```
1  add_executable(my_exe)
2  target_link_libraries(my_exe boost_icl)
```

Interface targets

Group build properties for convenient consumption

```
1  target_link_libraries(windows_specific
2  INTERFACE directX)
3  target_compile_definitions(windows_specific
4  INTERFACE USE_DIRECTX)
5  target_sources(windows_specific
6  INTERFACE network_win.cpp)
7
8  add_library(platform_specific INTERFACE)
9  target_link_libraries(platform_specific INTERFACE
10 $<$<BOOL:${WIN32}>:windows_specific>
11 $<$<NOT:$<BOOL:${WIN32}>>:posix_specific>
12 )
```

Interface targets

3.1

```
1 target_link_libraries(mytarget
2 platform_specific
3 helper_library
4 )
```

Alias targets

2.8.12

```
add_library(detail::platform_specific
    ALIAS platform_specific)
)
```

```
target_link_libraries(mytarget
  detail::platform_specific
)
```

Alias targets

2.8.12

```
add_library(boost::mpl
    ALIAS boost_mpl)
)
```

```
target_link_libraries(mytarget
  boost::mpl
)
```

Dependencies

External dependencies

3.1

```
1  cmake_minimum_required(VERSION 3.5)
2  project(myproject)
3
4  find_package(Qt5Widgets REQUIRED)
5  find_package(Qt53D REQUIRED)
6
7  add_executable(hello main.cpp)
8  target_link_libraries(hello
9    Qt5::Widgets Qt5::3DCore
10 )
```

External dependencies

3.1

```
1  cmake_minimum_required(VERSION 3.5)
2  project(myproject)
3
4  find_package(Qt5Widgets REQUIRED)
5
6  add_library(locallib STATIC locallib.cpp)
7  target_link_libraries(locallib PUBLIC
8    Qt5::Widgets
9  )
10 add_executable(hello main.cpp)
11 target_link_libraries(hello
12   locallib
13 )
```

Legacy pattern


```
1  cmake_minimum_required(VERSION 3.5)
2  project(myproject)
3
4  find_package(Qt5Core REQUIRED)
5  find_package(Qt5Gui REQUIRED)
6  find_package(Qt5Widgets REQUIRED)
7
8  add_definitions(
9     ${Qt5Core_DEFINITIONS}
10    ${Qt5Gui_DEFINITIONS}
11    ${Qt5Widgets_DEFINITIONS}
12 )
```

Legacy pattern


```
1  include_directories(
2    ${Qt5Core_INCLUDE_DIRS}
3    ${Qt5Gui_INCLUDE_DIRS}
4    ${Qt5Widgets_INCLUDE_DIRS}
5  )
6
7  set(CMAKE_CXX_FLAGS
8    "${CMAKE_CXX_FLAGS} ${Qt5Core_CXX_FLAGS} \
9      ${Qt5Gui_CXX_FLAGS} ${Qt5Widgets_CXX_FLAGS}")
```

Legacy pattern


```
1  add_library(locallib SHARED
2    locallib.cpp
3  )
4  target_link_libraries(locallib
5    ${Qt5Core_LIBRARIES} ${Qt5Gui_LIBRARIES}
6    ${Qt5Widgets_LIBRARIES}
7  )
8  set(locallib_LIBRARIES locallib
9    ${Qt5Core_LIBRARIES} ${Qt5Gui_LIBRARIES}
10   ${Qt5Widgets_LIBRARIES}
11 )
```

Old style packages populate variables


```
1  find_package(Foo REQUIRED)
2
3  add_executable(hello main.cpp)
4  target_include_directories(hello
5    PRIVATE ${Foo_INCLUDE_DIRS}
6  )
7  target_compile_definitions(hello
8    PRIVATE ${Foo_COMPILE_DEFINITIONS}
9  )
10 target_link_libraries(hello
11   ${Foo_LIBRARIES}
12 )
```

Modern CMake packages define IMPORTED targets.


```
1 find_package(Foo REQUIRED)
2
3 add_executable(hello main.cpp)
4 target_link_libraries(hello
5   Foo::Core
6 )
```

- Compare with syntax using `ALIAS` libraries.
- CMake code for `hello` doesn't change under refactoring

Modern CMake Guidelines

- Maintain up-to-date policy settings
- Write target-centric code
  - Use `target_` command variants
  - Specify usage requirements for targets
- Avoid unnecessary variables
- Use generate-time conditions correctly
- Use `IMPORTED` targets for external dependencies

Creating Packages

```
1  add_library(mylib ...)
2
3  install(TARGETS mylib
4      EXPORT exp
5      ARCHIVE DESTINATION lib)
6
7  install(EXPORT exp
8      NAMESPACE ns:: DESTINATION share/cmake)
9  install(FILES mylibConfig.cmake
10     DESTINATION share/cmake)
```

Modern CMake Guidelines

- Maintain up-to-date policy settings
- Write target-centric code
  - Use `target_` command variants
  - Specify usage requirements for targets
- Avoid unnecessary variables
- Use generate-time conditions correctly
- Use `IMPORTED` targets for external dependencies
- Install 'rich' targets and export packages

Where to get more information

- Avoid the wiki
- Use the documentation
- Use the cmake mailing list
- Use stack overflow

Thanks & Questions

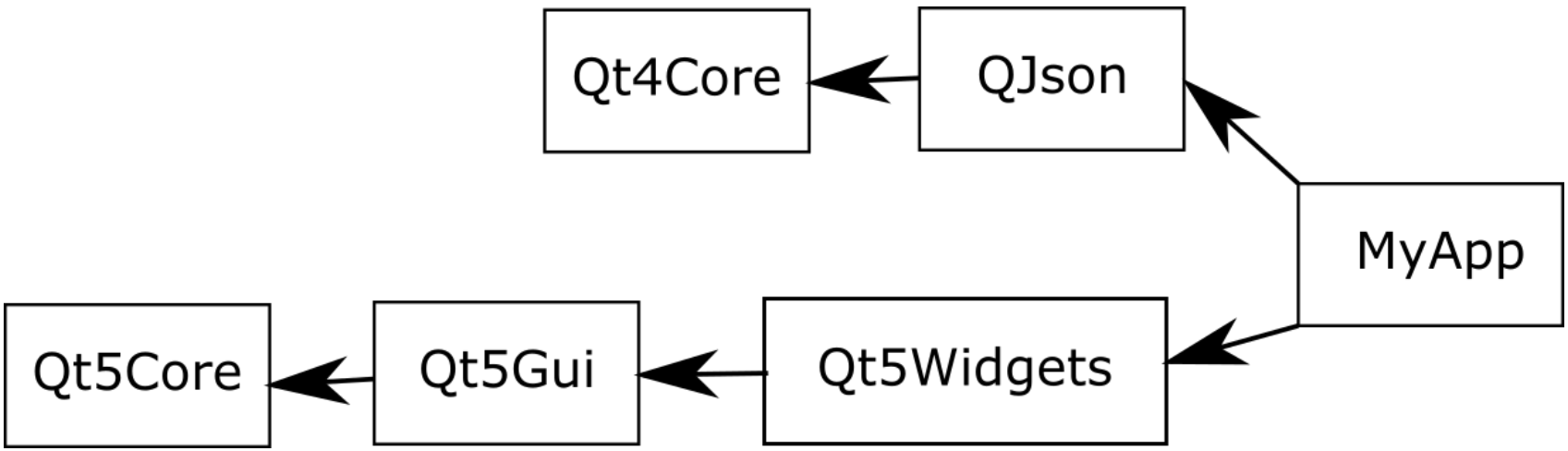
Special thanks:

- Brad King
- Ben & Robert
- Reddit <3

Compatible interfaces

```
1 find_package(qjson REQUIRED)
2
3 add_executable(main main.cpp)
4 target_link_libraries(main
5     Qt5::Widgets qjson::qjson)
```

Compatible interfaces



Compatible interfaces

```
1  set_property(TARGET Qt4::QtCore PROPERTY
2      INTERFACE_QT_MAJOR_VERSION 4
3  )
4  set_property(TARGET Qt4::QtCore APPEND PROPERTY
5      COMPATIBLE_INTERFACE_STRING QT_MAJOR_VERSION
6  )
7  # ...
8  set_property(TARGET Qt5::Core PROPERTY
9      INTERFACE_QT_MAJOR_VERSION 5
10 )
11 set_property(TARGET Qt5::Core APPEND PROPERTY
12     COMPATIBLE_INTERFACE_STRING QT_MAJOR_VERSION
13 )
```

Creating build-dir Packages

```
1 add_library(mylib ...)
2
3 install(TARGETS mylib
4     EXPORT exp
5     ARCHIVE DESTINATION lib)
6
7 export(EXPORT exp
8     NAMESPACE ns:: FILE buildDirTargets.cmake)
```

- Non-relocatable (use `install(EXPORT)` for that)
- Suitable in cross-compiles or superbuilds.