Introduction

The Cross-Platform Make facility (CMake) manages the build process—in a portable manner—across Windows and Unix platforms (hopefully Mac as well in the near future). CMake can be used to compile source code, create libraries, and build executables in arbitrary combinations. On Unix platforms, CMake produces makefiles that may be used with the standard make facility. In the Microsoft Visual C++ environment, CMake creates projects and workspaces that can be imported into MSVC.

CMake is designed to support complex directory hierarchies and applications dependent on several libraries. For example, CMake supports projects consisting of multiple toolkits (i.e., libraries), where each toolkit might contain several directories, and the application depends on the toolkits plus additional code. CMake can also handle situations where executables must be built in order to generate code that is then compiled and linked into a final application.

Using CMake is simple. The build process is controlled by creating a CMakeLists.txt file in each directory (including subdirectories) of a project. Each CMakeLists.txt file consists of one or more commands. Each command has the form COMMAND (args…) where COMMAND is the name of the command, and args is a white-space separated list of arguments. CMake provides many pre-defined commands, but if you need to, you can add your own commands. In addition, the advanced user can add other makefile generators for particular compiler/OS combinations.

Installing CMake

You can download and install precompiled binaries of CMake for Windows and UNIX from http://public.kitware.com/CMake. If you want to build CMake yourself, you can download the source code using CVS (available at http://cvshome.org) and typing:

cvs -d :pserver:anonymous@public.kitware.com:/insight/cmakecvsroot login
(respond with password cmake)

Follow this command by checking out the source code:
cvs -d :pserver:anonymous@public.kitware.com:/insight/cmakecvsroot co CMake

Then you can build CMake on Windows by loading the CMake.Source/CMake.dsp file into Microsoft Visual Studio or on UNIX by running

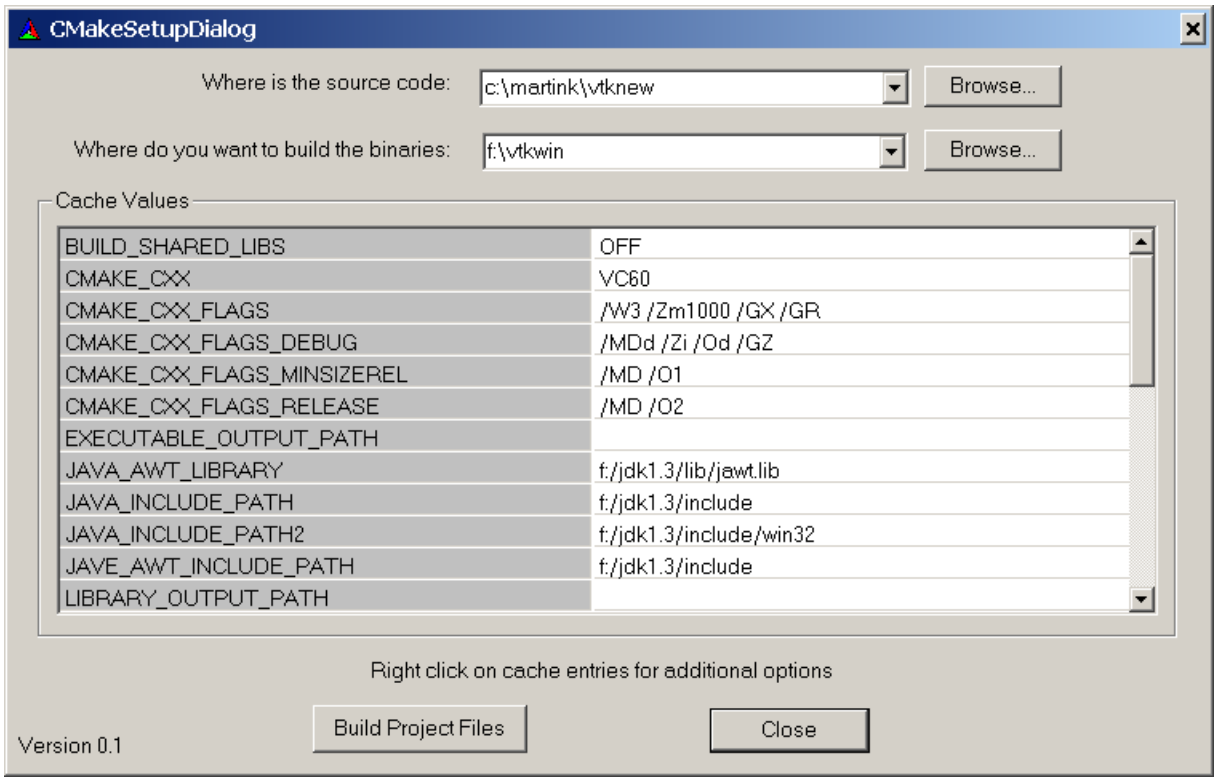
cd CMake
./configure
make
make install

Running CMake

Once CMake has been installed on your system using it to build a project is easy. We will cover the process for Windows and then UNIX.

Running CMake for Windows / Microsoft Visual C++ (MSVC)

Run CMakeSetup.exe which should be in your Start menu under Program Files, there may also be a shortcut on your desktop. A GUI will appear similar to what is shown below (but possibly different as CMake is still being developed). 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.



The cache values area is where you can specify different options for the build process. The example shown below is for VTK which has a large number of options. Once you have specified the source code and binary directories you should click the Build Project Files button. This will cause CMake to read in the CMakeLists.txt files from the source code directory and produce Microsoft Visual C++ workspaces. It will also cause the cache area to be updated to display any new options for the project. Adjust your cache settings if desired and click the Build project Files button again. Keep repeating this process until you are happy with the cache options (typically just once or twice).

CMakeSetup.exe generates a MSVC workspace file in the binary directory you specified. Typically this file has the same name as what you are compiling (e.g. VTK.dsp, ITK.dsw etc).
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.

Running CMake on Unix

Using CMake to build a project on UNIX is a simple process. Change directory into where you want the binaries to be placed. This can be the same directory as the source code for what we call in-place builds (the binaries are in the same place as the source code) or it can be a new directory you create. For an in-place build you then run cmake and it will produce a CMakeCache.txt file that contains build options that you can adjust using any text editor. For non in-place builds the process is the same except you run cmake and provide the path to the source code as its argument. Once you have edited the CMakeCache.txt file you rerun cmake, repeat this process until you are happy with the cache settings. The type make and your project should compile. Some projects will have install targets as well so you can type make install to install them.

Two examples of CMake usage on the Unix platform follow for a hello world project called Hello. In the first example, and in-place build is performed, i.e., the binaries are placed in the same directory as the source code.

```
cd Hello
cmake
(optionally edit CMakeCache.txt and rerun cmake)
make
```

In the second example, an out-of-place build is performed, i.e., the source code, libraries, and executables are produced in a directory separate from the source code directory(ies).

```
mkdir Hello-Linux
cd Hello-Linux
cmake ../Hello
(optionally edit CMakeCache.txt and rerun cmake ../Hello)
```

make

Why do I have to edit the cache more than once for some projects?

Some projects are very complex and setting one value in the cache may cause new options to appear the next time the cache is built. For example, VTK supports the use of MPI for performing distributed computing. This requires the build process to determine where the MPI libraries and header files are and to let the user adjust their values. But MPI is only available if another option VTK_USE_PARALLEL is first turned on in VTK. So to avoid confusion for people who don't know what MPI is, we hide those options until VTK_USE_PARALLEL is turned on. So CMake shows the VTK_USE_PARALLEL option in the cache area, if the user turns that on and rebuilds the cache, new options will appear for MPI that they can then set. The rule is to keep building the cache until it doesn't change. For most projects this will be just once. For some complicated ones it will be twice.

Developer's Guide

This section describes how to use CMake from the software developer's point of view. That is, if your aim is to use CMake to manage your build process, read this section first. An Extension Guide follows later in this document to explain the internals of CMake, and how to setup the CMake environment. Read that section only if you plan to install, extend, or enhance the features of CMake. This section of the User's Guide begins with a description of the CMake inputs. Examples then follow to clarify these descriptions.

Input to CMake

CMake's input is the text file CMakeLists.txt in your source directory. This input file specifies the things that need to be built in the current directory. The CMakeLists.txt consists of one or more commands. Each command is of the form:

COMMAND(args…)

Where COMMAND is the name of the command, and args is a white-space separated list of arguments to the command. (Arguments with embedded white-space should be quoted.) Typically there will be a CMakeLists.txt file for each directory of the project. Let's start with a simple example. Consider building hello world. You would have a source tree with the following files:

Hello.c CMakeLists.txt

The CMakeLists.txt file would contain two lines:

```
PROJECT (Hello)
ADD_EXECUTABLE(Hello Hello.c)
```

To build the Hello executable you just follow the process described in **Running CMake** above 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.

```
PROJECT (HELLO)
SOURCE_FILES(HELLO_SRCS Hello.c File2.c File3.c)
IF (WIN32)
  SOURCE_FILES(HELLO_SRCS WinSupport.c)
ELSE (WIN32)
  SOURCE_FILES(HELLO_SRCS UnixSupport.c)
ENDIF (WIN32)
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_ADD_LIBRARY (Hello TCL_LIBRARY)
ENDIF (TCL_LIBRARY)
```

In this example the SOURCE_FILES 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. And finally the ADD_EXECUTABLE command is used to build the executable with the files listed in the source list HELLO_SRCS. The FIND_LIBRARY command looks for the Tcl library under a few different names and in a few different paths, and if it is found adds it to the link line for the Hello executable target. Note the use of the # character to denote a comment line.

CMake always defines some variables for use within CMakeList files. For example, WIN32 is always defined on windows systems and UNIX is always defined for UNIX systems. CMake defines a number of commands. A brief summary of the most commonly used commands follows here. Later in the document an exhaustive list of all pre-defined commands is presented. (You may also add your own commands, see the Extension Guide for more information.)

A) Build Targets:

SOURCE_FILES()

SUBDIRS()
ADD_LIBRARY()
ADD_EXECUTABLE()
AUX_SOURCE_DIRECTORY()
PROJECT()

CMake works recursively, descending from the current directory into any subdirectories listed in the SUBDIRS command. The command SOURCE_FILES is used for grouping source files together for later use. (Note: currently only C and C++ code can be compiled.) ADD_LIBRARY adds a library to the list of targets this makefile will produce. ADD_EXECUTABLE adds an executable to the list of targets this makefile will produce. (Note: source code is compiled first, then libraries are built, and then executables are created.) The AUX_SOURCE_DIRECTORY is a directory where other source code, not in this directory, whose object code is to be inserted into the current LIBRARY. All source files in the AUX_SOURCE_DIRECTORY are compiled (e.g. *.c, *.cxx, *.cpp, etc.). PROJECT (PojectName) is a special variable used in the MSVC to create the project for the compiler, it also defines two useful variables for CMAKE: ProjectName_SOURCE_DIR and ProjectName_BINARY_DIR.

B) Build flags and options. In addition to the commands listed above, CMakeLists.txt often contain the following commands:

INCLUDE_DIRECTORIES()
LINK_DIRECTORIES()
LINK_LIBRARIES()
TARGET_LINK_LIBRARIES()

These commands define directories and libraries used to compile source code and build executables. An important feature of the commands listed above is that are inherited by any subdirectories. That is, as CMake descends through a directory hierarchy (defined by SUBDIRS()) these commands are expanded each time a definition for a command is encountered. For example, if in the top-level CMakeLists file has INCLUDE_DIRECTORIES(/usr/include), with SUBDIRS(./subdir1), and the file ./subdir1/CMakeLists.txt has INCLUDE_DIRECTORIES(/tmp/foobar), then the net result is

INCLUDE_DIRECTORIES(/usr/include /tmp/foobar)

C) CMake comes with a number of modules that look for commonly used packages such as OpenGL or Java. These modules save you from having to write all the CMake code to find these packages yourself. Modules can be used by including them into your CMakeList file as shown below.

INCLUDE (${CMAKE_ROOT}/Modules/FindTCL.cmake)

CMAKE_ROOT is always defined in CMake and can be used to point to where CMake was installed. Looking through some of the files in the Modules subdirectory can provide good ideas on how to use some of the CMake commands.

Adding A New Directory to a project

A common way to extend a project is to add a new directory. This involves three steps:

1. Create the new directory somewhere in your source directory hierarchy.
2. Add the new directory to the SUBDIRS command in the parent directories CMakeLists.txt
3. Create a CMakeLists.txt in the new directory with the appropriate commands

CMake Commands

The following is an exhaustive list of pre-defined CMake commands, with brief descriptions.

ABSTRACT_FILES - A list of abstract classes, useful for wrappers.
Usage: ABSTRACT_FILES(file1 file2 ..)

ADD_CUSTOM_TARGET - Add an extra target to the build system that does not produce output, so it is run each time the target is built.
Usage: ADD_CUSTOM_TARGET(Name "command to run" ALL)
The ALL option is optional. If it is specified it indicates that this target should be added to the Build all target.

ADD_DEFINITIONS - Add -D define flags to command line for environments.
Usage: ADD_DEFINITIONS(-DFOO -DBAR ...)
Add -D define flags to command line for environments.

ADD_EXECUTABLE - Add an executable to the project that uses the specified srclists
Usage: ADD_EXECUTABLE(exename srclist srclist srclist ...)
ADD_EXECUTABLE(exename WIN32 srclist srclist srclist ...)This command adds an executable target to the current directory. The executable will be built from the source files / source lists specified. The second argument to this command can be WIN32 which indicates that the executable (when compiled on windows) is a windows app (using WinMain)not a console app (using main).

ADD_LIBRARY - Add an library to the project that uses the specified srclists
Usage: ADD_LIBRARY(libname srclist srclist srclist ...)

ADD_TEST - Add a test to the project with the specified arguments.
Usage: ADD_TEST(testname exename arg1 arg2 arg3 ...)
This command adds a test target to the current directory. The tests are run by the testing subsystem by executing exename with the specified

arguments. exename can be either an executable built by built by this project or an arbitrary executable on the system (like tclsh).

AUX_SOURCE_DIRECTORY - Add all the source files found in the specified directory to the build as source list NAME.
Usage: AUX_SOURCE_DIRECTORY(dir srcListName)

BUILD_COMMAND - Determine the command line that will build this project.
Usage: BUILD_COMMAND(NAME)
Within CMAKE set NAME to the command that will build this project from the command line.

BUILD_NAME - Set a CMAKE variable to the build type.
Usage: BUILD_NAME(NAME)
Within CMAKE sets NAME to the build type.

BUILD_SHARED_LIBRARIES - Build shared libraries instead of static
Usage: BUILD_SHARED_LIBRARIES()

CABLE_CLASS_SET - Define a set of classes for use in other CABLE commands.
Usage: CABLE_CLASS_SET(set_name class1 class2 ...)
Defines a set with the given name containing classes and their associated header files. The set can later be used by other CABLE commands.

CABLE_WRAP_TCL - Wrap a set of classes in Tcl.
Usage: CABLE_WRAP_TCL(target class1 class2 ...)
Wrap the given set of classes in Tcl using the CABLE tool. The set of source files produced for the given package name will be added to a source list with the given name.

CONFIGURE_FILE - Create a file from an autoconf style file.in file.
Usage: CONFIGURE_FILE(InputFile OutputFile [COPYONLY])
The Input and Ouput files have to have full paths.
They can also use variables like CMAKE_BINARY_DIR,CMAKE_SOURCE_DIR.
This command replaces any variables in the input file with their values as determined by CMake. If a variables in not defined, it will be replaced with nothing. If COPYONLY is passed in, then then no varible expansion will take place.

ELSE - starts the else portion of an if block
Usage: ELSE(define)

ENABLE_TESTING - Enable testing for this directory and below.
Usage: ENABLE_TESTING()
Enables testing for this directory and below. See also the ADD_TEST command.

ENDIF - ends an if block
Usage: ENDIF(define)

EXEC_PROGRAM - Run and executable program during the processing of the CMakeList.txt file.
Usage: EXEC_PROGRAM(Executble)

FIND_FILE - Find a file.
Usage: FIND_FILE(NAME file extrapath extrapath ...)

FIND_LIBRARY - Find a library.
Usage: FIND_LIBRARY(DEFINE_PATH libraryName [NAMES] name1 name2 name3 [PATHS path1 path2 path3...])
If the library is found, then DEFINE_PATH is set to the full path where it was found

FIND_PATH - Find a path for a file.
Usage: FIND_PATH(PATH_DEFINE fileName path1 path2 path3...)
If the file is found, then PATH_DEFINE is set to the path where it was found

FIND_PROGRAM - Find an executable program.
Usage: FIND_PROGRAM(NAME executable1 extrapath extrapath ...)

GET_FILENAME_COMPONENT - Get a specific component of a full filename.
Usage: GET_FILENAME_COMPONENT(VarName FileName PATH|NAME|EXT|NAME_WE)
Set VarName to be the path (PATH), file name (NAME), file extension (EXT) or file name without extension (NAME_WE) of FileName.
Note that the path is converted to Unix slashes format and has no trailing slashes. The longest file extension is always considered.

IF - start an if block
Usage: IF (define) Starts an if block. Optionally there it can be invoked as IF (NOT Define) the matching ELSE and ENDIF require the NOT as well.

INCLUDE - Basically identical to a C #include "somthing" command.
Usage: INCLUDE(file1 file2)

INCLUDE_DIRECTORIES - Add include directories to the build.
Usage: INCLUDE_DIRECTORIES(dir1 dir2 ...)

INCLUDE_REGULAR_EXPRESSION - Set the regular expression used for dependency checking.
Usage: INCLUDE_REGULAR_EXPRESSION(regex)
Sets the regular expression used in dependency checking. Only include files matching this regular expression will be traced.

INSTALL_FILES - Create install rules for files
Usage: INSTALL_FILES(path extension srclist file file srclist ...)
Create rules to install the listed files into the path. Path is relative to the variable PREFIX. The files can be specified explicitly or by referenceing source lists. All files must either have the extension specified or exist with the extension appended. A typical extension is .h etc...

INSTALL_TARGETS - Create install rules for targets

Usage: INSTALL_TARGETS(path target target)
Create rules to install the listed targets into the path. Path is relative to the variable PREFIX

LINK_DIRECTORIES - Specify link directories.
Usage: LINK_DIRECTORIES(directory1 directory2 ...)
Specify the paths to the libraries that will be linked in. The directories can use built in definitions like CMAKE_BINARY_DIR and CMAKE_SOURCE_DIR.

LINK_LIBRARIES - Specify a list of libraries to be linked into executables or shared objects.
Usage: LINK_LIBRARIES(library1 <debug | optimized> library2 ...)
Specify a list of libraries to be linked into executables or shared objects. This command is passed down to all other commands. The debug and optimized strings may be used to indicate that the next library listed is to be used only for that specific type of build

LOAD_CACHE - load in the values from another cache.
Usage: LOAD_CACHE(pathToCacheFile)
Load in the values from another cache. This is useful for a project that depends on another project built in a different tree.

MAKE_DIRECTORY - Create a directory in the build tree if it does not exist. Parent directories will be created if the do not exist..
Usage: MAKE_DIRECTORY(directory)

MESSAGE - Display a message to the user.
Usage: MESSAGE("the message to display" "Title for dialog")
The first argument is the message to display. The second argument is optional and is the title for the dialog box on windows.

OPTION - Provides an option that the user can optionally select
Usage: OPTION(USE_MPI "help string decribing the option" [initial value])
Provide an option for the user to select

PROJECT - Set a name for the entire project. One argument.
Usage: PROJECT(projectname) Sets the name of the Microsoft workspace .dsw file.

SET - Set a CMAKE variable to a value
Usage: SET(VAR [VALUE] [CACHE TYPE DOCSTRING])
Within CMAKE sets VAR to the value VALUE. VALUE is expanded before VAR is set to it. If CACHE is present, then the VAR is put in the cache. TYPE and DOCSTRING are required. If TYPE is INTERNAL, then the VALUE is Always written into the cache, replacing any values existing in the cache. If it is not a CACHE VAR, then this always writes into the current makefile.

SITE_NAME - Set a CMAKE variable to the name of this computer.
Usage: SITE_NAME(NAME)
Within CMAKE sets NAME to the host name of the computer.

SOURCE_FILES - Add a list of source files, associate them with a NAME.
Usage: SOURCE_FILES(NAME file1 file2 ...)

SOURCE_GROUP - Define a grouping for sources in the makefile.
Usage: SOURCE_GROUP(name regex)
Defines a new source group. Any file whose name matches the regular expression will be placed in this group. The LAST regular expression of all defined SOURCE_GROUPs that matches the file will be selected.

SUBDIRS - Add a list of subdirectories to the build.
Usage: SUBDIRS(dir1 dir2 ...)
Add a list of subdirectories to the build. This will cause any CMakeLists.txt files in the sub directories to be processed by CMake.

TARGET_LINK_LIBRARIES - Specify a list of libraries to be linked into executables or shared objects.
Usage: TARGET_LINK_LIBRARIES(target library1 <debug | optimized> library2 ...)
Specify a list of libraries to be linked into the specified target The debug and optimized strings may be used to indicate that the next library listed is to be used only for that specific type of build

UTILITY_SOURCE - Specify the source tree of a third-party utility.
Usage: UTILITY_SOURCE(cache_entry executable_name path_to_source [file1 file2 ...])
When a third-party utility's source is included in the distribution, this command specifies its location and name. The cache entry will not be set unless the path_to_source and all listed files exist. It is assumed that the source tree of the utility will have been built before it is needed.

VTK_WRAP_JAVA - Create Java Wrappers.
Usage: VTK_WRAP_JAVA(resultingLibraryName SourceListName SourceLists ...)

VTK_WRAP_PYTHON - Create Python Wrappers.
Usage: VTK_WRAP_PYTHON(resultingLibraryName SourceListName SourceLists ...)

VTK_WRAP_TCL - Create Tcl Wrappers for VTK classes.
Usage: VTK_WRAP_TCL(resultingLibraryName [SOURCES] SourceListName SourceLists ... [COMMANDS CommandName1 CommandName2 ...])

WRAP_EXCLUDE_FILES - A list of classes, to exclude from wrapping.
Usage: WRAP_EXCLUDE_FILES(file1 file2 ..)

Extending CMake Guide

This section describes some of the internals of CMake. Read this section only if you intend to add new commands to the CMake executable or debug CMake. First you must download and install the source code for CMake as described in the **Installing CMake** section.

Adding a New Rule

Rules can be added to CMake by deriving new commands from the class cmCommand (defined in CMake/Source/cmCommand.h/.cxx). Typically each rule is implemented in a class called cmRuleNameCommand stored in cmRuleNameCommand.h and cmRuleNameCommand.cxx. If you want to create a rule the best bet is to take a look at some of the existing rules in CMake. They tend to be fairly short.

Adding a New Makefile Generator

Different types of makefiles (corresponding to a different compiler and/or operating system) can be added by subclassing from cmMakefileGenerator (defined in cmMakefileGenerator.h/.cxx). Makefile generators process the information defined by the commands in CMakeLists.txt to generate the appropriate makefile(s). Again, the best bet is to work from one of the existing generators.

Further Information

Much of the development of CMake was performed at Kitware http://www.kitware.com/. The developers can be reached at mailto:kitware@kitware.com. CMake was initially developed for the NIH/NLM Insight Segmentation and Registration Toolkit, see the Web site at http://public.kitware.com/Insight.html.