CMake Tutorial

Contents

The CMake tutorial provides a step-by-step guide that covers common build system issues that CMake helps address. Seeing how various topics all work together in an example project can be very helpful. The tutorial documentation and source code for examples can be found in the `Help/guide/tutorial` directory of the CMake source code tree. Each step has its own subdirectory containing code that may be used as a starting point. The tutorial examples are progressive so that each step provides the complete solution for the previous step.

A Basic Starting Point (Step 1)

The most basic project is an executable built from source code files. For simple projects, a three line CMakeLists file is all that is required. This will be the starting point for our tutorial. Create a `CMakeLists.txt` file in the `Step1` directory that looks like:

```
cmake_minimum_required(VERSION 3.10)

# set the project name
project(Tutorial)

# add the executable
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` is provided in the

Step1 directory and can be used to compute the square root of a number.

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 we could do this exclusively in the source code, using CMakeLists provides more flexibility.

First, modify the CMakeLists file to set the version number.

```
cmake_minimum_required(VERSION 3.10)

# set the project name and version
project(Tutorial VERSION 1.0)
```

Then, configure a header file to pass the version number to the source code:

```
configure_file(TutorialConfig.h.in TutorialConfig.h)
```

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. Add the following lines to the end of the CMakeLists file:

```
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                           )
```

Using your favorite editor, create `TutorialConfig.h.in` in the source directory 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.

Next modify `tutorial.cxx` to include the configured header file, `TutorialConfig.h`.

Finally, let's print out the version number by updating `tutorial.cxx` as follows:

```
  if (argc < 2) {
    // report version
    std::cout << argv[0] << " Version " << Tutorial_VERSION_MAJOR << "."
              << Tutorial_VERSION_MINOR << std::endl;
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
    return 1;
  }
```

Specify the C++ Standard

Next let's add some C++11 features to our project by replacing `atof` with `std::stod` in `tutorial.cxx`. At the same time, remove `#include <cstdlib>`.

```
  const double inputValue = std::stod(argv[1]);
```

We will need to explicitly state in the CMake code that it should use the correct flags. The easiest way to enable support for a specific C++ standard in CMake is by using the `CMAKE_CXX_STANDARD` variable. For this tutorial, set the `CMAKE_CXX_STANDARD` variable in the CMakeLists file to 14.

```
cmake_minimum_required(VERSION 3.10)

# set the project name and version
project(Tutorial VERSION 1.0)

# specify the C++ standard
set(CMAKE_CXX_STANDARD 14)
```

If our compiler does not support C++14, CMake will fallback to C++11, if supported. Which variable can we set in the CMakeLists file to disable this decay behavior and treat the `CMAKE_CXX_STANDARD` value as a requirement?

Build and Test

Run **cmake** or **cmake-gui** to configure the project and then build it with your chosen build tool.

For example, from the command line we could navigate to the `Help/guide/tutorial` directory of the CMake source code tree and run the following commands:

```
mkdir Step1_build
cd Step1_build
cmake ../Step1
cmake --build
```

Navigate to the directory where Tutorial was built (likely the make directory or a Debug or Release build configuration subdirectory) and run these commands:

```
Tutorial 4294967296
Tutorial 10
Tutorial
```

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. This directory already contains a header file, `MathFunctions.h`, and a source file `mysqrt.cxx`. The source file has one function called `mysqrt` that provides similar functionality to the compiler's `sqrt` function.

Add the following one line `CMakeLists.txt` file to the MathFunctions directory:

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

To make use of the new library we will add an `add_subdirectory` call in the top-level CMakeLists file so that the library will get built. We add the new library to the executable, and add MathFunctions as an include directory so that the `mqsqrt.h` header file can be found. The last few lines of the top-level CMakeLists file should now look like:

```
# add the MathFunctions library
add_subdirectory(MathFunctions)

# add the executable
add_executable(Tutorial tutorial.cxx)

target_link_libraries(Tutorial PUBLIC MathFunctions)

# add the binary tree to the search path for include files
# so that we will find TutorialConfig.h
target_include_directories(Tutorial PUBLIC
                          "${PROJECT_BINARY_DIR}"
                          "${PROJECT_SOURCE_DIR}/MathFunctions"
                          )
```

Now let us make the MathFunctions library optional. While for the tutorial there really isn't any need to do so, for larger projects this is a common occurrence. The first step is to add an option to the top-level CMakeLists file.

```
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(TutorialConfig.h.in TutorialConfig.h)
```

Exercise: Why is it important that we configure `TutorialConfig.h.in` after the option for `USE_MYMATH`? What would happen if we inverted the two?

This option will be displayed in the CMake GUI and ccmake with a default value of ON that can be changed by the user. This setting will be stored in the cache so that the user does not need to set the value each time they run CMake on a build directory.

The next change is to make building and linking the MathFunctions library conditional. To do this we change the end of the top-level CMakeLists file to look like the following:

```
if(USE_MYMATH)
  add_subdirectory(MathFunctions)
  list(APPEND EXTRA_LIBS MathFunctions)
  list(APPEND EXTRA_INCLUDES "${PROJECT_SOURCE_DIR}/MathFunctions")
endif()

# add the executable
add_executable(Tutorial tutorial.cxx)

target_link_libraries(Tutorial PUBLIC ${EXTRA_LIBS})

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

Note the use of the variable `EXTRA_LIBS` to collect up any optional libraries to later be linked into the executable. The variable `EXTRA_INCLUDES` is used similarly for optional header files. This is a classic approach when dealing with many optional components, we will cover the modern approach in the next step.

The corresponding changes to the source code are fairly straightforward. First, in `tutorial.cxx`, include the MathFunctions header if we need it:

```
#ifdef USE_MYMATH
#include "MathFunctions.h"
#endif
```

Then, in the same file, make which square root function is used dependent on `USE_MYMATH`:

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

Since the source code now requires `USE_MYMATH` we can add it to `TutorialConfig.h.in` with the following line:

```
#cmakedefine USE_MYMATH
```

Run **cmake** or **cmake-gui** to configure the project and then build it with your chosen build tool. Then run the built Tutorial executable.

Use ccmake or the CMake GUI to update the value of `USE_MYMATH`. Rebuild and run the tutorial again. Which function gives better results, sqrt or mysqrt?

Adding Usage Requirements for Library (Step 3)

Usage requirements allow for far better control over a library or executable's link and include line while also giving more control over the transitive property of targets inside CMake. The primary commands that leverage usage requirements are:

- `target_compile_definitions`
- `target_compile_options`
- `target_include_directories`
- `target_link_libraries`

Let's refactor our code from Adding a Library (Step 2) to use the modern CMake approach of usage requirements. We first state that anybody linking to MathFunctions needs to include the current source directory, while MathFunctions itself doesn't. So this can become an `INTERFACE` usage requirement.

Remember `INTERFACE` means things that consumers require but the producer doesn't. Add the following lines to the end of `MathFunctions/CMakeLists.txt`:

```
target_include_directories(MathFunctions
          INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
          )
```

Now that we've specified usage requirements for MathFunctions we can safely remove our uses of the `EXTRA_INCLUDES` variable from the top-level CMakeLists, here:

```
if(USE_MYMATH)
  add_subdirectory(MathFunctions)
  list(APPEND EXTRA_LIBS MathFunctions)
endif()
```

And here:

```
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                           )
```

Once this is done, run **cmake** or **cmake-gui** to configure the project and then build it with your chosen build tool or by using `cmake --build .` from the build directory.

Installing and Testing (Step 4)

Now we can start adding install rules and testing support to our project.

Install Rules

The install rules are fairly simple: for MathFunctions we want to install the library and header file and for the application we want to install the executable and configured header.

So to the end of `MathFunctions/CMakeLists.txt` we add:

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

And to the end of the top-level `CMakeLists.txt` we add:

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

That is all that is needed to create a basic local install of the tutorial.

Run **cmake** or **cmake-gui** to configure the project and then build it with your chosen build tool. Run the install step by typing `cmake --install .` from the command line, or build the `INSTALL` target from an IDE. This 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. If using `cmake --install` a custom installation directory can be given via `--prefix` argument.

Verify that the installed Tutorial runs.

Testing Support

Next let’s test our application. At the end of the top-level CMakeLists file we can enable testing and then add a number of basic tests to verify that the application is working correctly.

```
enable_testing()

# does the application run
add_test(NAME Runs COMMAND Tutorial 25)

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

# define a function to simplify adding tests
function(do_test target arg result)
  add_test(NAME Comp${arg} COMMAND ${target} ${arg})
  set_tests_properties(Comp${arg}
    PROPERTIES PASS_REGULAR_EXPRESSION ${result}
    )
endfunction(do_test)

# do a bunch of result based tests
do_test(Tutorial 4 "4 is 2")
do_test(Tutorial 9 "9 is 3")
do_test(Tutorial 5 "5 is 2.236")
do_test(Tutorial 7 "7 is 2.645")
do_test(Tutorial 25 "25 is 5")
do_test(Tutorial -25 "-25 is [-nan|nan|0]")
do_test(Tutorial 0.0001 "0.0001 is 0.01")
```

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 test makes 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 the usage message is printed when an incorrect number of arguments are provided.

Lastly, we have a function called `do_test` that runs the application and verifies that the computed square root is correct for given input. For each invocation of `do_test`, another test is added to the project with a name, input, and expected results based on the passed arguments.

Rebuild the application and then cd to the binary directory and run `ctest -N` and `ctest -VV`. For multi-config generators (e.g. Visual Studio), the configuration type must be specified. To run tests in Debug mode, for example, use `ctest -C Debug -VV` from the build directory (not the Debug subdirectory!). Alternatively, build the `RUN_TESTS` target from the IDE.

Adding System Introspection (Step 5)

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 not common.

If the platform has `log` and `exp` then we will use them to compute the square root in the `mysqrt` function. We first test for the availability of these functions using the `CheckSymbolExists.cmake` macro in the top-level CMakeLists. We're going to use the new defines in `TutorialConfig.h.in`, so be sure to set them before that file is configured.

```
include(CheckSymbolExists)
set(CMAKE_REQUIRED_LIBRARIES "m")
check_symbol_exists(log "math.h" HAVE_LOG)
check_symbol_exists(exp "math.h" HAVE_EXP)
```

Now let's add these defines to `TutorialConfig.h.in` so that we can use them from `mysqrt.cxx`:

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

Modify `mysqrt.cxx` to include cmath. Next, in that same file 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 (don't forget the `#endif` before returning the result!):

```
#if defined(HAVE_LOG) && defined(HAVE_EXP)
  double result = exp(log(x) * 0.5);
  std::cout << "Computing sqrt of " << x << " to be " << result
            << " using log and exp" << std::endl;
#else
  double result = x;
```

Run **cmake** or **cmake-gui** to configure the project and then build it with your chosen build tool and run the Tutorial executable.

You will notice that we're not using `log` and `exp`, even if we think they should be available. We should realize quickly that we have forgotten to include `TutorialConfig.h` in `mysqrt.cxx`.

We will also need to update MathFunctions/CMakeLists so `mysqrt.cxx` knows where this file is located:

```
target_include_directories(MathFunctions
          INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
          PRIVATE ${CMAKE_BINARY_DIR}
          )
```

After making this update, go ahead and build the project again and run the built Tutorial executable. If `log` and `exp` are still not being used, open the generated `TutorialConfig.h` file from the build directory. Maybe they aren't available on the current system?

Which function gives better results now, sqrt or mysqrt?

Specify Compile Definition

Is there a better place for us to save the HAVE_LOG and HAVE_EXP values other than in TutorialConfig.h? Let's try to use target_compile_definitions.

First, remove the defines from TutorialConfig.h.in. We no longer need to include TutorialConfig.h from mysqrt.cxx or the extra include in MathFunctions/CMakeLists.

Next, we can move the check for HAVE_LOG and HAVE_EXP to MathFunctions/CMakeLists and then add specify those values as PRIVATE compile definitions.

```
include(CheckSymbolExists)
set(CMAKE_REQUIRED_LIBRARIES "m")
check_symbol_exists(log "math.h" HAVE_LOG)
check_symbol_exists(exp "math.h" HAVE_EXP)

if(HAVE_LOG AND HAVE_EXP)
  target_compile_definitions(MathFunctions
                             PRIVATE "HAVE_LOG" "HAVE_EXP")
endif()
```

After making these updates, go ahead and build the project again. Run the built Tutorial executable and verify that the results are same as earlier in this step.

Adding a Custom Command and Generated File (Step 6)

Suppose, for the purpose of this tutorial, we decide that we never want to use the platform log and exp functions and instead would like to generate a table of precomputed values to use in the mysqrt function. In this section, we will create the table as part of the build process, and then compile that table into our application.

First, let's remove the check for the log and exp functions in MathFunctions/CMakeLists. Then remove the check for HAVE_LOG and HAVE_EXP from mysqrt.cxx. At the same time, we can remove #include <cmath>.

In the MathFunctions subdirectory, a new source file named MakeTable.cxx has been provided to generate the table.

After reviewing the file, we can see that the table is produced as valid C++ code and that the output filename 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.

First, at the top of MathFunctions/CMakeLists, the executable for MakeTable is added as any other executable would be added.

```
add_executable(MakeTable MakeTable.cxx)
```

Then we add a custom command that specifies how to produce `Table.h` by running MakeTable.

```
add_custom_command(
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  COMMAND MakeTable ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  DEPENDS 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.

```
add_library(MathFunctions
            mysqrt.cxx
            ${CMAKE_CURRENT_BINARY_DIR}/Table.h
            )
```

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`.

```
target_include_directories(MathFunctions
          INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
          PRIVATE ${CMAKE_CURRENT_BINARY_DIR}
          )
```

Now let's use the generated table. First, modify `mysqrt.cxx` to include `Table.h`. Next, we can rewrite the mysqrt function to use the table:

```
double mysqrt(double x)
{
  if (x <= 0) {
    return 0;
  }

  // use the table to help find an initial value
  double result = x;
  if (x >= 1 && x < 10) {
    std::cout << "Use the table to help find an initial value " << std::endl;
    result = sqrtTable[static_cast<int>(x)];
  }

  // do ten iterations
  for (int i = 0; i < 10; ++i) {
    if (result <= 0) {
      result = 0.1;
    }
    double delta = x - (result * result);
    result = result + 0.5 * delta / result;
    std::cout << "Computing sqrt of " << x << " to be " << result << std::endl;
  }

  return result;
}
```

Run **cmake** or **cmake-gui** to configure the project and then build it with your chosen build tool.

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.

Run the Tutorial executable and verify that it is using the table.

Building an Installer (Step 7)

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 install we did previously in Installing and Testing (Step 4) , 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. To accomplish this we will use CPack to create platform specific installers. Specifically we need to add a few lines to the bottom of our top-level `CMakeLists.txt` file.

```
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 was set earlier in this tutorial and the `license.txt` has been included in the top-level source directory for this step.

Finally we include the CPack module which will use these variables and some other properties of the current system 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, from the binary directory run:

```
cpack
```

To create a source distribution you would type:

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

Alternatively, run `make package` or right click the `Package` target and `Build Project` from an IDE.

Run the installer found in the binary directory. Then run the installed executable and verify that it works.

Adding Support for a Dashboard (Step 8)

Adding support for submitting our test results to a dashboard is very easy. We already defined a number of tests for our project in Testing Support. Now 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 top-level `CMakeLists.txt`.

Replace:

```
# enable testing
enable_testing()
```

With:

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

The CTest module will automatically call `enable_testing()`, so we can remove it from our CMake files.

We will also need to create a `CTestConfig.cmake` file in the top-level directory where we can specify the name of the project and where to submit the dashboard.

```
## This file should be placed in the root directory of your project.
## Then modify the CMakeLists.txt file in the root directory of your
## project to incorporate the testing dashboard.
##
## # The following are required to submit to the CDash dashboard:
##   ENABLE_TESTING()
##   INCLUDE(CTest)

set(CTEST_PROJECT_NAME "CMakeTutorial")
set(CTEST_NIGHTLY_START_TIME "00:00:00 EST")

set(CTEST_DROP_METHOD "http")
set(CTEST_DROP_SITE "my.cdash.org")
set(CTEST_DROP_LOCATION "/submit.php?project=CMakeTutorial")
set(CTEST_DROP_SITE_CDASH TRUE)
```

CTest will read in this file when it runs. To create a simple dashboard you can run **cmake** or **cmake-gui** to configure the project, but do not build it yet. Instead, change directory to the binary tree, and then run:

```
'ctest [-VV] –D Experimental'
```

Remember, for multi-config generators (e.g. Visual Studio), the configuration type must be specified:

```
'ctest [-VV] -C Debug –D Experimental'
```

Or, from an IDE, build the `Experimental` target.

Ctest will build and test the project and submit the results to the Kitware public dashboard. The results of your dashboard will be uploaded to Kitware's public dashboard here: https://my.cdash.org/index.php?project=CMakeTutorial.

Mixing Static and Shared (Step 9)

In this section we will show how by using the `BUILD_SHARED_LIBS` variable we can control the default behavior of `add_library`, and allow control over how libraries without an explicit type (STATIC/SHARED/MODULE/OBJECT) are built.

To accomplish this we need to add `BUILD_SHARED_LIBS` to the top-level `CMakeLists.txt`. We use the `option` command as it allows users to optionally select if the value should be On or Off.

Next we are going to refactor MathFunctions to become a real library that encapsulates using `mysqrt` or `sqrt`, instead of requiring the calling code to do this logic. This will also mean that `USE_MYMATH` will not control building MathFuctions, but instead will control the behavior of this library.

The first step is to update the starting section of the top-level `CMakeLists.txt` to look like:

```
cmake_minimum_required(VERSION 3.10)

# set the project name and version
project(Tutorial VERSION 1.0)

# specify the C++ standard
set(CMAKE_CXX_STANDARD 14)

# control where the static and shared libraries are built so that on windows
# we don't need to tinker with the path to run the executable
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")

option(BUILD_SHARED_LIBS "Build using shared libraries" ON)

# configure a header file to pass the version number only
configure_file(TutorialConfig.h.in TutorialConfig.h)

# add the MathFunctions library
add_subdirectory(MathFunctions)

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

Now that we have made MathFunctions always be used, we will need to update the logic of that library. So, in `MathFunctions/CMakeLists.txt` we need to create a SqrtLibrary that will conditionally be built when `USE_MYMATH` is enabled. Now, since this is a tutorial, we are going to explicitly require that SqrtLibrary is built statically.

The end result is that `MathFunctions/CMakeLists.txt` should look like:

```
# add the library that runs
add_library(MathFunctions MathFunctions.cxx)

# state that anybody linking to us needs to include the current source dir
# to find MathFunctions.h, while we don't.
target_include_directories(MathFunctions
                           INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
                           )

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

```
if(USE_MYMATH)

  # 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
    )

  # library that just does sqrt
  add_library(SqrtLibrary STATIC
              mysqrt.cxx
              ${CMAKE_CURRENT_BINARY_DIR}/Table.h
              )

  # state that we depend on our binary dir to find Table.h
  target_include_directories(SqrtLibrary PRIVATE
                             ${CMAKE_CURRENT_BINARY_DIR}
                             )

  target_link_libraries(MathFunctions PRIVATE SqrtLibrary)
endif()

# define the symbol stating we are using the declspec(dllexport) when
# building on windows
target_compile_definitions(MathFunctions PRIVATE "EXPORTING_MYMATH")

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

Next, update `MathFunctions/mysqrt.cxx` to use the `mathfunctions` and `detail` namespaces:

```
#include "MathFunctions.h"
#include <iostream>

// include the generated table
#include "Table.h"

namespace mathfunctions {
namespace detail {
// a hack square root calculation using simple operations
double mysqrt(double x)
{
  if (x <= 0) {
    return 0;
  }

  // use the table to help find an initial value
  double result = x;
  if (x >= 1 && x < 10) {
    std::cout << "Use the table to help find an initial value " << std::endl;
    result = sqrtTable[static_cast<int>(x)];
  }

  // do ten iterations
  for (int i = 0; i < 10; ++i) {
    if (result <= 0) {
      result = 0.1;
    }
    double delta = x - (result * result);
    result = result + 0.5 * delta / result;
    std::cout << "Computing sqrt of " << x << " to be " << result << std::endl;
  }

  return result;
}
}
}
```

We also need to make some changes in `tutorial.cxx`, so that it no longer uses `USE_MYMATH`:

1. Always include `MathFunctions.h`
2. Always use `mathfunctions::sqrt`

Finally, update `MathFunctions/MathFunctions.h` to use dll export defines:

```
#if defined(_WIN32)
#  if defined(EXPORTING_MYMATH)
#    define DECLSPEC __declspec(dllexport)
#  else
#    define DECLSPEC __declspec(dllimport)
#  endif
#else // non windows
#  define DECLSPEC
#endif

namespace mathfunctions {
double DECLSPEC sqrt(double x);
}
```

At this point, if you build everything, you will notice that linking fails as we are combining a static library without position enabled code with a library that has position enabled code. The solution to this is to explicitly set the `POSITION_INDEPENDENT_CODE` target property of SqrtLibrary to be True no matter the build type.

Exercise: We modified `MathFunctions.h` to use dll export defines. Using CMake documentation can you find a helper module to simplify this?

Adding Generator Expressions (Step 10)

Generator expressions are evaluated during build system generation to produce information specific to each build configuration.

Generator expressions are allowed in the context of many target properties, such as `LINK_LIBRARIES`, `INCLUDE_DIRECTORIES`, `COMPILE_DEFINITIONS` and others. They may also be used when using commands to populate those properties, such as `target_link_libraries()`, `target_include_directories()`, `target_compile_definitions()` and others.

Generator expressions may be used to enable conditional linking, conditional definitions used when compiling, conditional include directories and more. The conditions may be based on the build configuration, target properties, platform information or any other queryable information.

There are different types of generator expressions including Logical, Informational, and Output expressions.

Logical expressions are used to create conditional output. The basic expressions are the 0 and 1 expressions. A `$<0:...>` results in the empty string, and `<1:...>` results in the content of "…". They can also be nested.

A common usage of generator expressions is to conditionally add compiler flags, such as those as language levels or warnings. A nice pattern is to associate this information to an `INTERFACE` target allowing this information to propagate. Lets start by constructing an `INTERFACE` target and specifying the required C++ standard level of `14` instead of using `CMAKE_CXX_STANDARD`.

So the following code:

```
# specify the C++ standard
set(CMAKE_CXX_STANDARD 14)
```

Would be replaced with:

```
add_library(tutorial_compiler_flags INTERFACE)
target_compile_features(tutorial_compiler_flags INTERFACE cxx_std_14)
```

Next we add the desired compiler warning flags that we want for our project. As warning flags vary based on the compiler we use the `COMPILE_LANG_AND_ID` generator expression to control which flags to apply given a language and a set of compiler ids as seen below:

```
set(gcc_like_cxx "$<COMPILE_LANG_AND_ID:CXX,ARMClang,AppleClang,Clang,GNU>")
set(msvc_cxx "$<COMPILE_LANG_AND_ID:CXX,MSVC>")
target_compile_options(tutorial_compiler_flags INTERFACE
  "$<${gcc_like_cxx}:$<BUILD_INTERFACE:-Wall;-Wextra;-Wshadow;-Wodr;-Wformat=2;-Wunused>>"
  "$<${msvc_cxx}:$<BUILD_INTERFACE:-W3>>"
)
```

Looking at this we see that the warning flags are encapsulated inside a `BUILD_INTERFACE` condition. This is done so that consumers of our installed project will not inherit our warning flags.

Exercise: Modify `MathFunctions/CMakeLists.txt` so that all targets have a `target_link_libraries()` call to `tutorial_compiler_flags`.

Adding Export Configuration (Step 11)

During Installing and Testing (Step 4) of the tutorial we added the ability for CMake to install the library and headers of the project. During Building an Installer (Step 7) we added the ability to package up this information so it could be distributed to other people.

The next step is to add the necessary information so that other CMake projects can use our project, be it from a build directory, a local install or when packaged.

The first step is to update our `install(TARGETS)` commands to not only specify a `DESTINATION` but also an `EXPORT`. The `EXPORT` keyword generates and installs a CMake file containing code to import all targets listed in the install command from the installation tree. So let's go ahead and explicitly `EXPORT` the MathFunctions library by updating the `install` command in `MathFunctions/CMakeLists.txt` to look like:

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

Now that we have MathFunctions being exported, we also need to explicitly install the generated `MathFunctionsTargets.cmake` file. This is done by adding the following to the bottom of the top-level `CMakeLists.txt`:

At this point you should try and run CMake. If everything is setup properly you will see that CMake will generate an error that looks like:

```
Target "MathFunctions" INTERFACE_INCLUDE_DIRECTORIES property contains
path:

  "/Users/robert/Documents/CMakeClass/Tutorial/Step11/MathFunctions"

which is prefixed in the source directory.
```

What CMake is trying to say is that during generating the export information it will export a path that is intrinsically tied to the current machine and will not be valid on other machines. The solution to this is to update the MathFunctions `target_include_directories` to understand that it needs different `INTERFACE` locations when being used from within the build directory and from an install / package. This means converting the `target_include_directories` call for MathFunctions to look like:

```
target_include_directories(MathFunctions
                           INTERFACE
                            $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}>
                            $<INSTALL_INTERFACE:include>
                           )
```

Once this has been updated, we can re-run CMake and see verify that it doesn’t warn anymore.

At this point, we have CMake properly packaging the target information that is required but we will still need to generate a `MathFunctionsConfig.cmake` so that the CMake `find_package command` can find our project. So let’s go ahead and add a new file to the top-level of the project called `Config.cmake.in` with the following contents:

```
@PACKAGE_INIT@

include ( "${CMAKE_CURRENT_LIST_DIR}/MathFunctionsTargets.cmake" )
```

Then, to properly configure and install that file, add the following to the bottom of the top-level CMakeLists:

At this point, we have generated a relocatable CMake Configuration for our project that can be used after the project has been installed or packaged. If we want our project to also be used from a build directory we only have to add the following to the bottom of the top level CMakeLists:

With this export call we now generate a `Targets.cmake`, allowing the configured `MathFunctionsConfig.cmake` in the build directory to be used by other projects, without needing it to be installed.

Import a CMake Project (Consumer)

This examples shows how a project can find other CMake packages that generate `Config.cmake` files.

It also shows how to state a project's external dependencies when generating a `Config.cmake`.

Packaging Debug and Release (MultiPackage)

By default CMake is model is that a build directory only contains a single configuration, be it Debug, Release, MinSizeRel, or RelWithDebInfo.

But it is possible to setup CPack to bundle multiple build directories at the same time to build a package that contains multiple configurations of the same project.

First we need to ahead and construct a directory called `multi_config` this will contain all the builds that we want to package together.

Second create a `debug` and `release` directory underneath `multi_config`. At the end you should have a layout that looks like:

─ multi_config
├── debug └── release

Now we need to setup debug and release builds, which would roughly entail the following:

```
cd debug
cmake -DCMAKE_BUILD_TYPE=Debug ../../MultiPackage/
cmake --build .
cd ../release
cmake -DCMAKE_BUILD_TYPE=Release ../../MultiPackage/
cmake --build .
cd ..
```

Now that both the debug and release builds are complete we can now use the custom MultiCPackConfig to package both builds into a single release.

```
cpack --config ../../MultiPackage/MultiCPackConfig.cmake
```