Writing CMakeLists Files

This chapter will cover the basics of writing effective CMakeLists files for your software. It will cover all of the basic commands and issues you will need to handle most projects. It will also discuss how to convert existing UNIX or Windows projects into CMakeLists files. While CMake can handle extremely complex projects, for most projects you will find this chapter's contents will tell you all you need to know. CMake is driven by the CMakeLists.txt files written for a software project. The CMakeLists files determine everything from what options to put into the cache, to what source files to compile. In addition to discussing how to write a CMakeLists file this chapter will also cover how to make them robust and maintainable. The basic syntax of a CMakeLists.txt file and key concepts of CMake have already been discussed in chapters 2 and 3. This chapter will expand on those concepts and introduce a few new ones.

4.1 CMake Syntax

CMakeLists files follow a simple syntax consisting of comments, commands, and white space. A comment is indicated using the # character and runs from that character until the end of the line. A command consists of the command name, opening parenthesis, white space separated arguments and a closing parenthesis. All white space (spaces, line feeds, tabs) are ignored except to separate arguments. Anything within a set of double quotes is treated as one argument as is typical for most languages. The backslash can be used to escape characters preventing the normal interpretation of them. The subsequent examples in this chapter will help to clear up some of these syntactic issues. You might wonder why CMake decided to have its own language instead of using an existing one such as Python, Java, or Tcl. The main reason is that we did not want to make CMake require an additional tool to run. By requiring one of these other languages all users of CMake would be required to have that language installed, and potentially a specific version of that language. This is on top of the language

extensions that would be required to do some of the CMake work, for both performance and capability reasons.

4.2 Basic Commands

While the previous chapters have already introduced many of the basic commands for CMakeLists files, this chapter will review and expand on them. The first command the top-level CMakeLists file should have is the `PROJECT` command. This command both names the project and optionally specifies what languages will be used by it. Its syntax is as follows:

```
project (projectname [CXX] [C] [Java] [NONE])
```

If no languages are specified then CMake defaults to supporting C and C++. If the `NONE` language is passed then CMake includes no language specific support. Whenever C++ language support is specified then C language support will also be loaded.

For each project command that appears in a project, CMake will create a top level IDE project file. The project will contain all targets that are in the CMakeLists.txt file, and any of its subdirectories as specified by the `add_subdirectory` command. If the `EXCLUDE_FROM_ALL` option is used in the `add_subdirectory` command, then the generated project will not appear in the top level Makefile or IDE project file. This is useful for generating sub projects that do not make sense as part of the main build process. Consider that a project with a number of examples could use this feature to generate the build files for each example with one run of CMake, but not have the examples built as part of the normal build process.

The `set` command is probably one of the most used commands since it is used for defining and modifying variables and lists. Complimenting the `set` command are the `remove` and `separate_arguments` commands. The `remove` command can be used to remove a value from a variable list, while the `separate_arguments` command can be used to take a single variable value (as opposed to a list) and break it into a list based on spaces.

The `add_executable` and `add_library` commands are the main commands for defining what libraries and executables to build, and what source files comprise them. For Visual Studio projects the source files will show up in the IDE as usual, but any header files the project uses will not be there. To have the header files show up as well you simply add them to the list of source files for the executable or library. This can be done for all generators. Any generators that do not use the header files directly (such as Makefile based generators) will simply ignore them.

4.3 Flow Control

In many ways writing a CMakeLists file is like a writing a program in a simple language. Like most languages CMake provides flow control structures to help you along your way. CMake provides three flow control structures;

- conditional statements (e.g. `if`)
- looping constructs (e.g. `foreach` and `while`)
- procedure definitions (e.g. `macro` and `function`)

First we will consider the `if` command. In many ways the `if` command in CMake is just like the `if` command in any other language. It evaluates its expression and based on that either executes the code in its body or optionally the code in the `else` clause. For example:

```
if (FOO)
  # do something here
else (FOO)
  # do something else
endif (FOO)
```

One difference you might notice is that the conditional of the `if` statement is repeated in the `else` and `endif` clauses. This is optional and in this book you will see examples of both styles. You could just as well choose to write:

```
if (FOO)
  # do something here
else ()
  # do something else
endif ()
```

When you include conditionals in the `else` and `endif` clause they are used to provide additional error checking. As such they must exactly match the original conditional of the `if` statement. The following code would not work:

```
set (FOO 1)

if (${FOO})
  # do something
endif (1)
# ERROR, it doesn't match the original if conditional
```

Fortunately CMake provides verbose error messages in the case where an `if` statement is not properly matched with an `endif`. This should help you to track down any problems with matching conditionals. Providing the conditionals on the `else` and `endif` commands also has the added benefit of helping to document your CMakeLists file. With a long `if` statement it can be easy to lose track of what `if` statement the `endif` is closing. `if` statements can be nested to any depth, and any command can be used inside of an `if` or `else` clause.

As with many other languages, CMake supports `elseif` so that you can sequentially test for multiple conditions. For example:

```
if (MSVC80)
  # do something here
elseif (MSVC90)
  # do something else
elseif (APPLE)
  # do something else
endif ()
```

The `if` command has a limited set of operations that you can use. It does not support general purpose C style expressions such as `${FOO} && ${BAR} || ${FUBAR}`, instead it supports a limited subset of expressions that should work for most cases. Specifically `if` supports:

if (variable)

True if the variable's value is not empty, `0`, `FALSE`, `OFF`, or `NOTFOUND`.

if (NOT variable)

True if the variable's value is empty, `0`, `FALSE`, `OFF`, or `NOTFOUND`

if (variable1 AND variable2)

True if both variables would be considered true individually.

if (variable1 OR variable2)

True if either variable would be considered true individually.

if (COMMAND command-name)

True if the given name is a command that can be invoked.

if (DEFINED variable)

True if the given variable has been set, regardless of what value it was set to.

if (EXISTS file-name)

if (EXISTS directory-name)

True if the named file or directory exists.

if (IS_DIRECTORY name)

if (IS_ABSOLUTE name)

True if the given name is a directory, or absolute path respectively.

if (name1 IS_NEWER_THAN name2)

True if the file specified by name1 has a more recent modification time than the file specified by name2.

if (variable MATCHES regex)

if (string MATCHES regex)

True if the given string or variable's value matches the given regular expression.

Options such as `EQUAL`, `LESS`, and `GREATER` are available for numeric comparisons. `STRLESS`, `STREQUAL`, and `STRGREATER` can be used for lexicographic comparisons. `VERSION_LESS`, `VERSION_EQUAL`, and `VERSION_GREATER` can be used to compare versions of the form `major[.minor[.patch[.tweak]]]`. Similar to C and C++ these expressions can be combined to create more powerful conditionals. For example consider the following conditionals:

```
if ((1 LESS 2) AND (3 LESS 4))
  message ("sequence of numbers")
endif ()

if (1 AND 3 AND 4)
  message ("series of true values")
endif (1 AND 3 AND 4)

if (NOT 0 AND 3 AND 4)
  message ("a false value")
endif (NOT 0 AND 3 AND 4)

if (0 OR 3 AND 4)
  message ("or statements")
endif (0 OR 3 AND 4)

if (EXISTS ${PROJECT_SOURCE_DIR}/Help.txt AND COMMAND IF)
  message ("Help exists")
endif (EXISTS ${PROJECT_SOURCE_DIR}/Help.txt AND COMMAND IF)
```

```
set (fooba 0)

if (NOT DEFINED foobar)
  message ("foobar is not defined")
endif (NOT DEFINED foobar)

if (NOT DEFINED fooba)
  message ("fooba not defined")
endif (NOT DEFINED fooba)
```

In compound `if` statements there is an order of precedence that specifies the order that the operations will be evaluated. For example, in the statement below, the `NOT` will be evaluated first then the `AND`, not the other way around. Thus the statement will be false and the message never printed. Had the `AND` been evaluated first the statement would be true.

```
if (NOT 0 AND 0)
  message ("This line is never executed")
endif (NOT 0 AND 0)
```

CMake defines the order of operations such that parenthetical groups are evaluated first, then `EXISTS`, `COMMAND`, `DEFINED` and similar prefix operators are evaluated, then any `EQUAL`, `LESS`, `GREATER`, `STREQUAL`, `STRLESS`, `STRGREATER`, and `MATCHES` operators. The `NOT` operators are evaluated next, and finally the `AND` and `OR` expressions will be evaluated. With operations that have the same level of precedence, such as `AND` and `OR`, they will be evaluated from left to right. Once all of the expressions have been evaluated the final result will be tested to see if it is true or false. CMake considers any of the following values to be true: `ON`, `1`, `YES`, `TRUE`, `Y`. The following values are all considered to be false: `OFF`, `0`, `NO`, `FALSE`, `N`, `NOTFOUND`, `*-NOTFOUND`, `IGNORE`. This test is case insensitive so `true`, `True`, and `TRUE` are all treated the same.

Now let us consider the other flow control commands. The `foreach`, `while`, `macro`, and `function` commands are the best way to reduce the size of your CMakeLists files and keep them maintainable. The `foreach` command enables you to execute a group of CMake commands repeatedly on the members of a list. Consider the following example adapted from VTK:

```
foreach (tfile
        TestAnisotropicDiffusion2D
        TestButterworthLowPass
        TestButterworthHighPass
        TestCityBlockDistance
```

```
        TestConvolve
        )
  add_test(${tfile}-image ${VTK_EXECUTABLE}
    ${VTK_SOURCE_DIR}/Tests/rtImageTest.tcl
    ${VTK_SOURCE_DIR}/Tests/${tfile}.tcl
    -D ${VTK_DATA_ROOT}
    -V Baseline/Imaging/${tfile}.png
    -A ${VTK_SOURCE_DIR}/Wrapping/Tcl
    )
endforeach ( tfile )
```

The first argument of the `foreach` command is the name of the variable that will take on a different value with each iteration of the loop. The remaining arguments are the list of values over which to loop. In this example the body of the `foreach` loop is just one CMake command, `add_test`. In the body of the `foreach` loop any time the loop variable (`tfile` in this example) is referenced it will be replaced with the current value from the list. In the first iteration, occurrences of `${tfile}` will be replaced with `TestAnisotropicDiffusion2D`. In the next iteration, `${tfile}` will be replaced with `TestButterworthLowPass`. The `foreach` loop will continue to loop until all of the arguments have been processed.

It is worth mentioning that `foreach` loops can be nested and that the loop variable is replaced prior to any other variable expansion. This means that in the body of a `foreach` loop you can construct variable names using the loop variable. In the code below the loop variable `tfile` is expanded, and then concatenated with `_TEST_RESULT`. That new variable name is then expanded and tested to see if it matches `FAILED`.

```
if (${${tfile}}_TEST_RESULT} MATCHES FAILED)
  message ("Test ${tfile} failed.")
endif ()
```

The `while` command provides for looping based on a test condition. The format for the test expression in the `while` command is the same as that for the `if` command described earlier. Consider the following example, which is used by CTest. Note that CTest updates the value of CTEST_ELAPSED_TIME internally.

```
#####################################################
# run paraview and ctest test dashboards for 6 hours
#
while (${CTEST_ELAPSED_TIME} LESS 36000)
  set (START_TIME ${CTEST_ELAPSED_TIME})
  ctest_run_script ( "dash1_ParaView_vs71continuous.cmake" )
  ctest_run_script ( "dash1_cmake_vs71continuous.cmake" )
```

```
endwhile ()
```

The `foreach` and `while` commands allow you to handle repetitive tasks that occur in sequence, whereas the `macro` and `function` commands support repetitive tasks that may be scattered throughout your CMakeLists files. Once a macro or function is defined it can be used by any CMakeLists files processed after its definition.

A function in CMake is very much like a function in C or C++. You can pass arguments into it, and the arguments passed in become variables within the function. Likewise some standard variables such as `ARGC`, `ARGV`, `ARGN`, and `ARGV0`, `ARGV1`, etc are defined. Within a function you are in a new variable scope, much like when you drop into a subdirectory using the `add_subdirectory` command you are in a new variable scope. All the variables that were defined when the function was called are still defined, but any changes to variables or new variables only exist within the function. When the function returns those variables will go away. Put more simply, when you invoke a function a new variable scope is pushed and when it returns that variable scope is popped.

The first argument is the name of the function to define. All additional arguments are formal parameters to the function.

```
function(DetermineTime _time)
  # pass the result up to whatever invoked this
  set (${_time} "1:23:45" PARENT_SCOPE)
endfunction()

# now use the function we just defined
DetermineTime( current_time )

if( DEFINED current_time )
  message(STATUS "The time is now: ${curret_time}")
endif()
```

Note that in this example `_time` is used to pass the name of the return variable. The `set` command is invoked with the value of `_time`, which in this example will be `current_time`. Finally the `set` command uses the `PARENT_SCOPE` option to set that variable in the parent's scope instead of the local scope.

Macros are defined and called in the same manner as functions. The main differences are that a macro does not push and pop a new variable scope, and the arguments to a macro are not treated as variables but are string replaced prior to execution. This is very much like the differences between a macro and a function in C or C++. The first argument is the name of the macro to create. All additional arguments are formal parameters to the macro[.

```
# define a simple macro

macro (assert TEST COMMENT)
  if (NOT ${TEST})
    message ("Assertion failed: ${COMMENT}")
  endif (NOT ${TEST})
endmacro (assert)

# use the macro
find_library (FOO_LIB foo /usr/local/lib)
assert ( ${FOO_LIB} "Unable to find library foo" )
```

The simple example above creates a macro called `assert`. The macro is defined to take two arguments. The first argument is a value to test and the second argument is a comment to print out if the test fails. The body of the macro is a simple `if` command with a `message` command inside of it. The macro body ends when the `endmacro` command is found. The macro can be invoked simply by using its name as if it were a command. In the above example if `FOO_LIB` was not found a message would be displayed indicating the error condition.

The `macro` command also supports defining macros that take variable argument lists. This can be useful if you want to define a macro that has optional arguments or multiple signatures. Variable arguments can be referenced using `ARGC` and `ARGV0`, `ARGV1`, etc., instead of the formal parameters. `ARGV0` represents the first argument to the macro, `ARGV1` represents the next, and so forth. You can even use a mixture of formal arguments and variable arguments, as shown in the example below.

```
# define a macro that takes at least two arguments
# (the formal arguments) plus an optional third argument

macro (assert TEST COMMENT)
  if (NOT ${TEST})
    message ("Assertion failed: ${COMMENT}")

    # if called with three arguments then also write the
    # message to a file specified as the third argument
    if (${ARGC} MATCHES 3)
      file (APPEND ${ARGV2} "Assertion failed: ${COMMENT}")
    endif (${ARGC} MATCHES 3)

  endif (NOT ${TEST})
endmacro (ASSERT)
```

```
# use the macro
find_library (FOO_LIB foo /usr/local/lib)
assert ( ${FOO_LIB} "Unable to find library foo" )
```

In this example the two required arguments are TEST and COMMENT. These required arguments can be referenced by name, as they are in this example, or they can be referenced using ARGV0 and ARGV1. If you want to process the arguments as a list you can use the ARGV and ARGN variables. ARGV (as opposed to ARGV0, ARGV1, etc) is a list of all the arguments to the macro, while ARGN is a list of all the arguments after the formal arguments. Inside your macro you can use the foreach command to iterate over ARGV or ARGN as desired.

CMake has two commands for interrupting the processing flow. The break command will break out of a foreach or while loop before it would normally end. The return command will return from a function or listfile before the function or listfile has reached its end.

4.4 Regular Expressions

A few CMake commands, such as if and string, make use of regular expressions, or can take a regular expression as an argument. In its simplest form, a regular-expression is a sequence of characters used to search for exact character matches. However, many times the exact sequence to be found is not known, or only a match at the beginning or end of a string is desired. Since there are several different conventions for specifying regular expressions CMake's standard is described below. The description is based on the open source regular expression class from Texas Instruments, which is used by CMake for parsing regular expressions.

Regular expressions can be specified by using combinations of standard alphanumeric characters and the following regular expression meta-characters:

^ Matches at beginning of a line or string.

$ Matches at end of a line or string.

. Matches any single character other than a newline.

[ ] Matches any character(s) inside the brackets.

[^] Matches any character(s) not inside the brackets.

[-] Matches any character in range on either side of a dash.

* Matches preceding pattern zero or more times.

+ Matches preceding pattern one or more times.

? Matches preceding pattern zero or once only.

() Saves a matched expression and uses it in a later replacement.

(|) Matches either the left or right side of the bar.

Note that more than one of these meta-characters can be used in a single regular expression in order to create complex search patterns. For example, the pattern [^ab1-9] indicates to match any character sequence that does not begin with the characters "a" or "b" or numbers in the series one through nine. The following examples may help clarify regular expression usage:

- The regular expression "^hello" matches a "hello" only at the beginning of a search string. It would match "hello there", but not "hi,\nhello there".
- The regular expression "long$" matches a "long" only at the end of a search string. It would match "so long", but not "long ago".
- The regular expression "t..t..g" will match anything that has a "t", then any two characters, another "t", any two characters, and then a "g". It would match "testing" or "test again", but would not match "toasting".
- The regular expression "[1-9ab]" matches any number one through nine, and the characters "a" and "b". It would match "hello 1" or "begin", but would not match "no-match".
- The regular expression "[^1-9ab]" matches any character that is not a number one through nine, or an "a" or "b". It would NOT match "1ab2" or "b2345a", but would match "no-match".
- The regular expression "br* " matches something that begins with a "b", is followed by zero or more "r"s, and ends in a space. It would match "brrrrr " and "b ", but would not match "brrh ".
- The regular expression "br+ " matches something that begins with a "b", is followed by one or more "r"s, and ends in a space. It would match "brrrrr ", and "br ", but would not match "b " or "brrh ".
- The regular expression "br? " matches something that begins with a "b", is followed by zero or one "r"s, and ends in a space. It would match "br ", and "b ", but would not match "brrrr " or "brrh ".
- The regular expression "(..p)b" matches something ending with pb and beginning with whatever the two characters before the first p encountered in the line were. It would find "repb" in "rep drepaqrepb". The regular expression "(..p)a" would find "repa qrepb" in "rep drepa qrepb"

- The regular expression "d(..p)" matches something ending with p, beginning with d, and having two characters in between that are the same as the two characters before the first p encountered in the line. It would match "drepa qrepb" in "rep drepa qrepb".

4.5 Checking Versions of CMake

CMake is an evolving program and as new versions are released, new features or commands may be introduced. As a result, there may be instances where you might want to use a command that is in a current version of CMake but not in previous versions. There are a couple of ways to handle this. One option is to use the `if` command to check whether a new command exists. For example:

```
# test if the command exists

if (COMMAND some_new_command)
  # use the command
  some_new_command ( ARGS...)
endif (COMMAND some_new_command)
```

The above approach should work in most cases, but if you need more information you can test against the actual version of CMake that is being run by evaluating the `CMAKE_VERSION` variables, as in the following example:

```
# look for newer versions of CMake

if (${CMAKE_VERSION} VERSION_GREATER 1.6.1)
  # do something special here
endif ()
```

When writing your CMakeLists files you might decide that you do not want to support old versions of CMake. To do this you can place the following command at the top of your CMakeLists file:

```
cmake_minimum_required (VERSION 2.2)
```

This indicates that the person running CMake on your project must have at least CMake version 2.2. If they are running an older version of CMake then an error message will be displayed telling them that the project requires at least the specified version of CMake.

Finally, in some cases a new release of CMake might come out that no longer supports some commands you were using (although we try to avoid this). In these cases you can use CMake policies, as discussed in section 4.7.

4.6 Using Modules

Code reuse is a valuable technique in software development and CMake has been designed to support it. Allowing CMakeLists files to make use of reusable modules enables the entire CMake community to share reusable sections of code. For CMake these sections of code are called modules and can be found in the Modules subdirectory of your CMake installation. Modules are simply sections of CMake commands put into a file. They can then be included into other CMakeLists files using the `include` command. For example, the following commands will include the `FindTCL` module from CMake and then add the Tcl library to the target FOO.

```
include (FindTCL)

target_link_libraries (FOO ${TCL_LIBRARY})
```

A module's location can be specified using the full path to the module file, or by letting CMake find the module by itself. CMake will look for modules in the directories specified by `CMAKE_MODULE_PATH` and if it cannot find it there, it will look in the Modules subdirectory of CMake. This way projects can override modules that CMake provides, to customize them for their needs. Modules can be broken into a few main categories:

Find Modules

These modules determine the location of software elements such as header files or libraries.

System Introspection Modules

These modules test the system for properties such as the size of a float, support for ANSI C++ streams, etc.

Utility Modules

These modules provide added functionality such as support for situations where one CMake project depends on another and other convenience routines.

Now let us consider these three types of modules in more detail. CMake includes a large number of Find modules. The purpose of a Find module is to locate software elements such as header or library files. If they cannot be found then they provide a cache entry so that the user can set the required properties. Consider the following module that finds the PNG library.

```
#
# Find the native PNG includes and library
#

# This module defines
# PNG_INCLUDE_DIR, where to find png.h, etc.
# PNG_LIBRARIES, the libraries to link against to use PNG.
# PNG_DEFINITIONS - You should call
# add_definitions (${PNG_DEFINITIONS}) before compiling code
# that includes png library files.
# PNG_FOUND, If false, do not try to use PNG.

# also defined, but not for general use are
# PNG_LIBRARY, where to find the PNG library.

# None of the above will be defined unless zlib can be found.

# PNG depends on Zlib
include ( FindZLIB.cmake )

if (ZLIB_FOUND)
  find_path (PNG_PNG_INCLUDE_DIR png.h
  /usr/local/include
  /usr/include
  )

  find_library (PNG_LIBRARY png
  /usr/lib
  /usr/local/lib
  )

  if (PNG_LIBRARY)
    if (PNG_PNG_INCLUDE_DIR)
      # png.h includes zlib.h. Sigh.
      set (PNG_INCLUDE_DIR
          ${PNG_PNG_INCLUDE_DIR} ${ZLIB_INCLUDE_DIR} )
      set (PNG_LIBRARIES ${PNG_LIBRARY} ${ZLIB_LIBRARY})
      set (PNG_FOUND "YES")

      if (CYGWIN)
        if (BUILD_SHARED_LIBS)
          # No need to define PNG_USE_DLL here, because
          # it's default for Cygwin.
        else (BUILD_SHARED_LIBS)
```

```
            set (PNG_DEFINITIONS -DPNG_STATIC)
          endif (BUILD_SHARED_LIBS)
        endif (CYGWIN)

      endif ()
    endif ()

endif ()
```

The top of the module clearly documents what the module will do and what variables it will set. Next it includes another module, the FindZLIB module, that determines if the ZLib library is installed. Next, if ZLib is found, the `find_path` command is used to locate the PNG include files. The first argument is the name of the variable to store the result in, the second argument is the name of the header file to look for, the remaining arguments are paths to search for the header file. If it is not found in the system path then the variable is set to `PNG_PNG_INCLUDE_DIR-NOTFOUND`, allowing the user to set it.

Note that the paths to search for the PNG library can include hard coded directories, registry entries, and directories made up of other CMake variables. The next command finds the actual PNG library using the `find_library` command. This command performs additional checks to find a proper library name, such as adding "lib" in front of the name and ".so" at the end of the name on Linux systems.

After the find calls, some CMake variables are set that developers using FindPNG can use in their projects (such as the include paths, and library name). Finally `PNG_FOUND` is set correctly, which lets developers know that the PNG library was properly found.

This structure is fairly common to all Find modules in CMake. Usually they are fairly short, but in some cases, such as `FindOpenGL` they can be a few pages long. They are normally independent of other modules, but there is no restriction on the use of other modules.

System introspection modules provide information about the target platform or compiler. Many of these modules have names prefixed with `Test` or `Check`, such as `TestBigEndian` and `CheckTypeSize`. Many of the system introspection modules actually try to compile code in order to determine the correct result. In these cases the source code is usually named the same as the module, but with a `.c` or `.cxx` extension. System introspection modules are covered in more detail in chapter 5.

CMake includes a few Utility modules to help make using CMake a little easier. `CMakeExportBuildSettings` and `CMakeImportBuildSettings` provide tools to help verify that two C++ projects are compiled with the same compiler and key flags. The `CMakePrintSystemInformation` module prints out a number of key CMake settings to aid in debugging.

Using CMake with SWIG

One example of how modules can be used is to look at wrapping your C/C++ code in another language using SWIG. SWIG (Simplified Wrapper and Interface Generator) www.swig.org is a tool that reads annotated C/C++ header files, and creates wrapper code (glue code) in order to make the corresponding C/C++ libraries available to other programming languages such as Tcl, Python, or Java. CMake supports SWIG with the `find_package` command. Although SWIG can be used from CMake using custom commands, the SWIG package provides several macros that make building SWIG projects with CMake simpler. To use the SWIG macros, first you must call the `find_package` command with the name SWIG. Then you need to include the file referenced by the variable `SWIG_USE_FILE`. This will define several macros and set up CMake to easily build SWIG based projects.

Two very useful macros are `SWIG_ADD_MODULE` and `SWIG_LINK_LIBRARIES`. `SWIG_ADD_MODULE` works much like the `add_library` command in CMake. The command is invoked like this:

```
SWIG_ADD_MODULE (module_name language source1 source2 ... sourceN)
```

The first argument is the name of the module to create. The next argument is the target language SWIG is producing a wrapper for. The rest of the arguments consist of a list of source files used to create the shared module. The big difference is that SWIG `.i` interface files can be used directly as sources. The macro will create the correct custom commands to run SWIG, and generate the C or C++ wrapper code from the SWIG interface files. The sources can also be regular C or C++ files that need to be compiled in with the wrappers.

The `SWIG_LINK_LIBRARIES` macro is used to link support libraries to the module. This macro is used because depending on the language being wrapped by SWIG, the name of the module may be different. The actual name of the module is stored in a variable called `SWIG_MODULE_${name}_REAL_NAME` where `${name}` is the name passed into the `SWIG_ADD_MODULE` macro. For example, `SWIG_ADD_MODULE(foo tcl foo.i)` would create a variable called `SWIG_MODULE_foo_REAL_NAME` which would contain the name of the actual module created.

Now consider the following example that uses the SWIG example found in SWIG under `Examples/python/class`.

```
# Find SWIG and include the use swig file
find_package (SWIG REQUIRED)
include (${SWIG_USE_FILE})

# Find python library and add include path for python headers
find_package (PythonLibs)
include_directories (${PYTHON_INCLUDE_PATH})
```

```
# set the global swig flags to empty
set (CMAKE_SWIG_FLAGS "")

# let swig know that example.i is c++ and add the -includeall
# flag to swig
set_source_files_properties (example.i PROPERTIES CPLUSPLUS ON)
set_source_files_properties (example.i
                             PROPERTIES SWIG_FLAGS "-includeall")

# Create the swig module called example
# using the example.i source and example.cxx
# swig will be used to create wrap_example.cxx from example.i
SWIG_ADD_MODULE (example python example.i example.cxx)
SWIG_LINK_LIBRARIES (example ${PYTHON_LIBRARIES})
```

This example first uses `find_package` to locate SWIG. Next it includes the `SWIG_USE_FILE` defining the SWIG CMake macros. Then it finds the Python libraries and sets up CMake to build with the Python library. Notice that the SWIG input file example.i is used like any other source file in CMake, and properties are set on the file telling SWIG that the file is C++ and that the SWIG flag –includeall should be used when running SWIG on that source file. The module is created by telling SWIG the name of the module, the target language and the list of source files. Finally, the Python libraries are linked to the module.

Using CMake with Qt

Projects using the popular widget toolkit Qt from Nokia, qt.nokia.com, can be built with CMake. CMake supports multiple versions of Qt, including versions 3 and 4. The first step is to tell CMake what version(s) of Qt to look for. Many Qt applications are designed to work with Qt3 or Qt4, but not both. If your application is designed for Qt4 then you can use the FindQt4 module, for Qt3 you should use the FindQt3 module. If your project can work with either version of Qt then you can use the generic FindQt module. All of the modules provide helpful tools for building Qt projects. The following is a simple example of building a project that uses Qt4.

```
find_package ( Qt4 )

if (QT4_FOUND)
  include (${QT_USE_FILE})

  # what are our ui files?
  set (QTUI_SRCS qtwrapping.ui)
  QT4_WRAP_UI (QTUI_H_SRCS ${QTUI_SRCS})
  QT4_WRAP_CPP (QT_MOC_SRCS TestMoc.h)
```

```
  add_library (myqtlib ${QTUI_H_SRCS} ${QT_MOC_SRCS})
  target_link_libraries (myqtlib ${QT_LIBRARIES} )

  add_executable (qtwrapping qtwrappingmain.cxx)
  target_link_libraries (qtwrapping myqtlib)

endif (QT4_FOUND)
```

Using CMake with FLTK

CMake also supports the The Fast Light Toolkit (FLTK) with special FLTK CMake commands. The `FLTK_WRAP_UI` command is used to run the fltk fluid program on a .fl file and produce a C++ source file as part of the build. The following example shows how to use FLTK with CMake.

```
find_package (FLTK)
if (FLTK_FOUND)
  set (FLTK_SRCS
       fltk1.fl
       )
  FLTK_WRAP_UI (wraplibFLTK ${FLTK_SRCS})
  add_library (wraplibFLTK ${wraplibFLTK_UI_SRCS} )
endif (FLTK_FOUND)
```

4.7 Policies

For various reasons, sometimes a new feature or change is made to CMake that is not fully backwards compatible with older versions of CMake. This can create problems when someone tries to use an old CMakeLists file with a new version of CMake. To help both end users and developers through such issues, we have introduced policies. Policies are a mechanism in CMake to help improve backwards compatibility and track compatibility issues between different versions of CMake.

Design Goals

There were four main design goals for the CMake policy mechanism:

1. Existing projects should build with versions of CMake newer than that used by the project authors.
   - Users should not need to edit code to get the projects to build.
   - Warnings may be issued but the projects should build.

2. Correctness of new interfaces or bugs fixes in old interfaces should not be inhibited by compatibility requirements. Any reduction in correctness of the latest interface is not fair on new projects.
3. Every change made to CMake that may require changes to a project's CMakeLists files should be documented.
   - Each change should also have a unique identifier that can be referenced by warning and error messages.
   - The new behavior is enabled only when the project has somehow indicated it is supported.
4. We must be able to eventually remove code that implements compatibility with ancient CMake versions.
   - Such removal is necessary to keep the code clean and to allow for internal refactoring.
   - After such removal, attempts to build projects written for ancient versions must fail with an informative message.

All policies in CMake are assigned a name of the form `CMPNNNN` where `NNNN` is an integer value. Policies typically support both an old behavior that preserves compatibility with earlier versions of CMake, and a new behavior that is considered correct and preferred for use by new projects. Every policy has documentation detailing the motivation for the change, and the old and new behaviors

Setting Policies

Projects may configure the setting of each policy to request old or new behavior. When CMake encounters user code that may be affected by a particular policy it checks to see whether the project has set the policy. If the policy has been set (to OLD or NEW) then CMake follows the behavior specified. If the policy has not been set then the old behavior is used, but a warning is issued telling the project author to set the policy.

There are a couple ways to set the behavior of a policy. The quickest way is to set all policies to a version corresponding to the release version of CMake for which the project was written. Setting the policy version requests the new behavior for all policies introduced in the corresponding version of CMake or earlier. Policies introduced in later versions are marked as not set in order to produce proper warning messages. The policy version is set using the `cmake_policy` command's `VERSION` signature. For example, the code

```
cmake_policy (VERSION 2.6)
```

will request the new behavior for all policies introduced in CMake 2.6 or earlier. The `cmake_minimum_required` command will also set the policy version, which is convenient for use at the top of projects. A project should typically begin with the lines

```
cmake_minimum_required (VERSION 2.6)
project (MyProject)
# ...code using CMake 2.6 policies
```

Of course one should replace "2.6" with whatever version of CMake you are currently writing to. You can also set each policy individually if you wish. This is sometimes helpful for project authors who want to incrementally convert their projects to use the new behavior, or silence warnings about dependence on old behavior. The `cmake_policy` command's `SET` option may be used to explicitly request old or new behavior for a particular policy.

For example, CMake 2.6 introduced policy `CMP0002`, which requires all logical target names to be globally unique (duplicate target names previously worked in some cases by accident but were not diagnosed). Projects using duplicate target names and working accidentally will receive warnings referencing the policy. The warnings may be silenced by the code

```
cmake_policy (SET CMP0002 OLD)
```

which explicitly tells CMake to use the old behavior for the policy (silently accept duplicate target names). Another option is to use the code

```
cmake_policy (SET CMP0002 NEW)
```

to explicitly tell CMake to use new behavior and produce an error when a duplicate target is created. Once this is added to the project it will not build until the author removes any duplicate target names.

When a new version of CMake is released that introduces new policies it will still build old projects, because by default they do not request NEW behavior for any of the new policies. When starting a new project one should always specify the most recent release of CMake to be supported as the policy version level. This will make sure that the project is written to work using policies from that version of CMake and not using any old behavior. If no policy version is set CMake will warn and assume a policy version of 2.4. This allows existing projects that do not specify `cmake_minimum_required` to build as they would have with CMake 2.4.

The Policy Stack

Policy settings are scoped using a stack. A new level of the stack is pushed when entering a new subdirectory of the project (with `add_subdirectory`) and popped when leaving it. Therefore setting a policy in one directory of a project will not affect parent or sibling directories, but will affect subdirectories.

This is useful when a project contains subprojects maintained separately but built inside the tree. The top-level CMakeLists file in a project may write

```
cmake_policy (VERSION 2.6)
project (MyProject)
add_subdirectory (OtherProject)
# ... code requiring new behavior as of CMake 2.6 ...
```

while the `OtherProject/CMakeLists.txt` file contains

```
cmake_policy (VERSION 2.4)
project (OtherProject)
# ... code that builds with CMake 2.4 ...
```

This allows a project to be updated to CMake 2.6 while subprojects, modules, and included files continue to build with CMake 2.4 until their maintainers update them.

User code may use the `cmake_policy` command to push and pop its own stack levels as long as every push is paired with a pop. This is useful to temporarily request different behavior for a small section of code. For example, policy `CMP0003` removes extra link directories that used to be included when new behavior is used. While incrementally updating a project it may be difficult to build a particular target with the new behavior but all other targets are okay. The code

```
cmake_policy (PUSH)
cmake_policy (SET CMP0003 OLD) # use old-style link for now
add_executable (myexe ...)
cmake_policy (POP)
```

will silence the warning and use the old behavior for that target. You can get a list of policies and help on specific policies by running cmake from the command line as follows

```
cmake --help-command cmake_policy
cmake --help-policies
cmake --help-policy CMP0003
```

Updating a Project For a New Version of CMake

When a CMake release introduces new policies it may generate warnings for some existing projects. These warnings indicate that changes to the project may need to be made to deal correctly with the new policies. While old releases of the project can continue to build with

the warnings the project development tree should be updated to take the new policies into account. There are two approaches to updating a tree: one-shot and incremental. Which one is easier depends on the size of the project and what new policies produce warnings.

The One-Shot Approach

The simplest approach to updating a project for a new version of CMake is simply to change the policy version set at the top of the project, try building with the new CMake version, and fix problems. For example, to update a project to build with CMake 2.8 one might write

```
cmake_minimum_required (VERSION 2.8)
```

at the beginning of the top-level CMakeLists file. This tells CMake to use the new behavior for every policy introduced in CMake 2.8 and below. When building this project with CMake 2.8 no warnings will be produced about policies because it knows of no policies introduced in later versions. However, if the project was depending on the old behavior of a policy it may not build since CMake now uses the new behavior without warning. It is up to the project author who added the policy version line to fix these issues.

The Incremental Approach

Another approach to updating a project for a new version of CMake is to deal with each warning one-by-one. One advantage of this approach is that the project will continue to build throughout the process, so the changes can be made incrementally.

When CMake encounters a situation where it needs to know whether to use the old or new behavior for a policy, it checks whether the project has set the policy. If the policy is set CMake silently uses the corresponding behavior. If the policy is not set, CMake uses the old behavior but warns that the policy is not set.

In many cases the warning message will point at the exact line of code in the CMakeLists files that caused the warning. In some cases the situation cannot be diagnosed until CMake is generating the native build system rules for the project, so the warning will not include explicit context information. In these cases CMake will try to provide some information about where code may need to be changed. The documentation for these "generation-time" policies should indicate the point in the project code at which the policy should be set to take effect.

In order to incrementally update a project one warning should be addressed at a time. Several cases may occur as described below.

Silence a Warning When the Code is Correct

Many policy warnings may be produced simply because the project has not set the policy even though the project may work correctly with the new behavior (there is no way for CMake to know the difference). For a warning about some policy `CMP<NNNN>` one may check whether this is the case by adding

```
cmake_policy (SET CMP<NNNN> NEW)
```

to the top of the project and trying to build it. If the project builds correctly with the new behavior one may move on to the next policy warning. If the project does not build correctly one of the other cases may apply.

Silence a Warning Without Updating the Code

One may suppress all instances of a warning `CMP<NNNN>` by adding

```
cmake_policy (SET CMP<NNNN> OLD)
```

at the top of a project. However, we encourage project authors to update their code to work with the new behavior for all policies. This is especially important because versions of CMake in the (distant) future may remove support for the old behavior and produce an error for projects requesting it (which tells the user to get an older CMake to build the project).

Silence a Warning by Updating Code

When a project does not work correctly with the NEW behavior for a policy its code needs to be updated. In order to deal with a warning for some policy `CMP<NNNN>` one may add

```
cmake_policy (SET CMP<NNNN> NEW)
```

at the top of the project and then fix the code to work with the NEW behavior.

If many instances of the warning occur fixing all of them simultaneously may be too difficult. Instead a developer may fix one at a time. This may be done using the PUSH/POP signatures of the `cmake_policy` command:

```
cmake_policy (PUSH)
cmake_policy (SET CMP<NNNN> NEW)
# ... code updated for new policy behavior ...
cmake_policy (POP)
```

This will request the new behavior for a small region of code that has been fixed. Other instances of the policy warning may still appear and must be fixed separately.

Updating the Project Policy Version

After addressing all policy warnings and getting the project to build cleanly with the new CMake version one step remains. The policy version set at the top of the project should now be updated to match the new CMake version, just as in the one-shot approach above. For

example, after updating a project to build cleanly with CMake 2.8 one may update the top of the project with the line

```
cmake_minimum_required(VERSION 2.8)
```

This will set all policies introduced in CMake 2.8 or below to use the new behavior. Then one may sweep through the rest of the code and remove all the calls to the `cmake_policy` command used to request the new behavior incrementally. The end result should look the same as the one-shot approach above but could be attained step-by-step.

Supporting Multiple CMake Versions

Some projects might want to support a few releases of CMake simultaneously. The goal is to build with an older version but also work with newer versions without warnings. In order to support both CMake 2.4 and 2.6, one may write code like

```
cmake_minimum_required (VERSION 2.4)
if (COMMAND cmake_policy)
  # policy settings ...
  cmake_policy (SET CMP0003 NEW)
endif (COMMAND cmake_policy)
```

This will set the policies when building with CMake 2.6 and just ignore them for CMake 2.4. In order to support both CMake 2.6 and some policies of CMake 2.8, one may write code like

```
cmake_minimum_required (VERSION 2.6)
if (POLICY CMP1234)
  # policies not known to CMake 2.6 ...
  cmake_policy (SET CMP1234 NEW)
endif (POLICY CMP1234)
```

This will set the policies when building with CMake 2.8 and just ignore them for CMake 2.6. If it is known that the project builds with both CMake 2.6 and CMake 2.8's new policies one may write

```
cmake_minimum_required (VERSION 2.6)
if (NOT ${CMAKE_VERSION} VERSION_LESS 2.8)
   cmake_policy (VERSION 2.8)
endif ()
```

4.8 Linking Libraries

In CMake 2.6 and later a new approach to generating link lines for targets has been implemented. Consider these libraries:

```
/path/to/libfoo.a
/path/to/libfoo.so
```

Previously if someone wrote

```
target_link_libraries (myexe /path/to/libfoo.a)
```

CMake would generate this code to link it:

```
  ... -L/path/to -Wl,-Bstatic -lfoo -Wl,-Bdynamic ...
```

This worked most of the time, but some platforms (such as Mac OS X) do not support the `-Bstatic` or equivalent flag. This made it impossible to link to the static version of a library without creating a symlink in another directory and using that one instead. Now CMake will generate this code:

```
  ... /path/to/libfoo.a ...
```

This guarantees that the correct library is chosen. However there are some caveats to keep in mind. In the past a project could write this (incorrect) code, and it would work by accident:

```
add_executable (myexe myexe.c)
target_link_libraries (myexe /path/to/libA.so B)
```

where "`B`" is meant to link "`/path/to/libB.so`". This code is incorrect because it asks CMake to link to `B` but does not provide the proper linker search path for it. It used to work by accident because the `-L/path/to` would get added as part of the implementation of linking to A. The correct code would be either

```
link_directories (/path/to)
add_executable (myexe myexe.c)
target_link_libraries (myexe /path/to/libA.so B)
```

or even better

```
add_executable (myexe myexe.c)
target_link_libraries (myexe /path/to/libA.so /path/to/libB.so)
```

Linking to System Libraries

System libraries on UNIX-like systems are typically provided in `/usr/lib` or `/lib`. These directories are considered implicit linker search paths because linkers automatically search these locations, even without a flag like `-L/usr/lib`. Consider the code

```
find_library (M_LIB m)
target_link_libraries (myexe ${M_LIB})
```

Typically the `find_library` command would find the math library `/usr/lib/libm.so`, but some platforms provide multiple versions of libraries correesponding to different architectures. For example, on an IRIX machine one might find the libraries

```
/usr/lib/libm.so          (ELF o32)
/usr/lib32/libm.so        (ELF n32)
/usr/lib64/libm.so        (ELF 64)
```

On a Solaris machine one might find

```
/usr/lib/libm.so           (sparcv8 architecture)
/usr/lib/sparcv9/libm.so   (sparcv9 architecture)
```

Unfortunately, `find_library` may not know about all of the architecture-specific system search paths used by the linker. In fact, when it finds `/usr/lib/libm.so`, it may be finding a library with the incorrect architecture. If the link computation were to produce the line

```
... /usr/lib/libm.so ...
```

the linker might complain if `/usr/lib/libm.so` does not match the architecture it wants. One solution to this problem is for the link computation to recognize that the library is in a system directory and ask the linker to search for the library. It could produce the link line

```
... -lm ...
```

and the linker would search through its architecture-specific implicit link directories to find the correct library. Unfortunately, this solution suffers from the original problem of distinguishing between static and shared versions. In order to ask the linker to find a static system library with the correct architecture it must produce the link line

```
... -Wl,-Bstatic -lm ... -Wl,-Bshared ...
```

Since not all platforms support such flags CMake compromises. Libraries that are not in implicit system locations are linked by passing the full library path to the linker. Libraries that are in implicit system locations (such as `/usr/lib`) are linked by passing the `-l` option if a flag like `-Bstatic` is available, and by passing the full library path to the linker otherwise.

Specifying Optimized or Debug Libraries with a Target

On Windows platforms it is often required to link debug libraries with debug libraries, and optimized libraries with optimized libraries. CMake helps satisfy this requirement with the `target_link_libraries` command, which accepts an optional flag that is debug or optimized. So, if a library is preceded with either debug or optimized, then that library will only be linked in with the like configuration type. For example:

```
add_executable (foo foo.c)
target_link_libraries (foo debug libdebug optimized libopt)
```

In this case foo will be linked against libdebug if a debug build was selected, or against libopt if an optimized build was selected.

4.9 Shared Libraries and Loadable Modules

Shared libraries and loadable modules are very powerful tools for software developers. They can be used to create extension modules or plugins for off-the-shelf software, and can be used to decrease the compile/link/run cycles for C and C++ programs. However, despite years of use, the cross platform creation of shared libraries and modules remains a black art understood by only a few developers. CMake has the ability to aid developers in the creation of shared libraries and modules. CMake knows the correct tools and flags to use in order to produce the shared libraries for most modern operating systems that support them. Unfortunately, CMake cannot do all the work, and developers must sometimes alter source code and understand the basic concepts and common pitfalls associated with shared libraries before they can be used effectively. This section will describe many of the issues required to take advantage of shared libraries and loadable modules.

A shared library should be thought of more like an executable than a static library, and on most systems actually requires executable permissions to be set on the shared library file. This

means that shared libraries can link to other shared libraries when they are created in the same way as an executable. Unlike a static library where the atomic unit is the object file, for shared libraries, the entire library is the atomic unit. This can cause some unexpected linker errors when converting from static to shared libraries. If an object file is part of a static library, but the executable linking to the library does not use any of the symbols in that object file, then the file is simply excluded from the final linked executable. With shared libraries, all the object files that make up the library and all of the dependencies that they require come as one unit. For example, suppose you had a library with an object file defining the function `DisplayOnXWindow()` which required the X11 library. If you linked an executable to that library, but did not call the `DisplayOnXWindow()` function, the static library version would not require X11, but the shared library version would require the X11 library. This is because a shared library has to be taken as one unit, and a static library is only an archive of object files from which linkers can choose which objects are needed. This means that static linked executables can be smaller, as they only contain the object code actually used.

Another difference between shared and static libraries is library order. With static libraries the order on the link line can make a difference. This is because most linkers only use the symbols that are needed in a single pass over all the given libraries. So, the library order should go from the library that uses the most other libraries to the library that uses no other libraries. CMake will preserve and remember the order of libraries and library dependencies of a project. This means that each library in a project should use the `target_link_libraries` command to specify all of the libraries that it directly depends on. The libraries will be linked with each other for shared builds, but not static builds. However, the link information is used in static builds when executables are linked. An executable that only links library libA will get libA plus libB and libC as long as libA's dependency on libB and libC was properly specified using `target_link_libraries (libA libB libC)`.

At this point, one might wonder why shared libraries would be preferred over static libraries. There are several reasons. First, shared libraries can decrease the compile/link/run cycle time. This is because the linker does not have to do as much work when linking to shared libraries because there are fewer decisions to be made about which object files to keep. Also, often times, the executable does not even need to be re-linked after the shared library is rebuilt. So, developers can work on a library compiling and linking only the small part of the program that is currently being developed, and then re-run the executable after each build of the shared library. Also, if a library is used by many different executables on a system, then there only needs to be one copy of the library on disk, and often in memory too.

In addition to the concept of a software library, shared libraries can also be used on many systems as run time loadable modules. This means that a program can at run time, load and execute object code that was not part of the original software. This allows developers to create software that is both open and closed. (For more information see Object Oriented Software Construction by Bertrand Meyer.) Closed software is software that cannot be modified. It has been through a testing cycle and can be certified to perform specific tasks with regression

tests. However, a seemingly opposite goal is sought after by developers of object oriented software. This is the concept of Open software that can be extended by future developers. This can be done via inheritance and polymorphism with object systems. Shared libraries that can be loaded at run time, allow for these seemingly opposing goals to be achieved in the same software package. Many common applications support the idea of plugins. The most common of these applications is the web browser. Internet Explorer uses plugins to support video over the web and 3D visualization. In addition to plugins, loadable factories can be used to replace C++ objects at run time, as is done in VTK.

Once it is decided that shared libraries or loadable modules are the right choice for a particular project, there are a few issues that developers need to be aware of. The first question that must be answered is which symbols are exported by the shared library? This may sound like a simple question, but the answer is different from platform to platform. On many, but not all UNIX systems, the default behavior is to export all the symbols much like a static library. However, on Windows systems, developers must explicitly tell the linker and compiler which symbols are to be exported and imported from shared libraries. This is often a big problem for UNIX developers moving to Windows. There are two ways to tell the compiler/linker which symbols to export/import on Windows. The most common approach is to decorate the code with a Microsoft™ C/C++ language extension. An alternative is to create an extra file called a .def file. This file is a simple ASCII file containing the names of all the symbols to be exported from a library.

The Microsoft™ extension uses the `__declspec` directive. If a symbol has `__declspec( dllexport )` in front of it, it will be exported, and if it has `__declspec( dllimport )` it will be imported. Since the same file may be shared during the creation and use of a library, it must be both exported and imported in the same source file. This can only be done with the preprocessor. The developer can create a macro called `LIBRARY_EXPORT` that is defined to `dllexport` when building the library and `dllimport` when using the library. CMake helps this process by automatically defining `${LIBNAME}_EXPORTS` when building a DLL (dynamic link library, a.k.a. a shared library) on Windows.

The following code snippet is from the VTK library vtkCommon, and is included by all files in the vtkCommon library:

```
#if defined(WIN32)

 #if defined(vtkCommon_EXPORTS)
  #define VTK_COMMON_EXPORT __declspec( dllexport )
 #else
  #define VTK_COMMON_EXPORT __declspec( dllimport )
 #endif
#else
  #define VTK_COMMON_EXPORT
#endif
```

The example checks for Windows and checks the `vtkCommon_EXPORTS` macro provided by CMake. So, on UNIX `VTK_COMMON_EXPORT` is defined to nothing, and on Windows during the building of vtkCommon.dll it is defined as `__declspec(dllexport)`, and when the file is being used by another file, it is defined to `__declspec(dllimport)`.

The second approach requires a .def file to specify the symbols to be exported. This file could be created by hand, but for a large and changing C++ library that could be time consuming and error prone. CMake's custom commands can be used to run a pre-link program that will create a .def file from the compiled object files automatically. In the following example, a Perl script called `makedef.pl` is used, the script runs the `DUMPBIN` program on the .obj files and extracts all of the exportable symbols and writes a .def file with the correct exports for all the symbols in the library mylib.

```
----CMakeLists.txt-----

cmake_minimum_required (VERSION 2.6)
project (myexe)

set (SOURCES mylib.cxx mylib2.cxx)

# create a list of all the object files
string (REGEX REPLACE "\\.cxx" ".obj" OBJECTS "${SOURCES}")

# create a shared library with the .def file
add_library (mylib SHARED ${SOURCES}
  ${CMAKE_CURRENT_BINARY_DIR}/mylib.def
  )
# set the .def file as generated
set_source_files_properties (
  ${CMAKE_CURRENT_BINARY_DIR}/mylib.def
  PROPERTIES GENERATED 1
  )

# create an executable
add_executable (myexe myexe.cxx)

# link the executable to the dll
target_link_libraries(myexe mylib)

#convert to windows slashes
set (OUTDIR
  ${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_CFG_INTDIR}
  )
```

```
string (REGEX REPLACE "/" "\\\\" OUTDIR ${OUTDIR})

# create a custom pre link command that runs
# a perl script to create a .def file using dumpbin
add_custom_command (
  TARGET mylib PRE_LINK
  COMMAND perl
  ARGS ${CMAKE_CURRENT_SOURCE_DIR}/makedef.pl
  ${CMAKE_CURRENT_BINARY_DIR}\\mylib.def mylib
  ${OUTDIR} ${OBJECTS}
  COMMENT "Create .def file"
  )
```

```
---myexe.cxx----
#include <iostream>
#include "mylib.h"
int main()
{
  std::cout << myTen() << "\n";
  std::cout << myEight() << "\n";
}
```

```
---mylib.cxx--
int myTen()
{
  return 10;
}
```

```
--mylib2.cxx---
int myEight()
{
  return 8;
}
```

There is a significant difference between Windows and most UNIX systems with respect to the requirements of symbols. DLLs on Windows are required to be fully resolved, this means that they must link every symbol at creation. UNIX systems allow shared libraries to get symbols from the executable or other shared libraries at run time. On UNIX systems that support this feature, CMake will compile with the flags that allow executable symbols to be used by shared libraries. This small difference can cause large problems. A common, but hard to track down bug with DLLs happens with C++ template classes and static members. Two DLLs can end up with separate copies of what is supposed to be a single global static member of a class. There are also problems with the approach taken on most UNIX systems. The start up time for large applications with many symbols can be long since much of the linking is deferred to run time.

Another common pitfall occurs with C++ global objects. These objects require that constructors must be called before they can be used. The `main` that links or loads C++ shared libraries MUST be linked with the C++ compiler, or globals like `cout` may not be initialized before they are used, causing strange crashes at start up time.

Since executables that link to shared libraries must be able to find the libraries at run time, special environment variables and linker flags must be used. There are tools that can be used to show which libraries an executable is actually using. On many UNIX systems there is a tool called `ldd` (`otool -L` on Mac OS X) that shows which libraries are used by an executable. On Windows, a program called `depends` can be used to find the same type of information. On many UNIX systems there are also environment variables like `LD_LIBRARY_PATH` that tell the program where to find the libraries at run time. Where supported CMake will add run time library path information into the linked executables, so that `LD_LIBRARY_PATH` is not required. This feature can be turned off by setting the cache entry `CMAKE_SKIP_RPATH` to false. This may be desirable for installed software that should not be looking in the build tree for shared libraries. On Windows there is only one `PATH` environment variable that is used for both DLLs and finding executables.

4.10 Shared Library Versioning

When an executable is linked to a shared library, it is important that the copy of the shared library loaded at runtime matches that expected by the executable. On some UNIX systems, a shared library has an associated "soname" intended to solve this problem. When an executable links against the library, its soname is copied into the executable. At runtime, the dynamic linker uses this name from the executable to search for the library.

Consider a hypothetical shared library "foo" providing a few C functions that implement some functionality. The interface to foo is called an Application Programming Interface (API). If the implementation of these C functions changes in a new version of foo, but the API remains the same, then executables linked against foo will still run correctly. When the API changes,

old executables will no longer run with a new copy of foo, so a new API version number must be associated with foo.

This can be implemented by creating the original version of foo with a soname and file name such as libfoo.so.1. A symbolic link such as libfoo.so -> libfoo.so.1 will allow standard linkers to work with the library and create executables. The new version of foo can be called libfoo.so.2 and the symbolic link updated so that new executables use the new library. When an old executable runs, the dynamic linker will look for libfoo.so.1, find the old copy of the library, and run correctly. When a new executable runs, the dynamic linker will look for libfoo.so.2 and correctly load the new version.

This scheme can be expanded to handle the case of changes to foo that do not modify the API. We introduce a second set of version numbers that is totally independent of the first. This new set corresponds to the software version providing foo. For example, some larger project may have introduced the existence of library foo starting in version 3.4. In this case, the file name for foo might be libfoo.so.3.4, but the soname would still be libfoo.so.1 because the API for foo is still on its first version. A symbolic link from libfoo.so.1 -> libfoo.so.3.4 will allow executables linked against the library to run. When a bug is fixed in the software without changing the API to foo, then the new library file name might be libfoo.so.3.5, and the symbolic link can be updated to allow existing executables to run.

CMake supports this soname-based version number encoding on platforms supporting soname natively. A target property for the shared library named "VERSION" specifies the version number used to create the file name for the library. This version should correspond to that of the software package providing foo. On Windows the VERSION property is used to set the binary image number, using major.minor format. Another target property named "SOVERSION" specifies the version number used to create the soname for the library. This version should correspond to the API version number for foo. These target properties are ignored on platforms where CMake does not support this scheme.

The following CMake code configures the version numbers of the shared library foo:

```
set_target_properties (foo PROPERTIES VERSION 1.2 SOVERSION 4)
```

This results in the following library and symbolic links:

```
libfoo.so.1.2
libfoo.so.4 -> libfoo.so.1.2
libfoo.so -> libfoo.so.4
```

If only one of the two properties is specified, the other defaults to its value automatically. For example, the code

```
set_target_properties (foo PROPERTIES VERSION 1.2)
```

results in the following shared library and symbolic link:

```
libfoo.so.1.2
libfoo.so -> libfoo.so.1.2
```

CMake makes no attempt to enforce sensible version numbers. It is up to the programmer to utilize this feature in a productive manner.

4.11 Installing Files

Software is typically installed into a directory separate from the source and build trees. This allows it to be distributed in a clean form and isolates users from the details of the build process. CMake provides the `install` command to specify how a project is to be installed. This command is invoked by a project in the CMakeLists file and tells CMake how to generate installation scripts. The scripts are executed at install time to perform the actual installation of files. For Makefile generators (UNIX, NMake, Borland, MinGW, etc.), the user simply runs "`make install`" (or "`nmake install`") and the make tool will invoke CMake's installation module. With GUI based systems (Visual Studio, Xcode, etc.) the user simply builds the target called `INSTALL`.

Each call to the `install` command defines some installation rules. Within one CMakeLists file (source directory) these rules will be evaluated in the order in which the corresponding commands are invoked. The order across multiple directories is not specified.

The `install` command has several signatures designed for common installation use cases. A particular invocation of the command specifies the signature as the first argument. The signatures are `TARGETS`, `FILES`, `PROGRAMS`, `DIRECTORY`, `SCRIPT`, and `CODE`.

install (TARGETS ...)

Install the binary files corresponding to targets built inside the project.

install (FILES ...)

General-purpose file installation. It is typically used for installation of header files, documentation, and data files required by your software.

install (PROGRAMS ...)

Installs executable files not built by the project, such as shell scripts. It is identical to `install (FILES)` except that the default permissions of the installed file include the executable bit.

install (DIRECTORY ...)

Install an entire directory tree. This may be used for installing directories with resources such as icons and images.

install (SCRIPT ...)

Specify a user-provided CMake script file to be executed during installation. Typically this is used to define pre-install or post-install actions for other rules.

install (CODE ...)

Specify user-provided CMake code to be executed during the installation. This is similar to `install (SCRIPT)` but the code is provided inline in the call as a string.

The `TARGETS`, `FILES`, `PROGRAMS`, `DIRECTORY` signatures are all meant to create install rules for files. The targets, files, or directories to be installed are listed immediately after the signature name argument. Additional details can be specified using keyword arguments followed by corresponding values. Keyword arguments provided by most of the signatures are as follows.

DESTINATION

Specifies the location in which the installation rule will place files. This argument must be followed by a directory path indicating the location. If the directory is specified as a full path it will be evaluated at install time as an absolute path. If the directory is specified as a relative path it will be evaluated at install time relative to the installation prefix. The prefix may be set by the user through the cache variable `CMAKE_INSTALL_PREFIX`. A platform-specific default is provided by CMake: "`/usr/local`" on UNIX and "<SystemDrive>`/Program Files/`<ProjectName>" on Windows, where SystemDrive is something like "`C:`" and ProjectName is the name given to the top-most `PROJECT` command.

PERMISSIONS

Specifies file permissions to be set on the installed files. This option is needed only to override the default permissions selected by a particular `INSTALL` command signature. Valid permissions are `OWNER_READ`, `OWNER_WRITE`, `OWNER_EXECUTE`, `GROUP_READ`, `GROUP_WRITE`, `GROUP_EXECUTE`, `WORLD_READ`, `WORLD_WRITE`, `WORLD_EXECUTE`, `SETUID`, and `SETGID`. Some platforms do not support all of these permissions, on such platforms those permission names are ignored.

CONFIGURATIONS

Specifies a list of build configurations for which an installation rule applies (Debug, Release, etc.). For Makefile generators the build configuration is specified by the `CMAKE_BUILD_TYPE` cache variable. For Visual Studio and Xcode generators the configuration is selected when the `INSTALL` target is built. An installation rule will

be evaluated only if the current install configuration matches an entry in the list provided to this argument. Configuration name comparison is case-insensitive.

COMPONENT

Specifies the installation component for which the installation rule applies. Some projects divide their installations into multiple components for separate packaging. For example, a project may define a "Runtime" component that contains the files needed to run a tool, a "Development" component containing the files needed to build extensions to the tool, and a "Documentation" component containing the manual pages and other help files. The project may then package each component separately for distribution by installing only one component at a time. By default all components are installed. Component-specific installation is an advanced feature intended for use by package maintainers. It requires manual invocation of the installation scripts with an argument defining the COMPONENT variable to name the desired component. Note that component names are not defined by CMake. Each project may define its own set of components.

OPTIONAL

Specifies that it is not an error if the input file to be installed does not exist. If the input file exists it will be installed as requested. If it does not exist it will be silently not installed.

Projects typically install some of the library and executable files created during their build process. The install command provides the TARGETS signature for this purpose:

```
install (TARGETS targets...
        [[ARCHIVE|LIBRARY|RUNTIME|FRAMEWORK|BUNDLE|
          PRIVATE_HEADER|PUBLIC_HEADER|RESOURCE]
         [DESTINATION <dir>]
         [PERMISSIONS permissions...]
         [CONFIGURATIONS [Debug|Release|...]]
         [COMPONENT <component>]
         [OPTIONAL]
         [EXPORT <export name>]
         [NAMELINK_ONLY|NAMELINK_SKIP]
        ] [...])
```

The TARGETS keyword is immediately followed by a list of the targets created using add_executable or add_library to be installed. One or more files corresponding to each target will be installed.

Files installed with this signature may be divided into three categories: ARCHIVE, LIBRARY, and RUNTIME. These categories are designed to group target files by typical installation

destination. The corresponding keyword arguments are optional, but if present specify that other arguments following them apply only to target files of that type. Target files are categorized as follows:

executables - `RUNTIME`

Created by `add_executable` (.exe on Windows, no extension on UNIX)

loadable modules - `LIBRARY`

Created by `add_library` with the `MODULE` option (.dll on Windows, .so on UNIX)

shared libraries - `LIBRARY`

Created by `add_library` with the `SHARED` option on UNIX-like platforms (.so on most UNIX, .dylib on Mac)

dynamic-link libraries - `RUNTIME`

Created by `add_library` with the `SHARED` option on Windows platforms (.dll)

import libraries - `ARCHIVE`

Linkable file created by a dynamic-link library that exports symbols (.lib on most Windows, .dll.a on Cygwin and MinGW).

static libraries - `ARCHIVE`

Created by `add_library` with the `STATIC` option (.lib on Windows, .a on UNIX, Cygwin, and MinGW)

Consider a project that defines an executable `myExecutable` that links to a shared library `mySharedLib`. It also provides a static library `myStaticLib` and a plugin module to the executable called `myPlugin` that also links to the shared library. The executable, static library, and plugin file may be installed individually using the commands

```
install (TARGETS myExecutable DESTINATION bin)
install (TARGETS myStaticLib DESTINATION lib/myproject)
install (TARGETS myPlugin DESTINATION lib)
```

The executable will not be able to run from the installed location until the shared library to which it links is also installed. Installation of the library requires a bit more care in order to support all platforms. It must be installed to a location searched by the dynamic linker on each platform. On UNIX-like platforms the library is typically installed to `lib`,while on Windows it should be placed next to the executable in `bin`. An additional challenge is that the import library associated with the shared library on Windows should be treated like the static library and installed to `lib/myproject`. In other words we have three different kinds of files created with a single target name that must be installed to three different destinations!

Fortunately this problem can be solved using the category keyword arguments. The shared library may be installed using the command

```
install (TARGETS mySharedLib
         RUNTIME DESTINATION bin
         LIBRARY DESTINATION lib
         ARCHIVE DESTINATION lib/myproject)
```

This tells CMake that the RUNTIME file (.dll) should be installed to bin, the LIBRARY file (.so) should be installed to lib, and the ARCHIVE (.lib) file should be installed to lib/myproject. On UNIX the LIBRARY file will be installed and on Windows the RUNTIME and ARCHIVE files will be installed.

If the above sample project is to be packaged into separate runtime and development components we must assign the appropriate component to each target file installed. The executable, shared library, and plugin are required in order to run the application, so they belong in a Runtime component. Meanwhile the import library (corresponding to the shared library on Windows) and the static library are only required to develop extensions to the application, and therefore belong in a Development component.

Component assignments may be specified by adding the COMPONENT argument to each of the commands above. We may also combine all of the installation rules into a single command invocation. This single command is equivalent to all of the above commands with components added. The files generated by each target are installed using the rule for their category.

```
install (TARGETS myExecutable mySharedLib myStaticLib myPlugin
         RUNTIME DESTINATION bin            COMPONENT Runtime
         LIBRARY DESTINATION lib            COMPONENT Runtime
         ARCHIVE DESTINATION lib/myproject COMPONENT Development)
```

Either NAMELINK_ONLY or NAMELINK_SKIP may be specified as a LIBRARY option. On some platforms a versioned shared library has a symbolic link such as

```
lib<name>.so -> lib<name>.so.1
```

where lib<name>.so.1 is the soname of the library and lib<name>.so is a "namelink" that helps linkers to find the library when given -l<name>. The NAMELINK_ONLY option causes installation of only the namelink when a library target is installed. The NAMELINK_SKIP option causes installation of library files other than the namelink when a library target is installed. When neither option is given both portions are installed. On

platforms where versioned shared libraries do not have namelinks, or when a library is not versioned, the NAMELINK_SKIP option installs the library and the NAMELINK_ONLY option installs nothing. See the VERSION and SOVERSION target properties for details on creating versioned shared libraries.

Projects may install files other than those that are created with add_executable or add_library, such as header files or documentation. General-purpose installation of files is specified using the FILES signature:

```
install (FILES files... DESTINATION <dir>
         [PERMISSIONS permissions...]
         [CONFIGURATIONS [Debug|Release|...]]
         [COMPONENT <component>]
         [RENAME <name>] [OPTIONAL])
```

The FILES keyword is immediately followed by a list of files to be installed. Relative paths are evaluated with respect to the current source directory. Files will be installed to the given DESTINATION directory. For example, the command

```
install (FILES my-api.h ${CMAKE_CURRENT_BINARY_DIR}/my-config.h
         DESTINATION include)
```

Installs the file my-api.h from the source tree and the file my-config.h from the build tree into the include directory under the installation prefix. By default installed files are given permissions OWNER_WRITE, OWNER_READ, GROUP_READ, and WORLD_READ, but this may be overridden by specifying the PERMISSIONS option. Consider the case in which we want to install a global configuration file on a UNIX system that is readable only by its owner (such as root). We may accomplish this with the command

```
install (FILES my-rc DESTINATION /etc
         PERMISSIONS OWNER_WRITE OWNER_READ)
```

which installs the file my-rc with owner read/write permission into the absolute path /etc.

The RENAME argument specifies a name for an installed file that may be different from the original file. Renaming is allowed only when a single file is installed by the command. For example, the command

```
install(FILES version.h DESTINATION include RENAME my-version.h)
```

will install the file `version.h` from the source directory to `include/my-version.h` under the installation prefix.

Projects may also install helper programs such as shell scripts or python scripts that are not actually compiled as targets. These may be installed with the `FILES` signature using the `PERMISSIONS` option to add execute permission. However this case is common enough to justify a simpler interface. CMake provides the `PROGRAMS` signature for this purpose:

```
install (PROGRAMS files... DESTINATION <dir>
         [PERMISSIONS permissions...]
         [CONFIGURATIONS [Debug|Release|...]]
         [COMPONENT <component>]
         [RENAME <name>] [OPTIONAL])
```

The `PROGRAMS` keyword is immediately followed by a list of scripts to be installed. This command is identical to the `FILES` signature except that the default permissions additionally include `OWNER_EXECUTE`, `GROUP_EXECUTE`, and `WORLD_EXECUTE`. For example, we may install a python utility script with the command

```
install (PROGRAMS my-util.py DESTINATION bin)
```

which installs `my-util.py` to the `bin` directory under the installation prefix and gives it owner, group, and world read and execute permission plus owner write.

Projects may also provide a whole directory full of resource files such as icons or html documentation. An entire directory may be installed using the `DIRECTORY` signature:

```
install (DIRECTORY dirs... DESTINATION <dir>
         [FILE_PERMISSIONS permissions...]
         [DIRECTORY_PERMISSIONS permissions...]
         [USE_SOURCE_PERMISSIONS]
         [CONFIGURATIONS [Debug|Release|...]]
         [COMPONENT <component>]
         [[PATTERN <pattern> | REGEX <regex>]
         [EXCLUDE] [PERMISSIONS permissions...]] [...])
```

The `DIRECTORY` keyword is immediately followed by a list of directories to be installed. Relative paths are evaluated with respect to the current source directory. Each named directory is installed to the destination directory. The last component of each input directory

name is appended to the destination directory as that directory is copied. For example, the command

```
install (DIRECTORY data/icons DESTINATION share/myproject)
```

will install the `data/icons` directory from the source tree into `share/myproject/icons` under the installation prefix. A trailing slash will leave the last component empty and install the contents of the input directory to the destination. The command

```
install (DIRECTORY doc/html/ DESTINATION doc/myproject)
```

installs the contents of `doc/html` from the source directory into `doc/myproject` under the installation prefix. If no input directory names are given, as in

```
install (DIRECTORY DESTINATION share/myproject/user)
```

the destination directory will be created but nothing will be installed into it.

Files installed by the `DIRECTORY` signature are given the same default permissions as the `FILES` signature. Directories installed by the `DIRECTORY` signature are given the same default permissions as the `PROGRAMS` signature. The `FILE_PERMISSIONS` and `DIRECTORY_PERMISSIONS` options may be used to override these defaults. Consider the case in which a directory full of example shell scripts is to be installed into a directory that is both owner and group writable. We may use the command

```
install (DIRECTORY data/scripts DESTINATION share/myproject
         FILE_PERMISSIONS
           OWNER_READ OWNER_EXECUTE OWNER_WRITE
           GROUP_READ GROUP_EXECUTE
           WORLD_READ WORLD_EXECUTE
         DIRECTORY_PERMISSIONS
           OWNER_READ OWNER_EXECUTE OWNER_WRITE
           GROUP_READ GROUP_EXECUTE GROUP_WRITE
           WORLD_READ WORLD_EXECUTE)
```

which installs the directory `data/scripts` into `share/myproject/scripts` and sets the desired permissions. In some cases a fully prepared input directory created by the project may have the desired permissions already set. The `USE_SOURCE_PERMISSIONS` option tells CMake to use the file and directory permissions from the input directory during installation. If

in the previous example the input directory were to have already been prepared with correct permissions the following command may have been used instead.

```
install (DIRECTORY data/scripts DESTINATION share/myproject
        USE_SOURCE_PERMISSIONS)
```

If the input directory to be installed is under source management, such as CVS, there may be extra subdirectories in the input that we do not wish to install. There may also be specific files which should not be installed, or be installed with different permissions, while most files get the defaults. The `PATTERN` and `REGEX` options may be used for this purpose. A `PATTERN` option is followed first by a globbing pattern and then by an `EXCLUDE` or `PERMISSIONS` option. A `REGEX` option is followed first by a regular expression and then by `EXCLUDE` or `PERMISSIONS`. The `EXCLUDE` option skips installation of those files or directories matching the preceding pattern or expression, while the `PERMISSIONS` option assigns specific permissions to them.

Each input file and directory is tested against the pattern or regular expression as a full path with forward slashes. A pattern will match only complete file or directory names occurring at the end of the full path while a regular expression may match any portion. For example, the pattern "`foo*`" will match "`.../foo.txt`" but not "`.../myfoo.txt`" or "`.../foo/bar.txt`" but the regular expression "`foo`" will match all of them.

Returning to the above example of installing an icons directory, consider the case in which the input directory is managed by CVS and also contains some extra text files that we do not want to install. The command

```
install (DIRECTORY data/icons DESTINATION share/myproject
        PATTERN "CVS" EXCLUDE
        PATTERN "*.txt" EXCLUDE)
```

installs the icons directory while ignoring any CVS directory or text file contained. The equivalent command using the `REGEX` option is

```
install (DIRECTORY data/icons DESTINATION share/myproject
        REGEX "/CVS$" EXCLUDE
        REGEX "/[^/]*.txt$" EXCLUDE)
```

which uses '`/`' and '`$`' to constrain the match in the same way as the patterns. Consider a similar case in which the input directory contains shell scripts and text files that we wish to install with different permissions than the other files. The command

```
install (DIRECTORY data/other/ DESTINATION share/myproject
        PATTERN "CVS" EXCLUDE
        PATTERN "*.txt"
          PERMISSIONS OWNER_READ OWNER_WRITE
        PATTERN "*.sh"
          PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE)
```

will install the contents of `data/other` from the source directory to `share/myproject` while ignoring CVS directories and giving specific permissions to `.txt` and `.sh` files.

Project installations may need to perform tasks other than just placing files in the installation tree. Third-party packages may provide their own mechanisms to register new plugins which must be invoked during project installation. The `SCRIPT` signature is provided for this purpose:

```
install (SCRIPT <file>)
```

The `SCRIPT` keyword is immediately followed by the name of a CMake script. CMake will execute the script during installation. If the file name given is a relative path it will be evaluated with respect to the current source directory. A simple use case is printing a message during installation. We first write a `message.cmake` file containing the code

```
message ("Installing My Project")
```

and then reference this script using the command

```
install (SCRIPT message.cmake)
```

Custom installation scripts are not executed during the main CMakeLists file processing. They are executed during the installation process itself. Variables and macros defined in the code containing the `install (SCRIPT)` call will not be accessible from the script. However there are a few variables defined during the script execution which may be used to get information about the installation. The variable `CMAKE_INSTALL_PREFIX` is set to the actual installation prefix. This may be different from the corresponding cache variable value because the installation scripts may be executed by a packaging tool that uses a different prefix. An environment variable `ENV{DESTDIR}` may be set by the user or packaging tool. Its value is prepended to the installation prefix and to absolute installation paths to determine the location to which files are installed. In order to reference an install location on disk the custom script may use `$ENV{DESTDIR}${CMAKE_INSTALL_PREFIX}` as the top portion of the path. The variable `CMAKE_INSTALL_CONFIG_NAME` is set to the name of the build configuration

currently being installed (Debug, Release, etc.). During component-specific installation the variable `CMAKE_INSTALL_COMPONENT` is set to the name of the current component.

Custom installation scripts, as simple as the message above, may be more easily created with the script code placed inline in the call to the `INSTALL` command. The `CODE` signature is provided for this purpose:

```
install (CODE "<code>")
```

The `CODE` keyword is immediately followed by a string containing the code to place in the installation script. An install-time message may be created using the command

```
install (CODE "MESSAGE(\"Installing My Project\")")
```

which has the same effect as the `message.cmake` script but contains the code inline.

Installing Prerequisite Shared Libraries

Executables are frequently built using shared libraries as building blocks. When you install such an executable, you must also install its prerequisite shared libraries, called "prerequisites" because the executable requires their presence in order to load and run properly. The three main sources for shared libraries are the operating system itself, the build products of your own project and third party libraries belonging to an external project. The ones from the operating system may be relied upon to be present without installing anything: they are on the base platform on which your executable runs. The build products in your own project presumably have add_library build rules in the CMakeLists files, and so it should be straightforward to create CMake install rules for them. It is the third party libraries that frequently become a high maintenance item when there are more than a handful of them or when the set of them fluctuates from version to version of the third party project. Libraries may be added, code may be reorganized, and the third party shared libraries themselves may actually have additional prerequisites that are not obvious at first glance.

CMake provides two modules to make it easier to deal with required shared libraries. The first module, GetPrerequisites.cmake, provides the `get_prerequisites` function to analyze and classify the prerequisite shared libraries upon which an executable depends. Given an executable file as input, it will produce a list of the shared libraries required to run that executable, including any prerequisites of the discovered shared libraries themselves. It uses native tools on the various underlying platforms to perform this analysis: dumpbin (Windows), otool (Mac) and ldd (Linux). The second module, BundleUtilities.cmake, provides the `fixup_bundle` function to copy and fixup prerequisite shared libraries using well-defined locations relative to the executable. For Mac bundle applications, it embeds the libraries inside the bundle, fixing them up with `install_name_tool` to make a self-

contained unit. On Windows, it copies the libraries into the same directory with the executable since executables will search in their own directories for their required DLLs.

The `fixup_bundle` function helps you create relocatable install trees. Mac users appreciate self-contained bundle applications: you can drag them anywhere, double click them and they still work. They do not rely on anything being installed in a certain location other than the operating system itself. Similarly Windows users without administrative privileges appreciate a relocatable install tree where an executable and all of its required DLLs are installed in the same directory and it works no matter where you install it. You can even move things around after installing them and is will still work.

To use `fixup_bundle`, first install one of your executable targets. Then, configure a CMake script that can be called at install time. Inside the configured CMake script, simply include BundleUtilities and call the `fixup_bundle` function with appropriate arguments.

In CMakeLists.txt:

```
install (TARGETS myExecutable DESTINATION bin)

# To install, for example, MSVC runtime libraries:
include (InstallRequiredSystemLibraries)

# To install other/non-system 3rd party required libraries:
configure_file (
  ${CMAKE_CURRENT_SOURCE_DIR}/FixBundle.cmake.in
  ${CMAKE_CURRENT_BINARY_DIR}/FixBundle.cmake
  @ONLY
  )

install (SCRIPT ${CMAKE_CURRENT_BINARY_DIR}/FixBundle.cmake)
```

In FixBundle.cmake.in:

```
include (BundleUtilities)

# Set bundle to the full path name of the executable already
# existing in the install tree:
set (bundle
"${CMAKE_INSTALL_PREFIX}/myExecutable@CMAKE_EXECUTABLE_SUFFIX@")

# Set other_libs to a list of full path names to additional
# libraries that cannot be reached by dependency analysis.
# (Dynamically loaded PlugIns, for example.)
```

```
set (other_libs "")

# Set dirs to a list of directories where prerequisite libraries
# may be found:
set (dirs "@LIBRARY_OUTPUT_PATH@")

fixup_bundle ("${bundle}" "${other_libs}" "${dirs}")
```

You are responsible for verifying that you have permission to copy and distribute the prerequisite shared libraries for your executable. Some libraries may have restrictive software licenses that prohibit making copies a la `fixup_bundle`.

Exporting and Importing Targets

CMake 2.6 introduced support for exporting targets from one CMake-based project and importing them into another. The main feature allowing this functionality is the notion of an `IMPORTED` target. Here we present imported targets and then show how CMake files may be generated by a project to export its targets for use by other projects.

Importing Targets

Imported targets are used to convert files outside of the project on disk into logical targets inside a CMake project. They are created using the `IMPORTED` option to the `add_executable` and `add_library` commands. No build files are generated for imported targets. They are used simply for convenient, flexible reference to outside executables and libraries. Consider the following example which creates and uses an `IMPORTED` executable target:

```
add_executable (generator IMPORTED)                              # 1
set_property (TARGET generator PROPERTY
              IMPORTED_LOCATION "/path/to/some_generator") # 2

add_custom_command (OUTPUT generated.c
                    COMMAND generator generated.c)          # 3

add_executable (myexe src1.c src2.c generated.c)
```

Line #1 creates a new CMake target called `generator`. Line #2 tells CMake the location of the target on disk to import. Line #3 references the target in a custom command. Once CMake is run the generated build system will contain a command line such as

```
/path/to/some_generator /project/binary/dir/generated.c
```

in the rule to generate the source file. In a similar manner libraries from other projects may be used through `IMPORTED` targets:

```
add_library (foo IMPORTED)
set_property (TARGET foo PROPERTY
              IMPORTED_LOCATION "/path/to/libfoo.a")
add_executable (myexe src1.c src2.c)
target_link_libraries (myexe foo)
```

On Windows a .dll and its .lib import library may be imported together:

```
add_library (bar IMPORTED)
set_property (TARGET bar PROPERTY
              IMPORTED_LOCATION "c:/path/to/bar.dll")
set_property (TARGET bar PROPERTY
              IMPORTED_IMPLIB "c:/path/to/bar.lib")
add_executable (myexe src1.c src2.c)
target_link_libraries (myexe bar)
```

A library with multiple configurations may be imported with a single target:

```
add_library (foo IMPORTED)
set_property (TARGET foo PROPERTY
              IMPORTED_LOCATION_RELEASE "c:/path/to/foo.lib")
set_property (TARGET foo PROPERTY
              IMPORTED_LOCATION_DEBUG "c:/path/to/foo_d.lib")
add_executable (myexe src1.c src2.c)
target_link_libraries (myexe foo)
```

The generated build system will link `myexe` to `foo.lib` when it is built in the release configuration and `foo_d.lib` when built in the debug configuration.

Exporting Targets

Imported targets on their own are useful, but they still require the project that imports them to know the locations of the target files on disk. The real power of imported targets is when the project providing the target files also provides a file to help import them.

The `install(TARGETS)` and `install(EXPORT)` commands work together to install both a target and a CMake file to help import it. For example, the code

```
add_executable (generator generator.c)
```

```
install (TARGETS generator DESTINATION lib/myproj/generators
         EXPORT myproj-targets)
install (EXPORT myproj-targets DESTINATION lib/myproj)
```

will install the two files

```
<prefix>/lib/myproj/generators/generator
<prefix>/lib/myproj/myproj-targets.cmake
```

The first is the regular executable named `generator`. The second file, `myproj-targets.cmake`, is a CMake file designed to make it easy to import `generator`. This file contains code such as

```
get_filename_component (_self "${CMAKE_CURRENT_LIST_FILE}" PATH)
get_filename_component (PREFIX "${_self}/../.." ABSOLUTE)
add_executable (generator IMPORTED)
set_property (TARGET generator PROPERTY
  IMPORTED_LOCATION "${PREFIX}/lib/myproj/generators/generator")
```

(note that `${PREFIX}` is computed relative to the file location). An outside project may now use generator as follows:

```
include (${PREFIX}/lib/myproj/myproj-targets.cmake) # 1
add_custom_command (OUTPUT generated.c
                    COMMAND generator generated.c)  # 2
add_executable (myexe src1.c src2.c generated.c)
```

Line #1 loads the target import script (see section 5.7 to make this automatic). The script may import any number of targets. Their locations are computed relative to the script location so the install tree may be easily moved. Line #2 references the generator executable in a custom command. The resulting build system will run the executable from its installed location. Libraries may also be exported and imported:

```
add_library (foo STATIC foo1.c)
install (TARGETS foo DESTINATION lib EXPORTS myproj-targets)
install (EXPORT myproj-targets DESTINATION lib/myproj)
```

This installs the library and an import file referencing it. Outside projects may simply write

```
include (${PREFIX}/lib/myproj/myproj-targets.cmake)
add_executable (myexe src1.c)
target_link_libraries (myexe foo)
```

and the executable will be linked to the library `foo` exported and installed by the original project.

Any number of target installations may be associated with the same export name. The export names are considered global so any directory may contribute a target installation. Only one call to the `install (EXPORT)` command is needed to install an import file that references all targets. Both of the examples above may be combined into a single export file, even if they are in different subdirectories of the project as shown in the code below.

```
# A/CMakeLists.txt
add_executable (generator generator.c)
install (TARGETS generator DESTINATION lib/myproj/generators
         EXPORT myproj-targets)

# B/CMakeLists.txt
add_library (foo STATIC foo1.c)
install (TARGETS foo DESTINATION lib EXPORTS myproj-targets)

# Top CMakeLists.txt
add_subdirectory (A)
add_subdirectory (B)
install (EXPORT myproj-targets DESTINATION lib/myproj)
```

Typically projects are built and installed before being used by an outside project. However in some cases it is desirable to export targets directly from a build tree. The targets may then be used by an outside project that references the build tree with no installation involved. The `export` command is used to generate a file exporting targets from a project build tree. For example, the code

```
add_executable (generator generator.c)
export (TARGETS generator FILE myproj-exports.cmake)
```

will create a file in the project build tree called `myproj-exports.cmake` that contains the required code to import the target. This file may be loaded by an outside project that is aware of the project build tree in order to use the executable to generate a source file. An example application of this feature is for building a generator executable on a host platform when cross compiling. The project containing the generator executable may be built on the host platform and then the project that is being cross-compiled for another platform may load it.

4.12 Advanced Commands

There are a few commands that can be very useful but are not typically used in writing CMakeLists files. This section will discuss a few of these commands and when they are useful. First consider the `add_dependencies` command which creates a dependency between two targets. CMake automatically creates dependencies between targets when it can determine them. For example, CMake will automatically create a dependency for an executable target that depends on a library target. The `add_dependencies` command is typically used to specify inter target dependencies between targets where at least one of the targets is a custom target (see section 6.4 for more information on custom targets).

The `include_regular_expression` command also relates to dependencies. This command controls the regular expression that is used for tracing source code dependencies. By default CMake will trace all the dependencies for a source file including system include files such as `stdio.h`. If you specify a regular expression with the `include_regular_expression` command that regular expression will be used to limit what include files are processed. For example; if your software project's include files all started with the prefix foo (e.g. `fooMain.c fooStruct.h` etc) then you could specify a regular expression of `^foo.*$` to limit the dependency checking to just the files of your project.

Occasionally you might want to get a listing of all the source files that another source file depends on. This is useful when you have a program that uses pieces of a large library but you are not sure what pieces it is using. The `output_required_files` command will take a source file and produce a list of all the other source files it depends on. You could then use this list to produce a reduced version of the library that only contains the necessary files for your program.

Some tools such as Rational Purify on the Sun platform are run by inserting an extra command before the final link step. So, instead of

```
CC foo.o -o foo
```

The link step would be

```
purify CC foo.o -o foo
```

It is possible to do this with CMake. To run an extra program in front of the link line change the rule variables `CMAKE_CXX_LINK_EXECUTABLE`, and `CMAKE_C_LINK_EXECUTABLE`. Rule variables are described in chapter 11. The values for these variables are contained in the file `Modules/CMakeDefaultMakeRuleVariables.cmake`, and they are sometimes redefined

in `Modules/Platform/*.cmake`. Make sure it is set after the `PROJECT` command in the CMakeLists file. Here is a small example of using purify to link a program called foo:

```
project (foo)

set (CMAKE_CXX_LINK_EXECUTABLE
    "purify ${CMAKE_CXX_LINK_EXECUTABLE}")
add_executable (foo foo.cxx)
```

Of course, for a generic CMakeLists file you should have some `if` checks for the correct platform. This will only work for the Makefile generators because the rule variables are not used by the IDE generators. Another option would be to use $(PURIFY) instead of plain purify. This would pass through CMake into the Makefile and be a make variable. The variable could be defined on the command line like this: make PURIFY=purify. If not specified then it would just use the regular rule for linking a C++ executable as PURIFY would be expanded by make to nothing.