NAME

cmake-generator-expressions − CMake Generator Expressions

INTRODUCTION

Generator expressions are evaluated during build system generation to produce information specific to each build configuration. They have the form **$<...>**. For example:

```
target_include_directories(tgt PRIVATE /opt/include/$<CXX_COMPILER_ID>)
```

This would expand to **/opt/include/GNU**, **/opt/include/Clang**, etc. depending on the C++ compiler used.

Generator expressions are allowed in the context of many target properties, such as *LINK_LIBRARIES*, *INCLUDE_DIRECTORIES*, *COMPILE_DEFINITIONS* and others. They may also be used when using commands to populate those properties, such as *target_link_libraries()*, *target_include_directories()*, *target_compile_definitions()* and others. They enable conditional linking, conditional definitions used when compiling, conditional include directories, and more. The conditions may be based on the build configuration, target properties, platform information, or any other queryable information.

Generator expressions can be nested:

```
target_compile_definitions(tgt PRIVATE
  $<$<VERSION_LESS:$<CXX_COMPILER_VERSION>,4.2.0>:OLD_COMPILER>
)
```

The above would expand to **OLD_COMPILER** if the *CMAKE_CXX_COMPILER_VERSION* is less than 4.2.0.

WHITESPACE AND QUOTING

Generator expressions are typically parsed after command arguments. If a generator expression contains spaces, new lines, semicolons or other characters that may be interpreted as command argument separators, the whole expression should be surrounded by quotes when passed to a command. Failure to do so may result in the expression being split and it may no longer be recognized as a generator expression.

When using *add_custom_command()* or *add_custom_target()*, use the **VERBATIM** and **COMMAND_EXPAND_LISTS** options to obtain robust argument splitting and quoting.

```
# WRONG: Embedded space will be treated as an argument separator.
# This ends up not being seen as a generator expression at all.
add_custom_target(run_some_tool
  COMMAND some_tool -I$<JOIN:$<TARGET_PROPERTY:tgt,INCLUDE_DIRECTORIES>, -I>
  VERBATIM
)

# Better, but still not robust. Quotes prevent the space from splitting the
# expression. However, the tool will receive the expanded value as a single
# argument.
add_custom_target(run_some_tool
  COMMAND some_tool "-I$<JOIN:$<TARGET_PROPERTY:tgt,INCLUDE_DIRECTORIES>, -I>"
  VERBATIM
)

# Nearly correct. Using a semicolon to separate arguments and adding the
# COMMAND_EXPAND_LISTS option means that paths with spaces will be handled
# correctly. Quoting the whole expression ensures it is seen as a generator
# expression. But if the target property is empty, we will get a bare -I
# with nothing after it.
```

```
add_custom_target(run_some_tool
  COMMAND some_tool "-I$<JOIN:$<TARGET_PROPERTY:tgt,INCLUDE_DIRECTORIES>,;-I>"
  COMMAND_EXPAND_LISTS
  VERBATIM
)
```

Using variables to build up a more complex generator expression is also a good way to reduce errors and improve readability. The above example can be improved further like so:

```
# The $<BOOL:...> check prevents adding anything if the property is empty,
# assuming the property value cannot be one of CMake's false constants.
set(prop "$<TARGET_PROPERTY:tgt,INCLUDE_DIRECTORIES>")
add_custom_target(run_some_tool
  COMMAND some_tool "$<$<BOOL:${prop}>:-I$<JOIN:${prop},;-I>>"
  COMMAND_EXPAND_LISTS
  VERBATIM
)
```

Finally, the above example can be expressed in a more simple and robust way using an alternate generator expression:

```
add_custom_target(run_some_tool
  COMMAND some_tool "$<LIST:TRANSFORM,$<TARGET_PROPERTY:tgt,INCLUDE_DIRECTORIES>,
  COMMAND_EXPAND_LISTS
  VERBATIM
)
```

A common mistake is to try to split a generator expression across multiple lines with indenting:

```
# WRONG: New lines and spaces all treated as argument separators, so the
# generator expression is split and not recognized correctly.
target_compile_definitions(tgt PRIVATE
  $<$<AND:
      $<CXX_COMPILER_ID:GNU>,
      $<VERSION_GREATER_EQUAL:$<CXX_COMPILER_VERSION>,5>
    >:HAVE_5_OR_LATER>
)
```

Again, use helper variables with well-chosen names to build up a readable expression instead:

```
set(is_gnu "$<CXX_COMPILER_ID:GNU>")
set(v5_or_later "$<VERSION_GREATER_EQUAL:$<CXX_COMPILER_VERSION>,5>")
set(meet_requirements "$<AND:${is_gnu},${v5_or_later}>")
target_compile_definitions(tgt PRIVATE
  "$<${meet_requirements}:HAVE_5_OR_LATER>"
)
```

DEBUGGING

Since generator expressions are evaluated during generation of the buildsystem, and not during processing of **CMakeLists.txt** files, it is not possible to inspect their result with the *message()* command. One possible way to generate debug messages is to add a custom target:

```
add_custom_target(genexdebug COMMAND ${CMAKE_COMMAND} -E echo "$<...>")
```

After running **cmake**, you can then build the **genexdebug** target to print the result of the **$<...>** expression

(i.e. run the command *cmake −−build ... −−target genexdebug*).

Another way is to write debug messages to a file with *file(GENERATE)*:

```
file(GENERATE OUTPUT filename CONTENT "$<...>")
```

GENERATOR EXPRESSION REFERENCE

NOTE:

This reference deviates from most of the CMake documentation in that it omits angular brackets **<...>** around placeholders like **condition**, **string**, **target**, etc. This is to prevent an opportunity for those placeholders to be misinterpreted as generator expressions.

Conditional Expressions

A fundamental category of generator expressions relates to conditional logic. Two forms of conditional generator expressions are supported:

$<condition:true_string>

Evaluates to **true_string** if **condition** is **1**, or an empty string if **condition** evaluates to **0**. Any other value for **condition** results in an error.

$<IF:condition,true_string,false_string>

Added in version 3.8.

Evaluates to **true_string** if **condition** is **1**, or **false_string** if **condition** is **0**. Any other value for **condition** results in an error.

Added in version 3.28: This generator expression short−circuits such that generator expressions in **false_string** will not evaluate when **condition** is **1**, and generator expressions in **true_string** will not evaluate when condition is **0**.

Typically, the **condition** is itself a generator expression. For instance, the following expression expands to **DEBUG_MODE** when the **Debug** configuration is used, and the empty string for all other configurations:

```
$<$<CONFIG:Debug>:DEBUG_MODE>
```

Boolean−like **condition** values other than **1** or **0** can be handled by wrapping them with the **$<BOOL:...>** generator expression:

$<BOOL:string>

Converts **string** to **0** or **1**. Evaluates to **0** if any of the following is true:

- **string** is empty,
- **string** is a case−insensitive equal of **0**, **FALSE**, **OFF**, **N**, **NO**, **IGNORE**, or **NOTFOUND**, or
- **string** ends in the suffix **−NOTFOUND** (case−sensitive).

Otherwise evaluates to **1**.

The **$<BOOL:...>** generator expression is often used when a **condition** is provided by a CMake variable:

```
$<$<BOOL:${HAVE_SOME_FEATURE}>:-DENABLE_SOME_FEATURE>
```

Logical Operators

The common boolean logic operators are supported:

$<AND:conditions>
where **conditions** is a comma−separated list of boolean expressions, all of which must evaluate to either **1** or **0**. The whole expression evaluates to **1** if all conditions are **1**. If any condition is **0**, the whole expression evaluates to **0**.

$<OR:conditions>
where **conditions** is a comma−separated list of boolean expressions. all of which must evaluate to either **1** or **0**. The whole expression evaluates to **1** if at least one of the **conditions** is **1**. If all**conditions** evaluate to **0**, the whole expression evaluates to **0**.

$<NOT:condition>
condition must be **0** or **1**. The result of the expression is **0** if **condition** is **1**, else **1**.

Added in version 3.28: Logical operators short−circuit such that generator expressions in the arguments list will not be evaluated once a return value can be determined.

Primary Comparison Expressions
CMake supports a variety of generator expressions that compare things. This section covers the primary and most widely used comparison types. Other more specific comparison types are documented in their own separate sections further below.

String Comparisons
$<STREQUAL:string1,string2>
1 if **string1** and **string2** are equal, else **0**. The comparison is case−sensitive. For a case−insensitive comparison, combine with a *string transforming generator expression*. For example, the following evaluates to **1** if **${foo}** is any of **BAR**, **Bar**, **bar**, etc.

```
$<STREQUAL:$<UPPER_CASE:${foo}>,BAR>
```

$<EQUAL:value1,value2>
1 if **value1** and **value2** are numerically equal, else **0**.

Version Comparisons
$<VERSION_LESS:v1,v2>
1 if **v1** is a version less than **v2**, else **0**.

$<VERSION_GREATER:v1,v2>
1 if **v1** is a version greater than **v2**, else **0**.

$<VERSION_EQUAL:v1,v2>
1 if **v1** is the same version as **v2**, else **0**.

$<VERSION_LESS_EQUAL:v1,v2>
Added in version 3.7.

1 if **v1** is a version less than or equal to **v2**, else **0**.

$<VERSION_GREATER_EQUAL:v1,v2>
Added in version 3.7.

1 if **v1** is a version greater than or equal to **v2**, else **0**.

String Transformations
$<LOWER_CASE:string>
Content of **string** converted to lower case.

$<UPPER_CASE:string>
Content of **string** converted to upper case.

$<MAKE_C_IDENTIFIER:...>
Content of **...** converted to a C identifier. The conversion follows the same behavior as *string(MAKE_C_IDENTIFIER)*.

List Expressions
Most of the expressions in this section are closely associated with the *list()* command, providing the same capabilities, but in the form of a generator expression.

In each of the following list−related generator expressions, the **list** must not contain any commas if that generator expression expects something to be provided after the **list**. For example, the expression **$<LIST:FIND,list,value>** requires a **value** after the **list**. Since a comma is used to separate the**list** and the **value**, the **list** cannot itself contain a comma. This restriction does not apply to the *list()* command, it is specific to the list−handling generator expressions only.

List Comparisons
$<IN_LIST:string,list>
Added in version 3.12.

1 if **string** is an item in the semicolon−separated **list**, else **0**. It uses case−sensitive comparisons.

List Queries
$<LIST:LENGTH,list>
Added in version 3.27.

The number of items in the **list**.

$<LIST:GET,list,index,...>
Added in version 3.27.

Expands to the list of items specified by indices from the **list**.

$<LIST:SUBLIST,list,begin,length>
Added in version 3.27.

A sublist of the given **list**. If**length** is 0, an empty list will be returned. If **length** is −1 or the list is smaller than **begin + length**, the remaining items of the list starting at **begin** will be returned.

$<LIST:FIND,list,value>
Added in version 3.27.

The index of the first item in **list** with the specified **value**, or −1 if **value** is not in the **list**.

List Transformations
$<LIST:JOIN,list,glue>
Added in version 3.27.

Converts **list** to a single string with the content of the **glue** string inserted between each item. This is conceptually the same operation as *$<JOIN:list,glue>*, but the two have different behavior with regard to empty items. **$<LIST :JOIN,list,glue>** preserves all empty items, whereas *$<JOIN:list,glue>* drops all empty items from the list.

$<LIST:APPEND,list,item,...>
Added in version 3.27.

The **list** with each **item** appended. Multiple items should be separated by commas.

$<LIST:PREPEND,list,item,...>
Added in version 3.27.

The **list** with each **item** inserted at the beginning. If there are multiple items, they should be separated by commas, and the order of the prepended items will be preserved.

$<LIST:INSERT,list,index,item,...>
Added in version 3.27.

The **list** with the **item** (or multiple items) inserted at the specified **index**. Multiple items should be separated by commas.

It is an error to specify an out−of−range **index**. Valid indexes are 0 to N, where N is the length of the list, inclusive. An empty list has length 0.

$<LIST:POP_BACK,list>
Added in version 3.27.

The **list** with the last item removed.

$<LIST:POP_FRONT,list>
Added in version 3.27.

The **list** with the first item removed.

$<LIST:REMOVE_ITEM,list,value,...>
Added in version 3.27.

The **list** with all instances of the given **value** (or values) removed. If multiple values are given, they should be separated by commas.

$<LIST:REMOVE_AT,list,index,...>
Added in version 3.27.

The **list** with the item at each given **index** removed.

$<LIST:REMOVE_DUPLICATES,list>
Added in version 3.27.

The **list** with all duplicated items removed. The relative order of items is preserved, but if duplicates are encountered, only the first instance is preserved. The result is the same as *$<REMOVE_DUPLICATES:list>*.

$<LIST:FILTER,list,INCLUDE|EXCLUDE,regex>
Added in version 3.27.

A list of items from the **list** which match (**INCLUDE**) or do not match (**EXCLUDE**) the regular expression **regex**. The result is the same as*$<FIL TER:list,INCLUDE|EXCLUDE,regex>*.

$<LIST:TRANSFORM,list,ACTION[,SELECTOR]>
Added in version 3.27.

The **list** transformed by applying an **ACTION** to all or, by specifying a **SELECTOR**, to the selected list items.

NOTE:
The **TRANSFORM** sub−command does not change the number of items in the list. If a **SELECTOR** is specified, only some items will be changed, the other ones will remain the same as before the transformation.

ACTION specifies the action to apply to the items of the list. The actions have exactly the same semantics as for the *list(TRANSFORM)* command. **ACTION** must be one of the following:

*APPEND***,** *PREPEND*
Append, prepend specified value to each item of the list.

```
$<LIST:TRANSFORM,list,(APPEND|PREPEND),value[,SELECTOR]>
```

*TOLOWER***,** *TOUPPER*
Convert each item of the list to lower, upper characters.

```
$<LIST:TRANSFORM,list,(TOLOWER|TOUPPER)[,SELECTOR]>
```

STRIP Remove leading and trailing spaces from each item of the list.

```
$<LIST:TRANSFORM,list,STRIP[,SELECTOR]>
```

*REPLACE***:**
Match the regular expression as many times as possible and substitute the replacement expression for the match for each item of the list.

```
$<LIST:TRANSFORM,list,REPLACE,regular_expression,replace_expressior
```

SELECTOR determines which items of the list will be transformed. Only one type of selector can be specified at a time. When given, **SELECTOR** must be one of the following:

AT Specify a list of indexes.

```
$<LIST:TRANSFORM,list,ACTION,AT,index[,index...]>
```

FOR Specify a range with, optionally, an increment used to iterate over the range.

```
$<LIST:TRANSFORM,list,ACTION,FOR,start,stop[,step]>
```

REGEX
Specify a regular expression. Only items matching the regular expression will be transformed.

```
$<LIST:TRANSFORM,list,ACTION,REGEX,regular_expression>
```

$<JOIN:list,glue>
Joins the **list** with the content of the **glue** string inserted between each item. This is conceptually the same operation as *$<LIST:JOIN,list,glue>*, but the two have different behavior with regard to empty items. *$<LIST :JOIN,list,glue>* preserves all empty items, whereas **$<JOIN,list,glue>** drops all empty items from the list.

$<REMOVE_DUPLICATES:list>
Added in version 3.15.

Removes duplicated items in the given **list**. The relative order of items is preserved, and if duplicates are encountered, only the first instance is retained. The result is the same as *$<LIST:REMOVE_DUPLICATES,list>*.

$<FILTER:list,INCLUDE|EXCLUDE,regex>
Added in version 3.15.

Includes or removes items from **list** that match the regular expression **regex**. The result is the same as *$<LIST:FILTER,list,INCLUDE|EXCLUDE,regex>*.

List Ordering

$<LIST:REVERSE,list>
Added in version 3.27.

The **list** with the items in reverse order.

$<LIST:SORT,list[,(COMPARE:option|CASE:option|ORDER:option)]...>
Added in version 3.27.

The **list** sorted according to the specified options.

Use one of the **COMPARE** options to select the comparison method for sorting:

STRING
Sorts a list of strings alphabetically. This is the default behavior if the **COMPARE** option is not given.

FILE_BASENAME
Sorts a list of file paths by their basenames.

NATURAL
Sorts a list of strings using natural order (see the man page for **strverscmp(3)**), such that contiguous digits are compared as whole numbers. For example, the following list **10.0 1.1 2.1 8.0 2.0 3.1** will be sorted as **1.1 2.0 2.1 3.1 8.0 10.0** if the **NATURAL** comparison is selected, whereas it will be sorted as **1.1 10.0 2.0 2.1 3.1 8.0** with the **STRING** comparison.

Use one of the **CASE** options to select a case−sensitive or case−insensitive sort mode:

SENSITIVE
List items are sorted in a case−sensitive manner. This is the default behavior if the **CASE** option is not given.

INSENSITIVE
List items are sorted in a case−insensitive manner. The order of items which differ only by upper/lowercase is not specified.

To control the sort order, one of the **ORDER** options can be given:

ASCENDING
Sorts the list in ascending order. This is the default behavior when the **ORDER** option is not given.

DESCENDING
Sorts the list in descending order.

Options can be specified in any order, but it is an error to specify the same option multiple times.

```
$<LIST:SORT,list,CASE:SENSITIVE,COMPARE:STRING,ORDER:DESCENDING>
```

Path Expressions

Most of the expressions in this section are closely associated with the *cmake_path()* command, providing the same capabilities, but in the form of a generator expression.

For all generator expressions in this section, paths are expected to be in cmake−style format. The *$<PATH:CMAKE_PATH>* generator expression can be used to convert a native path to a cmake−style one.

Path Comparisons

$<PATH_EQUAL:path1,path2>
Added in version 3.24.

Compares the lexical representations of two paths. No normalization is performed on either path. Returns **1** if the paths are equal, **0** otherwise.

See *cmake_path(COMPARE)* for more details.

Path Queries

These expressions provide the generation−time capabilities equivalent to the *Query* options of the *cmake_path()* command. All paths are expected to be in cmake−style format.

$<PATH:HAS_*,path>
Added in version 3.24.

The following operations return **1** if the particular path component is present, **0** otherwise. See *Path Structure And Terminology* for the meaning of each path component.

```
$<PATH:HAS_ROOT_NAME,path>
$<PATH:HAS_ROOT_DIRECTORY,path>
$<PATH:HAS_ROOT_PATH,path>
$<PATH:HAS_FILENAME,path>
$<PATH:HAS_EXTENSION,path>
$<PATH:HAS_STEM,path>
$<PATH:HAS_RELATIVE_PART,path>
$<PATH:HAS_PARENT_PATH,path>
```

Note the following special cases:

- For **HAS_ROOT_PATH**, a true result will only be returned if at least one of **root−name** or **root−directory** is non−empty.
- For **HAS_PARENT_PATH**, the root directory is also considered to have a parent, which will be itself. The result is true except if the path consists of just a *filename*.

$<PATH:IS_ABSOLUTE,path>
Added in version 3.24.

Returns **1** if the path is *absolute*, **0** otherwise.

$<PATH:IS_RELATIVE,path>
Added in version 3.24.

This will return the opposite of **IS_ABSOLUTE**.

$<PATH:IS_PREFIX[,NORMALIZE],path,input>
Added in version 3.24.

Returns **1** if **path** is the prefix of **input**, **0** otherwise.

When the **NORMALIZE** option is specified, **path** and **input** are *normalized* before the check.

Path Decomposition
These expressions provide the generation−time capabilities equivalent to the *Decomposition* options of the *cmake_path()* command. All paths are expected to be in cmake−style format.

$<PATH:GET_*,...>
Added in version 3.24.

The following operations retrieve a different component or group of components from a path. See *Path Structure And Terminology* for the meaning of each path component.

Changed in version 3.27: All operations now accept a list of paths as argument. When a list of paths is specified, the operation will be applied to each path.

```
$<PATH:GET_ROOT_NAME,path...>
$<PATH:GET_ROOT_DIRECTORY,path...>
$<PATH:GET_ROOT_PATH,path...>
$<PATH:GET_FILENAME,path...>
$<PATH:GET_EXTENSION[,LAST_ONLY],path...>
$<PATH:GET_STEM[,LAST_ONLY],path...>
$<PATH:GET_RELATIVE_PART,path...>
$<PATH:GET_PARENT_PATH,path...>
```

If a requested component is not present in the path, an empty string is returned.

Path Transformations
These expressions provide the generation−time capabilities equivalent to the *Modification* and *Generation* options of the *cmake_path()* command. All paths are expected to be in cmake−style format.

Changed in version 3.27: All operations now accept a list of paths as argument. When a list of paths is specified, the operation will be applied to each path.

$<PATH:CMAKE_PATH[,NORMALIZE],path...>
Added in version 3.24.

Returns **path**. If **path** is a native path, it is converted into a cmake−style path with forward−slashes (/). On Windows, the long filename marker is taken into account.

When the **NORMALIZE** option is specified, the path is *normalized* after the conversion.

$<PATH:NATIVE_PATH[,NORMALIZE],path...>
Added in version 4.0.

Returns **path** converted into a native format with platform−specific slashes (\ on Windows hosts and / elsewhere).

When the **NORMALIZE** option is specified, the path is *normalized* before the conversion.

$<PATH:APPEND,path...,input,...>
Added in version 3.24.

Returns all the **input** arguments appended to **path** using / as the **directory−separator**. Depending on the **input**, the value of **path** may be discarded.

See *cmake_path(APPEND)* for more details.

$<PATH:REMOVE_FILENAME,path...>
Added in version 3.24.

Returns **path** with filename component (as returned by **$<PATH:GET_FILENAME>**) removed. After removal, any trailing **directory−separator** is left alone, if present.

See *cmake_path(REMOVE_FILENAME)* for more details.

$<PATH:REPLACE_FILENAME,path...,input>
Added in version 3.24.

Returns **path** with the filename component replaced by **input**. If **path** has no filename component (i.e. **$<PATH:HAS_FILENAME>** returns **0**), **path** is unchanged.

See *cmake_path(REPLACE_FILENAME)* for more details.

$<PATH:REMOVE_EXTENSION[,LAST_ONLY],path...>
Added in version 3.24.

Returns **path** with the *extension* removed, if any.

See *cmake_path(REMOVE_EXTENSION)* for more details.

$<PATH:REPLACE_EXTENSION[,LAST_ONLY],path...,input>
Added in version 3.24.

Returns **path** with the *extension* replaced by **input**, if any.

See *cmake_path(REPLACE_EXTENSION)* for more details.

$<PATH:NORMAL_PATH,path...>
Added in version 3.24.

Returns **path** normalized according to the steps described in *Normalization*.

$<PATH:RELATIVE_PATH,path...,base_directory>
Added in version 3.24.

Returns **path**, modified to make it relative to the **base_directory** argument.

See *cmake_path(RELATIVE_PATH)* for more details.

$<PATH:ABSOLUTE_PATH[,NORMALIZE],path...,base_directory>
Added in version 3.24.

Returns **path** as absolute. If **path** is a relative path (**$<PATH:IS_RELATIVE>** returns **1**), it is evaluated relative to the given base directory specified by **base_directory** argument.

When the **NORMALIZE** option is specified, the path is *normalized* after the path computation.

See *cmake_path(ABSOLUTE_PATH)* for more details.

Shell Paths

$<SHELL_PATH:...>
Added in version 3.4.

Content of **...** converted to shell path style. For example, slashes are converted to backslashes in Windows shells and drive letters are converted to posix paths in MSYS shells. The **...** must be an absolute path.

Added in version 3.14: The **...** may be a *semicolon−separated list* of paths, in which case each path is converted individually and a result list is generated using the shell path separator (**:** on POSIX and **;** on Windows). Be sure to enclose the argument containing this genex in double quotes in CMake source code so that **;** does not split arguments.

Configuration Expressions

$<CONFIG>
Configuration name. Use this instead of the deprecated *CONFIGURATION* generator expression.

$<CONFIG:cfgs>
1 if config is any one of the entries in comma−separated list **cfgs**, else **0**. This is a case−insensitive comparison. The mapping in *MAP_IMPORTED_CONFIG_<CONFIG>* is also considered by this expression when it is evaluated on a property of an *IMPORTED* target.

Changed in version 3.19: Multiple configurations can be specified for **cfgs**. CMake 3.18 and earlier only accepted a single configuration.

$<OUTPUT_CONFIG:...>
Added in version 3.20.

Only valid in *add_custom_command()* and *add_custom_target()* as the outer−most generator expression in an argument. With the *Ninja Multi−Config* generator, generator expressions in **...** are evaluated using the custom command's "output config". With other generators, the content of **...** is evaluated normally.

$<COMMAND_CONFIG:...>
Added in version 3.20.

Only valid in *add_custom_command()* and *add_custom_target()* as the outer−most generator expression in an argument. With the *Ninja Multi−Config* generator, generator expressions in **...** are evaluated using the custom command's "command config". With other generators, the content of **...** is evaluated normally.

Toolchain And Language Expressions

Platform

$<PLATFORM_ID>
The current system's CMake platform id. See also the *CMAKE_SYSTEM_NAME* variable.

$<PLATFORM_ID:platform_ids>
1 if CMake's platform id matches any one of the entries in comma−separated list **platform_ids**, otherwise **0**. See also the*CMAKE_SYSTEM_N AME* variable.

Compiler Version

See also the *CMAKE_<LANG>_COMPILER_VERSION* variable, which is closely related to the expressions in this sub−section.

$<C_COMPILER_VERSION>
The version of the C compiler used.

$<C_COMPILER_VERSION:version>
1 if the version of the C compiler matches **version**, otherwise **0**.

$<CXX_COMPILER_VERSION>
The version of the CXX compiler used.

$<CXX_COMPILER_VERSION:version>
1 if the version of the C++ compiler matches **version**, otherwise **0**.

$<CUDA_COMPILER_VERSION>
Added in version 3.15.

The version of the CUDA compiler used.

$<CUDA_COMPILER_VERSION:version>
Added in version 3.15.

1 if the version of the C++ compiler matches **version**, otherwise **0**.

$<OBJC_COMPILER_VERSION>
Added in version 3.16.

The version of the Objective−C compiler used.

$<OBJC_COMPILER_VERSION:version>
Added in version 3.16.

1 if the version of the Objective−C compiler matches **version**, otherwise **0**.

$<OBJCXX_COMPILER_VERSION>
Added in version 3.16.

The version of the Objective−C++ compiler used.

$<OBJCXX_COMPILER_VERSION:version>
Added in version 3.16.

1 if the version of the Objective−C++ compiler matches **version**, otherwise **0**.

$<Fortran_COMPILER_VERSION>
The version of the Fortran compiler used.

$<Fortran_COMPILER_VERSION:version>
1 if the version of the Fortran compiler matches **version**, otherwise **0**.

$<HIP_COMPILER_VERSION>
Added in version 3.21.

The version of the HIP compiler used.

$<HIP_COMPILER_VERSION:version>
Added in version 3.21.

1 if the version of the HIP compiler matches **version**, otherwise **0**.

$<ISPC_COMPILER_VERSION>
Added in version 3.19.

The version of the ISPC compiler used.

$<ISPC_COMPILER_VERSION:version>
Added in version 3.19.

1 if the version of the ISPC compiler matches **version**, otherwise **0**.

Compiler Language, ID, and Frontend−Variant
See also the *CMAKE_<LANG>_COMPILER_ID* and *CMAKE_<LANG>_COMPILER_FRONTEND_VARIANT* variables, which are closely related to most of the expressions in this sub−section.

$<C_COMPILER_ID>
CMake's compiler id of the C compiler used.

$<C_COMPILER_ID:compiler_ids>
where **compiler_ids** is a comma−separated list. **1** if CMake's compiler id of the C compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

Changed in version 3.15: Multiple **compiler_ids** can be specified. CMake 3.14 and earlier only accepted a single compiler ID.

$<CXX_COMPILER_ID>
CMake's compiler id of the C++ compiler used.

$<CXX_COMPILER_ID:compiler_ids>
where **compiler_ids** is a comma−separated list. **1** if CMake's compiler id of the C++ compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

Changed in version 3.15: Multiple **compiler_ids** can be specified. CMake 3.14 and earlier only

accepted a single compiler ID.

$<CUDA_COMPILER_ID>
Added in version 3.15.

CMake's compiler id of the CUDA compiler used.

$<CUDA_COMPILER_ID:compiler_ids>
Added in version 3.15.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler id of the CUDA compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<OBJC_COMPILER_ID>
Added in version 3.16.

CMake's compiler id of the Objective−C compiler used.

$<OBJC_COMPILER_ID:compiler_ids>
Added in version 3.16.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler id of the Objective−C compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<OBJCXX_COMPILER_ID>
Added in version 3.16.

CMake's compiler id of the Objective−C++ compiler used.

$<OBJCXX_COMPILER_ID:compiler_ids>
Added in version 3.16.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler id of the Objective−C++ compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<Fortran_COMPILER_ID>
CMake's compiler id of the Fortran compiler used.

$<Fortran_COMPILER_ID:compiler_ids>
where **compiler_ids** is a comma−separated list. **1** if CMake's compiler id of the Fortran compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

Changed in version 3.15: Multiple **compiler_ids** can be specified. CMake 3.14 and earlier only accepted a single compiler ID.

$<HIP_COMPILER_ID>
Added in version 3.21.

CMake's compiler id of the HIP compiler used.

$<HIP_COMPILER_ID:compiler_ids>
Added in version 3.21.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler id of the HIP compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<ISPC_COMPILER_ID>
Added in version 3.19.

CMake's compiler id of the ISPC compiler used.

$<ISPC_COMPILER_ID:compiler_ids>
Added in version 3.19.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler id of the ISPC compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<C_COMPILER_FRONTEND_VARIANT>
Added in version 3.30.

CMake's compiler frontend variant of the C compiler used.

$<C_COMPILER_FRONTEND_VARIANT:compiler_ids>
Added in version 3.30.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler frontend variant of the C compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<CXX_COMPILER_FRONTEND_VARIANT>
Added in version 3.30.

CMake's compiler frontend variant of the C++ compiler used.

$<CXX_COMPILER_FRONTEND_VARIANT:compiler_ids>
Added in version 3.30.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler frontend variant of the C++ compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<CUDA_COMPILER_FRONTEND_VARIANT>
Added in version 3.30.

CMake's compiler id of the CUDA compiler used.

$<CUDA_COMPILER_FRONTEND_VARIANT:compiler_ids>
Added in version 3.30.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler frontend variant of the CUDA compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<OBJC_COMPILER_FRONTEND_VARIANT>
Added in version 3.30.

CMake's compiler frontend variant of the Objective−C compiler used.

$<OBJC_COMPILER_FRONTEND_VARIANT:compiler_ids>
Added in version 3.30.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler frontend variant of the Objective−C compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<OBJCXX_COMPILER_FRONTEND_VARIANT>
Added in version 3.30.

CMake's compiler frontend variant of the Objective−C++ compiler used.

$<OBJCXX_COMPILER_FRONTEND_VARIANT:compiler_ids>
Added in version 3.30.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler frontend variant of the Objective−C++ compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<Fortran_COMPILER_FRONTEND_VARIANT>
Added in version 3.30.

CMake's compiler id of the Fortran compiler used.

$<Fortran_COMPILER_FRONTEND_VARIANT:compiler_ids>
Added in version 3.30.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler frontend variant of the Fortran compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<HIP_COMPILER_FRONTEND_VARIANT>
Added in version 3.30.

CMake's compiler id of the HIP compiler used.

$<HIP_COMPILER_FRONTEND_VARIANT:compiler_ids>
Added in version 3.30.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler frontend variant of the HIP compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<ISPC_COMPILER_FRONTEND_VARIANT>
Added in version 3.30.

CMake's compiler id of the ISPC compiler used.

$<ISPC_COMPILER_FRONTEND_VARIANT:compiler_ids>
Added in version 3.30.

where **compiler_ids** is a comma−separated list. **1** if CMake's compiler frontend variant of the ISPC compiler matches any one of the entries in **compiler_ids**, otherwise **0**.

$<COMPILE_LANGUAGE>
Added in version 3.3.

The compile language of source files when evaluating compile options. See *the related boolean expression* **$<COMPILE_LANGUAGE:language>** for notes about the portability of this generator expression.

$<COMPILE_LANGUAGE:languages>
Added in version 3.3.

Changed in version 3.15: Multiple languages can be specified for **languages**. CMake 3.14 and earlier only accepted a single language.

1 when the language used for compilation unit matches any of the comma−separated entries in **languages**, otherwise **0**. This expression may be used to specify compile options, compile definitions, and include directories for source files of a particular language in a target. For example:

```
add_executable(myapp main.cpp foo.c bar.cpp zot.cu)
target_compile_options(myapp
  PRIVATE $<$<COMPILE_LANGUAGE:CXX>:-fno-exceptions>
)
target_compile_definitions(myapp
  PRIVATE $<$<COMPILE_LANGUAGE:CXX>:COMPILING_CXX>
          $<$<COMPILE_LANGUAGE:CUDA>:COMPILING_CUDA>
)
target_include_directories(myapp
  PRIVATE $<$<COMPILE_LANGUAGE:CXX,CUDA>:/opt/foo/headers>
)
```

This specifies the use of the **−fno−exceptions** compile option, **COMPILING_CXX** compile definition, and **cxx_headers** include directory for C++ only (compiler id checks elided). It also specifies a **COMPILING_CUDA** compile definition for CUDA.

Note that with *Visual Studio Generators* and *Xcode* there is no way to represent target−wide compile definitions or include directories separately for **C** and **CXX** languages. Also, with *Visual Studio Generators* there is no way to represent target−wide flags separately for **C** and **CXX** languages. Under these generators, expressions for both C and C++ sources will be evaluated using **CXX** if there are any C++ sources and otherwise using **C**. A workaround is to create separate libraries for each source file language instead:

```
add_library(myapp_c foo.c)
add_library(myapp_cxx bar.cpp)
target_compile_options(myapp_cxx PUBLIC -fno-exceptions)
add_executable(myapp main.cpp)
target_link_libraries(myapp myapp_c myapp_cxx)
```

$<COMPILE_LANG_AND_ID:language,compiler_ids>
Added in version 3.15.

1 when the language used for compilation unit matches **language** and CMake's compiler id of the **language** compiler matches any one of the comma−separated entries in **compiler_ids**, otherwise **0**. This expression is a short form for the combination of **$<COMPILE_LANGUAGE:language>** and **$<LANG_COMPILER_ID:compiler_ids>**. This expression may be used to specify compile options, compile definitions, and include directories for source files of a particular language and compiler combination in a target. For example:

```
add_executable(myapp main.cpp foo.c bar.cpp zot.cu)
target_compile_definitions(myapp
  PRIVATE $<$<COMPILE_LANG_AND_ID:CXX,AppleClang,Clang>:COMPILING_CXX_WITH_C
          $<$<COMPILE_LANG_AND_ID:CXX,Intel>:COMPILING_CXX_WITH_INTEL>
          $<$<COMPILE_LANG_AND_ID:C,Clang>:COMPILING_C_WITH_CLANG>
)
```

This specifies the use of different compile definitions based on both the compiler id and compilation language. This example will have a **COMPILING_CXX_WITH_CLANG** compile definition when Clang is the CXX compiler, and **COMPILING_CXX_WITH_INTEL** when Intel is the CXX compiler. Likewise, when the C compiler is Clang, it will only see the **COMPILING_C_WITH_CLANG** definition.

Without the **COMPILE_LANG_AND_ID** generator expression, the same logic would be expressed as:

```
target_compile_definitions(myapp
  PRIVATE $<$<AND:$<COMPILE_LANGUAGE:CXX>,$<CXX_COMPILER_ID:AppleClang,Clang
          $<$<AND:$<COMPILE_LANGUAGE:CXX>,$<CXX_COMPILER_ID:Intel>>:COMPILI
          $<$<AND:$<COMPILE_LANGUAGE:C>,$<C_COMPILER_ID:Clang>>:COMPILING_C_
)
```

Compile Features

$<COMPILE_FEATURES:features>
Added in version 3.1.

where **features** is a comma−separated list. Evaluates to **1** if all of the **features** are available for the 'head' target, and **0** otherwise. If this expression is used while evaluating the link implementation of a target and if any dependency transitively increases the required *C_STANDARD* or *CXX_STANDARD* for the 'head' target, an error is reported. See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

Compile Context

$<COMPILE_ONLY:...>
Added in version 3.27.

Content of **...**, when collecting *transitive compile properties*, otherwise it is the empty string. This is intended for use in an *INTERFACE_LINK_LIBRARIES* and *LINK_LIBRARIES* target properties, typically populated via the *target_link_libraries()* command. Provides compilation usage requirements without any linking requirements.

Use cases include header−only usage where all usages are known to not have linking requirements (e.g., all−**inline** or C++ template libraries).

Note that for proper evaluation of this expression requires policy *CMP0099* to be set to **NEW**.

Linker Language And ID

$<LINK_LANGUAGE>

Added in version 3.18.

The link language of the target when evaluating link options. See *the related boolean expression* **$<LINK_LANGUAGE:languages>** for notes about the portability of this generator expression.

NOTE:

This generator expression is not supported by the link libraries properties to avoid side−effects due to the double evaluation of these properties.

$<LINK_LANGUAGE:languages>

Added in version 3.18.

1 when the language used for link step matches any of the comma−separated entries in **languages**, otherwise **0**. This expression may be used to specify link libraries, link options, link directories and link dependencies of a particular language in a target. For example:

```
add_library(api_C ...)
add_library(api_CXX ...)
add_library(api INTERFACE)
target_link_options(api   INTERFACE $<$<LINK_LANGUAGE:C>:-opt_c>
                                    $<$<LINK_LANGUAGE:CXX>:-opt_cxx>)
target_link_libraries(api INTERFACE $<$<LINK_LANGUAGE:C>:api_C>
                                    $<$<LINK_LANGUAGE:CXX>:api_CXX>)

add_executable(myapp1 main.c)
target_link_options(myapp1 PRIVATE api)

add_executable(myapp2 main.cpp)
target_link_options(myapp2 PRIVATE api)
```

This specifies to use the **api** target for linking targets **myapp1** and **myapp2**. In practice, **myapp1** will link with target **api_C** and option **−opt_c** because it will use **C** as link language. And **myapp2** will link with **api_CXX** and option **−opt_cxx** because **CXX** will be the link language.

NOTE:

To determine the link language of a target, it is required to collect, transitively, all the targets which will be linked to it. So, for link libraries properties, a double evaluation will be done. During the first evaluation, **$<LINK_LANGUAGE:..>** expressions will always return **0**. The link language computed after this first pass will be used to do the second pass. To avoid inconsistency, it is required that the second pass do not change the link language. Moreover, to avoid unexpected side−effects, it is required to specify complete entities as part of the **$<LINK_LANGUAGE:..>** expression. For example:

```
add_library(lib STATIC file.cxx)
add_library(libother STATIC file.c)

# bad usage
add_executable(myapp1 main.c)
target_link_libraries(myapp1 PRIVATE lib$<$<LINK_LANGUAGE:C>:other>)
```

```
# correct usage
add_executable(myapp2 main.c)
target_link_libraries(myapp2 PRIVATE $<$<LINK_LANGUAGE:C>:libother>)
```

In this example, for **myapp1**, the first pass will, unexpectedly, determine that the link language is **CXX** because the evaluation of the generator expression will be an empty string so **myapp1** will depends on target **lib** which is **C++**. On the contrary, for **myapp2**, the first evaluation will give **C** as link language, so the second pass will correctly add target **libother** as link dependency.

$<LINK_LANG_AND_ID:language,compiler_ids>

Added in version 3.18.

1 when the language used for link step matches **language** and the CMake's compiler id of the language linker matches any one of the comma−separated entries in **compiler_ids**, otherwise **0**. This expression is a short form for the combination of **$<LINK_LANGUAGE:language>** and **$<LANG_COMPILER_ID:compiler_ids>**. This expression may be used to specify link libraries, link options, link directories and link dependencies of a particular language and linker combination in a target. For example:

```
add_library(libC_Clang ...)
add_library(libCXX_Clang ...)
add_library(libC_Intel ...)
add_library(libCXX_Intel ...)

add_executable(myapp main.c)
if (CXX_CONFIG)
  target_sources(myapp PRIVATE file.cxx)
endif()
target_link_libraries(myapp
  PRIVATE $<$<LINK_LANG_AND_ID:CXX,Clang,AppleClang>:libCXX_Clang>
          $<$<LINK_LANG_AND_ID:C,Clang,AppleClang>:libC_Clang>
          $<$<LINK_LANG_AND_ID:CXX,Intel>:libCXX_Intel>
          $<$<LINK_LANG_AND_ID:C,Intel>:libC_Intel>)
```

This specifies the use of different link libraries based on both the compiler id and link language. This example will have target **libCXX_Clang** as link dependency when **Clang** or **AppleClang** is the **CXX** linker, and **libCXX_Intel** when **Intel** is the **CXX** linker. Likewise when the **C** linker is **Clang** or **AppleClang**, target **libC_Clang** will be added as link dependency and **libC_Intel** when **Intel** is the **C** linker.

See *the note related to* **$<LINK_LANGUAGE:language>** for constraints about the usage of this generator expression.

Link Features

$<LINK_LIBRARY:feature,library−list>

Added in version 3.24.

Specify a set of libraries to link to a target, along with a **feature** which provides details about *how* they should be linked. For example:

```
add_library(lib1 STATIC ...)
add_library(lib2 ...)
target_link_libraries(lib2 PRIVATE "$<LINK_LIBRARY:WHOLE_ARCHIVE,lib1>")
```

This specifies that **lib2** should link to **lib1** and use the **WHOLE_ARCHIVE** feature when doing so.

Feature names are case−sensitive and may only contain letters, numbers and underscores. Feature names defined in all uppercase are reserved for CMake's own built−in features. The pre−defined built−in library features are:

DEFAULT
This feature corresponds to standard linking, essentially equivalent to using no feature at all. It is typically only used with the*LINK_LIBRAR Y_OVERRIDE* and *LINK_LIBRARY_OVERRIDE_*<*LIBRARY*> target properties.

WHOLE_ARCHIVE
Force inclusion of all members of a static library when linked as a dependency of consuming *Executables*, *Shared Libraries*, and *Module Libraries*. This feature is only supported for the following platforms, with limitations as noted:

- Linux.
- All BSD variants.
- SunOS.
- All Apple variants. The library must be specified as a CMake target name, a library file name (such as **libfoo.a**), or a library file path (such as **/path/to/libfoo.a**). Due to a limitation of the Apple linker, it cannot be specified as a plain library name like **foo**, where **foo** is not a CMake target.
- Windows. When using a MSVC or MSVC−like toolchain, the MSVC version must be greater than 1900.
- Cygwin.
- MSYS.

NOTE:
Since *Static Libraries* are archives and not linked binaries, CMake records their link dependencies for transitive use when linking consuming binaries. Therefore **WHOLE_ARCHIVE** does not cause a static library's objects to be included in other static libraries. Use *Object Libraries* for that.

FRAMEWORK
This option tells the linker to search for the specified framework using the **−framework** linker option. It can only be used on Apple platforms, and only with a linker that understands the option used (i.e. the linker provided with Xcode, or one compatible with it).

The framework can be specified as a CMake framework target, a bare framework name, or a file path. If a target is given, that target must have the *FRAMEWORK* target property set to true. For a file path, if it contains a directory part, that directory will be added as a framework search path.

```
add_library(lib SHARED ...)
target_link_libraries(lib PRIVATE "$<LINK_LIBRARY:FRAMEWORK,/path/to/m

# The constructed linker command line will contain:
#   -F/path/to -framework my_framework
```

File paths must conform to one of the following patterns (***** is a wildcard, and optional parts are shown as **[...]**):

- **[/path/to/]FwName[.framework]**
- **[/path/to/]FwName.framework/FwName[suffix]**
- **[/path/to/]FwName.framework/Versions/*/FwName[suffix]**

Note that CMake recognizes and automatically handles framework targets, even without using the *$<LINK_LIBRARY:FRAMEWORK,...>* expression. The generator expression can still be used with a CMake target if the project wants to be explicit about it, but it is not required to do so. The linker command line may have some differences between using the generator expression or not, but the final result should be the same. On the other hand, if a file path is given, CMake will recognize some paths automatically, but not all cases. The project may want to use *$<LINK_LIBRARY:FRAMEWORK,...>* for file paths so that the expected behavior is clear.

Added in version 3.25: The *FRAMEWORK_MULTI_CONFIG_POSTFIX_<CONFIG>* target property as well as the **suffix** of the framework library name are now supported by the **FRAMEWORK** features.

NEEDED_FRAMEWORK
This is similar to the **FRAMEWORK** feature, except it forces the linker to link with the framework even if no symbols are used from it. It uses the **-needed_framework** option and has the same linker constraints as **FRAMEWORK**.

REEXPORT_FRAMEWORK
This is similar to the **FRAMEWORK** feature, except it tells the linker that the framework should be available to clients linking to the library being created. It uses the **-reexport_framework** option and has the same linker constraints as **FRAMEWORK**.

WEAK_FRAMEWORK
This is similar to the **FRAMEWORK** feature, except it forces the linker to mark the framework and all references to it as weak imports. It uses the **-weak_framework** option and has the same linker constraints as **FRAMEWORK**.

NEEDED_LIBRARY
This is similar to the **NEEDED_FRAMEWORK** feature, except it is for use with non-framework targets or libraries (Apple platforms only). It uses the **-needed_library** or **-needed-l** option as appropriate, and has the same linker constraints as **NEEDED_FRAMEWORK**.

REEXPORT_LIBRARY
This is similar to the **REEXPORT_FRAMEWORK** feature, except it is for use with non-framework targets or libraries (Apple platforms only). It uses the **-reexport_library** or **-reexport-l** option as appropriate, and has the same linker constraints as **REEXPORT_FRAMEWORK**.

WEAK_LIBRARY
This is similar to the **WEAK_FRAMEWORK** feature, except it is for use with non-framework targets or libraries (Apple platforms only). It uses the **-weak_library** or **-weak-l** option as appropriate, and has the same linker constraints as **WEAK_FRAMEWORK**.

Built-in and custom library features are defined in terms of the following variables:

- *CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED*
- *CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>*

- *CMAKE_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED*
- *CMAKE_LINK_LIBRARY_USING_<FEATURE>*

The value used for each of these variables is the value as set at the end of the directory scope in which the target was created. The usage is as follows:

1. If the language−specific *CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED* variable is true, the **feature** must be defined by the corresponding *CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>* variable.
2. If no language−specific**featur e** is supported, then the *CMAKE_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED* variable must be true and the **feature** must be defined by the corresponding *CMAKE_LINK_LIBRARY_USING_<FEATURE>* variable.

The following limitations should be noted:

- The **library−list** can specify CMake targets or libraries. Any CMake target of type *OBJECT* or *INTERFACE* will ignore the feature aspect of the expression and instead be linked in the standard way.
- The **$<LINK_LIBRARY:...>** generator expression can only be used to specify link libraries. In practice, this means it can appear in the *LINK_LIBRARIES*, *INTERFACE_LINK_LIBRARIES*, and *INTERFACE_LINK_LIBRARIES_DIRECT* target properties, and be specified in *target_link_libraries()* and *link_libraries()* commands.
- If a **$<LINK_LIBRARY:...>** generator expression appears in the *INTERFACE_LINK_LIBRARIES* property of a target, it will be included in the imported target generated by a *install(EXPORT)* command. It is the responsibility of the environment consuming this import to define the link feature used by this expression.
- Each target or library involved in the link step must have at most only one kind of library feature. The absence of a feature is also incompatible with all other features. For example:

```
add_library(lib1 ...)
add_library(lib2 ...)
add_library(lib3 ...)

# lib1 will be associated with feature1
target_link_libraries(lib2 PUBLIC "$<LINK_LIBRARY:feature1,lib1>")

# lib1 is being linked with no feature here. This conflicts with the
# use of feature1 in the line above and would result in an error.
target_link_libraries(lib3 PRIVATE lib1 lib2)
```

  Where it isn't possible to use the same feature throughout a build for a given target or library, the *LINK_LIBRARY_OVERRIDE* and *LINK_LIBRARY_OVERRIDE_<LIBRARY>* target properties can be used to resolve such incompatibilities.
- The **$<LINK_LIBRARY:...>** generator expression does not guarantee that the list of specified targets and libraries will be kept grouped together. To manage constructs like **−−start−group** and **−−end−group**, as supported by the GNU **ld** linker, use the *LINK_GROUP* generator expression instead.

$<LINK_GROUP:feature,library−list>
Added in version 3.24.

Specify a group of libraries to link to a target, along with a **feature** which defines how that group should be linked. For example:

```
add_library(lib1 STATIC ...)
add_library(lib2 ...)
target_link_libraries(lib2 PRIVATE "$<LINK_GROUP:RESCAN,lib1,external>")
```

This specifies that **lib2** should link to **lib1** and **external**, and that both of those two libraries should be included on the linker command line according to the definition of the **RESCAN** feature.

Feature names are case−sensitive and may only contain letters, numbers and underscores. Feature names defined in all uppercase are reserved for CMake's own built−in features. Currently, there is only one pre−defined built−in group feature:

RESCAN

Some linkers are single−pass only. For such linkers, circular references between libraries typically result in unresolved symbols. This feature instructs the linker to search the specified static libraries repeatedly until no new undefined references are created.

Normally, a static library is searched only once in the order that it is specified on the command line. If a symbol in that library is needed to resolve an undefined symbol referred to by an object in a library that appears later on the command line, the linker would not be able to resolve that reference. By grouping the static libraries with the **RESCAN** feature, they will all be searched repeatedly until all possible references are resolved. This will use linker options like **−−start−group** and **−−end−group**, or on SunOS, **−z rescan−start** and **−z rescan−end**.

Using this feature has a significant performance cost. It is best to use it only when there are unavoidable circular references between two or more static libraries.

This feature is available when using toolchains that target Linux, BSD, and SunOS. It can also be used when targeting Windows platforms if the GNU toolchain is used.

Built−in and custom group features are defined in terms of the following variables:

- *CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>_SUPPORTED*
- *CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>*
- *CMAKE_LINK_GROUP_USING_<FEATURE>_SUPPORTED*
- *CMAKE_LINK_GROUP_USING_<FEATURE>*

The value used for each of these variables is the value as set at the end of the directory scope in which the target was created. The usage is as follows:

1. If the language−specific *CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>_SUPPORTED* variable is true, the **feature** must be defined by the corresponding *CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>* variable.
2. If no language−specific**featur e** is supported, then the *CMAKE_LINK_GROUP_USING_<FEATURE>_SUPPORTED* variable must be true and the **feature** must be defined by the corresponding *CMAKE_LINK_GROUP_USING_<FEATURE>* variable.

The **LINK_GROUP** generator expression is compatible with the *LINK_LIBRARY* generator expression. The libraries involved in a group can be specified using the *LINK_LIBRARY* generator expression.

Each target or external library involved in the link step is allowed to be part of multiple groups, but only if all the groups involved specify the same **feature**. Such groups will not be merged on the linker command line, the individual groups will still be preserved. Mixing different group features for the same target or library is forbidden.

```
add_library(lib1 ...)
add_library(lib2 ...)
add_library(lib3 ...)
add_library(lib4 ...)
add_library(lib5 ...)

target_link_libraries(lib3 PUBLIC  "$<LINK_GROUP:feature1,lib1,lib2>")
target_link_libraries(lib4 PRIVATE "$<LINK_GROUP:feature1,lib1,lib3>")
# lib4 will be linked with the groups {lib1,lib2} and {lib1,lib3}.
# Both groups specify the same feature, so this is fine.

target_link_libraries(lib5 PRIVATE "$<LINK_GROUP:feature2,lib1,lib3>")
# An error will be raised here because both lib1 and lib3 are part of two
# groups with different features.
```

When a target or an external library is involved in the link step as part of a group and also as not part of any group, any occurrence of the non−group link item will be replaced by the groups it belongs to.

```
add_library(lib1 ...)
add_library(lib2 ...)
add_library(lib3 ...)
add_library(lib4 ...)

target_link_libraries(lib3 PUBLIC lib1)

target_link_libraries(lib4 PRIVATE lib3 "$<LINK_GROUP:feature1,lib1,lib2>")
# lib4 will only be linked with lib3 and the group {lib1,lib2}
```

Because **lib1** is part of the group defined for **lib4**, that group then gets applied back to the use of **lib1** for **lib3**. The end result will be as though the linking relationship for**lib3** had been specified as:

```
target_link_libraries(lib3 PUBLIC "$<LINK_GROUP:feature1,lib1,lib2>")
```

Be aware that the precedence of the group over the non−group link item can result in circular dependencies between groups. If this occurs, a fatal error is raised because circular dependencies are not allowed for groups.

```
add_library(lib1A ...)
add_library(lib1B ...)
add_library(lib2A ...)
add_library(lib2B ...)
add_library(lib3 ...)

# Non−group linking relationships, these are non−circular so far
target_link_libraries(lib1A PUBLIC lib2A)
target_link_libraries(lib2B PUBLIC lib1B)
```

```
# The addition of these groups creates circular dependencies
target_link_libraries(lib3 PRIVATE
  "$<LINK_GROUP:feat,lib1A,lib1B>"
  "$<LINK_GROUP:feat,lib2A,lib2B>"
)
```

Because of the groups defined for **lib3**, the linking relationships for **lib1A** and **lib2B** effectively get expanded to the equivalent of:

```
target_link_libraries(lib1A PUBLIC "$<LINK_GROUP:feat,lib2A,lib2B>")
target_link_libraries(lib2B PUBLIC "$<LINK_GROUP:feat,lib1A,lib1B>")
```

This creates a circular dependency between groups: **lib1A −−> lib2B −−> lib1A**.

The following limitations should also be noted:

- The **library−list** can specify CMake targets or libraries. Any CMake target of type *OBJECT* or *INTERFACE* will ignore the feature aspect of the expression and instead be linked in the standard way.
- The **$<LINK_GROUP:...>** generator expression can only be used to specify link libraries. In practice, this means it can appear in the*LINK_LIBRARIES*, *INTERFACE_LINK_LIBRARIES*,and *INTERFACE_LINK_LIBRARIES_DIRECT* target properties, and be specified in *target_link_libraries()* and *link_libraries()* commands.
- If a **$<LINK_GROUP:...>** generator expression appears in the *INTERFACE_LINK_LIBRARIES* property of a target, it will be included in the imported target generated by a *install(EXPORT)* command. It is the responsibility of the environment consuming this import to define the link feature used by this expression.

Link Context

$<LINK_ONLY:...>

Added in version 3.1.

Content of **...**, except while collecting usage requirements from *transitive compile properties*, in which case it is the empty string. This is intended for use in an *INTERFACE_LINK_LIBRARIES* target property, typically populated via the *target_link_libraries()* command, to specify private link dependencies without other usage requirements such as include directories or compile options.

Added in version 3.24: **LINK_ONLY** may also be used in a *LINK_LIBRARIES* target property. See policy *CMP0131*.

$<DEVICE_LINK:list>

Added in version 3.18.

Returns the list if it is the device link step, an empty list otherwise. The device link step is controlled by *CUDA_SEPARABLE_COMPILATION* and *CUDA_RESOLVE_DEVICE_SYMBOLS* properties and policy *CMP0105*. This expression can only be used to specify link options.

$<HOST_LINK:list>

Added in version 3.18.

Returns the list if it is the normal link step, an empty list otherwise. This expression is mainly useful when a device link step is also involved (see *$<DEVICE_LINK:list>* generator expression).

This expression can only be used to specify link options.

Target−Dependent Expressions

Target Meta−Data

These expressions look up information about a target.

$<TARGET_EXISTS:tgt>

Added in version 3.12.

1 if **tgt** exists as a CMake target, else **0**.

$<TARGET_NAME_IF_EXISTS:tgt>

Added in version 3.12.

The target name **tgt** if the target exists, an empty string otherwise.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_NAME:tgt>

The target name **tgt** as written. This marks **tgt** as being the name of a target inside a larger expression, which is required if exporting targets to multiple dependent export sets. The **tgt** text must be a literal name of a target; it may not contain generator expressions. The target does not have to exist.

$<TARGET_POLICY:policy>

1 if the **policy** was **NEW** when the 'head' target was created, else **0**. If the**policy** w as not set, the warning message for the policy will be emitted. This generator expression only works for a subset of policies.

Target Properties

These expressions look up the values of *target properties*.

$<TARGET_PROPERTY:tgt,prop>

Value of the property **prop** on the target **tgt**, or empty if the property is not set.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

Changed in version 3.26: When encountered during evaluation of *Target Usage Requirements*, typically in an **INTERFACE_*** target property, lookup of the **tgt** name occurs in the directory of the target specifying the requirement, rather than the directory of the consuming target for which the expression is being evaluated.

$<TARGET_PROPERTY:prop>

Value of the property **prop** on the target for which the expression is being evaluated, or empty if the property is not set. Note that for generator expressions in *Target Usage Requirements* this is the consuming target rather than the target specifying the requirement.

The expressions have special evaluation rules for some properties:

Target Build Specification Properties

These evaluate as a *semicolon−separated list* representing the union of the value on the target itself with the values of the corresponding *Target Usage Requirements* on targets named by the target's *LINK_LIBRARIES*:

- For *Target Compile Properties*, evaluation of corresponding usage requirements is transitive over the closure of the linked targets' *INTERFACE_LINK_LIBRARIES excluding* entries guarded by the *LINK_ONLY* generator expression.

- For *Target Link Properties*, evaluation of corresponding usage requirements is transitive over the closure of the linked targets' *INTERFACE_LINK_LIBRARIES including* entries guarded by the *LINK_ONLY* generator expression. See policy *CMP0166*.

Evaluation of *LINK_LIBRARIES* itself is not transitive.

Target Usage Requirement Properties

These evaluate as a *semicolon−separated list* representing the union of the value on the target itself with the values of the same properties on targets named by the target's *INTERFACE_LINK_LIBRARIES*:

- For *Transitive Compile Properties*, evaluation is transitive over the closure of the target's *INTERFACE_LINK_LIBRARIES excluding* entries guarded by the *LINK_ONLY* generator expression.
- For *Transitive Link Properties*, evaluation is transitive over the closure of the target's *INTERFACE_LINK_LIBRARIES including* entries guarded by the *LINK_ONLY* generator expression. See policy *CMP0166*.

Evaluation of *INTERFACE_LINK_LIBRARIES* itself is not transitive.

Custom Transitive Properties

Added in version 3.30.

These are processed during evaluation as follows:

- Evaluation of *$<TARGET_PROPERTY:tgt,PROP>* for some property **PROP**, named without an **INTERFACE_** prefix, checks the *TRANSITIVE_COMPILE_PROPERTIES* and *TRANSITIVE_LINK_PROPERTIES* properties on target **tgt**, on targets named by its *LINK_LIBRARIES*, and on the transitive closure of targets named by the linked targets' *INTERFACE_LINK_LIBRARIES*.

  If **PROP** is listed by one of those properties, then it evaluates as a *semicolon−separated list* representing the union of the value on the target itself with the values of the corresponding **INTERFACE_PROP** on targets named by the target's *LINK_LIBRARIES*:

  - If **PROP** is named by *TRANSITIVE_COMPILE_PROPERTIES*, evaluation of the corresponding **INTERFACE_PROP** is transitive over the closure of the linked targets' *INTERFACE_LINK_LIBRARIES*, excluding entries guarded by the *LINK_ONLY* generator expression.
  - If **PROP** is named by *TRANSITIVE_LINK_PROPERTIES*, evaluation of the corresponding **INTERFACE_PROP** is transitive over the closure of the linked targets' *INTERFACE_LINK_LIBRARIES*, including entries guarded by the *LINK_ONLY* generator expression.

- Evaluation of *$<TARGET_PROPERTY:tgt,INTERFACE_PROP>* for some property **INTERFACE_PROP**, named with an**INTERF ACE_** prefix, checks the *TRANSITIVE_COMPILE_PROPERTIES* and *TRANSITIVE_LINK_PROPERTIES* properties on target **tgt**, and on the transitive closure of targets named by its *INTERFACE_LINK_LIBRARIES*.

  If the corresponding **PROP** is listed by one of those properties, then **INTERFACE_PROP** evaluates as a *semicolon−separated list* representing the union of the value on the target itself with the value of the same property on targets named by the target's *INTERFACE_LINK_LIBRARIES*:

  - If **PROP** is named by *TRANSITIVE_COMPILE_PROPERTIES*, evaluation of the corresponding **INTERFACE_PROP** is transitive over the closure of the target's *INTERFACE_LINK_LIBRARIES*, excluding entries guarded by the *LINK_ONLY* generator expression.

- If **PROP** is named by *TRANSITIVE_LINK_PROPERTIES*, evaluation of the corresponding **INTERFACE_PROP** is transitive over the closure of the target's *INTERFACE_LINK_LIBRARIES*, including entries guarded by the *LINK_ONLY* generator expression.

If a **PROP** is named by both *TRANSITIVE_COMPILE_PROPERTIES* and *TRANSITIVE_LINK_PROPERTIES*, the latter takes precedence.

Compatible Interface Properties

These evaluate as a single value combined from the target itself, from targets named by the target's *LINK_LIBRARIES*, and from the transitive closure of the linked targets' *INTERFACE_LINK_LIBRARIES*. Values of a compatible interface property from multiple targets combine based on the type of compatibility required by the **COMPATIBLE_INTERFACE_*** property defining it.

Target Artifacts

These expressions look up information about artifacts associated with a given target **tgt**. Unless otherwise stated, this can be any runtime artifact, namely:

- An executable target created by *add_executable()*.
- A shared library target (**.so**, **.dll** but not their **.lib** import library) created by *add_library()*.
- A static library target created by *add_library()*.

In the following, the phrase "the **tgt** filename" means the name of the **tgt** binary file. This has to be distinguished from the phrase "the target name", which is just the string **tgt**.

$<TARGET_FILE:tgt>

Full path to the **tgt** binary file.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on, unless the expression is being used in *add_custom_command()* or *add_custom_target()*.

$<TARGET_FILE_BASE_NAME:tgt>

Added in version 3.15.

Base name of **tgt**, i.e. **$<TARGET_FILE_NAME:tgt>** without prefix and suffix. For example, if the **tgt** filename is **libbase.so**, the base name is **base**.

See also the *OUTPUT_NAME*, *ARCHIVE_OUTPUT_NAME*, *LIBRARY_OUTPUT_NAME* and *RUNTIME_OUTPUT_NAME* target properties and their configuration specific variants *OUTPUT_NAME_<CONFIG>*, *ARCHIVE_OUTPUT_NAME_<CONFIG>*, *LIBRARY_OUTPUT_NAME_<CONFIG>* and *RUNTIME_OUTPUT_NAME_<CONFIG>*.

The *<CONFIG>_POSTFIX* and *DEBUG_POSTFIX* target properties can also be considered.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_FILE_PREFIX:tgt>

Added in version 3.15.

Prefix of the **tgt** filename (such as **lib**).

See also the *PREFIX* target property.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_FILE_SUFFIX:tgt>
Added in version 3.15.

Suffix of the **tgt** filename (extension such as **.so** or **.exe**).

See also the *SUFFIX* target property.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_FILE_NAME:tgt>
The **tgt** filename.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on (see policy *CMP0112*).

$<TARGET_FILE_DIR:tgt>
Directory of the **tgt** binary file.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on (see policy *CMP0112*).

$<TARGET_IMPORT_FILE:tgt>
Added in version 3.27.

Full path to the linker import file. On DLL platforms, it would be the **.lib** file. For executables on AIX, and for shared libraries on macOS, it could be, respectively, the **.imp** or **.tbd** import file, depending on the value of the *ENABLE_EXPORTS* property.

This expands to an empty string when there is no import file associated with the target.

$<TARGET_IMPORT_FILE_BASE_NAME:tgt>
Added in version 3.27.

Base name of the linker import file of the target **tgt** without prefix or suffix. For example, if the target file name is **libbase.tbd**, the base name is **base**.

See also the *OUTPUT_NAME* and *ARCHIVE_OUTPUT_NAME* target properties and their configuration specific variants *OUTPUT_NAME_<CONFIG>* and *ARCHIVE_OUTPUT_NAME_<CONFIG>*.

The *<CONFIG>_POSTFIX* and *DEBUG_POSTFIX* target properties can also be considered.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_IMPORT_FILE_PREFIX:tgt>
Added in version 3.27.

Prefix of the import file of the target **tgt**.

See also the *IMPORT_PREFIX* target property.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_IMPORT_FILE_SUFFIX:tgt>
Added in version 3.27.

Suffix of the import file of the target **tgt**.

The suffix corresponds to the file extension (such as **.lib** or **.tbd**).

See also the *IMPORT_SUFFIX* target property.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_IMPORT_FILE_NAME:tgt>
Added in version 3.27.

Name of the import file of the target **tgt**.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_IMPORT_FILE_DIR:tgt>
Added in version 3.27.

Directory of the import file of the target **tgt**.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_FILE:tgt>
File used when linking to the **tgt** target. This will usually be the library that**tgt** represents (**.a**, **.lib**, **.so**), but for a shared library on DLL platforms, it would be the **.lib** import library associated with the DLL.

Added in version 3.27: On macOS, it could be the **.tbd** import file associated with the shared library, depending on the value of the *ENABLE_EXPORTS* property.

This generator expression is equivalent to *$<TARGET_LINKER_LIBRARY_FILE>* or *$<TARGET_LINKER_IMPORT_FILE>* generator expressions, depending on the characteristics of the target and the platform.

$<TARGET_LINKER_FILE_BASE_NAME:tgt>
Added in version 3.15.

Base name of file used to link the target **tgt**, i.e. *$<TARGET_LINKER_FILE_NAME:tgt>* without prefix and suffix. For example, if target file name is **libbase.a**, the base name is **base**.

See also the *OUTPUT_NAME*, *ARCHIVE_OUTPUT_NAME*, and *LIBRARY_OUTPUT_NAME* target properties and their configuration specific variants *OUTPUT_NAME_<CONFIG>*, *ARCHIVE_OUTPUT_NAME_<CONFIG>* and *LIBRARY_OUTPUT_NAME_<CONFIG>*.

The *<CONFIG>_POSTFIX* and *DEBUG_POSTFIX* target properties can also be considered.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_FILE_PREFIX:tgt>
Added in version 3.15.

Prefix of file used to link target **tgt**.

See also the *PREFIX* and *IMPORT_PREFIX* target properties.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_FILE_SUFFIX:tgt>
Added in version 3.15.

Suffix of file used to link where **tgt** is the name of a target.

The suffix corresponds to the file extension (such as ".so" or ".lib").

See also the *SUFFIX* and *IMPORT_SUFFIX* target properties.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_FILE_NAME:tgt>
Name of file used to link target **tgt**.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on (see policy *CMP0112*).

$<TARGET_LINKER_FILE_DIR:tgt>
Directory of file used to link target **tgt**.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on (see policy *CMP0112*).

$<TARGET_LINKER_LIBRARY_FILE:tgt>
Added in version 3.27.

File used when linking o the **tgt** target is done using directly the library, and not an import file. This will usually be the library that **tgt** represents (**.a**, **.so**, **.dylib**). So, on DLL platforms, it will be an empty string.

$<TARGET_LINKER_LIBRARY_FILE_BASE_NAME:tgt>
Added in version 3.27.

Base name of library file used to link the target **tgt**, i.e. *$<TARGET_LINKER_LIBRARY_FILE_NAME:tgt>* without prefix and suffix. For example, if target file name is **libbase.a**, the base name is **base**.

See also the *OUTPUT_NAME*, *ARCHIVE_OUTPUT_NAME*, and *LIBRARY_OUTPUT_NAME* target properties and their configuration specific variants *OUTPUT_NAME_<CONFIG>*, *ARCHIVE_OUTPUT_NAME_<CONFIG>* and *LIBRARY_OUTPUT_NAME_<CONFIG>*.

The *<CONFIG>_POSTFIX* and *DEBUG_POSTFIX* target properties can also be considered.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_LIBRARY_FILE_PREFIX:tgt>
Added in version 3.27.

Prefix of the library file used to link target **tgt**.

See also the *PREFIX* target property.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_LIBRARY_FILE_SUFFIX:tgt>
Added in version 3.27.

Suffix of the library file used to link target **tgt**.

The suffix corresponds to the file extension (such as ".a" or ".dylib").

See also the *SUFFIX* target property.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_LIBRARY_FILE_NAME:tgt>
Added in version 3.27.

Name of the library file used to link target **tgt**.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_LIBRARY_FILE_DIR:tgt>
Added in version 3.27.

Directory of the library file used to link target **tgt**.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_IMPORT_FILE:tgt>
Added in version 3.27.

File used when linking to the **tgt** target is done using an import file. This will usually be the import file that **tgt** represents (**.lib**, **.tbd**). So, when no import file is involved in the link step, an empty string is returned.

$<TARGET_LINKER_IMPORT_FILE_BASE_NAME:tgt>
Added in version 3.27.

Base name of the import file used to link the target **tgt**, i.e. *$<TARGET_LINKER_IMPORT_FILE_NAME:tgt>* without prefix and suffix. For example, if target file name is **libbase.tbd**, the base name is **base**.

See also the *OUTPUT_NAME* and *ARCHIVE_OUTPUT_NAME*, target properties and their configuration specific variants *OUTPUT_NAME_<CONFIG>* and *ARCHIVE_OUTPUT_NAME_<CONFIG>*.

The *<CONFIG>_POSTFIX* and *DEBUG_POSTFIX* target properties can also be considered.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_IMPORT_FILE_PREFIX:tgt>
Added in version 3.27.

Prefix of the import file used to link target **tgt**.

See also the *IMPORT_PREFIX* target property.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_IMPORT_FILE_SUFFIX:tgt>
Added in version 3.27.

Suffix of the import file used to link target **tgt**.

The suffix corresponds to the file extension (such as ".lib" or ".tbd").

See also the *IMPORT_SUFFIX* target property.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_IMPORT_FILE_NAME:tgt>
Added in version 3.27.

Name of the import file used to link target **tgt**.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_LINKER_IMPORT_FILE_DIR:tgt>
Added in version 3.27.

Directory of the import file used to link target **tgt**.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_SONAME_FILE:tgt>
File with soname (**.so.3**) where **tgt** is the name of a target.

$<TARGET_SONAME_FILE_NAME:tgt>
Name of file with soname (**.so.3**).

Note that **tgt** is not added as a dependency of the target this expression is evaluated on (see policy *CMP0112*).

$<TARGET_SONAME_FILE_DIR:tgt>
Directory of file with soname (**.so.3**).

Note that **tgt** is not added as a dependency of the target this expression is evaluated on (see policy *CMP0112*).

$<TARGET_SONAME_IMPORT_FILE:tgt>
Added in version 3.27.

Import file with soname (**.3.tbd**) where **tgt** is the name of a target.

$<TARGET_SONAME_IMPORT_FILE_NAME:tgt>
Added in version 3.27.

Name of the import file with soname (**.3.tbd**).

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_SONAME_IMPORT_FILE_DIR:tgt>
Added in version 3.27.

Directory of the import file with soname (**.3.tbd**).

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_PDB_FILE:tgt>
Added in version 3.1.

Full path to the linker generated program database file (.pdb) where **tgt** is the name of a target.

See also the *PDB_NAME* and *PDB_OUTPUT_DIRECTORY* target properties and their configuration specific variants *PDB_NAME_<CONFIG>* and *PDB_OUTPUT_DIRECTORY_<CONFIG>*.

$<TARGET_PDB_FILE_BASE_NAME:tgt>
Added in version 3.15.

Base name of the linker generated program database file (.pdb) where **tgt** is the name of a target.

The base name corresponds to the target PDB file name (see**$<TARGET_PDB_FILE_NAME:tgt>**) without prefix and suffix. For example, if target file name is **base.pdb**, the base name is **base**.

See also the *PDB_NAME* target property and its configuration specific variant *PDB_NAME_<CONFIG>*.

The *<CONFIG>_POSTFIX* and *DEBUG_POSTFIX* target properties can also be considered.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on.

$<TARGET_PDB_FILE_NAME:tgt>
Added in version 3.1.

Name of the linker generated program database file (.pdb).

Note that **tgt** is not added as a dependency of the target this expression is evaluated on (see policy *CMP0112*).

$<TARGET_PDB_FILE_DIR:tgt>
Added in version 3.1.

Directory of the linker generated program database file (.pdb).

Note that **tgt** is not added as a dependency of the target this expression is evaluated on (see policy *CMP0112*).

$<TARGET_BUNDLE_DIR:tgt>
Added in version 3.9.

Full path to the bundle directory (**/path/to/my.app**, **/path/to/my.framework**, or **/path/to/my.bundle**), where **tgt** is the name of a target.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on (see policy *CMP0112*).

$<TARGET_BUNDLE_DIR_NAME:tgt>
Added in version 3.24.

Name of the bundle directory (**my.app**, **my.framework**, or **my.bundle**), where **tgt** is the name of a target.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on (see policy *CMP0112*).

$<TARGET_BUNDLE_CONTENT_DIR:tgt>
Added in version 3.9.

Full path to the bundle content directory where **tgt** is the name of a target. For the macOS SDK it leads to **/path/to/my.app/Contents**, **/path/to/my.framework**, or **/path/to/my.bundle/Contents**. For all other SDKs (e.g. iOS) it leads to **/path/to/my.app**, **/path/to/my.framework**, or **/path/to/my.bundle** due to the flat bundle structure.

Note that **tgt** is not added as a dependency of the target this expression is evaluated on (see policy *CMP0112*).

$<TARGET_OBJECTS:tgt>
Added in version 3.1.

List of objects resulting from building **tgt**. This would typically be used on *object library* targets.

$<TARGET_RUNTIME_DLLS:tgt>
Added in version 3.21.

List of DLLs that the target depends on at runtime. This is determined by the locations of all the **SHARED** targets in the target's transitive dependencies. If only the directories of the DLLs are needed, see the *TARGET_RUNTIME_DLL_DIRS* generator expression. Using this generator expression on targets other than executables, **SHARED** libraries, and **MODULE** libraries is an error. **On non−DLL platforms, this expression always evaluates to an empty string**.

This generator expression can be used to copy all of the DLLs that a target depends on into its output directory in a **POST_BUILD** custom command using the *cmake −E copy −t* command. For example:

```
find_package(foo CONFIG REQUIRED) # package generated by install(EXPORT)

add_executable(exe main.c)
target_link_libraries(exe PRIVATE foo::foo foo::bar)
add_custom_command(TARGET exe POST_BUILD
  COMMAND ${CMAKE_COMMAND} −E copy −t $<TARGET_FILE_DIR:exe> $<TARGET_RUNTI
  COMMAND_EXPAND_LISTS
)
```

NOTE:

Imported Targets are supported only if they know the location of their **.dll** files. An imported **SHARED** library must have *IMPORTED_LOCATION* set to its **.dll** file. See the *add_library imported libraries* section for details. Many *Find Modules* produce imported targets with the **UNKNOWN** type and therefore will be ignored.

On platforms that support runtime paths (**RPATH**), refer to the *INSTALL_RPATH* target property. On Apple platforms, refer to the *INSTALL_NAME_DIR* target property.

$<TARGET_RUNTIME_DLL_DIRS:tgt>

Added in version 3.27.

List of the directories which contain the DLLs that the target depends on at runtime (see *TARGET_RUNTIME_DLLS*). This is determined by the locations of all the **SHARED** targets in the target's transitive dependencies. Using this generator expression on targets other than executables, **SHARED** libraries, and **MODULE** libraries is an error. **On non−DLL platforms, this expression always evaluates to an empty string**.

This generator expression can e.g. be used to create a batch file using *file(GENERATE)* which sets the PATH environment variable accordingly.

Export And Install Expressions

$<INSTALL_INTERFACE:...>

Content of **...** when the property is exported using *install(EXPORT)*, and empty otherwise.

$<BUILD_INTERFACE:...>

Content of **...** when the property is exported using *export()*, or when the target is used by another target in the same buildsystem. Expands to the empty string otherwise.

$<BUILD_LOCAL_INTERFACE:...>

Added in version 3.26.

Content of **...** when the target is used by another target in the same buildsystem. Expands to the empty string otherwise.

$<INSTALL_PREFIX>

Content of the install prefix when the target is exported via *install(EXPORT)*, or when evaluated in the *INSTALL_NAME_DIR* property or the **INSTALL_NAME_DIR** argument of *install(RUNTIME_DEPENDENCY_SET)*, and empty otherwise.

Changed in version 3.27: Evaluates to the content of the install prefix in the code argument of *install(CODE)* or the file argument of *install(SCRIPT)*.

Multi−level Expression Evaluation

$<GENEX_EVAL:expr>
Added in version 3.12.

Content of **expr** evaluated as a generator expression in the current context. This enables consumption of generator expressions whose evaluation results itself in generator expressions.

$<TARGET_GENEX_EVAL:tgt,expr>
Added in version 3.12.

Content of **expr** evaluated as a generator expression in the context of **tgt** target. This enables consumption of custom target properties that themselves contain generator expressions.

Having the capability to evaluate generator expressions is very useful when you want to manage custom properties supporting generator expressions. For example:

```
add_library(foo ...)

set_property(TARGET foo PROPERTY
  CUSTOM_KEYS $<$<CONFIG:DEBUG>:FOO_EXTRA_THINGS>
)

add_custom_target(printFooKeys
  COMMAND ${CMAKE_COMMAND} -E echo $<TARGET_PROPERTY:foo,CUSTOM_KEYS>
)
```

This naive implementation of the **printFooKeys** custom command is wrong because **CUSTOM_KEYS** target property is not evaluated and the content is passed as is (i.e. **$<$<CONFIG:DEBUG>:FOO_EXTRA_THINGS>**).

To have the expected result (i.e. **FOO_EXTRA_THINGS** if config is **Debug**), it is required to evaluate the output of **$<TARGET_PROPERTY:foo,CUSTOM_KEYS>**:

```
add_custom_target(printFooKeys
  COMMAND ${CMAKE_COMMAND} -E
    echo $<TARGET_GENEX_EVAL:foo,$<TARGET_PROPERTY:foo,CUSTOM_KEYS>>
)
```

Escaped Characters

These expressions evaluate to specific string literals. Use them in place of the actual string literal where you need to prevent them from having their special meaning.

$<ANGLE-R>
A literal >. Used for example to compare strings that contain a >.

$<COMMA>
A literal **,**. Used for example to compare strings which contain a **,**.

$<SEMICOLON>
A literal **;**. Used to prevent list expansion on an argument with **;**.

$<QUOTE>
Added in version 3.30.

A literal ". Used to allow string literal quotes inside a generator expression.

Deprecated Expressions

$<CONFIGURATION>

Configuration name. Deprecated since CMake 3.0. Use *CONFIG* instead.

COPYRIGHT