Getting Started

2.1 Getting and Installing CMake on Your Computer

Before using CMake you will need to install or build the CMake binaries on your system. On many systems you may find that CMake is already installed, or is available for install with the standard package manager tool for the system. Cygwin, Debian, FreeBSD, Mac OS X Fink, and many others all have CMake distributions. If your system does not have a CMake package, you can find CMake precompiled for most common architectures at www.cmake.org. If you do not find binaries for your system precompiled, then you can build CMake from source. To build CMake you will need a modern C++ compiler.

UNIX and Mac Binary Installations

If your system provides CMake as one of its standard packages, follow your system's package installation instructions. If your system does not have CMake, or has an out of date version of CMake, you can download precompiled binaries from www.cmake.org. The binaries from www.cmake.org come in the form of a compressed tar file. The tar file contains a README file and an enclosed tar file. The README file contains a manifest of the files contained in the enclosed tar file, and some instructions. To install, simply extract the enclosed tar file into a destination directory (typically `/usr/local`). However, it can be any directory, and does not require root privileges for installation.

Windows Binary Installation

For Windows CMake has a NullSoft install file available for download from www.cmake.org. To install this file, simply run the executable on the windows machine on which you want to install CMake. You will be able to run CMake from the Start Menu after it is installed.

2.2 Building CMake Yourself

If binaries are not available for your system, or if binaries are not available for the version of CMake you wish to use, you can build CMake from the source code. You can obtain the CMake source code by following the instructions at www.cmake.org. Once you have the source code it can be built in two different ways. If you have a version of CMake on your system you can use it to build other versions of CMake. Generally the current development version of CMake can always be built from the previous release of CMake. This is how new versions of CMake are built on most Windows systems.

The second way to build CMake is by running its bootstrap build script. To do this you change directory into your CMake source directory and type

```
./bootstrap
make
make install
```

The make install step is optional since CMake can run directly from the build directory if desired. On UNIX, if you are not using the GNU C++ compiler, you need to tell the bootstrap script which compiler you want to use. This is done by setting the environment variable `CXX` before running bootstrap. If you need to use any special flags with your compiler, set the `CXXFLAGS` environment variable. For example, on the SGI with the 7.3X compiler, you would build CMake like this:

```
cd CMake
(setenv CXX CC; setenv CXXFLAGS "-LANG:std"; ./bootstrap)
make
make install
```

2.3 Basic CMake Usage and Syntax

Using CMake is simple. The build process is controlled by creating one or more CMakeLists files (actually CMakeLists.txt but this guide will leave off the extension in most cases) in each of the directories that make up a project. The CMakeLists files should contain the project description in CMake's simple language. The language is expressed as a series of commands. Each command is evaluated in the order that it appears in the CMakeLists file. The commands have the form

```
command (args...)
```

where `command` is the name of the command, and `args` is a white-space separated list of arguments. (Arguments with embedded white-space should be double quoted.) CMake is case insensitive to command names as of version 2.2. So where you see `command` you could use `COMMAND` or `Command` instead. Older versions of CMake only accepted uppercase commands.

CMake supports simple variables that can be either strings or lists of strings. Variables are referenced using a `${VAR}` syntax. Multiple arguments can be grouped together into a list using the `set` command. All other commands expand the lists as if they had been passed into the command with white-space separation. For example, `set(Foo a b c)` will result in setting the variable `Foo` to `a b c`, and if `Foo` is passed into another command `command(${Foo})` it would be equivalent to `command(a b c)`. If you want to pass a list of arguments to a command as if it were a single argument simply double quote it. For example `command("${Foo}")` would be invoked passing only one argument equivalent to `command( "a b c" )`.

System environment variables and Windows registry values can be accessed directly in CMake. To access system environment variables the syntax `$ENV{VAR}` is used. CMake can also reference registry entries in many commands using a syntax of the form `[HKEY_CURRENT_USER\\Software\\path1\\path2;key]`, where the paths are built from the registry tree and key.

2.4 Hello World for CMake

For starters let us consider the simplest possible CMakeLists file. To compile an executable from one source file the CMakeLists file would contain two lines:

```
project (Hello)
add_executable (Hello Hello.c)
```

To build the Hello executable you follow the process described in Running CMake (See section 2.5) to generate the Makefiles or Microsoft project files. The `project` command indicates what the name of the resulting workspace should be and the `add_executable` command adds an executable target to the build process. That's all there is to it for this simple example. If your project requires a few files it is also quite easy, just modify the `add_executable` line as shown below.

```
add_executable (Hello Hello.c File2.c File3.c File4.c)
```

`add_executable` is just one of many commands available in CMake. Consider the more complicated example below.

```
cmake_minimum_required (2.6)
project (HELLO)

set (HELLO_SRCS Hello.c File2.c File3.c)

if (WIN32)
  set(HELLO_SRCS ${HELLO_SRCS} WinSupport.c)
else ()
  set(HELLO_SRCS ${HELLO_SRCS} UnixSupport.c)
endif ()

add_executable (Hello ${HELLO_SRCS})

# look for the Tcl library
find_library (TCL_LIBRARY
  NAMES tcl tcl84 tcl83 tcl82 tcl80
  PATHS /usr/lib /usr/local/lib
  )

if (TCL_LIBRARY)
  target_link_library (Hello ${TCL_LIBRARY})
endif ()
```

In this example the `set` command is used to group together source files into a list. The `if` command is used to add either WinSupport.c or UnixSupport.c to this list based on whether or not CMake is running on Windows. Finally, the `add_executable` command is used to build the executable with the files listed in the variable `HELLO_SRCS`. The `find_library` command looks for the Tcl library under a few different names and in a few different paths. An `if` command checks if the `TCL_LIBRARY` was found and if so adds it to the link line for the Hello executable target. Note the use of the `#` character to denote a comment line. All characters from the `#` to the end of the line are considered to be part of the comment.

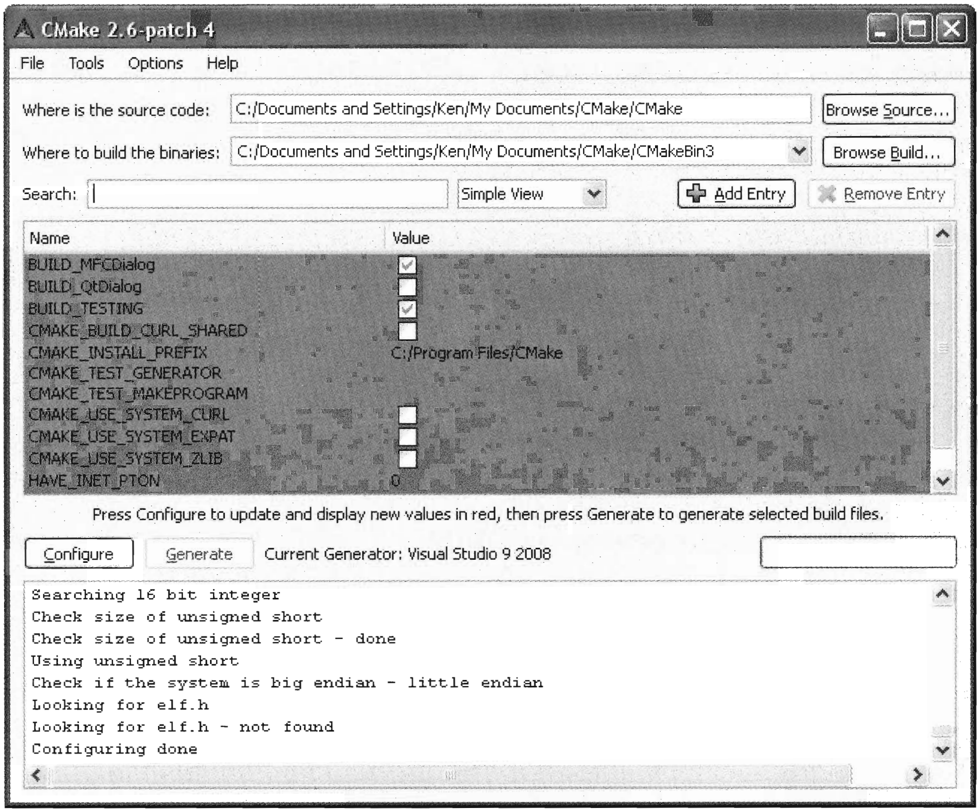
2.5 How to Run CMake?

Once CMake has been installed on your system, using it to build a project is easy. There are two main directories CMake uses when building a project: the source directory and the binary directory. The source directory is where the source code for your project is located. This is also where the CMakeLists files will be found. The binary directory is where you want CMake to put the resulting object files, libraries, and executables. Typically CMake will not write any files to the source directory, only the binary directory. If you want to you can set the source and binary directories to be the same. This is known as an in-source build, in contrast to an out-of-source build where they are different.

CMake supports both in-source and out-of-source builds on all operating systems. This means that you can configure your build to be completely outside of the source code tree which makes it very easy to remove all of the files generated by a build. Having the build tree differ from the source tree also makes it easy to support having multiple builds of a single source tree. This is useful when you want to have multiple builds with different options but just one copy of the source code. Now let us consider the specifics of running CMake using its Qt based GUI and command line interfaces.

Running CMake's Qt Interface

CMake includes a Qt based user interface developed by Clinton Stimpson that can be used on most platforms, including UNIX, Mac OS X, and Windows. This interface is included in the CMake source code, but you will need an installation of Qt on your system in order to build it.



Where is the source code: C:/Documents and Settings/Ken/My Documents/CMake/CMake

Where to build the binaries: C:/Documents and Settings/Ken/My Documents/CMake/CMakeBin3

Search: Simple View

Name	Value
BUILD_MFCDialog	✓
BUILD_QtDialog	☐
BUILD_TESTING	✓
CMAKE_BUILD_CURL_SHARED	☐
CMAKE_INSTALL_PREFIX	C:/Program Files/CMake
CMAKE_TEST_GENERATOR	
CMAKE_TEST_MAKEPROGRAM	
CMAKE_USE_SYSTEM_CURL	☐
CMAKE_USE_SYSTEM_EXPAT	☐
CMAKE_USE_SYSTEM_ZLIB	☐
HAVE_INET_PTON	0

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 9 2008

```
Searching 16 bit integer
Check size of unsigned short
Check size of unsigned short - done
Using unsigned short
Check if the system is big endian - little endian
Looking for elf.h
Looking for elf.h - not found
Configuring done
```

Figure 1 – Qt based CMake GUI

On Windows the executable is named `cmake-gui.exe` and should be in your Start menu under Program Files. There may also be a shortcut on your desktop, or if you built CMake from source, it will be in the build directory. For UNIX and Mac users the executable is named cmake-gui and it can be found where you installed the CMake executables.A GUI will appear similar to what is shown in Figure 1. The top two entries are the source code and binary directories. They allow you to specify where the source code is for what you want to compile and where the resulting binaries should be placed. You should set these two values first. If the binary directory you specify does not exist, it will be created for you. If the binary directory has been configured by CMake before then it will automatically set the source tree.

The middle area is where you can specify different options for the build process. More obscure variables may be hidden, but can be seen if you select "Advanced View" from the view pulldown. You can search for values in the middle area by typing all or part of the name into the Search box. This can be handy for finding specific settings or options in a large project. The bottom area of the window includes the Configure and Generate buttons as well as a progress bar and scrollable output window.

Once you have specified the source code and binary directories you should click the Configure button. This will cause CMake to read in the CMakeLists files from the source code directory and then update the cache area to display any new options for the project. If you are running cmake-gui for the first time on this binary directory it will prompt you to determine what generator you wish to use, as shown in Figure 2. This dialog also presents options for customizing and tweaking the compilers you wish to use for this build.

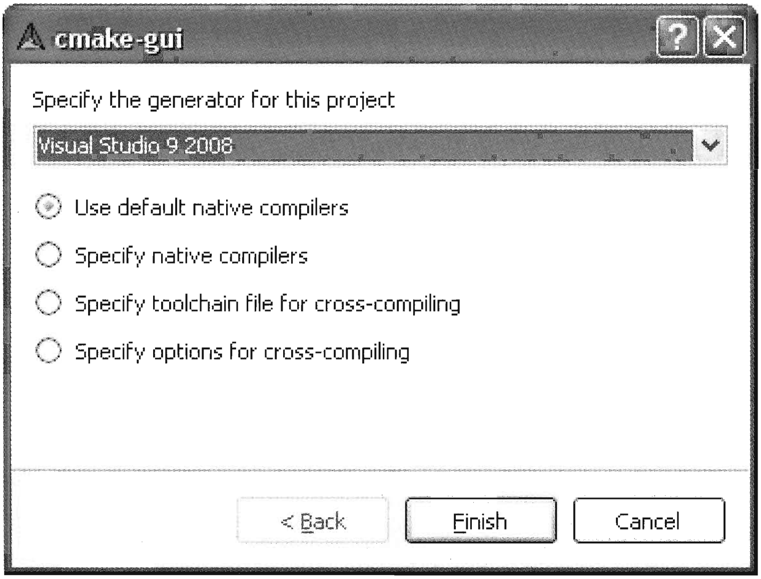

cmake-gui
Specify the generator for this project
Visual Studio 9 2008
Use default native compilers
Specify native compilers
Specify toolchain file for cross-compiling
Specify options for cross-compiling
< Back
Finish
Cancel

Figure 2 – Selecting a Generator

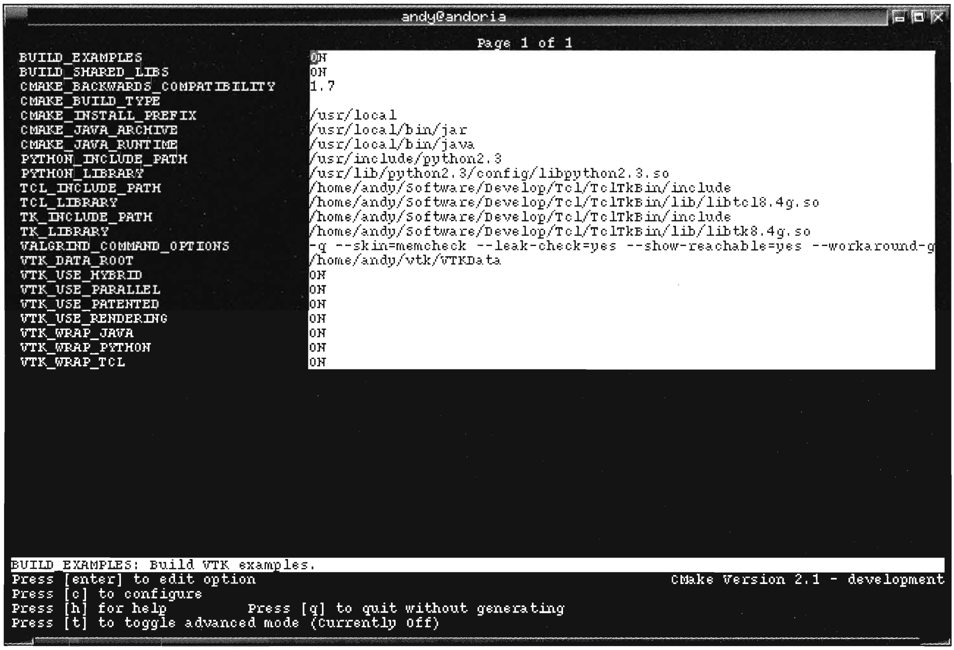
After the first configure you can adjust your cache settings if desired and click the Configure button again. New values that were created by the configure process will be colored red. To be sure you have seen all possible values you should click Configure until no values are red and you are happy with all the settings. Once you are done configuring, click the Generate button, this will produce the appropriate files.

It is important that you make sure that your environment is suitable for running cmake-gui. If you are using an IDE such as Visual Studio then your environment will be setup correctly for you. If you are using NMake or MinGW then you need to make sure that the compiler can run from your environment. You can either directly set the required environment variables for your compiler or use a shell in which they are already set. For example, Microsoft Visual Studio has an option on the start menu for creating a Visual Studio Command Prompt. This opens up a command prompt window that has its environment already setup for Visual Studio. You should run cmake-gui from this command prompt if you want to use NMake Makefiles. The same approach applies to MinGW, you should run cmake-gui from a MinGW shell that has a working compiler in its path.

When cmake-gui finishes it will have generated the build files in the binary directory you specified. If Visual Studio was selected as the generator, a MSVC workspace (or solution) file is created. This file's name is based on the name of the project you specified in the `PROJECT` command at the beginning of your CMakeLists file. For many other generator types, Makefiles are generated. The next step in this process is to open the workspace with MSVC. Once open, the project can be built in the normal manner of Microsoft Visual C++. The `ALL_BUILD` target can be used to build all of the libraries and executables in the package. If you are using a Makefile build type, then you would build by running make or nmake on the resulting Makefiles.

Running the ccmake Curses Interface

On most UNIX platforms, if the curses library is supported, CMake provides an executable called ccmake. This interface is a terminal-based text application that is very similar to the Qt based GUI. To run ccmake, change directory (cd) to the directory where you want the binaries to be placed. This can be the same directory as the source code for what we call in-source builds or it can be a new directory you create. Then run ccmake with the path to the source directory on the command line. For in-source builds use "." for the source directory. This will start the text interface as shown in Figure 3 (in this case the cache variables are from VTK and most are set automatically).



```
                                                  Page 1 of 1
 BUILD_EXAMPLES                   ON
 BUILD_SHARED_LIBS                ON
 CMAKE_BACKWARDS_COMPATIBILITY    1.7
 CMAKE_BUILD_TYPE
 CMAKE_INSTALL_PREFIX             /usr/local
 CMAKE_JAVA_ARCHIVE               /usr/local/bin/jar
 CMAKE_JAVA_RUNTIME               /usr/local/bin/java
 PYTHON_INCLUDE_PATH              /usr/include/python2.3
 PYTHON_LIBRARY                   /usr/lib/python2.3/config/libpython2.3.so
 TCL_INCLUDE_PATH                 /home/andy/Software/Develop/Tcl/TclTkBin/include
 TCL_LIBRARY                      /home/andy/Software/Develop/Tcl/TclTkBin/lib/libtcl8.4g.so
 TK_INCLUDE_PATH                  /home/andy/Software/Develop/Tcl/TclTkBin/include
 TK_LIBRARY                       /home/andy/Software/Develop/Tcl/TclTkBin/lib/libtk8.4g.so
 VALGRIND_COMMAND_OPTIONS         -q --skin=memcheck --leak-check=yes --show-reachable=yes --workaround-g
 VTK_DATA_ROOT                    /home/andy/vtk/VTKData
 VTK_USE_HYBRID                   ON
 VTK_USE_PARALLEL                 ON
 VTK_USE_PATENTED                 ON
 VTK_USE_RENDERING                ON
 VTK_WRAP_JAVA                    ON
 VTK_WRAP_PYTHON                  ON
 VTK_WRAP_TCL                     ON

BUILD_EXAMPLES: Build VTK examples.
Press [enter] to edit option                                      CMake Version 2.1 - development
Press [c] to configure
Press [h] for help          Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Figure 3 - ccmake running on UNIX

Brief instructions are displayed in the bottom of the window. If you hit the "c" key, it will configure the project. You should always configure after changing values in the cache. To change values, use the arrow keys to select cache entries, and then the enter key to edit them. Boolean values will toggle with the enter key. Once you have set all the values as you like, you can hit the "g" key to generate the Makefiles and exit. You can also hit "h" for help, "q" to quit, and "t" to toggle the viewing of advanced cache entries. Two examples of CMake usage on the UNIX platform follow for a hello world project called Hello. In the first example, an in-source build is performed.

```
cd Hello
ccmake .
make
```

In the second example, an out-of-source build is performed.

```
mkdir Hello-Linux
cd Hello-Linux
ccmake ../Hello
make
```

Running CMake from the Command Line

From the command line, CMake can be run as an interactive question and answer session or as a non-interactive program. To run in interactive mode, just pass the "-i" option to CMake. This will cause CMake to ask you for a value for each entry in the cache file for the project. CMake will provide reasonable defaults, just like it does in the GUI and curses based interfaces. The process stops when there are no longer any more questions to ask. An example of using the interactive mode of CMake is provided below.

```
$ cmake -i -G "NMake Makefiles" ../CMake
Would you like to see advanced options? [No]:
Please wait while cmake processes CMakeLists.txt files....

Variable Name: BUILD_TESTING
Description: Build the testing tree.
Current Value: ON
New Value (Enter to keep current value):

Variable Name: CMAKE_INSTALL_PREFIX
Description: Install path prefix, prepended onto install
directories.
Current Value: C:/Program Files/CMake
New Value (Enter to keep current value):

Please wait while cmake processes CMakeLists.txt files....

CMake complete, run make to build project.
```

Using CMake to build a project in non-interactive mode is a simple process if the project has few or no options. For larger projects like VTK, using `ccmake`, `cmake -i`, or `cmake-gui` is recommended. To build a project with a non-interactive CMake, first change directory to where you want the binaries to be placed. For an in-source build you then run `cmake .` and pass in any options using the `-D` flag. For out-of-source builds the process is the same except you run `cmake` and also provide the path to the source code as its argument. Then type `make` and your project should compile. Some projects will have install targets as well, you can type `make install` to install them.

Specifying the Compiler to CMake

On some systems you may have more than one compiler to choose from or your compiler may be in a non-standard place. In these cases you will need to specify to CMake where your desired compiler is located. There are three ways to specify this; the generator can specify the compiler, an environment variable can be set, or a cache entry can be set. Some generators are tied to a specific compiler, for example the Visual Studio 6 generator always uses the

Microsoft Visual Studio 6 compiler. For Makefile based generators CMake will try a list of usual compilers until it finds a working compiler. The list can be found in the files:

```
Modules/CMakeDeterminCCompiler.cmake and
Modules/CMakeDetermineCXXCompiler.cmake
```

The lists can be preempted with environment variables that can be set before CMake is run. The `CC` environment variable specifies the C compiler while `CXX` specifies the C++ compiler. You can specify the compilers directly on the command line by using –`DCMAKE_CXX_COMPILER=cl` for example. If those are not set, CMake will try the following list of compilers:

```
c++ g++ CC aCC cl bcc xlC.
```

Once CMake has been run and picked a compiler, you can change the selection by changing the cache entries `CMAKE_CXX_COMPILER` and `CMAKE_C_COMPILER`, although this is not recommended. The problem with doing this is that the project you are configuring may have already run some tests on the compiler to determine what it supports. Changing the compiler does not normally cause these tests to be rerun which can lead to incorrect results. If you must change the compiler, start over with an empty binary directory. The flags for the compiler and the linker can also be changed by setting environment variables. Setting `LDFLAGS` will initialize the cache values for link flags, while `CXXFLAGS` and `CFLAGS` will initialize `CMAKE_CXX_FLAGS` and `CMAKE_C_FLAGS` respectively.

Dependency Analysis

CMake has powerful built-in dependency analysis capabilities for C and C++ source code files. CMake also has limited support for Fortran and Java dependencies. Since Integrated Development Environments (IDEs) support and maintain dependency information, CMake skips this step for those build systems. However, Makefiles with a make program do not know how to automatically compute and keep dependency information up-to-date. For these builds, CMake automatically computes dependency information for C, C++ and Fortran files. Both the generation and maintenance of these dependencies are automatically done by CMake. Once a project is initially configured by CMake, users only need to run make, and CMake does the rest of the work. CMake's dependencies fully support parallel builds for multiprocessor systems.

Although users do not need to know how CMake does this work, it may be useful to look at the dependency information files for a project. This information for each target is stored in four files called `depend.make`, `flags.make`, `build.make`, and `DependInfo.cmake`. `depend.make` stores the depend information for all the object files in the directory. `flags.make` contains the compile flags used for the source files of this target. If they change then the files will be recompiled. `DependInfo.cmake` is used to keep the dependency information up-to-date and contains information about what files are part of the project and what languages they are in. Finally, the rules for building the dependencies are stored in `build.make`. If a dependency is out of date then all of the dependencies for that target will be recomputed, keeping the dependency information current.

2.6 Editing CMakeLists Files

CMakeLists files can be edited in almost any text editor. Some editors, such as Notepad++, come with CMake syntax highlighting and indentation support built in. For editors such as Emacs or Vim CMake includes indentation and syntax highlighting modes. These can be found in the Docs directory of the source distribution, or downloaded from the CMake web site. The file `cmake-mode.el` is the Emacs mode, and `cmake-indent.vim` and `cmake-syntax.vim` are used by Vim. Within Visual Studio the CMakeLists files are listed as part of the project and you can edit them simply by double clicking on them. Within any of the supported generators (Makefiles, Visual Studio, etc) if you edit a CMakeLists file and rebuild, there are rules that will automatically invoke CMake to update the generated files (e.g. Makefiles or project files) as required. This helps to assure that your generated files are always in sync with your CMakeLists files.

Since CMake computes and maintains dependency information, the CMake executables must always be available (though they don't have to be in your PATH) when make or an IDE is being run on CMake generated files. This means that if a CMake input file changes on disk, your build system will automatically re-run CMake and produce up-to-date build files. For this reason you generally should not generate Makefiles or projects with CMake and move them to another machine that does not have CMake installed.

2.7 Setting Initial Values for CMake

While CMake works well in an interactive mode, sometimes you will need to setup cache entries without running a GUI. This is common when setting up nightly dashboards or if you will be creating many build trees with the same cache values. In these cases the CMake cache can be initialized in two different ways. The first way is to pass the cache values on the CMake command line using `-DCACHE_VAR:TYPE=VALUE` arguments. For example, consider the following nightly dashboard script for a UNIX machine:

```
#!/bin/tcsh

cd ${HOME}

# wipe out the old binary tree and then create it again
rm -rf Foo-Linux
mkdir Foo-Linux
```

```
cd Foo-Linux

# run cmake to setup the cache
cmake -DBUILD_TESTING:BOOL=ON <etc...> ../Foo

# generate the dashboard
ctest -D Nightly
```

The same idea can be used with a batch file on Windows. The second way is to create a file to be loaded using CMake's `-C` option. In this case instead of setting up the cache with `-D` options it is done though a file that is parsed by CMake. The syntax for this file is standard CMakeLists syntax and it is typically just a series of `set` commands such as:

```
#Build the vtkHybrid kit.
set (VTK_USE_HYBRID ON CACHE BOOL "doc string")
```

In some cases there might be an existing cache and you want to force the cache values to be set a certain way. For example say you want to turn Hybrid on even if the user has previously run CMake and turned it off. Then you can do:

```
#Build the vtkHybrid kit always.
set (VTK_USE_HYBRID ON CACHE BOOL "doc" FORCE)
```

Another option is that you want to set and then hide options so the user will not be tempted to adjust them later on. This can be done using the following commands:

```
#Build the vtkHybrid kit always and don't distract
#the user by showing the option.
set (VTK_USE_HYBRID ON CACHE INTERNAL "doc" FORCE)
mark_as_advanced (VTK_USE_HYBRID)
```

You might be tempted to edit the cache file directly, or to "initialize" a project by giving it an initial cache file. This may not work and could cause additional problems in the future. First, the syntax of the CMake cache is subject to change. Second, cache files have full paths in them that make them unsuitable for moving between binary trees. So if you want to initialize a cache file use one of the two standard methods described above.

2.8 Building Your Project

After you have run CMake your project will be ready to be built. If your target generator is based on Makefiles then you can build your project by changing directory to your binary tree and typing make (or gmake or nmake as appropriate). If you generated files for an IDE such as Visual Studio, you can start your IDE, load the project files into it, and build as you normally would.

Another option is to use CMake's –build option from the command line. This option is simply a convenience that allows you to build your project from the command line, even if that requires launching an IDE. The command line options for –build include:

```
Usage: cmake --build <dir> [options] [-- [native-options]]
Options:
  <dir>          = Project binary directory to be built.
  --target <tgt> = Build <tgt> instead of default targets.
  --config <cfg> = For multi-configuration tools, choose <cfg>.
  --clean-first  = Build target 'clean' first, then build.
                   (To clean only, use --target 'clean'.)
  --             = Pass remaining options to the native tool.
```

So even if you are using Visual Studio as your generator you can type the following to build your project from the command line if you wish.

```
cmake --build <your binary dir>
```

That is all there is to installing and running CMake for simple projects. In the following chapters we will consider CMake in more detail and how to use it on more complex software projects.