

NAME

cmake-policies – CMake Policies Reference

INTRODUCTION

CMake policies introduce behavior changes while preserving compatibility for existing project releases. Policies are deprecation mechanisms, not feature toggles. Each policy documents a deprecated **OLD** behavior and a preferred **NEW** behavior. Projects must be updated over time to use the **NEW** behavior, but their existing releases will continue to work with the **OLD** behavior.

Updating Projects

When policies are newly introduced by a version of CMake, their **OLD** behaviors are immediately deprecated by that version of CMake and later. Projects should be updated to use the **NEW** behaviors of the policies as soon as possible.

Use the `cmake_minimum_required()` command to record the latest version of CMake for which a project has been updated. For example:

```
cmake_minimum_required(VERSION 3.10...3.31)
```

This uses the `<min>...<max>` syntax to enable the **NEW** behaviors of policies introduced in CMake 3.31 and earlier while only requiring a minimum version of CMake 3.10. The project is expected to work with both the **OLD** and **NEW** behaviors of policies introduced between those versions.

Transition Schedule

To help projects port to the **NEW** behaviors of policies on their own schedule, CMake offers a transition period:

- If a policy is not set by a project, CMake uses its **OLD** behavior, but may warn that the policy has not been set.
- Users running CMake may silence the warning without modifying a project by setting the `CMAKE_POLICY_DEFAULT_CMP<NNNN>` variable as a cache entry on the `cmake(1)` command line:

```
cmake -DCMAKE_POLICY_DEFAULT_CMP0990=OLD ...
```

- Projects may silence the warning by using the `cmake_policy()` command to explicitly set the policy to **OLD** or **NEW** behavior:

```
if(POLICY CMP0990)
  cmake_policy(SET CMP0990 NEW)
endif()
```

NOTE:

A policy should almost never be set to **OLD**, except to silence warnings in an otherwise frozen or stable codebase, or temporarily as part of a larger migration path.

- If a policy is set to **OLD** by a project, CMake versions released at least 2 years after the version that introduced a policy may issue a warning that the policy's **OLD** behavior will be removed from a future version of CMake.
- If a policy is not set to **NEW** by a project, CMake versions released at least 6 years after the version that introduced a policy, and whose major version number is higher, may issue an error that the policy's **OLD** behavior has been removed.

SUPPORTED POLICIES

The following policies are supported.

Policies Introduced by CMake 4.0**CMP0185**

Added in version 4.0.

FindRuby no longer provides upper-case **RUBY_*** variables.

FindRuby in CMake 3.31 and below provided result variables named with an upper-case **RUBY_*** prefix. CMake 3.18 renamed them with a **Ruby_*** prefix to match the name of the package, but continued to provide the upper-case **RUBY_*** variables for compatibility.

CMake 4.0 and above prefer to provide only **Ruby_*** variables and no longer provide **RUBY_*** variables. This policy provides compatibility with projects that have not been updated.

The **OLD** behavior for this policy is to provide both **Ruby_*** and **RUBY_*** variables. The **NEW** behavior for this policy is to provide only **Ruby_*** variables and not **RUBY_*** variables.

This policy was introduced in CMake version 4.0. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0184

Added in version 4.0.

MSVC runtime checks flags are selected by an abstraction.

Compilers targeting the MSVC ABI have flags to select the runtime checks. Runtime checks selection typically varies with build configuration.

In CMake 3.31 and below, runtime checks flags are added to the default *CMAKE_<LANG>_FLAGS_<CONFIG>* cache entries by CMake automatically. This allows users to edit their cache entries to adjust the flags. However, the presence of such default flags is problematic for projects that want to choose different runtime checks programmatically. In particular, it requires string editing of the *CMAKE_<LANG>_FLAGS_<CONFIG>* variables with knowledge of the CMake builtin defaults so they can be replaced.

CMake 4.0 and above prefer to leave the runtime checks flags out of the default *CMAKE_<LANG>_FLAGS_<CONFIG>* values and instead offer a first-class abstraction. The *CMAKE_MSVC_RUNTIME_CHECKS* variable and *MSVC_RUNTIME_CHECKS* target property may be set to select the MSVC runtime checks. If they are not set, CMake enables runtime checks in **Debug** configuration using the default value `$$<CONFIG:Debug>:StackFrameErrorCheck;UninitializedVariable`, if supported by the compiler, or empty value otherwise.

This policy provides compatibility with projects that have not been updated to be aware of the abstraction. The policy setting takes effect as of the first *project()* or *enable_language()* command that enables a language whose compiler targets the MSVC ABI.

NOTE:

Once the policy has taken effect at the top of a project, that choice will be used throughout the tree. In projects that have nested projects in subdirectories, be sure to confirm if everything is working with the selected policy behavior.

The **OLD** behavior for this policy is to place MSVC runtimes checks flags in the default `CMAKE_<LANG>_FLAGS_<CONFIG>` cache entries and ignore the `CMAKE_MSVC_RUNTIME_CHECKS` abstraction. The **NEW** behavior for this policy is to *not* place MSVC runtime checks flags in the default cache entries and use the abstraction instead.

This policy was introduced in CMake version 4.0. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0183

Added in version 4.0.

`add_feature_info()` supports full *Condition Syntax*.

The `<enabled>` parameter accepts a *semicolon-separated list* of conditions. CMake 3.31 and lower evaluate each **condition** as `if(${condition})`, which does not properly handle conditions with nested paren groups. CMake 4.0 and above instead prefer to evaluate each **condition** as `if(<condition>)`, where `<condition>` is re-parsed as if literally written in a call to `if()`. This allows expressions like:

```
"A AND (B OR C)"
```

but requires expressions like:

```
"FOO MATCHES (UPPER|lower)"
```

to be re-written as:

```
"FOO MATCHES \"(UPPER|lower)\""
```

Policy **CMP0183** provides compatibility for projects that have not been updated to expect the new behavior.

This policy was introduced in CMake version 4.0. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0182

Added in version 4.0.

Create shared library archives by default on AIX.

CMake 3.30 and below always represented **SHARED** library targets as plain shared object `.so` files. This is consistent with other UNIX platforms, but is not the preferred convention on AIX. CMake 3.31 added the `AIX_SHARED_LIBRARY_ARCHIVE` target property to create a shared library archive: the shared object `.so` file is placed inside an archive `.a` file. However, the behavior was disabled by default for compatibility with existing projects that do not set `AIX_SHARED_LIBRARY_ARCHIVE`.

CMake 4.0 and above prefer, when `AIX_SHARED_LIBRARY_ARCHIVE` is not set, to enable creation of

shared library archives by default because it is the preferred convention on AIX. This policy provides compatibility for projects that have not been updated.

The **OLD** behavior for this policy is to disable shared library archives when `AIX_SHARED_LIBRARY_ARCHIVE` is not set. The **NEW** behavior for this policy is to enable shared library archives when `AIX_SHARED_LIBRARY_ARCHIVE` is not set.

This policy was introduced in CMake version 4.0. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0181

Added in version 4.0.

The `CMAKE_EXE_LINKER_FLAGS`, `CMAKE_EXE_LINKER_FLAGS_<CONFIG>`, `CMAKE_SHARED_LINKER_FLAGS`, `CMAKE_SHARED_LINKER_FLAGS_<CONFIG>`, `CMAKE_MODULE_LINKER_FLAGS`, and `CMAKE_MODULE_LINKER_FLAGS_<CONFIG>` variables are parsed and re-quoted and support the **LINKER:** prefix.

CMake 3.31 and below use the content of these variables as is.

CMake 4.0 and above parse the content of these variables and manage the escaping of special characters. Moreover, the **LINKER:** prefix is now recognized and expanded.

The **OLD** behavior of this policy is to consume the content of the `CMAKE_EXE_LINKER_FLAGS`, `CMAKE_EXE_LINKER_FLAGS_<CONFIG>`, `CMAKE_SHARED_LINKER_FLAGS`, `CMAKE_SHARED_LINKER_FLAGS_<CONFIG>`, `CMAKE_MODULE_LINKER_FLAGS`, and `CMAKE_MODULE_LINKER_FLAGS_<CONFIG>` variables as is.

The **NEW** behavior of this policy is to parse and re-quote the content of the `CMAKE_EXE_LINKER_FLAGS`, `CMAKE_EXE_LINKER_FLAGS_<CONFIG>`, `CMAKE_SHARED_LINKER_FLAGS`, `CMAKE_SHARED_LINKER_FLAGS_<CONFIG>`, `CMAKE_MODULE_LINKER_FLAGS`, and `CMAKE_MODULE_LINKER_FLAGS_<CONFIG>` variables as well as to expand the **LINKER:** prefix.

This policy was introduced in CMake version 4.0. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.31

CMP0180

Added in version 3.31.

`project()` always sets `<PROJECT-NAME>_*` as normal variables.

In CMake 3.29 and below, the `project()` command set `<PROJECT-NAME>_SOURCE_DIR`, `<PROJECT-NAME>_BINARY_DIR`, and `<PROJECT-NAME>_IS_TOP_LEVEL` as cache entries, but not as normal variables. CMake 3.30 started setting them as normal variables, but only if they are already set

as normal variables. This was needed to preserve support for some *FetchContent* use cases under policy *CMP0169*'s **NEW** behavior, while also preserving behavior of nested directories that call *project()* with the same project name. See release notes for 3.30.3, 3.30.4, and 3.30.5 for details.

CMake 3.31 and later prefer to always set `<PROJECT-NAME>_SOURCE_DIR`, `<PROJECT-NAME>_BINARY_DIR`, and `<PROJECT-NAME>_IS_TOP_LEVEL`, as both cache entries and normal variables, regardless of what cache or normal variables already exist. This policy provides compatibility for projects that have not been updated to expect this behavior.

The **OLD** behavior for this policy will only set normal variables for `<PROJECT-NAME>_SOURCE_DIR`, `<PROJECT-NAME>_BINARY_DIR`, and `<PROJECT-NAME>_IS_TOP_LEVEL` if there is already a normal variable by that name when *project()* is called. The **NEW** behavior for this policy will always set normal variables for `<PROJECT-NAME>_SOURCE_DIR`, `<PROJECT-NAME>_BINARY_DIR`, and `<PROJECT-NAME>_IS_TOP_LEVEL` when *project()* is called.

This policy was introduced in CMake version 3.31. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0179

Added in version 3.31.

De-duplication of static libraries on link lines keeps first occurrence. This policy is only relevant when policy *CMP0156* is set to **NEW**.

Based on the linker capabilities, the static libraries can be de-duplicated. See policy *CMP0156* for more information.

CMake 3.30 and below may choose to keep, on some platforms, the last occurrence of the static libraries rather than the first occurrence when they are de-duplicated.

CMake 3.31 and above prefer to keep, on all platforms, the first occurrence of the static libraries when they are de-duplicated.

The **OLD** behavior for this policy is to keep, on some platforms, the last occurrence of the static libraries when they are de-duplicated. The **NEW** behavior for this policy is to keep the first occurrence of the static libraries when they are de-duplicated, regardless of the platform.

This policy was introduced in CMake version 3.31. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0178

Added in version 3.31.

Test command lines preserve empty arguments.

Empty values in the `TEST_LAUNCHER` and `CROSSCOMPILING_EMULATOR` target properties are now preserved for tests added by the following:

- The `add_test()` command.
- The `ExternalData_Add_Test()` command from the `ExternalData` module.
- The `gtest_add_tests()` or `gtest_discover_tests()` commands from the `GoogleTest` module.

For the `gtest_add_tests()` and `gtest_discover_tests()` commands, empty elements in the values passed after the **EXTRA_ARGS** keyword are also now preserved.

The **OLD** behavior of this policy silently discards empty list items from the `TEST_LAUNCHER` and `CROSSCOMPILING_EMULATOR` target properties in the above-mentioned cases. It also silently discards empty items from the values given after **EXTRA_ARGS** for the `gtest_add_tests()` and `gtest_discover_tests()` commands.

The **NEW** behavior of this policy preserves empty list items in the `TEST_LAUNCHER` and `CROSSCOMPILING_EMULATOR` target properties, and in values given after **EXTRA_ARGS** for `gtest_add_tests()` and `gtest_discover_tests()`.

This policy was introduced in CMake version 3.31. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0177

Added in version 3.31.

`install()` **DESTINATION** paths are normalized.

The `install()` command has a number of different forms, and most of them take a **DESTINATION** keyword, some in more than one place. CMake 3.30 and earlier used the value given after the **DESTINATION** keyword as provided with no transformations. The `install(EXPORT)` form assumes the path contains no `..` or `.` path components when computing a path relative to the **DESTINATION**, and if the project provided a path that violated that assumption, the computed path would be incorrect.

CMake 3.31 normalizes all **DESTINATION** values given in any form of the `install()` command, except for the **INCLUDES DESTINATION** of the `install(TARGETS)` form. The normalization performed is the same as for the `cmake_path()` command (see *Normalization*).

The **OLD** behavior of this policy performs no translation on the **DESTINATION** values of any `install()` command. They are used exactly as provided. If a destination path contains `..` or `.` path components, `install(EXPORT)` will use the same wrong paths as CMake 3.30 and earlier.

The **NEW** behavior will normalize all **DESTINATION** values except for **INCLUDES DESTINATION**. If a destination path contains a generator expression, it will be wrapped in a `$(PATH:CMAKE_PATH,NORMALIZE,...)` generator expression.

This policy was introduced in CMake version 3.31. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake will warn if it detects a path that would be different if normalized, and uses **OLD** behavior. If a destination path contains a generator expression, no such warning will be issued regardless of the value.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0176

Added in version 3.31.

`execute_process()` **ENCODING** is **UTF-8** by default.

The **ENCODING** option is meaningful only on Windows. It specifies the character encoding expected in the process's output on stdout and stderr. In CMake 3.14 and below the default encoding was **NONE**, which corresponds to CMake's internal UTF-8 encoding. In CMake 3.15 through CMake 3.30 the default encoding was accidentally changed to **AUTO**, but the change went unnoticed and was not documented.

CMake 3.31 and above prefer the **ENCODING** default to be **UTF-8**. This policy provides compatibility with projects that may have been relying on the default being **AUTO**.

The **OLD** behavior of this policy is for `execute_process()` to use **AUTO** by default if no **ENCODING** is specified. The **NEW** behavior for this policy is to use **UTF-8** as the default **ENCODING**.

This policy was introduced in CMake version 3.31. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0175

Added in version 3.31.

`add_custom_command()` rejects invalid arguments.

CMake 3.30 and earlier silently ignored unsupported keywords and missing or invalid arguments for the different forms of the `add_custom_command()` command. CMake 3.31 implements more rigorous argument checking and will flag invalid or missing arguments as errors.

The **OLD** behavior of this policy will accept the same invalid keywords or arguments as CMake 3.30 and earlier. The **NEW** behavior will flag the following as errors that previously went unreported:

- The **OUTPUT** form does not accept **PRE_BUILD**, **PRE_LINK**, or **POST_BUILD** keywords.
- When the **APPEND** keyword is given, the **OUTPUT** form also does not accept **BYPRODUCTS**, **COMMAND_EXPAND_LISTS**, **DEPENDS_EXPLICIT_ONLY**, **DEPFILE**, **JOB_POOL**, **JOB_SERVER_AWARE**, **USES_TERMINAL**, or **VERBATIM** keywords.
- The **TARGET** form requires exactly one of **PRE_BUILD**, **PRE_LINK**, or **POST_BUILD** to be given. Previously, if none were given, **POST_BUILD** was assumed, or if multiple keywords were given, the last one was used.
- The **TARGET** form does not accept **DEPENDS**, **DEPENDS_EXPLICIT_ONLY**, **DEPFILE**, **IMPLICIT_DEPENDS**, **MAIN_DEPENDENCY**, **JOB_POOL**, or **JOB_SERVER_AWARE** keywords.
- The **TARGET** form now requires at least one **COMMAND** to be given.
- If a keyword expects a value to be given after it, but no value is provided, that was previously treated as though the keyword was not given at all.

- The **COMMENT** keyword expects exactly one value after it. If multiple values are given, or if the **COMMENT** keyword is given more than once, this is an error.

This policy was introduced in CMake version 3.31. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0174

Added in version 3.31.

`cmake_parse_arguments(PARSE_ARGV)` defines a variable for an empty string after a single-value keyword.

One of the main reasons for using the **PARSE_ARGV** form of the `cmake_parse_arguments()` command is to more robustly handle corner cases related to empty values. The non-**PARSE_ARGV** form doesn't preserve empty arguments, but the **PARSE_ARGV** form does. For each single-value keyword given, a variable should be defined if the keyword is present, even if it is followed by an empty string.

Prior to CMake 3.31, no variable would be defined if the value given after a single-value keyword was an empty string. This meant the code could not detect the difference between the keyword not being given, and it being given but with an empty value, except by iterating over all the arguments and checking if the keyword is present.

For the **OLD** behavior of this policy, `cmake_parse_arguments(PARSE_ARGV)` does not define a variable for a single-value keyword followed by an empty string, or followed by no value at all.

For the **NEW** behavior, `cmake_parse_arguments(PARSE_ARGV)` always defines a variable for each keyword given in the arguments, even a single-value keyword with an empty string as its value or no value at all. With the **NEW** behavior, the code can robustly check if a single-value keyword was given using just `if(DEFINED <prefix>_<keyword>)`.

This policy was introduced in CMake version 3.31. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0173

Added in version 3.31.

The `CMakeFindFrameworks` module is removed.

CMake's framework handling has evolved well beyond what the **CMakeFindFrameworks** module supports. The module lacks any handling of XCFrameworks, it never documented the one command it provides, and `find_library()` provides superior capabilities in all respects.

The **OLD** behavior of this policy is for `CMakeFindFrameworks` to continue to provide the undocumented `cmake_find_frameworks()` command. The **NEW** behavior halts with a fatal error if anything tries to include the module.

This policy was introduced in CMake version 3.31. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0172

Added in version 3.31.

The *CPack* module enables per-machine installation by default in the *CPack WIX Generator*.

The *CPack WIX Generator*'s `CPACK_WIX_INSTALL_SCOPE` option controls the scope of the generated Windows Installer package. When `CPACK_WIX_VERSION` is set to 4 or higher, the default scope is **perMachine**. However, when using WIX 3 the default scope is **NONE**, and CPack does not set any **InstallScope** in the package specification. The resulting installer requires administrative privileges and installs into the system-wide **ProgramFiles** directory, but the start menu entry and uninstaller registration are created only for the current user.

The *CPack* module in CMake 3.30 and older does not specify any `CPACK_WIX_INSTALL_SCOPE` value by default, so CPack uses no installation scope by default with WIX 3. CMake 3.31 and newer instead prefer to set `CPACK_WIX_INSTALL_SCOPE` to **perMachine** by default to make the behavior consistent across all WIX versions. This policy provides compatibility for projects that have not been updated to expect **perMachine** behavior.

The **OLD** behavior for this policy is to not set `CPACK_WIX_INSTALL_SCOPE` by default. The **NEW** behavior for this policy is to set `CPACK_WIX_INSTALL_SCOPE` to **perMachine** by default.

This policy was introduced in CMake version 3.31. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn by default, and uses **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0172` variable to control the warning.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0171

Added in version 3.31.

codegen is a reserved target name.

CMake 3.30 and earlier did not reserve **codegen** as a builtin target name, leaving projects free to create their own target with that name. CMake 3.31 and later prefer to reserve **codegen** as a builtin target name to drive custom commands created with the **CODEGEN** option to `add_custom_command()`. In order to support building the **codegen** target in scripted environments, e.g., `cmake --build . --target codegen`, the **codegen** target needs to be generated even if no custom commands use the **CODEGEN** option. This policy provides compatibility for projects that have not been updated to avoid creating a target named **codegen**.

The **OLD** behavior of this policy allows projects to create a target with the name **codegen**. The **NEW** behavior halts with a fatal error if a target with the name **codegen** is created.

This policy was introduced in CMake version 3.31. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.30**CMP0170**

Added in version 3.30.

When **FETCHCONTENT_FULLY_DISCONNECTED** is set to true, *FetchContent_MakeAvailable()* and *FetchContent_Populate()* enforce the constraint that their source directory must already be populated. The requirement has always been documented, but it was not checked or enforced with CMake 3.29 or older. This sometimes led to hard-to-trace errors when a project expected a dependency to have been populated, but its population was silently skipped.

CMake 3.30 and above prefers to check and enforce the constraint. This policy provides compatibility for situations where the user cannot easily prevent **FETCHCONTENT_FULLY_DISCONNECTED** from being inappropriately set to true.

The **OLD** behavior of this policy allows **FETCHCONTENT_FULLY_DISCONNECTED** to be set to true even if a dependency's source directory has not been populated. The **NEW** behavior halts with a fatal error if **FETCHCONTENT_FULLY_DISCONNECTED** is set to true and a dependency population would be skipped, but that dependency's source directory doesn't exist.

This policy was introduced in CMake version 3.30. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0169

Added in version 3.30.

Calling *FetchContent_Populate()* with a single argument (the name of a declared dependency) is deprecated.

Prior to the introduction of *FetchContent_MakeAvailable()*, projects populated previously declared content (with *FetchContent_Declare()*) using the following pattern:

```
FetchContent_GetProperties(depname)
if(NOT depname_POPULATED)
  FetchContent_Populate(depname)
  add_subdirectory(${depname_SOURCE_DIR} ${depname_BINARY_DIR})
endif()
```

The above pattern does not support a number of features that have been added to *FetchContent* over time. It ignores options like **SYSTEM** and **EXCLUDE_FROM_ALL** which may be given to *FetchContent_Declare()*, but can't be made known to the above project code. It also does not support *dependency providers*. Projects should call *FetchContent_MakeAvailable()* instead of using the above pattern.

CMake 3.30 and above prefers to reject calls to *FetchContent_Populate()* with the name of a declared dependency. This policy provides compatibility for projects that have not been updated to call *FetchContent_MakeAvailable()* instead.

The **OLD** behavior of this policy allows *FetchContent_Populate()* to be called with the name of a declared dependency. The **NEW** behavior halts with a fatal error in such cases.

NOTE:

Calling *FetchContent_Populate()* with the full population details as command arguments rather than just a dependency name remains fully supported. Only the form calling *FetchContent_Populate()* with a single argument (the name of a previously declared dependency) is deprecated with this policy.

This policy was introduced in CMake version 3.30. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0168

Added in version 3.30.

The *FetchContent* module implements steps directly instead of through a sub-build.

CMake 3.29 and below implement *FetchContent* as a separate sub-build. This required configuring that separate project and using a build tool. This approach can be very slow with some generators and operating systems. CMake 3.30 and above prefer to implement the download, update, and patch steps directly as part of the main project.

The **NEW** behavior has the following characteristics:

- No sub-build is used. All operations are implemented directly from the main project's CMake configure step. When running in CMake script mode, no build tool needs to be available.
- Generator expressions and GNU Make variables of the form **\$(SOMEVAR)** are not supported. They should not be used in any argument to *FetchContent_Declare()* or *FetchContent_Populate()*.
- All **LOG_...** and **USES_TERMINAL_...** options, the **QUIET** option, and the *FETCHCONTENT_QUIET* variable are ignored. *FetchContent* output is always part of the main project's configure output. This also means it now respects the message logging level (see *CMAKE_MESSAGE_LOG_LEVEL* and *--log-level*). The default message log level should be comparable to using **QUIET** with the **OLD** policy setting, except that warnings will now be shown.
- The **PREFIX**, **TMP_DIR**, **STAMP_DIR**, **LOG_DIR**, and **DOWNLOAD_DIR** options and their associated directory properties are ignored. The *FetchContent* module controls those locations internally.
- *cmake --fresh* will remove the stamp and script files used for tracking and populating the dependency. This will force the dependency's download, update, and patch steps to be re-executed. The directory used for downloads is not affected by *cmake --fresh*, so any previously downloaded files for the **URL** download method can still be reused.

The **OLD** behavior has the following characteristics:

- A sub-build is always used to implement the download, update, and patch steps. A build tool must be available, even when using *FetchContent_Populate()* in CMake script mode.
- Generator expressions and GNU Make variables of the form **\$(SOMEVAR)** can be used, although such use is almost always inappropriate. They are evaluated in the sub-build, so they do not see any information from the main build.
- All logging, terminal control, and directory options related to the download, update, or patch steps are supported.

- If the **QUIET** option is used, or the `FETCHCONTENT_QUIET` variable is set to true, warnings will not be shown in the output.
- `cmake --fresh` has no effect on the dependency's stamp or script files. Previously executed steps will only re-run if details about the dependency have changed.

There's a reasonably good chance that users can set the `CMAKE_POLICY_DEFAULT_CMP0168` variable to **NEW** to globally switch to the **NEW** behavior while waiting for the project and its dependencies to be updated use the **NEW** policy setting by default. Projects don't typically make use of the features that the **NEW** behavior no longer supports, and even those projects that do will often still work fine when those options are ignored. Before setting this behavior globally, check whether any `FetchContent_Declare()` or `FetchContent_Populate()` calls use the ignored options in a way that would change observable behavior, other than putting temporary or internally-generated files in different locations.

This policy was introduced in CMake version 3.30. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0167

Added in version 3.30.

The `FindBoost` module is removed.

CMake 3.29 and below provide a **FindBoost** module, but it needs constant updates to keep up with upstream Boost releases. Upstream Boost 1.70 and above provide a **BoostConfig.cmake** package configuration file. `find_package(Boost CONFIG)` finds the upstream package directly, without the find module.

CMake 3.30 and above prefer to not provide the **FindBoost** module so that `find_package(Boost)` calls, without the **CONFIG** or **NO_MODULE** options, find the upstream **BoostConfig.cmake** directly. This policy provides compatibility for projects that have not been ported to use the upstream Boost package.

The **OLD** behavior of this policy is for `find_package(Boost)` to load CMake's `FindBoost` module. The **NEW** behavior is for `find_package(Boost)` to search for the upstream **BoostConfig.cmake**.

This policy was introduced in CMake version 3.30. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0166

Added in version 3.30.

`TARGET_PROPERTY` evaluates link properties transitively over private dependencies of static libraries.

In CMake 3.29 and below, the `TARGET_PROPERTY` generator expression evaluates properties `INTERFACE_LINK_OPTIONS`, `INTERFACE_LINK_DIRECTORIES`, and `INTERFACE_LINK_DEPENDS` as if they were *Transitive Compile Properties* rather than *Transitive Link Properties*, even when policy `CMP0099` is set to **NEW**. Private dependencies of static libraries, which appear in their `INTERFACE_LINK_LIBRARIES` guarded by `LINK_ONLY` generator expressions, are not followed. This is

inconsistent with evaluation of the same target properties during buildsystem generation.

CMake 3.30 and above prefer that *TARGET_PROPERTY* evaluates properties *INTERFACE_LINK_OPTIONS*, *INTERFACE_LINK_DIRECTORIES*, and *INTERFACE_LINK_DEPENDS* as *Transitive Link Properties* such that private dependencies of static libraries, which appear in their *INTERFACE_LINK_LIBRARIES* guarded by *LINK_ONLY* generator expressions, are followed. This policy provides compatibility for projects that have not been updated to expect the new behavior.

The **OLD** behavior for this policy is for *TARGET_PROPERTY* to evaluate properties *INTERFACE_LINK_OPTIONS*, *INTERFACE_LINK_DIRECTORIES*, and *INTERFACE_LINK_DEPENDS* as if they were *Transitive Compile Properties* by not following private dependencies of static libraries. The **NEW** behavior for this policy is to evaluate them as *Transitive Link Properties* by following private dependencies of static libraries.

This policy was introduced in CMake version 3.30. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0165

Added in version 3.30.

enable_language() must not be called before *project()*.

In CMake 3.29 and below, if a project called *enable_language()* before the first call to *project()*, the language would be enabled but possibly using unset details that were expected to be set. In CMake 3.30 and above, *enable_language()* prefers to reject this case and stop with a fatal error instead if it detects that *project()* has not yet been called. This policy provides compatibility for projects that happened to work when *enable_language()* was called before *project()* and have not been updated to call these commands in the required order.

The **OLD** behavior for this policy is to allow *enable_language()* to be called before *project()*. The **NEW** behavior for this policy is to fail with a fatal error in such cases.

This policy was introduced in CMake version 3.30. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0164

Added in version 3.30.

add_library() rejects **SHARED** libraries when not supported by the platform.

In CMake 3.29 and below, on platforms that do not support shared libraries (*TARGET_SUPPORTS_SHARED_LIBS* is false), the *add_library()* command automatically converts **SHARED** libraries to **STATIC** libraries to help users build projects on such platforms. However, the semantics of shared and static libraries are different enough that such automatic conversion cannot work in general. Projects using shared libraries need to be ported to such platforms on a case-by-case basis.

In CMake 3.30 and above, `add_library()` prefers to reject creation of shared libraries on platforms that do not support them, and fail with a fatal error message. This policy provides compatibility for projects that happened to work with the automatic conversion to static libraries and have not been updated with an explicit port.

The **OLD** behavior for this policy is to implicitly create a static library with a developer warning. The **NEW** behavior for this policy is to fail.

This policy was introduced in CMake version 3.30. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn about the behavior change, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0163

Added in version 3.30.

The *GENERATED* source file property is now visible in all directories.

In CMake 3.29 and below, the *GENERATED* source file property, like other source file properties, was scoped in every directory separately. Although policy *CMP0118* allowed sources marked **GENERATED** in one directory to be used in other directories without manually marking them as **GENERATED** again, the **GENERATED** property was still not visible to `get_property()` and `get_source_file_property()` calls.

Whether or not a source file is generated is an all-or-nothing global property of the source: a source is either generated or it is not. CMake 3.30 and above prefer to treat the *GENERATED* source file property as globally scoped. Once it is set in one directory, it is immediately visible to `get_property()` and `get_source_file_property()` calls in other directories. This policy provides compatibility for projects that have not been updated for this behavior.

The **OLD** behavior of this policy is for the **GENERATED** source file property to be visible only in the directories in which it is set. The **NEW** behavior of this policy is to allow the **GENERATED** source file property to be visible in all directories once set in any directory. Furthermore, the **NEW** behavior of this policy implies the **NEW** behavior of policy *CMP0118*: the **GENERATED** property may be set only to boolean values, and may not be turned off once turned on.

This policy was introduced in CMake version 3.30. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0162

Added in version 3.30.

Visual Studio Generators add **UseDebugLibraries** indicators by default.

The "Use Debug Libraries" setting in Visual Studio projects indicates what configurations are considered debug configurations. In standalone projects, this may affect MSBuild's default selection of MSVC runtime library, optimization flags, runtime checks, and similar settings. CMake typically generates all those settings explicitly based on the project's specification, so CMake 3.29 and below do not write any

UseDebugLibraries indicators to **.vcxproj** files.

CMake 3.30 and above prefer to write **UseDebugLibraries** indicators because they are useful for reference by both humans and tools, and may also affect the behavior of platform-specific SDKs. The indicator for each configuration of a target is determined as follows:

- If the target compiles sources for a known MSVC runtime library (such as that specified by `MSVC_RUNTIME_LIBRARY`), then **UseDebugLibraries** is **true** for configurations that compile for a "Debug" runtime library, and **false** for others.
- Otherwise, such as in targets created by `add_custom_target()`, **UseDebugLibraries** is **true** for the **Debug** configuration, and **false** for others.

This policy provides compatibility for projects that have not been updated to expect the indicators. The policy setting is recorded by each target as it is created and used to determine the default behavior for that target's **.vcxproj** file.

The **OLD** behavior for this policy is to not generate **UseDebugLibraries** indicators by default. The **NEW** behavior for this policy is to generate **UseDebugLibraries** indicators by default.

If the `CMAKE_VS_USE_DEBUG_LIBRARIES` variable and/or `VS_USE_DEBUG_LIBRARIES` target property is set, it explicitly controls **UseDebugLibraries** generation regardless of this policy.

This policy was introduced in CMake version 3.30. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.29

CMP0161

Added in version 3.29.

The `CPACK_PRODUCTBUILD_DOMAINS` variable defaults to true.

Before CMake 3.29, the `CPACK_PRODUCTBUILD_DOMAINS` variable is unset by default. When using the *CPack productbuild Generator*, this disables the use of the **domains** attribute in the productbuild Distribution XML, and falls back to the **auth** attribute instead. These attributes control where a productbuild package is allowed to be installed. But the **auth** attribute has been deprecated by Apple, so projects should migrate to using **domains** instead.

CMake 3.29 and above prefer to use a default value of true for `CPACK_PRODUCTBUILD_DOMAINS`, which means **domains** will be used by default unless the project explicitly sets `CPACK_PRODUCTBUILD_DOMAINS` to false. This policy provides compatibility with projects that enabled the *CPack productbuild Generator*, but did not explicitly set `CPACK_PRODUCTBUILD_DOMAINS`.

The **OLD** behavior for this policy is to leave `CPACK_PRODUCTBUILD_DOMAINS` unset if it hasn't been set. The **NEW** behavior for this policy is to use a default value of true for `CPACK_PRODUCTBUILD_DOMAINS`.

This policy was introduced in CMake version 3.29. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

Note that a warning will only be emitted if the `CPACK_BINARY_PRODUCTBUILD` variable is set to true

and the project is being built for an Apple platform.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0160

Added in version 3.29.

More read-only target properties now error when trying to set them.

The `set_target_properties()` and `set_property()` commands are intended to error out on all read-only properties. However, CMake 3.28 and below only did this for the following properties:

- `HEADER_SETS`
- `INTERFACE_HEADER_SETS`
- `IMPORTED_GLOBAL`
- `MANUALLY_ADDED_DEPENDENCIES`
- `NAME`
- `TYPE`

This policy enforces the read-only nature of the following target properties:

- `ALIAS_GLOBAL`
- `BINARY_DIR`
- `CXX_MODULE_SETS`
- `IMPORTED`
- `INTERFACE_CXX_MODULE_SETS`
- `LOCATION`
- `LOCATION_<CONFIG>`
- `SOURCE_DIR`

The **OLD** behavior for this policy is to only error out for the properties `MANUALLY_ADDED_DEPENDENCIES`, `NAME`, and `TYPE`. The **NEW** behavior for this policy is to error out on all target properties that are documented as read-only.

This policy was introduced in CMake version 3.29. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0159

Added in version 3.29.

`file(STRINGS)` with **REGEX** updates `CMAKE_MATCH_<n>`.

In CMake 3.28 and below the `file(STRINGS)` command's **REGEX** option does not affect `CMAKE_MATCH_<n>` variables. CMake 3.29 and above prefer to update the `CMAKE_MATCH_<n>`

variables using captures from the last match in the file, similar to the *string(REGEX MATCHALL)* command. This policy provides compatibility for projects that have not been updated to expect the behavior.

The **OLD** behavior for this policy is for *file(STRINGS)* with **REGEX** to not store capture groups in *CMAKE_MATCH_<n>* variables. The **NEW** behavior is to store the capture groups.

This policy was introduced in CMake version 3.29. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0158

Added in version 3.29.

add_test() honors *CMAKE_CROSSCOMPILING_EMULATOR* only when *cross-compiling*.

In CMake 3.28 and below, *add_test()* unconditionally used the *CROSSCOMPILING_EMULATOR* target property (initialized by the *CMAKE_CROSSCOMPILING_EMULATOR* variable) to run test commands naming executable targets. CMake 3.29 and above prefer to use the emulator only when the *CMAKE_CROSSCOMPILING* variable is enabled. The *CMAKE_TEST_LAUNCHER* variable may be used instead when not cross-compiling. This policy provides compatibility for projects that have not been updated.

The **OLD** behavior for this policy is for *add_test()* to use the *CROSSCOMPILING_EMULATOR* target property unconditionally. The **NEW** behavior for this policy is for *add_test()* to use the *CROSSCOMPILING_EMULATOR* target property only when *cross-compiling*.

This policy was introduced in CMake version 3.29. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0157

Added in version 3.29.

Swift compilation mode is selected by an abstraction.

The Swift compiler can compile modules in different modes. The desired build mode depends whether the developer is iterating and wants to incrementally make changes, or if they are building a release for distribution and want more optimizations applied to the resulting binary.

CMake versions 3.26 through 3.28 build Swift binaries with whole-module optimizations enabled when configured in a non-debug build type. For CMake versions earlier than 3.26, the developer needs to specify the necessary flag manually for the *Ninja Generators*, and cannot not specify whole-module optimizations to the *Xcode* generator.

CMake versions 3.29 and above prefer to set the compilation mode using the *Swift_COMPILATION_MODE* target property, which can be initialized by the *CMAKE_Swift_COMPILATION_MODE* variable.

This policy provides compatibility for projects that have not been updated. The policy setting takes effect as of the first `project()` or `enable_language()` command that enables the **Swift** language.

NOTE:

Once the policy has taken effect at the top of a project, that choice must be used throughout the tree. In projects that have nested projects in subdirectories, be sure to convert everything together.

The **OLD** behavior for this policy builds all Swift targets in **wholemodule** mode for non-debug configurations. *Ninja Generators* prepend the `-wmo` flag to the default set of Swift flags. The *Xcode* generator sets the **SWIFT_COMPILATION_MODE** attribute to **wholemodule** in the generated Xcode project file.

The **NEW** behavior for this policy is to apply the compilation mode specified in the `Swift_COMPILATION_MODE` target property, initialized as each target is created by the `CMAKE_Swift_COMPILATION_MODE` variable.

This policy was introduced in CMake version 3.29. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0156

Added in version 3.29.

De-duplicate libraries on link lines based on linker capabilities.

Traditional linkers maintain a set of undefined symbols during linking. The linker processes each file in the order in which it appears on the command line, until the set of undefined symbols becomes empty. An object file is linked into the output object when it is encountered, with its undefined symbols added to the set. Upon encountering an archive file a traditional linker searches the objects contained therein, and processes those that satisfy symbols in the unresolved set.

Handling mutually dependent archives may be awkward when using a traditional linker. Archive files may have to be specified multiple times.

Some linkers (for instance Apple or Windows linkers, as well as **LLVM LLD**) record all symbols found in objects and archives as they iterate over command line arguments. When one of these linkers encounters an undefined symbol that can be resolved by an object file contained in a previously processed archive file, it immediately extracts and links it into the output object.

CMake 3.28 and below may generate link lines that repeat static libraries as a traditional linker would need, even when using a linker that does not need it. They may also de-duplicate shared libraries by keeping their last occurrence, which on Windows platforms can change DLL load order.

CMake 3.29 and above prefer to apply different strategies based on linker capabilities. So, when targeting Apple and Windows platforms, all libraries are de-duplicated. Moreover, on Windows platforms, libraries are de-duplicated by keeping their first occurrence, thus respecting the project-specified order. This policy provides compatibility with projects that have not been updated to expect the latter behavior.

NOTE:

When this policy is set to **NEW**, the policy `CMP0179` controls which occurrence of the static libraries is kept when they are de-duplicated.

The **OLD** behavior for this policy is to always repeat static libraries as if using a traditional linker, and always de-duplicate shared libraries by keeping the last occurrence of each. The **NEW** behavior for this policy is to apply different strategies based on linker capabilities.

This policy was introduced in CMake version 3.29. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.28

CMP0155

Added in version 3.28.

C++ sources in targets with at least C++20 are scanned for imports when supported.

CMake 3.27 and below assume that C++ sources do not **import** modules. CMake 3.28 and above prefer to assume that C++ sources in targets using C++20 or higher might **import** modules, and must be scanned before compiling, unless explicitly disabled. This policy provides compatibility for projects that use C++20 or higher, without modules, that have not been updated to turn off scanning, e.g., via the `CMAKE_CXX_SCAN_FOR_MODULES` variable. See the *cmak e-cxxmodules(7)* manual for more details on C++ module support.

The **OLD** behavior for this policy is to assume that C++ 20 and newer sources do not import modules. The **NEW** behavior for this policy is to assume that C++ 20 and newer files may import modules if the compiler understands how to scan for their dependencies, and need to be scanned.

This policy was introduced in CMake version 3.28. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0154

Added in version 3.28.

Generated files are private by default in targets using *File Sets*.

CMake 3.27 and below assume that any file generated as an output or byproduct of `add_custom_command()` or `add_custom_target()` may be a public header file meant for inclusion by dependents' source files. This requires *Ninja Generators* to add conservative order-only dependencies that prevent a target's source files from compiling before custom commands from the target's dependencies are finished, even if those custom commands only produce sources private to their own target.

File Sets, introduced by CMake 3.23, provide a way to express the visibility of generated header files. CMake 3.28 and above prefer to assume that, in targets using file sets, generated files are private to their own target by default. Generated public headers must be specified as members of a **PUBLIC** (or **INTERFACE**) **FILE_SET**, typically of type **HEADERS**. With this information, *Ninja Generators* may omit the above-mentioned conservative dependencies and produce more efficient build graphs.

Additionally, if the custom command's output is a member of a file set of type **CXX_MODULES**, it will additionally not be required to exist before compiling other sources in the same target. Since these files

should not be included at compile time directly, they may not be implicitly required to exist for other compilation rules.

This policy provides compatibility for projects using file sets in targets with generated header files that have not been updated. Such projects should be updated to express generated public headers in a file set. For example:

```
add_custom_command(
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/foo.h
  ...
)
target_sources(foo
  PUBLIC FILE_SET HEADERS
    BASE_DIRS ${CMAKE_CURRENT_BINARY_DIR}
    FILES     ${CMAKE_CURRENT_BINARY_DIR}/foo.h
)
```

The **OLD** behavior for this policy is to assume generated files are public, even in targets using file sets, and for *Ninja Generators* to produce conservative build graphs. The **NEW** behavior for this policy is to assume generated files are private in targets using file sets, and for *Ninja Generators* to produce more efficient build graphs.

This policy was introduced in CMake version 3.28. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0153

Added in version 3.28.

The *exec_program()* command should not be called.

This command has long been superseded by the *execute_process()* command and has been deprecated since CMake 3.0.

CMake >= 3.28 prefer that this command never be called. The **OLD** behavior for this policy is to allow the command to be called. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** when the command is called.

This policy was introduced in CMake version 3.28. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0152

Added in version 3.28.

file(REAL_PATH) resolves symlinks before collapsing ../ components.

In CMake 3.27 and below, *file(REAL_PATH)* collapsed any ../ components in a path before resolving

symlinks. This produced incorrect results when the `../` collapsed a way a symlink.

The **OLD** behavior for this policy is to collapse `../` components before resolving symlinks. The **NEW** behavior for this policy is to resolve all symlinks before collapsing `../` components.

This policy was introduced in CMake version 3.28. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.27

CMP0151

Added in version 3.27.

AUTOMOC include directory is a system include directory by default.

Headers generated for *AUTOMOC* are placed in target-specific include directories. CMake 3.26 and older added these as normal include directories. CMake 3.27 and newer prefer to add them as system include directories. This policy provides compatibility for projects that have not been updated to expect this.

If the `AUTOGEN_USE_SYSTEM_INCLUDE` target property is set, perhaps via the `CMAKE_AUTOGEN_USE_SYSTEM_INCLUDE` variable, then its value is used regardless of the setting of this policy.

The **OLD** behavior for this policy is to add autogen include directory to the target's include directories. The **NEW** behavior for this policy is to add autogen include directory to the target's system include directories.

This policy was introduced in CMake version 3.27. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0150

Added in version 3.27.

`ExternalProject_Add()` and `FetchContent_Declare()` commands treat relative **GIT_REPOSITORY** paths as being relative to the parent project's remote.

Earlier versions of these commands always treated relative paths in **GIT_REPOSITORY** as local paths, but the base directory it was treated as relative to was both undocumented and unintuitive. The **OLD** behavior for this policy is to interpret relative paths used for **GIT_REPOSITORY** as local paths relative to the following:

- The parent directory of **SOURCE_DIR** for `ExternalProject_Add()`.
- **FETCHCONTENT_BASE_DIR** for `FetchContent_Declare()`.

The **NEW** behavior is to determine the remote from the parent project and interpret the path relative to that remote. The value of `CMAKE_CURRENT_SOURCE_DIR` when `ExternalProject_Add()` or `FetchContent_Declare()` is called determines the parent project. The remote is selected according to the

following (the first match is used):

- If the parent project is checked out on a branch with an upstream remote defined, use that remote.
- If only one remote is defined, use that remote.
- If multiple remotes are defined and one of them is named **origin**, use **origin**'s remote but also issue a warning.

If an appropriate remote cannot be determined from the above, a fatal error will be raised.

This policy was introduced in CMake version 3.27. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns when a relative path is encountered, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0149

Added in version 3.27.

Visual Studio Generators select latest Windows SDK by default.

Visual Studio Generators select a Windows SDK version to put in the **WindowsTargetPlatformVersion** setting in **.vcxproj** files. CMake sets the `CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION` variable to the selected SDK version.

Prior to CMake 3.27, the SDK version was always selected by the value of the `CMAKE_SYSTEM_VERSION` variable. Users or toolchain files could set that variable to one of the exact Windows SDK versions available on the host system. Since `CMAKE_SYSTEM_VERSION` defaults to `CMAKE_HOST_SYSTEM_VERSION`, and it is not guaranteed that a matching Windows SDK version is available, CMake had to fall back to using the latest Windows SDK version if no exact match was available. This approach was problematic:

- The latest Windows SDK might or might not be selected based on whether the host version of Windows happens to match an available SDK version.
- An old Windows SDK version might be selected that has not been updated for newer language standards such as C11.

CMake 3.27 and higher prefer to ignore the exact value of `CMAKE_SYSTEM_VERSION` and by default select the latest SDK version available. An exact SDK version may be specified explicitly using a **version=** field in the `CMAKE_GENERATOR_PLATFORM` variable. See *Visual Studio Platform Selection*.

This policy provides compatibility for projects, toolchain files, and build scripts that have not been ported away from using `CMAKE_SYSTEM_VERSION` to specify an exact SDK version.

NOTE:

This policy must be set before the first `project()` or `enable_language()` command invocation at the top of the project. That is when *Visual Studio Generators* select a Windows SDK.

The **OLD** behavior for this policy is to use the exact value of `CMAKE_SYSTEM_VERSION` if possible. The **NEW** behavior for this policy is to ignore it.

This policy was introduced in CMake version 3.27. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0148

Added in version 3.27.

The *FindPythonInterp* and *FindPythonLibs* modules are removed.

These modules have been deprecated since CMake 3.12. CMake 3.27 and above prefer to not provide the modules. This policy provides compatibility for projects that have not been ported away from them.

Projects using the *FindPythonInterp* and/or *FindPythonLibs* modules should be updated to use one of their replacements:

- *FindPython3*
- *FindPython2*
- *FindPython*

The **OLD** behavior of this policy is for **find_package(PythonInterp)** and **find_package(PythonLibs)** to load the deprecated modules. The **NEW** behavior is for uses of the modules to fail as if they do not exist.

This policy was introduced in CMake version 3.27. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0147

Added in version 3.27.

Visual Studio Generators build custom commands in parallel.

Visual Studio 15.8 (2017) and newer support building custom commands in parallel. CMake 3.27 and above prefer to enable this behavior by adding a **BuildInParallel** setting to custom commands in **.vcxproj** files. This policy provides compatibility for projects that have not been updated to expect this, e.g., because their custom commands were accidentally relying on serial execution by MSBuild. To control this behavior in a more precise way, refer to *VS_CUSTOM_COMMAND_DISABLE_PARALLEL_BUILD*.

The **OLD** behavior for this policy is to not add **BuildInParallel**. The **NEW** behavior for this policy is to add **BuildInParallel** for VS 15.8 and newer.

This policy was introduced in CMake version 3.27. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0146

Added in version 3.27.

The *FindCUDA* module is removed.

The *FindCUDA* module has been deprecated since CMake 3.10. CMake 3.27 and above prefer to not provide the module. This policy provides compatibility for projects that have not been ported away from it.

Projects using the *FindCUDA* module should be updated to use CMake's first-class **CUDA** language support. List **CUDA** among the languages named in the top-level call to the *project()* command, or call the *enable_language()* command with **CUDA**. Then one can add CUDA (**.cu**) sources directly to targets, similar to other languages.

The **OLD** behavior of this policy is for **find_package(CUDA)** to load the deprecated module. The **NEW** behavior is for uses of the module to fail as if it does not exist.

This policy was introduced in CMake version 3.27. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0145

Added in version 3.27.

The *Dart* and *FindDart* modules are removed.

These modules were added very early in CMake's development to support driving tests with a "DART" tool, but DART has not been distributed or maintained for many years. Projects would **include(Dart)** to use it, and the **Dart** module would run **find_package(Dart)** internally. Since *ctest(1)* was created, the **Dart** module has just been a compatibility shim that finds **Dart** to support some legacy functionality and then forwards to the *CTest* module.

CMake 3.27 and above prefer to not provide the *Dart* or *FindDart* modules. This policy provides compatibility for projects that have not been ported away from them. Projects using the **Dart** module should be updated to use the *CTest* module directly.

The **OLD** behavior of this policy is for **include(Dart)** and **find_package(Dart)** to load the deprecated modules. The **NEW** behavior is for uses of the modules to fail as if they do not exist.

This policy was introduced in CMake version 3.27. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0144

Added in version 3.27.

find_package() uses upper-case **<PACKAGENAME>_ROOT** variables.

In CMake 3.27 and above the *find_package(<PackageName>)* command now searches prefixes specified by the upper-case **<PACKAGENAME>_ROOT** CMake variable and the **<PACKAGENAME>_ROOT** environment variable in addition to the case-preserved **<PackageName>_ROOT** and **<PackageName>_ROOT** variables used since policy *CMP0074*. This policy provides compatibility with projects that have not been

updated to avoid using `<PACKAGENAME>_ROOT` variables for other purposes.

The **OLD** behavior for this policy is to ignore `<PACKAGENAME>_ROOT` variables if the original `<PackageName>` has lower-case characters. The **NEW** behavior for this policy is to use `<PACKAGE-NAME>_ROOT` variables.

This policy was introduced in CMake version 3.27. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.26

CMP0143

Added in version 3.26.

`USE_FOLDERS` global property is treated as **ON** by default.

When using CMake 3.25 or earlier, `USE_FOLDERS` is treated as **OFF** by default unless projects enable the feature. For example:

```
cmake_minimum_required(VERSION 3.25)
project(foofoo LANGUAGES CXX)
set_property(GLOBAL PROPERTY USE_FOLDERS ON)
```

CMake 3.26 and later prefer to enable the feature by default.

Note that it is the policy setting at the **end** of the top level `CMakeLists.txt` file that matters. The policy setting applies globally to the whole project.

This policy provides compatibility with projects that have not been updated to expect enabling of folders. Enabling folders causes projects to appear differently in IDEs.

This policy was introduced in CMake version 3.26. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.25

CMP0142

Added in version 3.25.

The `Xcode` generator does not append per-config suffixes to library search paths.

In CMake 3.24 and below, the `Xcode` generator preceded each entry of a library search path with a copy of itself appended with `$(CONFIGURATION)$(EFFECTIVE_PLATFORM_NAME)`. This was left from very early versions of CMake in which per-config directories were not well modeled. Such paths often do not exist, resulting in warnings from the toolchain. CMake 3.25 and above prefer to not add such library search paths. This policy provides compatibility for projects that may have been accidentally relying on the old behavior.

The **OLD** behavior for this policy is to append `$(CONFIGURATION)$EFFECTIVE_PLATFORM_NAME)` to all library search paths. The **NEW** behavior is to not modify library search paths.

This policy was introduced in CMake version 3.25. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0141

Added in version 3.25.

MSVC debug information format flags are selected by an abstraction.

Compilers targeting the MSVC ABI have flags to select the debug information format. Debug information format selection typically varies with build configuration.

In CMake 3.24 and below, debug information format flags are added to the default `CMAKE_<LANG>_FLAGS_<CONFIG>` cache entries by CMake automatically. This allows users to edit their cache entries to adjust the flags. However, the presence of such default flags is problematic for projects that want to choose a different debug information format programmatically. In particular, it requires string editing of the `CMAKE_<LANG>_FLAGS_<CONFIG>` variables with knowledge of the CMake builtin defaults so they can be replaced.

CMake 3.25 and above prefer to leave the debug information format flags out of the default `CMAKE_<LANG>_FLAGS_<CONFIG>` values and instead offer a first-class abstraction. The `CMAKE_MSVC_DEBUG_INFORMATION_FORMAT` variable and `MSVC_DEBUG_INFORMATION_FORMAT` target property may be set to select the MSVC debug information format. If they are not set, CMake enables debug information in debug configurations using the default value `$$<CONFIG:Debug,RelWithDebInfo>:ProgramDatabase>`, if supported by the compiler, and otherwise `$$<CONFIG:Debug,RelWithDebInfo>:Embedded>`.

This policy provides compatibility with projects that have not been updated to be aware of the abstraction. The policy setting takes effect as of the first `project()` or `enable_language()` command that enables a language whose compiler targets the MSVC ABI.

NOTE:

Once the policy has taken effect at the top of a project, that choice will be used throughout the tree. In projects that have nested projects in subdirectories, be sure to confirm if everything is working with the selected policy behavior.

The **OLD** behavior for this policy is to place MSVC debug information format flags in the default `CMAKE_<LANG>_FLAGS_<CONFIG>` cache entries and ignore the `CMAKE_MSVC_DEBUG_INFORMATION_FORMAT` abstraction. The **NEW** behavior for this policy is to *not* place MSVC debug information format flags in the default cache entries and use the abstraction instead.

This policy was introduced in CMake version 3.25. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0140

Added in version 3.25.

The `return()` command checks its parameters.

The **OLD** behavior for this policy is to ignore any parameters given to the command. The **NEW** behavior is to check the validity of the parameters.

This policy was introduced in CMake version 3.25. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.24**CMP0139**

Added in version 3.24.

The `if()` command supports path comparisons using **PATH_EQUAL** operator.

The **OLD** behavior for this policy is to ignore the **PATH_EQUAL** operator. The **NEW** behavior is to interpret the **PATH_EQUAL** operator.

This policy was introduced in CMake version 3.24. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0138

Added in version 3.24.

`CheckIPOSupported` uses flags from calling project.

The `CheckIPOSupported` module `check_ipo_supported()` command compiles a test project to determine whether the toolchain supports `INTERPROCEDURAL_OPTIMIZATION`. CMake 3.23 and below run the check with the default values of the `CMAKE_<LANG>_FLAGS` and `CMAKE_<LANG>_FLAGS_<CONFIG>` variables for the current environment and toolchain settings. However, some projects may modify these flag variables to add flags that affect availability of the toolchain's IPO features. CMake 3.24 and above prefer to honor the calling project's values for these variables. This policy provides compatibility for projects that have not been updated to expect this behavior.

The **OLD** behavior for this policy is to ignore the calling project's values of `CMAKE_<LANG>_FLAGS` and `CMAKE_<LANG>_FLAGS_<CONFIG>`. The **NEW** behavior for this policy is to use the values of those variables as compiler flags in the test project.

This policy was introduced in CMake version 3.24. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of

CMake.

CMP0137

Added in version 3.24.

try_compile() passes platform variables in project mode.

The *try_compile()* command *source file* signature propagates CMake variables containing platform settings, and those specified by the `CMAKE_TRY_COMPILE_PLATFORM_VARIABLES` variable, into the generated test project. This helps the test project drive the toolchain the same way the calling project will. In CMake 3.23 and below, the *whole-project* signature does not propagate platform variables automatically. CMake 3.24 and above prefer to propagate platform variables in the *whole-project* signature. This policy provides compatibility with projects that have not been updated to expect the behavior.

The **OLD** behavior for this policy is to not pass any additional variables to the *whole-project* signature. The **NEW** behavior for this policy is to pass the same variables that the *source file* signature does.

Regardless of the policy setting, the `CMAKE_TRY_COMPILE_NO_PLATFORM_VARIABLES` variable may be set to suppress passing the platform variables through either signature.

This policy was introduced in CMake version 3.24. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0136

Added in version 3.24.

Watcom runtime library flags are selected by an abstraction.

Compilers targeting the Watcom ABI have flags to select the Watcom runtime library.

In CMake 3.23 and below, Watcom runtime library selection flags are added to the default `CMAKE_<LANG>_FLAGS_<CONFIG>` cache entries by CMake automatically. This allows users to edit their cache entries to adjust the flags. However, the presence of such default flags is problematic for projects that want to choose a different runtime library programmatically. In particular, it requires string editing of the `CMAKE_<LANG>_FLAGS_<CONFIG>` variables with knowledge of the CMake builtin defaults so they can be replaced.

CMake 3.24 and above prefer to leave the Watcom runtime library selection flags out of the default `CMAKE_<LANG>_FLAGS_<CONFIG>` values and instead offer a first-class abstraction. The `CMAKE_WATCOM_RUNTIME_LIBRARY` variable and `WATCOM_RUNTIME_LIBRARY` target property may be set to select the Watcom runtime library. If they are not set then CMake uses the default value **MultiThreadedDLL** on Windows and **SingleThreaded** on other platforms, which is equivalent to the original flags.

This policy provides compatibility with projects that have not been updated to be aware of the abstraction. The policy setting takes effect as of the first *project()* or *enable_language()* command that enables a language whose compiler targets the Watcom ABI.

NOTE:

Once the policy has taken effect at the top of a project, that choice must be used throughout the tree. In

projects that have nested projects in subdirectories, be sure to convert everything together.

The **OLD** behavior for this policy is to place Watcom runtime library flags in the default `CMAKE_<LANG>_FLAGS_<CONFIG>` cache entries and ignore the `CMAKE_WATCOM_RUNTIME_LIBRARY` abstraction. The **NEW** behavior for this policy is to *not* place Watcom runtime library flags in the default cache entries and use the abstraction instead.

This policy was introduced in CMake version 3.24. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0135

Added in version 3.24.

When using the **URL** download method with the `ExternalProject_Add()` or `FetchContent_Declare()` commands, CMake 3.23 and below sets the timestamps of the extracted contents to the same as the timestamps in the archive. When the **URL** changes, the new archive is downloaded and extracted, but the timestamps of the extracted contents might not be newer than the previous contents. Anything that depends on the extracted contents might not be rebuilt, even though the contents may change.

CMake 3.24 and above prefers to set the timestamps of all extracted contents to the time of the extraction. This ensures that anything that depends on the extracted contents will be rebuilt whenever the **URL** changes.

The **DOWNLOAD_EXTRACT_TIMESTAMP** option to the `ExternalProject_Add()` and `FetchContent_Declare()` commands can be used to explicitly specify how timestamps should be handled. When **DOWNLOAD_EXTRACT_TIMESTAMP** is not given, this policy controls the default behavior. The **OLD** behavior for this policy is to restore the timestamps from the archive. The **NEW** behavior sets the timestamps of extracted contents to the time of extraction.

This policy was introduced in CMake version 3.24. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0134

Added in version 3.24.

The default registry view is **TARGET** for the `find_file()`, `find_path()`, `find_library()`, and `find_package()` commands and **BOTH** for the `find_program()` command.

The default registry views in CMake 3.23 and below are selected using the following rules:

- if `CMAKE_SIZEOF_VOID_P` has value **8**:
 - Use view **64** for all `find_*` commands except `find_program()` command.
 - Use view **64_32** for `find_program()` command.

- if `CMAKE_SIZEOF_VOID_P` has value **4** or is undefined:
 - Use view **32** for all `find_*` commands except `find_program()` command.
 - Use view **32_64** for `find_program()` command.

The **OLD** behavior for this policy is to use registry views **64** and **64_32** or **32_64** and **32** as default, depending of `CMAKE_SIZEOF_VOID_P` variable value. The **NEW** behavior for this policy is to use registry views **TARGET** and **BOTH** as default.

This policy was introduced in CMake version 3.24. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0133

Added in version 3.24.

The `CPack` module disables SLA by default in the `CPack DragNDrop Generator`.

The `CPack DragNDrop Generator` in CMake 3.22 and below attach a Software License Agreement (SLA) to `.dmg` files using the file specified by `CPACK_RESOURCE_FILE_LICENSE`, if set to a non-default value. macOS 12.0 deprecated the tools used to do this, so CMake 3.23 added the `CPACK_DMG_SLA_USE_RESOURCE_FILE_LICENSE` option to control the behavior. CMake 3.23 enables that option by default for compatibility with older versions. CMake 3.24 and above prefer to *not* enable the `CPACK_DMG_SLA_USE_RESOURCE_FILE_LICENSE` option by default. This policy provides compatibility with projects that have not been updated to account for the lack of a SLA in their `.dmg` packages.

The **OLD** behavior for this policy is to enable `CPACK_DMG_SLA_USE_RESOURCE_FILE_LICENSE` by default. The **NEW** behavior for this policy is to not enable it by default.

This policy was introduced in CMake version 3.24. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn by default, and uses **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0133` variable to control the warning.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0132

Added in version 3.24.

Do not set compiler environment variables on first run.

Apart from when using the Xcode generator and some Visual Studio generators, CMake 3.23 and below will set environment variables like `CC`, `CXX`, etc. when the corresponding language is enabled. This only occurs on the very first time CMake is run in a build directory, and the environment variables are only defined at configure time, not build time. On subsequent CMake runs, these environment variables are not set, opening up the opportunity for different behavior between the first and subsequent CMake runs. CMake 3.24 and above prefer to not set these environment variables when a language is enabled, even on the first run in a build directory.

The **OLD** behavior for this policy sets the relevant environment variable on the first run when a language is enabled. The **NEW** behavior for this policy does not set any such environment variables.

This policy was introduced in CMake version 3.24. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0131

Added in version 3.24.

`LINK_LIBRARIES` supports the `$<LINK_ONLY:...>` generator expression.

CMake 3.23 and below documented the `$<LINK_ONLY:...>` generator expression only for use in `INTERFACE_LINK_LIBRARIES`. When used in `LINK_LIBRARIES`, the content guarded inside `$<LINK_ONLY:...>` was always used, even when collecting non-linking usage requirements such as `INTERFACE_COMPILE_DEFINITIONS`.

CMake 3.24 and above prefer to support `$<LINK_ONLY:...>`, when used in `LINK_LIBRARIES`, by using the guarded content only for link dependencies and not other usage requirements. This policy provides compatibility for projects that have not been updated to account for this change.

The **OLD** behavior for this policy is to use `LINK_LIBRARIES` content guarded by `$<LINK_ONLY:...>` even for non-linking usage requirements. The **NEW** behavior for this policy is to use the guarded content only for link dependencies.

This policy was introduced in CMake version 3.24. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0130

Added in version 3.24.

`while()` diagnoses condition evaluation errors.

CMake 3.23 and below accidentally tolerated errors encountered while evaluating the condition passed to the `while()` command (but not the `if()` command). For example, the code

```
set(paren "(")
while(${paren})
endwhile()
```

creates an unbalanced parenthesis during condition evaluation.

CMake 3.24 and above prefer to diagnose such errors. This policy provides compatibility for projects that have not been updated to fix their condition errors.

The **OLD** behavior for this policy is to ignore errors in `while()` conditions. The **NEW** behavior for this policy is to diagnose errors in `while()` conditions.

This policy was introduced in CMake version 3.24. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.23

CMP0129

Added in version 3.23.

Compiler id for MCST LCC compilers is now **LCC**, not **GNU**.

CMake 3.23 and above recognize MCST LCC compiler as a different from **GNU**, with its own command line and set of capabilities. CMake now prefers to present this to projects by setting the `CMAKE_<LANG>_COMPILER_ID` variable to **LCC** instead of **GNU**. However, existing projects may assume the compiler id for LCC is **GNU** as it was in CMake versions prior to 3.23. Therefore this policy determines for MCST LCC compiler which compiler id to report in the `CMAKE_<LANG>_COMPILER_ID` variable after language `<LANG>` is enabled by the `project()` or `enable_language()` command. The policy must be set prior to the invocation of either command.

The **OLD** behavior for this policy is to use compiler id **GNU** (and set `CMAKE_<LANG>_COMPILER_VERSION` to the supported GNU compiler version.) **NEW** behavior for this policy is to use compiler id **LCC**, and set `CMAKE_<LANG>_SIMULATE_ID` to **GNU**, and `CMAKE_<LANG>_SIMULATE_VERSION` to the supported GNU compiler version.

This policy was introduced in CMake version 3.23. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn by default, and uses **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0129` variable to control the warning.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.22

CMP0128

Added in version 3.22.

When this policy is set to **NEW**:

- `<LANG>_EXTENSIONS` is initialized to `CMAKE_<LANG>_EXTENSIONS` if set, otherwise falling back to `CMAKE_<LANG>_EXTENSIONS_DEFAULT`.
- Extensions are correctly enabled/disabled if `<LANG>_STANDARD` is unset or satisfied by the default.
- Standard mode-affecting flags aren't added unless necessary to achieve the specified mode.

The **OLD** behavior:

- Initializes `<LANG>_EXTENSIONS` to `CMAKE_<LANG>_EXTENSIONS` if set, otherwise falling back to **ON**.
- Always adds a flag if `<LANG>_STANDARD` is set and `<LANG>_STANDARD_REQUIRED` is **OFF**.

- If `<LANG>_STANDARD` is unset:
 - Doesn't disable extensions even if `<LANG>_EXTENSIONS` is **OFF**.
 - Fails to enable extensions if `<LANG>_EXTENSIONS` is **ON** except for the **IAR** compiler.

Code may need to be updated for the **NEW** behavior in the following cases:

- If a standard mode flag previously overridden by CMake's and not used during compiler detection now takes effect due to CMake no longer adding one as the default detected is appropriate.

Such code should be converted to either:

- Use `<LANG>_STANDARD` and `<LANG>_EXTENSIONS` instead of manually adding flags.
- Or ensure the manually-specified flags are used during compiler detection.
- If extensions were disabled without `<LANG>_STANDARD` being set CMake previously wouldn't actually disable extensions.

Such code should be updated to not disable extensions if they are required.

- If extensions were enabled/disabled when `<LANG>_STANDARD` was satisfied by the compiler's default CMake previously wouldn't actually enable/disable extensions.

Such code should be updated to set the correct extensions mode.

If compiler flags affecting the standard mode are used during compiler detection (for example in a *tool-chain file* using `CMAKE_<LANG>_FLAGS_INIT`) then they will affect the detected default *standard* and *extensions*.

This policy was introduced in CMake version 3.22. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn by default, and uses **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0128` variable to control the warning.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0127

Added in version 3.22.

`cmake_dependent_option()` supports full *Condition Syntax*.

The `<depends>` parameter accepts a *semicolon-separated list* of conditions. CMake 3.21 and lower evaluates each **condition** as `if(${condition})`, which does not properly handle conditions with nested paren groups. CMake 3.22 and above instead prefer to evaluate each **condition** as `if(<condition>)`, where `<condition>` is re-parsed as if literally written in a call to `if()`. This allows expressions like:

```
"A AND (B OR C)"
```

but requires expressions like:

```
"FOO MATCHES (UPPER|lower)"
```

to be re-written as:

"FOO MATCHES \"(UPPER|lower)\""

Policy **CMP0127** provides compatibility for projects that have not been updated to expect the new behavior.

This policy was introduced in CMake version 3.22. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.21

CMP0126

Added in version 3.21.

When this policy is set to **NEW**, the `set(CACHE)` command does not remove any normal variable of the same name from the current scope. The **OLD** behavior removes any normal variable of the same name from the current scope in the following situations:

- No cache variable of that name existed previously.
- A cache variable of that name existed previously, but it had no type. This can occur when the variable was set on the command line using a form like `cmake -DMYVAR=blah` instead of `cmake -DMYVAR:STRING=blah`.
- The **FORCE** or **INTERNAL** keywords were used when setting the cache variable.

Note that the **NEW** behavior has an important difference to the similar **NEW** behavior of policy **CMP0077**. The `set(CACHE)` command always sets the cache variable if it did not exist previously, regardless of the **CMP0126** policy setting. The `option()` command will *not* set the cache variable if a non-cache variable of the same name already exists and **CMP0077** is set to **NEW**.

This policy was introduced in CMake version 3.21. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn by default, and uses **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0126` variable to control the warning.

The `CMAKE_POLICY_DEFAULT_CMP0126` variable may be used to set the policy for a third-party project in a subdirectory without modifying it.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0125

Added in version 3.21.

The `find_file()`, `find_path()`, `find_library()` and `find_program()` commands cache their result in the variable specified by their first argument. Prior to CMake 3.21, if a cache variable of that name already existed before the call but the cache variable had no type, any non-cache variable of the same name would be discarded and the cache variable was always used (see also **CMP0126** for a different but similar behavior). This contradicts the convention that a non-cache variable should take precedence over a cache variable of the same name. Such a situation can arise if a user sets a cache variable on the command line without specifying a type, such as `cmake -DMYVAR=blah ...` instead of `cmake -DMYVAR:FILEPATH=blah`.

Related to the above, if a cache variable of the specified name already exists and it *does* have a type, the various **find_...()** commands would return that value unchanged. In particular, if it contained a relative path, it would not be converted to an absolute path in this situation.

When policy **CMP0125** is set to **OLD** or is unset, the behavior is as described above. When it is set to **NEW**, the behavior is as follows:

- If a non-cache variable of the specified name exists when the **find_...()** command is called, its value will be used regardless of whether a cache variable of the same name already exists or not. A cache variable will not be created in this case if no such cache variable existed before. If a cache variable of the specified name did already exist, the cache will be updated to match the non-cache variable.
- The various **find_...()** commands will always provide an absolute path in the result variable, except where a relative path provided by a cache or non-cache variable cannot be resolved to an existing path.

This policy was introduced in CMake version 3.21. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0124

Added in version 3.21.

`foreach()` loop variables are only available in the loop scope.

CMake 3.20 and below always leave the loop variable set at the end of the loop, either to the value it had before the loop, if any, or to the empty string. CMake 3.21 and above prefer to leave the loop variable in the state it had before the loop started, either set or unset. This policy provides compatibility for projects that expect the loop variable to always be left set.

The **OLD** behavior for this policy is to set the loop variable at the end of the loop, either to its original value, or to an empty value. The **NEW** behavior for this policy is to restore the loop variable to the state it had before the loop started, either set or unset.

For example:

```
set(items a b c)

set(var1 "value")
unset(var2)

foreach(var1 IN LISTS items)
endforeach()

foreach(var2 IN LISTS items)
endforeach()

if(DEFINED var1)
  message("var1: ${var1}")
endif()
if(DEFINED var2)
  message("var2: ${var2}")
endif()
```

Under the **OLD** behavior, this code prints **var1: value** and **var2:**. Under the **NEW** behavior, this code prints only **var1: value**.

This policy was introduced in CMake version 3.21. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0123

Added in version 3.21.

ARMClang cpu/arch compile and link flags must be set explicitly.

CMake 3.20 and lower automatically maps the `CMAKE_SYSTEM_PROCESSOR` variable and an undocumented `CMAKE_SYSTEM_ARCH` to compile and link options for **ARMClang**. For example, the `-mcpu=cortex-m33` flag is added when `CMAKE_SYSTEM_PROCESSOR` equals `cortex-m33`. CMake requires projects to set either variable or it raises a fatal error. However, the project may need to additionally specify CPU features using e.g. `-mcpu=cortex-m33+nodsp`, conflicting with the `-mcpu=cortex-m33` added by CMake. This results in either link errors or unusable binaries.

CMake 3.21 and above prefer instead to not add any cpu/arch compile and link flags automatically. Instead, projects must specify them explicitly. This policy provides compatibility for projects that have not been updated.

The **OLD** behavior of this policy requires projects that use **ARMClang** to set either `CMAKE_SYSTEM_PROCESSOR` or `CMAKE_SYSTEM_ARCH` and it automatically adds a compile option `-mcpu=` or `-march=` and a link option `--cpu=` based on those variables. The **NEW** behavior does not add compile or link options, and projects are responsible for setting correct options.

This policy was introduced in CMake version 3.21. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0122

Added in version 3.21.

UseSWIG use library name conventions for **CSharp** language.

Starting with CMake 3.21, **UseSWIG** generates now a library using default naming conventions. This policy provides compatibility with projects that expect the legacy behavior.

This policy was introduced in CMake version 3.21. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0121

Added in version 3.21.

The *list()* command now detects invalid indices.

Prior to CMake version 3.21, the *list()* command's **GET**, **INSERT**, **SUBLIST**, and **REMOVE_AT** sub-commands did not detect invalid index arguments.

The **OLD** behavior of this policy is for invalid indices to be treated as their integer value (if any) at the start of the string. For example, **2good4you** is a **2** and **not_an_integer** is a **0**. The **NEW** behavior is for invalid indices to trigger an error.

This policy was introduced in CMake version 3.21. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.20**CMP0120**

Added in version 3.20.

The *WriteCompilerDetectionHeader* module is removed.

CMake versions 3.1 through 3.19 provide this module to generate a C++ compatibility layer by reusing information from CMake's table of preprocessor checks for *cmake-compile-features(7)*. However:

- Those granular features have been superseded by meta-features for *Requiring Language Standards* such as **cxx_std_11**. Therefore no new granular feature checks will be added and projects will need to use other means to conditionally use new C++ features.
- The module exposes some of CMake's implementation details directly to C++ translation units.
- The module's approach effectively provides a header file with CMake, thus tying the version of the header to the version of CMake. Many projects found that the *WriteCompilerDetectionHeader* was best used by manually generating its header locally with a recent version of CMake and then bundling it with the project source so that it could be used with older CMake versions.

For reasons including the above, CMake 3.20 and above prefer to not provide the *WriteCompilerDetectionHeader* module. This policy provides compatibility for projects that have not been ported away from it. Projects using the module should be updated to stop using it. Alternatives include:

- Bundle a copy of the generated header in the project's source.
- Use a third-party alternative, such as the CC0-licensed *Hedley*.
- Drop support for compilers too old to provide the features natively.

The **OLD** behavior of this policy is for inclusion of the deprecated *WriteCompilerDetectionHeader* module to work. The **NEW** behavior is for inclusion of the module to fail as if it does not exist.

This policy was introduced in CMake version 3.20. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of

CMake.

CMP0119

Added in version 3.20.

LANGUAGE source file property explicitly compiles as specified language.

The *LANGUAGE* source file property is documented to mean that the source file is written in the specified language. In CMake 3.19 and below, setting this property causes CMake to compile the source file using the compiler for the specified language. However, it only passes an explicit flag to tell the compiler to treat the source as the specified language for MSVC-like, XL, and Embarcadero compilers for the **CXX** language. CMake 3.20 and above prefer to also explicitly tell the compiler to use the specified language using a flag such as `-x c` on all compilers for which such flags are known.

This policy provides compatibility for projects that have not been updated to expect this behavior. For example, some projects were setting the *LANGUAGE* property to **C** on assembly-language *.S* source files in order to compile them using the C compiler. Such projects should be updated to use **enable_language(ASM)**, for which CMake will often choose the C compiler as the assembler on relevant platforms anyway.

The **OLD** behavior for this policy is to interpret the *LANGUAGE* **<LANG>** property using its undocumented meaning to "use the **<LANG>** compiler". The **NEW** behavior for this policy is to interpret the *LANGUAGE* **<LANG>** property using its documented meaning to "compile as a **<LANG>** source".

This policy was introduced in CMake version 3.20. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0118

Added in version 3.20.

GENERATED sources may be used across directories without manual marking.

In CMake 3.19 and below, the *GENERATED* source file property, like other source file properties, was scoped in every directory separately. If a source file was generated in one directory, projects had to manually set the **GENERATED** property in another directory in order to use the file.

Whether or not a source file is generated is an all-or-nothing global property of the source: a source is either generated or it is not. CMake 3.20 and above prefer to allow source files generated in one directory to be used in other directories without manually marking them as **GENERATED**. Additionally, the **GENERATED** property may now be set only to boolean values, and may not be turned off once turned on. This policy provides compatibility for projects that have not been updated for this behavior.

The **OLD** behavior of this policy is to allow generated files to be used only in directories in which their **GENERATED** property has been turned on. The **NEW** behavior of this policy is to allow generated files to be used in other directories without explicitly turning on the **GENERATED** property for those directories.

Added in version 3.30: Policy *CMP0163* additionally makes the *GENERATED* source file property visible to *get_property()* and *get_source_file_property()* calls in other directories.

This policy was introduced in CMake version 3.20. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns about setting the **GENERATED** property to a non-boolean value, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0117

Added in version 3.20.

MSVC RTTI flag **/GR** is not added to `CMAKE_CXX_FLAGS` by default.

When using MSVC-like compilers in CMake 3.19 and below, the RTTI flag **/GR** is added to `CMAKE_CXX_FLAGS` by default. This behavior is left from support for MSVC versions from Visual Studio 2003 and below that did not enable RTTI by default. It is no longer necessary. Furthermore, it is problematic for projects that want to change to **/GR** programmatically. In particular, it requires string editing of the `CMAKE_CXX_FLAGS` variable with knowledge of the CMake builtin default so it can be replaced.

CMake 3.20 and above prefer to leave out **/GR** from the value of `CMAKE_CXX_FLAGS` by default.

This policy provides compatibility with projects that have not been updated to expect the lack of the **/GR** flag. The policy setting takes effect as of the first `project()` or `enable_language()` command that initializes `CMAKE_CXX_FLAGS`.

NOTE:

Once the policy has taken effect at the top of a project for a given language, that choice must be used throughout the tree for that language. In projects that have nested projects in subdirectories, be sure to convert everything together.

The **OLD** behavior for this policy is to place the MSVC **/GR** flag in the default `CMAKE_CXX_FLAGS` cache entry. The **NEW** behavior for this policy is to *not* place the MSVC **/GR** flag in the default cache entry.

This policy was introduced in CMake version 3.20. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0116

Added in version 3.20.

Ninja generators transform **DEPFILE**s from `add_custom_command()`.

In CMake 3.19 and below, files given to the **DEPFILE** argument of `add_custom_command()` were passed directly to Ninja's **depfile** variable without any path resolution. This meant that if `add_custom_command()` was called from a subdirectory (created by `add_subdirectory()`), the **DEPFILE** argument would have to be either an absolute path or a path relative to `CMAKE_BINARY_DIR`, rather than `CMAKE_CURRENT_BINARY_DIR`. In addition, no transformation was done on the file listed in **DEPFILE**, which meant that the paths within the **DEPFILE** had the same restrictions.

Starting with CMake 3.20, the **DEPFILE** argument is relative to `CMAKE_CURRENT_BINARY_DIR`.

(unless it is absolute), and the paths in the **DEPFILE** are also relative to **CMAKE_CURRENT_BINARY_DIR**. CMake automatically transforms the paths in the **DEPFILE** (unless they are absolute) after the custom command is run. The file listed in **DEPFILE** is not modified in any way. Instead, CMake writes the transformation to its own internal file, and passes this internal file to Ninja's **dep-file** variable. This transformation happens regardless of whether or not **DEPFILE** is relative, and regardless of whether or not `add_custom_command()` is called from a subdirectory.

The **OLD** behavior for this policy is to pass the **DEPFILE** to Ninja unaltered. The **NEW** behavior for this policy is to transform the **DEPFILE** after running the custom command. The status of **CMP0116** is recorded at the time of the custom command's creation, and you can have custom commands in the same directory with different values for **CMP0116** by setting the policy before each custom command.

This policy was introduced in CMake version 3.20. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn by default (unless **DEPFILE** is used in a subdirectory), and uses **OLD** behavior.

See documentation of the **CMAKE_POLICY_WARNING_CMP0116** variable to control the warning.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0115

Added in version 3.20.

Source file extensions must be explicit.

In CMake 3.19 and below, if a source file could not be found by the name specified, it would append a list of known extensions to the name to see if the file with the extension could be found. For example, this would allow the user to run:

```
add_executable(exe main)
```

and put **main.c** in the executable without specifying the extension.

Starting in CMake 3.20, CMake prefers all source files to have their extensions explicitly listed:

```
add_executable(exe main.c)
```

The **OLD** behavior for this policy is to implicitly append known extensions to source files if they can't be found. The **NEW** behavior of this policy is to not append known extensions and require them to be explicit.

This policy was introduced in CMake version 3.20. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.19

CMP0114

Added in version 3.19.

ExternalProject step targets fully adopt their steps.

The *ExternalProject_Add()* **STEP_TARGETS** option, and the *ExternalProject_Add_StepTargets()* function, can be used to create build targets for individual steps of an external project.

In CMake 3.18 and below, step targets have some limitations:

- Step targets always depend on targets named by the *ExternalProject_Add()* **DEPENDS** option even though not all steps need them. In order to allow step targets to be created without those dependencies, the *ExternalProject_Add()* **INDEPENDENT_STEP_TARGETS** option or the *ExternalProject_Add_StepTargets()* **NO_DEPENDS** option may be used. However, adding such "independent" step targets makes sense only for specific steps such as **download**, **update**, and **patch** because they do not need any of the external project's build dependencies. Furthermore, it does not make sense to create independent step targets for steps that depend on non-independent steps. Such rules are not enforced, and projects that do not follow them can generate build systems with confusing and generator-specific behavior.
- Step targets hold copies of the custom commands implementing their steps that are separate from the copies in the primary target created by *ExternalProject_Add()*, and the primary target does not depend on the step targets. In parallel builds that drive the primary target and step targets concurrently, multiple copies of the steps' commands may run concurrently and race each other.

Also, prior to policy *CMP0113*, the step targets generated by *Makefile Generators* also contain all the custom commands on which their step depends. This can lead to repeated execution of those steps even in serial builds.

In CMake 3.19 and above, the *ExternalProject* module prefers a revised design to address these problems:

- Each step is classified as "independent" if it does not depend on other targets named by the *ExternalProject_Add()* **DEPENDS**. The predefined steps are automatically classified by default:
 - The **download**, **update**, and **patch** steps are independent.
 - The **configure**, **build**, **test**, and **install** steps are not.

For custom steps, the *ExternalProject_Add_Step()* command provides an **INDEPENDENT** option to mark them as independent. It is an error to mark a step as independent if it depends on other steps that are not. Note that this use of the term "independent" refers only to independence from external targets and is orthogonal to a step's dependencies on other steps.

- Step targets created by the *ExternalProject_Add()* **STEP_TARGETS** option or the *ExternalProject_Add_Step()* function are now independent if and only if their steps are marked as independent. The *ExternalProject_Add()* **INDEPENDENT_STEP_TARGETS** option and *ExternalProject_Add_StepTargets()* **NO_DEPENDS** option are no longer allowed.
- Step targets, when created, are fully responsible for holding the custom commands implementing their steps. The primary target created by *ExternalProject_Add()* depends on the step targets, and the step targets depend on each other. The target-level dependencies match the file-level dependencies used by the custom commands for each step.

When the *ExternalProject_Add()* **UPDATE_DISCONNECTED** or **TEST_EXCLUDE_FROM_MAIN** option is used, or the *ExternalProject_Add_Step()* **EXCLUDE_FROM_MAIN** option is used for a custom step, some step targets may be created automatically. These are needed to hold the steps commonly depended upon by the primary target and the disconnected step targets.

Policy **CMP0114** provides compatibility for projects that have not been updated to expect the new behavior. The **OLD** behavior for this policy is to use the above-documented behavior from 3.18 and below. The **NEW** behavior for this policy is to use the above-documented behavior preferred by 3.19 and above.

This policy was introduced in CMake version 3.19. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0113

Added in version 3.19.

Makefile Generators do not repeat custom commands from target dependencies.

Consider a chain of custom commands split across two dependent targets:

```
add_custom_command(OUTPUT output-not-created
  COMMAND ... DEPENDS ...)
set_property(SOURCE output-not-created PROPERTY SYMBOLIC 1)
add_custom_command(OUTPUT output-created
  COMMAND ... DEPENDS ${CMAKE_CURRENT_BINARY_DIR}/output-not-created)
add_custom_target(first DEPENDS output-not-created)
add_custom_target(second DEPENDS output-created)
add_dependencies(second first)
```

In CMake 3.18 and lower, the Makefile generators put a copy of both custom commands in the Makefile for target **second** even though its dependency on target **first** ensures that the first custom command runs before the second. Running **make second** would cause the first custom command to run once in the **first** target and then again in the **second** target.

CMake 3.19 and above prefer to not duplicate custom commands in a target that are already generated in other targets on which the target depends (directly or indirectly). This policy provides compatibility for projects that have not been updated to expect the new behavior. In particular, projects that relied on the duplicate execution or that did not properly set the *SYMBOLIC* source file property may be affected.

The **OLD** behavior for this policy is to duplicate custom commands in dependent targets. The **NEW** behavior of this policy is to not duplicate custom commands in dependent targets.

This policy was introduced in CMake version 3.19. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0112

Added in version 3.19.

Target file component generator expressions do not add target dependencies.

The following target-based generator expressions that query for directory or file name components no longer add a dependency on the evaluated target.

- **TARGET_FILE_NAME**
- **TARGET_FILE_DIR**
- **TARGET_LINKER_FILE_BASE_NAME**

- **TARGET_LINKER_FILE_NAME**
- **TARGET_LINKER_FILE_DIR**
- **TARGET_SONAME_FILE_NAME**
- **TARGET_SONAME_FILE_DIR**
- **TARGET_PDB_FILE_NAME**
- **TARGET_PDB_FILE_DIR**
- **TARGET_BUNDLE_DIR**
- **TARGET_BUNDLE_DIR_NAME**
- **TARGET_BUNDLE_CONTENT_DIR**

In CMake 3.18 and lower a dependency on the evaluated target of the above generator expressions would always be added. CMake 3.19 and above prefer to not add this dependency. This policy provides compatibility for projects that have not been updated to expect the new behavior. The policy setting is recorded on each target when it is created, and decides whether generator expressions referencing that target imply a dependency on it.

The **OLD** behavior for this policy is to add a dependency on the evaluated target for the above generator expressions. The **NEW** behavior of this policy is to not add a dependency on the evaluated target for the above generator expressions.

This policy was introduced in CMake version 3.19. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn by default, and uses **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0112` variable to control the warning.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0111

Added in version 3.19.

An imported target missing its location property fails during generation.

Imported Targets for library files and executables require that their location on disk is specified in a target property such as `IMPORTED_LOCATION`, `IMPORTED_IMPLIB`, or a per-configuration equivalent. If a needed location property is not set, CMake 3.18 and below generate the string `<TARGET_NAME>-NOT-FOUND` in its place, which results in failures of the corresponding rules at build time. CMake 3.19 and above prefer instead to raise an error during generation. This policy provides compatibility for projects that have not been updated to expect the new behavior.

The **OLD** behavior of this policy is to generate the location of an imported unknown, static or shared library target as `<TARGET_NAME>-NOTFOUND` if not set. The **NEW** behavior is to raise an error.

This policy was introduced in CMake version 3.19. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0110

Added in version 3.19.

`add_test()` supports arbitrary characters in test names.

`add_test()` can now (officially) create tests with whitespace and other special characters in its name. Before CMake version 3.19 that was not allowed, however, it was possible to work around this limitation by explicitly putting escaped quotes around the test's name in the **add_test** command.

Although never officially supported several projects in the wild found and implemented this workaround. However, the new change which officially allows the **add_test** command to support whitespace and other special characters in test names now breaks that workaround. In order for these projects to work smoothly with newer CMake versions, this policy was introduced.

The **OLD** behavior of this policy is to still prevent **add_test** from handling whitespace and special characters properly (if not using the mentioned workaround). The **NEW** behavior on the other hand allows names with whitespace and special characters for tests created by **add_test**.

This policy was introduced in CMake version 3.19. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0109

Added in version 3.19.

`find_program()` requires permission to execute but not to read.

In CMake 3.18 and below, the `find_program()` command on UNIX would find files that are readable without requiring execute permission, and would not find files that are executable without read permission. In CMake 3.19 and above, **find_program** now prefers to require execute permission but not read permission. This policy provides compatibility with projects that have not been updated to expect the new behavior.

The **OLD** behavior for this policy is for **find_program** to require read permission but not execute permission. The **NEW** behavior for this policy is for **find_program** to require execute permission but not read permission.

This policy was introduced in CMake version 3.19. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.18**CMP0108**

Added in version 3.18.

A target is not allowed to link to itself even through an **ALIAS** target.

In CMake 3.17 and below, a target can link to a target aliased to itself.

The **OLD** behavior for this policy is to allow a target to link to a target aliased to itself.

The **NEW** behavior of this policy is to prevent a target to link to itself through an **ALIAS** target.

This policy was introduced in CMake version 3.17. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0107

Added in version 3.18.

It is not allowed to create an **ALIAS** target with the same name as an another target.

In CMake 3.17 and below, an **ALIAS** target can overwrite silently an existing target with the same name.

The **OLD** behavior for this policy is to allow target overwrite.

The **NEW** behavior of this policy is to prevent target overwriting.

This policy was introduced in CMake version 3.17. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0106

Added in version 3.18.

The *Documentation* module is removed.

The *Documentation* was added as a support mechanism for the VTK project and was tuned for that project. Instead of CMake providing this module with (now old) VTK patterns for cache variables and required packages, the module is now deprecated by CMake itself.

The **OLD** behavior of this policy is for *Documentation* to add cache variables and find VTK documentation dependent packages. The **NEW** behavior is to act as an empty module.

This policy was introduced in CMake version 3.18. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0105

Added in version 3.18.

`LINK_OPTIONS` and `INTERFACE_LINK_OPTIONS` target properties are now used for the device link step.

In CMake 3.17 and below, link options are not used by the device link step.

The **OLD** behavior for this policy is to ignore the link options during the device link step.

The **NEW** behavior of this policy is to use the link options during the device link step.

This policy was introduced in CMake version 3.18. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0104

Added in version 3.18.

Initialize `CMAKE_CUDA_ARCHITECTURES` when `CMAKE_CUDA_COMPILER_ID` is **NVIDIA**. Raise an error if `CUDA_ARCHITECTURES` is empty.

`CMAKE_CUDA_ARCHITECTURES` introduced in CMake 3.18 is used to initialize `CUDA_ARCHITECTURES`, which passes correct code generation flags to the CUDA compiler.

Previous to this users had to manually specify the code generation flags. This policy is for backwards compatibility with manually specifying code generation flags.

The **OLD** behavior for this policy is to not initialize `CMAKE_CUDA_ARCHITECTURES` when `CMAKE_CUDA_COMPILER_ID` is **NVIDIA**. Empty `CUDA_ARCHITECTURES` is allowed.

The **NEW** behavior of this policy is to initialize `CMAKE_CUDA_ARCHITECTURES` when `CMAKE_CUDA_COMPILER_ID` is **NVIDIA** and raise an error if `CUDA_ARCHITECTURES` is empty during generation.

If `CUDA_ARCHITECTURES` is set to a false value no architectures flags are passed to the compiler. This is intended to support packagers and the rare cases where full control over the passed flags is required.

This policy was introduced in CMake version 3.18. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Examples

```
set_target_properties(tgt PROPERTIES CUDA_ARCHITECTURES "35;50;72")
```

Generates code for real and virtual architectures **30**, **50** and **72**.

```
set_property(TARGET tgt PROPERTY CUDA_ARCHITECTURES 70-real 72-virtual)
```

Generates code for real architecture **70** and virtual architecture **72**.

```
set_property(TARGET tgt PROPERTY CUDA_ARCHITECTURES OFF)
```

CMake will not pass any architecture flags to the compiler.

CMP0103

Added in version 3.18.

Multiple calls to `export()` command with same **FILE** without **APPEND** is no longer allowed.

In CMake 3.17 and below, multiple calls to `export()` command with the same **FILE** without **APPEND** are accepted silently but only the last occurrence is taken into account during the generation.

The OLD behavior for this policy is to ignore the multiple occurrences of
`export()` command except the last one.

The **NEW** behavior of this policy is to raise an error on second call to `export()` command with same **FILE** without **APPEND**.

This policy was introduced in CMake version 3.18. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.17**CMP0102**

Added in version 3.17.

The `mark_as_advanced()` command no longer creates a cache entry if one does not already exist.

In CMake 3.16 and below, if a variable was not defined at all or just defined locally, the `mark_as_advanced()` command would create a new cache entry with an **UNINITIALIZED** type and no value. When a `find_path()` (or other similar **find_** command) would next run, it would find this undefined cache entry and set it up with an empty string value. This process would end up deleting the local variable in the process (due to the way the cache works), effectively clearing any stored **find_** results that were only available in the local scope.

The **OLD** behavior for this policy is to create the empty cache definition. The **NEW** behavior of this policy is to ignore variables which do not already exist in the cache.

This policy was introduced in CMake version 3.17. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn by default, and uses **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0102` variable to control the warning.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0101

Added in version 3.17.

`target_compile_options()` now always honors the **BEFORE** keyword.

In CMake 3.16 and below, the `target_compile_options()` command ignores the **BEFORE** keyword when inserting items into the `COMPILE_OPTIONS` target property (**PRIVATE** and **PUBLIC** items). CMake 3.17

and later honors the **BEFORE** keyword in all cases. This policy provides compatibility for projects that have not been updated to expect the new behavior.

The behavior of inserting items into the `INTERFACE_COMPILE_OPTIONS` target property (**PUBLIC** and **INTERFACE** items) is not affected by this policy. The **BEFORE** keyword has always been honored when adding items to `INTERFACE_COMPILE_OPTIONS`.

The **OLD** behavior for this policy is to not honor the **BEFORE** keyword when inserting into the `COMPILE_OPTIONS` property. The **NEW** behavior for this policy is to honor the **BEFORE** keyword in all cases.

This policy was introduced in CMake version 3.17. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0100

Added in version 3.17.

Let `AUTOMOC` and `AUTOUIC` process header files that end with a **.hh** extension.

Since version 3.17, CMake processes header files that end with a **.hh** extension in `AUTOMOC` and `AUTOUIC`. In earlier CMake versions, these header files were ignored by `AUTOMOC` and `AUTOUIC`.

This policy affects how header files that end with a **.hh** extension get treated in `AUTOMOC` and `AUTOUIC`.

The **OLD** behavior for this policy is to ignore **.hh** header files in `AUTOMOC` and `AUTOUIC`.

The **NEW** behavior for this policy is to process **.hh** header files in `AUTOMOC` and `AUTOUIC` just like other header files.

NOTE:

To silence the **CMP0100** warning source files can be excluded from `AUTOMOC` and `AUTOUIC` processing by setting the source file properties `SKIP_AUTOMOC`, `SKIP_AUTOUIC` or `SKIP_AUTOGEN`.

```
# Source skip example:
set_property(SOURCE /path/to/file1.hh PROPERTY SKIP_AUTOMOC ON)
set_property(SOURCE /path/to/file2.hh PROPERTY SKIP_AUTOUIC ON)
set_property(SOURCE /path/to/file3.hh PROPERTY SKIP_AUTOGEN ON)
```

This policy was introduced in CMake version 3.17.0. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0099

Added in version 3.17.

Link properties are transitive over private dependencies of static libraries.

In CMake 3.16 and below, evaluation of target properties *INTERFACE_LINK_OPTIONS*, *INTERFACE_LINK_DIRECTORIES*, and *INTERFACE_LINK_DEPENDS* during buildsystem generation does not follow private dependencies of static libraries, which appear in their *INTERFACE_LINK_LIBRARIES* guarded by *LINK_ONLY* generator expressions. Only the libraries themselves are propagated to link the dependent binary.

CMake 3.17 and later prefer to propagate all interface link properties. This policy provides compatibility for projects that have not been updated to expect the new behavior.

The **OLD** behavior for this policy is to not propagate interface link properties. The **NEW** behavior of this policy is to propagate interface link properties.

Added in version 3.30: Policy *CMP0166* makes *TARGET_PROPERTY* evaluation of these three transitive link properties follow private dependencies of static libraries too.

This policy was introduced in CMake version 3.17. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0098

Added in version 3.17.

FindFLEX runs **flex** in directory *CMAKE_CURRENT_BINARY_DIR* when executing.

The module provides a **FLEX_TARGET** macro which generates FLEX output. In CMake 3.16 and below the macro would generate a custom command that runs **flex** in the current source directory. CMake 3.17 and later prefer to run it in the build directory and use *CMAKE_CURRENT_BINARY_DIR* as the **WORKING_DIRECTORY** of its *add_custom_command()* invocation. This ensures that any implicitly generated file is written relative to the build tree rather than the source tree, unless the generated file is provided as absolute path.

This policy provides compatibility for projects that have not been updated to expect the new behavior.

The **OLD** behavior for this policy is for **FLEX_TARGET** to use the current source directory for the **WORKING_DIRECTORY** and where to generate implicit files. The **NEW** behavior of this policy is to use the current binary directory for the **WORKING_DIRECTORY** relative to which implicit files are generated unless provided as absolute path.

This policy was introduced in CMake version 3.17. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.16

CMP0097

Added in version 3.16.

ExternalProject_Add() with **GIT_SUBMODULES** "" initializes no submodules.

The commands provide a **GIT_SUBMODULES** option which controls what submodules to initialize and update. Starting with CMake 3.16, explicitly setting **GIT_SUBMODULES** to an empty string means no submodules will be initialized or updated.

This policy provides compatibility for projects that have not been updated to expect the new behavior.

The **OLD** behavior for this policy is for **GIT_SUBMODULES** when set to an empty string to initialize and update all git submodules. The **NEW** behavior for this policy is for **GIT_SUBMODULES** when set to an empty string to initialize and update no git submodules.

This policy was introduced in CMake version 3.16. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

This policy also applies to `FetchContent_Declare()`, which uses the same download and update features as `ExternalProject_Add()`. However, due to an implementation deficiency present since the policy was first introduced, CMake 3.16 and later always uses the **NEW** behavior for `FetchContent_Declare()`, regardless of the policy setting. Formally, this forcing of **NEW** behavior for `FetchContent_Declare()` will continue to apply in future CMake releases.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0096

Added in version 3.16.

The `project()` command preserves leading zeros in version components.

When a **VERSION** `<major>[.<minor>[.<patch>[.<tweak>]]]` argument is given to the `project()` command, it stores the version string in the **PROJECT_VERSION** variable and stores individual integer version components in **PROJECT_VERSION_{MAJOR,MINOR,PATCH,TWEAK}** variables (see policy `CMP0048`). CMake 3.15 and below dropped leading zeros from each component. CMake 3.16 and higher prefer to preserve leading zeros. This policy provides compatibility for projects that have not been updated to expect the new behavior.

The **OLD** behavior of this policy drops leading zeros in all components, e.g. such that version **1.07.06** becomes **1.7.6**. The **NEW** behavior of this policy preserves the leading zeros in all components, such that version **1.07.06** remains unchanged.

This policy was introduced in CMake version 3.16. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0095

Added in version 3.16.

RPATH entries are properly escaped in the intermediary CMake install script.

In CMake 3.15 and earlier, **RPATH** entries set via `CMAKE_INSTALL_RPATH` or via `INSTALL_RPATH` have not been escaped before being inserted into the `cmake_install.cmake` script. Dynamic linkers on

ELF-based systems (e.g. Linux and FreeBSD) allow certain keywords in **RPATH** entries, such as **\${ORIGIN}** (More details are available in the **ld.so** man pages on those systems). The syntax of these keywords can match CMake's variable syntax. In order to not be substituted (usually to an empty string) already by the intermediary **cmake_install.cmake** script, the user had to double-escape such **RPATH** keywords, e.g. **set(CMAKE_INSTALL_RPATH "\\\${ORIGIN}/../lib")**. Since the intermediary **cmake_install.cmake** script is an implementation detail of CMake, CMake 3.16 and later will make sure **RPATH** entries are inserted literally by escaping any coincidental CMake syntax.

The **OLD** behavior of this policy is to not escape **RPATH** entries in the intermediary **cmake_install.cmake** script. The **NEW** behavior is to properly escape coincidental CMake syntax in **RPATH** entries when generating the intermediary **cmake_install.cmake** script.

This policy was introduced in CMake version 3.16. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns when it detects use of CMake-like syntax, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.15

CMP0094

Added in version 3.15.

Modules *FindPython3*, *FindPython2* and *FindPython* use **LOCATION** for lookup strategy.

Starting with CMake 3.15, Modules *FindPython3*, *FindPython2* and *FindPython* set value **LOCATION** for, respectively, variables **Python3_FIND_STRATEGY**, **Python2_FIND_STRATEGY** and **Python_FIND_STRATEGY**. This policy provides compatibility with projects that expect the legacy behavior.

The **OLD** behavior for this policy set value **VERSION** for variables **Python3_FIND_STRATEGY**, **Python2_FIND_STRATEGY** and **Python_FIND_STRATEGY**.

This policy was introduced in CMake version 3.15. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0093

Added in version 3.15.

FindBoost reports **Boost_VERSION** in **x.y.z** format.

In CMake 3.14 and below the module would report the Boost version number as specified in the preprocessor definition **BOOST_VERSION** in the **boost/version.hpp** file. In CMake 3.15 and later it is preferred that the reported version number matches the **x.y.z** format reported by the CMake package shipped with Boost **1.70.0** and later. The macro value is still reported in the **Boost_VERSION_MACRO** variable.

The **OLD** behavior for this policy is for *FindBoost* to report **Boost_VERSION** as specified in the preprocessor definition **BOOST_VERSION** in **boost/version.hpp**. The **NEW** behavior for this policy is for *FindBoost* to report **Boost_VERSION** in **x.y.z** format.

This policy was introduced in CMake version 3.15. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0092

Added in version 3.15.

MSVC warning flags are not in `CMAKE_<LANG>_FLAGS` by default.

When using MSVC-like compilers in CMake 3.14 and below, warning flags like `/W3` are added to `CMAKE_<LANG>_FLAGS` by default. This is problematic for projects that want to choose a different warning level programmatically. In particular, it requires string editing of the `CMAKE_<LANG>_FLAGS` variables with knowledge of the CMake builtin defaults so they can be replaced.

CMake 3.15 and above prefer to leave out warning flags from the value of `CMAKE_<LANG>_FLAGS` by default.

This policy provides compatibility with projects that have not been updated to expect the lack of warning flags. The policy setting takes effect as of the first `project()` or `enable_language()` command that initializes `CMAKE_<LANG>_FLAGS` for a given language `<LANG>`.

NOTE:

Once the policy has taken effect at the top of a project for a given language, that choice must be used throughout the tree for that language. In projects that have nested projects in subdirectories, be sure to convert everything together.

The **OLD** behavior for this policy is to place MSVC warning flags in the default `CMAKE_<LANG>_FLAGS` cache entries. The **NEW** behavior for this policy is to *not* place MSVC warning flags in the default cache entries.

This policy was introduced in CMake version 3.15. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0091

Added in version 3.15.

MSVC runtime library flags are selected by an abstraction.

Compilers targeting the MSVC ABI have flags to select the MSVC runtime library. Runtime library selection typically varies with build configuration because there is a separate runtime library for Debug builds.

In CMake 3.14 and below, MSVC runtime library selection flags are added to the default `CMAKE_<LANG>_FLAGS_<CONFIG>` cache entries by CMake automatically. This allows users to edit their cache entries to adjust the flags. However, the presence of such default flags is problematic for projects that want to choose a different runtime library programmatically. In particular, it requires string editing of the `CMAKE_<LANG>_FLAGS_<CONFIG>` variables with knowledge of the CMake builtin defaults so they can be replaced.

CMake 3.15 and above prefer to leave the MSVC runtime library selection flags out of the default `CMAKE_<LANG>_FLAGS_<CONFIG>` values and instead offer a first-class abstraction. The `CMAKE_MSVC_RUNTIME_LIBRARY` variable and `MSVC_RUNTIME_LIBRARY` target property may be set to select the MSVC runtime library. If they are not set then CMake uses the default value **Multi-Threaded\$<\$<CONFIG:Debug>:Debug>DLL** which is equivalent to the original flags.

This policy provides compatibility with projects that have not been updated to be aware of the abstraction. The policy setting takes effect as of the first `project()` or `enable_language()` command that enables a language whose compiler targets the MSVC ABI.

NOTE:

Once the policy has taken effect at the top of a project, that choice must be used throughout the tree. In projects that have nested projects in subdirectories, be sure to convert everything together.

The **OLD** behavior for this policy is to place MSVC runtime library flags in the default `CMAKE_<LANG>_FLAGS_<CONFIG>` cache entries and ignore the `CMAKE_MSVC_RUNTIME_LIBRARY` abstraction. The **NEW** behavior for this policy is to *not* place MSVC runtime library flags in the default cache entries and use the abstraction instead.

This policy was introduced in CMake version 3.15. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0090

Added in version 3.15.

`export(PACKAGE)` does not populate package registry by default.

In CMake 3.14 and below the `export(PACKAGE)` command populated the user package registry by default and users needed to set the `CMAKE_EXPORT_NO_PACKAGE_REGISTRY` to disable it, e.g. in automated build and packaging environments. Since the user package registry is stored outside the build tree, this side effect should not be enabled by default. Therefore CMake 3.15 and above prefer that `export(PACKAGE)` does nothing unless an explicit `CMAKE_EXPORT_PACKAGE_REGISTRY` variable is set to enable it. This policy provides compatibility with projects that have not been updated.

The **OLD** behavior for this policy is for `export(PACKAGE)` command to populate the user package registry unless `CMAKE_EXPORT_NO_PACKAGE_REGISTRY` is enabled. The **NEW** behavior is for `export(PACKAGE)` command to do nothing unless the `CMAKE_EXPORT_PACKAGE_REGISTRY` is enabled.

This policy was introduced in CMake version 3.15. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0089

Added in version 3.15.

Compiler id for IBM Clang-based XL compilers is now **XLClang**.

CMake 3.15 and above recognize that IBM's Clang-based XL compilers that define `__ibmxl__` are a new front-end distinct from `xlc` with a different command line and set of capabilities. CMake now prefers to present this to projects by setting the `CMAKE_<LANG>_COMPILER_ID` variable to `XLClang` instead of `XL`. However, existing projects may assume the compiler id for Clang-based XL is just `XL` as it was in CMake versions prior to 3.15. Therefore this policy determines for Clang-based XL compilers which compiler id to report in the `CMAKE_<LANG>_COMPILER_ID` variable after language `<LANG>` is enabled by the `project()` or `enable_language()` command. The policy must be set prior to the invocation of either command.

The **OLD** behavior for this policy is to use compiler id `XL`. The **NEW** behavior for this policy is to use compiler id `XLClang`.

This policy was introduced in CMake version 3.15. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn by default, and uses **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0089` variable to control the warning.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.14

CMP0088

Added in version 3.14.

`FindBISON` runs bison in `CMAKE_CURRENT_BINARY_DIR` when executing.

The module provides a **BISON_TARGET** macro which generates BISON output. In CMake 3.13 and below the macro would generate a custom command that runs `bison` in the source directory. CMake 3.14 and later prefer to run it in the build directory and use `CMAKE_CURRENT_BINARY_DIR` as the **WORKING_DIRECTORY** of its `add_custom_command()` invocation. This ensures that any implicitly generated file is written to the build tree rather than the source.

This policy provides compatibility for projects that have not been updated to expect the new behavior.

The **OLD** behavior for this policy is for **BISON_TARGET** to use the current source directory for the **WORKING_DIRECTORY** and where to generate implicit files. The **NEW** behavior of this policy is to use the current binary directory for the **WORKING_DIRECTORY** and where to generate implicit files.

This policy was introduced in CMake version 3.14. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0087

Added in version 3.14.

`install(CODE)` and `install(SCRIPT)` support generator expressions.

In CMake 3.13 and earlier, `install(CODE)` and `install(SCRIPT)` did not evaluate generator expressions. CMake 3.14 and later will evaluate generator expressions for `install(CODE)` and `install(SCRIPT)`.

The **OLD** behavior of this policy is for *install(CODE)* and *install(SCRIPT)* to not evaluate generator expressions. The **NEW** behavior is to evaluate generator expressions for *install(CODE)* and *install(SCRIPT)*.

Note that it is the value of this policy setting at the end of the directory scope that is important, not its setting at the time of the call to *install(CODE)* or *install(SCRIPT)*. This has implications for calling these commands from places that have their own policy scope but not their own directory scope (e.g. from files brought in via *include()* rather than *add_subdirectory()*).

This policy was introduced in CMake version 3.14. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0086

Added in version 3.14.

UseSWIG honors **SWIG_MODULE_NAME** via **-module** flag.

Starting with CMake 3.14, *UseSWIG* passes option **-module** <module_name> to **SWIG** compiler if the file property **SWIG_MODULE_NAME** is specified. This policy provides compatibility with projects that expect the legacy behavior.

The **OLD** behavior for this policy is to never pass **-module** option. The **NEW** behavior is to pass **-module** option to **SWIG** compiler if **SWIG_MODULE_NAME** is specified.

This policy was introduced in CMake version 3.14. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0085

Added in version 3.14.

\$<IN_LIST:...> handles empty list items.

In CMake 3.13 and lower, the **\$<IN_LIST:...>** generator expression always returned **0** if the first argument was empty, even if the list contained an empty item. This behavior is inconsistent with the **IN_LIST** behavior of *if()*, which this generator expression is meant to emulate. CMake 3.14 and later handles this case correctly.

The **OLD** behavior of this policy is for **\$<IN_LIST:...>** to always return **0** if the first argument is empty. The **NEW** behavior is to return **1** if the first argument is empty and the list contains an empty item.

This policy was introduced in CMake version 3.14. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0084

Added in version 3.14.

The *FindQt* module does not exist for *find_package()*.

The existence of *FindQt* means that for Qt upstream to provide package config files that can be found by **find_package(Qt)**, the consuming project has to explicitly specify **find_package(Qt CONFIG)**. Removing this module gives Qt a path forward for exporting its own config files which can easily be found by consuming projects.

This policy pretends that CMake's internal *FindQt* module does not exist for *find_package()*. If a project really wants to use Qt 3 or 4, it can call **find_package(Qt[34])**, **include(FindQt)**, or add *FindQt* to their *CMAKE_MODULE_PATH*.

The **OLD** behavior of this policy is for *FindQt* to exist for *find_package()*. The **NEW** behavior is to pretend that it doesn't exist for *find_package()*.

This policy was introduced in CMake version 3.14. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0083

Added in version 3.14.

To control generation of Position Independent Executable (**PIE**) or not, some flags are required at link time.

CMake 3.13 and lower did not add these link flags when *POSITION_INDEPENDENT_CODE* is set.

The **OLD** behavior for this policy is to not manage **PIE** link flags. The **NEW** behavior is to add link flags if *POSITION_INDEPENDENT_CODE* is set:

- Set to **TRUE**: flags to produce a position independent executable are passed to the linker step. For example **-pie** for **GCC**.
- Set to **FALSE**: flags not to produce a position independent executable are passed to the linker step. For example **-no-pie** for **GCC**.
- Not set: no flags are passed to the linker step.

Since a given linker may not support **PIE** flags in all environments in which it is used, it is the project's responsibility to use the *CheckPIESupported* module to check for support to ensure that the *POSITION_INDEPENDENT_CODE* target property for executables will be honored at link time.

This policy was introduced in CMake version 3.14. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

Android platform has a special handling of **PIE** so it is not required to use the *CheckPIESupported* module to ensure flags are passed to the linker.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of

CMake.

Examples

Behave like CMake 3.13 and do not apply any **PIE** flags at link stage.

```
cmake_minimum_required(VERSION 3.13)
project(foo)

# ...

add_executable(foo ...)
set_property(TARGET foo PROPERTY POSITION_INDEPENDENT_CODE TRUE)
```

Use the *CheckPIESupported* module to detect whether **PIE** is supported by the current linker and environment. Apply **PIE** flags only if the linker supports them.

```
cmake_minimum_required(VERSION 3.14) # CMP0083 NEW
project(foo)

include(CheckPIESupported)
check_pie_supported()

# ...

add_executable(foo ...)
set_property(TARGET foo PROPERTY POSITION_INDEPENDENT_CODE TRUE)
```

CMP0082

Added in version 3.14.

Install rules from *add_subdirectory()* calls are interleaved with those in caller.

CMake 3.13 and lower ran the install rules from *add_subdirectory()* after all other install rules, even if *add_subdirectory()* was called before the other install rules. CMake 3.14 and above prefer to interleave these *add_subdirectory()* install rules with the others so that they are run in the order they are declared. This policy provides compatibility for projects that have not been updated to expect the new behavior.

The **OLD** behavior for this policy is to run the install rules from *add_subdirectory()* after the other install rules. The **NEW** behavior for this policy is to run all install rules in the order they are declared.

This policy was introduced in CMake version 3.14. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn by default, and uses **OLD** behavior.

See documentation of the *CMAKE_POLICY_WARNING_CMP0082* variable to control the warning.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.13

CMP0081

Added in version 3.13.

Relative paths not allowed in *LINK_DIRECTORIES* target property.

CMake 3.12 and lower allowed the *LINK_DIRECTORIES* directory property to contain relative paths. The base path for such relative entries is not well defined. CMake 3.13 and later will issue a **FATAL_ERROR** if the *LINK_DIRECTORIES* target property (which is initialized by the *LINK_DIRECTORIES* directory property) contains a relative path.

The **OLD** behavior for this policy is not to warn about relative paths in the *LINK_DIRECTORIES* target property. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** if *LINK_DIRECTORIES* contains a relative path.

This policy was introduced in CMake version 3.13. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0080

Added in version 3.13.

BundleUtilities cannot be included at configure time.

The macros provided by *BundleUtilities* are intended to be invoked at install time rather than at configure time, because they depend on the listed targets already existing at the time they are invoked. If they are invoked at configure time, the targets haven't been built yet, and the commands will fail.

This policy restricts the inclusion of *BundleUtilities* to **cmake -P** style scripts and install rules. Specifically, it looks for the presence of *CMAKE_GENERATOR* and throws a fatal error if it exists.

The **OLD** behavior of this policy is to allow *BundleUtilities* to be included at configure time. The **NEW** behavior of this policy is to disallow such inclusion.

This policy was introduced in CMake version 3.13. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0079

Added in version 3.13.

target_link_libraries() allows use with targets in other directories.

Prior to CMake 3.13 the *target_link_libraries()* command did not accept targets not created in the calling directory as its first argument for calls that update the *LINK_LIBRARIES* of the target itself. It did accidentally accept targets from other directories on calls that only update the *INTERFACE_LINK_LIBRARIES*, but would simply add entries to the property as if the call were made in the original directory. Thus link interface libraries specified this way were always looked up by generators in the scope of the original target rather than in the scope that called *target_link_libraries()*.

CMake 3.13 now allows the *target_link_libraries()* command to be called from any directory to add link dependencies and link interface libraries to targets created in other directories. The entries are added to *LINK_LIBRARIES* and *INTERFACE_LINK_LIBRARIES* using a special (internal) suffix to tell the generators to look up the names in the calling scope rather than the scope that created the target.

This policy provides compatibility with projects that already use `target_link_libraries()` with the **INTERFACE** keyword on a target in another directory to add `INTERFACE_LINK_LIBRARIES` entries to be looked up in the target's directory. Such projects should be updated to be aware of the new scoping rules in that case.

The **OLD** behavior of this policy is to disallow `target_link_libraries()` calls naming targets from another directory except in the previously accidentally allowed case of using the **INTERFACE** keyword only. The **NEW** behavior of this policy is to allow all such calls but use the new scoping rules.

This policy was introduced in CMake version 3.13. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0078

Added in version 3.13.

`UseSWIG` generates standard target names.

Starting with CMake 3.13, `UseSWIG` generates now standard target names. This policy provides compatibility with projects that expect the legacy behavior.

The **OLD** behavior for this policy relies on `UseSWIG_TARGET_NAME_PREFERENCE` variable that can be used to specify an explicit preference. The value may be one of:

- **LEGACY:** legacy strategy is applied. Variable `SWIG_MODULE_<name>_REAL_NAME` must be used to get real target name. This is the default if not specified.
- **STANDARD:** target name matches specified name.

This policy was introduced in CMake version 3.13. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0077

Added in version 3.13.

`option()` honors normal variables.

The `option()` command is typically used to create a cache entry to allow users to set the option. However, there are cases in which a normal (non-cached) variable of the same name as the option may be defined by the project prior to calling the `option()` command. For example, a project that embeds another project as a subdirectory may want to hard-code options of the subproject to build the way it needs.

For historical reasons in CMake 3.12 and below the `option()` command *removes* a normal (non-cached) variable of the same name when:

- a cache entry of the specified name does not exist at all, or

- a cache entry of the specified name exists but has not been given a type (e.g. via `-D<name>=ON` on the command line).

In both of these cases (typically on the first run in a new build tree), the `option()` command gives the cache entry type **BOOL** and removes any normal (non-cached) variable of the same name. In the remaining case that the cache entry of the specified name already exists and has a type (typically on later runs in a build tree), the `option()` command changes nothing and any normal variable of the same name remains set.

In CMake 3.13 and above the `option()` command prefers to do nothing when a normal variable of the given name already exists. It does not create or update a cache entry or remove the normal variable. The new behavior is consistent between the first and later runs in a build tree. This policy provides compatibility with projects that have not been updated to expect the new behavior.

When the `option()` command sees a normal variable of the given name:

- The **OLD** behavior for this policy is to proceed even when a normal variable of the same name exists. If the cache entry does not already exist and have a type then it is created and/or given a type and the normal variable is removed.
- The **NEW** behavior for this policy is to do nothing when a normal variable of the same name exists. The normal variable is not removed. The cache entry is not created or updated and is ignored if it exists.

See `CMP0126` for a similar policy for the `set(CACHE)` command, but note that there are some differences in **NEW** behavior between the two policies.

This policy was introduced in CMake version 3.13. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

Use the `CMAKE_POLICY_DEFAULT_CMP0077` variable to set the policy for a third-party project in a subdirectory without modifying it.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0076

Added in version 3.13.

The `target_sources()` command converts relative paths to absolute.

In CMake 3.13 and above, the `target_sources()` command now converts relative source file paths to absolute paths in the following cases:

- Source files are added to the target's `INTERFACE_SOURCES` property.
- The target's `SOURCE_DIR` property differs from `CMAKE_CURRENT_SOURCE_DIR`.

A path that begins with a generator expression is always left unmodified.

This policy provides compatibility with projects that have not been updated to expect this behavior. The **OLD** behavior for this policy is to leave all relative source file paths unmodified. The **NEW** behavior of this policy is to convert relative paths to absolute according to above rules.

This policy was introduced in CMake version 3.13. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.12**CMP0075**

Added in version 3.12.

Include file check macros honor **CMAKE_REQUIRED_LIBRARIES**.

In CMake 3.12 and above, the

- **check_include_file** macro in the *CheckIncludeFile* module, the
- **check_include_file_cxx** macro in the *CheckIncludeFileCXX* module, and the
- **check_include_files** macro in the *CheckIncludeFiles* module

now prefer to link the check executable to the libraries listed in the **CMAKE_REQUIRED_LIBRARIES** variable. This policy provides compatibility with projects that have not been updated to expect this behavior.

The **OLD** behavior for this policy is to ignore **CMAKE_REQUIRED_LIBRARIES** in the include file check macros. The **NEW** behavior of this policy is to honor **CMAKE_REQUIRED_LIBRARIES** in the include file check macros.

This policy was introduced in CMake version 3.12. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0074

Added in version 3.12.

find_package() uses **<PackageName>_ROOT** variables.

In CMake 3.12 and above the *find_package(<PackageName>)* command now searches prefixes specified by the **<PackageName>_ROOT** CMake variable and the **<PackageName>_ROOT** environment variable. Package roots are maintained as a stack so nested calls to all **find_*** commands inside find modules and config packages also search the roots as prefixes. This policy provides compatibility with projects that have not been updated to avoid using **<PackageName>_ROOT** variables for other purposes.

The **OLD** behavior for this policy is to ignore **<PackageName>_ROOT** variables. The **NEW** behavior for this policy is to use **<PackageName>_ROOT** variables.

This policy was introduced in CMake version 3.12. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0073

Added in version 3.12.

Do not produce legacy **_LIB_DEPENDS** cache entries.

Ancient CMake versions once used **<tgt>_LIB_DEPENDS** cache entries to propagate library link dependencies. This has long been done by other means, leaving the *export_library_dependencies()* command as the only user of these values. That command has long been disallowed by policy *CMP0033*, but the **<tgt>_LIB_DEPENDS** cache entries were left for compatibility with possible non-standard uses by projects.

CMake 3.12 and above now prefer to not produce these cache entries at all. This policy provides compatibility with projects that have not been updated to avoid using them.

The **OLD** behavior for this policy is to set **<tgt>_LIB_DEPENDS** cache entries. The **NEW** behavior for this policy is to not set them.

This policy was introduced in CMake version 3.12. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake does *not* warn, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.11**CMP0072**

Added in version 3.11.

FindOpenGL prefers GLVND by default when available.

The *FindOpenGL* module provides an **OpenGL::GL** target and an **OPENGL_LIBRARIES** variable for projects to use for legacy GL interfaces. When both a legacy GL library (e.g. **libGL.so**) and GLVND libraries for OpenGL and GLX (e.g. **libOpenGL.so** and **libGLX.so**) are available, the module must choose between them. It documents an **OpenGL_GL_PREFERENCE** variable that can be used to specify an explicit preference. When no such preference is set, the module must choose a default preference.

CMake 3.11 and above prefer to choose GLVND libraries. This policy provides compatibility with projects that expect the legacy GL library to be used.

The **OLD** behavior for this policy is to set **OpenGL_GL_PREFERENCE** to **LEGACY**. The **NEW** behavior for this policy is to set **OpenGL_GL_PREFERENCE** to **GLVND**.

This policy was introduced in CMake version 3.11. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.10**CMP0071**

Added in version 3.10.

Let *AUTOMOC* and *AUTOUIC* process *GENERATED* files.

Since version 3.10, CMake processes **regular** and *GENERATED* source files in *AUTOMOC* and *AUTOUIC*. In earlier CMake versions, only **regular** source files were processed. *GENERATED* source files were ignored silently.

This policy affects how source files that are *GENERATED* get treated in *AUTOMOC* and *AUTOUIC*.

The **OLD** behavior for this policy is to ignore *GENERATED* source files in *AUTOMOC* and *AUTOUIC*.

The **NEW** behavior for this policy is to process *GENERATED* source files in *AUTOMOC* and *AUTOUIC* just like regular source files.

NOTE:

To silence the **CMP0071** warning source files can be excluded from *AUTOMOC* and *AUTOUIC* processing by setting the source file properties *SKIP_AUTOMOC*, *SKIP_AUTOUIC* or *SKIP_AUTOGEN*.

Source skip example:

```
# ...
set_property(SOURCE /path/to/file1.h PROPERTY SKIP_AUTOMOC ON)
set_property(SOURCE /path/to/file2.h PROPERTY SKIP_AUTOUIC ON)
set_property(SOURCE /path/to/file3.h PROPERTY SKIP_AUTOGEN ON)
# ...
```

This policy was introduced in CMake version 3.10. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

CMP0070

Added in version 3.10.

Define *file(GENERATE)* behavior for relative paths.

CMake 3.10 and newer define that relative paths given to **INPUT** and **OUTPUT** arguments of **file(GENERATE)** are interpreted relative to the current source and binary directories, respectively. CMake 3.9 and lower did not define any behavior for relative paths but did not diagnose them either and accidentally treated them relative to the process working directory. Policy **CMP0070** provides compatibility with projects that used the old undefined behavior.

This policy affects behavior of relative paths given to **file(GENERATE)**. The **OLD** behavior for this policy is to treat the paths relative to the working directory of CMake. The **NEW** behavior for this policy is to interpret relative paths with respect to the current source or binary directory of the caller.

This policy was introduced in CMake version 3.10. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.9**CMP0069**

Added in version 3.9.

INTERPROCEDURAL_OPTIMIZATION is enforced when enabled.

CMake 3.9 and newer prefer to add IPO flags whenever the *INTERPROCEDURAL_OPTIMIZATION* target property is enabled and produce an error if flags are not known to CMake for the current compiler. Since a given compiler may not support IPO flags in all environments in which it is used, it is now the project's responsibility to use the *CheckIPOSupported* module to check for support before enabling the *INTERPROCEDURAL_OPTIMIZATION* target property. This approach allows a project to conditionally activate IPO when supported. It also allows an end user to set the *CMAKE_INTERPROCEDURAL_OPTIMIZATION* variable in an environment known to support IPO even if the project does not enable the property.

Since CMake 3.8 and lower only honored *INTERPROCEDURAL_OPTIMIZATION* for the Intel compiler on Linux, some projects may unconditionally enable the target property. Policy **CMP0069** provides compatibility with such projects.

This policy takes effect whenever the IPO property is enabled. The **OLD** behavior for this policy is to add IPO flags only for Intel compiler on Linux. The **NEW** behavior for this policy is to add IPO flags for the current compiler or produce an error if CMake does not know the flags.

This policy was introduced in CMake version 3.9. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Examples

Behave like CMake 3.8 and do not apply any IPO flags except for Intel compiler on Linux:

```
cmake_minimum_required(VERSION 3.8)
project(foo)

# ...

set_property(TARGET ... PROPERTY INTERPROCEDURAL_OPTIMIZATION TRUE)
```

Use the *CheckIPOSupported* module to detect whether IPO is supported by the current compiler, environment, and CMake version. Produce a fatal error if support is not available:

```
cmake_minimum_required(VERSION 3.9) # CMP0069 NEW
project(foo)

include(CheckIPOSupported)
check_ipo_supported()

# ...

set_property(TARGET ... PROPERTY INTERPROCEDURAL_OPTIMIZATION TRUE)
```

Apply IPO flags only if compiler supports it:

```

cmake_minimum_required(VERSION 3.9) # CMP0069 NEW
project(foo)

include(CheckIPOSupported)

# ...

check_ipo_supported(RESULT result)
if(result)
    set_property(TARGET ... PROPERTY INTERPROCEDURAL_OPTIMIZATION TRUE)
endif()

```

Apply IPO flags without any checks. This may lead to build errors if IPO is not supported by the compiler in the current environment. Produce an error if CMake does not know IPO flags for the current compiler:

```

cmake_minimum_required(VERSION 3.9) # CMP0069 NEW
project(foo)

# ...

set_property(TARGET ... PROPERTY INTERPROCEDURAL_OPTIMIZATION TRUE)

```

CMP0068

Added in version 3.9.

RPATH settings on macOS do not affect **install_name**.

CMake 3.9 and newer remove any effect the following settings may have on the **install_name** of a target on macOS:

- *BUILD_WITH_INSTALL_RPATH* target property
- *SKIP_BUILD_RPATH* target property
- *CMAKE_SKIP_RPATH* variable
- *CMAKE_SKIP_INSTALL_RPATH* variable

Previously, setting *BUILD_WITH_INSTALL_RPATH* had the effect of setting both the **install_name** of a target to *INSTALL_NAME_DIR* and the **RPATH** to *INSTALL_RPATH*. In CMake 3.9, it only affects setting of **RPATH**. However, if one wants *INSTALL_NAME_DIR* to apply to the target in the build tree, one may set *BUILD_WITH_INSTALL_NAME_DIR*.

If *SKIP_BUILD_RPATH*, *CMAKE_SKIP_RPATH* or *CMAKE_SKIP_INSTALL_RPATH* were used to strip the directory portion of the **install_name** of a target, one may set *INSTALL_NAME_DIR=""* instead.

The **OLD** behavior of this policy is to use the **RPATH** settings for **install_name** on macOS. The **NEW** behavior of this policy is to ignore the **RPATH** settings for **install_name** on macOS.

This policy was introduced in CMake version 3.9. It may be set by *cmake_policy()* or *cmake_minimum_required()*. If it is not set, CMake warns, and uses **OLD** behavior.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.8**CMP0067**

Added in version 3.8.

Honor language standard in `try_compile()` source-file signature.

The `try_compile()` source file signature is intended to allow callers to check whether they will be able to compile a given source file with the current toolchain. In order to match compiler behavior, any language standard mode should match. However, CMake 3.7 and below did not do this. CMake 3.8 and above prefer to honor the language standard settings for **C**, **CXX** (C++), and **CUDA** using the values of the variables:

- `CMAKE_C_STANDARD`
- `CMAKE_C_STANDARD_REQUIRED`
- `CMAKE_C_EXTENSIONS`
- `CMAKE_CXX_STANDARD`
- `CMAKE_CXX_STANDARD_REQUIRED`
- `CMAKE_CXX_EXTENSIONS`
- `CMAKE_CUDA_STANDARD`
- `CMAKE_CUDA_STANDARD_REQUIRED`
- `CMAKE_CUDA_EXTENSIONS`

This policy provides compatibility for projects that do not expect the language standard settings to be used automatically.

The **OLD** behavior of this policy is to ignore language standard setting variables when generating the **try_compile** test project. The **NEW** behavior of this policy is to honor language standard setting variables.

This policy was introduced in CMake version 3.8. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn by default, and uses **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0067` variable to control the warning.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

Policies Introduced by CMake 3.7**CMP0066**

Added in version 3.7.

Honor per-config flags in `try_compile()` source-file signature.

The source file signature of the `try_compile()` command uses the value of the `CMAKE_<LANG>_FLAGS` variable in the test project so that the test compilation works as it would in the main project. However, CMake 3.6 and below do not also honor config-specific compiler flags such as those in the `CMAKE_<LANG>_FLAGS_DEBUG` variable. CMake 3.7 and above prefer to honor config-specific compiler flags too. This policy provides compatibility for projects that do not expect config-specific compiler flags to be used.

The **OLD** behavior of this policy is to ignore config-specific flag variables like

`CMAKE_<LANG>_FLAGS_DEBUG` and only use CMake's built-in defaults for the current compiler and platform.

The **NEW** behavior of this policy is to honor config-specific flag variables like `CMAKE_<LANG>_FLAGS_DEBUG`.

This policy was introduced in CMake version 3.7. It may be set by `cmake_policy()` or `cmake_minimum_required()`. If it is not set, CMake does *not* warn by default, and uses **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0066` variable to control the warning.

NOTE:

The **OLD** behavior of a policy is *deprecated by definition* and may be removed in a future version of CMake.

UNSUPPORTED POLICIES

The following policies are no longer supported. Projects' calls to `cmake_minimum_required(VERSION)` or `cmake_policy(VERSION)` must set them to **NEW**. Their **OLD** behaviors have been removed from CMake.

Policies Introduced by CMake 3.4, Removed by CMake 4.0

CMP0065

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Added in version 3.4.

Do not add flags to export symbols from executables without the `ENABLE_EXPORTS` target property.

CMake 3.3 and below, for historical reasons, always linked executables on some platforms with flags like `-rdynamic` to export symbols from the executables for use by any plugins they may load via `dlopen`. CMake 3.4 and above prefer to do this only for executables that are explicitly marked with the `ENABLE_EXPORTS` target property.

The **OLD** behavior of this policy is to always use the additional link flags when linking executables regardless of the value of the `ENABLE_EXPORTS` target property.

The **NEW** behavior of this policy is to only use the additional link flags when linking executables if the `ENABLE_EXPORTS` target property is set to **True**.

This policy was introduced in CMake version 3.4. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake did *not* warn by default, and used **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0065` variable to control the warning in CMake versions before 4.0.

CMP0064

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Added in version 3.4.

Recognize **TEST** as a operator for the *if()* command.

The **TEST** operator was added to the *if()* command to determine if a given test name was created by the *add_test()* command.

The **OLD** behavior for this policy is to ignore the **TEST** operator. The **NEW** behavior is to interpret the **TEST** operator.

This policy was introduced in CMake version 3.4. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

Policies Introduced by CMake 3.3, Removed by CMake 4.0

CMP0063

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Added in version 3.3.

Honor visibility properties for all target types.

The `<LANG>_VISIBILITY_PRESET` and `VISIBILITY_INLINES_HIDDEN` target properties affect visibility of symbols during dynamic linking. When first introduced these properties affected compilation of sources only in shared libraries, module libraries, and executables with the `ENABLE_EXPORTS` property set. This was sufficient for the basic use cases of shared libraries and executables with plugins. However, some sources may be compiled as part of static libraries or object libraries and then linked into a shared library later. CMake 3.3 and above prefer to honor these properties for sources compiled in all target types. This policy preserves compatibility for projects expecting the properties to work only for some target types.

The **OLD** behavior for this policy is to ignore the visibility properties for static libraries, object libraries, and executables without exports. The **NEW** behavior for this policy is to honor the visibility properties for all target types.

This policy was introduced in CMake version 3.3. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0062

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Added in version 3.3.

Disallow *install()* of *export()* result.

The *export()* command generates a file containing *Imported Targets*, which is suitable for use from the build directory. It is not suitable for installation because it contains absolute paths to buildsystem locations, and is particular to a single build configuration.

The *install(EXPORT)* generates and installs files which contain *Imported Targets*. These files are generated with relative paths (unless the user specifies absolute paths), and are designed for multi-configuration use. See *Creating Packages* for more.

CMake 3.3 no longer allows the use of the *install(FILE)* command with the result of the *export()* command.

The **OLD** behavior for this policy is to allow installing the result of an *export()* command. The **NEW** behavior for this policy is not to allow installing the result of an *export()* command.

This policy was introduced in CMake version 3.3. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0061

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Added in version 3.3.

CTest does not by default tell **make** to ignore errors (**-i**).

The *ctest_build()* and *build_command()* commands no longer generate build commands for *Makefile Generators* with the **-i** option. Previously this was done to help build as much of tested projects as possible. However, this behavior is not consistent with other generators and also causes the return code of the **make** tool to be meaningless.

Of course users may still add this option manually by setting *CTEST_BUILD_COMMAND* or the **MAKECOMMAND** cache entry. See the *CTest Build Step MakeCommand* setting documentation for their effects.

The **OLD** behavior for this policy is to add **-i** to **make** calls in CTest. The **NEW** behavior for this policy is to not add **-i**.

This policy was introduced in CMake version 3.3. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake did *not* warn, and used **OLD** behavior.

CMP0060

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Added in version 3.3.

Link libraries by full path even in implicit directories.

Policy *CMP0003* was introduced with the intention of always linking library files by full path when a full path is given to the *target_link_libraries()* command. However, on some platforms (e.g. HP-UX) the compiler front-end adds alternative library search paths for the current architecture (e.g. */usr/lib/<arch>* has alternatives to libraries in */usr/lib* for the current architecture). On such platforms the *find_library()* may find a library such as */usr/lib/libfoo.so* that does not belong to the current architecture.

Prior to policy *CMP0003* projects would still build in such cases because the incorrect library path would be converted to **-lfoo** on the link line and the linker would find the proper library in the arch-specific search path provided by the compiler front-end implicitly. At the time we chose to remain compatible with such projects by always converting library files found in implicit link directories to **-lfoo** flags to ask the

linker to search for them. This approach allowed existing projects to continue to build while still linking to libraries outside implicit link directories via full path (such as those in the build tree).

CMake does allow projects to override this behavior by using an *IMPORTED library target* with its *IMPORTED_LOCATION* property set to the desired full path to a library file. In fact, many *Find Modules* are learning to provide *Imported Targets* instead of just the traditional **Foo_LIBRARIES** variable listing library files. However, this makes the link line generated for a library found by a Find Module depend on whether it is linked through an imported target or not, which is inconsistent. Furthermore, this behavior has been a source of confusion because the generated link line for a library file depends on its location. It is also problematic for projects trying to link statically because flags like **-Wl,-Bstatic -lfoo -Wl,-Bdynamic** may be used to help the linker select **libfoo.a** instead of **libfoo.so** but then leak dynamic linking to following libraries. (See the *LINK_SEARCH_END_STATIC* target property for a solution typically used for that problem.)

When the special case for libraries in implicit link directories was first introduced the list of implicit link directories was simply hard-coded (e.g. **/lib**, **/usr/lib**, and a few others). Since that time, CMake has learned to detect the implicit link directories used by the compiler front-end. If necessary, the *find_library()* command could be taught to use this information to help find libraries of the proper architecture.

For these reasons, CMake 3.3 and above prefer to drop the special case and link libraries by full path even when they are in implicit link directories. Policy **CMP0060** provides compatibility for existing projects.

The **OLD** behavior for this policy is to ask the linker to search for libraries whose full paths are known to be in implicit link directories. The **NEW** behavior for this policy is to link libraries by full path even if they are in implicit link directories.

This policy was introduced in CMake version 3.3. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake did *not* warn by default, and used **OLD** behavior.

See documentation of the *CMAKE_POLICY_WARNING_CMP0060* variable to control the warning in CMake versions before 4.0.

CMP0059

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Added in version 3.3.

Do not treat **DEFINITIONS** as a built-in directory property.

CMake 3.3 and above no longer make a list of definitions available through the *DEFINITIONS* directory property. The *COMPILE_DEFINITIONS* directory property may be used instead.

The **OLD** behavior for this policy is to provide the list of flags given so far to the *add_definitions()* command. The **NEW** behavior is to behave as a normal user-defined directory property.

This policy was introduced in CMake version 3.3. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0058

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW**

by a call to `cmake_minimum_required()` or `cmake_policy()`.

Added in version 3.3.

Ninja requires custom command byproducts to be explicit.

When an intermediate file generated during the build is consumed by an expensive operation or a large tree of dependents, one may reduce the work needed for an incremental rebuild by updating the file timestamp only when its content changes. With this approach the generation rule must have a separate output file that is always updated with a new timestamp that is newer than any dependencies of the rule so that the build tool re-runs the rule only when the input changes. We refer to the separate output file as a rule's *witness* and the generated file as a rule's *byproduct*.

Byproducts may not be listed as outputs because their timestamps are allowed to be older than the inputs. No build tools (like **make**) that existed when CMake was designed have a way to express byproducts. Therefore CMake versions prior to 3.2 had no way to specify them. Projects typically left byproducts undeclared in the rules that generate them. For example:

```
add_custom_command(
  OUTPUT witness.txt
  COMMAND ${CMAKE_COMMAND} -E copy_if_different
           ${CMAKE_CURRENT_SOURCE_DIR}/input.txt
           byproduct.txt # timestamp may not change
  COMMAND ${CMAKE_COMMAND} -E touch witness.txt
  DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/input.txt
)
add_custom_target(Provider DEPENDS witness.txt)
add_custom_command(
  OUTPUT generated.c
  COMMAND expensive-task -i byproduct.txt -o generated.c
  DEPENDS ${CMAKE_CURRENT_BINARY_DIR}/byproduct.txt
)
add_library(Consumer generated.c)
add_dependencies(Consumer Provider)
```

This works well for all generators except *Ninja*. The Ninja build tool sees a rule listing **byproduct.txt** as a dependency and no rule listing it as an output. Ninja then complains that there is no way to satisfy the dependency and stops building even though there are order-only dependencies that ensure **byproduct.txt** will exist before its consumers need it. See discussion of this problem in *Ninja Issue 760* for further details on why Ninja works this way.

Instead of leaving byproducts undeclared in the rules that generate them, Ninja expects byproducts to be listed along with other outputs. Such rules may be marked with a **restat** option that tells Ninja to check the timestamps of outputs after the rules run. This prevents byproducts whose timestamps do not change from causing their dependents to re-build unnecessarily.

Since the above approach does not tell CMake what custom command generates **byproduct.txt**, the Ninja generator does not have enough information to add the byproduct as an output of any rule. CMake 2.8.12 and above work around this problem and allow projects using the above approach to build by generating **phony** build rules to tell Ninja to tolerate such missing files. However, this workaround prevents Ninja from diagnosing a dependency that is really missing. It also works poorly in in-source builds where every custom command dependency, even on source files, needs to be treated this way because CMake does not have enough information to know which files are generated as byproducts of custom commands.

Introducing Byproducts

CMake 3.2 introduced the **BYPRODUCTS** option to the `add_custom_command()` and `add_custom_target()` commands. This option allows byproducts to be specified explicitly:

```
add_custom_command(
  OUTPUT witness.txt
  BYPRODUCTS byproduct.txt # explicit byproduct specification
  COMMAND ${CMAKE_COMMAND} -E copy_if_different
           ${CMAKE_CURRENT_SOURCE_DIR}/input.txt
           byproduct.txt # timestamp may not change
  ...
```

The **BYPRODUCTS** option is used by the *Ninja* generator to list byproducts among the outputs of the custom commands that generate them, and is ignored by other generators.

CMake 3.3 and above prefer to require projects to specify custom command byproducts explicitly so that it can avoid using the **phony** rule workaround altogether. Policy **CMP0058** was introduced to provide compatibility with existing projects that still need the workaround.

This policy has no effect on generators other than *Ninja*. The **OLD** behavior for this policy is to generate *Ninja* **phony** rules for unknown dependencies in the build tree. The **NEW** behavior for this policy is to not generate these and instead require projects to specify custom command **BYPRODUCTS** explicitly.

The policy setting must be in scope at the end of the top-level **CMakeLists.txt** file of the project and has global effect.

This policy was introduced in CMake version 3.3. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned when it saw unknown dependencies in out-of-source build trees, and used **OLD** behavior.

CMP0057

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Added in version 3.3.

Support new *if()* IN_LIST operator.

CMake 3.3 adds support for the new IN_LIST operator.

The **OLD** behavior for this policy is to ignore the IN_LIST operator. The **NEW** behavior is to interpret the IN_LIST operator.

This policy was introduced in CMake version 3.3. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

Policies Introduced by CMake 3.2, Removed by CMake 4.0

CMP0056

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Added in version 3.2.

Honor link flags in *try_compile()* source-file signature.

The *try_compile()* command source-file signature generates a **CMakeLists.txt** file to build the source file into an executable. In order to compile the source the same way as it might be compiled by the calling project, the generated project sets the value of the *CMAKE_<LANG>_FLAGS* variable to that in the calling project. The value of the *CMAKE_EXE_LINKER_FLAGS* variable may be needed in some cases too, but CMake 3.1 and lower did not set it in the generated project. CMake 3.2 and above prefer to set it so that linker flags are honored as well as compiler flags. This policy provides compatibility with the pre-3.2 behavior.

The **OLD** behavior for this policy is to not set the value of the *CMAKE_EXE_LINKER_FLAGS* variable in the generated test project. The **NEW** behavior for this policy is to set the value of the *CMAKE_EXE_LINKER_FLAGS* variable in the test project to the same as it is in the calling project.

If the project code does not set the policy explicitly, users may set it on the command line by defining the *CMAKE_POLICY_DEFAULT_CMP0056* variable in the cache.

This policy was introduced in CMake version 3.2. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake did *not* warn by default, and used **OLD** behavior.

See documentation of the *CMAKE_POLICY_WARNING_CMP0056* variable to control the warning in CMake versions before 4.0.

CMP0055

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Added in version 3.2.

Strict checking for the *break()* command.

CMake 3.1 and lower allowed calls to the *break()* command outside of a loop context and also ignored any given arguments. This was undefined behavior.

The **OLD** behavior for this policy is to allow *break()* to be placed outside of loop contexts and ignores any arguments. The **NEW** behavior for this policy is to issue an error if a misplaced break or any arguments are found.

This policy was introduced in CMake version 3.2. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

Policies Introduced by CMake 3.1, Removed by CMake 4.0

CMP0054

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Added in version 3.1.

Only interpret *if()* arguments as variables or keywords when unquoted.

CMake 3.1 and above no longer implicitly dereference variables or interpret keywords in an *if()* command argument when it is a *Quoted Argument* or a *Bracket Argument*.

The **OLD** behavior for this policy is to dereference variables and interpret keywords even if they are quoted or bracketed. The **NEW** behavior is to not dereference variables or interpret keywords that have been quoted or bracketed.

Given the following partial example:

```
set(A E)
set(E "")

if("${A}" STREQUAL "")
    message("Result is TRUE before CMake 3.1 or when CMP0054 is OLD")
else()
    message("Result is FALSE in CMake 3.1 and above if CMP0054 is NEW")
endif()
```

After explicit expansion of variables this gives:

```
if("E" STREQUAL "")
```

With the policy set to **OLD** implicit expansion reduces this semantically to:

```
if("" STREQUAL "")
```

With the policy set to **NEW** the quoted arguments will not be further dereferenced:

```
if("E" STREQUAL "")
```

This policy was introduced in CMake version 3.1. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0053

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Added in version 3.1.

Simplify variable reference and escape sequence evaluation.

CMake 3.1 introduced a much faster implementation of evaluation of the *Variable References* and *Escape Sequences* documented in the *cmake-language(7)* manual. While the behavior is identical to the legacy implementation in most cases, some corner cases were cleaned up to simplify the behavior. Specifically:

- Expansion of **@VAR@** reference syntax defined by the *configure_file()* and *string(CONFIGURE)* commands is no longer performed in other contexts.
- Literal **\${VAR}** reference syntax may contain only alphanumeric characters (**A-Z**, **a-z**, **0-9**) and the characters **_**, **.**, **/**, **-**, and **+**. Note that **\$** is technically allowed in the **NEW** behavior, but is invalid for **OLD** behavior. This is due to an oversight during the implementation of **CMP0053** and its use as a literal variable reference is discouraged for this reason. Variables with other characters in their name may still be referenced indirectly, e.g.

```
set(varname "otherwise & disallowed $ characters")
message("${${varname}}")
```

- The setting of policy *CMP0010* is not considered, so improper variable reference syntax is always an error.
- More characters are allowed to be escaped in variable names. Previously, only `()#\"@^` were valid characters to escape. Now any non-alphanumeric, non-semicolon, non-NUL character may be escaped following the **escape_identity** production in the *Escape Sequences* section of the *cmake-language(7)* manual.

The **OLD** behavior for this policy is to honor the legacy behavior for variable references and escape sequences. The **NEW** behavior is to use the simpler variable expansion and escape sequence evaluation rules.

This policy was introduced in CMake version 3.1. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0052

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Added in version 3.1.

Reject source and build dirs in installed *INTERFACE_INCLUDE_DIRECTORIES*.

CMake 3.0 and lower allowed subdirectories of the source directory or build directory to be in the *INTERFACE_INCLUDE_DIRECTORIES* of installed and exported targets, if the directory was also a subdirectory of the installation prefix. This makes the installation depend on the existence of the source dir or binary dir, and the installation will be broken if either are removed after installation.

See *Include Directories and Usage Requirements* for more on specifying include directories for targets.

The **OLD** behavior for this policy is to export the content of the *INTERFACE_INCLUDE_DIRECTORIES* with the source or binary directory. The **NEW** behavior for this policy is to issue an error if such a directory is used.

This policy was introduced in CMake version 3.1. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0051

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Added in version 3.1.

List *TARGET_OBJECTS* in *SOURCES* target property.

CMake 3.0 and lower did not include the **TARGET_OBJECTS** *generator expression* when returning the *SOURCES* target property.

Configure-time CMake code is not able to handle generator expressions. If using the *SOURCES* target property at configure time, it may be necessary to first remove generator expressions using the *string(GENEX_STRIP)* command. Generate-time CMake code such as *file(GENERATE)* can handle the content without stripping.

The **OLD** behavior for this policy is to omit **TARGET_OBJECTS** expressions from the *SOURCES* target property. The **NEW** behavior for this policy is to include **TARGET_OBJECTS** expressions in the output.

This policy was introduced in CMake version 3.1. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

Policies Introduced by CMake 3.0, Removed by CMake 4.0

CMP0050

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Disallow *add_custom_command* SOURCE signatures.

CMake 2.8.12 and lower allowed a signature for *add_custom_command()* which specified an input to a command. This was undocumented behavior. Modern use of CMake associates custom commands with their output, rather than their input.

The **OLD** behavior for this policy is to allow the use of *add_custom_command()* SOURCE signatures. The **NEW** behavior for this policy is to issue an error if such a signature is used.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0049

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Do not expand variables in target source entries.

CMake 2.8.12 and lower performed an extra layer of variable expansion when evaluating source file names:

```
set(a_source foo.c)
add_executable(foo ${a_source})
```

This was undocumented behavior.

The **OLD** behavior for this policy is to expand such variables when processing the target sources. The **NEW** behavior for this policy is to issue an error if such variables need to be expanded.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0048

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

The *project()* command manages **VERSION** variables.

CMake version 3.0 introduced the **VERSION** option of the *project()* command to specify a project version as well as the name. In order to keep *PROJECT_VERSION* and related variables consistent with variable *PROJECT_NAME* it is necessary to set the **VERSION** variables to the empty string when no **VERSION** is given to *project()*. However, this can change behavior for existing projects that set **VERSION** variables themselves since *project()* may now clear them. This policy controls the behavior for compatibility with such projects.

The **OLD** behavior for this policy is to leave **VERSION** variables untouched. The **NEW** behavior for this policy is to set **VERSION** as documented by the *project()* command.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0047

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Use **QCC** compiler id for the qcc drivers on QNX.

CMake 3.0 and above recognize that the QNX qcc compiler driver is different from the GNU compiler. CMake now prefers to present this to projects by setting the *CMAKE_<LANG>_COMPILER_ID* variable to **QCC** instead of **GNU**. However, existing projects may assume the compiler id for QNX qcc is just **GNU** as it was in CMake versions prior to 3.0. Therefore this policy determines for QNX qcc which compiler id to report in the *CMAKE_<LANG>_COMPILER_ID* variable after language *<LANG>* is enabled by the *project()* or *enable_language()* command. The policy must be set prior to the invocation of either command.

The **OLD** behavior for this policy is to use the **GNU** compiler id for the qcc and QCC compiler drivers. The **NEW** behavior for this policy is to use the **QCC** compiler id for those drivers.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake did *not* warn by default, and used **OLD** behavior.

See documentation of the *CMAKE_POLICY_WARNING_CMP0047* variable to control the warning in CMake versions before 4.0.

CMP0046

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Error on non-existent dependency in *add_dependencies*.

CMake 2.8.12 and lower silently ignored non-existent dependencies listed in the *add_dependencies()* command.

The **OLD** behavior for this policy is to silently ignore non-existent dependencies. The **NEW** behavior for this policy is to report an error if non-existent dependencies are listed in the *add_dependencies()* command.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD**

behavior.

CMP0045

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Error on non-existent target in `get_target_property`.

In CMake 2.8.12 and lower, the `get_target_property()` command accepted a non-existent target argument without issuing any error or warning. The result variable is set to a **NOTFOUND** value.

The **OLD** behavior for this policy is to issue no warning and set the result variable to a **NOTFOUND** value. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** if the command is called with a non-existent target.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0044

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Case sensitive `<LANG>_COMPILER_ID` generator expressions

CMake 2.8.12 introduced the `<LANG>_COMPILER_ID generator expressions` to allow comparison of the `CMAKE_<LANG>_COMPILER_ID` with a test value. The possible valid values are lowercase, but the comparison with the test value was performed case-insensitively.

The **OLD** behavior for this policy is to perform a case-insensitive comparison with the value in the `<LANG>_COMPILER_ID` expression. The **NEW** behavior for this policy is to perform a case-sensitive comparison with the value in the `<LANG>_COMPILER_ID` expression.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0043

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Ignore `COMPILE_DEFINITIONS_<Config>` properties

CMake 2.8.12 and lower allowed setting the `COMPILE_DEFINITIONS_<CONFIG>` target property and `COMPILE_DEFINITIONS_<CONFIG>` directory property to apply configuration-specific compile definitions.

Since CMake 2.8.10, the `COMPILE_DEFINITIONS` property has supported *generator expressions* for setting configuration-dependent content. The continued existence of the suffixed variables is redundant, and causes a maintenance burden. Population of the `COMPILE_DEFINITIONS_DEB` UG property may be replaced with a population of `COMPILE_DEFINITIONS` directly or via `target_compile_definitions()`:

```
# Old Interfaces:
```

```

    set_property(TARGET tgt APPEND PROPERTY
        COMPILE_DEFINITIONS_DEBUG DEBUG_MODE
    )
    set_property(DIRECTORY APPEND PROPERTY
        COMPILE_DEFINITIONS_DEBUG DIR_DEBUG_MODE
    )

    # New Interfaces:
    set_property(TARGET tgt APPEND PROPERTY
        COMPILE_DEFINITIONS $$<CONFIG:Debug>:DEBUG_MODE>
    )
    target_compile_definitions(tgt PRIVATE $$<CONFIG:Debug>:DEBUG_MODE>)
    set_property(DIRECTORY APPEND PROPERTY
        COMPILE_DEFINITIONS $$<CONFIG:Debug>:DIR_DEBUG_MODE>
    )

```

The **OLD** behavior for this policy is to consume the content of the suffixed `COMPILE_DEFINITIONS_<CONFIG>` target property when generating the compilation command. The **NEW** behavior for this policy is to ignore the content of the `COMPILE_DEFINITIONS_<CONFIG>` target property .

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0042

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

`MACOSX_RPATH` is enabled by default.

CMake 2.8.12 and newer has support for using `@rpath` in a target's install name. This was enabled by setting the target property `MACOSX_RPATH`. The `@r path` in an install name is a more flexible and powerful mechanism than `@executable_path` or `@loader_path` for locating shared libraries.

CMake 3.0 and later prefer this property to be ON by default. Projects wanting `@rpath` in a target's install name may remove any setting of the `INSTALL_NAME_DIR` and `CMAKE_INSTALL_NAME_DIR` variables.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0041

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Error on relative include with generator expression.

Diagnostics in CMake 2.8.12 and lower silently ignored an entry in the `INTERFACE_INCLUDE_DIRECTORIES` of a target if it contained a generator expression at any position.

The path entries in that target property should not be relative. High-level API should ensure that by adding either a source directory or a install directory prefix, as appropriate.

As an additional diagnostic, the *INTERFACE_INCLUDE_DIRECTORIES* generated on an *IMPORTED* target for the install location should not contain paths in the source directory or the build directory.

The **OLD** behavior for this policy is to ignore relative path entries if they contain a generator expression. The **NEW** behavior for this policy is to report an error if a generator expression appears in another location and the path is relative.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0040

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

The target in the **TARGET** signature of *add_custom_command()* must exist and must be defined in the current directory.

CMake 2.8.12 and lower silently ignored a custom command created with the **TARGET** signature of *add_custom_command()* if the target is unknown or was defined outside the current directory.

The **OLD** behavior for this policy is to ignore custom commands for unknown targets. The **NEW** behavior for this policy is to report an error if the target referenced in *add_custom_command()* is unknown or was defined outside the current directory.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0039

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Utility targets may not have link dependencies.

CMake 2.8.12 and lower allowed using utility targets in the left hand side position of the *target_link_libraries()* command. This is an indicator of a bug in user code.

The **OLD** behavior for this policy is to ignore attempts to set the link libraries of utility targets. The **NEW** behavior for this policy is to report an error if an attempt is made to set the link libraries of a utility target.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0038

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Targets may not link directly to themselves.

CMake 2.8.12 and lower allowed a build target to link to itself directly with a *target_link_libraries()* call. This is an indicator of a bug in user code.

The **OLD** behavior for this policy is to ignore targets which list themselves in their own link implementation. The **NEW** behavior for this policy is to report an error if a target attempts to link to itself.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0037

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Target names should not be reserved and should match a validity pattern.

CMake 2.8.12 and lower allowed creating targets using `add_library()`, `add_executable()` and `add_custom_target()` with unrestricted choice for the target name. Newer cmake features such as `cmake-generator-expressions(7)` and some diagnostics expect target names to match a restricted pattern.

Target names may contain upper and lower case letters, numbers, the underscore character (`_`), dot (`.`), plus (`+`) and minus (`-`). As a special case, **ALIAS** and **IMPORTED** targets may contain two consecutive colons.

Target names reserved by one or more CMake generators are not allowed. Among others these include **all**, **clean**, **help**, and **install**.

Target names associated with optional features, such as **test** and **package**, may also be reserved. CMake 3.10 and below always reserve them. CMake 3.11 and above reserve them only when the corresponding feature is enabled (e.g. by including the `CTest` or `CPack` modules).

The **OLD** behavior for this policy is to allow creating targets with reserved names or which do not match the validity pattern. The **NEW** behavior for this policy is to report an error if an `add_*` command is used with an invalid target name.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0036

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

The `build_name()` command should not be called.

This command was added in May 2001 to compute a name for the current operating system and compiler combination. The command has long been documented as discouraged and replaced by the `CMAKE_SYSTEM` and `CMAKE_<LANG>_COMPILER` variables.

CMake $\geq$ 3.0 prefer that this command never be called. The **OLD** behavior for this policy is to allow the command to be called. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** when the command is called.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0035**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

The *variable_requires()* command should not be called.

This command was introduced in November 2001 to perform some conditional logic. It has long been replaced by the *if()* command.

CMake >= 3.0 prefer that this command never be called. The **OLD** behavior for this policy is to allow the command to be called. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** when the command is called.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0034**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

The *utility_source()* command should not be called.

This command was introduced in March 2001 to help build executables used to generate other files. This approach has long been replaced by *add_executable()* combined with *add_custom_command()*.

CMake >= 3.0 prefer that this command never be called. The **OLD** behavior for this policy is to allow the command to be called. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** when the command is called.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0033**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

The *export_library_dependencies()* command should not be called.

This command was added in January 2003 to export <tgt>_LIB_DEPENDS internal CMake cache entries to a file for installation with a project. This was used at the time to allow transitive link dependencies to work for applications outside of the original build tree of a project. The functionality has been superseded by the *export()* and *install(EXPORT)* commands.

CMake >= 3.0 prefer that this command never be called. The **OLD** behavior for this policy is to allow the command to be called. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** when the command is called.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0032**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

The *output_required_files()* command should not be called.

This command was added in June 2001 to expose the then-current CMake implicit dependency scanner. CMake's real implicit dependency scanner has evolved since then but is not exposed through this command. The scanning capabilities of this command are very limited and this functionality is better achieved through dedicated outside tools.

CMake $\geq$ 3.0 prefer that this command never be called. The **OLD** behavior for this policy is to allow the command to be called. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** when the command is called.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0031**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

The *load_command()* command should not be called.

This command was added in August 2002 to allow projects to add arbitrary commands implemented in C or C++. However, it does not work when the toolchain in use does not match the ABI of the CMake process. It has been mostly superseded by *themacro()* and *function()* commands.

CMake $\geq$ 3.0 prefer that this command never be called. The **OLD** behavior for this policy is to allow the command to be called. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** when the command is called.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0030**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

The *use_mangled_mesa()* command should not be called.

This command was created in September 2001 to support VTK before modern CMake language and custom command capabilities. VTK has not used it in years.

CMake $\geq$ 3.0 prefer that this command never be called. The **OLD** behavior for this policy is to allow the command to be called. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** when the command is called.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0029**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

The `subdir_depends()` command should not be called.

The implementation of this command has been empty since December 2001 but was kept in CMake for compatibility for a long time.

CMake $\geq$ 3.0 prefer that this command never be called. The **OLD** behavior for this policy is to allow the command to be called. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** when the command is called.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0028**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Double colon in target name means **ALIAS** or **IMPORTED** target.

CMake 2.8.12 and lower allowed the use of targets and files with double colons in `target_link_libraries()`, with some buildsystem generators.

The use of double-colons is a common pattern used to namespace **IMPORTED** targets and **ALIAS** targets. When computing the link dependencies of a target, the name of each dependency could either be a target, or a file on disk. Previously, if a target was not found with a matching name, the name was considered to refer to a file on disk. This can lead to confusing error messages if there is a typo in what should be a target name.

See also the `LINK_LIBRARIES_ONLY_TARGETS` target property.

The **OLD** behavior for this policy is to search for targets, then files on disk, even if the search term contains double-colons. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** if a link dependency contains double-colons but is not an **IMPORTED** target or an **ALIAS** target.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0027**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Conditionally linked imported targets with missing include directories.

CMake 2.8.11 introduced the concept of `INTERFACE_INCLUDE_DIRECTORIES`, and a check at cmake time that the entries in the `INTERFACE_INCLUDE_DIRECTORIES` of an **IMPORTED** target actually exist. CMake 2.8.11 also introduced generator expression support in the `target_link_libraries()` command. However, if an imported target is linked as a result of a generator expression evaluation, the entries in the `INTERFACE_INCLUDE_DIRECTORIES` of that target were not checked for existence as they should

be.

The **OLD** behavior of this policy is to report a warning if an entry in the *INTERFACE_INCLUDE_DIRECTORIES* of a generator-expression conditionally linked **IMPORTED** target does not exist.

The **NEW** behavior of this policy is to report an error if an entry in the *INTERFACE_INCLUDE_DIRECTORIES* of a generator-expression conditionally linked **IMPORTED** target does not exist.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0026

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Disallow use of the *LOCATION* property for build targets.

CMake 2.8.12 and lower allowed reading the *LOCATION* target property (and configuration-specific variants) to determine the eventual location of build targets. This relies on the assumption that all necessary information is available at configure-time to determine the final location and filename of the target. However, this property is not fully determined until later at generate-time. At generate time, the *\$<TARGET_FILE>* generator expression can be used to determine the eventual *LOCATION* of a target output.

Code which reads the *LOCATION* target property can be ported to use the *\$<TARGET_FILE>* generator expression together with the *file(GENERATE)* subcommand to generate a file containing the target location.

The **OLD** behavior for this policy is to allow reading the *LOCATION* properties from build-targets. The **NEW** behavior for this policy is to not allow reading the *LOCATION* properties from build-targets.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0025

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Compiler id for Apple Clang is now **AppleClang**.

CMake 3.0 and above recognize that Apple Clang is a different compiler than upstream Clang and that they have different version numbers. CMake now prefers to present this to projects by setting the *CMAKE_<LANG>_COMPILER_ID* variable to **AppleClang** instead of **Clang**. However, existing projects may assume the compiler id for Apple Clang is just **Clang** as it was in CMake versions prior to 3.0. Therefore this policy determines for Apple Clang which compiler id to report in the *CMAKE_<LANG>_COMPILER_ID* variable after language *<LANG>* is enabled by the *project()* or *enable_language()* command. The policy must be set prior to the invocation of either command.

The **OLD** behavior for this policy is to use compiler id **Clang**. The **NEW** behavior for this policy is to use compiler id **AppleClang**.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake did *not* warn by default, and used **OLD** behavior.

See documentation of the `CMAKE_POLICY_WARNING_CMP0025` variable to control the warning in CMake versions before 4.0.

CMP0024

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Disallow include export result.

CMake 2.8.12 and lower allowed use of the `include()` command with the result of the `export()` command. This relies on the assumption that the `export()` command has an immediate effect at configure-time during a cmake run. Certain properties of targets are not fully determined until later at generate-time, such as the link language and complete list of link libraries. Future refactoring will change the effect of the `export()` command to be executed at generate-time. Use **ALIAS** targets instead in cases where the goal is to refer to targets by another name.

The **OLD** behavior for this policy is to allow including the result of an `export()` command. The **NEW** behavior for this policy is not to allow including the result of an `export()` command.

This policy was introduced in CMake version 3.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

Policies Introduced by CMake 2.8, Removed by CMake 4.0

CMP0023

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Plain and keyword `target_link_libraries()` signatures cannot be mixed.

CMake 2.8.12 introduced the `target_link_libraries()` signature using the **PUBLIC**, **PRIVATE**, and **INTERFACE** keywords to generalize the **LINK_PUBLIC** and **LINK_PRIVATE** keywords introduced in CMake 2.8.7. Use of signatures with any of these keywords sets the link interface of a target explicitly, even if empty. This produces confusing behavior when used in combination with the historical behavior of the plain `target_link_libraries()` signature. For example, consider the code:

```
target_link_libraries(mylib A)
target_link_libraries(mylib PRIVATE B)
```

After the first line the link interface has not been set explicitly so CMake would use the link implementation, A, as the link interface. However, the second line sets the link interface to empty. In order to avoid this subtle behavior CMake now prefers to disallow mixing the plain and keyword signatures of `target_link_libraries()` for a single target.

The **OLD** behavior for this policy is to allow keyword and plain `target_link_libraries()` signatures to be mixed. The **NEW** behavior for this policy is to not allow mixing of the keyword and plain signatures.

This policy was introduced in CMake version 2.8.12. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0022**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

`INTERFACE_LINK_LIBRARIES` defines the link interface.

CMake 2.8.11 constructed the 'link interface' of a target from properties matching **(IMPORTED_)?LINK_INTERFACE_LIBRARIES(<CONFIG>)?**. The modern way to specify config-sensitive content is to use generator expressions and the **IMPORTED_** prefix makes uniform processing of the link interface with generator expressions impossible. The `INTERFACE_LINK_LIBRARIES` target property was introduced as a replacement in CMake 2.8.12. This new property is named consistently with the `INTERFACE_COMPILE_DEFINITIONS`, `INTERFACE_INCLUDE_DIRECTORIES` and `INTERFACE_COMPILE_OPTIONS` properties. For in-build targets, CMake will use the `INTERFACE_LINK_LIBRARIES` property as the source of the link interface only if policy **CMP0022** is **NEW**. When exporting a target which has this policy set to **NEW**, only the `INTERFACE_LINK_LIBRARIES` property will be processed and generated for the **IMPORTED** target by default. A new option to the `install(EXPORT)` and export commands allows export of the old-style properties for compatibility with downstream users of CMake versions older than 2.8.12. The `target_link_libraries()` command will no longer populate the properties matching **LINK_INTERFACE_LIBRARIES(<CONFIG>)?** if this policy is **NEW**.

Warning-free future-compatible code which works with CMake 2.8.7 onwards can be written by using the **LINK_PRIVATE** and **LINK_PUBLIC** keywords of `target_link_libraries()`.

The **OLD** behavior for this policy is to ignore the `INTERFACE_LINK_LIBRARIES` property for in-build targets. The **NEW** behavior for this policy is to use the `INTERFACE_LINK_LIBRARIES` property for in-build targets, and ignore the old properties matching **(IMPORTED_)?LINK_INTERFACE_LIBRARIES(<CONFIG>)?**.

This policy was introduced in CMake version 2.8.12. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0021**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Fatal error on relative paths in `INCLUDE_DIRECTORIES` target property.

CMake 2.8.10.2 and lower allowed the `INCLUDE_DIRECTORIES` target property to contain relative paths. The base path for such relative entries is not well defined. CMake 2.8.12 issues a **FATAL_ERROR** if the `INCLUDE_DIRECTORIES` property contains a relative path.

The **OLD** behavior for this policy is not to warn about relative paths in the `INCLUDE_DIRECTORIES` target property. The **NEW** behavior for this policy is to issue a **FATAL_ERROR** if `INCLUDE_DIRECTORIES` contains a relative path.

This policy was introduced in CMake version 2.8.12. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0020**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW**

by a call to *cmake_minimum_required()* or *cmake_policy()*.

Automatically link Qt executables to **qtmain** target on Windows.

CMake 2.8.10 and lower required users of Qt to always specify a link dependency to the **qtmain.lib** static library manually on Windows. CMake 2.8.11 gained the ability to evaluate generator expressions while determining the link dependencies from **IMPORTED** targets. This allows CMake itself to automatically link executables which link to Qt to the **qtmain.lib** library when using **IMPORTED** Qt targets. For applications already linking to **qtmain.lib**, this should have little impact. For applications which supply their own alternative WinMain implementation and for applications which use the QAxServer library, this automatic linking will need to be disabled as per the documentation.

The **OLD** behavior for this policy is not to link executables to **qtmain.lib** automatically when they link to the QtCore **IMPORTED** target. The **NEW** behavior for this policy is to link executables to **qtmain.lib** automatically when they link to QtCore **IMPORTED** target.

This policy was introduced in CMake version 2.8.11. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0019

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Do not re-expand variables in include and link information.

CMake 2.8.10 and lower re-evaluated values given to the *include_directories*, *link_directories*, and *link_libraries* commands to expand any leftover variable references at the end of the configuration step. This was for strict compatibility with VERY early CMake versions because all variable references are now normally evaluated during CMake language processing. CMake 2.8.11 and higher prefer to skip the extra evaluation.

The **OLD** behavior for this policy is to re-evaluate the values for strict compatibility. The **NEW** behavior for this policy is to leave the values untouched.

This policy was introduced in CMake version 2.8.11. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0018

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Ignore **CMAKE_SHARED_LIBRARY_<Lang>_FLAGS** variable.

CMake 2.8.8 and lower compiled sources in **SHARED** and **MODULE** libraries using the value of the undocumented **CMAKE_SHARED_LIBRARY_<Lang>_FLAGS** platform variable. The variable contained platform-specific flags needed to compile objects for shared libraries. Typically it included a flag such as **-fPIC** for position independent code but also included other flags needed on certain platforms. CMake 2.8.9 and higher prefer instead to use the *POSITION_INDEPENDENT_CODE* target property to determine what targets should be position independent, and new undocumented platform variables to select flags while ignoring **CMAKE_SHARED_LIBRARY_<Lang>_FLAGS** completely.

The default for either approach produces identical compilation flags, but if a project modifies **CMAKE_SHARED_LIBRARY_<Lang>_FLAGS** from its original value this policy determines which

approach to use.

The **OLD** behavior for this policy is to ignore the *POSITION_INDEPENDENT_CODE* property for all targets and use the modified value of *CMAKE_SHARED_LIBRARY_<Lang>_FLAGS* for **SHARED** and **MODULE** libraries.

The **NEW** behavior for this policy is to ignore *CMAKE_SHARED_LIBRARY_<Lang>_FLAGS* whether it is modified or not and honor the *POSITION_INDEPENDENT_CODE* target property.

This policy was introduced in CMake version 2.8.9. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0017

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Prefer files from the CMake module directory when including from there.

Starting with CMake 2.8.4, if a *cmake-module* shipped with CMake (i.e. located in the CMake module directory) calls *include()* or *find_package()*, the files located in the CMake module directory are preferred over the files in *CMAKE_MODULE_PATH*. This makes sure that the modules belonging to CMake always get those files included which they expect, and against which they were developed and tested. In all other cases, the files found in *CMAKE_MODULE_PATH* still take precedence over the ones in the CMake module directory. The **OLD** behavior is to always prefer files from *CMAKE_MODULE_PATH* over files from the CMake modules directory.

This policy was introduced in CMake version 2.8.4. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0016

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

target_link_libraries() reports error if its only argument is not a target.

In CMake 2.8.2 and lower the *target_link_libraries()* command silently ignored if it was called with only one argument, and this argument wasn't a valid target. In CMake 2.8.3 and above it reports an error in this case.

This policy was introduced in CMake version 2.8.3. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0015

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

link_directories() treats paths relative to the source dir.

In CMake 2.8.0 and lower the *link_directories()* command passed relative paths unchanged to the linker. In CMake 2.8.1 and above the *link_directories()* command prefers to interpret relative paths with respect to

CMAKE_CURRENT_SOURCE_DIR, which is consistent with *include_directories()* and other commands. The **OLD** behavior for this policy is to use relative paths verbatim in the linker command. The **NEW** behavior for this policy is to convert relative paths to absolute paths by appending the relative path to **CMAKE_CURRENT_SOURCE_DIR**.

This policy was introduced in CMake version 2.8.1. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0014

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Input directories must have **CMakeLists.txt**.

CMake versions before 2.8 silently ignored missing **CMakeLists.txt** files in directories referenced by *add_subdirectory()* or *subdirs()*, treating them as if present but empty. In CMake 2.8.0 and above this *cmake_policy()* determines whether or not the case is an error. The **OLD** behavior for this policy is to silently ignore the problem. The **NEW** behavior for this policy is to report an error.

This policy was introduced in CMake version 2.8.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0013

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

Duplicate binary directories are not allowed.

CMake 2.6.3 and below silently permitted *add_subdirectory()* calls to create the same binary directory multiple times. During build system generation files would be written and then overwritten in the build tree and could lead to strange behavior. CMake 2.6.4 and above explicitly detect duplicate binary directories. CMake 2.6.4 always considers this case an error. In CMake 2.8.0 and above this policy determines whether or not the case is an error. The **OLD** behavior for this policy is to allow duplicate binary directories. The **NEW** behavior for this policy is to disallow duplicate binary directories with an error.

This policy was introduced in CMake version 2.8.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0012

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

if() recognizes numbers and boolean constants.

In CMake versions 2.6.4 and lower the *if()* command implicitly dereferenced arguments corresponding to variables, even those named like numbers or boolean constants, except for **0** and **1**. Numbers and boolean constants such as **true**, **false**, **yes**, **no**, **on**, **off**, **y**, **n**, **notfound**, **ignore** (all case insensitive) were recognized in some cases but not all. For example, the code **if(TRUE)** might have evaluated as **false**. Numbers such as 2 were recognized only in boolean expressions like **if(NOT 2)** (leading to **false**) but not as a single-argument like **if(2)** (also leading to **false**). Later versions of CMake prefer to treat numbers and boolean

constants literally, so they should not be used as variable names.

The **OLD** behavior for this policy is to implicitly dereference variables named like numbers and boolean constants. The **NEW** behavior for this policy is to recognize numbers and boolean constants without dereferencing variables with such names.

This policy was introduced in CMake version 2.8.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

Policies Introduced by CMake 2.6, Removed by CMake 4.0

CMP0011

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Included scripts do automatic `cmake_policy()` PUSH and POP.

In CMake 2.6.2 and below, CMake Policy settings in scripts loaded by the `include()` and `find_package()` commands would affect the includer. Explicit invocations of **cmake_policy(PUSH)** and **cmake_policy(POP)** were required to isolate policy changes and protect the includer. While some scripts intend to affect the policies of their includer, most do not. In CMake 2.6.3 and above, `include()` and `find_package()` by default **PUSH** and **POP** an entry on the policy stack around an included script, but provide a **NO_POLICY_SCOPE** option to disable it. This policy determines whether or not to imply **NO_POLICY_SCOPE** for compatibility. The **OLD** behavior for this policy is to imply **NO_POLICY_SCOPE** for `include()` and `find_package()` commands. The **NEW** behavior for this policy is to allow the commands to do their default `cmake_policy` **PUSH** and **POP**.

This policy was introduced in CMake version 2.6.3. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0010

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Bad variable reference syntax is an error.

In CMake 2.6.2 and below, incorrect variable reference syntax such as a missing close-brace (**`${FOO}`**) was reported but did not stop processing of CMake code. This policy determines whether a bad variable reference is an error. The **OLD** behavior for this policy is to warn about the error, leave the string untouched, and continue. The **NEW** behavior for this policy is to report an error.

If `CMP0053` is set to **NEW**, this policy has no effect and is treated as always being **NEW**.

This policy was introduced in CMake version 2.6.3. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0009

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

FILE `GLOB_RECURSE` calls should not follow symlinks by default.

In CMake 2.6.1 and below, `file(GLOB_RECURSE)` calls would follow through symlinks, sometimes coming up with unexpectedly large result sets because of symlinks to top level directories that contain hundreds of thousands of files.

This policy determines whether or not to follow symlinks encountered during a `file(GLOB_RECURSE)` call. The **OLD** behavior for this policy is to follow the symlinks. The **NEW** behavior for this policy is not to follow the symlinks by default, but only if **FOLLOW_SYMLINKS** is given as an additional argument to the **FILE** command.

This policy was introduced in CMake version 2.6.2. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0008

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Libraries linked by full-path must have a valid library file name.

In CMake 2.4 and below it is possible to write code like

```
target_link_libraries(myexe /full/path/to/somelib)
```

where **somelib** is supposed to be a valid library file name such as **libsomelib.a** or **somelib.lib**. For Makefile generators this produces an error at build time because the dependency on the full path cannot be found. For *Visual Studio Generators* IDE and *Xcode* generators this used to work by accident because CMake would always split off the library directory and ask the linker to search for the library by name (**-lsomelib** or **somelib.lib**). Despite the failure with Makefiles, some projects have code like this and build only with Visual Studio and/or Xcode. This version of CMake prefers to pass the full path directly to the native build tool, which will fail in this case because it does not name a valid library file.

This policy determines what to do with full paths that do not appear to name a valid library file. The **OLD** behavior for this policy is to split the library name from the path and ask the linker to search for it. The **NEW** behavior for this policy is to trust the given path and pass it directly to the native build tool unchanged.

This policy was introduced in CMake version 2.6.1. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0007

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

list command no longer ignores empty elements.

This policy determines whether the list command will ignore empty elements in the list. CMake 2.4 and below list commands ignored all empty elements in the list. For example, **a;b;;c** would have length 3 and not 4. The **OLD** behavior for this policy is to ignore empty list elements. The **NEW** behavior for this policy is to correctly count empty elements in a list.

This policy was introduced in CMake version 2.6.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0006**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Installing `MACOSX_BUNDLE` targets requires a **BUNDLE DESTINATION**.

This policy determines whether the `install(TARGETS)` command must be given a **BUNDLE DESTINATION** when asked to install a target with the `MACOSX_BUNDLE` property set. CMake 2.4 and below did not distinguish application bundles from normal executables when installing targets. CMake 2.6 provides a **BUNDLE** option to the `install(TARGETS)` command that specifies rules specific to application bundles on the Mac. Projects should use this option when installing a target with the `MACOSX_BUNDLE` property set.

The **OLD** behavior for this policy is to fall back to the **RUNTIME DESTINATION** if a **BUNDLE DESTINATION** is not given. The **NEW** behavior for this policy is to produce an error if a bundle target is installed without a **BUNDLE DESTINATION**.

This policy was introduced in CMake version 2.6.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0005**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Preprocessor definition values are now escaped automatically.

This policy determines whether or not CMake should generate escaped preprocessor definition values added via `add_definitions`. CMake versions 2.4 and below assumed that only trivial values would be given for macros in `add_definitions` calls. It did not attempt to escape non-trivial values such as string literals in generated build rules. CMake versions 2.6 and above support escaping of most values, but cannot assume the user has not added escapes already in an attempt to work around limitations in earlier versions.

The **OLD** behavior for this policy is to place definition values given to `add_definitions` directly in the generated build rules without attempting to escape anything. The **NEW** behavior for this policy is to generate correct escapes for all native build tools automatically. See documentation of the **COMPILE_DEFINITIONS** target property for limitations of the escaping implementation.

This policy was introduced in CMake version 2.6.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0004**WARNING:**

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Libraries linked may not have leading or trailing whitespace.

CMake versions 2.4 and below silently removed leading and trailing whitespace from libraries linked with code like

```
target_link_libraries(myexe " A ")
```

This could lead to subtle errors in user projects.

The **OLD** behavior for this policy is to silently remove leading and trailing whitespace. The **NEW** behavior for this policy is to diagnose the existence of such whitespace as an error. The setting for this policy used when checking the library names is that in effect when the target is created by an `add_executable()` or `add_library()` command.

This policy was introduced in CMake version 2.6.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0003

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Libraries linked via full path no longer produce linker search paths.

This policy affects how libraries whose full paths are NOT known are found at link time, but was created due to a change in how CMake deals with libraries whose full paths are known. Consider the code

```
target_link_libraries(myexe /path/to/libA.so)
```

CMake 2.4 and below implemented linking to libraries whose full paths are known by splitting them on the link line into separate components consisting of the linker search path and the library name. The example code might have produced something like

```
... -L/path/to -lA ...
```

in order to link to library A. An analysis was performed to order multiple link directories such that the linker would find library A in the desired location, but there are cases in which this does not work. CMake versions 2.6 and above use the more reliable approach of passing the full path to libraries directly to the linker in most cases. The example code now produces something like

```
... /path/to/libA.so ....
```

Unfortunately this change can break code like

```
target_link_libraries(myexe /path/to/libA.so B)
```

where **B** is meant to find `/path/to/libB.so`. This code is wrong because the user is asking the linker to find library B but has not provided a linker search path (which may be added with the `link_directories` command). However, with the old linking implementation the code would work accidentally because the linker search path added for library A allowed library B to be found.

In order to support projects depending on linker search paths added by linking to libraries with known full paths, the **OLD** behavior for this policy will add the linker search paths even though they are not needed for their own libraries. When this policy is set to **OLD**, CMake will produce a link line such as

```
... -L/path/to /path/to/libA.so -lB ...
```

which will allow library B to be found as it was previously. When this policy is set to **NEW**, CMake will produce a link line such as

```
... /path/to/libA.so -lB ...
```

which more accurately matches what the project specified.

The setting for this policy used when generating the link line is that in effect when the target is created by an `add_executable` or `add_library` command. For the example described above, the code

```
cmake_policy(SET CMP0003 OLD) # or cmake_policy(VERSION 2.4)
add_executable(myexe myexe.c)
target_link_libraries(myexe /path/to/libA.so B)
```

will work and suppress the warning for this policy. It may also be updated to work with the corrected linking approach:

```
cmake_policy(SET CMP0003 NEW) # or cmake_policy(VERSION 2.6)
link_directories(/path/to) # needed to find library B
add_executable(myexe myexe.c)
target_link_libraries(myexe /path/to/libA.so B)
```

Even better, library B may be specified with a full path:

```
add_executable(myexe myexe.c)
target_link_libraries(myexe /path/to/libA.so /path/to/libB.so)
```

When all items on the link line have known paths CMake does not check this policy so it has no effect.

Note that the warning for this policy will be issued for at most one target. This avoids flooding users with messages for every target when setting the policy once will probably fix all targets.

This policy was introduced in CMake version 2.6.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0002

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

Logical target names must be globally unique.

Targets names created with `add_executable()`, `add_library()`, or `add_custom_target()` are logical build target names. Logical target names must be globally unique because:

- Unique names may be referenced unambiguously both in CMake code and on make tool command lines.
- Logical names are used by Xcode and VS IDE generators to produce meaningful project names for the targets.

The logical name of executable and library targets does not have to correspond to the physical file names built. Consider using the `OUTPUT_NAME` target property to create two targets with the same physical name while keeping logical names distinct. Custom targets must simply have globally unique names (unless one uses the global property `ALLOW_DUPLICATE_CUSTOM_TARGETS` with a Makefiles generator).

This policy was introduced in CMake version 2.6.0. Prior to removal in CMake version 4.0, it could be set by `cmake_policy()` or `cmake_minimum_required()`. If it was not set, CMake warned, and used **OLD** behavior.

CMP0001

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to `cmake_minimum_required()` or `cmake_policy()`.

CMAKE_BACKWARDS_COMPATIBILITY should no longer be used.

The behavior is to check **CMAKE_BACKWARDS_COMPATIBILITY** and present it to the user. The **NEW** behavior is to ignore **CMAKE_BACKWARDS_COMPATIBILITY** completely.

In CMake 2.4 and below the variable **CMAKE_BACKWARDS_COMPATIBILITY** was used to request compatibility with earlier versions of CMake. In CMake 2.6 and above all compatibility issues are handled by policies and the *cmake_policy()* command. However, CMake must still check **CMAKE_BACKWARDS_COMPATIBILITY** for projects written for CMake 2.4 and below.

This policy was introduced in CMake version 2.6.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

CMP0000

WARNING:

The **OLD** behavior of this policy was removed in CMake version 4.0. This policy must be set to **NEW** by a call to *cmake_minimum_required()* or *cmake_policy()*.

A minimum required CMake version must be specified.

CMake requires that projects specify the version of CMake to which they have been written. This policy has been put in place so users trying to build the project may be told when they need to update their CMake. Specifying a version also helps the project build with CMake versions newer than that specified. Use the *cmake_minimum_required()* command at the top of your main **CMakeLists.txt** file:

```
cmake_minimum_required(VERSION <major>.<minor>)
```

where **<major>.<minor>** is the version of CMake you want to support (such as **3.14**). The command will ensure that at least the given version of CMake is running and help newer versions be compatible with the project. See documentation of *cmake_minimum_required()* for details.

Note that the command invocation must appear in the **CMakeLists.txt** file itself; a call in an included file is not sufficient. The **OLD** behavior was to silently ignore the missing invocation. The **NEW** behavior is to issue an error instead of a warning.

This policy was introduced in CMake version 2.6.0. Prior to removal in CMake version 4.0, it could be set by *cmake_policy()* or *cmake_minimum_required()*. If it was not set, CMake warned, and used **OLD** behavior.

COPYRIGHT

2000-2025 Kitware, Inc. and Contributors