More Modern CMake

Working with CMake 3.12 and later

Deniz Bahadir BENOCS

cmake@deniz.bahadir.email

What is *CMake*?

WHAT IS *CMAKE*?

- CMake is a build-system *generator*

What is *CMake*?

- CMake is a build-system *generator*
  - **Not** a build-*system*, though!
  - generates *input files* for build-generators.
    - Supports: Make, Ninja, Visual Studio, XCode, ...

What is *CMake*?

- CMake is a build-system *generator*
  - **Not** a build-*system*, though!
  - generates *input files* for build-generators.
    - Supports: Make, Ninja, Visual Studio, XCode, ...
- It is cross-platform.
  - Supports *running* on: Linux, Windows, OSX, ...
  - Supports *building* cross-platform, too.
    - If the compiler supports that.

What is *CMake*?

- CMake is a build-system *generator*
  - **Not** a build-*system*, though!
  - generates *input files* for build-generators.
    - Supports: Make, Ninja, Visual Studio, XCode, ...
- It is cross-platform.
  - Supports *running* on: Linux, Windows, OSX, ...
  - Supports *building* cross-platform, too.
    - If the compiler supports that.
- Supports generating build-systems for multiple languages.
  - **C/C++** , FORTRAN, C#, CUDA...

A (very) brief history of *CMake*.

A (very) brief history of *CMake*.

- CMake started around 1999/2000.
  - **Not** modern.

A (VERY) BRIEF HISTORY OF *CMAKE*.

- CMake started around 1999/2000.
  - **Not** modern.

- CMake is **"modern"** since version 3.0. *(around 2014)*
  - New concept: *"everything is a (self-contained) target"*.

A (VERY) BRIEF HISTORY OF *CMAKE*.

- CMake started around 1999/2000.
  - **Not** modern.

- CMake is **"modern"** since version 3.0. *(around 2014)*
  - New concept: ***"everything is a (self-contained) target"***.

- CMake 3.11 was released *(March 2018)*
  - Unifies several commands.
- CMake 3.12 was released *(July 2018)*
  - Some of the big left-out tasks of *Modern CMake* were completed.

A (VERY) BRIEF HISTORY OF *CMAKE*.

- CMake started around 1999/2000.
  - **Not** modern.
- ⇒ Traditional CMake

- CMake is "**modern**" since version 3.0. *(around 2014)*
  - New concept: *"everything is a (self-contained) target"*.
- ⇒ Modern CMake

- CMake 3.11 was released *(March 2018)*
  - Unifies several commands.
- CMake 3.12 was released *(July 2018)*
  - Some of the big left-out tasks of *Modern CMake* were completed.
- ⇒ More Modern CMake

Some terms used throughout this talk

Build-requirements of a target

BUILD-REQUIREMENTS OF A TARGET

"Everything that is needed to (successfully) ***build*** *that target."*

BUILD-REQUIREMENTS OF A TARGET

"Everything that is needed to (successfully) ***build*** *that target."*

- source-files
- include search-paths
- pre-processor macros
- link-dependencies
- compiler/linker-options
- compiler/linker-features
    - *(e.g. support for a C++-standard)*

Usage-requirements of a target

USAGE-REQUIREMENTS OF A TARGET

*"Everything that is needed to (successfully) **use** that target."*

USAGE-REQUIREMENTS OF A TARGET

*"Everything that is needed to (successfully) **use** that target."*

"As a dependency of another target."

USAGE-REQUIREMENTS OF A TARGET

*"Everything that is needed to (successfully) **use** that target."*

"As a dependency of another target."

- source-files
  - *(but normally not)*
- include search-paths
- pre-processor macros
- link-dependencies
- compiler/linker-options
- compiler/linker-features
  - *(e.g. support for a C++-standard)*

Traditional CMake? Modern CMake?

What's the difference?

Comparision

	Traditional CMake	Modern CMake

Comparision

	Traditional CMake	Modern CMake
build-requirements are set on?	on **environment** (mainly) *e.g. directory scope*	on **targets***

* Or already on dependencies (as we will see later).

Comparision

	Traditional CMake	Modern CMake
build-requirements are set on?	on **environment** (mainly) *e.g. directory scope*	on **targets***
keeping track of *usage-requirements*	via (cache-)**variables**	via **targets** *(keep track themselves)*

* Or already on dependencies (as we will see later).

Comparision

	Traditional CMake	Modern CMake
build-requirements are set on?	on **environment** (mainly) *e.g. directory scope*	on **targets***
keeping track of *usage-requirements*	via (cache-)**variables**	via **targets** *(keep track themselves)*
usage-requirements propagation from dependency (by using `target_link_libraries` command)	**explicit propagation by hand****	**automatic propagation**

* Or already on dependencies (as we will see later).
** Only paths to library-files are propagated by default.

Comparision

	Traditional CMake	Modern CMake
build-requirements are set on?	on **environment** (mainly) *e.g. directory scope*	on **targets***
keeping track of *usage-requirements*	via (cache-)**variables**	via **targets** *(keep track themselves)*
usage-requirements propagation from dependency (by using `target_link_libraries` command)	**explicit propagation by hand****	**automatic propagation**
	More **error-prone**!	Less error-prone! Allows for more **fine-grained configuration.**

* Or already on dependencies (as we will see later).
** Only paths to library-files are propagated by default.

Modern CMake

setting Build-requirements vs setting Usage-requirements

```
# Adding build-requirements

target_include_directories( <target> PRIVATE <include-search-dir>... )
target_compile_definitions( <target> PRIVATE <macro-definitions>... )
target_compile_options(     <target> PRIVATE <compiler-option>... )
target_compile_features(    <target> PRIVATE <feature>... )
target_sources(             <target> PRIVATE <source-file>... )
target_link_libraries(      <target> PRIVATE <dependency>... )
target_link_options(        <target> PRIVATE <linker-option>... )
target_link_directories(    <target> PRIVATE <linker-search-dir>... )
```

```
# Adding usage-requirements

target_include_directories( <target> INTERFACE <include-search-dir>... )
target_compile_definitions( <target> INTERFACE <macro-definitions>... )
target_compile_options(     <target> INTERFACE <compiler-option>... )
target_compile_features(    <target> INTERFACE <feature>... )
target_sources(             <target> INTERFACE <source-file>... )
target_link_libraries(      <target> INTERFACE <dependency>... )
target_link_options(        <target> INTERFACE <linker-option>... )
target_link_directories(    <target> INTERFACE <linker-search-dir>... )
```

Modern CMake

setting Build-requirements vs setting Usage-requirements

```cmake
# Adding build-requirements

target_include_directories( <target> PRIVATE <include-search-dir>... )
target_compile_definitions( <target> PRIVATE <macro-definitions>... )
target_compile_options(     <target> PRIVATE <compiler-option>... )
target_compile_features(    <target> PRIVATE <feature>... )
target_sources(             <target> PRIVATE <source-file>... )
target_link_libraries(      <target> PRIVATE <dependency>... )
target_link_options(        <target> PRIVATE <linker-option>... )
target_link_directories(    <target> PRIVATE <linker-search-dir>... )
```

```cmake
# Adding usage-requirements

target_include_directories( <target> INTERFACE <include-search-dir>... )
target_compile_definitions( <target> INTERFACE <macro-definitions>... )
target_compile_options(     <target> INTERFACE <compiler-option>... )
target_compile_features(    <target> INTERFACE <feature>... )
target_sources(             <target> INTERFACE <source-file>... )
target_link_libraries(      <target> INTERFACE <dependency>... )
target_link_options(        <target> INTERFACE <linker-option>... )
target_link_directories(    <target> INTERFACE <linker-search-dir>... )
```

Modern CMake

setting Build-requirements vs setting Usage-requirements

```
# Adding build-requirements

target_include_directories( <target> PRIVATE <include-search-dir>... )
target_compile_definitions( <target> PRIVATE <macro-definitions>... )
target_compile_options(     <target> PRIVATE <compiler-option>... )
target_compile_features(    <target> PRIVATE <feature>... )
target_sources(             <target> PRIVATE <source-file>... )
target_link_libraries(      <target> PRIVATE <dependency>... )
target_link_options(        <target> PRIVATE <linker-option>... )
target_link_directories(    <target> PRIVATE <linker-search-dir>... )
```

```
# Adding usage-requirements

target_include_directories( <target> INTERFACE <include-search-dir>... )
target_compile_definitions( <target> INTERFACE <macro-definitions>... )
target_compile_options(     <target> INTERFACE <compiler-option>... )
target_compile_features(    <target> INTERFACE <feature>... )
target_sources(             <target> INTERFACE <source-file>... )
target_link_libraries(      <target> INTERFACE <dependency>... )
target_link_options(        <target> INTERFACE <linker-option>... )
target_link_directories(    <target> INTERFACE <linker-search-dir>... )
```

Modern CMake

setting Build-requirements vs setting Usage-requirements

```
# Adding build-requirements

target_include_directories( <target> PRIVATE <include-search-dir>... )
target_compile_definitions( <target> PRIVATE <macro-definitions>... )
target_compile_options(     <target> PRIVATE <compiler-option>... )
```

```
# Adding build- and usage-requirements

target_include_directories( <target> PUBLIC <include-search-dir>... )
target_compile_definitions( <target> PUBLIC <macro-definitions>... )
target_compile_options(     <target> PUBLIC <compiler-option>... )
target_compile_features(    <target> PUBLIC <feature>... )
target_sources(             <target> PUBLIC <source-file>... )
target_link_libraries(      <target> PUBLIC <dependency>... )
target_link_options(        <target> PUBLIC <linker-option>... )
target_link_directories(    <target> PUBLIC <linker-search-dir>... )
```

```
target_compile_definitions( <target> INTERFACE <macro-definitions>... )
target_compile_options(     <target> INTERFACE <compiler-option>... )
target_compile_features(    <target> INTERFACE <feature>... )
target_sources(             <target> INTERFACE <source-file>... )
target_link_libraries(      <target> INTERFACE <dependency>... )
target_link_options(        <target> INTERFACE <linker-option>... )
target_link_directories(    <target> INTERFACE <linker-search-dir>... )
```

Modern CMake

setting Build-requirements vs setting Usage-requirements

```
# Adding build-requirements

target_include_directories( <target> PRIVATE <include-search-dir>... )
target_compile_definitions( <target> PRIVATE <macro-definitions>... )
target_compile_options(     <target> PRIVATE <compiler-option>... )
```

```
# Adding build- and usage-requirements

target_include_directories( <target> PUBLIC <include-search-dir>... )
target_compile_definitions( <target> PUBLIC <macro-definitions>... )
target_compile_options(     <target> PUBLIC <compiler-option>... )
target_compile_features(    <target> PUBLIC <feature>... )
target_sources(             <target> PUBLIC <source-file>... )
target_link_libraries(      <target> PUBLIC <dependency>... )
target_link_options(        <target> PUBLIC <linker-option>... )
target_link_directories(    <target> PUBLIC <linker-search-dir>... )
```

```
                                                                   )
target_compile_definitions( <target> INTERFACE <macro-definitions>... )
target_compile_options(     <target> INTERFACE <compiler-option>... )
target_compile_features(    <target> INTERFACE <feature>... )
target_sources(             <target> INTERFACE <source-file>... )
target_link_libraries(      <target> INTERFACE <dependency>... )
target_link_options(        <target> INTERFACE <linker-option>... )
target_link_directories(    <target> INTERFACE <linker-search-dir>... )
```

- **Warning:** Although `target_link_libraries` can be used without these keywords, you should **never forget to use these keywords** in *Modern CMake*!

Traditional CMake vs. Modern CMake vs. More Modern CMake

demonstrated with example-project "Calculator app"

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Traditional CMake vs. Modern CMake vs. More Modern CMake

demonstrated with example-project "Calculator app"

- A *free* executable `FreeCalculator`

- A *non-free* executable `PremiumCalculator` (with extended functionality)

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Traditional CMake vs. Modern CMake vs. More Modern CMake

Demonstrated with example-project "Calculator app"

- A *free* executable `FreeCalculator`
  - dynamically linked with **Boost.Program_Options** shared-library.

- A *non-free* executable `PremiumCalculator` (with extended functionality)
  - dynamically linked with **Boost.Program_Options** shared-library.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Traditional CMake vs. Modern CMake vs. More Modern CMake

demonstrated with example-project "Calculator app"

- A *free* executable `FreeCalculator`
  - dynamically linked with **Boost.Program_Options** shared-library.
  - dynamically linked with `basicmath` shared-library.
- A *non-free* executable `PremiumCalculator` (with extended functionality)
  - dynamically linked with **Boost.Program_Options** shared-library.
  - dynamically linked with `extmath` shared-library

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Traditional CMake vs. Modern CMake vs. More Modern CMake

demonstrated with example-project "Calculator app"

- A *free* executable `FreeCalculator`
  - dynamically linked with **Boost.Program_Options** shared-library.
  - dynamically linked with `basicmath` shared-library.
- A *non-free* executable `PremiumCalculator` (with extended functionality)
  - dynamically linked with **Boost.Program_Options** shared-library.
  - dynamically linked with `extmath` shared-library, which itself is
    - dynamically linked with **Boost.Graph** shared-library.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Traditional CMake vs. Modern CMake vs. More Modern CMake

demonstrated with example-project "Calculator app"

- A *free* executable `FreeCalculator`
  - dynamically linked with **Boost.Program_Options** shared-library.
  - dynamically linked with `basicmath` shared-library.
- A *non-free* executable `PremiumCalculator` (with extended functionality)
  - dynamically linked with **Boost.Program_Options** shared-library.
  - dynamically linked with `extmath` shared-library, which itself is
    - dynamically linked with **Boost.Graph** shared-library.
- All executables and libraries use **Boost.Outcome** header-only library internally.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Finding external dependency

Boost

Finding external dependency - *Boost*

Traditional and *Modern CMake* way

```cmake
# ./external/boost/CMakeLists.txt -- Traditional/Modern CMake

set( BOOST_VERSION 1.58.0 )

# Settings for finding correct Boost libraries.
set( Boost_USE_STATIC_LIBS       FALSE )
set( Boost_USE_MULTITHREADED     TRUE )
set( Boost_USE_STATIC_RUNTIME    FALSE )
set( Boost_ADDITIONAL_VERSIONS   "${BOOST_VERSION}" )
set( Boost_COMPILER              "-gcc" )

# Search for Boost libraries.
find_package( Boost ${BOOST_VERSION} EXACT REQUIRED
    COMPONENTS program_options
               graph )
```

- Using `find_package` to locate *Boost*.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Finding external dependency - *Boost*

Traditional and *Modern CMake* way

```
# ./external/boost/CMakeLists.txt -- Traditional/Modern CMake

set( BOOST_VERSION 1.58.0 )

# Settings for finding correct Boost libraries.
set( Boost_USE_STATIC_LIBS       FALSE )
set( Boost_USE_MULTITHREADED     TRUE )
set( Boost_USE_STATIC_RUNTIME    FALSE )
set( Boost_ADDITIONAL_VERSIONS   "${BOOST_VERSION}" )
set( Boost_COMPILER              "-gcc" )

# Search for Boost libraries.
find_package( Boost ${BOOST_VERSION} EXACT REQUIRED
    COMPONENTS program_options
               graph )
```

- Using `find_package` to locate *Boost*.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Finding external dependency - *Boost*

Traditional and *Modern CMake* way

```
# ./external/boost/CMakeLists.txt -- Traditional/Modern CMake

set( BOOST_VERSION 1.58.0 )

# Settings for finding correct Boost libraries.
set( Boost_USE_STATIC_LIBS       FALSE )
set( Boost_USE_MULTITHREADED     TRUE )
set( Boost_USE_STATIC_RUNTIME    FALSE )
set( Boost_ADDITIONAL_VERSIONS   "${BOOST_VERSION}" )
set( Boost_COMPILER              "-gcc" )

# Search for Boost libraries.
find_package( Boost ${BOOST_VERSION} EXACT REQUIRED
    COMPONENTS program_options
               graph )
```

- Using `find_package` to locate *Boost*.
- If found, sets variables:
  - `Boost_INCLUDE_DIRS`, containing include-path
  - `Boost_LIBRARIES`, containing file-paths to shared libraries

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

FINDING EXTERNAL DEPENDENCY - *BOOST*

TRADITIONAL AND *MODERN CMAKE* WAY

```
# ./external/boost/CMakeLists.txt -- Traditional/Modern CMake

set( BOOST_VERSION 1.58.0 )

# Settings for finding correct Boost libraries.
set( Boost_USE_STATIC_LIBS       FALSE )
set( Boost_USE_MULTITHREADED     TRUE )
set( Boost_USE_STATIC_RUNTIME    FALSE )
set( Boost_ADDITIONAL_VERSIONS   "${BOOST_VERSION}" )
set( Boost_COMPILER              "-gcc" )

# Search for Boost libraries.
find_package( Boost ${BOOST_VERSION} EXACT REQUIRED
    COMPONENTS program_options
               graph )
```

- Using `find_package` to locate *Boost*.
- If found, sets variables:
  - `Boost_INCLUDE_DIRS`, containing include-path
  - `Boost_LIBRARIES`, containing file-paths to shared libraries
- *Modern CMake* additionally creates `IMPORTED` targets:
  - `Boost::boost`
  - `Boost::program_options`
  - `Boost::graph`

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Finding external dependency - *Boost*

Traditional and *Modern CMake* way

```
# ./external/boost/CMakeLists.txt -- Traditional/Modern CMake

set( BOOST_VERSION 1.58.0 )

# Settings for finding correct Boost libraries.
set( Boost_USE_STATIC_LIBS          FALSE )
set( Boost_USE_MULTITHREADED        TRUE )
set( Boost_USE_STATIC_RUNTIME       FALSE )
set( Boost_ADDITIONAL_VERSIONS      "${BOOST_VERSION}" )
set( Boost_COMPILER                 "-gcc" )

# Search for Boost libraries.
find_package( Boost ${BOOST_VERSION} EXACT REQUIRED
    COMPONENTS program_options
               graph )
```

- Using `find_package` to locate *Boost*.
- If found, sets variables:
  - `Boost_INCLUDE_DIRS`, containing include-path
  - `Boost_LIBRARIES`, containing file-paths to shared libraries
- *Modern CMake* additionally creates `IMPORTED` targets:
  - `Boost::boost`
  - `Boost::program_options`
  - `Boost::graph`
- `IMPORTED` targets carry and can propagate their *usage-requirements*.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Finding external dependency - *Boost*

Traditional and *Modern CMake* way

- Variables and `IMPORTED` targets created by `find_package` have **non-global** scope (by convention).

```cmake
#./CMakeLists.txt

cmake_minimum_required( VERSION 2.8.10 )

project( Example_for_CMake )
set( VERSION 2.8.10 )
set( DESCRIPTION "Example project for CMake" )

 # Always use '-fPIC'/'-fPIE' option.
set( CMAKE_POSITION_INDEPENDENT_CODE ON )

# Make external libraries globally available.
add_subdirectory( external )  # Does NOT work for Boost!


# Create targets for building the (local) libraries.
add_subdirectory( library )

# Create the targets for the entire example-app.
add_subdirectory( app )
```

```cmake
# ./external/CMakeLists.txt

add_subdirectory( boost )
add_subdirectory( boost_outcome )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

FINDING EXTERNAL DEPENDENCY - *BOOST*

TRADITIONAL AND *MODERN CMAKE* WAY

- Variables and `IMPORTED` targets created by `find_package` have **non-global** scope (by convention).

```
#./CMakeLists.txt

cmake_minimum_required( VERSION 2.8.10 )

project( Example_for_CMake )
set( VERSION 2.8.10 )
set( DESCRIPTION "Example project for CMake" )

 # Always use '-fPIC'/'-fPIE' option.
set( CMAKE_POSITION_INDEPENDENT_CODE ON )

# Make external libraries globally available.
add_subdirectory( external )  # Does NOT work for Boost!


# Create targets for building the (local) libraries.
add_subdirectory( library )

# Create the targets for the entire example-app.
add_subdirectory( app )
```

```
# ./external/CMakeLists.txt

add_subdirectory( boost )
add_subdirectory( boost_outcome )
```

- Using `add_subdirectory` is preferred but does not help here.

Finding external dependency - *Boost*

Traditional and *Modern CMake* way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- Variables and `IMPORTED` targets created by `find_package` have **non-global** scope (by convention).

```cmake
#./CMakeLists.txt -- Traditional/Modern CMake

cmake_minimum_required( VERSION 2.8.10 )

project( Example_for_CMake )
set( VERSION 2.8.10 )
set( DESCRIPTION "Example project for CMake" )

 # Always use '-fPIC'/'-fPIE' option.
set( CMAKE_POSITION_INDEPENDENT_CODE ON )

# Make external libraries globally available.
add_subdirectory( external )
include( external/boost/CMakeLists.txt )

# Create targets for building the (local) libraries.
add_subdirectory( library )

# Create the targets for the entire example-app.
add_subdirectory( app )
```

```cmake
# ./external/CMakeLists.txt -- Traditional/Modern CMake


add_subdirectory( boost_outcome )
```

- Using `add_subdirectory` is preferred but does not help here.
- `CMakeLists.txt` file for *Boost* needs to be `include`d into the top `CMakeLists.txt` file directly.

Finding external dependency - *Boost*

Traditional and *Modern CMake* way

- Variables and `IMPORTED` targets created by `find_package` have **non-global** scope (by convention).

```
#./CMakeLists.txt -- Traditional/Modern CMake

cmake_minimum_required( VERSION 2.8.10 )

project( Example_for_CMake )
set( VERSION 2.8.10 )
set( DESCRIPTION "Example project for CMake" )

 # Always use '-fPIC'/'-fPIE' option.
set( CMAKE_POSITION_INDEPENDENT_CODE ON )

# Make external libraries globally available.
add_subdirectory( external )
include( external/boost/CMakeLists.txt )

# Create targets for building the (local) libraries.
add_subdirectory( library )

# Create the targets for the entire example-app.
add_subdirectory( app )
```

```
# ./external/CMakeLists.txt -- Traditional/Modern CMake


add_subdirectory( boost_outcome )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- Using `add_subdirectory` is preferred but does not help here.
- `CMakeLists.txt` file for *Boost* needs to be `include`d into the top `CMakeLists.txt` file directly.

However...

Finding external dependency - *Boost*

More Modern CMake way

- Since **CMake 3.11** `IMPORTED` targets can be promoted to **global** scope
  - by setting the `IMPORTED_GLOBAL` property to `TRUE`.

```
# ./external/boost/CMakeLists.txt -- Traditional/Modern CMake

set( BOOST_VERSION 1.58.0 )

# Settings for finding correct Boost libraries.
set( Boost_USE_STATIC_LIBS       FALSE )
set( Boost_USE_MULTITHREADED     TRUE )
set( Boost_USE_STATIC_RUNTIME    FALSE )
set( Boost_ADDITIONAL_VERSIONS   "${BOOST_VERSION}" )
set( Boost_COMPILER              "-gcc" )

# Search for Boost libraries.
find_package( Boost ${BOOST_VERSION} EXACT REQUIRED
    COMPONENTS program_options
               graph )
```

⇒

```
# ./external/boost/CMakeLists.txt -- More Modern CMake

set( BOOST_VERSION 1.58.0 )

# Settings for finding correct Boost libraries.
set( Boost_USE_STATIC_LIBS       FALSE )
set( Boost_USE_MULTITHREADED     TRUE )
set( Boost_USE_STATIC_RUNTIME    FALSE )
set( Boost_ADDITIONAL_VERSIONS   "${BOOST_VERSION}" )
set( Boost_COMPILER              "-gcc" )

# Search for Boost libraries.
find_package( Boost ${BOOST_VERSION} EXACT REQUIRED
    COMPONENTS program_options
               graph )

# Make found targets globally available.
if ( Boost_FOUND )
    set_target_properties( Boost::boost
                           Boost::program_options
                           Boost::graph
        PROPERTIES IMPORTED_GLOBAL TRUE )
endif ()
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Finding external dependency - *Boost*

More Modern CMake way

- Since **CMake 3.11** `IMPORTED` targets can be promoted to **global** scope
  - by setting the `IMPORTED_GLOBAL` property to `TRUE`.

```cmake
# ./external/boost/CMakeLists.txt -- Traditional/Modern CMake

set( BOOST_VERSION 1.58.0 )

# Settings for finding correct Boost libraries.
set( Boost_USE_STATIC_LIBS       FALSE )
set( Boost_USE_MULTITHREADED     TRUE )
set( Boost_USE_STATIC_RUNTIME    FALSE )
set( Boost_ADDITIONAL_VERSIONS   "${BOOST_VERSION}" )
set( Boost_COMPILER              "-gcc" )

# Search for Boost libraries.
find_package( Boost ${BOOST_VERSION} EXACT REQUIRED
    COMPONENTS program_options
               graph )
```

⇒

```cmake
# ./external/boost/CMakeLists.txt -- More Modern CMake

set( BOOST_VERSION 1.58.0 )

# Settings for finding correct Boost libraries.
set( Boost_USE_STATIC_LIBS       FALSE )
set( Boost_USE_MULTITHREADED     TRUE )
set( Boost_USE_STATIC_RUNTIME    FALSE )
set( Boost_ADDITIONAL_VERSIONS   "${BOOST_VERSION}" )
set( Boost_COMPILER              "-gcc" )

# Search for Boost libraries.
find_package( Boost ${BOOST_VERSION} EXACT REQUIRED
    COMPONENTS program_options
               graph )

# Make found targets globally available.
if ( Boost_FOUND )
    set_target_properties( Boost::boost
                           Boost::program_options
                           Boost::graph
        PROPERTIES IMPORTED_GLOBAL TRUE )
endif ()
```

- Promoted/Global `IMPORTED` targets *cannot* be demoted!

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Finding external dependency - *Boost*

More Modern CMake way

- With `IMPORTED` targets promoted to **global** scope, this works as desired.

```cmake
#./CMakeLists.txt -- More Modern CMake

cmake_minimum_required( VERSION 3.11 )

project( Example_for_CMake
         VERSION 3.11
         DESCRIPTION "Example project for CMake" )

 # Always use '-fPIC'/'-fPIE' option.
set( CMAKE_POSITION_INDEPENDENT_CODE ON )

# Make external libraries globally available.
add_subdirectory( external )  # Does also work for Boost!

# Create targets for building the (local) libraries.
add_subdirectory( library )

# Create the targets for the entire example-app.
add_subdirectory( app )
```

```cmake
# ./external/CMakeLists.txt -- More Modern CMake

add_subdirectory( boost )
add_subdirectory( boost_outcome )
```

- Using `add_subdirectory` is preferred and now works here.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Finding external dependency - *Boost*

More Modern CMake way

- With `IMPORTED` targets promoted to **global** scope, this works as desired.

```cmake
#./CMakeLists.txt -- More Modern CMake

cmake_minimum_required( VERSION 3.11 )

project( Example_for_CMake
         VERSION 3.11
         DESCRIPTION "Example project for CMake" )

 # Always use '-fPIC'/'-fPIE' option.
set( CMAKE_POSITION_INDEPENDENT_CODE ON )

# Make external libraries globally available.
add_subdirectory( external )  # Does also work for Boost!

# Create targets for building the (local) libraries.
add_subdirectory( library )

# Create the targets for the entire example-app.
add_subdirectory( app )
```

```cmake
# ./external/CMakeLists.txt -- More Modern CMake

add_subdirectory( boost )
add_subdirectory( boost_outcome )
```

- Using `add_subdirectory` is preferred and now works here.
  - You need to use the `IMPORTED` targets for *Boost*.
  - The variables created by `find_package` for *Boost* are not promoted to global scope!

Creating `IMPORTED` target for external dependency

Boost.Outcome

CREATING `IMPORTED` TARGET FOR EXTERNAL DEPENDENCY - *BOOST.OUTCOME*

TRADITIONAL AND *MODERN CMAKE* WAY

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```cmake
# ./external/boost_outcome/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( External.Reviewed_Boost.outcome )
set( VERSION 2.0 )
set( DESCRIPTION "Boost.Outcome header-only lib." )

# Provide target for "Boost.Outcome" library.
add_library( Boost::outcome UNKNOWN
    IMPORTED GLOBAL )
# Store include search-path containing headers
# of "Boost.Outcome".
set_target_properties( Boost::outcome PROPERTIES
    MY_INCLUDE_DIRS /opt/boost-outcome/include )
# Require at least compiling with C++14.
set_target_properties( Boost::outcome PROPERTIES
    MY_COMPILE_FEATURES "-std=c++14" )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Creating `IMPORTED` target for external dependency - *Boost.Outcome*

Traditional and *Modern CMake* way

```cmake
# ./external/boost_outcome/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( External.Reviewed_Boost.outcome )
set( VERSION 2.0 )
set( DESCRIPTION "Boost.Outcome header-only lib." )

# Provide target for "Boost.Outcome" library.
add_library( Boost::outcome UNKNOWN
    IMPORTED GLOBAL )
# Store include search-path containing headers
# of "Boost.Outcome".
set_target_properties( Boost::outcome PROPERTIES
    MY_INCLUDE_DIRS /opt/boost-outcome/include )
# Require at least compiling with C++14.
set_target_properties( Boost::outcome PROPERTIES
    MY_COMPILE_FEATURES "-std=c++14" )
```

⇒

```cmake
# ./external/boost_outcome/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( External.Reviewed_Boost.outcome
         VERSION 2.0
         DESCRIPTION "Boost.Outcome header-only lib." )

# Provide target for "Boost.Outcome" library.
add_library( Boost::outcome INTERFACE
    IMPORTED GLOBAL )
# Store include search-path containing headers
# of "Boost.Outcome".
set_target_properties( Boost::outcome PROPERTIES
    INTERFACE_INCLUDE_DIRS /opt/boost-outcome/include )
# Require at least compiling with C++14.
set_target_properties( Boost::outcome PROPERTIES
    INTERFACE_COMPILE_FEATURES cxx_std_14 )
```

- For *usage-requirements* the `INTERFACE_...` properties need to be set.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Creating `IMPORTED` target for external dependency - *Boost.Outcome*

Traditional and *Modern CMake* way

```cmake
# ./external/boost_outcome/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( External.Reviewed_Boost.outcome )
set( VERSION 2.0 )
set( DESCRIPTION "Boost.Outcome header-only lib." )

# Provide target for "Boost.Outcome" library.
add_library( Boost::outcome UNKNOWN
    IMPORTED GLOBAL )
# Store include search-path containing headers
# of "Boost.Outcome".
set_target_properties( Boost::outcome PROPERTIES
    MY_INCLUDE_DIRS /opt/boost-outcome/include )
# Require at least compiling with C++14.
set_target_properties( Boost::outcome PROPERTIES
    MY_COMPILE_FEATURES "-std=c++14" )
```

⇒

```cmake
# ./external/boost_outcome/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( External.Reviewed_Boost.outcome
         VERSION 2.0
         DESCRIPTION "Boost.Outcome header-only lib." )

# Provide target for "Boost.Outcome" library.
add_library( Boost::outcome INTERFACE
    IMPORTED GLOBAL )
# Store include search-path containing headers
# of "Boost.Outcome".
set_target_properties( Boost::outcome PROPERTIES
    INTERFACE_INCLUDE_DIRS /opt/boost-outcome/include )
# Require at least compiling with C++14.
set_target_properties( Boost::outcome PROPERTIES
    INTERFACE_COMPILE_FEATURES cxx_std_14 )
```

- For *usage-requirements* the `INTERFACE_...` properties need to be set.
- Sadly, using `target_...` commands does not work with `IMPORTED` libraries.

CREATING `IMPORTED` TARGET FOR EXTERNAL DEPENDENCY - *BOOST.OUTCOME*

TRADITIONAL AND *MODERN CMAKE* WAY

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```cmake
# ./external/boost_outcome/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( External.Reviewed_Boost.outcome )
set( VERSION 2.0 )
set( DESCRIPTION "Boost.Outcome header-only lib." )

# Provide target for "Boost.Outcome" library.
add_library( Boost::outcome UNKNOWN
    IMPORTED GLOBAL )
# Store include search-path containing headers
# of "Boost.Outcome".
set_target_properties( Boost::outcome PROPERTIES
    MY_INCLUDE_DIRS /opt/boost-outcome/include )
# Require at least compiling with C++14.
set_target_properties( Boost::outcome PROPERTIES
    MY_COMPILE_FEATURES "-std=c++14" )
```

⇒

```cmake
# ./external/boost_outcome/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( External.Reviewed_Boost.outcome
         VERSION 2.0
         DESCRIPTION "Boost.Outcome header-only lib." )

# Provide target for "Boost.Outcome" library.
add_library( Boost::outcome INTERFACE
    IMPORTED GLOBAL )
# Store include search-path containing headers
# of "Boost.Outcome".
set_target_properties( Boost::outcome PROPERTIES
    INTERFACE_INCLUDE_DIRS /opt/boost-outcome/include )
# Require at least compiling with C++14.
set_target_properties( Boost::outcome PROPERTIES
    INTERFACE_COMPILE_FEATURES cxx_std_14 )
```

- For *usage-requirements* the `INTERFACE_...` properties need to be set.
- Sadly, using `target_...` commands does not work with `IMPORTED` libraries.

HOWEVER...

Creating `IMPORTED` target for external dependency - *Boost.Outcome*

More Modern CMake way

- Since **CMake 3.11** `target_...` commands can be used to set *usage-requirements* of `IMPORTED` targets (as for all other targets).

```cmake
# ./external/boost_outcome/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( External.Reviewed_Boost.outcome
         VERSION 2.0
         DESCRIPTION "Boost.Outcome header-only lib." )

# Provide target for "Boost.Outcome" library.
add_library( Boost::outcome INTERFACE
    IMPORTED GLOBAL )
# Store include search-path containing headers
# of "Boost.Outcome".
set_target_properties( Boost::outcome PROPERTIES
    INTERFACE_INCLUDE_DIRS /opt/boost-outcome/include )
# Require at least compiling with C++14.
set_target_properties( Boost::outcome PROPERTIES
    INTERFACE_COMPILE_FEATURES cxx_std_14 )
```

⇒

```cmake
# ./external/boost_outcome/CMakeLists.txt -- More Modern CMake

cmake_minimum_required( VERSION 3.11 )

project( External.Reviewed_Boost.outcome
         VERSION 2.0
         DESCRIPTION "Boost.Outcome header-only lib." )

# Provide target for "Boost.Outcome" library.
add_library( Boost::outcome INTERFACE
    IMPORTED GLOBAL )
# Store include search-path containing headers
# of "Boost.Outcome".
target_include_directories( Boost::outcome SYSTEM
    INTERFACE /opt/boost-outcome/include )
# Require at least compiling with C++14.
target_compile_features( Boost::outcome
    INTERFACE cxx_std_14 )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

CREATING `IMPORTED` TARGET FOR EXTERNAL DEPENDENCY - *BOOST.OUTCOME*

MORE MODERN CMAKE WAY

- Since **CMake 3.11** `target_...` commands can be used to set *usage-requirements* of `IMPORTED` targets (as for all other targets).

```
# ./external/boost_outcome/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( External.Reviewed_Boost.outcome
         VERSION 2.0
         DESCRIPTION "Boost.Outcome header-only lib." )

# Provide target for "Boost.Outcome" library.
add_library( Boost::outcome INTERFACE
    IMPORTED GLOBAL )
# Store include search-path containing headers
# of "Boost.Outcome".
set_target_properties( Boost::outcome PROPERTIES
    INTERFACE_INCLUDE_DIRS /opt/boost-outcome/include )
# Require at least compiling with C++14.
set_target_properties( Boost::outcome PROPERTIES
    INTERFACE_COMPILE_FEATURES cxx_std_14 )
```

⇒

```
# ./external/boost_outcome/CMakeLists.txt -- More Modern CMake

cmake_minimum_required( VERSION 3.11 )

project( External.Reviewed_Boost.outcome
         VERSION 2.0
         DESCRIPTION "Boost.Outcome header-only lib." )

# Provide target for "Boost.Outcome" library.
add_library( Reviewed_Boost.outcome INTERFACE
    IMPORTED GLOBAL )
# Store include search-path containing headers
# of "Boost.Outcome".
target_include_directories( Reviewed_Boost.outcome SYSTEM
    INTERFACE /opt/boost-outcome/include )
# Require at least compiling with C++14.
target_compile_features( Reviewed_Boost.outcome
    INTERFACE cxx_std_14 )

# Create an alias for "Boost.Outcome".
add_library( Boost::outcome ALIAS Reviewed_Boost.outcome )
```

- Even `ALIAS`ing `IMPORTED` targets is now possible.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Creating libraries

basicmath / extmath

Creating libraries — an overview

Traditional CMake way

```
01 # ./library/CMakeLists.txt -- Traditional CMake
02
03 cmake_minimum_required( VERSION 2.8.10 )
04
05 project( MathLibs )
06 set( VERSION 1.0.0 )
07 set( DESCRIPTION "The internal math-libraries." )
08
09 # Sources for common functionality, used in both math-libraries.
10 set( BASIC_SOURCES
11     "src/BasicMath.cpp"
12     "src/HeavyMath.cpp"  # Takes loooooong to compile!
13     "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
14     "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
15 # An OBJECT-library, used to only compile common sources once!
16 add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )
17
18 # Required include search-paths and constexpr support.
19 include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
20 set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
21     STRING "C++ compile-flags" FORCE )
22
23 # Requires "Boost.Outcome" (which has some requirements, too).
24 include_directories( SYSTEM
25     "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
26 get_target_property( props Boost::outcome MY_COMPILE_FEATURES )
27 set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${props}" CACHE
28     STRING "C++ compile-flags" FORCE )
29
30 # A shared library for basic-math functionality.
31 add_library( basicmath SHARED
32     $<TARGET_OBJECTS:basicmath_ObjLib> )
33 # The usage-requirements of the OBJECT-library (and its
34 # direct dependency) are already set in directory scope
35 # (above in lines 19 to 28).
36
37 # A shared library for advanced-math functionality.
38 add_library( extmath SHARED
39     "src/ExtendedMath.cpp"  # Premium-content!
40     $<TARGET_OBJECTS:basicmath_ObjLib> )
41 # The usage-requirements of the OBJECT-library (and its
42 # direct dependency) are already set in directory scope
43 # (above in lines 19 to 28).
44
45 # Target 'extmath' requires include-path and needs
46 # to link to library-file of its extra dependency
47 # "Boost.Graph".
48 include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
49 target_link_libraries( extmath ${Boost_LIBRARIES} )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

CREATING LIBRARIES — AN OVERVIEW

TRADITIONAL CMAKE WAY

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```
01  # ./library/CMakeLists.txt -- Traditional CMake
02
03  cmake_minimum_required( VERSION 2.8.10 )
04
05  project( MathLibs )
06  set( VERSION 1.0.0 )
07  set( DESCRIPTION "The internal math-libraries." )
08
09  # Sources for common functionality, used in both math-libraries.
10  set( BASIC_SOURCES
11      "src/BasicMath.cpp"
12      "src/HeavyMath.cpp"  # Takes loooooong to compile!
13      "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
14      "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
15  # An OBJECT-library, used to only compile common sources once!
16  add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )
17
18  # Required include search-paths and constexpr support.
19  include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
20  set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
21      STRING "C++ compile-flags" FORCE )
22
23  # Requires "Boost.Outcome" (which has some requirements, too).
24  include_directories( SYSTEM
25      "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
26  get_target_property( props Boost::outcome MY_COMPILE_FEATURES )
27  set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${props}" CACHE
28      STRING "C++ compile-flags" FORCE )
29
30  # A shared library for basic-math functionality.
31  add_library( basicmath SHARED
32      $<TARGET_OBJECTS:basicmath_ObjLib> )
33  # The usage-requirements of the OBJECT-library (and its
34  # direct dependency) are already set in directory scope
35  # (above in lines 19 to 28).
36
37  # A shared library for advanced-math functionality.
38  add_library( extmath SHARED
39      "src/ExtendedMath.cpp"  # Premium-content!
40      $<TARGET_OBJECTS:basicmath_ObjLib> )
41  # The usage-requirements of the OBJECT-library (and its
42  # direct dependency) are already set in directory scope
43  # (above in lines 19 to 28).
44
45  # Target 'extmath' requires include-path and needs
46  # to link to library-file of its extra dependency
47  # "Boost.Graph".
48  include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
49  target_link_libraries( extmath ${Boost_LIBRARIES} )
```

OVERWHELMING?

LET'S LOOK AT IT STEP BY STEP...

CREATING TARGETS AND ADDING SOURCES

TRADITIONAL AND *MODERN CMAKE* WAY

- First, create a common `OBJECT` library for both libraries
  - so that long-compiling files only need to be compiled once!

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```cmake
# ./library/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( MathLibs )
set( VERSION 1.0.0 )
set( DESCRIPTION "The internal math-libraries." )

# Sources for common functionality, used in both math-libraries.
set( BASIC_SOURCES
    "src/BasicMath.cpp"
    "src/HeavyMath.cpp"  # Takes loooooong to compile!
    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
# An OBJECT-library, used to only compile common sources once!
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )
...
```

Creating targets and adding sources

Traditional and *Modern CMake* way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- First, create a common `OBJECT` library for both libraries
  - so that long-compiling files only need to be compiled once!
- *Modern CMake* allows to add intention to the source-files, by using `target_sources` command.

```cmake
# ./library/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( MathLibs )
set( VERSION 1.0.0 )
set( DESCRIPTION "The internal math-libraries." )

# Sources for common functionality, used in both math-libraries.
set( BASIC_SOURCES
    "src/BasicMath.cpp"
    "src/HeavyMath.cpp"   # Takes loooooong to compile!
    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
# An OBJECT-library, used to only compile common sources once!
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )
...
```

⇒

```cmake
# ./library/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
...
```

Creating targets and adding sources

Traditional and *Modern CMake* way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- First, create a common `OBJECT` library for both libraries
  - so that long-compiling files only need to be compiled once!
- *Modern CMake* allows to add intention to the source-files, by using `target_sources` command.
- Sadly, `add_library` requires at least one source-file argument.

```cmake
# ./library/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( MathLibs )
set( VERSION 1.0.0 )
set( DESCRIPTION "The internal math-libraries." )

# Sources for common functionality, used in both math-libraries.
set( BASIC_SOURCES
    "src/BasicMath.cpp"
    "src/HeavyMath.cpp"  # Takes loooooong to compile!
    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
# An OBJECT-library, used to only compile common sources once!
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )
...
```

⇒

```cmake
# ./library/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
...
```

Creating targets and adding sources

Traditional and *Modern CMake* way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- First, create a common `OBJECT` library for both libraries
  - so that long-compiling files only need to be compiled once!
- *Modern CMake* allows to add intention to the source-files, by using `target_sources` command.
- Sadly, `add_library` requires at least one source-file argument.

However...

```
# ./library/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( MathLibs )
set( VERSION 1.0.0 )
set( DESCRIPTION "The internal math-libraries." )

# Sources for common functionality, used in both math-libraries.
set( BASIC_SOURCES
    "src/BasicMath.cpp"
    "src/HeavyMath.cpp"   # Takes loooooong to compile!
    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
# An OBJECT-library, used to only compile common sources once!
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )
...
```

⇒

```
# ./library/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
...
```

Creating targets and adding sources

More Modern CMake way

- Since **CMake 3.11** the `add_library` command can be used without any source-parameters.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```cmake
# ./library/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
...
```

⇒

```cmake
# ./library/CMakeLists.txt -- More Modern CMake

cmake_minimum_required( VERSION 3.11 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
...
```

Creating targets and adding sources

More Modern CMake way

- Since **CMake 3.11** the `add_library` command can be used without any source-parameters.
  - Of course, sources need to be added later via `target_sources` command, or CMake will complain.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```cmake
# ./library/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
...
```

⇒

```cmake
# ./library/CMakeLists.txt -- More Modern CMake

cmake_minimum_required( VERSION 3.11 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
...
```

Creating targets and adding sources

More Modern CMake way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- Since **CMake 3.11** the `add_library` command can be used without any source-parameters.
  - Of course, sources need to be added later via `target_sources` command, or CMake will complain.
- The same applies to `add_executable` command.

```cmake
# ./library/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
...
```

⇒

```cmake
# ./library/CMakeLists.txt -- More Modern CMake

cmake_minimum_required( VERSION 3.11 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
...
```

Adding build/usage-requirements to targets

Traditional and *Modern CMake* way

- `include_directories` adds include-search-paths on directory-scope
  - applies to all targets from that `CMakeLists.txt`

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```cmake
# ./library/CMakeLists.txt -- Traditional CMake
...
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )

# Required include search-paths and constexpr support.
include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
    STRING "C++ compile-flags" FORCE )
...
```

Adding build/usage-requirements to targets

Traditional and *Modern CMake* way

- `include_directories` adds include-search-paths on directory-scope
  - applies to all targets from that `CMakeLists.txt`

    ⇒ Not transitive, only sets *build-requirements*!

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```
# ./library/CMakeLists.txt -- Traditional CMake
...
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )

# Required include search-paths and constexpr support.
include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
    STRING "C++ compile-flags" FORCE )
...
```

Adding build/usage-requirements to targets

Traditional and *Modern CMake* way

- `include_directories` adds include-search-paths on directory-scope
  - applies to all targets from that `CMakeLists.txt`
    ⇒ Not transitive, only sets *build-requirements*!
- Compile-features have to be set by hand.
  - set in `CMAKE_CXX_FLAGS` cache-variable

```
# ./library/CMakeLists.txt -- Traditional CMake
...
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )

# Required include search-paths and constexpr support.
include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
    STRING "C++ compile-flags" FORCE )
...
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Adding build/usage-requirements to targets

Traditional and *Modern CMake* way

- `include_directories` adds include-search-paths on directory-scope
  - applies to all targets from that `CMakeLists.txt`

    ⇒ Not transitive, only sets *build-requirements*!
- Compile-features have to be set by hand.
  - set in `CMAKE_CXX_FLAGS` cache-variable

    ⇒ Not transitive, instead a **global** *build-requirement*!

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```
# ./library/CMakeLists.txt -- Traditional CMake
...
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )

# Required include search-paths and constexpr support.
include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
    STRING "C++ compile-flags" FORCE )
...
```

Adding build/usage-requirements to targets

Traditional and *Modern CMake* way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- `include_directories` adds include-search-paths on directory-scope
  - applies to all targets from that `CMakeLists.txt`

    ⇒ Not transitive, only sets *build-requirements*!
- Compile-features have to be set by hand.
  - set in `CMAKE_CXX_FLAGS` cache-variable

    ⇒ Not transitive, instead a **global** *build-requirement*!
- *Modern CMake* is more flexible (and explicit).
  - by using `target_include_directories` and `target_compile_features` commands.

```
# ./library/CMakeLists.txt -- Traditional CMake
...
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )

# Required include search-paths and constexpr support.
include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
    STRING "C++ compile-flags" FORCE )
...
```

⇒

```
# ./library/CMakeLists.txt -- Modern CMake
...
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
...
# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )
...
```

Adding usage-requirements from dependencies to `OBJECT` targets

Traditional and *Modern CMake* way

- *Usage-requirements* of dependencies need to become *build-requirements* of dependent targets using the same commands.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```
# ./library/CMakeLists.txt -- Traditional CMake
...
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )

# Required include search-paths and constexpr support.
include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
    STRING "C++ compile-flags" FORCE )

# Requires "Boost.Outcome" (which has some requirements, too).
include_directories( SYSTEM
    "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
get_target_property( props Boost::outcome MY_COMPILE_FEATURES )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${props}" CACHE
    STRING "C++ compile-flags" FORCE )
...
```

ADDING USAGE-REQUIREMENTS FROM DEPENDENCIES TO `OBJECT` TARGETS

TRADITIONAL AND *MODERN CMAKE* WAY

- *Usage-requirements* of dependencies need to become *build-requirements* of dependent targets using the same commands.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```
# ./library/CMakeLists.txt -- Traditional CMake
...
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )

# Required include search-paths and constexpr support.
include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
    STRING "C++ compile-flags" FORCE )

# Requires "Boost.Outcome" (which has some requirements, too).
include_directories( SYSTEM
    "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
get_target_property( props Boost::outcome MY_COMPILE_FEATURES )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${props}" CACHE
    STRING "C++ compile-flags" FORCE )
...
```

Adding usage-requirements from dependencies to `OBJECT` targets

Traditional and *Modern CMake* way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- *Usage-requirements* of dependencies need to become *build-requirements* of dependent targets using the same commands.
- Sadly, even with *Modern CMake*, `OBJECT` targets
  - cannot declare dependency on other targets, and therefore
  - need to add *usage-requirements* of dependencies by hand.

```cmake
# ./library/CMakeLists.txt -- Traditional CMake
...
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )

# Required include search-paths and constexpr support.
include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
    STRING "C++ compile-flags" FORCE )

# Requires "Boost.Outcome" (which has some requirements, too).
include_directories( SYSTEM
    "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
get_target_property( props Boost::outcome MY_COMPILE_FEATURES )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${props}" CACHE
    STRING "C++ compile-flags" FORCE )
...
```

⇒

```cmake
# ./library/CMakeLists.txt -- Modern CMake
...
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
...
# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )

# Requires "Boost.Outcome" (which has some requirements, too).
target_include_directories( basicmath_ObjLib SYSTEM
    PUBLIC
        $<TARGET_PROPERTY:Boost::outcome,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( basicmath_ObjLib
    PUBLIC $<TARGET_PROPERTY:Boost::outcome,INTERFACE_COMPILE_FEATURES> )
...
```

Adding usage-requirements from dependencies to `OBJECT` targets

Traditional and *Modern CMake* way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- *Usage-requirements* of dependencies need to become *build-requirements* of dependent targets using the same commands.
- Sadly, even with *Modern CMake*, `OBJECT` targets
  - cannot declare dependency on other targets, and therefore
  - need to add *usage-requirements* of dependencies by hand.

However...

```
# ./library/CMakeLists.txt -- Traditional CMake
...
add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )

# Required include search-paths and constexpr support.
include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
    STRING "C++ compile-flags" FORCE )

# Requires "Boost.Outcome" (which has some requirements, too).
include_directories( SYSTEM
    "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
get_target_property( props Boost::outcome MY_COMPILE_FEATURES )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${props}" CACHE
    STRING "C++ compile-flags" FORCE )
...
```

⇒

```
# ./library/CMakeLists.txt -- Modern CMake
...
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
...
# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )

# Requires "Boost.Outcome" (which has some requirements, too).
target_include_directories( basicmath_ObjLib SYSTEM
    PUBLIC
        $<TARGET_PROPERTY:Boost::outcome,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( basicmath_ObjLib
    PUBLIC $<TARGET_PROPERTY:Boost::outcome,INTERFACE_COMPILE_FEATURES> )
...
```

Adding usage-requirements from dependencies to `OBJECT` targets

More Modern CMake way

- Since **CMake 3.12** the `target_link_libraries` command can be used with `OBJECT` targets.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```
# ./library/CMakeLists.txt -- Modern CMake
...
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
...
# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )

# Requires "Boost.Outcome" (which has some requirements, too).
target_include_directories( basicmath_ObjLib SYSTEM
    PUBLIC
        $<TARGET_PROPERTY:Boost::outcome,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( basicmath_ObjLib
    PUBLIC $<TARGET_PROPERTY:Boost::outcome,INTERFACE_COMPILE_FEATURES> )
...
```

⇒

```
# ./library/CMakeLists.txt -- More Modern CMake
...
add_library( basicmath_ObjLib OBJECT )
...
# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )

# Requires "Boost.Outcome" (which has some requirements, too).
target_link_libraries( basicmath_ObjLib PUBLIC Boost::outcome )
...
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Adding usage-requirements from dependencies to `OBJECT` targets

More Modern CMake way

- Since **CMake 3.12** the `target_link_libraries` command can be used with `OBJECT` targets.
  - *Usage-requirements* of dependencies can be propagated to `OBJECT` targets.

```
# ./library/CMakeLists.txt -- Modern CMake
...
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
...
# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )

# Requires "Boost.Outcome" (which has some requirements, too).
target_include_directories( basicmath_ObjLib SYSTEM
    PUBLIC
        $<TARGET_PROPERTY:Boost::outcome,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( basicmath_ObjLib
    PUBLIC $<TARGET_PROPERTY:Boost::outcome,INTERFACE_COMPILE_FEATURES> )
...
```

⇒

```
# ./library/CMakeLists.txt -- More Modern CMake
...
add_library( basicmath_ObjLib OBJECT )
...
# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )

# Requires "Boost.Outcome" (which has some requirements, too).
target_link_libraries( basicmath_ObjLib PUBLIC Boost::outcome )
...
```

Adding usage-requirements from dependencies to `OBJECT` targets

More Modern CMake way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- Since **CMake 3.12** the `target_link_libraries` command can be used with `OBJECT` targets.
  - *Usage-requirements* of dependencies can be propagated to `OBJECT` targets.
  - But, the other way around, propagation of *usage-requirements* of `OBJECT` targets to other targets, is more interesting.

```cmake
# ./library/CMakeLists.txt -- Modern CMake
...
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
...
# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )

# Requires "Boost.Outcome" (which has some requirements, too).
target_include_directories( basicmath_ObjLib SYSTEM
    PUBLIC
        $<TARGET_PROPERTY:Boost::outcome,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( basicmath_ObjLib
    PUBLIC $<TARGET_PROPERTY:Boost::outcome,INTERFACE_COMPILE_FEATURES> )
...
```

⇒

```cmake
# ./library/CMakeLists.txt -- More Modern CMake
...
add_library( basicmath_ObjLib OBJECT )
...
# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )

# Requires "Boost.Outcome" (which has some requirements, too).
target_link_libraries( basicmath_ObjLib PUBLIC Boost::outcome )
...
```

Adding usage-requirements from dependencies to `OBJECT` targets

More Modern CMake way

- Since **CMake 3.12** the `target_link_libraries` command can be used with `OBJECT` targets.
  - *Usage-requirements* of dependencies can be propagated to `OBJECT` targets.
  - But, the other way around, propagation of *usage-requirements* of `OBJECT` targets to other targets, is more interesting.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```cmake
09 # ./library/CMakeLists.txt -- Modern CMake
10 ...
11 add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
17 ...
18 # Required include search-paths and constexpr support.
19 target_include_directories( basicmath_ObjLib
20     PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
21 target_compile_features( basicmath_ObjLib
22     PUBLIC cxx_constexpr )
23
24 # Requires "Boost.Outcome" (which has some requirements, too).
25 target_include_directories( basicmath_ObjLib SYSTEM
26     PUBLIC
27         $<TARGET_PROPERTY:Boost::outcome,INTERFACE_INCLUDE_DIRECTORIES> )
28 target_compile_features( basicmath_ObjLib
29     PUBLIC $<TARGET_PROPERTY:Boost::outcome,INTERFACE_COMPILE_FEATURES> )
30 ...
```

⇒

```cmake
09 # ./library/CMakeLists.txt -- More Modern CMake
10 ...
11 add_library( basicmath_ObjLib OBJECT )
17 ...
18 # Required include search-paths and constexpr support.
19 target_include_directories( basicmath_ObjLib
20     PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
21 target_compile_features( basicmath_ObjLib
22     PUBLIC cxx_constexpr )
23
24 # Requires "Boost.Outcome" (which has some requirements, too).
25 target_link_libraries( basicmath_ObjLib PUBLIC Boost::outcome )
26 ...
```

Let's have a look...

`OBJECT` LIBRARIES

SOME PECULIARITIES

Linking `OBJECT` library targets — a Quiz

```cmake
# A common object-library target.
add_library(                commonObjLib OBJECT )
target_sources(             commonObjLib PRIVATE "common.cpp" )
target_include_directories( commonObjLib PUBLIC  "include/common" )

# A shared-library target.
add_library( sharedLib1 SHARED )

# Another object-library target...
add_library(                objLib2 OBJECT )
target_sources(             objLib2 PRIVATE "source2.cpp" )
target_include_directories( objLib2 PUBLIC  "include2" )
# ... and yet another one.
add_library(                objLib3 OBJECT )
target_sources(             objLib3 PRIVATE "source3.cpp" )
target_include_directories( objLib3 PUBLIC  "include3" )

# Dependency on 'commonObjLib'.
target_link_libraries( sharedLib1 PUBLIC    commonObjLib )
target_link_libraries( objLib2    PRIVATE   commonObjLib )
target_link_libraries( objLib3    INTERFACE commonObjLib )

# Create executables linked against these targets.
add_executable( exe1 )
add_executable( exe2 )
add_executable( exe3 )
target_sources( exe1 PRIVATE "main1.cpp" )
target_sources( exe2 PRIVATE "main2.cpp" )
target_sources( exe3 PRIVATE "main3.cpp" )
target_link_libraries( exe1 PRIVATE sharedLib1 )
target_link_libraries( exe2 PRIVATE objLib2 )
target_link_libraries( exe3 PRIVATE objLib3 )
```

Linking `OBJECT` library targets — a Quiz

```cmake
# A common object-library target.
add_library(                commonObjLib OBJECT )
target_sources(             commonObjLib PRIVATE "common.cpp" )
target_include_directories( commonObjLib PUBLIC  "include/common" )

# A shared-library target.
add_library( sharedLib1 SHARED )

# Another object-library target...
add_library(                objLib2 OBJECT )
target_sources(             objLib2 PRIVATE "source2.cpp" )
target_include_directories( objLib2 PUBLIC  "include2" )
# ... and yet another one.
add_library(                objLib3 OBJECT )
target_sources(             objLib3 PRIVATE "source3.cpp" )
target_include_directories( objLib3 PUBLIC  "include3" )

# Dependency on 'commonObjLib'.
target_link_libraries( sharedLib1 PUBLIC    commonObjLib )
target_link_libraries( objLib2    PRIVATE   commonObjLib )
target_link_libraries( objLib3    INTERFACE commonObjLib )

# Create executables linked against these targets.
add_executable( exe1 )
add_executable( exe2 )
add_executable( exe3 )
target_sources( exe1 PRIVATE "main1.cpp" )
target_sources( exe2 PRIVATE "main2.cpp" )
target_sources( exe3 PRIVATE "main3.cpp" )
target_link_libraries( exe1 PRIVATE sharedLib1 )
target_link_libraries( exe2 PRIVATE objLib2 )
target_link_libraries( exe3 PRIVATE objLib3 )
```

- ***Q:*** What (local) include-directories does each individual target (`exe1`, `exe2` and `exe3`) know during compilation?

Linking `OBJECT` library targets — a Quiz

```cmake
# A common object-library target.
add_library(                commonObjLib OBJECT )
target_sources(             commonObjLib PRIVATE "common.cpp" )
target_include_directories( commonObjLib PUBLIC  "include/common" )

# A shared-library target.
add_library( sharedLib1 SHARED )

# Another object-library target...
add_library(                objLib2 OBJECT )
target_sources(             objLib2 PRIVATE "source2.cpp" )
target_include_directories( objLib2 PUBLIC  "include2" )
# ... and yet another one.
add_library(                objLib3 OBJECT )
target_sources(             objLib3 PRIVATE "source3.cpp" )
target_include_directories( objLib3 PUBLIC  "include3" )

# Dependency on 'commonObjLib'.
target_link_libraries( sharedLib1 PUBLIC    commonObjLib )
target_link_libraries( objLib2    PRIVATE   commonObjLib )
target_link_libraries( objLib3    INTERFACE commonObjLib )

# Create executables linked against these targets.
add_executable( exe1 )
add_executable( exe2 )
add_executable( exe3 )
target_sources( exe1 PRIVATE "main1.cpp" )
target_sources( exe2 PRIVATE "main2.cpp" )
target_sources( exe3 PRIVATE "main3.cpp" )
target_link_libraries( exe1 PRIVATE sharedLib1 )
target_link_libraries( exe2 PRIVATE objLib2 )
target_link_libraries( exe3 PRIVATE objLib3 )
```

- ***Q:*** What (local) include-directories does each individual target (`exe1`, `exe2` and `exe3`) know during compilation?
- ***A:***

  `exe1` — `.` and `include/common`

  `exe2` — `.` and `include2`

  `exe3` — `.` and `include3` and `include/common`

Linking `OBJECT` library targets — a Quiz

```cmake
# A common object-library target.
add_library(                 commonObjLib OBJECT )
target_sources(              commonObjLib PRIVATE "common.cpp" )
target_include_directories( commonObjLib PUBLIC  "include/common" )

# A shared-library target.
add_library( sharedLib1 SHARED )

# Another object-library target...
add_library(                 objLib2 OBJECT )
target_sources(              objLib2 PRIVATE "source2.cpp" )
target_include_directories( objLib2 PUBLIC  "include2" )
# ... and yet another one.
add_library(                 objLib3 OBJECT )
target_sources(              objLib3 PRIVATE "source3.cpp" )
target_include_directories( objLib3 PUBLIC  "include3" )

# Dependency on 'commonObjLib'.
target_link_libraries( sharedLib1 PUBLIC    commonObjLib )
target_link_libraries( objLib2    PRIVATE   commonObjLib )
target_link_libraries( objLib3    INTERFACE commonObjLib )

# Create executables linked against these targets.
add_executable( exe1 )
add_executable( exe2 )
add_executable( exe3 )
target_sources( exe1 PRIVATE "main1.cpp" )
target_sources( exe2 PRIVATE "main2.cpp" )
target_sources( exe3 PRIVATE "main3.cpp" )
target_link_libraries( exe1 PRIVATE sharedLib1 )
target_link_libraries( exe2 PRIVATE objLib2 )
target_link_libraries( exe3 PRIVATE objLib3 )
```

- ***Q:*** What (local) include-directories does each individual target (`exe1`, `exe2` and `exe3`) know during compilation?
- ***A:***

  `exe1` — `.` and `include/common`

  `exe2` — `.` and `include2`

  `exe3` — `.` and `include3` and `include/common`
- ***Q:*** The generated output-binary of which executable target (`exe1`, `exe2` or `exe3`) contains the *object-files* for `common.cpp`?

Linking `OBJECT` library targets — a Quiz

```
# A common object-library target.
add_library(                commonObjLib OBJECT )
target_sources(             commonObjLib PRIVATE "common.cpp" )
target_include_directories( commonObjLib PUBLIC  "include/common" )

# A shared-library target.
add_library( sharedLib1 SHARED )

# Another object-library target...
add_library(                objLib2 OBJECT )
target_sources(             objLib2 PRIVATE "source2.cpp" )
target_include_directories( objLib2 PUBLIC  "include2" )
# ... and yet another one.
add_library(                objLib3 OBJECT )
target_sources(             objLib3 PRIVATE "source3.cpp" )
target_include_directories( objLib3 PUBLIC  "include3" )

# Dependency on 'commonObjLib'.
target_link_libraries( sharedLib1 PUBLIC    commonObjLib )
target_link_libraries( objLib2    PRIVATE   commonObjLib )
target_link_libraries( objLib3    INTERFACE commonObjLib )

# Create executables linked against these targets.
add_executable( exe1 )
add_executable( exe2 )
add_executable( exe3 )
target_sources( exe1 PRIVATE "main1.cpp" )
target_sources( exe2 PRIVATE "main2.cpp" )
target_sources( exe3 PRIVATE "main3.cpp" )
target_link_libraries( exe1 PRIVATE sharedLib1 )
target_link_libraries( exe2 PRIVATE objLib2 )
target_link_libraries( exe3 PRIVATE objLib3 )
```

- ***Q:*** What (local) include-directories does each individual target (`exe1`, `exe2` and `exe3`) know during compilation?
- ***A:***
  `exe1` — `.` and `include/common`
  `exe2` — `.` and `include2`
  `exe3` — `.` and `include3` and `include/common`
- ***Q:*** The generated output-binary of which executable target (`exe1`, `exe2` or `exe3`) contains the *object-files* for `common.cpp`?
- ***A:***
  None! — Only the output-binary of `sharedLib` contains the object-files received from `commonObjLib`.

Linking `OBJECT` library targets — a Quiz

```cmake
# A common object-library target.
add_library(                 commonObjLib OBJECT )
target_sources(              commonObjLib PRIVATE "common.cpp" )
target_include_directories( commonObjLib PUBLIC  "include/common" )

# A shared-library target.
add_library( sharedLib1 SHARED )

# Another object-library target...
add_library(                 objLib2 OBJECT )
target_sources(              objLib2 PRIVATE "source2.cpp" )
target_include_directories( objLib2 PUBLIC  "include2" )
# ... and yet another one.
add_library(                 objLib3 OBJECT )
target_sources(              objLib3 PRIVATE "source3.cpp" )
target_include_directories( objLib3 PUBLIC  "include3" )

# Dependency on 'commonObjLib'.
target_link_libraries( sharedLib1 PUBLIC    commonObjLib )
target_link_libraries( objLib2    PRIVATE   commonObjLib )
target_link_libraries( objLib3    INTERFACE commonObjLib )

# Create executables linked against these targets.
add_executable( exe1 )
add_executable( exe2 )
add_executable( exe3 )
target_sources( exe1 PRIVATE "main1.cpp" )
target_sources( exe2 PRIVATE "main2.cpp" )
target_sources( exe3 PRIVATE "main3.cpp" )
target_link_libraries( exe1 PRIVATE sharedLib1 )
target_link_libraries( exe2 PRIVATE objLib2 )
target_link_libraries( exe3 PRIVATE objLib3 )
```

- *Q:* What (local) include-directories does each individual target (`exe1`, `exe2` and `exe3`) know during compilation?
- *A:*
  `exe1` — `.` and `include/common`
  `exe2` — `.` and `include2`
  `exe3` — `.` and `include3` and `include/common`
- *Q:* The generated output-binary of which executable target (`exe1`, `exe2` or `exe3`) contains the *object-files* for `common.cpp`?
- *A:*
  None! — Only the output-binary of `sharedLib` contains the object-files received from `commonObjLib`.

Why is that so?

Rules for linking `OBJECT` library targets

Propagation of usage-requirements

```
# Knowing usage-requirements from 'anyTarget' when compiling to object-files,
# but not further propagating them to other targets.
target_link_libraries( objTarget PRIVATE anyTarget )
```

```
# Not knowing usage-requirements from 'anyTarget' when compiling to object-files,
# but further propagating them to other targets.
target_link_libraries( objTarget INTERFACE anyTarget )
```

```
# Knowing usage-requirements from 'anyTarget' when compiling to object-files,
# and further propagating them to other targets.
target_link_libraries( objTarget PUBLIC anyTarget )
```

- *Usage-requirements* of any target (even `OBJECT` library targets) on the *right-hand-side* of `target_link_libraries` are propagated to the `OBJECT` library target on the left-hand-side.

Rules for linking `OBJECT` library targets

Propagation of usage-requirements

```
# Knowing usage-requirements from 'anyTarget' when compiling to object-files,
# but not further propagating them to other targets.
target_link_libraries( objTarget PRIVATE anyTarget )
```

- *usage-requirements* of `anyTarget` ⇒ *build-requirements* of `objTarget`

```
# Not knowing usage-requirements from 'anyTarget' when compiling to object-files,
# but further propagating them to other targets.
target_link_libraries( objTarget INTERFACE anyTarget )
```

```
# Knowing usage-requirements from 'anyTarget' when compiling to object-files,
# and further propagating them to other targets.
target_link_libraries( objTarget PUBLIC anyTarget )
```

- *Usage-requirements* of any target (even `OBJECT` library targets) on the *right-hand-side* of `target_link_libraries` are propagated to the `OBJECT` library target on the left-hand-side.

Rules for linking `OBJECT` library targets

Propagation of usage-requirements

```
# Knowing usage-requirements from 'anyTarget' when compiling to object-files,
# but not further propagating them to other targets.
target_link_libraries( objTarget PRIVATE anyTarget )
```

- *usage-requirements* of `anyTarget` ⇒ *build-requirements* of `objTarget`

```
# Not knowing usage-requirements from 'anyTarget' when compiling to object-files,
# but further propagating them to other targets.
target_link_libraries( objTarget INTERFACE anyTarget )
```

- *usage-requirements* of `anyTarget` ⇒ *usage-requirements* of `objTarget`

```
# Knowing usage-requirements from 'anyTarget' when compiling to object-files,
# and further propagating them to other targets.
target_link_libraries( objTarget PUBLIC anyTarget )
```

- *Usage-requirements* of any target (even `OBJECT` library targets) on the *right-hand-side* of `target_link_libraries` are propagated to the `OBJECT` library target on the left-hand-side.

Rules for linking `OBJECT` library targets

Propagation of usage-requirements

```
# Knowing usage-requirements from 'anyTarget' when compiling to object-files,
# but not further propagating them to other targets.
target_link_libraries( objTarget PRIVATE anyTarget )
```

- *usage-requirements* of `anyTarget` ⇒ *build-requirements* of `objTarget`

```
# Not knowing usage-requirements from 'anyTarget' when compiling to object-files,
# but further propagating them to other targets.
target_link_libraries( objTarget INTERFACE anyTarget )
```

- *usage-requirements* of `anyTarget` ⇒ *usage-requirements* of `objTarget`

```
# Knowing usage-requirements from 'anyTarget' when compiling to object-files,
# and further propagating them to other targets.
target_link_libraries( objTarget PUBLIC anyTarget )
```

- *usage-requirements* of `anyTarget` ⇒ *usage/build-requirements* of `objTarget`

- *Usage-requirements* of any target (even `OBJECT` library targets) on the *right-hand-side* of `target_link_libraries` are propagated to the `OBJECT` library target on the left-hand-side.

Rules for linking `OBJECT` library targets

Propagation of usage-requirements

```
# Knowing usage-requirements from 'anyTarget' when compiling to object-files,
# but not further propagating them to other targets.
target_link_libraries( objTarget PRIVATE anyTarget )
```

- *usage-requirements* of `anyTarget` ⇒ *build-requirements* of `objTarget`

```
# Not knowing usage-requirements from 'anyTarget' when compiling to object-files,
# but further propagating them to other targets.
target_link_libraries( objTarget INTERFACE anyTarget )
```

- *usage-requirements* of `anyTarget` ⇒ *usage-requirements* of `objTarget`

```
# Knowing usage-requirements from 'anyTarget' when compiling to object-files,
# and further propagating them to other targets.
target_link_libraries( objTarget PUBLIC anyTarget )
```

- *usage-requirements* of `anyTarget` ⇒ *usage/build-requirements* of `objTarget`

- *Usage-requirements* of any target (even `OBJECT` library targets) on the *right-hand-side* of `target_link_libraries` are propagated to the `OBJECT` library target on the left-hand-side.
- **The same as for all other target types.**

Rules for linking `OBJECT` library targets

Propagation of object-files

```
# Object-files are only propagated to the build-requirements of another target
# if that other target is not an OBJECT library target itself!
target_link_libraries( nonObjTarget   PRIVATE objTarget )  # Directly propagate object-files.
target_link_libraries( otherObjTarget PRIVATE objTarget )  # No propagation of object-files.
```

```
# No object-files can be propagated to the usage-requirements of any other target!
target_link_libraries( anyTarget INTERFACE objTarget )  # No indirect propagation of object-files.
```

```
# Object-files are propagated exactly as for PRIVATE propagation of object-files.
target_link_libraries( nonObjTarget   PUBLIC objTarget )  # Directly propagate object-files.
target_link_libraries( otherObjTarget PUBLIC objTarget )  # No propagation of object-files.
```

- *Object-files* are only ever propagated to ***direct*** dependants!
  - And only, if that direct dependant is not an `OBJECT` library target itself.

Rules for linking `OBJECT` library targets

Propagation of object-files

```
# Object-files are only propagated to the build-requirements of another target
# if that other target is not an OBJECT library target itself!
target_link_libraries( nonObjTarget   PRIVATE objTarget )  # Directly propagate object-files.
target_link_libraries( otherObjTarget PRIVATE objTarget )  # No propagation of object-files.
```

- *object-files* of `objTarget` $\Rightarrow$ *build-requirements* of `nonObjTarget`
- *object-files* of `objTarget` $\not\Rightarrow$ *usage/build-requirements* of `otherObjTarget`

```
# No object-files can be propagated to the usage-requirements of any other target!
target_link_libraries( anyTarget INTERFACE objTarget )  # No indirect propagation of object-files.
```

```
# Object-files are propagated exactly as for PRIVATE propagation of object-files.
target_link_libraries( nonObjTarget   PUBLIC objTarget )  # Directly propagate object-files.
target_link_libraries( otherObjTarget PUBLIC objTarget )  # No propagation of object-files.
```

- *Object-files* are only ever propagated to ***direct*** dependants!
  - And only, if that direct dependant is not an `OBJECT` library target itself.

Rules for linking `OBJECT` library targets

Propagation of object-files

```
# Object-files are only propagated to the build-requirements of another target
# if that other target is not an OBJECT library target itself!
target_link_libraries( nonObjTarget   PRIVATE objTarget )  # Directly propagate object-files.
target_link_libraries( otherObjTarget PRIVATE objTarget )  # No propagation of object-files.
```

- *object-files* of `objTarget` $\Rightarrow$ *build-requirements* of `nonObjTarget`
- *object-files* of `objTarget` $\not\Rightarrow$ *usage/build-requirements* of `otherObjTarget`

```
# No object-files can be propagated to the usage-requirements of any other target!
target_link_libraries( anyTarget INTERFACE objTarget )  # No indirect propagation of object-files.
```

- *object-files* of `objTarget` $\not\Rightarrow$ *usage/build-requirements* of `anyTarget`

```
# Object-files are propagated exactly as for PRIVATE propagation of object-files.
target_link_libraries( nonObjTarget   PUBLIC objTarget )  # Directly propagate object-files.
target_link_libraries( otherObjTarget PUBLIC objTarget )  # No propagation of object-files.
```

- *Object-files* are only ever propagated to ***direct*** dependants!
  - And only, if that direct dependant is not an `OBJECT` library target itself.

Rules for linking `OBJECT` library targets

Propagation of object-files

```
# Object-files are only propagated to the build-requirements of another target
# if that other target is not an OBJECT library target itself!
target_link_libraries( nonObjTarget   PRIVATE objTarget )  # Directly propagate object-files.
target_link_libraries( otherObjTarget PRIVATE objTarget )  # No propagation of object-files.
```

- *object-files* of `objTarget` $\Rightarrow$ *build-requirements* of `nonObjTarget`
- *object-files* of `objTarget` $\not\Rightarrow$ *usage/build-requirements* of `otherObjTarget`

```
# No object-files can be propagated to the usage-requirements of any other target!
target_link_libraries( anyTarget INTERFACE objTarget )  # No indirect propagation of object-files.
```

- *object-files* of `objTarget` $\not\Rightarrow$ *usage/build-requirements* of `anyTarget`

```
# Object-files are propagated exactly as for PRIVATE propagation of object-files.
target_link_libraries( nonObjTarget   PUBLIC objTarget )  # Directly propagate object-files.
target_link_libraries( otherObjTarget PUBLIC objTarget )  # No propagation of object-files.
```

- *object-files* of `objTarget` $\Rightarrow$ *build-requirements* of `nonObjTarget`
- *object-files* of `objTarget` $\not\Rightarrow$ *usage/build-requirements* of `otherObjTarget`
  - *Object-files* are only ever propagated to ***direct*** dependants!
    - And only, if that direct dependant is not an `OBJECT` library target itself.

Rules for linking `OBJECT` library targets

Propagation of object-files

```
# Object-files are only propagated to the build-requirements of another target
# if that other target is not an OBJECT library target itself!
target_link_libraries( nonObjTarget   PRIVATE objTarget )  # Directly propagate object-files.
target_link_libraries( otherObjTarget PRIVATE objTarget )  # No propagation of object-files.
```

- *object-files* of `objTarget` $\Rightarrow$ *build-requirements* of `nonObjTarget`
- *object-files* of `objTarget` $\not\Rightarrow$ *usage/build-requirements* of `otherObjTarget`

```
# No object-files can be propagated to the usage-requirements of any other target!
target_link_libraries( anyTarget INTERFACE objTarget )  # No indirect propagation of object-files.
```

- *object-files* of `objTarget` $\not\Rightarrow$ *usage/build-requirements* of `anyTarget`

```
# Object-files are propagated exactly as for PRIVATE propagation of object-files.
target_link_libraries( nonObjTarget   PUBLIC objTarget )  # Directly propagate object-files.
target_link_libraries( otherObjTarget PUBLIC objTarget )  # No propagation of object-files.
```

- *object-files* of `objTarget` $\Rightarrow$ *build-requirements* of `nonObjTarget`
- *object-files* of `objTarget` $\not\Rightarrow$ *usage/build-requirements* of `otherObjTarget`
  - *Object-files* are only ever propagated to ***direct*** dependants!
    - And only, if that direct dependant is not an `OBJECT` library target itself.

Creating libraries (cont.)

basicmath / extmath

CREATING LIBRARIES — AN OVERVIEW (CONT.)

TRADITIONAL CMAKE WAY

```
01  # ./library/CMakeLists.txt -- Traditional CMake
02
03  cmake_minimum_required( VERSION 2.8.10 )
04
05  project( MathLibs )
06  set( VERSION 1.0.0 )
07  set( DESCRIPTION "The internal math-libraries." )
08
09  # Sources for common functionality, used in both math-libraries.
10  set( BASIC_SOURCES
11      "src/BasicMath.cpp"
12      "src/HeavyMath.cpp"  # Takes loooooong to compile!
13      "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
14      "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
15  # An OBJECT-library, used to only compile common sources once!
16  add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )
17
18  # Required include search-paths and constexpr support.
19  include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
20  set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
21      STRING "C++ compile-flags" FORCE )
22
23  # Requires "Boost.Outcome" (which has some requirements, too).
24  include_directories( SYSTEM
25      "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
26  get_target_property( props Boost::outcome MY_COMPILE_FEATURES )
27  set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${props}" CACHE
28      STRING "C++ compile-flags" FORCE )
29
30  # A shared library for basic-math functionality.
31  add_library( basicmath SHARED
32      $<TARGET_OBJECTS:basicmath_ObjLib> )
33  # The usage-requirements of the OBJECT-library (and its
34  # direct dependency) are already set in directory scope
35  # (above in lines 19 to 28).
36
37  # A shared library for advanced-math functionality.
38  add_library( extmath SHARED
39      "src/ExtendedMath.cpp"  # Premium-content!
40      $<TARGET_OBJECTS:basicmath_ObjLib> )
41  # The usage-requirements of the OBJECT-library (and its
42  # direct dependency) are already set in directory scope
43  # (above in lines 19 to 28).
44
45  # Target 'extmath' requires include-path and needs
46  # to link to library-file of its extra dependency
47  # "Boost.Graph".
48  include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
49  target_link_libraries( extmath ${Boost_LIBRARIES} )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Creating libraries — an overview (cont.)

Traditional CMake way

```
01  # ./library/CMakeLists.txt -- Traditional CMake
02
03  cmake_minimum_required( VERSION 2.8.10 )
04
05  project( MathLibs )
06  set( VERSION 1.0.0 )
07  set( DESCRIPTION "The internal math-libraries." )
08
09  # Sources for common functionality, used in both math-libraries.
10  set( BASIC_SOURCES
11      "src/BasicMath.cpp"
12      "src/HeavyMath.cpp"  # Takes loooooong to compile!
13      "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
14      "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
15  # An OBJECT-library, used to only compile common sources once!
16  add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )
17
18  # Required include search-paths and constexpr support.
19  include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
20  set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
21      STRING "C++ compile-flags" FORCE )
22
23  # Requires "Boost.Outcome" (which has some requirements, too).
24  include_directories( SYSTEM
25      "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
26  get_target_property( props Boost::outcome MY_COMPILE_FEATURES )
27  set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${props}" CACHE
28      STRING "C++ compile-flags" FORCE )
29
30  # A shared library for basic-math functionality.
31  add_library( basicmath SHARED
32      $<TARGET_OBJECTS:basicmath_ObjLib> )
33  # The usage-requirements of the OBJECT-library (and its
34  # direct dependency) are already set in directory scope
35  # (above in lines 19 to 28).
36
37  # A shared library for advanced-math functionality.
38  add_library( extmath SHARED
39      "src/ExtendedMath.cpp"  # Premium-content!
40      $<TARGET_OBJECTS:basicmath_ObjLib> )
41  # The usage-requirements of the OBJECT-library (and its
42  # direct dependency) are already set in directory scope
43  # (above in lines 19 to 28).
44
45  # Target 'extmath' requires include-path and needs
46  # to link to library-file of its extra dependency
47  # "Boost.Graph".
48  include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
49  target_link_libraries( extmath ${Boost_LIBRARIES} )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Let's continue step by step...

CREATING TARGETS FROM `OBJECT` TARGETS

TRADITIONAL AND *MODERN CMAKE* WAY

- Object-files from the `OBJECT` target are the *sources* of the `SHARED` library target.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```cmake
# ./library/CMakeLists.txt -- Traditional CMake
...
# A shared library for basic-math functionality.
add_library( basicmath SHARED
    $<TARGET_OBJECTS:basicmath_ObjLib> )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

CREATING TARGETS FROM `OBJECT` TARGETS

TRADITIONAL AND *MODERN CMAKE* WAY

- Object-files from the `OBJECT` target are the *sources* of the `SHARED` library target.

```cmake
# ./library/CMakeLists.txt -- Traditional CMake
...
# A shared library for basic-math functionality.
add_library( basicmath SHARED
    $<TARGET_OBJECTS:basicmath_ObjLib> )
```

⇒

```cmake
# ./library/CMakeLists.txt -- Modern CMake
...
# A shared library for basic-math functionality.
add_library( basicmath SHARED "src/dummy.cpp" )
target_sources( basicmath
    PRIVATE $<TARGET_OBJECTS:basicmath_ObjLib> )
```

CREATING TARGETS FROM `OBJECT` TARGETS

TRADITIONAL AND *MODERN CMAKE* WAY

- Object-files from the `OBJECT` target are the *sources* of the `SHARED` library target.
- *Traditional CMake*:
  The *usage-requirements* of the `OBJECT` library were set on directory-scope
  - and are therefore already set on the `SHARED` library target.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```
28 # ./library/CMakeLists.txt -- Traditional CMake
29 ...
30 # A shared library for basic-math functionality.
31 add_library( basicmath SHARED
32     $<TARGET_OBJECTS:basicmath_ObjLib> )
33 # The usage-requirements of the OBJECT-library (and its
34 # direct dependency) are already set in directory scope
35 # (above in lines 19 to 28).
36 ...
```

```
29 # ./library/CMakeLists.txt -- Modern CMake
30 ...
31 # A shared library for basic-math functionality.
32 add_library( basicmath SHARED "src/dummy.cpp" )
33 target_sources( basicmath
34     PRIVATE $<TARGET_OBJECTS:basicmath_ObjLib> )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

CREATING TARGETS FROM `OBJECT` TARGETS

TRADITIONAL AND *MODERN CMAKE* WAY

- Object-files from the `OBJECT` target are the *sources* of the `SHARED` library target.
- *Traditional CMake*:
  The *usage-requirements* of the `OBJECT` library were set on directory-scope
  - and are therefore already set on the `SHARED` library target.
- *Modern CMake* needs to propagate *usage-requirements* explicitly.

```
# ./library/CMakeLists.txt -- Traditional CMake
...
# A shared library for basic-math functionality.
add_library( basicmath SHARED
    $<TARGET_OBJECTS:basicmath_ObjLib> )
# The usage-requirements of the OBJECT-library (and its
# direct dependency) are already set in directory scope
# (above in lines 19 to 28).
...
```

⇒

```
# ./library/CMakeLists.txt -- Modern CMake
...
# A shared library for basic-math functionality.
add_library( basicmath SHARED "src/dummy.cpp" )
target_sources( basicmath
    PRIVATE $<TARGET_OBJECTS:basicmath_ObjLib> )
# Inherit the usage-requirements from direct dependency 'basicmath_ObjLib'
# (and its indirect dependency "Boost.Outcome").
target_include_directories( basicmath
    PUBLIC
        $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( basicmath
    PUBLIC $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_COMPILE_FEATURES> )
...
```

CREATING TARGETS FROM `OBJECT` TARGETS

TRADITIONAL AND *MODERN CMAKE* WAY

- Object-files from the `OBJECT` target are the *sources* of the `SHARED` library target.
- *Traditional CMake*:
  The *usage-requirements* of the `OBJECT` library were set on directory-scope
  - and are therefore already set on the `SHARED` library target.
- *Modern CMake* needs to propagate *usage-requirements* explicitly.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```
28 # ./library/CMakeLists.txt -- Traditional CMake
29 ...
30 # A shared library for basic-math functionality.
31 add_library( basicmath SHARED
32     $<TARGET_OBJECTS:basicmath_ObjLib> )
33 # The usage-requirements of the OBJECT-library (and its
34 # direct dependency) are already set in directory scope
35 # (above in lines 19 to 28).
36 ...
```

⇒

```
29 # ./library/CMakeLists.txt -- Modern CMake
30 ...
31 # A shared library for basic-math functionality.
32 add_library( basicmath SHARED "src/dummy.cpp" )
33 target_sources( basicmath
34     PRIVATE $<TARGET_OBJECTS:basicmath_ObjLib> )
35 # Inherit the usage-requirements from direct dependency 'basicmath_ObjLib'
36 # (and its indirect dependency "Boost.Outcome").
37 target_include_directories( basicmath
38     PUBLIC
39         $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_INCLUDE_DIRECTORIES> )
40 target_compile_features( basicmath
41     PUBLIC $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_COMPILE_FEATURES> )
42 ...
```

HOWEVER (AS YOU ALREADY KNOW)...

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Creating targets from `OBJECT` targets

More Modern CMake way

- Since **CMake 3.12** the `target_link_libraries` command can be used with `OBJECT` targets (as you have seen already).
  - *Object-files* files are propagated to direct dependants (only).
  - *Usage-requirements* are propagated as for all other targets.

```cmake
# ./library/CMakeLists.txt -- Modern CMake
...
# A shared library for basic-math functionality.
add_library( basicmath SHARED "src/dummy.cpp" )
target_sources( basicmath
    PRIVATE $<TARGET_OBJECTS:basicmath_ObjLib> )
# Inherit the usage-requirements from direct dependency 'basicmath_ObjLib'
# (and its indirect dependency "Boost.Outcome").
target_include_directories( basicmath
    PUBLIC
        $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( basicmath
    PUBLIC $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_COMPILE_FEATURES> )
...
```

⇒

```cmake
# ./library/CMakeLists.txt -- More Modern CMake
...
# A shared library for basic-math functionality.
add_library( basicmath SHARED )
target_link_libraries( basicmath PUBLIC basicmath_ObjLib )
```

CREATING TARGETS FROM `OBJECT` TARGETS

TRADITIONAL AND *MODERN CMAKE* WAY

- The same applies to the second library.
  - Additionally, a dependency to *Boost.Graph* is needed.

```
34  # ./library/CMakeLists.txt -- Traditional CMake
35  ...
36  # A shared library for advanced-math functionality.
37  add_library( extmath SHARED
38      "src/ExtendedMath.cpp"  # Premium-content!
39      $<TARGET_OBJECTS:basicmath_ObjLib> )
40  # The usage-requirements of the OBJECT-library (and its
41  # direct dependency) are already set in directory scope
42  # (above in lines 19 to 28).
43
44  # Target 'extmath' requires include-path and needs
45  # to link to library-file of its extra dependency
46  # "Boost.Graph".
47  include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
48  target_link_libraries( extmath ${Boost_LIBRARIES} )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

CREATING TARGETS FROM `OBJECT` TARGETS

TRADITIONAL AND *MODERN CMAKE* WAY

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- The same applies to the second library.
  - Additionally, a dependency to *Boost.Graph* is needed.

```cmake
# ./library/CMakeLists.txt -- Traditional CMake
...
# A shared library for advanced-math functionality.
add_library( extmath SHARED
    "src/ExtendedMath.cpp"  # Premium-content!
    $<TARGET_OBJECTS:basicmath_ObjLib> )
# The usage-requirements of the OBJECT-library (and its
# direct dependency) are already set in directory scope
# (above in lines 19 to 28).

# Target 'extmath' requires include-path and needs
# to link to library-file of its extra dependency
# "Boost.Graph".
include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
target_link_libraries( extmath ${Boost_LIBRARIES} )
```

⇒

```cmake
# ./library/CMakeLists.txt -- Modern CMake
...
# A shared library for advanced-math functionality.
add_library( extmath SHARED "src/dummy.cpp" )
target_sources( extmath
    PRIVATE "src/ExtendedMath.cpp"  # Premium-content!
            $<TARGET_OBJECTS:basicmath_ObjLib> )
# Inherit the usage-requirements from direct dependency 'basicmath_ObjLib'
# (and its indirect dependency "Boost.Outcome").
target_include_directories( extmath
    PUBLIC
        $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( extmath
    PUBLIC $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_COMPILE_FEATURES> )
# "Boost.Graph" is an additional dependency.
target_link_libraries( extmath PRIVATE Boost::graph )

# Create ALIAS targets.
add_library( MyCalc::basicmath ALIAS basicmath )
add_library( MyCalc::extmath ALIAS extmath )
```

Creating targets from `OBJECT` targets

Traditional and *Modern CMake* way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

- The same applies to the second library.
  - Additionally, a dependency to *Boost.Graph* is needed.
  - Aliases might come in handy, too.

```cmake
# ./library/CMakeLists.txt -- Traditional CMake
...
# A shared library for advanced-math functionality.
add_library( extmath SHARED
    "src/ExtendedMath.cpp"  # Premium-content!
    $<TARGET_OBJECTS:basicmath_ObjLib> )
# The usage-requirements of the OBJECT-library (and its
# direct dependency) are already set in directory scope
# (above in lines 19 to 28).

# Target 'extmath' requires include-path and needs
# to link to library-file of its extra dependency
# "Boost.Graph".
include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
target_link_libraries( extmath ${Boost_LIBRARIES} )
```

⇒

```cmake
# ./library/CMakeLists.txt -- Modern CMake
...
# A shared library for advanced-math functionality.
add_library( extmath SHARED "src/dummy.cpp" )
target_sources( extmath
    PRIVATE "src/ExtendedMath.cpp"  # Premium-content!
            $<TARGET_OBJECTS:basicmath_ObjLib> )
# Inherit the usage-requirements from direct dependency 'basicmath_ObjLib'
# (and its indirect dependency "Boost.Outcome").
target_include_directories( extmath
    PUBLIC
        $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( extmath
    PUBLIC $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_COMPILE_FEATURES> )
# "Boost.Graph" is an additional dependency.
target_link_libraries( extmath PRIVATE Boost::graph )

# Create ALIAS targets.
add_library( MyCalc::basicmath ALIAS basicmath )
add_library( MyCalc::extmath ALIAS extmath )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Creating targets from `OBJECT` targets

More Modern CMake way

```cmake
# ./library/CMakeLists.txt -- Modern CMake
...
# A shared library for advanced-math functionality.
add_library( extmath SHARED "src/dummy.cpp" )
target_sources( extmath
    PRIVATE "src/ExtendedMath.cpp"  # Premium-content!
            $<TARGET_OBJECTS:basicmath_ObjLib> )
# Inherit the usage-requirements from direct dependency 'basicmath_ObjLib'
# (and its indirect dependency "Boost.Outcome").
target_include_directories( extmath
    PUBLIC
        $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( extmath
    PUBLIC $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_COMPILE_FEATURES> )
# "Boost.Graph" is an additional dependency.
target_link_libraries( extmath PRIVATE Boost::graph )

# Create ALIAS targets.
add_library( MyCalc::basicmath ALIAS basicmath )
add_library( MyCalc::extmath ALIAS extmath )
```

⇒

- With CMake 3.12:

```cmake
# ./library/CMakeLists.txt -- More Modern CMake
...
# A shared library for advanced-math functionality.
add_library( extmath SHARED )
target_sources( extmath
    PRIVATE "src/ExtendedMath.cpp" )  # Premium-content!
target_link_libraries( extmath PUBLIC basicmath_ObjLib )
# "Boost.Graph" is an additional dependency.
target_link_libraries( extmath PRIVATE Boost::graph )

# Create ALIAS targets.
add_library( MyCalc::basicmath ALIAS basicmath )
add_library( MyCalc::extmath ALIAS extmath )
```

CREATING LIBRARIES — AN OVERVIEW (CONT.)

TRADITIONAL CMAKE WAY

```
01  # ./library/CMakeLists.txt -- Traditional CMake
02
03  cmake_minimum_required( VERSION 2.8.10 )
04
05  project( MathLibs )
06  set( VERSION 1.0.0 )
07  set( DESCRIPTION "The internal math-libraries." )
08
09  # Sources for common functionality, used in both math-libraries.
10  set( BASIC_SOURCES
11      "src/BasicMath.cpp"
12      "src/HeavyMath.cpp"  # Takes loooooong to compile!
13      "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
14      "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
15  # An OBJECT-library, used to only compile common sources once!
16  add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )
17
18  # Required include search-paths and constexpr support.
19  include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
20  set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
21      STRING "C++ compile-flags" FORCE )
22
23  # Requires "Boost.Outcome" (which has some requirements, too).
24  include_directories( SYSTEM
25      "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
26  get_target_property( props Boost::outcome MY_COMPILE_FEATURES )
27  set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${props}" CACHE
28      STRING "C++ compile-flags" FORCE )
29
30  # A shared library for basic-math functionality.
31  add_library( basicmath SHARED
32      $<TARGET_OBJECTS:basicmath_ObjLib> )
33  # The usage-requirements of the OBJECT-library (and its
34  # direct dependency) are already set in directory scope
35  # (above in lines 19 to 28).
36
37  # A shared library for advanced-math functionality.
38  add_library( extmath SHARED
39      "src/ExtendedMath.cpp"  # Premium-content!
40      $<TARGET_OBJECTS:basicmath_ObjLib> )
41  # The usage-requirements of the OBJECT-library (and its
42  # direct dependency) are already set in directory scope
43  # (above in lines 19 to 28).
44
45  # Target 'extmath' requires include-path and needs
46  # to link to library-file of its extra dependency
47  # "Boost.Graph".
48  include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
49  target_link_libraries( extmath ${Boost_LIBRARIES} )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Creating libraries — an overview (cont.)

Traditional CMake way

```
01 # ./library/CMakeLists.txt -- Traditional CMake
02
03 cmake_minimum_required( VERSION 2.8.10 )
04
05 project( MathLibs )
06 set( VERSION 1.0.0 )
07 set( DESCRIPTION "The internal math-libraries." )
08
09 # Sources for common functionality, used in both math-libraries.
10 set( BASIC_SOURCES
11     "src/BasicMath.cpp"
12     "src/HeavyMath.cpp"  # Takes loooooooong to compile!
13     "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
14     "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )
15 # An OBJECT-library, used to only compile common sources once!
16 add_library( basicmath_ObjLib OBJECT ${BASIC_SOURCES} )
17
18 # Required include search-paths and constexpr support.
19 include_directories( "${CMAKE_CURRENT_SOURCE_DIR}/include" )
20 set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11" CACHE
21     STRING "C++ compile-flags" FORCE )
22
23 # Requires "Boost.Outcome" (which has some requirements, too).
24 include_directories( SYSTEM
25     "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
26 get_target_property( props Boost::outcome MY_COMPILE_FEATURES )
27 set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${props}" CACHE
28     STRING "C++ compile-flags" FORCE )
29
30 # A shared library for basic-math functionality.
31 add_library( basicmath SHARED
32     $<TARGET_OBJECTS:basicmath_ObjLib> )
33 # The usage-requirements of the OBJECT-library (and its
34 # direct dependency) are already set in directory scope
35 # (above in lines 19 to 28).
36
37 # A shared library for advanced-math functionality.
38 add_library( extmath SHARED
39     "src/ExtendedMath.cpp"  # Premium-content!
40     $<TARGET_OBJECTS:basicmath_ObjLib> )
41 # The usage-requirements of the OBJECT-library (and its
42 # direct dependency) are already set in directory scope
43 # (above in lines 19 to 28).
44
45 # Target 'extmath' requires include-path and needs
46 # to link to library-file of its extra dependency
47 # "Boost.Graph".
48 include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
49 target_link_libraries( extmath ${Boost_LIBRARIES} )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

⇒

Creating libraries — an overview (cont.)

Modern CMake way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
          oost/
          └── CMakeLists.txt
          oost_outcome/
          └── CMakeLists.txt
          ry/
          CMakeLists.txt
          nclude/
          ├── MathAPI.h
          └── Math.h
          src/
          ├── BasicMath.cpp
          ├── ExtendedMath.cpp
          └── HeavyMath.cpp
```

```
# ./library/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )

# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )

# Requires "Boost.Outcome" (which has some requirements, too).
target_include_directories( basicmath_ObjLib SYSTEM
    PUBLIC
        $<TARGET_PROPERTY:Boost::outcome,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( basicmath_ObjLib
    PUBLIC $<TARGET_PROPERTY:Boost::outcome,INTERFACE_COMPILE_FEATURES> )

# A shared library for basic-math functionality.
add_library( basicmath SHARED "src/dummy.cpp" )
target_sources( basicmath
    PRIVATE $<TARGET_OBJECTS:basicmath_ObjLib> )
# Inherit the usage-requirements from direct dependency 'basicmath_ObjLib'
# (and its indirect dependency "Boost.Outcome").
target_include_directories( basicmath
    PUBLIC
        $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( basicmath
    PUBLIC $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_COMPILE_FEATURES> )

# A shared library for advanced-math functionality.
add_library( extmath SHARED "src/dummy.cpp" )
target_sources( extmath
    PRIVATE "src/ExtendedMath.cpp"  # Premium-content!
            $<TARGET_OBJECTS:basicmath_ObjLib> )
# Inherit the usage-requirements from direct dependency 'basicmath_ObjLib'
# (and its indirect dependency "Boost.Outcome").
target_include_directories( extmath
    PUBLIC
        $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( extmath
    PUBLIC $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_COMPILE_FEATURES> )
# "Boost.Graph" is an additional dependency.
target_link_libraries( extmath PRIVATE Boost::graph )

# Create ALIAS targets.
add_library( MyCalc::basicmath ALIAS basicmath )
add_library( MyCalc::extmath ALIAS extmath )
```

Creating libraries — an overview (cont.)

Modern CMake way

```
# ./library/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT "src/dummy.cpp" )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )

# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )

# Requires "Boost.Outcome" (which has some requirements, too).
target_include_directories( basicmath_ObjLib SYSTEM
    PUBLIC
        $<TARGET_PROPERTY:Boost::outcome,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( basicmath_ObjLib
    PUBLIC $<TARGET_PROPERTY:Boost::outcome,INTERFACE_COMPILE_FEATURES> )

# A shared library for basic-math functionality.
add_library( basicmath SHARED "src/dummy.cpp" )
target_sources( basicmath
    PRIVATE $<TARGET_OBJECTS:basicmath_ObjLib> )
# Inherit the usage-requirements from direct dependency 'basicmath_ObjLib'
# (and its indirect dependency "Boost.Outcome").
target_include_directories( basicmath
    PUBLIC
        $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( basicmath
    PUBLIC $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_COMPILE_FEATURES> )

# A shared library for advanced-math functionality.
add_library( extmath SHARED "src/dummy.cpp" )
target_sources( extmath
    PRIVATE "src/ExtendedMath.cpp"  # Premium-content!
            $<TARGET_OBJECTS:basicmath_ObjLib> )
# Inherit the usage-requirements from direct dependency 'basicmath_ObjLib'
# (and its indirect dependency "Boost.Outcome").
target_include_directories( extmath
    PUBLIC
        $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_INCLUDE_DIRECTORIES> )
target_compile_features( extmath
    PUBLIC $<TARGET_PROPERTY:basicmath_ObjLib,INTERFACE_COMPILE_FEATURES> )
# "Boost.Graph" is an additional dependency.
target_link_libraries( extmath PRIVATE Boost::graph )

# Create ALIAS targets.
add_library( MyCalc::basicmath ALIAS basicmath )
add_library( MyCalc::extmath ALIAS extmath )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
        boost/
        └── CMakeLists.txt
        oost_outcome/
        └── CMakeLists.txt
        ry/
        CMakeLists.txt
        nclude/
        ├── MathAPI.h
        └── Math.h
        src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

⇒

Creating libraries — an overview (cont.)

More Modern CMake way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```cmake
# ./library/CMakeLists.txt -- More Modern CMake

cmake_minimum_required( VERSION 3.12 )

project( MathLibs
         VERSION 1.0.0
         DESCRIPTION "The internal math-libraries." )

# An OBJECT-library, used to only compile common sources once
# which are used in both math-libraries.
add_library( basicmath_ObjLib OBJECT )
target_sources( basicmath_ObjLib
    PRIVATE   "src/BasicMath.cpp"
              "src/HeavyMath.cpp"  # Takes loooooong to compile!
    PUBLIC    "${CMAKE_CURRENT_SOURCE_DIR}/include/Math.h"
    INTERFACE "${CMAKE_CURRENT_SOURCE_DIR}/include/MathAPI.h" )

# Required include search-paths and constexpr support.
target_include_directories( basicmath_ObjLib
    PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include" )
target_compile_features( basicmath_ObjLib
    PUBLIC cxx_constexpr )

# Requires "Boost.Outcome" (which has some requirements, too).
target_link_libraries( basicmath_ObjLib PUBLIC Boost::outcome )

# A shared library for basic-math functionality.
add_library( basicmath SHARED )
target_link_libraries( basicmath PUBLIC basicmath_ObjLib )

# A shared library for advanced-math functionality.
add_library( extmath SHARED )
target_sources( extmath
    PRIVATE "src/ExtendedMath.cpp" )  # Premium-content!
target_link_libraries( extmath PUBLIC basicmath_ObjLib )
# "Boost.Graph" is an additional dependency.
target_link_libraries( extmath PRIVATE Boost::graph )

# Create ALIAS targets.
add_library( MyCalc::basicmath ALIAS basicmath )
add_library( MyCalc::extmath ALIAS extmath )
```

Linking all together into executables

FreeCalculator / PremiumCalculator

Linking all together into executables

Traditional and *Modern CMake* way

```cmake
# ./app/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( CalculatorApp )
set( VERSION 1.0 )
set( DESCRIPTION "My fancy calculator app." )

# Freely available calculator app.
add_executable( FreeCalculator "src/main.cpp" )
target_link_libraries( FreeCalculator basicmath )

# Possibly, compile key-file into premium-calculator, too.
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/src/key-file.cpp")
    set( EXTRA_SOURCES "src/key-file.cpp" )
endif ()

# Premium-calculator app (with advanced functionality)
add_executable( PremiumCalculator "src/main.cpp"
                                  ${EXTRA_SOURCES} )
target_link_libraries( PremiumCalculator extmath )

# Both targets require "Boost.Program_Options".
include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
target_link_libraries( FreeCalculator ${Boost_LIBRARIES} )
target_link_libraries( PremiumCalculator
                                      ${Boost_LIBRARIES} )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Linking all together into executables

Traditional and *Modern CMake* way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```cmake
# ./app/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( CalculatorApp )
set( VERSION 1.0 )
set( DESCRIPTION "My fancy calculator app." )

# Freely available calculator app.
add_executable( FreeCalculator "src/main.cpp" )
target_link_libraries( FreeCalculator basicmath )

# Possibly, compile key-file into premium-calculator, too.
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/src/key-file.cpp")
    set( EXTRA_SOURCES "src/key-file.cpp" )
endif ()

# Premium-calculator app (with advanced functionality)
add_executable( PremiumCalculator "src/main.cpp"
                                  ${EXTRA_SOURCES} )
target_link_libraries( PremiumCalculator extmath )

# Both targets require "Boost.Program_Options".
include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
target_link_libraries( FreeCalculator ${Boost_LIBRARIES} )
target_link_libraries( PremiumCalculator
                                      ${Boost_LIBRARIES} )
```

⇒

```cmake
# ./app/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( CalculatorApp
         VERSION 1.0
         DESCRIPTION "My fancy calculator app." )

# Freely available calculator app.
add_executable( FreeCalculator "src/dummy.cpp" )
target_sources( FreeCalculator PRIVATE "src/main.cpp" )
target_link_libraries( FreeCalculator
    PRIVATE MyCalc::basicmath
            Boost::program_options )

# Premium-calculator app (with advanced functionality)
add_executable( PremiumCalculator "src/dummy.cpp" )
target_sources( PremiumCalculator PRIVATE "src/main.cpp" )
target_link_libraries( PremiumCalculator
    PRIVATE MyCalc::extmath
            Boost::program_options )

# Possibly, compile key-file into premium-calculator, too.
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/src/key-file.cpp")
    target_sources( PremiumCalculator
        PRIVATE "src/key-file.cpp" )
endif ()
```

Linking all together into executables

Traditional and *Modern CMake* way

```cmake
# ./app/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( CalculatorApp )
set( VERSION 1.0 )
set( DESCRIPTION "My fancy calculator app." )

# Freely available calculator app.
add_executable( FreeCalculator "src/main.cpp" )
target_link_libraries( FreeCalculator basicmath )

# Possibly, compile key-file into premium-calculator, too.
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/src/key-file.cpp")
    set( EXTRA_SOURCES "src/key-file.cpp" )
endif ()

# Premium-calculator app (with advanced functionality)
add_executable( PremiumCalculator "src/main.cpp"
                                  ${EXTRA_SOURCES} )
target_link_libraries( PremiumCalculator extmath )

# Both targets require "Boost.Program_Options".
include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
target_link_libraries( FreeCalculator ${Boost_LIBRARIES} )
target_link_libraries( PremiumCalculator
                                      ${Boost_LIBRARIES} )
```

⇒

```cmake
# ./app/CMakeLists.txt -- Modern CMake

cmake_minimum_required( VERSION 3.10 )

project( CalculatorApp
         VERSION 1.0
         DESCRIPTION "My fancy calculator app." )

# Freely available calculator app.
add_executable( FreeCalculator "src/dummy.cpp" )
target_sources( FreeCalculator PRIVATE "src/main.cpp" )
target_link_libraries( FreeCalculator
    PRIVATE MyCalc::basicmath
            Boost::program_options )

# Premium-calculator app (with advanced functionality)
add_executable( PremiumCalculator "src/dummy.cpp" )
target_sources( PremiumCalculator PRIVATE "src/main.cpp" )
target_link_libraries( PremiumCalculator
    PRIVATE MyCalc::extmath
            Boost::program_options )

# Possibly, compile key-file into premium-calculator, too.
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/src/key-file.cpp")
    target_sources( PremiumCalculator
        PRIVATE "src/key-file.cpp" )
endif ()
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Linking all together into executables

Traditional and *More Modern CMake* way

```
# ./app/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( CalculatorApp )
set( VERSION 1.0 )
set( DESCRIPTION "My fancy calculator app." )

# Freely available calculator app.
add_executable( FreeCalculator "src/main.cpp" )
target_link_libraries( FreeCalculator basicmath )

# Possibly, compile key-file into premium-calculator, too.
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/src/key-file.cpp")
    set( EXTRA_SOURCES "src/key-file.cpp" )
endif ()

# Premium-calculator app (with advanced functionality)
add_executable( PremiumCalculator "src/main.cpp"
                                  ${EXTRA_SOURCES} )
target_link_libraries( PremiumCalculator extmath )

# Both targets require "Boost.Program_Options".
include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
target_link_libraries( FreeCalculator ${Boost_LIBRARIES} )
target_link_libraries( PremiumCalculator
                                      ${Boost_LIBRARIES} )
```

⇒

```
# ./app/CMakeLists.txt -- More Modern CMake

cmake_minimum_required( VERSION 3.11 )

project( CalculatorApp
         VERSION 1.0
         DESCRIPTION "My fancy calculator app." )

# Freely available calculator app.
add_executable( FreeCalculator )
target_sources( FreeCalculator PRIVATE "src/main.cpp" )
target_link_libraries( FreeCalculator
    PRIVATE MyCalc::basicmath
            Boost::program_options )

# Premium-calculator app (with advanced functionality)
add_executable( PremiumCalculator )
target_sources( PremiumCalculator PRIVATE "src/main.cpp" )
target_link_libraries( PremiumCalculator
    PRIVATE MyCalc::extmath
            Boost::program_options )

# Possibly, compile key-file into premium-calculator, too.
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/src/key-file.cpp")
    target_sources( PremiumCalculator
        PRIVATE "src/key-file.cpp" )
endif ()
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

CMaking and compiling everything

Modern and *More Modern CMake* way

CMaking

Compiling

```
#$  cd /tmp/cmake/more_modern/build
#$  cmake -G "Unix Makefiles"../source/
-- The C compiler identification is GNU 6.5.0
-- The CXX compiler identification is GNU 6.5.0
-- Check for working C compiler: /usr/bin/gcc-6
-- Check for working C compiler: /usr/bin/gcc-6 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/g++-6
-- Check for working CXX compiler: /usr/bin/g++-6 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Boost version: 1.58.0
-- Found the following Boost libraries:
--   program_options
--   graph
-- Configuring done
-- Generating done
-- Build files have been written to: /tmp/cmake/more_modern/build
```

CMaking and compiling everything

Modern and *More Modern CMake* way

CMaking

Success!

Compiling

```
#$  cd /tmp/cmake/more_modern/build
#$  cmake -G "Unix Makefiles"../source/
-- The C compiler identification is GNU 6.5.0
-- The CXX compiler identification is GNU 6.5.0
-- Check for working C compiler: /usr/bin/gcc-6
-- Check for working C compiler: /usr/bin/gcc-6 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/g++-6
-- Check for working CXX compiler: /usr/bin/g++-6 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Boost version: 1.58.0
-- Found the following Boost libraries:
--   program_options
--   graph
-- Configuring done
-- Generating done
-- Build files have been written to: /tmp/cmake/more_modern/build
```

```
#$  cd /tmp/cmake/more_modern/build
#$  cmake --build .
Scanning dependencies of target basicmath_ObjLib
[ 10%] Building CXX object library/CMakeFiles/basicmath_ObjLib.dir/
                                                    src/BasicMath.cpp.o
[ 20%] Building CXX object library/CMakeFiles/basicmath_ObjLib.dir/
                                                    src/HeavyMath.cpp.o
[ 20%] Built target basicmath_ObjLib
Scanning dependencies of target extmath
[ 30%] Building CXX object library/CMakeFiles/extmath.dir/src/ExtendedMath.cpp.o
[ 40%] Linking CXX shared library libextmath.so
[ 40%] Built target extmath
Scanning dependencies of target basicmath
[ 50%] Linking CXX shared library libbasicmath.so
[ 50%] Built target basicmath
Scanning dependencies of target PremiumCalculator
[ 60%] Building CXX object app/CMakeFiles/PremiumCalculator.dir/src/main.cpp.o
[ 70%] Building CXX object app/CMakeFiles/PremiumCalculator.dir/src/key-file.cpp.o
[ 80%] Linking CXX shared library PremiumCalculator
[ 80%] Built target PremiumCalculator
Scanning dependencies of target FreeCalculator
[ 90%] Building CXX object app/CMakeFiles/FreeCalculator.dir/src/main.cpp.o
[100%] Linking CXX shared library FreeCalculator
[100%] Built target FreeCalculator
```

CMaking and compiling everything

Traditional CMake way

CMaking

Compiling

```
#$  cd /tmp/cmake/traditional/build
#$  cmake -G "Unix Makefiles"../source/
-- The C compiler identification is GNU 6.5.0
-- The CXX compiler identification is GNU 6.5.0
-- Check for working C compiler: /usr/bin/gcc-6
-- Check for working C compiler: /usr/bin/gcc-6 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/g++-6
-- Check for working CXX compiler: /usr/bin/g++-6 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Boost version: 1.58.0
-- Found the following Boost libraries:
--   program_options
--   graph
-- Configuring done
-- Generating done
-- Build files have been written to: /tmp/cmake/traditional/build
```

CMaking and compiling everything

Traditional CMake way

CMaking

Failure!

Compiling

```
#$  cd /tmp/cmake/traditional/build
#$  cmake -G "Unix Makefiles"../source/
-- The C compiler identification is GNU 6.5.0
-- The CXX compiler identification is GNU 6.5.0
-- Check for working C compiler: /usr/bin/gcc-6
-- Check for working C compiler: /usr/bin/gcc-6 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/g++-6
-- Check for working CXX compiler: /usr/bin/g++-6 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Boost version: 1.58.0
-- Found the following Boost libraries:
--   program_options
--   graph
-- Configuring done
-- Generating done
-- Build files have been written to: /tmp/cmake/traditional/build
```

```
#$  cd /tmp/cmake/traditional/build
#$  cmake --build .
Scanning dependencies of target basicmath_ObjLib
[ 10%] Building CXX object library/CMakeFiles/basicmath_ObjLib.dir/
                                                               src/BasicMath.cpp.o
[ 20%] Building CXX object library/CMakeFiles/basicmath_ObjLib.dir/
                                                               src/HeavyMath.cpp.o
[ 20%] Built target basicmath_ObjLib
Scanning dependencies of target extmath
[ 30%] Building CXX object library/CMakeFiles/extmath.dir/src/ExtendedMath.cpp.o
[ 40%] Linking CXX shared library libextmath.so
[ 40%] Built target extmath
Scanning dependencies of target basicmath
[ 50%] Linking CXX shared library libbasicmath.so
[ 50%] Built target basicmath
Scanning dependencies of target PremiumCalculator
[ 60%] Building CXX object app/CMakeFiles/PremiumCalculator.dir/src/main.cpp.o
/tmp/cmake/traditional/source/app/src/main.cpp:4:18: fatal error:
                                                Math.h: No such file or directory
 #include "Math.h"
                  ^
compilation terminated.
...
```

Linking all together into executables

Traditional CMake way

```
# ./app/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( CalculatorApp )
set( VERSION 1.0 )
set( DESCRIPTION "My fancy calculator app." )

# Freely available calculator app.
add_executable( FreeCalculator "src/main.cpp" )
target_link_libraries( FreeCalculator basicmath )

# Possibly, compile key-file into premium-calculator, too.
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/src/key-file.cpp")
    set( EXTRA_SOURCES "src/key-file.cpp" )
endif ()

# Premium-calculator app (with advanced functionality)
add_executable( PremiumCalculator "src/main.cpp"
                                  ${EXTRA_SOURCES} )
target_link_libraries( PremiumCalculator extmath )

# Both targets require "Boost.Program_Options".
include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
target_link_libraries( FreeCalculator ${Boost_LIBRARIES} )
target_link_libraries( PremiumCalculator
                                      ${Boost_LIBRARIES} )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Linking all together into executables

Fixed *Traditional CMake* way

```cmake
# ./app/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( CalculatorApp )
set( VERSION 1.0 )
set( DESCRIPTION "My fancy calculator app." )

# Freely available calculator app.
add_executable( FreeCalculator "src/main.cpp" )
target_link_libraries( FreeCalculator basicmath )

# Possibly, compile key-file into premium-calculator, too.
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/src/key-file.cpp")
    set( EXTRA_SOURCES "src/key-file.cpp" )
endif ()

# Premium-calculator app (with advanced functionality)
add_executable( PremiumCalculator "src/main.cpp"
                                  ${EXTRA_SOURCES} )
target_link_libraries( PremiumCalculator extmath )

# Both targets require "Boost.Program_Options".
include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
target_link_libraries( FreeCalculator ${Boost_LIBRARIES} )
target_link_libraries( PremiumCalculator
                                      ${Boost_LIBRARIES} )
```

```cmake
# Both targets require the include-path to their direct
# dependency 'basicmath'/'extmath' and indirect dependency
# "Boost.Outcome".
include_directories( "../library/include" )  # This is ugly!
include_directories( SYSTEM
    "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
```

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Linking all together into executables

Fixed *Traditional CMake* way

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

```cmake
# ./app/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( CalculatorApp )
set( VERSION 1.0 )
set( DESCRIPTION "My fancy calculator app." )

# Freely available calculator app.
add_executable( FreeCalculator "src/main.cpp" )
target_link_libraries( FreeCalculator basicmath )

# Possibly, compile key-file into premium-calculator, too.
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/src/key-file.cpp")
    set( EXTRA_SOURCES "src/key-file.cpp" )
endif ()

# Premium-calculator app (with advanced functionality)
add_executable( PremiumCalculator "src/main.cpp"
                                  ${EXTRA_SOURCES} )
target_link_libraries( PremiumCalculator extmath )

# Both targets require "Boost.Program_Options".
include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
target_link_libraries( FreeCalculator ${Boost_LIBRARIES} )
target_link_libraries( PremiumCalculator
                                      ${Boost_LIBRARIES} )
# Both targets require the include-path to their direct
# dependency 'basicmath'/'extmath' and indirect dependency
# "Boost.Outcome".
include_directories( "../library/include" )  # This is ugly!
include_directories( SYSTEM
    "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
```

- *Modern CMake* does not need this!
  - Dependencies propagate their include-seach-paths automatically, together with all other *usage-requirements*.

Linking all together into executables

Fixed *Traditional CMake* way

```cmake
# ./app/CMakeLists.txt -- Traditional CMake

cmake_minimum_required( VERSION 2.8.10 )

project( CalculatorApp )
set( VERSION 1.0 )
set( DESCRIPTION "My fancy calculator app." )

# Freely available calculator app.
add_executable( FreeCalculator "src/main.cpp" )
target_link_libraries( FreeCalculator basicmath )

# Possibly, compile key-file into premium-calculator, too.
if (EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/src/key-file.cpp")
    set( EXTRA_SOURCES "src/key-file.cpp" )
endif ()

# Premium-calculator app (with advanced functionality)
add_executable( PremiumCalculator "src/main.cpp"
                                  ${EXTRA_SOURCES} )
target_link_libraries( PremiumCalculator extmath )

# Both targets require "Boost.Program_Options".
include_directories( SYSTEM "${Boost_INCLUDE_DIRS}" )
target_link_libraries( FreeCalculator ${Boost_LIBRARIES} )
target_link_libraries( PremiumCalculator
                                      ${Boost_LIBRARIES} )
```

```cmake
# Both targets require the include-path to their direct
# dependency 'basicmath'/'extmath' and indirect dependency
# "Boost.Outcome".
include_directories( "../library/include" )  # This is ugly!
include_directories( SYSTEM
    "$<TARGET_PROPERTY:Boost::outcome,MY_INCLUDE_DIRS>" )
```

- *Modern CMake* does not need this!
  - Dependencies propagate their include-seach-paths automatically, together with all other *usage-requirements*.

```
.
├── CMakeLists.txt
├── app/
│   ├── CMakeLists.txt
│   └── src/
│       ├── key-file.cpp
│       └── main.cpp
├── external/
│   ├── CMakeLists.txt
│   ├── boost/
│   │   └── CMakeLists.txt
│   └── boost_outcome/
│       └── CMakeLists.txt
└── library/
    ├── CMakeLists.txt
    ├── include/
    │   ├── MathAPI.h
    │   └── Math.h
    └── src/
        ├── BasicMath.cpp
        ├── ExtendedMath.cpp
        └── HeavyMath.cpp
```

Let's check...

Cmaking and compiling everything

Traditional CMake way

CMaking

```
#$  cd /tmp/cmake/more_modern/build
#$  cmake -G "Unix Makefiles"../source/
-- The C compiler identification is GNU 6.5.0
-- The CXX compiler identification is GNU 6.5.0
-- Check for working C compiler: /usr/bin/gcc-6
-- Check for working C compiler: /usr/bin/gcc-6 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/g++-6
-- Check for working CXX compiler: /usr/bin/g++-6 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Boost version: 1.58.0
-- Found the following Boost libraries:
--   program_options
--   graph
-- Configuring done
-- Generating done
-- Build files have been written to: /tmp/cmake/more_modern/build
```

Compiling

CMaking and compiling everything

Traditional CMake way

CMaking

Finally!

Compiling

```
#$  cd /tmp/cmake/more_modern/build
#$  cmake -G "Unix Makefiles"../source/
-- The C compiler identification is GNU 6.5.0
-- The CXX compiler identification is GNU 6.5.0
-- Check for working C compiler: /usr/bin/gcc-6
-- Check for working C compiler: /usr/bin/gcc-6 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/g++-6
-- Check for working CXX compiler: /usr/bin/g++-6 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Boost version: 1.58.0
-- Found the following Boost libraries:
--   program_options
--   graph
-- Configuring done
-- Generating done
-- Build files have been written to: /tmp/cmake/more_modern/build
```

```
#$  cd /tmp/cmake/more_modern/build
#$  cmake --build .
Scanning dependencies of target basicmath_ObjLib
[ 10%] Building CXX object library/CMakeFiles/basicmath_ObjLib.dir/
                                                       src/BasicMath.cpp.o
[ 20%] Building CXX object library/CMakeFiles/basicmath_ObjLib.dir/
                                                       src/HeavyMath.cpp.o
[ 20%] Built target basicmath_ObjLib
Scanning dependencies of target extmath
[ 30%] Building CXX object library/CMakeFiles/extmath.dir/src/ExtendedMath.cpp.o
[ 40%] Linking CXX shared library libextmath.so
[ 40%] Built target extmath
Scanning dependencies of target basicmath
[ 50%] Linking CXX shared library libbasicmath.so
[ 50%] Built target basicmath
Scanning dependencies of target PremiumCalculator
[ 60%] Building CXX object app/CMakeFiles/PremiumCalculator.dir/src/main.cpp.o
[ 70%] Building CXX object app/CMakeFiles/PremiumCalculator.dir/src/key-file.cpp.o
[ 80%] Linking CXX shared library PremiumCalculator
[ 80%] Built target PremiumCalculator
Scanning dependencies of target FreeCalculator
[ 90%] Building CXX object app/CMakeFiles/FreeCalculator.dir/src/main.cpp.o
[100%] Linking CXX shared library FreeCalculator
[100%] Built target FreeCalculator
```

`TARGET_LINK_LIBRARIES` LIBRARIES

SOME PECULIARITIES

This does not work.

```
# subdir/CMakeLists.txt
...
add_library( someTarget SHARED )
target_sources( someTarget PRIVATE source1.cpp )
```

```
# CMakeLists.txt
...
add_subdirectory( subdir )

add_library( someOtherTarget SHARED )
target_sources( someOtherTarget PRIVATE source2.cpp )

# Add a dependency to the build-requirements of a target from another directory.
target_link_libraries( someTarget PRIVATE someOtherTarget )
```

THIS DOES NOT WORK.

```
# subdir/CMakeLists.txt
...
add_library( someTarget SHARED )
target_sources( someTarget PRIVATE source1.cpp )
```

```
# CMakeLists.txt
...
add_subdirectory( subdir )

add_library( someOtherTarget SHARED )
target_sources( someOtherTarget PRIVATE source2.cpp )

# Add a dependency to the build-requirements of a target from another directory.
target_link_libraries( someTarget PRIVATE someOtherTarget )
```

- `target_link_libraries` called from *another directory scope*
  - cannot modify **build-requirements** of a target

However, this does work.

```
# subdir/CMakeLists.txt
...
add_library( someTarget SHARED )
target_sources( someTarget PRIVATE source1.cpp )
```

```
# CMakeLists.txt
...
add_subdirectory( subdir )

add_library( someOtherTarget SHARED )
target_sources( someOtherTarget PRIVATE source2.cpp )

# Add a dependency to the usage-requirements of a target from another directory.
target_link_libraries( someTarget INTERFACE someOtherTarget )
```

- `target_link_libraries` called from *another directory scope*
  - cannot modify **build-requirements** of a target, but
  - can modify **usage-requirements** of a target

HOWEVER, THIS DOES WORK.

```
# subdir/CMakeLists.txt
...
add_library( someTarget SHARED )
target_sources( someTarget PRIVATE source1.cpp )
```

```
# CMakeLists.txt
...
add_subdirectory( subdir )

add_library( someOtherTarget SHARED )
target_sources( someOtherTarget PRIVATE source2.cpp )

# Add a dependency to the usage-requirements of a target from another directory.
target_link_libraries( someTarget INTERFACE someOtherTarget )
```

- `target_link_libraries` called from *another directory scope*
  - cannot modify **build-requirements** of a target, but
  - can modify **usage-requirements** of a target by accident!

CMake 3.13 lifts this restriction

- `target_link_libraries` can now be called from everywhere to add dependencies. (https://gitlab.kitware.com/cmake/cmake/issues/17943)

CMake 3.13 lifts this restriction

- `target_link_libraries` can now be called from everywhere to add dependencies. (https://gitlab.kitware.com/cmake/cmake/issues/17943)
- Additionally, CMake 3.13 now also allows to `install` targets created in different directory scope. (https://gitlab.kitware.com/cmake/cmake/issues/14444)

Some ideas for future changes to CMake

`ALIAS`ING `UNKNOWN IMPORTED` TARGETS

- Although `ALIAS`ing now also works with `IMPORTED` targets, it does not work completely.

`ALIAS`ING `UNKNOWN IMPORTED` TARGETS

- Although `ALIAS`ing now also works with `IMPORTED` targets, it does not work completely.
  - `UNKNOWN IMPORTED` targets still cannot be aliased.

`ALIAS`ING `UNKNOWN IMPORTED` TARGETS

- Although `ALIAS`ing now also works with `IMPORTED` targets, it does not work completely.
  - `UNKNOWN IMPORTED` targets still cannot be aliased.

⇒ This should be fixed.

https://gitlab.kitware.com/cmake/cmake/issues/18327

Simplify propagation of *object-files*

and

Allow circular-dependencies among `OBJECT` targets

- The propagation of *object-files* from `OBJECT` targets is weird (as you have seen).

Simplify propagation of *object-files*

and

Allow circular-dependencies among `OBJECT` targets

- The propagation of *object-files* from `OBJECT` targets is weird (as you have seen).
  - Especially, because propagation from one `OBJECT` target to another does not happen.

Simplify propagation of *object-files* and Allow circular-dependencies among `OBJECT` targets

- The propagation of *object-files* from `OBJECT` targets is weird (as you have seen).
  - Especially, because propagation from one `OBJECT` target to another does not happen.

⇒ This should become simpler.

https://gitlab.kitware.com/cmake/cmake/issues/17905
https://gitlab.kitware.com/cmake/cmake/issues/18090

`ADD_CUSTOM_COMMAND` AND `OBJECT` TARGETS

- `add_custom_command` can be used with `PRE_BUILD` or `POST_BUILD` option to call some commands *before* or *after* building a target.

`ADD_CUSTOM_COMMAND` AND `OBJECT` TARGETS

- `add_custom_command` can be used with `PRE_BUILD` or `POST_BUILD` option to call some commands *before* or *after* building a target.
  - However, this still does not work with `OBJECT` targets.

`ADD_CUSTOM_COMMAND` AND `OBJECT` TARGETS

- `add_custom_command` can be used with `PRE_BUILD` or `POST_BUILD` option to call some commands *before* or *after* building a target.
  - However, this still does not work with `OBJECT` targets.

⇒ This should be fixed.

https://gitlab.kitware.com/cmake/cmake/issues/17081

Let's wrap it up

Takeaway

Takeaway

- Switch to *Modern CMake*!

TAKEAWAY

- Switch to ~~*Modern CMake*!~~ *More Modern CMake*!

Takeaway

- Switch to ~~*Modern CMake*!~~ *More Modern CMake*!
    - It is cleaner.
    - It is smaller.
    - It is less error-prone.

Takeaway

- Switch to ~~*Modern CMake*~~! *More Modern CMake*!
  - It is cleaner.
  - It is smaller.
  - It is less error-prone.
- Use newest CMake version if possible. (At least CMake 3.12.)

TAKEAWAY

- Switch to ~~*Modern CMake*~~! *More Modern CMake*!
  - It is cleaner.
  - It is smaller.
  - It is less error-prone.
- Use newest CMake version if possible. (At least CMake 3.12.)
  - Always create targets with no sources, first.

TAKEAWAY

- Switch to ~~*Modern CMake*~~! *More Modern CMake*!
  - It is cleaner.
  - It is smaller.
  - It is less error-prone.
- Use newest CMake version if possible. (At least CMake 3.12.)
  - Always create targets with no sources, first.
  - Use `target_...` commands to add *build-/usage-requirements*.

Takeaway

- Switch to ~~*Modern CMake*~~! *More Modern CMake*!
  - It is cleaner.
  - It is smaller.
  - It is less error-prone.
- Use newest CMake version if possible. (At least CMake 3.12.)
  - Always create targets with no sources, first.
  - Use `target_...` commands to add *build-/usage-requirements*.
  - Use `IMPORTED` targets for external libraries.
    But, prefer `find_package` or `EXPORT`ed targets to creating them yourself.

Takeaway

- Switch to ~~*Modern CMake*!~~ *More Modern CMake*!
  - It is cleaner.
  - It is smaller.
  - It is less error-prone.
- Use newest CMake version if possible. (At least CMake 3.12.)
  - Always create targets with no sources, first.
  - Use `target_...` commands to add *build-/usage-requirements*.
  - Use `IMPORTED` targets for external libraries.
    But, prefer `find_package` or `EXPORT`ed targets to creating them yourself.
- Contribute to CMake!
  - Write issues in CMake's issue-tracker for errors and missing features.
  - Try to provide merge-requests for these (and other) issues.

References

- *Craig Scott*'s **"Professional CMake: A Practical Guide"** e-book
  Buy it at: https://crascit.com/professional-cmake/
- The living document about *Modern CMake*
  Read/Contribute at: https://cliutils.gitlab.io/modern-cmake/
- *Daniel Pfeiffer*'s **"Effective CMake"** talk
  Watch it at: https://www.youtube.com/watch?v=bsXLMQ6WgIk
- *Steven Kelly*'s "**Embracing Modern CMake**" talk
  Watch it at: https://www.youtube.com/watch?v=JsjI5xr1jxM
- *Mathieu Ropert*'s "**Modern CMake for modular design**" talk
  Watch it at: https://www.youtube.com/watch?v=ztrnb-bVVPo

Thank you!

Any questions?

Examples from slides: http://github.com/Bagira80/More-Modern-CMake

Bonus slides

Basics for (More) Modern CMake

CMAKE_MINIMUM_REQUIRED

```
# Minimal version 3.12, maximal version 3.13.
cmake_minimum_required( VERSION 3.12...3.13 )
```

CMAKE_MINIMUM_REQUIRED

```
# Minimal version 3.12, maximal version 3.13.
cmake_minimum_required( VERSION 3.12...3.13 )
```

- Call *`cmake_minimum_required`*

CMAKE_MINIMUM_REQUIRED

```
# Minimal version 3.12, maximal version 3.13.
cmake_minimum_required( VERSION 3.12...3.13 )
```

- Call `cmake_minimum_required`
  - *required:* at begin of ***top-level*** `CMakeLists.txt` file.

CMAKE_MINIMUM_REQUIRED

```
# Minimal version 3.12, maximal version 3.13.
cmake_minimum_required( VERSION 3.12...3.13 )
```

- Call `cmake_minimum_required`
  - *required:* at begin of ***top-level*** `CMakeLists.txt` file.
  - *easier:* at begin of ***all*** `CMakeLists.txt` files.

CMAKE_MINIMUM_REQUIRED

```
# Minimal version 3.12, maximal version 3.13.
cmake_minimum_required( VERSION 3.12...3.13 )
```

- Call `cmake_minimum_required`
  - *required:* at begin of ***top-level*** `CMakeLists.txt` file.
  - *easier:* at begin of ***all*** `CMakeLists.txt` files.
- Sets CMake *policies* to defaults of specific CMake version.
  - `cmake_policy` allows to modify policies again.
    (*Policy-scopes* exist, too.)

CMAKE_MINIMUM_REQUIRED

```
# Minimal version 3.12, maximal version 3.13.
cmake_minimum_required( VERSION 3.12...3.13 )
```

- Call `cmake_minimum_required`
  - *required:* at begin of ***top-level*** `CMakeLists.txt` file.
  - *easier:* at begin of ***all*** `CMakeLists.txt` files.
- Sets CMake *policies* to defaults of specific CMake version.
  - `cmake_policy` allows to modify policies again.
    (*Policy-scopes* exist, too.)
- At least use version 3.12 as minimal version!

CMAKE_MINIMUM_REQUIRED

```
# Minimal version 3.12, maximal version 3.13.
cmake_minimum_required( VERSION 3.12...3.13 )
```

- Call `cmake_minimum_required`
  - *required:* at begin of ***top-level*** `CMakeLists.txt` file.
  - *easier:* at begin of ***all*** `CMakeLists.txt` files.
- Sets CMake *policies* to defaults of specific CMake version.
  - `cmake_policy` allows to modify policies again.
    (*Policy-scopes* exist, too.)
- At least use version 3.12 as minimal version!
  - The *version range* `<min-version>...<max-version>` syntax was introduced in *3.12*, but is backwards-compatible.

PROJECT

```
# Define a project for the current CMakeLists.txt.
project( <project_name>
         VERSION <major>[.<minor>[.<patch>[.<tweak>]]]
         DESCRIPTION <project_description_string>
         [HOMEPAGE_URL <url_string>]
         [LANGUAGES <language_name>...] )
```

PROJECT

```
# Define a project for the current CMakeLists.txt.
project( <project_name>
         VERSION <major>[.<minor>[.<patch>[.<tweak>]]]
         DESCRIPTION <project_description_string>
         [HOMEPAGE_URL <url_string>]
         [LANGUAGES <language_name>...] )
```

- Call *after* `cmake_minimum_required`
  - but as early as possible.

PROJECT

```
# Define a project for the current CMakeLists.txt.
project( <project_name>
          VERSION <major>[.<minor>[.<patch>[.<tweak>]]]
          DESCRIPTION <project_description_string>
          [HOMEPAGE_URL <url_string>]
          [LANGUAGES <language_name>...] )
```

- Call *after* `cmake_minimum_required`
  - but as early as possible.
- Sets variables containing: project-name, version etc.

PROJECT

```
# Define a project for the current CMakeLists.txt.
project( <project_name>
         VERSION <major>[.<minor>[.<patch>[.<tweak>]]]
         DESCRIPTION <project_description_string>
         [HOMEPAGE_URL <url_string>]
         [LANGUAGES <language_name>...] )
```

- Call *after* `cmake_minimum_required`
  - but as early as possible.
- Sets variables containing: project-name, version etc.
- Default values for `LANGUAGES`: `C` and `CXX`
  - Other values: `FORTRAN`, `CUDA`, `CSharp`, `ASM`, `Java` (!) ...

INCLUDE

```
# Verbatim copy the content of a file.
include( <file_path | module>
         [OPTIONAL]
         [RESULT_VARIABLE <variable_name>]
         [NO_POLICY_SCOPE] )
```

INCLUDE

```
# Verbatim copy the content of a file.
include( <file_path | module>
           [OPTIONAL]
           [RESULT_VARIABLE <variable_name>]
           [NO_POLICY_SCOPE] )
```

- Similar to `#include "file_path"` as used in C/C++.
  - Includes the content of a file / module in the current file.

INCLUDE

```
# Verbatim copy the content of a file.
include( <file_path | module>
         [OPTIONAL]
         [RESULT_VARIABLE <variable_name>]
         [NO_POLICY_SCOPE] )
```

- Similar to `#include "file_path"` as used in C/C++.
  - Includes the content of a file / module in the current file.
- If not a file ⇒ a module `<module>.cmake`

INCLUDE

```
# Verbatim copy the content of a file.
include( <file_path | module>
         [OPTIONAL]
         [RESULT_VARIABLE <variable_name>]
         [NO_POLICY_SCOPE] )
```

- Similar to `#include "file_path"` as used in C/C++.
  - Includes the content of a file / module in the current file.
- If not a file ⇒ a module `<module>.cmake`
  1. searched in paths from `CMAKE_MODULE_PATH` variable.
  2. searched in CMake's default module-search path.

INCLUDE

```
# Verbatim copy the content of a file.
include( <file_path | module>
         [OPTIONAL]
         [RESULT_VARIABLE <variable_name>]
         [NO_POLICY_SCOPE] )
```

- Similar to `#include "file_path"` as used in C/C++.
  - Includes the content of a file / module in the current file.
- If not a file ⇒ a module `<module>.cmake`
  1. searched in paths from `CMAKE_MODULE_PATH` variable.
  2. searched in CMake's default module-search path.
- `RESULT_VARIABLE`: The full filename of the included file will be stored in `<variable_name>`.

INCLUDE

```
# Verbatim copy the content of a file.
include( <file_path | module>
         [OPTIONAL]
         [RESULT_VARIABLE <variable_name>]
         [NO_POLICY_SCOPE] )
```

- Similar to `#include "file_path"` as used in C/C++.
  - Includes the content of a file / module in the current file.
- If not a file ⇒ a module `<module>.cmake`
  1. searched in paths from `CMAKE_MODULE_PATH` variable.
  2. searched in CMake's default module-search path.
- `RESULT_VARIABLE`: The full filename of the included file will be stored in `<variable_name>`.
- `OPTIONAL` and `NO_POLICY_SCOPE`: Mostly, not needed.

INCLUDE_GUARD

```
# Prevent including this file multiple times.
include_guard( [DIRECTORY | GLOBAL] )
```

INCLUDE_GUARD

```cmake
# Prevent including this file multiple times.
include_guard( [DIRECTORY | GLOBAL] )
```

- Similar to `#pragma once` as used by many C++ compilers.

INCLUDE_GUARD

```cmake
# Prevent including this file multiple times.
include_guard( [DIRECTORY | GLOBAL] )
```

- Similar to `#pragma once` as used by many C++ compilers.
- Prevents including the current file again
  - `GLOBAL`: everywhere,
  - `DIRECTORY`: in this directory or its subdirectories,
  - *no args*: in variable scope.

Example for `CMakeLists.txt` begin

```
# CMakeLists.txt

cmake_minimum_required( VERSION 3.12...3.13 )

# Get version and description from other file.
include( "${CMAKE_CURRENT_LIST_DIR}/project-meta-info.in" )
# Use version and description variables from other file.
project( MyExampleProject
         VERSION ${project_version}
         DESCRIPTION ${project_description} )
...
```

```
# project-meta-info.in

include_guard()

# The version number of this project.
set( project_version 0.98.2 )
# The description of this project.
set( project_description "This is just an example project. "
                         "Do not expect anything here." )
```