Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 7

Below is a step-by-step tutorial covering common build system issues that CMake helps to address. Many of these topics have been introduced in *Mastering CMake* as separate issues but seeing how they all work together in an example project can be very helpful. This tutorial can be found in the Tests/Tutorial directory of the CMake source code tree. Each step has its own subdirectory containing a complete copy of the tutorial for that step

A Basic Starting Point (Step1)

The most basic project is an executable built from source code files. For simple projects a two line CMakeLists file is all that is required. This will be the starting point for our tutorial. The CMakeLists file looks like:

```
cmake_minimum_required (VERSION 2.6)
project (Tutorial)
add_executable(Tutorial tutorial.cxx)
```

Note that this example uses lower case commands in the CMakeLists file. Upper, lower, and mixed case commands are supported by CMake. The source code for tutorial.cxx will compute the square root of a number and the first version of it is very simple, as follows:

```
// A simple program that computes the square root of a number
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main (int argc, char *argv[])
{
  if (argc < 2)
    {
    fprintf(stdout,"Usage: %s number\n",argv[0]);
    return 1;
    }
  double inputValue = atof(argv[1]);
  double outputValue = sqrt(inputValue);
  fprintf(stdout,"The square root of %g is %g\n",
```

```
          inputValue, outputValue);
  return 0;
}
```

Adding a Version Number and Configured Header File

The first feature we will add is to provide our executable and project with a version number. While you can do this exclusively in the source code, doing it in the CMakeLists file provides more flexibility. To add a version number we modify the CMakeLists file as follows:

```
cmake_minimum_required (VERSION 2.6)
project (Tutorial)
# The version number.
set (Tutorial_VERSION_MAJOR 1)
set (Tutorial_VERSION_MINOR 0)

# configure a header file to pass some of the CMake settings
# to the source code
configure_file (
  "${PROJECT_SOURCE_DIR}/TutorialConfig.h.in"
  "${PROJECT_BINARY_DIR}/TutorialConfig.h"
  )

# add the binary tree to the search path for include files
# so that we will find TutorialConfig.h
include_directories("${PROJECT_BINARY_DIR}")

# add the executable
add_executable(Tutorial tutorial.cxx)
```

Since the configured file will be written into the binary tree we must add that directory to the list of paths to search for include files. We then create a TutorialConfig.h.in file in the source tree with the following contents:

```
// the configured options and settings for Tutorial
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
```

When CMake configures this header file the values for @Tutorial_VERSION_MAJOR@ and @Tutorial_VERSION_MINOR@ will be replaced by the values from the CMakeLists file. Next we modify tutorial.cxx to include the configured header file and to make use of the version numbers. The resulting source code is listed below.

```
// A simple program that computes the square root of a number
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "TutorialConfig.h"

int main (int argc, char *argv[])
```

```
{
  if (argc < 2)
    {
    fprintf(stdout,"%s Version %d.%d\n",
            argv[0],
            Tutorial_VERSION_MAJOR,
            Tutorial_VERSION_MINOR);
    fprintf(stdout,"Usage: %s number\n",argv[0]);
    return 1;
    }
  double inputValue = atof(argv[1]);
  double outputValue = sqrt(inputValue);
  fprintf(stdout,"The square root of %g is %g\n",
          inputValue, outputValue);
  return 0;
}
```

The main changes are the inclusion of the TutorialConfig.h header file and printing out a version number as part of the usage message.

Adding a Library (Step 2)

Now we will add a library to our project. This library will contain our own implementation for computing the square root of a number. The executable can then use this library instead of the standard square root function provided by the compiler. For this tutorial we will put the library into a subdirectory called MathFunctions. It will have the following one line CMakeLists file:

```
add_library(MathFunctions mysqrt.cxx)
```

The source file mysqrt.cxx has one function called mysqrt that provides similar functionality to the compiler's sqrt function. To make use of the new library we add an add_subdirectory call in the top level CMakeLists file so that the library will get built. We also add another include directory so that the MathFunctions/mysqrt.h header file can be found for the function prototype. The last change is to add the new library to the executable. The last few lines of the top level CMakeLists file now look like:

```
include_directories ("${PROJECT_SOURCE_DIR}/MathFunctions")
add_subdirectory (MathFunctions)

# add the executable
add_executable (Tutorial tutorial.cxx)
target_link_libraries (Tutorial MathFunctions)
```

Now let us consider making the MathFunctions library optional. In this tutorial there really isn't any reason to do so, but with larger libraries or libraries that rely on third party code you might want to. The first step is to add an option to the top level CMakeLists file.

```
# should we use our own math functions?
option (USE_MYMATH
        "Use tutorial provided math implementation" ON)
```

This will show up in the CMake GUI with a default value of ON that the user can change as desired. This setting will be stored in the cache so that the user does not need to keep setting it each time they run CMake on this project. The next change is to make the build and linking of the MathFunctions library conditional. To do this we change the end of the top level CMakeLists file to look like the following:

```
# add the MathFunctions library?
#
if (USE_MYMATH)
  include_directories ("${PROJECT_SOURCE_DIR}/MathFunctions")
  add_subdirectory (MathFunctions)
  set (EXTRA_LIBS ${EXTRA_LIBS} MathFunctions)
endif (USE_MYMATH)

# add the executable
add_executable (Tutorial tutorial.cxx)
target_link_libraries (Tutorial  ${EXTRA_LIBS})
```

This uses the setting of USE_MYMATH to determine if the MathFunctions should be compiled and used. Note the use of a variable (EXTRA_LIBS in this case) to collect up any optional libraries to later be linked into the executable. This is a common approach used to keep larger projects with many optional components clean. The corresponding changes to the source code are fairly straight forward and leave us with:

```
// A simple program that computes the square root of a number
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "TutorialConfig.h"
#ifdef USE_MYMATH
#include "MathFunctions.h"
#endif

int main (int argc, char *argv[])
{
  if (argc < 2)
    {
    fprintf(stdout,"%s Version %d.%d\n", argv[0],
            Tutorial_VERSION_MAJOR,
            Tutorial_VERSION_MINOR);
    fprintf(stdout,"Usage: %s number\n",argv[0]);
    return 1;
    }

  double inputValue = atof(argv[1]);

#ifdef USE_MYMATH
  double outputValue = mysqrt(inputValue);
#else
  double outputValue = sqrt(inputValue);
#endif
```

```
  fprintf(stdout,"The square root of %g is %g\n",
          inputValue, outputValue);
  return 0;
}
```

In the source code we make use of USE_MYMATH as well. This is provided from CMake to the source code through the TutorialConfig.h.in configured file by adding the following line to it:

```
#cmakedefine USE_MYMATH
```

Installing and Testing (Step 3)

For the next step we will add install rules and testing support to our project. The install rules are fairly straight forward. For the MathFunctions library we setup the library and the header file to be installed by adding the following two lines to MathFunctions' CMakeLists file:

```
install (TARGETS MathFunctions DESTINATION bin)
install (FILES MathFunctions.h DESTINATION include)
```

For the application the following lines are added to the top level CMakeLists file to install the executable and the configured header file:

```
# add the install targets
install (TARGETS Tutorial DESTINATION bin)
install (FILES "${PROJECT_BINARY_DIR}/TutorialConfig.h"
         DESTINATION include)
```

That is all there is to it. At this point you should be able to build the tutorial, then type make install (or build the INSTALL target from an IDE) and it will install the appropriate header files, libraries, and executables. The CMake variable CMAKE_INSTALL_PREFIX is used to determine the root of where the files will be installed. Adding testing is also a fairly straight forward process. At the end of the top level CMakeLists file we can add a number of basic tests to verify that the application is working correctly.

```
include(CTest)

# does the application run
add_test (TutorialRuns Tutorial 25)

# does it sqrt of 25
add_test (TutorialComp25 Tutorial 25)

set_tests_properties (TutorialComp25
  PROPERTIES PASS_REGULAR_EXPRESSION "25 is 5")

# does it handle negative numbers
add_test (TutorialNegative Tutorial -25)
```

```
set_tests_properties (TutorialNegative
  PROPERTIES PASS_REGULAR_EXPRESSION "-25 is 0")

# does it handle small numbers
add_test (TutorialSmall Tutorial 0.0001)
set_tests_properties (TutorialSmall
  PROPERTIES PASS_REGULAR_EXPRESSION "0.0001 is 0.01")

# does the usage message work?
add_test (TutorialUsage Tutorial)
set_tests_properties (TutorialUsage
  PROPERTIES
  PASS_REGULAR_EXPRESSION "Usage:.*number")
```

After building one may run the “ctest” command line tool to run the tests.

The first test simply verifies that the application runs, does not segfault or otherwise crash, and has a zero return value. This is the basic form of a CTest test. The next few tests all make use of the PASS_REGULAR_EXPRESSION test property to verify that the output of the test contains certain strings. In this case verifying that the computed square root is what it should be and that the usage message is printed when an incorrect number of arguments are provided. If you wanted to add a lot of tests to test different input values you might consider creating a macro like the following:

```
#define a macro to simplify adding tests, then use it
macro (do_test arg result)
  add_test (TutorialComp${arg} Tutorial ${arg})
  set_tests_properties (TutorialComp${arg}
    PROPERTIES PASS_REGULAR_EXPRESSION ${result})
endmacro (do_test)

# do a bunch of result based tests
do_test (25 "25 is 5")
do_test (-25 "-25 is 0")
```

For each invocation of do_test, another test is added to the project with a name, input, and results based on the passed arguments.

Adding System Introspection (Step 4)

Next let us consider adding some code to our project that depends on features the target platform may not have. For this example we will add some code that depends on whether or not the target platform has the log and exp functions. Of course almost every platform has these functions but for this tutorial assume that they are less common. If the platform has log then we will use that to compute the square root in the mysqrt function. We first test for the availability of these functions using the CheckFunctionExists.cmake macro in the top level CMakeLists file as follows:

```
# does this system provide the log and exp functions?
include (CheckFunctionExists)
```

```
check_function_exists (log HAVE_LOG)
check_function_exists (exp HAVE_EXP)
```

Next we modify the TutorialConfig.h.in to define those values if CMake found them on the platform as follows:

```
// does the platform provide exp and log functions?
#cmakedefine HAVE_LOG
#cmakedefine HAVE_EXP
```

It is important that the tests for log and exp are done before the configure_file command for TutorialConfig.h. The configure_file command immediately configures the file using the current settings in CMake. Finally in the mysqrt function we can provide an alternate implementation based on log and exp if they are available on the system using the following code:

```
// if we have both log and exp then use them
#if defined (HAVE_LOG) && defined (HAVE_EXP)
  result = exp(log(x)*0.5);
#else // otherwise use an iterative approach
  . . .
```

Adding a Generated File and Generator (Step 5)

In this section we will show how you can add a generated source file into the build process of an application. For this example we will create a table of precomputed square roots as part of the build process, and then compile that table into our application. To accomplish this we first need a program that will generate the table. In the MathFunctions subdirectory a new source file named MakeTable.cxx will do just that.

```
// A simple program that builds a sqrt table
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main (int argc, char *argv[])
{
  int i;
  double result;

  // make sure we have enough arguments
  if (argc < 2)
    {
    return 1;
    }

  // open the output file
  FILE *fout = fopen(argv[1],"w");
  if (!fout)
```

```
    {
    return 1;
    }

  // create a source file with a table of square roots
  fprintf(fout,"double sqrtTable[] = {\n");
  for (i = 0; i < 10; ++i)
    {
    result = sqrt(static_cast<double>(i));
    fprintf(fout,"%g,\n",result);
    }

  // close the table with a zero
  fprintf(fout,"0};\n");
  fclose(fout);
  return 0;
}
```

Note that the table is produced as valid C++ code and that the name of the file to write the output to is passed in as an argument. The next step is to add the appropriate commands to MathFunctions' CMakeLists file to build the MakeTable executable, and then run it as part of the build process. A few commands are needed to accomplish this, as shown below.

```
# first we add the executable that generates the table
add_executable(MakeTable MakeTable.cxx)

# add the command to generate the source code
add_custom_command (
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  COMMAND MakeTable ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  DEPENDS MakeTable
  )

# add the binary tree directory to the search path for
# include files
include_directories( ${CMAKE_CURRENT_BINARY_DIR} )

# add the main library
add_library(MathFunctions mysqrt.cxx ${CMAKE_CURRENT_BINARY_DIR}/Table.h  )
```

First the executable for MakeTable is added as any other executable would be added. Then we add a custom command that specifies how to produce Table.h by running MakeTable. Next we have to let CMake know that mysqrt.cxx depends on the generated file Table.h. This is done by adding the generated Table.h to the list of sources for the library MathFunctions. We also have to add the current binary directory to the list of include directories so that Table.h can be found and included by mysqrt.cxx. When this project is built it will first build the MakeTable executable. It will then run MakeTable to produce Table.h. Finally, it will compile mysqrt.cxx which includes Table.h to produce the MathFunctions library. At this point the top level CMakeLists file with all the features we have added looks like the following:

```
cmake_minimum_required (VERSION 2.6)
```

```
project (Tutorial)
include(CTest)

# The version number.
set (Tutorial_VERSION_MAJOR 1)
set (Tutorial_VERSION_MINOR 0)

# does this system provide the log and exp functions?
include (${CMAKE_ROOT}/Modules/CheckFunctionExists.cmake)

check_function_exists (log HAVE_LOG)
check_function_exists (exp HAVE_EXP)

# should we use our own math functions
option(USE_MYMATH
  "Use tutorial provided math implementation" ON)

# configure a header file to pass some of the CMake settings
# to the source code
configure_file (
  "${PROJECT_SOURCE_DIR}/TutorialConfig.h.in"
  "${PROJECT_BINARY_DIR}/TutorialConfig.h"
  )

# add the binary tree to the search path for include files
# so that we will find TutorialConfig.h
include_directories ("${PROJECT_BINARY_DIR}")

# add the MathFunctions library?
if (USE_MYMATH)
  include_directories ("${PROJECT_SOURCE_DIR}/MathFunctions")
  add_subdirectory (MathFunctions)
  set (EXTRA_LIBS ${EXTRA_LIBS} MathFunctions)
endif (USE_MYMATH)

# add the executable
add_executable (Tutorial tutorial.cxx)
target_link_libraries (Tutorial  ${EXTRA_LIBS})

# add the install targets
install (TARGETS Tutorial DESTINATION bin)
install (FILES "${PROJECT_BINARY_DIR}/TutorialConfig.h"
         DESTINATION include)

# does the application run
add_test (TutorialRuns Tutorial 25)

# does the usage message work?
add_test (TutorialUsage Tutorial)
set_tests_properties (TutorialUsage
  PROPERTIES
  PASS_REGULAR_EXPRESSION "Usage:.*number"
  )


#define a macro to simplify adding tests
macro (do_test arg result)
  add_test (TutorialComp${arg} Tutorial ${arg})
  set_tests_properties (TutorialComp${arg}
    PROPERTIES PASS_REGULAR_EXPRESSION ${result}
    )
endmacro (do_test)

# do a bunch of result based tests
do_test (4 "4 is 2")
```

```
do_test (9 "9 is 3")
do_test (5 "5 is 2.236")
do_test (7 "7 is 2.645")
do_test (25 "25 is 5")
do_test (-25 "-25 is 0")
do_test (0.0001 "0.0001 is 0.01")
```

TutorialConfig.h.in looks like:

```
// the configured options and settings for Tutorial
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
#cmakedefine USE_MYMATH

// does the platform provide exp and log functions?
#cmakedefine HAVE_LOG
#cmakedefine HAVE_EXP
```

And the CMakeLists file for MathFunctions looks like:

```
# first we add the executable that generates the table
add_executable(MakeTable MakeTable.cxx)
# add the command to generate the source code
add_custom_command (
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  DEPENDS MakeTable
  COMMAND MakeTable ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  )
# add the binary tree directory to the search path
# for include files
include_directories( ${CMAKE_CURRENT_BINARY_DIR} )

# add the main library
add_library(MathFunctions mysqrt.cxx ${CMAKE_CURRENT_BINARY_DIR}/Table.h)

install (TARGETS MathFunctions DESTINATION bin)
install (FILES MathFunctions.h DESTINATION include)
```

Building an Installer (Step 6)

Next suppose that we want to distribute our project to other people so that they can use it. We want to provide both binary and source distributions on a variety of platforms. This is a little different from the instal we did previously in section Installing and Testing (Step 3), where we were installing the binaries that we had built from the source code. In this example we will be building installation packages that support binary installations and package management features as found in cygwin, debian, RPMs etc. To accomplish this we will use CPack to create platform specific installers as described in Chapter Packaging with CPack. Specifically we need to add a few lines to the bottom of our toplevel CMakeLists.txt file.

```
# build a CPack driven installer package
include (InstallRequiredSystemLibraries)
set (CPACK_RESOURCE_FILE_LICENSE
     "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
set (CPACK_PACKAGE_VERSION_MAJOR "${Tutorial_VERSION_MAJOR}")
set (CPACK_PACKAGE_VERSION_MINOR "${Tutorial_VERSION_MINOR}")
include (CPack)
```

That is all there is to it. We start by including InstallRequiredSystemLibraries. This module will include any runtime libraries that are needed by the project for the current platform. Next we set some CPack variables to where we have stored the license and version information for this project. The version information makes use of the variables we set earlier in this tutorial. Finally we include the CPack module which will use these variables and some other properties of the system you are on to setup an installer.

The next step is to build the project in the usual manner and then run CPack on it. To build a binary distribution you would run:

```
cpack --config CPackConfig.cmake
```

To create a source distribution you would type

```
cpack --config CPackSourceConfig.cmake
```

Adding Support for a Dashboard (Step 7)

Adding support for submitting our test results to a dashboard is very easy. We already defined a number of tests for our project in the earlier steps of this tutorial. We just have to run those tests and submit them to a dashboard. To include support for dashboards we include the CTest module in our toplevel CMakeLists file.

```
# enable dashboard scripting
include (CTest)
```

We also create a CTestConfig.cmake file where we can specify the name of this project for the dashboard.

```
set (CTEST_PROJECT_NAME "Tutorial")
```

CTest will read in this file when it runs. To create a simple dashboard you can run CMake on your project, change directory to the binary tree, and then run ctest –D Experimental. The results of your dashboard will be uploaded to Kitware's public dashboard here.