Key Concepts

3.1 Main Structures

This chapter provides an introduction to CMake's key concepts. As you start working with CMake you will run into a variety of concepts such as targets, generators, and commands. In CMake these concepts are implemented as C++ classes and are referenced in many of CMake's commands. Understanding these concepts will provide you with the working knowledge you need to create effective CMakeLists files.

Before going into detail about CMake's classes it is worth understanding their basic relationships. At the lowest level there are source files. These correspond to typical C or C++ source code files. Source files are combined into targets. A target is typically an executable or library. A directory represents a directory in the source tree and typically has a CMakeLists file and one or more targets associated with it. Every directory has a local generator that is responsible for generating the Makefiles or project files for that directory. All of the local generators share a common global generator that oversees the build process. Finally, the global generator is created and driven by the `cmake` class itself.

Figure 4 shows the basic class structure of CMake. We will now consider CMake's concepts in a bit more detail. CMake's execution begins by creating an instance of the `cmake` class and passing the command line arguments to it. This class manages the overall configuration process and holds information that is global to the build process such as the cache values. One of the first things the `cmake` class does is to create the correct global generator based on the user's selection of what generator to use (such as Visual Studio 10, Borland Makefiles, or UNIX Makefiles). At this point the `cmake` class passes control to the global generator it created by invoking the configure and generate methods.

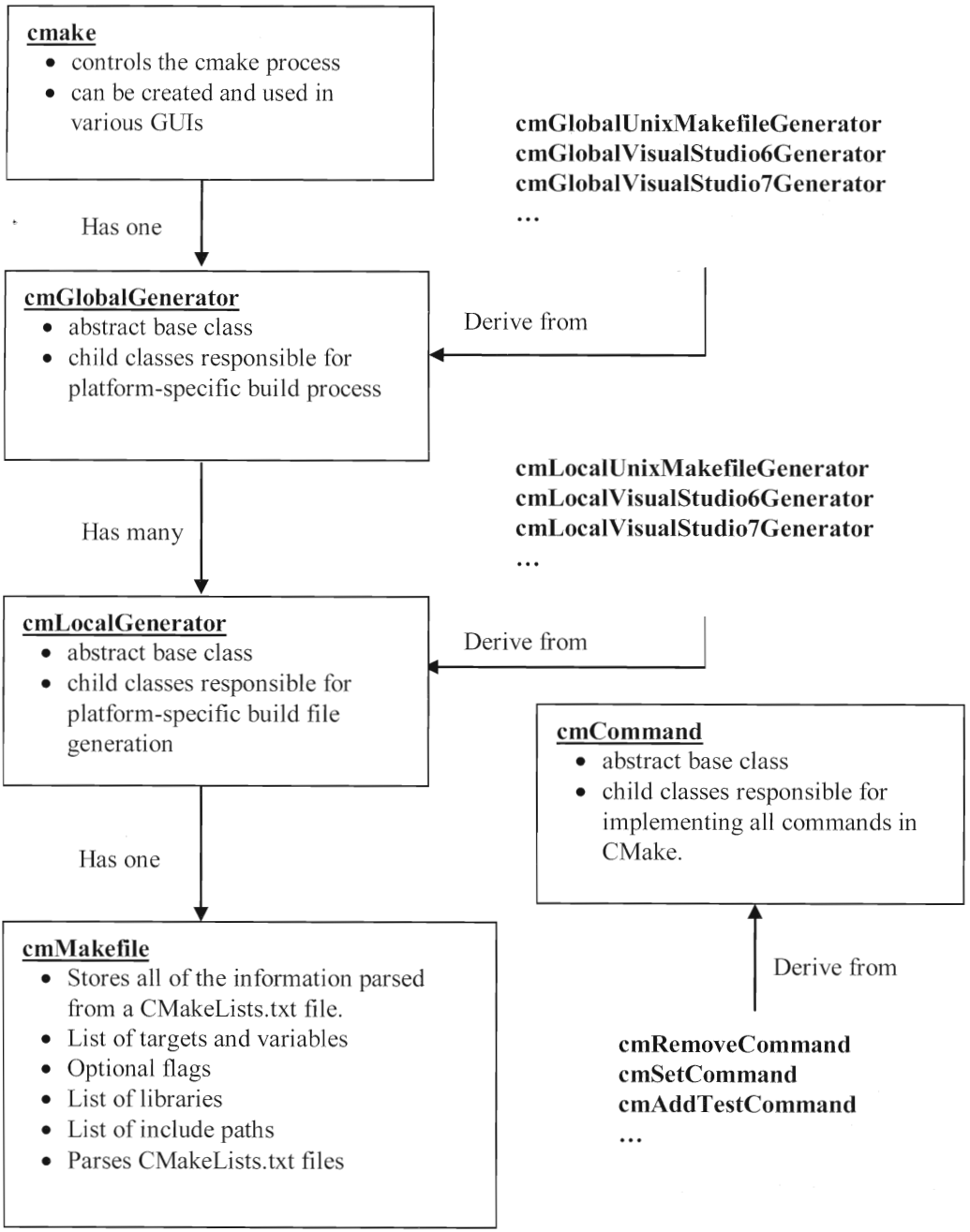

Figure 4 - CMake Internals

The global generator is responsible for managing the configuration and generation of all of the Makefiles (or project files) for a project. In practice most of the work is actually done by local generators which are created by the global generator. One local generator is created for each directory of the project that is processed. So while a project will have only one global generator it may have many local generators. For example, under Visual Studio 7 the global generator creates a solution file for the entire project while the local generators create a project file for each target in their directory.

In the case of the "Unix Makefiles" generator, the local generators create most of the Makefiles and the global generator simply orchestrates the process and creates the main top-level Makefile. Implementation details vary widely among generators. The Visual Studio 6 generators make use of .dsp and .dsw file templates and perform variable replacements on them. The generators for Visual Studio 7 and later directly generate the XML output without using any file templates. The Makefile generators including UNIX, NMake, Borland, etc use a set of rule templates and replacements to generate their Makefiles.

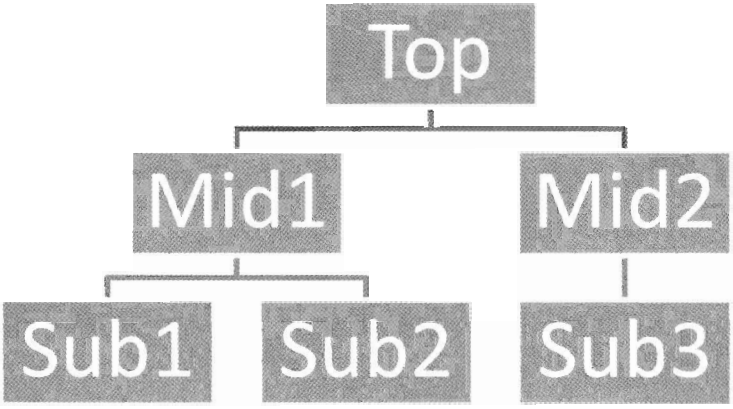

Figure 5 - Sample Directory Tree

Each local generator has an instance of the class cmMakefile, cmMakefile is where the results of parsing the CMakeLists files are stored. Specifically, for each directory in a project there will be a single cmMakefile instance which is why the cmMakefile class is often referred to as the directory. This is clearer for build systems that do not use Makefiles. That instance will hold all of the information from parsing that directory's CMakeLists file (see Figure 5). One way to think of the cmMakefile class is as a structure that starts out initialized with a few variables from its parent directory, and is then filled in as the CMakeLists file is processed. Reading in the CMakeLists file is simply a matter of CMake executing the commands it finds in the order it encounters them.

Each command in CMake is implemented as a separate C++ class, and has two main parts. The first part of a command is the InitialPass method. The InitialPass method receives the arguments and the `cmMakefile` instance for the directory currently being processed, and then

performs its operations. In the case of the `set` command, it processes its arguments and if the arguments are correct it calls a method on the `cmMakefile` to set the variable. The results of the command are always stored in the `cmMakefile` instance. Information is never stored in a command. The last part of a command is the FinalPass. The FinalPass of a command is executed after all commands (for the entire CMake project) have had their InitialPass invoked. Most commands do not have a FinalPass, but in some rare cases a command must do something with global information that may not be available during the initial pass.

Once all of the CMakeLists files have been processed the generators use the information collected into the `cmMakefile` instances to produce the appropriate files for the target build system (such as Makefiles).

3.2 Targets

Now that we have discussed the overall process of CMake, let us consider some of the key items stored in the `cmMakefile` instance. Probably the most important item is targets. Targets represent executables, libraries, and utilities built by CMake. Every `add_library`, `add_executable`, and `add_custom_target` command creates a target. For example, the following command will create a target named foo that is a static library, with `foo1.c` and `foo2.c` as source files.

```
add_library (foo STATIC foo1.c foo2.c)
```

The name foo is now available for use as a library name everywhere else in the project, and CMake will know how to expand the name into the library when needed. Libraries can be declared to be of a particular type such as `STATIC`, `SHARED`, `MODULE`, or left undeclared. `STATIC` indicates that the library must be built as a static library. Likewise `SHARED` indicates it must be built as a shared library. `MODULE` indicates that the library must be created so that it can be dynamically loaded into an executable. On many operating systems this is the same as `SHARED`, but on other systems such as Mac OS X it is different. If none of these options are specified this indicates that the library could be built as either shared or static. In that case CMake uses the setting of the variable `BUILD_SHARED_LIBS` to determine if the library should be `SHARED` or `STATIC`. If it is not set, then CMake defaults to building static libraries.

Likewise executables have some options. By default an executable will be a traditional console application that has a `main (int argc, const char*argv[])`. If `WIN32` is specified after the executable name then the executable will be compiled as a MS Windows executable and the operating system will call WinMain instead of main at startup. `WIN32` has no effect on non-Windows systems.

In addition to storing their type, targets also keep track of general properties. These properties can be set and retrieved using the `set_target_properties` and `get_target_property`

commands, or the more general `set_property` and `get_property` commands. The most commonly used property is `LINK_FLAGS`, which is used to specify link flags for a specific target. Targets store a list of libraries that they link against which are set using the `target_link_libraries` command. Names passed into this command can be libraries, full paths to libraries, or the name of a library from an `add_library` command. They also store the link directories to use when linking, the install location for the target, and custom commands to execute after linking.

For each library CMake creates, it keeps track of all the libraries on which that library depends. Since static libraries do not link to the libraries on which they depend, it is important for CMake to keep track of the libraries so they can be specified on the link line of the executable being created. For example,

```
add_library (foo foo.cxx)
target_link_libraries (foo bar)

add_executable (foobar foobar.cxx)
target_link_libraries (foobar foo)
```

This will link the libraries foo and bar into the executable foobar even, although only foo was explicitly linked into foobar. With shared or DLL builds this linking is not always needed, but the extra linkage is harmless. For static builds this is required. Since the foo library uses symbols from the bar library, foobar will most likely also need bar since it uses foo.

3.3 Source Files

The source file structure is in many ways similar to a target. It stores the filename, extension, and a number of general properties related to a source file. Like targets you can set and get properties using `set_source_files_properties` and `get_source_file_property`, or the more generic versions. The most common properties include:

COMPILE_FLAGS

Compile flags specific to this source file. These can include source specific –D and –I flags.

GENERATED

The `GENERATED` property indicates that the source file is generated as part of the build process. In this case CMake will treat it differently for computation of dependencies because the source file may not exist when CMake is first run.

OBJECT_DEPENDS

Adds additional files on which this source file should depend. CMake automatically performs dependency analysis to determine the usual C, C++ and Fortran dependencies. This parameter is used rarely in cases where there is an unconventional dependency or the source files do not exist at dependency analysis time.

ABSTRACT

WRAP_EXCLUDE

CMake doesn't directly use these properties. Some loaded commands and extensions to CMake look at these properties to determine how and when to wrap a C++ class into languages such as Tcl, Python, etc.

3.4 Directories, Generators, Tests, and Properties

In addition to targets and source files you may find yourself occasionally working with other classes such as directories, generators, and tests. Normally such interactions take the shape of setting or getting properties from these objects. All of these classes have properties associated with them, as do source files and targets. A property is a key-value pair attached to a specific object such as a target. The most generic way to access properties is through the `set_property` and `get_property` commands. These commands allow you to set or get a property from any class in CMake that has properties. Some of the properties for targets and source files have already been covered. Some useful properties for a directory include:

ADDITIONAL_MAKE_CLEAN_FILES

This property specifies a list of additional files that will be cleaned as a part of the "make clean" stage. By default CMake will clean up any generated files that it knows about, but your build process may use other tools that leave files behind. This property can be set to a list of those files so that they also will be properly cleaned up.

EXCLUDE_FROM_ALL

This property indicates if all the targets in this directory and all sub directories should be excluded from the default build target. If it is not, then with a Makefile for example typing make will cause these targets to be built as well. The same concept applies to the default build of other generators.

LISTFILE_STACK

This property is mainly useful when trying to debug errors in your CMake scripts. It returns a list of what list files are currently being processed, in order. So if one CMakeLists file does an `include` command then that is effectively pushing the included CMakeLists file onto the stack.

A full list of properties supported in CMake can be obtained by running `cmake` with the `-help-property-list` option. The generators and directories are automatically created for you as CMake processes your source tree.

3.5 Variables and Cache Entries

CMakeLists files use variables much like any programming language. Variables are used to store values for later use, and can be a single value such as "ON" or "OFF", or they can represent a list such as (`/usr/include /home/foo/include /usr/local/include`). A number of useful variables are automatically defined by CMake and are discussed in Appendix A - Variables.

Variables in CMake are referenced using a `${VARIABLE}` notation, and they are defined in the order of execution of the `set` commands. Consider the following example:

```
# FOO is undefined

set (FOO 1)
# FOO is now set to 1

set (FOO 0)
# FOO is now set to 0
```

This may seem straightforward, but consider the following example:

```
set (FOO 1)

if (${FOO} LESS 2)
  set (FOO 2)
else (${FOO} LESS 2)
  set (FOO 3)
endif (${FOO} LESS 2)
```

Clearly the `if` statement is true, which means that the body of the `if` statement will be executed. That will set the variable `FOO` to 2, and so when the `else` statement is encountered `FOO` will have a value of 2. Normally in CMake the new value of `FOO` would be used, but the `else` statement is a rare exception to the rule and always refers back to the value of the variable when the `if` statement was executed. So in this case the body of the `else` clause will not be executed. To further understand the scope of variables consider this example:

```
set (foo 1)

# process the dir1  subdirectory
add_subdirectory (dir1)

# include and process the commands in file1.cmake
include (file1.cmake)

set (bar 2)
# process the dir2 subdirectory
add_subdirectory (dir2)

# include and process the commands in file2.cmake
include (file2.cmake)
```

In this example because the variable `foo` is defined at the beginning, it will be defined while processing both dir1 and dir2. In contrast `bar` will only be defined when processing dir2. Likewise `foo` will be defined when processing both file1.cmake and file2.cmake, whereas `bar` will only be defined while processing file2.cmake.

Variables in CMake have a scope that is a little different from most languages. When you set a variable it is visible to the current CMakeLists file or function, as well as any subdirectory's CMakeLists files, any functions or macros that are invoked, and any files that are included using the `INCLUDE` command. When a new subdirectory is processed (or a function called) a new variable scope is created and initialized with the current value of all variables in the calling scope. Any new variables created in the child scope, or changes made to existing variables, will not impact the parent scope. Consider the following example:

```
function (foo)
  message (${test}) # test is 1 here
  set (test 2)
  message (${test}) # test is 2 here, but only in this scope
endfunction()

set (test 1)
foo()
message (${test}) # test will still be 1 here
```

In some cases you might want a function or subdirectory to set a variable in its parent's scope. This is one way for CMake to return a value from a function, and it can be done by using the `PARENT_SCOPE` option with the `set` command. We can modify the prior example so that the function `foo` changes the value of test in its parent's scope as follows:

```
function (foo)
  message (${test}) # test is 1 here
  set (test 2 PARENT_SCOPE)
  message (${test}) # test still 1 in this scope
endfunction()

set (test 1)
foo()
message (${test}) # test will now be 2 here
```

Variables can also represent a list of values. In these cases when the variable is expanded it will be expanded into multiple values. Consider the following example:

```
# set a list of items
set (items_to_buy apple orange pear beer)

# loop over the items
foreach (item ${items_to_buy})
  message ( "Don't forget to buy one ${item}" )
endforeach ()
```

In some cases you might want to allow the user building your project to set a variable from the CMake user interface. In that case the variable must be a cache entry. Whenever CMake is run it produces a cache file in the directory where the binary files are to be written. The values of this cache file are displayed by the CMake user interface. There are a few purposes of this cache. The first is to store the user's selections and choices, so that if they should run CMake again they will not need to reenter that information. For example, the `option` command creates a Boolean variable and stores it in the cache.

```
option (USE_JPEG "Do you want to use the jpeg library")
```

The above line would create a variable called `USE_JPEG` and put it into the cache. That way the user can set that variable from the user interface and its value will remain in case the user should run CMake again in the future. To create a variable in the cache you can use commands like `option`, `find_file`, or you can use the standard `set` command with the `CACHE` option.

```
set (USE_JPEG ON CACHE BOOL "include jpeg support?")
```

When you use the cache option you must also provide the type of the variable and a documentation string. The type of the variable is used by the GUI to control how that variable

is set and displayed. Variable types include BOOL, PATH, FILEPATH, and STRING. The documentation string is used by the GUI to provide online help.

The other purpose of the cache is to store key variables that are expensive to determine. These variables may not be visible or adjustable by the user. Typically these values are system dependent variables such as CMAKE_WORDS_BIGENDIAN, which require CMake to compile and run a program to determine their value. Once these values have been determined, they are stored in the cache to avoid having to recompute them every time CMake is run. Generally CMake tries to limit these variables to properties that should never change (such as the byte order of the machine you are on). If you significantly change your computer, either by changing the operating system, or switching to a different compiler, you will need to delete the cache file (and probably all of your binary tree's object files, libraries, and executables).

Variables that are in the cache also have a property indicating if they are advanced or not. By default when a CMake GUI is run (such as ccmake or cmake-gui) the advanced cache entries are not displayed. This is so that the user can focus on the cache entries that they should consider changing. The advanced cache entries are other options that the user can modify, but typically will not. It is not unusual for a large software project to have fifty or more options, and the advanced property lets a software project divide them into key options for most users and advanced options for advanced users. Depending on the project there may not be any non-advanced cache entries. To make a cache entry advanced the mark_as_advanced command is used with the name of the variable (a.k.a. cache entry) to make advanced.

In some cases you might want to restrict a cache entry to a limited set of predefined options. You can do this by setting the STRINGS property on the cache entry. The following CMakeLists code illustrates this by creating a property named CRYPTOBACKEND as usual, and then setting the STRINGS property on it to a set of three options.

```
set (CRYPTOBACKEND "OpenSSL" CACHE STRING
      "Select a cryptography backend")
set_property (CACHE CRYPTOBACKEND PROPERTY STRINGS
              "OpenSSL" "LibTomCrypt" "LibDES")
```

When cmake-gui is run and the user selects the CRYPTOBACKEND cache entry, they will be presented with a pulldown to select which option they want, as shown in Figure 6.

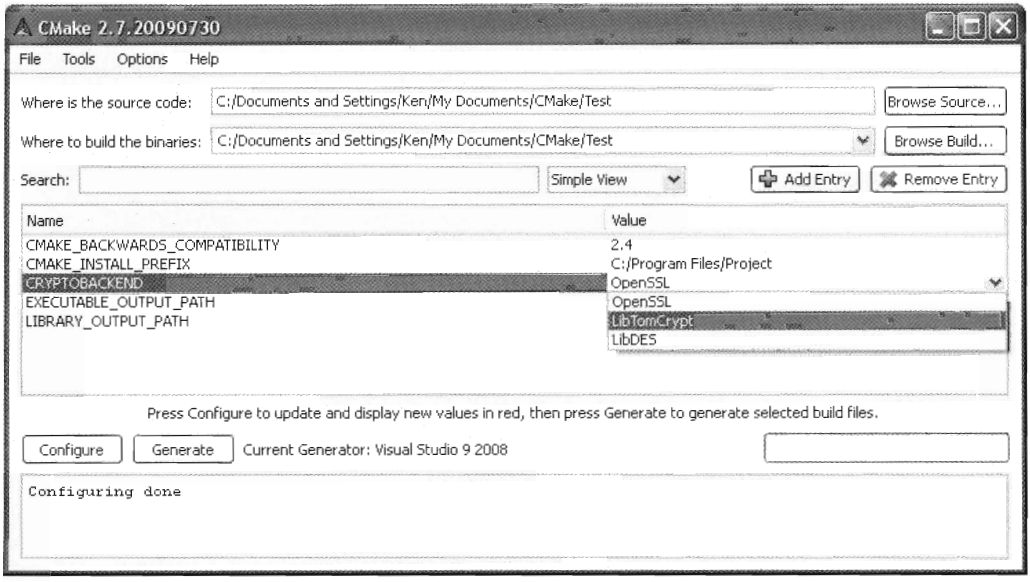

Figure 6 – Cache Value Options in cmake-gui

A few final points should be made concerning variables and their interaction with the cache. If a variable is in the cache, it can still be overridden in a CMakeLists file using the `set` command without the `CACHE` option. Cache values are checked only if the variable is not found in the current `cmMakefile` instance before CMakeLists file processing begins. The `set` command will set the variable for processing the current CMakeLists file (and subdirectories as usual) without changing the value in the cache.

```
# assume that FOO is set to ON in the cache

set (FOO OFF)
# sets foo to OFF for processing this CMakeLists file
# and subdirectories; the value in the cache stays ON
```

Once a variable is in the cache, its "cache" value cannot normally be modified from a CMakeLists file. The reasoning behind this is that once CMake has put the variable into the cache with its initial value, the user may then modify that value from the GUI. If the next invocation of CMake overwrote their change back to the `set` value, the user would never be able to make a change that CMake wouldn't overwrite. So a `set (FOO ON CACHE BOOL "doc")` command will typically only do something when the cache doesn't have the variable in it. Once the variable is in the cache, that command will have no effect.

In the rare event that you really want to change a cached variable's value you can use the FORCE option in combination with the CACHE option to the set command. The FORCE option will cause the set command to override and change the cache value of a variable.

3.6 Build Configurations

Build configurations allow a project to be built in different ways for debug, optimized, or any other special set of flags. CMake supports, by default, Debug, Release, MinSizeRel, and RelWithDebInfo configurations. Debug has the basic debug flags turned on. Release has the basic optimizations turned on. MinSizeRel has the flags that produce the smallest object code, but not necessarily the fastest code. RelWithDebInfo builds an optimized build with debug information as well.

CMake handles the configurations in slightly different ways depending on what generator is being used. The conventions of the native build system are followed when possible. This means that configurations impact the build in different ways when using Makefiles versus using Visual Studio project files.

The Visual Studio IDE supports the notion of Build Configurations. A default project in Visual Studio usually has Debug and Release configurations. From the IDE you can select build Debug, and the files will be built with Debug flags. The IDE puts all of the binary files into directories with the name of the active configuration. This brings about an extra complexity for projects that build programs that need to be run as part of the build process from custom commands. See the CMAKE_CFG_INTDIR variable and the custom commands section for more information about how to handle this issue. The variable CMAKE_CONFIGURATION_TYPES is used to tell CMake which configurations to put in the workspace.

With Makefile based generators, only one configuration can be active at the time CMake is run, and it is specified by the CMAKE_BUILD_TYPE variable. If the variable is empty then no flags are added to the build. If the variable is set to the name of a configuration, then the appropriate variables and rules (such as CMAKE_CXX_FLAGS_<ConfigName>) are added to the compile lines. Makefiles do not use special configuration subdirectories for object files. To build both debug and release trees, the user is expected to create multiple build directories using the out of source build feature of CMake, and to set the CMAKE_BUILD_TYPE to the desired selection for each build. For example,

```
# With source code in the directory MyProject
# to build MyProject-debug create that directory, cd into it and
(ccmake ../MyProject -DCMAKE_BUILD_TYPE:STRING=Debug)
# the same idea is used for the release tree MyProject-release
(ccmake ../MyProject -DCMAKE_BUILD_TYPE:STRING=Release)
```