
The CMake Programming Language

This section provides an overview of the CMake language syntax and features, but it discusses only a few of them in detail. After all, the language might at first seem different from GNU Make, Ant, or SCons, but you'll quickly realize that most of the concepts are the same. This section covers the following topics:

- CMake language basics: The basic syntax of invoking commands, setting and accessing variables, and managing source and object file properties
- Building executable programs and libraries: How to compile C source files into libraries or executable programs
- Control flow: How to test conditions, repeat operations in a loop, and define macros
- Cross-platform support: Locating tools, libraries, and header files on the native build machine
- Generating a build system: Generating a native build system (such as a makefile)

Throughout this discussion, keep in mind that the CMake build description must be easy to map into an equivalent description for other build tools. It's not reasonable for CMake to have too many advanced language features of its own. For example, if the CMake language provided support for a general-purpose scripting language, such as Python, it would be challenging to translate this into an equivalent GNU Make build description.

CMake Language Basics

The syntax of the CMake build description file (`CMakeLists.txt`) isn't too hard to understand, so you can learn the basics by looking at an example. This example defines a couple variables, sets a property on two different source files, and then displays some messages:

```
 1  project (basic-syntax C)
 2
 3  cmake_minimum_required (VERSION 2.6)
 4
 5  set (wife Grace)
 6  set (dog Stan)
 7  message ("${wife}, please take ${dog} for a walk")
 8
 9  set_property (SOURCE add.c PROPERTY Author Peter)
10  set_property (SOURCE mult.c PROPERTY Author John)
11  get_property (author_name SOURCE add.c PROPERTY Author)
12  message("The author of add.c is ${author_name}")
```

The first observation is that all commands are invoked using a standard syntax, with arguments separated by spaces.

```
command ( arg1 arg2 ... )
```

The arguments can be numbers, filenames, strings, or property names, with the exact syntax requirements depending on which command is used. To group multiple words into a single argument, place quotation marks around the entire string.

The `project` command (on line 1) provides a name to uniquely identify the build system. This is used by native build tools (such as Eclipse) that require a project name. It also states which programming languages (such as C, C++, or Java) are to be compiled.

The `cmake_minimum_required` command (on line 3) states that the build description uses commands supported only by CMake version 2.6 or higher. Note that `cmake_minimum_required` expects two arguments, with the first being the `VERSION` keyword. This informs CMake to interpret the second argument (`2.6`) as a version number.

Lines 5 and 6 demonstrate the creation of variables. The first argument is the variable's name, and the second is the value. Line 7 uses the familiar `${...}` syntax to access each variable's value.

Line 9 introduces the concept of a **property**, using the `set_property` command. Properties enable you to assign a value within the scope of a specific disk file. The build system simply associates the value with that file's name instead of modifying the file content itself. Other commands are free to reference the property's value.

This example sets the `Author` property on the `add.c` source file. This value is limited in scope to `add.c`, so it's possible to assign different `Author` values to other source files. Line 10 sets the `Author` property on `mult.c`, but to a different value.

On line 11, the `get_property` command fetches the `Author` property associated with the `add.c` file. The resulting value is assigned to the `author_name` variable and is displayed on line 12.

Now that you understand the basic syntax, let's see how libraries and executable programs are created.

Building Executable Programs and Libraries

As usual, the most popular operation is to compile source files into libraries and executable programs. CMake provides a number of commands that appear similar to builder methods in SCons but still have a few interesting features of their own.

Creating Executable Programs and Libraries

The following line of code shows how the `calculator` program is compiled from the four source files:

```
1  add_executable (calculator add sub mult calc)
```

This looks simple, but based on past experience with build tools, you can imagine that `add_executable` does a lot of work in the background. This includes constructing a suitable compiler command line, as well as adding the filenames into the dependency graph.

Another observation is that none of the filenames is given a file extension, so CMake must know the correct extension to use for each build machine. For example, when using Microsoft Windows, the resulting program is named `calculator.exe`.

Creating a new library is also similar to SCons, although with a few syntax differences:

```
1  add_library (math STATIC add sub mult)
2  add_executable (calculator calc)
3  target_link_libraries (calculator math)
```

Line 1 produces a static library by compiling `add.c`, `sub.c`, and `mult.c`. The resulting library is given a name that makes sense on the build machine, such as `libmath.a` for UNIX systems. Lines 2 and 3 state that the `calculator` program is created by compiling `calc.c` and then linking it with the `math` library.

To assist the `add_executable` and `add_library` commands, you could use `include_directories` and `link_directories`. These commands inform the C compiler where to find additional header files and tell the linker where it can find additional libraries. As you might expect, these directives are translated into the appropriate compiler flags (such as `-I` and `-L`) in the native build system.

One topic this chapter hasn't mentioned is how CMake determines the dependencies for each source file. In reality, the native build system and build tool do much of this work. If the native build tool already automates the dependency analysis, CMake has nothing to do. On the other hand, for Make-based tools that don't contain this feature, CMake adds the required functionality into the auto-generated build framework.

Setting Compilation Flags

Varying a compilation tool's options is also a useful activity. In contrast to build tools that are more platform-dependent, CMake discourages the use of hard-coded compiler flags. Instead, the build description states which type of output is required and CMake determines the compiler flags to use.

For example, to request that CMake produce a "debug" build in which the executable program contains source-level debugging information, you add the following command to the `CMakeLists.txt` file.

```
set (CMAKE_BUILD_TYPE Debug)
```

Even though the platform-specific flags are abstracted away, CMake generates a native build system with the correct flags for that machine. For example, on a UNIX system, the `-g` flag is added to the C compiler command line.

The same approach is used for adding C preprocessor definitions because each C compiler has its own set of command-line options. This time, you set a property on either the whole directory or an individual file.

```
set_property (DIRECTORY
     PROPERTY COMPILE_DEFINITIONS TEST=1)

set_property (SOURCE add.c
     PROPERTY COMPILE_DEFINITIONS QUICKADD=1)
```

In the first case, you ask the build system to define the `TEST` symbol when compiling all the C files in the current directory. In the second case, you ask that the compilation of `add.c` also include the `QUICKADD` symbol. The native build system adds the required command-line options to make this happen.

Adding Custom Commands and Targets

For more complex build requirements, you can define new compilation tools and have CMake add them to the native build system. Now look at `add_custom_command`, which resembles a standard GNU Make rule, and the

`add_custom_target` command, which is similar to GNU Make's concept of phony targets.

The following example shows how the `/tools/bin/make-data-file` UNIX command translates the `data.dat` source file into the `data.c` output file. In this case, `data.c` is an autogenerated source file.

```
 1  project (custom_command)
 2  cmake_minimum_required(VERSION 2.6)
 3
 4  set (input_data_file ${PROJECT_SOURCE_DIR}/data.dat)
 5  set (output_c_file data.c)
 6
 7  add_custom_command (
 8    OUTPUT ${output_c_file}
 9    COMMAND /tools/bin/make-data-file
10               < ${input_data_file}
11               > ${output_c_file}
12    DEPENDS ${input_data_file}
13  )
14
15  add_executable (print-data ${output_c_file})
```

The bulk of the work is done on lines 7–13, where the custom tool is added to the dependency graph. The `OUTPUT` directive (line 8) states which file will be created, whereas the `DEPENDS` directive (line 12) indicates the input for the command. Lines 9–11 contain the UNIX-dependent shell command to execute. This code is thus equivalent to the following GNU Make program:

```
$(output_c_file) : $(input_data_file)
     /tools/bin/make-data-file < $(input_data_file) >
$(output_c_file)
```

Line 15 is required to make sure there's a top-level target that causes the new tool to be invoked. If you don't define an executable program, the dependency graph won't be complete and you'd have no way to request that `data.c` be created. Unlike many other build tools, CMake makes a clear distinction between top-level targets and individual files in the build tree.

To focus more on this top-level target concept, the `add_custom_target` command facilitates the creation of new top-level targets and specifies the order in which they'll be executed. These are similar to GNU Make phony targets because they're not dependent on whether files are up-to-date and they don't produce any output. They're also similar to Ant targets that use the `depends` attribute to control the order in which targets are invoked.

```
 1  project (custom_target)
 2  cmake_minimum_required(VERSION 2.6)
 3
 4  add_custom_target (print-city ALL
 5        COMMAND echo "Vancouver is a nice city"
 6        COMMAND echo "Even when it rains")
 7
 8  add_custom_target (print-time
 9        COMMAND echo "It is now 2:17pm")
10
```

```
11  add_custom_target (print-day
12        COMMAND echo "Today is Monday")
13
14  add_dependencies (print-city print-time print-day)
```

An interesting part of this code is that line 4 contains the `ALL` keyword to state that `print-city` should be invoked as part of the default build (when the developer doesn't explicitly choose a target). Also, line 14 states that `print-city` depends on `print-time` and `print-day`.

Control Flow

Control flow (conditions, loops, and macros) is similar to other programming languages and doesn't require much explanation. The distinction with CMake is that the CMake generator, not the native build system, evaluates and executes control-flow commands. This might seem a little odd at first, but as long as the native build system ends up with the same behavior described in `CMakeLists.txt`, no problem should occur.

The syntax of the `if` command is fairly standard, perhaps with the exception that `()` is required after the `else` and `endif` statements.

```
set (my_var 1)
if (${my_var})
   message ("my_var is true")
else ()
   message ("my_var is false")
endif ()
```

It's also possible to perform Boolean operations, including `NOT`, `AND`, and `OR`, which, incidentally, don't require the standard `${...}` syntax around variable names.

```
if (NOT my_var)
...
endif ()
```

Variables can be tested against other variables or constant values:

```
if (${my_age} EQUAL 40)
...
endif ()
```

The existence of files can be tested, although keep in mind that the test is performed at the time the native build system is created; it isn't performed by the native build tool itself.

```
if (EXISTS file1.txt)
...
endif ()
```

Likewise, you can test whether one file is newer than a second file.

```
if (file1.txt IS_NEWER_THAN file2.txt)
```

```
...
endif ()
```

Finally, for more complex scenarios, it's possible to match a variable's value against a regular expression.

```
if (${symbol_name} MATCHES "^[a-z][a-z0-9]*$")
...
endif ()
```

The macro construct is similar to a function or method definition in other languages, making it possible to reuse common code. The macro syntax is easy to understand:

```
 1  project (macro)
 2  cmake_minimum_required (VERSION 2.6)
 3
 4  macro (my_macro ARG1 ARG2 ARG3)
 5     message ("The my_macro macro was passed the following
       arguments:")
 6     message ("${ARG1}, ${ARG2} and ${ARG3}")
 7  endmacro (my_macro)
 8
 9  my_macro (1 2 3)
10  my_macro (France Germany Russia)
```

Finally, the `foreach` loop iterates through a list of values:

```
1  project (foreach)
2  cmake_minimum_required (VERSION 2.6)
3
4  foreach (source_file add.c sub.c mult.c calc.c)
5    message ("Calculating cksum for ${source_file}")
6    add_custom_target (cksum-${source_file} ALL
7        COMMAND cksum ${PROJECT_SOURCE_DIR}/${source_file}
8    )
9  endforeach (source_file)
```

This last example requires more explanation. Line 6 adds a new top-level target for each of the source files in the list. Invoking one of these targets invokes the `cksum` command for the associated source file. Assuming that you generate a makefile-based build, you can invoke either all targets at once or each target individually. (The percentages are part of the autogenerated makefile framework.)

```
$ gmake
615245502 109 /home/psmith/loops/src/add.c
[ 25%] Built target cksum-add.c
2090159248 294 /home/psmith/loops/src/calc.c
[ 50%] Built target cksum-calc.c
4029979682 113 /home/psmith/loops/src/mult.c
[ 75%] Built target cksum-mult.c
3864170835 124 /home/psmith/loops/src/sub.c
```

```
[100%] Built target cksum-sub.c

$ gmake cksum-add.c
615245502 109 /home/psmith/loops/src/add.c
[100%] Built target cksum-add.c

$ gmake cksum-calc.c
2090159248 294 /home/psmith/loops/src/calc.c
[100%] Built target cksum-calc.c
```

As mentioned earlier, this looping construct isn't translated into the native build tool's looping construct. Instead, it provides the equivalent functionality by adding a number of different rules to the makefile.

Cross-Platform Support

Continuing with the approach that a CMake build description should be platform neutral, consider how to deal with build machine differences. CMake enables you to locate specific tools and files, and also to identify which features the underlying compiler supports.

Locating Files and Tools on the Build Machine

To create a build system that works on any type of build machine, you can't be too specific about where tools and files are located. At the very least, the tool or file must exist somewhere on the file system, but each machine might store it in a different location.

CMake provides a number of commands to search for files and tools in all the standard paths. The following code locates the `ls` program, the `stdio.h` header file, and the standard C math library.

```
1  project (finding)
2  cmake_minimum_required (VERSION 2.6)
3
4  find_program (LS_PATH ls)
5  message ("The path to the ls program is ${LS_PATH}")
6
7  find_file (STDIO_H_PATH stdio.h)
8  message ("The path to the stdio.h file is ${STDIO_H_
   PATH}")
9
10 find_library (LIB_MATH_PATH m /usr/local/lib /usr/lib64)
11 message ("The path to the math library is ${LIB_MATH_
   PATH}")
```

When this build description is translated into the native build system (by running the `cmake` tool), you see the following output:

```
The path to the ls program is /bin/ls
The path to the stdio.h file is /usr/include/stdio.h
The path to the math library is /usr/lib/libm.so
```

Each type of build machine might give different results, so the build description must reference these variables

to access the tool instead of using a hard-coded pathname.

Note that line 10 explicitly asks the `find_library` command to search for the math library in `/usr/local/lib` and `/usr/lib64`. These paths are searched in addition to CMake's default locations.

To make it easier to write build description files, CMake provides code modules for locating popular tools and libraries. As an example, by including the `FindPerl` module, you can easily locate your build machine's Perl interpreter:

```
 1  project (find-perl)
 2  cmake_minimum_required (VERSION 2.6)
 3
 4  include (FindPerl)
 5  if (PERL_FOUND)
 6    execute_process (
 7        COMMAND ${PERL_EXECUTABLE} ${PROJECT_SOURCE_DIR}/
          config.pl
 8    )
 9  else ()
10    message (SEND_ERROR "There is no perl interpreter on
      this system")
11  endif ()
```

The `FindPerl` module (on line 4) contains a small amount of CMake build description code, which is included by the `CMakeLists.txt` file. This module detects the presence of the Perl interpreter no matter what type of build machine you're executing on (such as Linux or Windows). If Perl can be located, the `PERL_EXECUTABLE` variable contains the absolute path of the program, and the `PERL_FOUND` variable is set to a true value.

Notice that the `execute_process` command (on line 6) passes the `config.pl` file into the Perl interpreter. This invocation takes place as part of *generating* the native build system. In contrast, the `add_custom_command` directive you saw earlier adds references to Perl *within* the native build system, to be invoked when the native build tool is used.

Testing for Source Code Capabilities

A second type of cross-platform support is the capability to test the underlying compilers. Before attempting to compile a program, you must determine whether the build machine's compiler provides all the required functions and header files. If it doesn't, you must substitute your own implementation or even abort the build process.

CMake provides the `try_compile` and `try_run` commands, enabling you to determine whether a snippet of C/C++ code compiles correctly. If it does compile, you can try to execute the program to see if it provides the correct output. To make it easy to use these commands, CMake wraps them in a number of prewritten macros. For example:

```
1  project (try-compile)
2  cmake_minimum_required(VERSION 2.6)
3
4  include (CheckFunctionExists)
5  include (CheckStructHasMember)
6
7  CHECK_FUNCTION_EXISTS(vsnprintf VSNPRINTF_EXISTS)
```

```
 8  if (NOT VSNPRINTF_EXISTS)
 9    message (SEND_ERROR "vsnprintf not available on this
      build machine")
10  endif ()
11
12  CHECK_STRUCT_HAS_MEMBER("struct rusage" ru_stime wait.h
    HAS_STIME)
13  if (NOT HAS_STIME)
14    message (SEND_ERROR "ru_stime field not available in
      struct rusage")
15  endif ()
```

Lines 7–10 demonstrate the use of the `CHECK_FUNCTION_EXISTS` macro, as defined in the `CheckFunctionExists` module (line 4). By using the `try_compile` command, this macro sets the `VSNPRINTF_EXISTS` variable to indicate whether the `vsnprintf` function was available to the underlying C compiler or linker.

Lines 12–15 perform a similar operation, but this time to determine whether the definition of `struct rusage` contains the `ru_stime` field. If not, the associated variable is left undefined and the build system fails with an error message.

Generating a Native Build System

As discussed earlier, using CMake involves two main phases. The first is to process the `CMakeLists.txt` file and generate a native build system. The second phase is to use a native build tool to actually compile the software. This generation process is a key part of CMake's design, providing support for a wide range of operating systems and native build tools.

Generating the Default Build System

The easiest way to generate a build system is to accept the default configuration. The developer simply creates a directory for the object files and then invokes the `cmake` tool within that directory. No file in the source directory is ever modified, making it possible to generate more than one object directory from the same source tree.

```
$ mkdir obj
$ cd obj
$ cmake ../src
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/psmith/obj
```

CMake attempts to locate each of the required development tools and determines which version of each tool is in use. Any `try_compile` and `try_run` commands in your `CMakeLists.txt` file are also executed at this time.

Assuming that you generated a Make-based native build system (the default for Linux and UNIX), the object directory now contains the following directory structure:

```
Makefile
cmake_install.cmake
CMakeCache.txt
CMakeFiles
CMakeFiles/calculator.dir
CMakeFiles/calculator.dir/cmake_clean.cmake
CMakeFiles/calculator.dir/build.make
CMakeFiles/calculator.dir/depend.make
CMakeFiles/calculator.dir/progress.make
CMakeFiles/calculator.dir/link.txt
CMakeFiles/calculator.dir/flags.make
CMakeFiles/calculator.dir/DependInfo.cmake
...
CMakeFiles/progress.make
CMakeFiles/Makefile.cmake
CMakeFiles/CMakeDetermineCompilerABI_C.bin
CMakeFiles/CMakeOutput.log
CMakeFiles/CMakeCXXCompiler.cmake
```

This build framework certainly includes a lot of files, but the most important files to notice are these:

- `Makefile` : The main entry point for the native build system.
- `CMakeCache.txt` : A text-based configuration file that contains the autodiscovered settings for this build machine. (You'll learn more about this shortly.)
- `CMakeFiles/` : A directory that contains all the autogenerated framework files. These are included by the main makefile.

The final step is to invoke the native build tool; in this case, you use `gmake`.

```
$ gmake
Scanning dependencies of target calculator
[ 25%] Building C object CMakeFiles/calculator.dir/add.c.o
[ 50%] Building C object CMakeFiles/calculator.dir/sub.c.o
[ 75%] Building C object CMakeFiles/calculator.dir/mult.c.o
[100%] Building C object CMakeFiles/calculator.dir/calc.c.o
Linking C executable calculator
[100%] Built target calculator
```

Notice how the autogenerated build framework displays a significant amount of customized output instead of simply showing the underlying commands as in the GNU Make examples.

Generating a Nondefault Build System

One of CMake's strengths is flexibility in selecting the type of native build system to be generated. By passing the `-G` option to the `cmake` command, you can override the default selection. For example, to generate a Visual Studio 10 project, enter the following command on your Windows build machine:

```
cmake -G "Visual Studio 10" ..\src
```

Likewise, to generate a build system for Eclipse CDT version 4 on Linux, use the following:

```
cmake -G "Eclipse CDT4 - Unix Makefiles" ../src
```

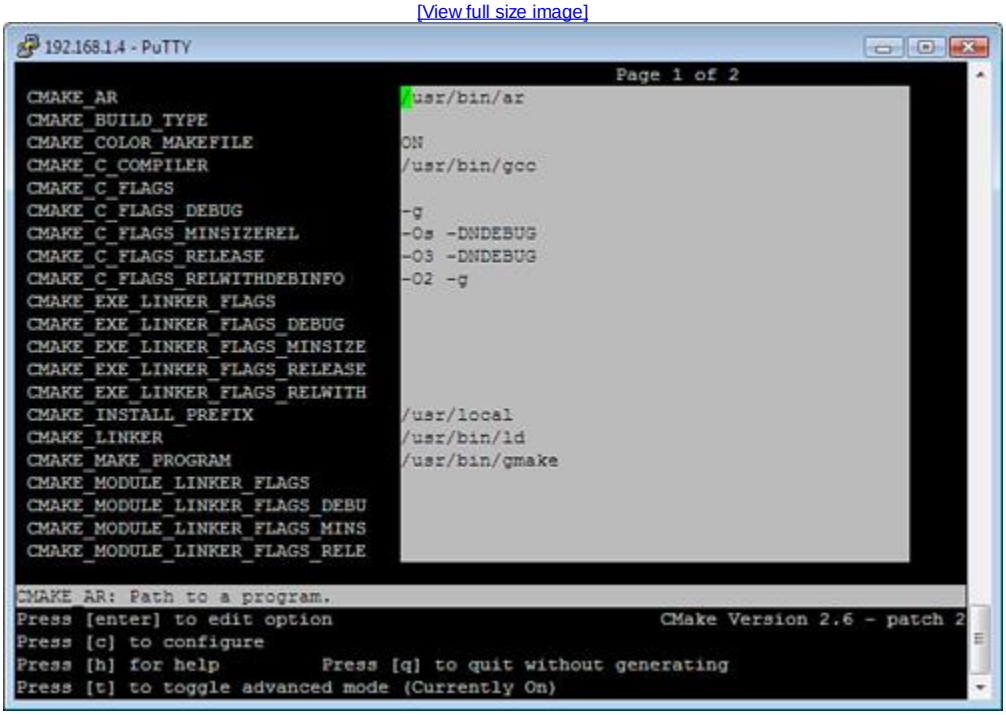
Naturally, it's possible to add more CMake generators if your choice of development environment isn't already supported, but doing so isn't an easy task.

Customizing the Generation Step

Although CMake's default behavior is to autodetect the build machine's compilation tools, it often makes sense to overwrite these values. In addition to the basic `cmake` command, you can use the `ccmake` command (see Figure 9.2) to interactively configure the native build system.

Figure 9.2. Configuration using the ccmake configuration tool.

[View full size image]



The variables in this list are collectively known as the **cache** and are stored in the object directory's `CMakeCache.txt` file. Each variable has a default value but can easily be modified to customize the build process. The following are some of the most commonly used cache variables:

- `CMAKE_AR`, `CMAKE_C_COMPILER`, and `CMAKE_LINKER`: The absolute path to the library archiver tool, the C compiler, and the object file linker. These can be overwritten if your build system uses custom tools in place of the build machine's standard tools.

- `CMAKE_MAKE_PROGRAM` : The absolute path to the native build tool, such as `/usr/bin/gmake`. This can be overwritten to use a nonstandard version of the tool.
- `CMAKE_BUILD_TYPE` : The type of the build tree you want to create. Possible values include these:
  - `Debug` : The generated object files and executable program will contain debugging information.
  - `Release` : The resulting executable program will be fully optimized and won't contain debug information.
  - `RelWithDebInfo` : The executable program will be optimized but will also contain debug information.
  - `MinSizeRel` : The executable program will be optimized to require as little memory space as possible
- `CMAKE_C_FLAGS_*` : For each of the four build types just listed, these variables state which C compilation flags should be used. That is, depending on the value in the `CMAKE_BUILD_TYPE` variable, CMake will use the C compiler flags listed in the corresponding cache variable.
- `CMAKE_EXE_LINKER_FLAGS_*` : Similar, but provides the linker flags for each of the four build types.

As you'll see later, it's possible to define your own cache variables and initialize them to a default value. The `CMakeLists.txt` build description can read all cache values as if they were normal variables. The values can also be written to using the standard `set` command.

Translation from `CMakeLists.txt` to the Native Build System

A final consideration in using CMake is to understand when and how each of the CMake commands is translated into the native build system. The chapter has already touched on this topic briefly, but it's worth mentioning the detail a second time.

CMake commands can be divided into two main groups:

1. Commands that take effect immediately when the `cmake` tool is invoked. These include control-flow commands such as `if`, `foreach`, and `macro`, as well as commands for setting and displaying variable values.

2. Commands that are translated into native build system constructs. These include `add_executable`, `add_library`, `add_custom_command`, and `add_custom_target`.

As you can imagine, the second category of commands contributes to the native build system's dependency graph. On the other hand, the first set of commands enables you to control what gets added. You can use variables to control the filenames that are added, loops to individually add a large number of files, and macros to simplify the build description. The important fact is that only the commands that impact the dependency graph are directly translated to the native build system.

Here's a simple example to illustrate the concept. The following build description compiles two separate programs, `calc` and `calculator`, both using the same source files.

```
 1  project (generating)
 2  cmake_minimum_required(VERSION 2.6)
 3
 4  set (prog1 calculator)
 5  set (prog2 calc)
 6
 7  execute_process (
 8    COMMAND date
 9    OUTPUT_VARIABLE TIME_NOW
10  )
11
12  foreach (prog_name ${prog1} ${prog2})
13    message ("Constructing program ${prog_name} at ${TIME_
      NOW}")
14    add_executable (${prog_name} add sub mult calc)
15  endforeach ()
```

To make things interesting, the description is more complex than it needs to be, although you shouldn't have trouble following along. When the `cmake` tool is invoked, you'll see the following output (note the additional messages):

```
$ cmake ../src
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
Constructing program calculator at Sun Jun  6 16:05:28 PDT 2010
Constructing program calc at Sun Jun  6 16:05:28 PDT 2010
-- Configuring done
-- Generating done
-- Build files have been written to: /home/psmith/obj
```

You can learn from this output that both `set` commands were executed, the `execute_process` command invoked the `date` shell command, the body of the `foreach` loop was executed twice, and the `message` command displayed two messages on the output. Effectively, the entire program was executed at the same time the native build system was generated.

If you also take the view that `add_executable` is supposed to do nothing more than add information to the native build system, its task is also complete. However, the executable programs (two of them) aren't actually created until you invoke the native build tool.

```
$ gmake
Scanning dependencies of target calc
[ 12%] Building C object CMakeFiles/calc.dir/add.c.o
```

```
[ 25%] Building C object CMakeFiles/calc.dir/sub.c.o
[ 37%] Building C object CMakeFiles/calc.dir/mult.c.o
[ 50%] Building C object CMakeFiles/calc.dir/calc.c.o
Linking C executable calc
[ 50%] Built target calc
Scanning dependencies of target calculator
[ 62%] Building C object CMakeFiles/calculator.dir/add.c.o
[ 75%] Building C object CMakeFiles/calculator.dir/sub.c.o
[ 87%] Building C object CMakeFiles/calculator.dir/mult.c.o
[100%] Building C object CMakeFiles/calculator.dir/calc.c.o
Linking C executable calculator
[100%] Built target calculator
```

It looks odd that all C files are compiled twice, but that's exactly what CMake is asked to do here. This makes sense, given the way the underlying makefile framework stores object files in `calc.dir` or `calculator.dir` instead of a directory that all programs share.

In summary, the commands in the CMake build description file aren't translated directly into commands in the native build system. (There's no one-to-one mapping of commands.) Instead, the native build system provides the same behavior, but using a different set of commands.

Other Interesting Features and Further Reading

CMake is certainly a complex and powerful tool, although this chapter hasn't gone into too much detail on all the features. This chapter has focused on the generation of the native build system instead of everything CMake supports. Following are a few other features that the CMake tool supports:

- String manipulation: The `string` command provides regular expression matching, substring replacement, string comparison, and conversion to upper- or lowercase.
- List manipulation: The `list` command provides support for inserting, removing, searching, and sorting values within a list.
- File manipulation: The `file` command enables a CMake build description to read or write external data files, as well as to create new directories or remove old files.
- Mathematical expressions: The `math` command provides a simple interface for computing expressions. Only the basic arithmetic, logical, and bitwise operations are supported.
- Customizing data files: The `configure_file` command generates a data file from a template by replacing all occurrences of `${VAR_NAME}` or `@VAR_NAME@` with the value of that variable.
- Testing of executable programs: The CTest module is an extension of CMake that provides automated testing of executable programs. By adding test information into the `CMakeLists.txt` file, executable programs can be sanity-tested immediately after they're built.
- Packaging and installation: The CPack module is another extension that supports the creation of software release packages, ready for installation on the target machine. Chapter 13, "Software Packaging and Installation," discusses packaging and installation in more detail.
- Platform-neutral shell commands: CMake provides built-in support for common shell script operations. In many build tools, a developer is left to deal with the difficulties of using shell commands that vary from

one machine to the next. To solve this problem CMake provides a uniform interface for invoking common shell operations. This is particularly important when compiling software for both Windows and UNIX when the shell commands are significantly different.

If you're interested in using CMake's more advanced features, you're strongly encouraged to learn more from the product's own documentation. The online wiki pages are available on the CMake web site [61], although advanced users should consider reading a book on the topic [62].