Converting Existing Systems to CMake

For many people the first thing they will do with CMake is to convert an existing project from using an older build system to use CMake. In many cases this can be a fairly easy process, but there are a few issues to consider. This section will address those issues and provide some suggestions for effectively converting a project over to CMake. The first issue to consider when converting to CMake is the project's directory structure.

7.1 Source Code Directory Structures

Most small projects will have their source code in either the top level directory or in a directory named `src` or `source`. Even if all of the source code is in a subdirectory we highly recommend creating a CMakeLists file for the top level directory. There are two reasons for this. First it can be confusing to some people that they must run CMake on the subdirectory of the project instead of the main directory. Second, you may want to install documentation or other support files from the other directories. By having a CMakeLists file at the top of the project you can use the `add_subdirectory` command to step down into the documentation directory where its CMakeLists file can install the documentation (you can have a CMakeLists file for a documentation directory with no targets or source code).

For projects that have source code in multiple directories there are a few options. One option that many Makefile based projects use is to have a single Makefile at the top level directory that lists all the source files to compile in their subdirectories. For example:

```
SOURCES=\
  subdir1/foo.cxx \
  subdir1/foo2.cxx \
  subdir2/gah.cxx \
  subdir2/bar.cxx
```

This approach works just as well with CMake using a similar syntax:

```
set (SOURCES
  subdir1/foo.cxx
  subdir1/foo2.cxx
  subdir1/gah.cxx
  subdir2/bar.cxx
  )
```

Another option is to have each subdirectory build a library or libraries that can then be linked into the executables. In that case each subdirectory would define its own list of source files and add the appropriate targets. A third option is a mixture of the first two. Each subdirectory can have a CMakeLists file that lists its sources, but the top level CMakeLists file will not use the `add_subdirectory` command to step into the subdirectories. Instead the top level CMakeLists file will use the `include` command to include each of the subdirectory's CMakeLists files. For example, a top level CMakeLists file might include the following code:

```
# collect the files for subdir1

include (subdir1/CMakeLists.txt)
foreach (FILE ${FILES})
  set (subdir1Files ${subdir1Files} subdir1/${FILE})
endforeach (FILE)

# collect the files for subdir2
include (subdir2/CMakeLists.txt)
foreach (FILE ${FILES})
  set (subdir2Files ${subdir2Files} subdir2/${FILE})
endforeach (FILE)

# add the source files to the executable
add_executable (foo ${subdir1Files} ${subdir2Files})
```

While the CMakeLists files in the subdirectories might look like the following:

```
# list the source files for this directory
set (FILES
  foo1.cxx
  foo2.cxx
  )
```

The approach you use is entirely up to you. For large projects, having multiple shared libraries can certainly improve build times when changes are made. For smaller projects the other two approaches have their advantages. The main suggestion here is to choose a strategy and stick with it.

7.2 Build Directories

The next issue to consider is where to put the resulting object files, libraries, and executables. There are a few different approaches commonly used, and some work better with CMake than others. Probably the most common approach is to produce the binary files in the same directory as the source files. For some Windows generators such as Visual Studio they are actually kept in a subdirectory matching the selected configuration, e.g. debug, release, etc. CMake supports this approach by default. A closely related approach is to put the binary files into a separate tree that has the same structure as the source tree. For example if the source tree looked like the following:

```
foo/
  subdir1
  subdir2
```

the binary tree might look like:

```
foobin/
  subdir1
  subdir2
```

CMake also supports this structure by default. Switching between in-source builds and out-of-source builds is simply a matter of changing the binary directory in CMake (see How to Run CMake? on page 10). Note that if you have already done an in-source build and want to switch to an out of source build, you should start with a fresh copy of the source tree. If you need to support multiple architectures from one source tree we highly recommend a directory structure like the following:

```
projectfoo/
  foo/
    subdir1
    subdir2
  foo-linux/
    subdir1
    subdir2
  foo-solaris/
    subdir1
    subdir2
  foo-hpux/
    subdir1
    subdir2
```

That way each architecture has its own build directory which will not interfere with any other architecture. Remember that not only are the object files kept in the binary directories but also any configured files are typically written to the binary tree. Another tree structure found primarily on UNIX projects is one where the binary files for different architectures are kept in subdirectories of the source tree. (see below) CMake does not work well with this structure, so we recommend switching to the separate build tree structure shown above.

```
foo/
  subdir1/
    linux
    solaris
    hpux
  subdir2/
    linux
    solaris
    hpux
```

CMake provides two variables for controlling where binary targets are written. They are the `EXECUTABLE_OUTPUT_PATH` and `LIBRARY_OUTPUT_PATH` variables. These variables control where the resulting libraries and executables will be written respectively. Setting these enables a project to place all the libraries and executables into a single directory. For projects with many subdirectories this can be a real time saver. A typical implementation is listed below:

```
# Setup output directories.
set (LIBRARY_OUTPUT_PATH
  ${PROJECT_BINARY_DIR}/bin
  CACHE PATH
  "Single directory for all libraries."
  )

set ( EXECUTABLE_OUTPUT_PATH
  ${PROJECT_BINARY_DIR}/bin
  CACHE PATH
  "Single directory for all executables."
  )

mark_as_advanced (
  LIBRARY_OUTPUT_PATH
  EXECUTABLE_OUTPUT_PATH
  )
```

The two variables are set to the `bin` subdirectory of the project's binary tree (in this example the bin subdirectory could just as easily be named "binaries"). These variables are cached so that they will impact the entire project. Finally the variables are marked as advanced for two reasons; first to reduce the number of choices the average user will see when running CMake, and second to reduce the chances of someone changing them to another value. Setting these variables is very useful for projects that make use of shared libraries (DLLs) since it collects all of the shared libraries into one directory. If the executables are placed in the same directory then they can find the required shared libraries more easily when run on Windows.

One final note on directory structures: with CMake it is perfectly acceptable to have a project within a project. For example, within the Visualization Toolkit's source tree is a directory that contains a complete copy of the zlib compression library. In writing the CMakeLists file for that library we use the `PROJECT` command to create a project named VTKZLIB even though it is within the VTK source tree and project. This has no real impact on VTK, but it does allow us to build zlib independent of VTK without having to modify its CMakeLists file.

7.3 Useful CMake Commands When Converting Projects

There are a few CMake commands that can make the job of converting an existing project easier and faster. The `file` command with the `GLOB` argument allows you to quickly set a variable containing a list of all the files that match the glob expression. For example:

```
# collect up the source files
file (GLOB SRC_FILES *.cxx)

# create the executable
add_executable (foo ${SRC_FILES})
```

will set the `SRC_FILES` variable to a list of all the "`.cxx`" files in the current source directory. Then it will create an executable using those source files. Windows developers should be aware that glob matches are case sensitive.

Two other useful commands are `make_directory` and `exec_program`. By default CMake will create all the output directories it needs for the object files, libraries, and executables. With existing projects there may be some part of the build process that creates directories that CMake would not normally create. In these cases the `make_directory` command can be used. As soon as CMake executes that command it will create the directory specified if it does not already exist. The `exec_program` command will execute a program when it is encountered by CMake. This is useful if you want to quickly convert a UNIX autoconf configured header file to CMake. Instead of doing the full conversion to CMake you could run configure from an `exec_program` command to generate the configured header file (on UNIX only of course).

7.4 Converting UNIX Makefiles

If your project is currently based on standard UNIX Makefiles (not autoconf and Makefile.in or imake) then their conversion to CMake should be fairly straightforward. Essentially for every directory in your project that has a Makefile you will create a matching CMakeLists file. How you handle multiple Makefiles in a directory really depends on their function. If the additional Makefiles (or Makefile type files) are simply included in the main Makefile then you can create matching CMake syntax files and include them into your main CMakeLists file in a similar manner. If the different Makefiles are meant to be invoked on the command line for different situations then you should consider creating a main CMakeLists file that uses some logic to choose which one to `include` based on a CMake option.

Converting the Makefile syntax to CMake is fairly easy. Frequently Makefiles have a list of object files to compile. These can be converted to CMake variables as follows:

```
OBJS= \
   foo1.o \
   foo2.o \
   foo3.o
```

becomes

```
set (SOURCES
  foo1.c
  foo2.c
  foo3.c
)
```

While the object files are typically listed in a Makefile, in CMake the focus is on the source files. If you used conditional statements in your Makefiles they can be converted over to CMake `if` commands. Since CMake handles generating dependencies, most dependencies or rules to generate dependencies can be eliminated. Where you have rules to build libraries or executables replace them with `add_library` or `add_executable` commands. Some UNIX build systems (and source code) make heavy use of the system architecture to determine what files to compile or what flags to use. Typically this information is stored in a Makefile variable called `ARCH` or `UNAME`.

The first choice in these cases is to replace the architecture specific code with a more generic test. For example, instead of switching your handling of byte order based on operating system, make the decision based on the results of a byte order test such as `CheckBigEndian.cmake`. With some software packages, there is too much architecture specific code for such a change to be reasonable, or you may want to make decisions based on architecture for other reasons. In those cases you can use the variables defined in the `CMakeDetermineSystem` module. They provide fairly detailed information on the operating system and version of the host computer.

7.5 Converting Autoconf Based Projects

Autoconf based projects primarily consist of three key pieces. The first is the configure.in file that drives the process. The second is Makefile.in which will become the resulting Makefile and the third piece is the remaining configured files that result from running configure. In converting an autoconf based project to CMake you will start with the configure.in and Makefile.in files.

The Makefile.in file can be converted to CMake syntax as explained in the preceding section on converting UNIX Makefiles. Once this has been done you need to convert the configure.in file into CMake syntax. Most functions (macros) in autoconf have corresponding commands in CMake. A short table of some of the basic conversions is listed below:

AC_ARG_WITH

use the `option` command

AC_CHECK_HEADER
: use the `CHECK_INCLUDE_FILE` macro from the `CheckIncludeFile` module

AC_MSG_CHECKING
: use the `message` command with the `STATUS` argument

AC_SUBST
: done automatically when using the `configure_file` command

AC_CHECK_LIB
: use the `CHECK_LIBRARY_EXISTS` macro from the `CheckLibraryExists` module

AC_CONFIG_SUBDIRS
: use the `add_subdirectory` command

AC_OUTPUT
: use the `configure_file` command

AC_TRY_COMPILE
: use the `try_compile` command

If your configure script performs test compilations using `AC_TRY_COMPILE` you can use the same code for CMake. You can either put it directly into your CMakeLists file if it is short, or preferably put it into a source code file for your project. We typically put such files into a CMake subdirectory for large projects that require such testing.

Where you are relying on autoconf to configure files you can use CMake's `configure_file` command. The basic approach is the same and we typically name input files to be configured with a `.in` extension just as autoconf does. This command replaces any variables in the input file referenced as `${VAR}` or `@VAR@` with their values as determined by CMake. If a variable is not defined, it will be replaced with nothing. Optionally, only variables of the form `@VAR@` will be replaced and `${VAR}` will be ignored. This is useful for configuring files for languages that use `${VAR}` as a syntax for evaluating variables. You can also conditionally define variables using the C pre processor by using `#cmakedefine VAR`. If the variable is defined then `configure_file` will convert the `#cmakedefine` into a `#define`, if it is not defined it will become a commented out `#undef`. For example:

```
/* what byte order is this system */
#cmakedefine CMAKE_WORDS_BIGENDIAN

/* what size is an INT */
#cmakedefine SIZEOF_INT @SIZEOF_INT@
```

7.6 Converting Windows Based Workspaces

To convert a Visual Studio workspace (or solution for Visual Studio .Net) to CMake involves a few steps. First you will need to create a CMakeLists file at the top of your source code directory. This file should start with a `project` command that defines the name of the project. This will become the name of the resulting workspace (or solution for Visual Studio .Net). Next you need to add all of your source files into CMake variables. For large projects that have multiple directories you can create a CMakeLists file in each directory as described previously in the section on source directory structures (page 115). You will then add your libraries and executables using `add_library` and `add_executable`. By default `add_executable` assumes that your executable is a console application. Adding the `WIN32` argument to `add_executable` indicates that it is a Windows application (using WinMain instead of main).

There are a few nice features that Visual Studio supports that CMake can take advantage of. One is support for class browsing. Typically in CMake only source files are added to a target, not header files. If you add header files to a target, they will show up in the workspace and then you will be able to browse them as usual. Visual Studio also supports the notion of groups of files. By default CMake creates groups for source files and header files. Using the `source_group` command you can create your own groups and assign files to them. If you have any custom build steps in your workspace, these can be added to your CMakeLists files using the `add_custom_command` command. Custom targets (Utility Targets) in Visual Studio can be added with the `add_custom_target` command.