Cross-Platform Make (CMake)

Introduction

The Cross-Platform Make facility (CMake) manages the build process—in a portable manner—across Microsoft Visual C++ and Unix platforms. CMake can be used to compile source code, create libraries, and build executables in arbitrary combination. On Unix platforms, CMake uses configure to build 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. (These projects and workspaces are created by running the program CMakeSetup.exe.)

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 one or more CMakeLists.txt files in each directory (including subdirectories) that make up a project. Each CMakeLists.txt 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 a particular compiler/OS combination.

User's Guide

This section describes how to use CMake from the user's point of view. That is, if your aim is to use CMake to manage your build process, read this section first. A Developer's Guide section 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 uses the following files and variables. You must set these variables and create these files before CMake will run properly.

1. Variables (expanded as necessary during the CMake process):

${CMAKE_SOURCE_DIR}	The root directory of the source code directory tree.

${CMAKE_BINARY_DIR}	The root directory of the build tree where binaries are placed. This includes object files, libraries, and executables.
${WIN32}	Defined on any Microsoft Windows systems
${UNIX}	Defined on any UNIX systems. Note that with cygwin on Microsoft Windows both WIN32 and UNIX will be defined.

Typically the user indirectly defines these variables by running a program. In MSVC, the user runs a program (CMakeSetup.exe) defining the location of the source code and where the binaries (object code, libraries, and executables) are placed. On Unix, configure is used to indicate where the source code and binaries are located. (The location of the configure file is the source directory, the directory in which configure is run is the binary directory.)

2. File CMakeLists.txt:
   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.) Note that for a given project, one or more CMakeLists.txt files may be required depending on the number and organization of the directories.

   CMake defines a number of commands. A brief summary of these 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 Developer's Guide for more information.)

   A) Build Targets:
      SOURCE_FILES()
      ABSTRACT_CLASSES()
      SUBDIRS()
      LIBRARY()
      EXECUTABLES()
      AUX_SOURCE_DIRECTORY()
      PROJECT()

      CMake works recursively, descending from the current directory into the subdirectories listed in the SUBDIRS variable. The variable SOURCE_FILES lists all source code that must be compiled for all platforms (Note: currently only C and C++ code can be compiled.) WIN32_SOURCE_FILES is a list of all source code that must be compiled in the MSVC

environment. Similarly, UNIX_SOURCE_FILES lists all source code that must be compiled on Unix platforms. ABSTRACT_CLASSES are modules that must be compiled but are not instantiable (i.e., instances cannot be created because the class has pure virtual functions). This is important in some applications where wrapper code can be generated around instantiable modules. LIBRARY defines the name of a library into which all object code is inserted and EXECUTABLES are any executables that must be built. (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 is a special variable used in the MSVC to create the project for the compiler.

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()
WIN32_LIBRARIES()
UNIX_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 recursively expanded as each subdirectory is visited. 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 directory is defined by INCLUDE_DIRECTORIES(/usr/include), with SUBDIRS(./subdir1), and the file ./subdir1/CMakeLists.txt defines INCLUDE_DIRECTORIES(/tmp/foobar), then the net result is

```
INCLUDE_DIRECTORIES(/usr/include /tmp/foobar)
```

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_TARGET - Add an extra target to the build system.
  Usage: ADD_TARGET(Name "command to run")

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

- EXECUTABLES - Add a list of executables files.
  Usage: EXECUTABLES(file1 file2 ...)

- FIND_INCLUDE - Find an include path.
  Usage: FIND_INCLUDE(DEFINE try1 try2 ...)

- FIND_LIBRARY - Find a library.
  Usage: FIND_LIBRARY(DEFINE try1 try2)

- FIND_PROGRARM - Find an executable program.
  Usage: FIND_PROGRAM(NAME executable1 executable2 ...)

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

- LIBRARY - Set a name for a library.
  Usage: LIBRARY(libraryname)

- 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 library2)
  Specify a list of libraries to be linked into executables or shared objects. This command is passed down to all other commands. The library name should be the same as the name used in the LIBRARY(library) command.

- PROJECT - Set a name for the entire project. One argument.
  Usage: PROJECT(projectname)

- SOURCE_FILES - Add a list of source files. Usage: SOURCE_FILES(file1 file2 ...)

- SOURCE_FILES_REQUIRE - Add a list of source files if the required variables are set.
  Usage: SOURCE_FILES_REQUIRE(var1 var2 ... SOURCES_BEGIN file1 file2 ...)

- 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.

- TESTS - Add a list of executables files that are run as tests.
  Usage: TESTS(file1 file2 ...)

- UNIX_DEFINES - Add -D flags to the command line for Unix only.
  Usage: UNIX_DEFINES(-DFOO -DBAR)
  Add -D flags to the command line for Unix only.

- UNIX_LIBRARIES - Add libraries that are only used for Unix programs.
  Usage: UNIX_LIBRARIES(library -lm ...)

- WIN32_DEFINES - Add -D define flags to command line for Win32 environments.
  Usage: WIN32_DEFINES(-DFOO -DBAR ...)
  Add -D define flags to command line for Win32 environments.

- WIN32_LIBRARIES - Add libraries that are only used for Win32 programs.
  Usage: WIN32_LIBRARIES(library -lm ...)

Using CMake

The instructions in this section assume that the CMake environment has been properly installed on your system. (See the Developer's Guide later in this document for instructions on installing the CMake environment.) These instructions are fairly general, each section has examples demonstrating the use of CMake in a particular circumstance.

A Simple Build

The build process varies depending on the platform as described in the following sections.

Microsoft Visual C++ (MSVC)

In the MSVC environment, two executable programs are used.

CMakeSetup.exe — GUI-based tool for configuring CMake in the MSVC environment.

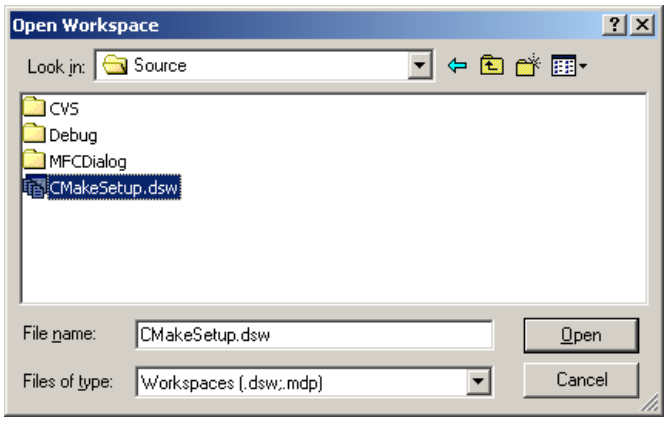
CMakeSetupCMD.exe — windows command line version of CmakeSetup

Typically the user will use CmakeSetup.exe for configuring in the MSVC environment. For those who prefer a scripting/command-line environment; however, CMakeSetupCMD.exe provides an alternative. (Note: these programs may have to be built from the CMake/Source directory, so the executable may not initially exist. Once you load and build the workspace as described in the following, the executables will be created.)

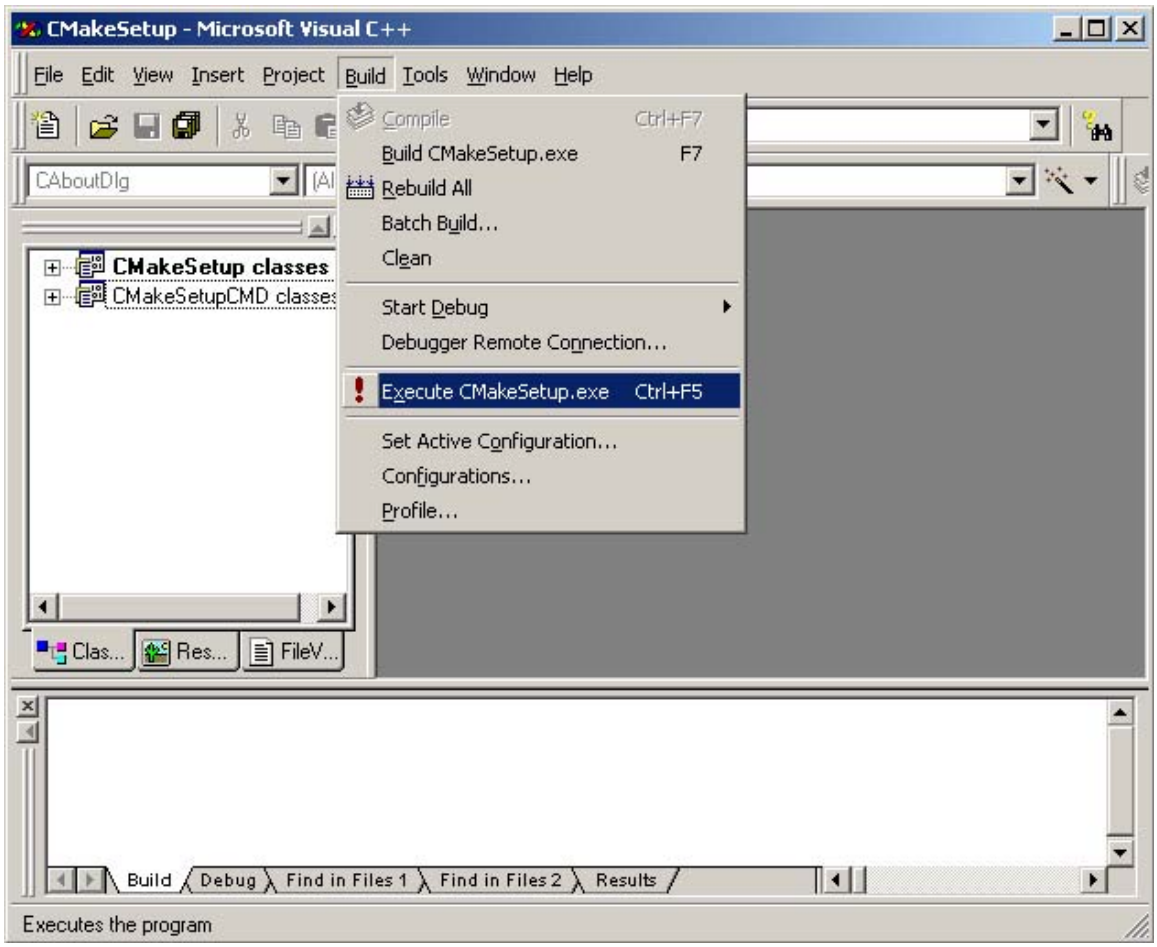
1. Using CMakeSetup:

- Load CMake/Source/CMakeSetup.dsw into MSVC.
- Build CMakeSetup.exe
- Run CMakeSetup.exe
- Define the variables CMAKE_SOURCE_DIR and CMAKE_BINARY_DIR by typing in the blanks in the GUI.
- Click OK.
- Load the generated project file. The project file will be found in CMAKE_BINARY_DIR.
- Build the appropriate project(s).

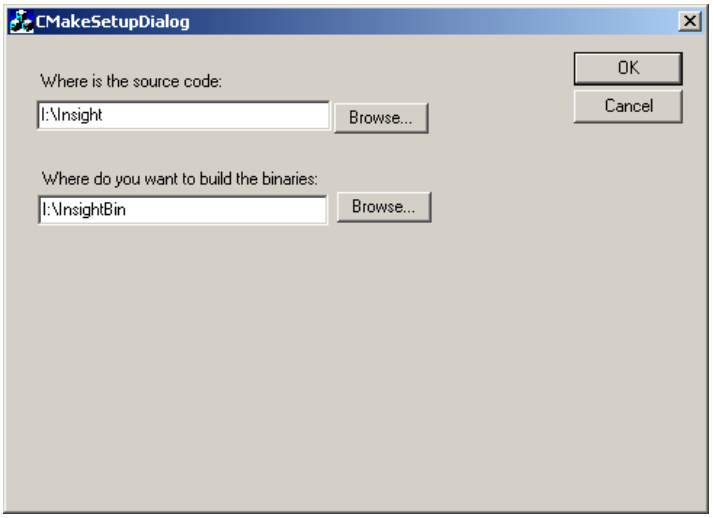
The following example demonstrates the use of CMake in the NLM Insight Segmentation and Registration Toolkit environment. Begin by opening the CMakeSetup.dsw workspace by choosing File/Open Workspace…



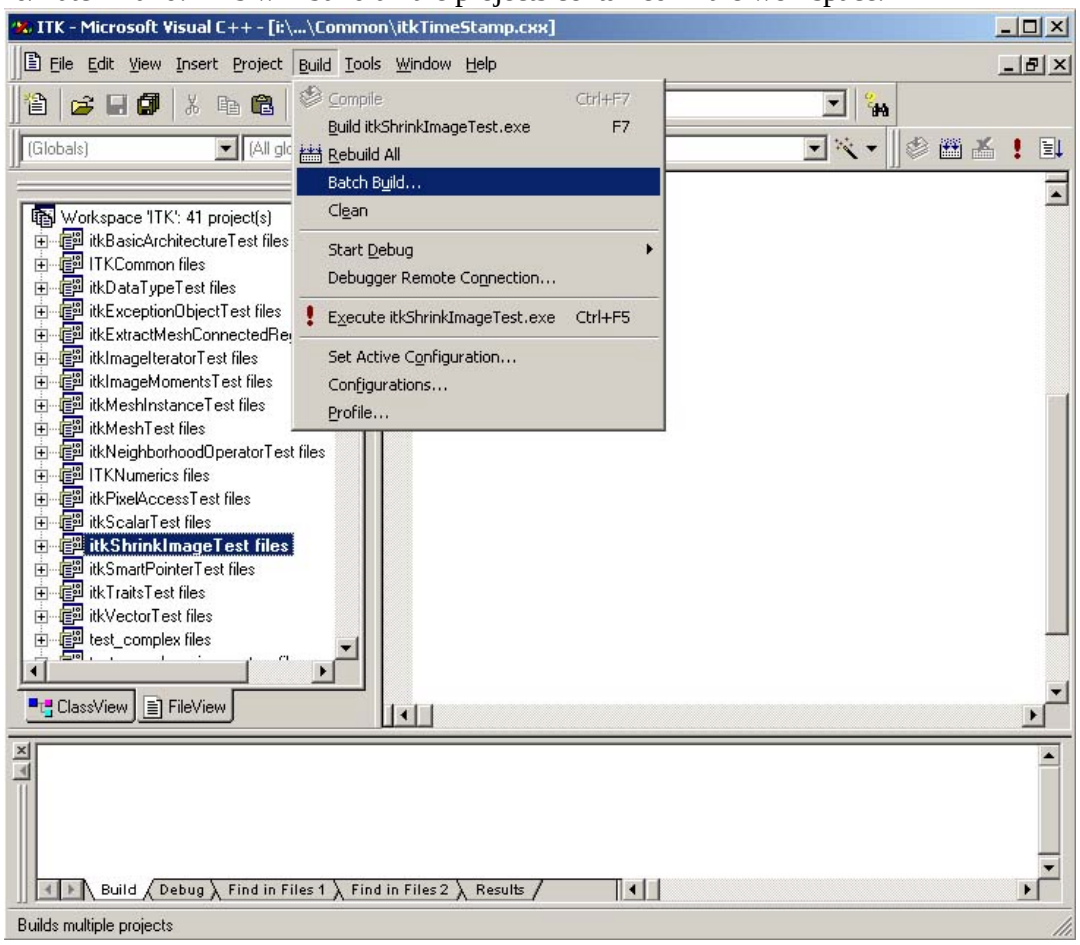
Next, run and build CMakeSetup.exe by choosing Build/Execute. MSVC may indicate that some files are out-of-date and need rebuilding. Answer YES and proceed.



Once CMakeSetup.exe is compiled, it will run and the following GUI will appear as shown below. Notice that the location of the source root directory (CMAKE_SOURCE_DIR) and the binary root directory (CMAKE_BINARY_DIR) are set. (Use the browse button if necessary to set/locate these directories. If the binary directory does not exist, it will be created.) CMakeSetup.exe will run for a second or two and then disappear.



CMakeSetup.exe generates a MSVC project file named according the CMake variable PROJECT described previously. In this example, PROJECT(ITK), so the MSVC workspace is ITK.dsw (and is placed in the CMAKE_BINARY_DIR, in this example I:\InsightBin). The next step in this process is to open the workspace with MSVC. Once open, the project can be built by choosing Build/Batch Build. This will build all the projects contained in the workspace.



2. Using CMakeSetupCMD.exe:

   - Load CMake/Source/CMakeSetup.dsw into MSVC.
   - Build CMakeSetup.exe (this will generate CMakeSetupCMD.exe).
   - Run CMakeSetupCMD.exe with the appropriate command-line arguments.
   - Load the generated project file. The project file will be found in CMAKE_BINARY_DIR.
   - Build the appropriate project(s).

   Using the command line version CMakeSetupCMD.exe is similar to the process described previously. The only difference is that CMakeSetupCMD.exe is run instead of CMakeSetup.exe.

The following example demonstrates the use of CMakeSetupCMD.exe in the NLM Insight Segmentation and Registration Toolkit environment. To run the program you must have a MS-DOS window or similar (cygwin shell). Then type:

```
CMakeSetupCMD.exe Makefile.in –DSW|-DSP \
        –H$CMAKE_SOURCE_DIR -B$CMAKE_BINARY_DIR \
        –D(current source directory) –O(current binary directory)
```

(The "\" indicates line continuation, you do not have to type this.) This usage reflects that CMakeSetupCMD.exe can be run anywhere in the source directory tree. Therefore, not only must you define the CMAKE_SOURCE_DIR and CMAKE_BIN_DIR variable definitions using the –H –B command line arguments. In addition, you must indicate where you are in the source tree using the –D argument, and the equivalent location in the binary directory tree with the –O argument. In addition, you must specify whether to build the workspace containing all the projects, or whether to build just the project corresponding to this directory.

Unix

On Unix, configure and the standard make utility are used as follows:

- Run configure from the CMAKE_BINARY_DIR.
- Type "make" in the same dir.

The CMake variables CMAKE_SOURCE_DIR and CMAKE_BINARY_DIR are defined when configure is run. The directory that configure is run from determines where the binaries are placed; the location of the configure program determines where the source tree is located.

Two examples of CMake usage on the Unix platform follow (using the Insight system). 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 (into the CMAKE_SOURCE_DIR directory)
./configure
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 Build-dir
cd Build-dir
$CMAKE_SOURCE_DIR /configure
make
```

Developing with CMake

The preceding instructions enable users to compile and build CMake projects. If all you need to do is use the resulting binary code, these instructions are sufficient. However, if you are planning to extend your installation by adding new source files, libraries, and/or directories, you will need to modify the CMakeLists.txt.

Adding A New Source Module

One of the most common ways to extend CMake is to add a new, compilable source module. (Here we mean that a compilable module is a source code module that when processed by a compiler generates object code. A .h header file or .txx template file is not a compilable module by this definition). Adding a source module means adding it the list of source files by adding it to the SOURCE_FILES command defined in the CMakeLists.txt file. Here is an example where the module itkFoo.cxx is added.

```
SOURCE_FILES(
itkDataObject
itkDirectory
itkFoo
)
```

Once the source module is added, you can compile as usual.

Adding A New Directory

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

1. Create the directory somewhere in the CMAKE_SOURCE_DIR directory hierarchy.

2. Add the directory to the SUBDIRS command in CMakeLists.txt

3. Create a CMakeLists.txt in the new directory with the appropriate variables defined.

Developers Guide

This section describes some of the internals of CMake. Read this section only if you intend to extend or debug CMake.

Installing CMake

To install CMake, you must copy the directory structure and source code found in the CMake directory into the CMAKE_SOURCE_DIR directory. More advanced users may also want to modify the configure.in files to control particular features of the project.

Additional Platform-Dependent Details

The following describes platform-dependent details.

Microsoft Visual C++ (MSVC)

1. CMakeSetupConfig.MSC – The configuration input file for CMakeSetup when Microsoft projects are created. This is used to generate header files that would normally be created by configure on Unix.

    # itk configure file, just copy the .h.in to the .h
    ${CMAKE_BINARY_DIR}:itkConfigure.h:${CMAKE_SOURCE_DIR}/itkConfigure.h.in

    # for the vcl configure copy the vc60.h config file
    ${CMAKE_BINARY_DIR}/Code/Insight3DParty/vxl/vcl:vcl_config.h:${CMAKE_SOURCE_DIR}/Code/Insight3DParty/vxl/vcl/vcl_config-vc60.h

Unix

1. Unix scripts and programs. In general you should never have to modify these.
    configure.in — used by autoconf to generate configure
    configure — run on Unix to configure the build
    CMakeBuildTargets — Unix program to read CMakeLists.txt and generate CMakeTargets.make
    makefile fragments:
    CMakeMaster.make.in — main file to be included by makefiles
    CMakeVariables.make.in — all make variables are set in this file
    CMakeRules.make.in — All build rules are here (except Simple Rules)
    CMakeSimpleRules.make.in - simple build rules for .o to .cxx, this is separate to be able to build CMakeBuildTargets itself.
    CMakeLocal.make.in — Place for hand configuration
    CMakeTargets.make — generated rules for make style build in each directory
    MakefileTemplate.make.in — master makefile template used by configure to generate Makefiles

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).

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).

Further Information

Bill Hoffman was the principal developers of CMake. Reach him at bill.hoffman@kitware.com. This document was written by Will Schroeder and Bill Hoffman. Reach Will at will.schroeder@kitware.com.

To learn more about the NIH/NLM Insight Segmentation and Registration Toolkit, see the Web site at http://www.kitware.com/Insight.html.