

CMake tutorial

- The remainder is a step-by-step tutorial covering common build system use cases that CMake helps to address:
 - Basic starting point
 - Adding a library
 - Installing and testing
 - Adding system introspection
 - Adding a generated file and generator
 - Building an installer

Basic starting point

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

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

Basic starting point

- The source code for `tutorial1.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;
}
```

Basic starting point

- Adding a version number and configured header file:
 - We add a feature to provide our executable and project with a version number
 - doing it inside `CMakeLists.txt` provides more flexibility
 - To add a version number we modify the `CMakeLists.txt` 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)
```

Basic starting point

- 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 content:

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

Basic starting point

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

Basic starting point

```
// 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;
}
```

Adding a library

- We will add a library that contains 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 the tutorial we will put the library into a subdirectory called `MathFunctions`. It will have the following one line `CMakeLists.txt` file:

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


Adding a library

- The source file `mysqrt.cxx` has one function called `mysqrt` that provides similar function to the compiler's
- We add an `add_subdirectory` call in the top level `CMakeLists.txt` file so that the library will get built
- We also add another include directory so that the `MathFunctions/MathFunctions.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.txt` file now look like:

Adding a library

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

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

Adding a library

- Now let us consider making the `MathFunctions` library optional. In larger libraries or libraries that rely on third party code we might need it. The first step is to add an option to the top level `CMakeLists.txt` file
- The option 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

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

Adding a library

- 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.txt` 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})
```

Adding a library

- 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:

Adding a library

- 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` configure file by adding the following line to it:

```
#cmakedefine USE_MYMATH
```

Adding a library

```
// 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;
}
```

Installing and testing

- 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.txt** file:

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


Installing and testing

- For the application, the following lines are added to the top level `CMakeLists.txt` 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)
```

Installing and testing

- At this point you should be able to build the tutorial, then type `make install` 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 straightforward process: At the end of the top level `CMakeLists.txt` file, we can add a number of basic tests to verify that the application is working correctly.

Installing and testing

```
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")
```

Installing and testing

- After building, run the `ctest`.
- First test simply verifies that the application runs, does not segfault or otherwise crash, and has a zero return value
- Next few tests make use of the `PASS_REGULAR_EXPRESSION` test property to verify that the output of the test contains certain strings
- If we wanted to add a lot of tests to test different input values, we might consider creating a `macro` like the following:

Installing and testing

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

```
#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")
```

Adding system introspection

- We will add some code that depends on whether or not the target platform has the `log` and `exp` functions
- 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.txt` 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)
```

Adding system introspection

- Next we modify `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
```

Adding system introspection

- Tests for `log` and `exp` should be done before the `configure_file` command for `TutorialConfig.h`.
- The `configure_file` command configures the file using the current settings in CMake.
- `mysqrt` function we can provide an alternate implementation based on `log` and `exp` if they are available on the system:

```
// 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 generated file and generator

- We will create a table of precomputed square roots as part of the build process
- 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.

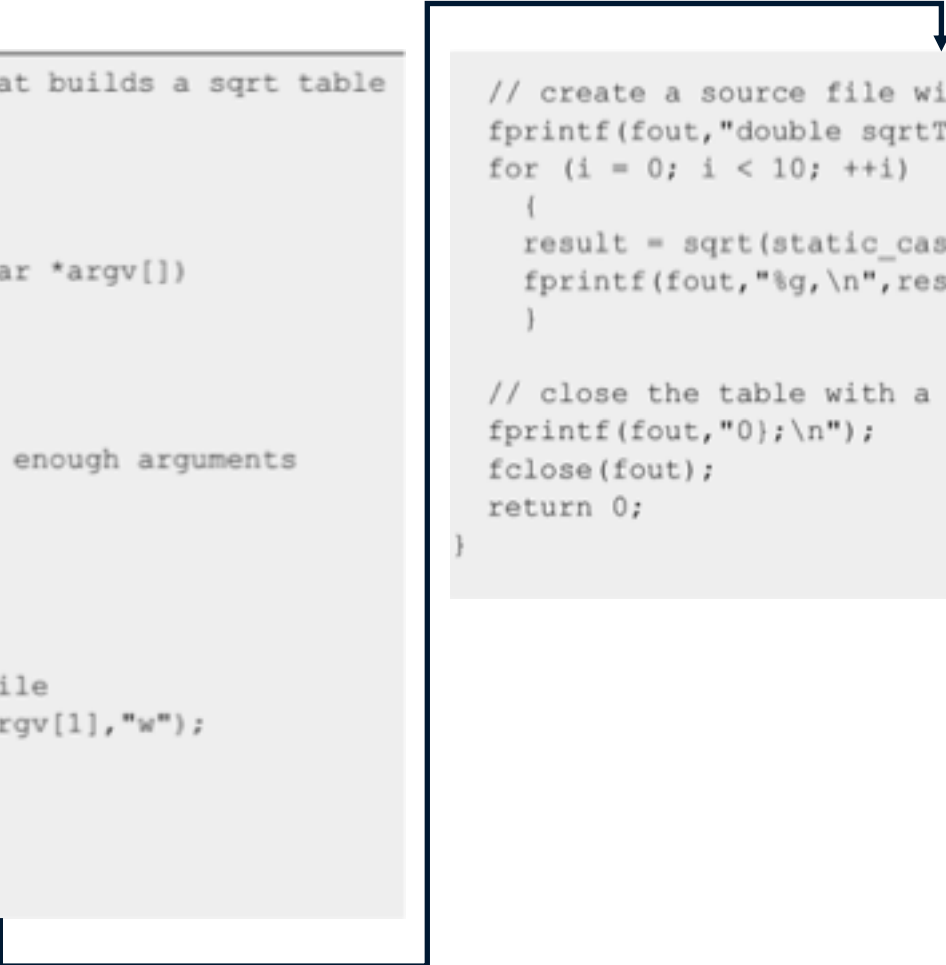
Adding generated file and generator

```
// 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;
}
```

Adding generated file and generator

- 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.txt` 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.

Adding generated file and generator

```
# 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 )
```

Adding generated file and generator

- 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`.

Adding generated file and generator

- 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.txt` file with all the features we have added looks like the following:

Adding generated file and generator

```
cmake_minimum_required (VERSION 2.6)
project (Tutorial)
include(CTest)
```

Basic starting point

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

Adding a version

```
# 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)
```

Adding system introspection

```
# 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"
)
```

Adding configured header file

```
# 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)
```

Adding MathFunctions library

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

Adding generated file and generator

```
# 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")
```

install the executable
and the configured
header file

add a number of basic
tests

creating a macro to add
a lot of tests

Adding generated file and generator

- `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@
#definecmakedefine USE_MYMATH

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

Adding generated file and generator

- And the `CMakeLists.txt` 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

- We want to:
 - Distribute our project to others so that they can use it
 - Provide both binary and source distributions on a variety of platforms
- We will build installation packages that support binary installations and package management features as found in `cygwin`, `debian`, `RPMs` etc.

Building an installer

- Use `CPack` to create platform specific installers
- The toplevel `CMakeLists.txt` file will be:

```
# 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)
```

Building an installer

- 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

Building 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
```