cmake-compile-features(7)

Contents

Introduction

Project source code may depend on, or be conditional on, the availability of certain features of the compiler. There are three use-cases which arise: Compile Feature Requirements, Optional Compile Features and Conditional Compilation Options.

While features are typically specified in programming language standards, CMake provides a primary user interface based on granular handling of the features, not the language standard that introduced the feature.

The **CMAKE_C_KNOWN_FEATURES** and **CMAKE_CXX_KNOWN_FEATURES** global properties contain all the features known to CMake, regardless of compiler support for the feature. The **CMAKE_C_COMPILE_FEATURES** and **CMAKE_CXX_COMPILE_FEATURES** variables contain all features CMake knows are known to the compiler, regardless of language standard or compile flags needed to use them.

Features known to CMake are named mostly following the same convention as the Clang feature test macros. The are some exceptions, such as CMake using `cxx_final` and `cxx_override` instead of the single `cxx_override_control` used by Clang.

Compile Feature Requirements

Compile feature requirements may be specified with the **target_compile_features()** command. For example, if a target must be compiled with compiler support for the **cxx_constexpr** feature:

```
add_library(mylib requires_constexpr.cpp)
target_compile_features(mylib PRIVATE cxx_constexpr)
```

In processing the requirement for the `cxx_constexpr` feature, **cmake(1)** will ensure that the in-use C++ compiler is capable of the feature, and will add any necessary flags such as `-std=gnu++11` to the compile lines of C++ files in the `mylib` target. A `FATAL_ERROR` is issued if the compiler is not capable of the feature.

The exact compile flags and language standard are deliberately not part of the user

interface for this use-case. CMake will compute the appropriate compile flags to use by considering the features specified for each target.

Such compile flags are added even if the compiler supports the particular feature without the flag. For example, the GNU compiler supports variadic templates (with a warning) even if `-std=gnu++98` is used. CMake adds the `-std=gnu++11` flag if `cxx_variadic_templates` is specified as a requirement.

In the above example, `mylib` requires `cxx_constexpr` when it is built itself, but consumers of `mylib` are not required to use a compiler which supports `cxx_constexpr`. If the interface of `mylib` does require the `cxx_constexpr` feature (or any other known feature), that may be specified with the `PUBLIC` or `INTERFACE` signatures of **`target_compile_features()`**:

```
add_library(mylib requires_constexpr.cpp)
# cxx_constexpr is a usage-requirement
target_compile_features(mylib PUBLIC cxx_constexpr)

# main.cpp will be compiled with -std=gnu++11 on GNU for cxx_constexpr.
add_executable(myexe main.cpp)
target_link_libraries(myexe mylib)
```

Feature requirements are evaluated transitively by consuming the link implementation. See **`cmake-buildsystem(7)`** for more on transitive behavior of build properties and usage requirements.

Because the **`CXX_EXTENSIONS`** target property is `ON` by default, CMake uses extended variants of language dialects by default, such as `-std=gnu++11` instead of `-std=c++11`. That target property may be set to `OFF` to use the non-extended variant of the dialect flag. Note that because most compilers enable extensions by default, this could expose cross-platform bugs in user code or in the headers of third-party dependencies.

Optional Compile Features

Compile features may be preferred if available, without creating a hard requirement. For example, a library may provides alternative implementations depending on whether the `cxx_variadic_templates` feature is available:

```
#if Foo_COMPILER_CXX_VARIADIC_TEMPLATES
template<int I, int... Is>
struct Interface;

template<int I>
struct Interface<I>
{
  static int accumulate()
  {
    return I;
  }
};

template<int I, int... Is>
struct Interface
{
  static int accumulate()
```

```
  {
    return I + Interface<Is...>::accumulate();
  }
};
#else
template<int I1, int I2 = 0, int I3 = 0, int I4 = 0>
struct Interface
{
  static int accumulate() { return I1 + I2 + I3 + I4; }
};
#endif
```

Such an interface depends on using the correct preprocessor defines for the compiler features. CMake can generate a header file containing such defines using the **WriteCompilerDetectionHeader** module. The module contains the `write_compiler_detection_header` function which accepts parameters to control the content of the generated header file:

```
write_compiler_detection_header(
  FILE "${CMAKE_CURRENT_BINARY_DIR}/foo_compiler_detection.h"
  PREFIX Foo
  COMPILERS GNU
  FEATURES
    cxx_variadic_templates
)
```

Such a header file may be used internally in the source code of a project, and it may be installed and used in the interface of library code.

For each feature listed in `FEATURES`, a preprocessor definition is created in the header file, and defined to either `1` or `0`.

Additionally, some features call for additional defines, such as the `cxx_final` and `cxx_override` features. Rather than being used in `#ifdef` code, the `final` keyword is abstracted by a symbol which is defined to either `final`, a compiler-specific equivalent, or to empty. That way, C++ code can be written to unconditionally use the symbol, and compiler support determines what it is expanded to:

```
struct Interface {
  virtual void Execute() = 0;
};

struct Concrete Foo_FINAL {
  void Execute() Foo_OVERRIDE;
};
```

In this case, `Foo_FINAL` will expand to `final` if the compiler supports the keyword, or to empty otherwise.

In this use-case, the CMake code will wish to enable a particular language standard if available from the compiler. The **CXX_STANDARD** target property variable may be set to the desired language standard for a particular target, and the **CMAKE_CXX_STANDARD** may be set to influence all following targets:

```
write_compiler_detection_header(
```

```
  FILE "${CMAKE_CURRENT_BINARY_DIR}/foo_compiler_detection.h"
  PREFIX Foo
  COMPILERS GNU
  FEATURES
    cxx_final cxx_override
)

# Includes foo_compiler_detection.h and uses the Foo_FINAL symbol
# which will expand to 'final' if the compiler supports the requested
# CXX_STANDARD.
add_library(foo foo.cpp)
set_property(TARGET foo PROPERTY CXX_STANDARD 11)

# Includes foo_compiler_detection.h and uses the Foo_FINAL symbol
# which will expand to 'final' if the compiler supports the feature,
# even though CXX_STANDARD is not set explicitly.  The requirement of
# cxx_constexpr causes CMake to set CXX_STANDARD internally, which
# affects the compile flags.
add_library(foo_impl foo_impl.cpp)
target_compile_features(foo_impl PRIVATE cxx_constexpr)
```

The `write_compiler_detection_header` function also creates compatibility code for other features which have standard equivalents. For example, the `cxx_static_assert` feature is emulated with a template and abstracted via the `<PREFIX>_STATIC_ASSERT` and `<PREFIX>_STATIC_ASSERT_MSG` function-macros.

Conditional Compilation Options

Libraries may provide entirely different header files depending on requested compiler features.

For example, a header at `with_variadics/interface.h` may contain:

```
template<int I, int... Is>
struct Interface;

template<int I>
struct Interface<I>
{
  static int accumulate()
  {
    return I;
  }
};

template<int I, int... Is>
struct Interface
{
  static int accumulate()
  {
    return I + Interface<Is...>::accumulate();
  }
};
```

while a header at `no_variadics/interface.h` may contain:

```
template<int I1, int I2 = 0, int I3 = 0, int I4 = 0>
struct Interface
```

```
{
  static int accumulate() { return I1 + I2 + I3 + I4; }
};
```

It would be possible to write a abstraction `interface.h` header containing something like:

```
#include "foo_compiler_detection.h"
#if Foo_COMPILER_CXX_VARIADIC_TEMPLATES
#include "with_variadics/interface.h"
#else
#include "no_variadics/interface.h"
#endif
```

However this could be unmaintainable if there are many files to abstract. What is needed is to use alternative include directories depending on the compiler capabilities.

CMake provides a `COMPILE_FEATURES` **generator expression** to implement such conditions. This may be used with the build-property commands such as **target_include_directories()** and **target_link_libraries()** to set the appropriate **buildsystem** properties:

```
add_library(foo INTERFACE)
set(with_variadics ${CMAKE_CURRENT_SOURCE_DIR}/with_variadics)
set(no_variadics ${CMAKE_CURRENT_SOURCE_DIR}/no_variadics)
target_include_directories(foo
  INTERFACE
    "$<$<COMPILE_FEATURES:cxx_variadic_templates>:${with_variadics}>"
    "$<$<NOT:$<COMPILE_FEATURES:cxx_variadic_templates>>:${no_variadics}>"
  )
```

Consuming code then simply links to the `foo` target as usual and uses the feature-appropriate include directory

```
add_executable(consumer_with consumer_with.cpp)
target_link_libraries(consumer_with foo)
set_property(TARGET consumer_with CXX_STANDARD 11)

add_executable(consumer_no consumer_no.cpp)
target_link_libraries(consumer_no foo)
```

Supported Compilers

CMake is currently aware of the **language standards** and **compile features** available from the following **compiler ids** as of the versions specified for each:

- `AppleClang`: Apple Clang for Xcode versions 4.4 though 6.2.
- `Clang`: Clang compiler versions 2.9 through 3.4.
- `GNU`: GNU compiler versions 4.4 through 5.0.
- `MSVC`: Microsoft Visual Studio versions 2010 through 2015.
- `SunPro`: Oracle SolarisStudio version 12.4.