NAME

cmake-compile-features − CMake Compile Features Reference

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**, **CMAKE_CUDA_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**, **CMAKE_CUDA_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. There are some exceptions, such as CMake using **cxx_final** and **cxx_override** instead of the single **cxx_override_control** used by Clang.

Note that there are no separate compile features properties or variables for the **OBJC** or **OBJCXX** languages. These are based off **C** or **C++** respectively, so the properties and variables for their corresponding base language should be used instead.

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**F ATAL_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.

Requiring Language Standards

In projects that use a large number of commonly available features from a particular language standard (e.g. C++ 11) one may specify a meta−feature (e.g. **cxx_std_11**) that requires use of a compiler mode that is at minimum aware of that standard, but could be greater. This is simpler than specifying all the features individually, but does not guarantee the existence of any particular feature. Diagnosis of use of unsupported features will be delayed until compile time.

For example, if C++ 11 features are used extensively in a project's header files, then clients must use a compiler mode that is no less than C++ 11. This can be requested with the code:

```
target_compile_features(mylib PUBLIC cxx_std_11)
```

In this example, CMake will ensure the compiler is invoked in a mode of at−least C++ 11 (or C++ 14, C++ 17, ...), adding flags such as **−std=gnu++11** if necessary. This applies to sources within**mylib** as well as any dependents (that may include headers from **mylib**).

Availability of Compiler Extensions

The **<LANG>_EXTENSIONS** target property defaults to the compiler's default (see **CMAKE_<LANG>_EXTENSIONS_DEFAULT**). Note that because most compilers enable extensions by default, this may expose portability bugs in user code or in the headers of third−party dependencies.

<LANG>_EXTENSIONS used to default to **ON**. See **CMP0128**.

OPTIONAL COMPILE FEATURES

Compile features may be preferred if available, without creating a hard requirement. This can be achieved by *not* specifying features with **target_compile_features()** and instead checking the compiler capabilities with preprocessor conditions in project code.

In this use−case, the project may wish to establish a particular language standard if available from the compiler, and use preprocessor conditions to detect the features actually available. A language standard may be established by *Requiring Language Standards* using **target_compile_features()** with meta−features like **cxx_std_11**, or by setting the **CXX_STANDARD** target property or **CMAKE_CXX_STANDARD** variable.

See also policy **CMP0120** and legacy documentation on Example Usage of the deprecated **WriteCompilerDetectionHeader** module.

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 may be possible to write an abstraction **interface.h** header containing something like:

```
#ifdef HAVE_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 **C++ 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+.
- **Clang**: Clang compiler versions 2.9+.
- **GNU**: GNU compiler versions 4.4+.
- **MSVC**: Microsoft Visual Studio versions 2010+.
- **SunPro**: Oracle SolarisStudio versions 12.4+.
- **Intel**: Intel compiler versions 12.1+.

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

- all compilers and versions listed above for C++.
- **GNU**: GNU compiler versions 3.4+

CMake is currently aware of the **C++ standards** and their associated meta−features (e.g. **cxx_std_11**) available from the following **compiler ids** as of the versions specified for each:

- **Cray**: Cray Compiler Environment version 8.1+.
- **Fujitsu**: Fujitsu HPC compiler 4.0+.
- **PGI**: PGI version 12.10+.
- **NVHPC**: NVIDIA HPC compilers version 11.0+.
- **TI**: Texas Instruments compiler.
- **XL**: IBM XL version 10.1+.

CMake is currently aware of the **C standards** and their associated meta−features (e.g. **c_std_99**) available from the following **compiler ids** as of the versions specified for each:

- all compilers and versions listed above with only meta−features for C++.

CMake is currently aware of the **CUDA standards** and their associated meta−features (e.g. **cuda_std_11**) available from the following **compiler ids** as of the versions specified for each:

- **Clang**: Clang compiler 5.0+.
- **NVIDIA**: NVIDIA nvcc compiler 7.5+.

COPYRIGHT