Custom Commands and Targets

Frequently, the build process for a software project goes beyond simply compiling libraries and executables. In many cases additional tasks may be required during or after the build process. Common examples include: compiling documentation using a documentation package, generating source files by running another executable, generating files using tools for which CMake doesn't have rules (such as lex, and yacc), moving the resulting executables, post processing the executable, etc. CMake supports these additional tasks using custom commands and custom targets. This chapter will describe how to use custom commands and targets to perform complex tasks that CMake does not inherently support.

6.1 Portable Custom Commands

Before going into detail on how to use custom commands, we will discuss how to deal with some of their portability issues. Custom commands typically involve running programs with files as inputs or outputs. Even a simple command such as copying a file can be tricky to do in a cross-platform way. For example, copying a file on UNIX is done with the cp command, while on windows it is done with the copy command. To make matters worse, frequently the names of the files will change on different platforms. Executables on Windows end with .exe while on UNIX they do not. Even between UNIX implementations there are differences such as what extensions are used for shared libraries; .so, .sl, .dylib, etc.

CMake provides two main tools for handling these differences. The first is the `-E` option (short for execute) to `cmake`. When the `cmake` executable is passed the `-E` option it acts as a general purpose cross-platform utility command. The arguments following the `-E` option indicate what `cmake` should do. Some of the options include:

chdir dir command args

Changes the current directory to `dir` and then execute the command with the provided arguments.

copy file destination

Copies a file from one directory or filename to another.

copy_if_different in-file out-file

copy_if_different first checks to see if the files are different before copying them. copy_if_different is critical in many rules since the build process is based on file modification times. If the copied file is used as the input to another build rule then copy_if_different can eliminate unnecessary recompilations.

copy_directory source destination

This option copies the source directory including any subdirectories to the destination directory.

remove file1 file2 …

Removes the listed files from the disk.

echo string

Echos a string to the console. This is useful for providing output during the build process.

time command args

Runs the command and times its execution.

These options provide a platform independent way to perform a few common tasks. The `cmake` executable can be referenced by using the `CMAKE_COMMAND` variable in your CMakeLists files as later examples will show.

The second tool CMake provides to address portability issues is a number of variables describing the characteristics of the platform. While most of these are covered in Appendix A - Variables, some are particularly useful for custom commands.

EXE_EXTENSION

This is the file extension for executables. Typically nothing on UNIX and .exe on Windows

CMAKE_CFG_INTDIR

Development environments such as Visual Studio and Xcode use subdirectories based on the build type selected, such as Release or Debug. When performing a custom command on a library, executable or object file you will typically need the full path to the file. `CMAKE_CFG_INTDIR` on UNIX will typically be "`./`" while for Visual Studio it will be set to "`$(INTDIR)/`", which at build time will be replaced by the selected configuration.

CMAKE_CURRENT_BINARY_DIR

This is the full path to the output directory associated with the current CMakeLists file. This may be different from `PROJECT_BINARY_DIR` which is the full path to the top of the current project's binary tree.

CMAKE_CURRENT_SOURCE_DIR

This is the full path to the source directory associated with the current CMakeLists file. This may be different from `PROJECT_SOURCE_DIR` which is the full path to the top of the current project's source tree.

EXECUTABLE_OUTPUT_PATH

Some projects specify a directory into which all the executables should be built. This variable, if defined, holds the full path to that directory.

LIBRARY_OUTPUT_PATH

Some projects specify a directory into which all the libraries should be built. This variable, if defined, holds the full path to that directory.

There are also a series of variables such as `CMAKE_SHARED_MODULE_PREFIX` and ...`SUFFIX` that describe the current platform's prefix and suffix for that type of file. These variables are defined for `SHARED_MODULE`, `SHARED_LIBRARY`, and `LIBRARY`. Using these variables, you can typically construct the full path to any CMake generated file that you need to. For libraries and executable targets you can also use `get_target_property` with the `LOCATION` argument to get the full path to the target.

CMake doesn't limit you to using `cmake -E` in all your commands. You can use any command that you like, although you should consider portability issues when doing it. A common practice is to use `find_program` to find an executable (perl for example), and then use that executable in your custom commands.

6.2 Using add_custom_command on a Target

Now we will consider the signature for `add_custom_command`. In Makefile terminology `add_custom_command` adds a rule to a Makefile. For those more familiar with Visual Studio, it adds a custom build step to a file. `add_custom_command` has two main signatures, one for adding a custom command to a target and one for adding a custom command to build a file. When adding a custom command to a target the signature is as follows:

```
add_custom_command (
  TARGET target
  PRE_BUILD | PRE_LINK | POST_BUILD
  COMMAND command [ARGS arg1 arg2 arg3 …]
  [COMMAND command [ARGS arg1 arg2 arg3 …] …]
  [COMMENT comment]
  )
```

The target is the name of a CMake target (executable, library, or custom) to which you want to add the custom command. There is a choice of when the custom command should be executed. `PRE_BUILD` indicates that the command should be executed before any other dependencies for the target are built. `PRE_LINK` indicates that the command should be executed after the dependencies are all built, but before the actual link command. `POST_BUILD` indicates that the custom command should be executed after the target has been built. The `COMMAND` argument is the command (executable) to run and `ARGS` provides an optional list of arguments to the command. Finally the `COMMENT` argument can be used to provide a quoted string to be used as output when this custom command is run. This is useful if you want to provide some feedback or documentation on what is happening during the build. You can specify as many commands as you want for a custom command. They will be executed in the order specified.

How to Copy an Executable Once it is Built?

Now let us consider a simple custom command for copying an executable once it has been built.

```
# first define the executable target as usual
add_executable (Foo bar.c)

# get where the executable will be located
get_target_property (EXE_LOC Foo LOCATION)

# then add the custom command to copy it
add_custom_command (
  TARGET Foo
  POST_BUILD
  COMMAND ${CMAKE_COMMAND}
  ARGS -E copy ${EXE_LOC} /testing_department/files
  )
```

The first command in this example is the standard command for creating an executable from a list of source files. In this case an executable named `Foo` is created from the source file `bar.c`. The next command is `get_target_property` which will set the variable called

EXE_LOC to where the executable will be built. Next is the add_custom_command invocation. In this case the target is simply Foo and we are adding a post build command. The command to execute is cmake which has its full path specified in the CMAKE_COMMAND variable. Its arguments are "-E copy" and the source and destination locations. In this case it will copy the Foo executable from where it was built into the /testing_department/files directory. Note that the TARGET parameter accepts a CMake target (Foo in this example), but most commands such as cmake -E copy will require the full path to the executable which can be retrieved using GET_TARGET_PROPERTY.

6.3 Using add_custom_command to Generate a File

The second use for add_custom_command is to add a rule for how to build an output file. In this case the rule provided will replace any current rules for building that file. The signature is as follows:

```
add_custom_command (OUTPUT output1 [output2 …]
  COMMAND command [ARGS [args...]]
  [COMMAND command [ARGS arg1 arg2 arg3 …] …]
  [MAIN_DEPENDENCY depend]
  [DEPENDS [depends...]]
  [COMMENT comment]
  )
```

The OUTPUT is the file (or files) that will result from running this custom command, the COMMAND and ARGS parameters are the command to execute and the arguments to pass to it. As with the prior signature you can have as many commands as you wish. The DEPENDS are files or executables on which this custom command depends. If any of these dependencies change this custom command will re-execute. The MAIN_DEPENDENCY is an optional argument that acts as a regular dependency and under Visual Studio it provides a suggestion for what file to hang this custom command onto. If the MAIN_DEPENDENCY is not specified then one will be created automatically by CMake. The MAIN_DEPENDENCY should not be a regular .c or .cxx file since the custom command will override the default build rule for the file. Finally the optional COMMENT is a comment that may be used by some generators to provide additional information during the build process.

Using an Executable to Build a Source File

Sometimes a software project builds an executable, which is then used for generating source files that are then used to build other executables or libraries. This may sound like an odd case, but it occurs quite frequently. One example is the build process for the TIFF library, which creates an executable that is then run to generate a source file that has system specific information in it. This file is then used as a source file in building the main TIFF library. Another example is the Visualization Toolkit that builds an executable called vtkWrapTcl that wraps C++ classes into Tcl. The executable is built and then used to create more source files for the build process.

```
###################################################
# Test using a compiled program to create a file
####################################################

# add the executable that will create the file
# build creator executable from creator.cxx
add_executable (creator creator.cxx)

get_target_property (creator EXE_LOC LOCATION)

# add the custom command to produce created.c
add_custom_command (
  OUTPUT ${PROJECT_BINARY_DIR}/created.c
  DEPENDS creator
  COMMAND ${EXE_LOC}
  ARGS ${PROJECT_BINARY_DIR}/created.c
  )

# add an executable that uses created.c
add_executable (Foo ${PROJECT_BINARY_DIR}/created.c)
```

The first part of this example produces the `creator` executable from the source file `creator.cxx`. The custom command then sets up a rule for producing the source file `created.c` by running the executable `creator`. The custom command depends on the `creator` target and writes its result into the output tree (`PROJECT_BINARY_DIR`). Finally, an executable target called `Foo` is added that is built using the `created.c` source file. CMake will create all the required rules in the Makefile (or Visual Studio workspace) such that when you build the project, first the `creator` executable will be built, then it will be run to create `created.c`, which will then be used to build the `Foo` executable.

6.4 Adding a Custom Target

In the discussion so far CMake targets have generally referred to executables and libraries. CMake supports a more general notion of targets, called custom targets, that can be used whenever you want the notion of a target but the end product will not be a library or executable. Examples of custom targets include targets to build documentation, run tests, or update web pages. To add a custom target you use the `add_custom_target` command with the following signature:

```
add_custom_target ( name [ALL]
  [command arg arg arg ... ]
  [DEPENDS depend depend depend ... ]
  )
```

The name specified will be the name given to the target. You can use that name to specifically build that target with Makefiles (make name) or Visual Studio (right click on the target and then select Build). If the optional `ALL` argument is specified then this target will be included in the `ALL_BUILD` target and will automatically be built whenever the Makefile or Project is built. The command and arguments are optional, and if specified will be added to the target as a post build command. For custom targets that will only execute a command this is all you will need. More complex custom targets may depend on other files, in these cases the `DEPENDS` arguments are used to list what files this target depends on. We will consider examples of both cases. First let us look at a custom target that has no dependencies:

```
add_custom_target ( FooJAR ALL
  ${JAR} -cvf "\"${PROJECT_BINARY_DIR}/Foo.jar\""
              "\"${PROJECT_SOURCE_DIR}/Java\""
  )
```

With the above definition, whenever the `FooJAR` target is built it will run Java's Archiver (jar) to create the `Foo.jar` file from java classes in the `${PROJECT_SOURCE_DIR}/Java` directory. In essence this type of custom target allows the developer to tie a command to a target so that it can be conveniently invoked during the build process. Now let us consider a more complex example that roughly models the generation of PDF files from LaTeX. In this case the custom target depends on other generated files (mainly the end product .pdf files):

```
# Add the rule to build the .dvi file from the .tex
# file. This relies on LATEX being set correctly
#
add_custom_command(
  OUTPUT   ${PROJECT_BINARY_DIR}/doc1.dvi
  DEPENDS  ${PROJECT_SOURCE_DIR}/doc1.tex
  COMMAND  ${LATEX}
  ARGS     ${PROJECT_SOURCE_DIR}/doc1.tex
  )

# Add the rule to produce the .pdf file from the .dvi
# file. This relies on DVIPDF being set correctly
#
add_custom_command(
  OUTPUT   ${PROJECT_BINARY_DIR}/doc1.pdf
  DEPENDS ${PROJECT_BINARY_DIR}/doc1.dvi
  COMMAND ${DVIPDF}
  ARGS    ${PROJECT_BINARY_DIR}/doc1.dvi
  )

# finally add the custom target that when invoked
# will cause the generation of the pdf file
#
add_custom_target ( TDocument ALL
  DEPENDS ${PROJECT_BINARY_DIR}/doc1.pdf
  )
```

This example makes use of both `add_custom_command` and `add_custom_target`. The two `add_custom_command` invocations are used to specify the rules for producing a .pdf file from a .tex file. In this case there are two steps and two custom commands. First a .dvi file is produced from the .tex file by running LaTeX, then the .dvi file is processed to produce the desired .pdf file. Finally a custom target is added called TDocument. Its command simply echoes out what it is doing, while the real work is done by the two custom commands. The `DEPENDS` argument sets up a dependency between the custom target and the custom commands. When TDocument is built it will first look to see if all of its dependencies are built. If any are not built it will invoke the appropriate custom commands to build them. This example can be shortened by combining the two custom commands into one custom command, as shown in the following example:

```
# Add the rule to build the .pdf file from the .tex
# file. This relies on LATEX and DVIPDF being set correctly
#
add_custom_command(
  OUTPUT  ${PROJECT_BINARY_DIR}/doc1.pdf
  DEPENDS ${PROJECT_SOURCE_DIR}/doc1.tex
  COMMAND ${LATEX}
  ARGS    ${PROJECT_SOURCE_DIR}/doc1.tex
  COMMAND ${DVIPDF}
  ARGS    ${PROJECT_BINARY_DIR}/doc1.dvi
  )

# finally add the custom target that when invoked
# will cause the generation of the pdf file
#
add_custom_target ( TDocument ALL
  DEPENDS ${PROJECT_BINARY_DIR}/doc1.pdf
  )
```

Now consider the case in which the documentation consists of multiple files. The above example can be modified to handle many files using a list of inputs and a foreach loop. For example:

```
# set the list of documents to process
set (DOCS doc1 doc2 doc3)

# add the custom commands for each document
foreach (DOC ${DOCS})

  add_custom_command (
    OUTPUT  ${PROJECT_BINARY_DIR}/${DOC}.pdf
    DEPENDS ${PROJECT_SOURCE_DIR}/${DOC}.tex
    COMMAND ${LATEX}
    ARGS    ${PROJECT_SOURCE_DIR}/${DOC}.tex
    COMMAND ${DVIPDF}
    ARGS    ${PROJECT_BINARY_DIR}/${DOC}.dvi
    )

  # build a list of all the results
  set (DOC_RESULTS ${DOC_RESULTS}
    ${PROJECT_BINARY_DIR}/${DOC}.pdf
    )

endforeach (DOC)

# finally add the custom target that when invoked
# will cause the generation of the pdf file
#
add_custom_target ( TDocument ALL
  DEPENDS ${DOC_RESULTS}
  )
```

In this example building the custom target `TDocument` will cause all of the specified .pdf files to be generated. Adding a new document to the list is simply a matter of adding its filename to the `DOCS` variable at the top of the example.

6.5 Specifying Dependencies and Outputs

When using custom commands and custom targets you will often be specifying dependencies. When you specify a dependency or the output of a custom command you should always specify the full path. For example, if the command produces `foo.h` in the binary tree then its output should be something like `${PROJECT_BINARY_DIR}/foo.h`. CMake will try to determine the correct path for the file if it is not specified, complex projects frequently end up using files in both the source and build trees, this can eventually lead to errors if the full paths are not specified.

When specifying a target as a dependency you can leave off the full path and executable extension, referencing it simply by its name. Consider the specification of the generator target as an `add_custom_command` dependency in the example on page 108. CMake recognizes `creator` as matching an existing target and properly handles the dependencies.

6.6 When There Isn't One Rule For One Output

There are a couple of unusual cases that can arise when using custom commands that warrant further explanation. The first is a case where one command (or executable) can create multiple outputs, and the second is the case where multiple commands can be used to create a single output.

A Single Command Producing Multiple Outputs

In CMake a custom command can produce multiple outputs simply by listing multiple outputs after the OUTPUT keyword. CMake will create the correct rules for your build system so that no matter which output is required for a target the right rules will be run. If the executable happens to produce a few outputs, but the build process is only using one of them, then you can simply ignore the other outputs when creating your custom command. Say that the executable produces a source file that is used in the build process and also an execution log that is not used. The custom command should specify the source file as the output and ignore the fact that a log file is also generated.

Another case of having one command with multiple outputs is the case where the command is the same but the arguments to it change. This is effectively the same as having a different command and each case should have its own custom command. An example of this was the documentation example on page 111, where a custom command was added for each .tex file. The command is the same but the arguments passed to it change each time.

Having One Output That Can Be Generated By Different Commands

In rare cases you may find that you have more than one command that you can use to generate an output. Most build systems such as make and Visual Studio do not support this and likewise CMake does not. There are two common approaches that can be used to resolve this. If you truly have two different commands that produce the same output, and no other significant outputs, then you can simply pick one of them and create a custom command for that one.

In the more complex case there are multiple commands with multiple outputs. For example:

```
Command1 produces foo.h and bar.h
Command2 produces widget.h and bar.h
```

There are a few approaches that can be used in this case. In some cases you might just combine both commands and all three outputs into a single custom command, so that whenever one output is required all three are built at the same time. You could also create three custom commands, one for each unique output. The custom command for `foo.h` would invoke Command1 while the one for `widget.h` would invoke Command2. When specifying the custom command for `bar.h` you could choose either Command1 or Command2.