CMake

2022年10月9日　　15:47

==========

https://cmake.org/cmake/help/latest/guide/tutorial/index.html
v3.24.2

Step 1: A Basic Starting Point

最基础的工程是 从源码文件 构建一个 可执行文件。
对于简单的工程，3行的 CMakeLists.txt 就够了。
在 Step1 目录下 创建一个 CMakeLists.txt，并且内容如下：

```
cmake_minimum_required(VERSION 3.10)
# set the project name
project(Tutorial)
# add the executable
add_executable(Tutorial tutorial.cxx)
```

。。Step1目录是 tutorial 首页 可以下载的 压缩包中的。

注意，这个例子中 使用了 小写的命令， CMake 支持 大写，小写 和混合。
tutorial.cxx 的源码 是在 Step1 中的。用来计算 一个数字的 平方根。

Build and Run
上面就是全部所需的，现在 我们可以 build 和 run 工程了。
首先，执行 cmake 可执行文件 或 cmake-gui 来配置工程，然后 使用你选择的tool 来build。

例如，通过命令行，我们 可以 进入 CMake 源代码树的 Help/guide/tutorial 目录，创建一个 build 目录
mkdir Step1_build

然后，进入 build 目录，执行 CMake 来配置工程 和 生成一个 本地build 系统
cd Step1_build
cmake ../Step1

然后，调用build 系统 来 真正 编译/link 工程：
cmake --build .

最后，尝试使用 构建出来的 Tutorial
Tutorial 4294967296
Tutorial 10
Tutorial

Adding a Version Number and Configured Header File
我们添加的第一个功能是 为 我们的可执行文件 和 工程 一个 版本号。
虽然我们可以在 源码中 实现这种功能，但是 使用 CMakeLists.txt 提供更多的灵活性。

首先，修改 CMakeLists.txt，来使用 project() 命令 设置 工程名 和 版本号
```
cmake_minimum_required(VERSION 3.10)
# set the project name and version
project(Tutorial VERSION 1.0)
```

然后，配置一个 header 文件来 传递 版本号 到 源码：
```
configure_file(TutorialConfig.h.in TutorialConfig.h)
```

由于已配置的文件 将被写入 binary tree，我们必须 添加 那个目录 来 列出 用于搜索 include 的文件 的路径。将下面添加到 CMakeLists.txt 的最后
```
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                           )
```

在source 目录 创建 TutorialConfig.h.in，内容如下：
```
// the configured options and settings for Tutorial
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
```

当CMake 配置 这个 header 文件时， @Tutorial_VERSION_MAJOR@ and @Tutorial_VERSION_MINOR@ 的值会被替换。

然后修改 tutorial.cxx 来包含 配置的header文件： TutorialConfig.h

最后，修改 tutorial.cxx 来打印 可执行文件的名字 和 版本号：
```
  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
现在，来增加一些 C++ 11 的功能 到 工程：在tutorial.cxx中 使用 std::stod 替换 atof。
同时，移除 #include <cstdlib>：
```
const double inputValue = std::stod(argv[1]);
```

我们 需要在 CMake 代码中 指明状态，来让 它使用正确的flag。最简单的方法 来支持 具体的 CPP 标准，是通过 CMAKE_CXX_STANDARD 变量。 对于本教程，在 CMakeLists.txt 中设置 CMAKE_CXX_STANDARD 变量为 11 ，CMAKE_CXX_STANDARD_REQUIRED 为 True。 确保 CMAKE_CXX_STANDARD 声明 在 调用 add_executable 上面。

```
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 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

Rebuild

再次构建工程，我们 已经创建了 build 目录 和 运行过了CMake，所以我们可以 跳到 build 步骤：

```
cd Step1_build
cmake --build .
```

我们现在可以 使用 Tutorial 来运行

```
Tutorial 4294967296
Tutorial 10
Tutorial
```

当不带参数 执行时，检查 版本号 是否被 打印。

Step 2: Adding a Library

现在我们增加一个 库 到工程，这个库 会包含 我们自己实现的 用于计算平方根的 代码。 程序直接使用这个 库 而不是 C++编译器提供的sqrt 方法。

我们将 库 放到 MathFunctions 子目录下。 这个 目录已经包含了 一个 header文件：MathFunctions.h， 和一个 源代码文件 mysqrt.cxx。 源代码文件包含 一个 mysqrt的方法 来提供 和编译器的sqrt方法 类似的功能。

在 MathFunctions 目录下，增加 只有一行的 CMakeLists.txt 文件

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

为了使用 新的库，我们添加一个 add_subdirectory() 到 CMakeLists.txt 的 top-level， 这样， 这个库 会被 构建。

我们增加 新的库 到 可执行文件中， 增加 MathFunctions 作为一个 include 目录，这样 MathFunctions.h 头文件 可以被 找到。 CMakeLists.txt 的 top-level 的一些行 应该如下：

```
# 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"
                           )
```

。。top-level 是指 CMakeLists.txt 的 最开始处，还是说 顶格写？

现在，让我们 使得 MathFunctions 库 变成可选的。
对于教程来说，没有必要这么做，但是对于 大工程，这是常见的。
第一步是增加一个 option 到 CMakeLists.txt 的top-level。

```
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)
```

这个选项 会在 ccmake 和 cmake-gui 中 展现，并且 默认 on， 用户可以改变它。这个配置会被 cache，所以 用户 不必每次 在 build目录上 运行CMake 时 设置 它。

下一个change 是 使得 MathFunctions 库的 build 和 link 变成 conditional。
要这么做，我们会创建一个 if 语句，来检测 option的值。 在if 内部， 把 之前 添加的 add_subdirectory() 命令 移动下来，并且 增加 额外的 list 命令 来 保存一些信息，这些信息 用于 link 库 和 增加子目录 作为 Tutorial 目标的 include 目录。
CMakeLists.txt 文件的 top-level 的 end 会 看起来如下：

```
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}
                           )
```

注意 EXTRA_LIBS 变量，用来 收集 可选的 库 来稍后 link 到 可执行文件。
EXTRA_INCLUDES 是类似的，不过用于 可选的 header 文件。
这是一个 经典的(老的)方法 用于 处理 许多可选的 组件， next step 会 介绍 现代的方法。

源码的对应修改 非常简单。
首先，在 tutorial.cxx 中，包含 MathFunctions.h 头，如果我们需要它：

```
#ifdef USE_MYMATH
#  include "MathFunctions.h"
#endif
```
。。？是不是多了一个 #

然后，在同一个文件中，使用 USE_MYMATH 来控制 使用哪个 平方根方法
```
#ifdef USE_MYMATH
  const double outputValue = mysqrt(inputValue);
#else
  const double outputValue = sqrt(inputValue);
#endif
```

由于源码现在 需要 USE_MYMATH，我们可以增加到 TutorialConfig.h.in 文件 ：
```
#cmakedefine USE_MYMATH
```

练习：为什么 在 USE_MYMATH 选项 之后 配置 TutorialConfig.h.in 是重要的？ 如果 顺序颠倒 会怎么样？

执行cmake 或 cmake-gui来配置工程 然后 使用你选择的tool 来build。 然后 执行 构建的 Tutorial 可执行文件。

现在，让我们更新 USE_MYMATH 的值， 最简单的方法是 使用 cmake-gui 或 ccmake，如果你在终端中， 或，你想在 命令行中 修改 选项，try：
```
cmake ../Step2 -DUSE_MYMATH=OFF
```

rebuild 和 run。
哪个方法 返回更好的结果， sqrt 还是 mysqrt？

Step 3: Adding Usage Requirements for a Library

usage requirements 允许 更好地 控制 库 或 可执行文件的 link，和 include line，同时也给与了 更多 对 CMake中target 的 属性的 控制。

usage requirement 的初级命令：
```
target_compile_definitions()
target_compile_options()
target_include_directories()
target_link_libraries()
```

让我们重构 我们 上一章的代码， 使用现代的 CMake 方法：useage requirements。
首先，我们state： link到 MathFunctions 的 任何东西 需要 包含 当前源代码目录，MathFunctions 不必。 所以这是一个 INTERFACE usage requirement

记住 INTERFACE 意味着 消费者需要，生产者不需要的 东西。
在 MathFunctions/CMakeLists.txt 的最后 添加
```
target_include_directories(MathFunctions
          INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
          )
```

现在，我们指定了 MathFunctions 的 usage requirement，我们可以安全地 移除 top-level CMakeLists.txt 中的 EXTRA_INCLUDES 变量：

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

。。对比之前的， 这里的少了一行。
。。看来 top-level 是指 "root" 文件目录。。

and 这里也需要修改：

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

完成后， 运行 cmake ，或 cmake-gui 来配置工程 并使用你选择的 tool 来build 工程。 或在build 目录中 cmake --build .

Step 4: Installing and Testing

现在，我们开始 增加 install rule 和 testing 支持 到 我们的 工程。

Install Rules

install rules 非常简单：对于 MathFunctions 我们希望 install 库 和 header， 对于 应用我们希望 install 可执行文件 和 配置后的header。

所以在 MathFunctions/CMakeLists.txt 的最后，增加：

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

在 顶层 CMakeLists.txt 的最后，增加：

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

上面就是 创建一个 基础的 local install 的 全部。

执行cmake 或 cmake-gui 来build

然后 使用 带 install 选项的 cmake 命令 (3.15开始，更早的版本使用 make install)。
对于 multi-configuration tools， 记得使用 --config 来指定配置。
如果使用 IDE，simply build the INSTALL target。
这步会 install 关联的 header，库，可执行文件，例如：

```
cmake --install .
```

CMake 变量 CMAKE_INSTALL_PREFIX 用来决定 文件被install 的 root。
如果使用 cmake --install 命令， install prefix 可以通过 --prefix 参数来 覆盖默认

```
cmake --install . --prefix "/home/myuser/installdir"
```

进入 install 目录，验证 Tutorial 是否可以运行。

Testing Support
接下里，我们测试我们的应用。在 顶层 CMakeLists.txt 文件的最后，我们可以 启用 testing，然后 增加 几个 基础test 来验证 应用 是否正确。

```
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 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")
```

第一个test，简单验证 是否能run，没有错误或崩溃，并且 具有 zero return value。

第二个，使用 PASS_REGULAR_EXPRESSION 来 验证 测试的 output 包含 指定 string。这种情况下，当 错误的 参数数量 被提供时， 会 打印 usage message。

最后，有一个 函数 do_test 来 run 应用 且 验证 结果是否正确。do_test 的每次调用，会增加一个 测试 到 工程，一个test 包含 名字，输入，期望输出。

rebuild 应用， 进入 binary directory，执行 ctest： ctest -N 和 ctest -VV
对于 multi-config generator (如 Visual Studio)， 配置类型 必须 通过 -C <mode> 来指定。
例如，要 以 debug 模式 运行 tests，在 binary directory 中使用 ctest -C Debug -VV (不是debug subdirectory)

要以 Release 模式运行： 在相同的 目录下，使用 -C Release。
或者， IDE 中 build RUN_TESTS target。

Step 5: Adding System Introspection

增加一些 代码 到 我们的工程， 这些代码是否添加 取决于 目标平台。
例如， 根据 目标平台 是否有 log 和 exp 方法 来决定 是否添加 代码。

如果 平台 有 log 和 exp， 那么我们会使用 这2个方法 来 增强 mysqrt 方法。
我们先测试 这些方法 是否可用， 通过 在 MathFunctions/CMakeLists.txt 中使用 CheckCXXSourceCompilers 模块

在 MathFunctions/CMakeList.txt 文件的 target_include_directories() 调用后，添加 log exp 的check：

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

# does this system provide the log and exp functions?
include(CheckCXXSourceCompiles)
check_cxx_source_compiles("
  #include <cmath>
  int main() {
    std::log(1.0);
    return 0;
  }
" HAVE_LOG)
check_cxx_source_compiles("
  #include <cmath>
  int main() {
    std::exp(1.0);
    return 0;
  }
" HAVE_EXP)
```

如果可用，使用 target_compile_definitions() 来指定 HAVE_LOG 和 HAVE_EXP 作为 PRIVATE 编译定义：

```
MathFunctions/CMakeLists.txt
if(HAVE_LOG AND HAVE_EXP)
  target_compile_definitions(MathFunctions
                             PRIVATE "HAVE_LOG" "HAVE_EXP")
endif()
```

如果log,exp可用，那么我们会使用 它们 来 完成 mysqrt 的 平方根方法。
增加下面的 代码 到 MathFunctions/mysqrt.cxx 的 mysqrt 方法 (不要忘记 #endif 在

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

我们也需要 增加一个 include 到 mysqrt.cxx
```
#include <cmath>
```

使用 cmake， cmake-gui 来 build。
运行 Tutorial 可执行文件。

Step 6: Adding a Custom Command and Generated File

假设，我们决定： 不使用平台的 log 和 exp 函数， 而是 在mysqrt中 使用 预先计算的 表。
本章中，我们会 创建这张表 作为 build 的一部分，然后 编译这张表 到 应用中。

首先，移除 MathFunctions/CMakeLists.txt 中的 log 和 exp 的 检查。
然后移除 mysqrt.cxx 中的 HAVE_LOG，HAVE_EXP 的检查。
同时，移除 cmath 的 #include

在MathFunctions 子目录中， 一个新的 MakeTable.cxx 用来 生成 表。
review 文件后，我们可以看到， 表是通过 C++ 生成的，导出的 文件名 是 通过 参数 传入的。

下一步，是增加 对应的 命令 到 MathFunctions/CMakeLists.txt 文件中， 来 build MakeTable 可执行文件，然后 run 它， 作为 build 的一部分。 需要一些命令来完成这项工作。

第一，在 MathFunctions/CMakeLists.txt 的 top， 增加了 MakeTable 的 可执行文件。
MathFunctions/CMakeLists.txt
```
add_executable(MakeTable MakeTable.cxx)
```

然后，我们增加 自定义命令，来指定 如何 通过 MakeTable 生成 Table.h
MathFunctions/CMakeLists.txt
```
add_custom_command(
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  COMMAND MakeTable ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  DEPENDS MakeTable
  )
```

然后，我们需要让 CMake 知道 mysqrt.cxx 依赖于 生成的 Table.h 文件。 通过增加 生成的 Table.h 到 MathFunctions库 的 source list 中。
MathFunctions/CMakeLists.txt
```
add_library(MathFunctions
            mysqrt.cxx
```

```
            ${CMAKE_CURRENT_BINARY_DIR}/Table.h
            )
```

我们 必须 add 当前 binary directory 到 include目录 的list 中， 这样 Table.h 能被找到，并被 mysqrt.cxx include。

MathFunctions/CMakeLists.txt

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

现在，让我们使用 生成的 table，首先，修改 mysqrt.cxx 来include Table.h。
然后 重写 mysqrt方法 来使用 表：

MathFunctions/mysqrt.cxx

```
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;
}
```

使用 cmake 或 cmake-gui 来 配置和build。

这个工程在build 时，
会先 build MakeTable，
然后run MakeTable 生成 Table.h。
最后，编译 include Table.h 的 mysqrt.cxx， 来提供 MathFunctions 库。

运行Tutorial，验证是否使用 table。

Step 7: Packaging an Installer

我们希望 分发我们的工程 给其他人 使用。
我们想 提供 多个平台的 binary 和 source。
这个和 上节的 Installing and Testing 有一些不同，上节中，我们 安装 从 source code中 build 的 binary。
这里，我们会 build installation package，这个会支持 binary installation 和 package management。
我们先要 使用 CPack 来 创建 平台特定 的 installer。具体来说，我们需要 在 顶层的 CMakeLists.txt 的 底部 增加一些行。
CMakeLists.txt

```
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}")
set(CPACK_SOURCE_GENERATOR "TGZ")
include(CPack)
```

这就是全部所需要做的。
首先，我们include InstallRequiredSystemLibraries，这个模块会 include 对于当前平台 所需的 任何 runtime libraries。

然后，我们设置 一些 CPack 变量 为 我们保存license 和 version info 的地方。version info 在之前设置了，License.txt 在这个stp的 顶层目录中。 CPAKC_SOURCE_GENERATOR 为 source package 选择了一种 file format。

最后，我们 include CPack 模块，这个模块使用了 上面的变量 和 当前系统的 一些属性 来 setup installer。

下一步是 以通常的方式 build 工程，然后 运行 cpack 来build 一个 binary distribution。 在 binary 目录中执行：

```
cpack
```

要指定 generator，使用 -G， 对于 多config build，使用-C 指定 配置。

```
cpack -G ZIP -C Debug
```

要获得 可用的 generator 的list， 查看
https://cmake.org/cmake/help/latest/manual/cpack-generators.7.html#manual:cpack-generators(7)
或 cpack --help。

一个 archive generator 比如 ZIP， 创建一个 所有installed file 的 压缩包。

要创建 一个 full source tree 的 archive，你需要：

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

或者执行，make package， 或 在IDE 上 右键Package target 和 Build Project。

执行 binary 目录中的 installer， 然后 运行 installed 的 可执行文件，并验证是否正常。

Step 8: Adding Support for a Testing Dashboard

增加一个功能：提交测试结果到 dashboard 是简单的。 我们已经 在上一步中 为我们的工程定义了 一些 test。 现在，我们只需要 run 这些 test，然后 提交它们到 dashboard。
要支持 dashboard，我们需要 在顶层 CMakeLists.txt 中包含 CTest 模块。

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

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

CTest 模块 会自动 调用 enable_testing()， 所以我们可以 remove 它。

我们也 需要 获取一个 CTestConfig.cmake 文件，放到 顶层目录中， 在这个文件中，我们可以 指定 CTest 的信息。 它包含：

1. 工程名
2. 工程"每晚"启动时间
    a. the time when a 24 hour "day" start for this project
3. CDash 实例的 URL， 生成的数据 会发往这个 url。

One has been provided for you in this directory. It would normally be downloaded from the Settings page of the project on the CDash instance that will host and display the test results. Once downloaded from CDash, the file should not be modified locally.

。。。

CTestConfig.cmake
```
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 运行时 会读取这个文件。
要创建一个 简单的 dashboard 你可以 执行 cmake，cmake-gui 来配置工程，但不build，而是 change 目录 到 binary 目录，然后 run
```
ctest [-VV] -D Experimental
```

记住，对于 多config generator (如 VS)， configuration type 必须指定
```
ctest [-VV] -C Debug -D Experimental
```

或者，在IDE中， 构建 Experimental 目标。

ctest 会 build 和 test 工程，然后 发送结果到 Kitware's public dashboard: https://my.cdash.org/index.php?project=CMakeTutorial.
。。这个真可以访问。。

Step 9: Selecting Static or Shared Libraries

本节，我们会展示 如何 使用 BUILD_SHARED_LIBS 变量 来 控制 add_library() 的默认行为，可以控制 没有明确type(static,shared,module,object) 的库 的 build

需要增加 BUILD_SHARED_LIBS 到 顶层 CMakeLists.txt。
我们使用了 option() 命令，它允许 用户 决定 是否选择。

我们将 重构 MathFunctions ，成为一个真正的 封装了 mysqrt 或 sqrt 的库，而不是需要代码来 实现这个逻辑。 这也意味着， USE_MYMATH 不会控制 构建MathFunctions，而是 控制 这个库的行为。

第一步，更新 顶层 CMakeLists.txt 的 starting section，变成如下：

```
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 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# 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)
```

现在，我们使得 MathFunctions 始终被使用，我们需要更新 库的 逻辑。所以，在 MathFunctions/CMakeLists.txt 中，我们需要创建 一个 SqrtLibrary，它会被 conditionally build 和 install， 当 USE_MYMATH 被启用。 现在，由于这个是 tutorial，我们确定 SqrtLibrary 是 静态建立的。

MathFunctions/CmakeLists.txt 的最终版本：

```
# 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)

  target_compile_definitions(MathFunctions PRIVATE "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
set(installable_libs MathFunctions)
if(TARGET SqrtLibrary)
  list(APPEND installable_libs SqrtLibrary)
```

```
endif()
install(TARGETS ${installable_libs} DESTINATION lib)
install(FILES MathFunctions.h DESTINATION include)
```

接下来，更新 MathFunctions/mysqrt.cxx ，使用 mathfunctions 和 detail 命名空间

```
#include <iostream>

#include "MathFunctions.h"

// 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;
}
}
}
```

我们还需要在tutorial.cxx 中做出一些修改，所以 不再使用 USE_MYMATH：

1. 总是 导入 MathFunctions.h
2. 总是使用 mathfunctions::sqrt
3. 不导入 cmath

最终，更新 MathFunctions/MathFunctions.h 来使用 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);
}
```

现在，如果你 build 所有，你可能会看到 linking fail， 因为 我们结合 一个 没有position independent code 的 库 和 一个有position independent code 的库。
解决方案是，显示设置 SqrtLibrary 的 POSITION_INDEPENDENT_CODE 目标属性 为 True， 不管 build type。
MathFunctions/CMakeLists.txt

```
  # state that SqrtLibrary need PIC when the default is shared libraries
  set_target_properties(SqrtLibrary PROPERTIES
                        POSITION_INDEPENDENT_CODE ${BUILD_SHARED_LIBS}
                        )
  target_link_libraries(MathFunctions PRIVATE SqrtLibrary)
```

Exercise：我们修改 MathFunctions.h 来使用 dll export defines， 在CMake 文档中，你可以找到一个 helper 模块 来简化它吗？

Step 10: Adding Generator Expressions

generator expression， 在 构建系统 生成 product信息 到 每个 build 配置 时 eval。

generator expression 在许多 target properties 的 context 中 可用， 比如 LINK_LIBRARIES, INCLUDE_DIRECTORIES,COMPILE_DEFINITIONS 等。
也可以在 命令行中使用，如 target_link_libraries(), target_include_directories(), target_compile_definitions() 等。

generator expression 可以用来 enable conditional linking， conditional definitions 在 编译时，conditional 包括 目录等。 condition 可能基于 build的 配置，target properties，平台信息 或 其他 可以查询的 信息。

generator expression 有不同的type，包括 logical，informational，output 表达式。

logical expression 用来 创建 conditional output。basic expression 是 0 和 1

expression。 一个 $<0:...> 产生 空字符串， <1:...> 产生 "..." 这个string。它们也可以嵌套。

generator expression 的 常见用法是， 有条件地 add 编译器flag，比如，language-level 或 警告。
一个好的pattern是 associate 这个信息 到 INTERFACE target ，这可以允许 这个info 扩散出去。
让我们以 构建INTERFACE target 为开始，并 指定 所需的 C++ 标准等级 为11 而不是使用 CMAKE_CXX_STANDARD。

所以，下面的代码
```
# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

会被替换为
```
add_library(tutorial_compiler_flags INTERFACE)
target_compile_features(tutorial_compiler_flags INTERFACE cxx_std_11)
```

注意： 下面的章节，需要使用 cmake_minimum_required()。下面用到的Generator Expression 在3.15中被引入。
CMakeLists.txt
```
cmake_minimum_required(VERSION 3.15)
```

接下来，我们增加 希望的编译器警告flag。 由于 警告flag 很大程度 基于 编译器，我们使用 COMPILE_LANG_AND_ID generator expression 来控制 哪个flag被应用到 语言 和 编译器ID 集合，如下：
CMakeLists.txt
```
set(gcc_like_cxx "$<COMPILE_LANG_AND_ID:CXX,ARMClang,AppleClang,Clang,GNU,LCC>")
set(msvc_cxx "$<COMPILE_LANG_AND_ID:CXX,MSVC>")
target_compile_options(tutorial_compiler_flags INTERFACE
  "$<${gcc_like_cxx}:$<BUILD_INTERFACE:-Wall;-Wextra;-Wshadow;-Wformat=2;-Wunused>>"
  "$<${msvc_cxx}:$<BUILD_INTERFACE:-W3>>"
)
```

我们看到，警告flag 被包在 一个 BUILD_INTERFACE 条件中。这样做是为了让 我们installed的工程的 用户 不会 继承 我们的 warning flag。

Exercise：修改 MathFunctions/CMakeLists.txt， 是的 所有的 target 有一个 target_link_libraries() to tutorial_compiler_flags。

Step 11: Adding Export Configuration

在 Installing and Testing 章节中，我们增加了 CMake 的能力，来install 工程的 库和 header。
在 Packing an Installer 中， 我们介绍了 CMake 的能力，来 打包，分发给其他人。

下一步是 add 必要的info，来让 其他CMake 工程 能够使用 我们的工程。

第一步是更新我们的 install(TARGETS) 命令，来 不仅指定 DESTINATION，也指定 EXPORT。EXPORT 关键字 生成 一个 CMake 文件 ，它包含了code 来 导入 所有的 在 install 命令的 installation tree 中 列举的 target。

所以，在MathFunctions/CMakeLists.txt 的 install 命令 中 指定 EXPORT 到 MathFunctions 库， 后 看起来：

```
set(installable_libs MathFunctions tutorial_compiler_flags)
if(TARGET SqrtLibrary)
  list(APPEND installable_libs SqrtLibrary)
endif()
install(TARGETS ${installable_libs}
        EXPORT MathFunctionsTargets
        DESTINATION lib)
install(FILES MathFunctions.h DESTINATION include)
```

现在，MathFunctions 被export 了， 我们也需要 明确 install 生成的 MathFunctionsTargets.cmake 文件。这个可以通过 在 顶层的CMakeLists.txt 的 最后添加：

```
install(EXPORT MathFunctionsTargets
  FILE MathFunctionsTargets.cmake
  DESTINATION lib/cmake/MathFunctions
)
```

此时，你应该 try run CMake。如果 每个都设置正确，你会看到 CMake 生成一个 error：

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

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

which is prefixed in the source directory.
```

CMake想说的是：在 生成 export 信息时，它会 的导出路径 是基于 当前电脑的， 无法 适用于其他的 机器。 解决方案是，更新 MathFunctions target_include_directories() 来 了解 它需要 不同的 INTERFACE 地址 when begin used from within the build directory and from an install/package。 这意味着 把 MathFunctions 的 target_include_directories() 调整如下：

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

一旦这个已经更新，我们以重新运行 CMake ，并验证 它不会发出警告了。

现在，CMake 正确地 打包 所需的target information， 但我们仍然需要 生成 MathFunctionsConfig.cmake，这样CMake find_package() 命令 可以找到 我们的工程。

让我们继续：增加一个 新的 file 到 工程的 顶层，名字为 Config.cmake.in，并且 带有下面的 内容：

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

然后，配置 和 install 那个文件，将下面的 内容添加到 顶层的 CMakeLists.txt 的 底部：

```
install(EXPORT MathFunctionsTargets
  FILE MathFunctionsTargets.cmake
  DESTINATION lib/cmake/MathFunctions
)
include(CMakePackageConfigHelpers)
```

接下来，我们只想 configure_package_config_file(). 这个命令 会配置 一个 提供的flie，但 和 标准的 config_file() 有一点点不同。 要正确使用 这个方法， input file 应该 有单独的一行 @PACKAGE_INIT@ 加上 期望的内容。 那个变量会被 替换为 代码块，代码块会引入 设置的值 到相对路径。这些 引入的 变量 可以 以 PACKACE_前缀 + 名字 来引用。

CMakeLists.txt

```
install(EXPORT MathFunctionsTargets
  FILE MathFunctionsTargets.cmake
  DESTINATION lib/cmake/MathFunctions
)
include(CMakePackageConfigHelpers)
# generate the config file that is includes the exports
configure_package_config_file(${CMAKE_CURRENT_SOURCE_DIR}/Config.cmake.in
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake"
  INSTALL_DESTINATION "lib/cmake/example"
  NO_SET_AND_CHECK_MACRO
  NO_CHECK_REQUIRED_COMPONENTS_MACRO
  )
```

接下来是 write_basic_package_version_file()，这个命令写入 "find_package" 使用的文件，包含了所需的package 的兼容性 和 版本。这里，我们使用 Tutorial_VERSION_* 变量，并且 声明 它和 AnyNewerVersion 兼容。

```
write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake"
  VERSION "${Tutorial_VERSION_MAJOR}.${Tutorial_VERSION_MINOR}"
  COMPATIBILITY AnyNewerVersion
)
```

最后，设置 2个生成的文件 需要 install

```
install(FILES
  ${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake
  ${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake
  DESTINATION lib/cmake/MathFunctions
  )
```

此时，我们 生成了一个 relocatable CMake 配置 为 我们的工程， 在 工程被install 或 package 时 可以 使用。

如果我们希望 我们的工程 也可以从 build 目录 被使用， 我们只需要 增加 下面的内容 到

顶层的 CMakeLists.txt 中。

```
export(EXPORT MathFunctionsTargets
  FILE "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsTargets.cmake"
)
```

通过这个 export调用，我们现在 生成了 一个 Targets.cmake， 允许 配置后的 在 build 目录的 MathFunctionsConfig.cmake 被其他工程 使用，而不需要 install。

Step 12: Packaging Debug and Release

Note：这个例子 对于 single-config generator 可用，对于 多config generator 不可用(如VS)

默认下，CMake 的模型是 一个build 目录 只包含 single config，无论是 debug,release,minSizeRel,relWithDebInfo。 然而，可以配置 CPack 来 绑定 多个 build 目录，构造一个 包含 同一个工程的 多个 配置的 package。

首先，我们希望确保：debug 和 release 的可执行文件 和 安装的库 使用 不同的名字，让我们增加 前缀d 到 debug 可执行文件和库。

在 顶层 CMakeLists.txt 文件的 开始处 附近 设置 CMAKE_DEBUG_POSTFIX。

```
set(CMAKE_DEBUG_POSTFIX d)
add_library(tutorial_compiler_flags INTERFACE)
```

在 tutorial 可执行文件上 增加 DEBUG_POSTFIX 属性

```
add_executable(Tutorial tutorial.cxx)
set_target_properties(Tutorial PROPERTIES DEBUG_POSTFIX ${CMAKE_DEBUG_POSTFIX})
target_link_libraries(Tutorial PUBLIC MathFunctions)
```

增加版本号 到 MathFunctions 库，在 MathFunctions/CMakeLists.txt 中，设置 VERSION 和 SOVERSION：

```
set_property(TARGET MathFunctions PROPERTY VERSION "1.0.0")
set_property(TARGET MathFunctions PROPERTY SOVERSION "1")
```

从step12 目录中，创建 debug 和 release 子目录。

设置debug 和 release build。我们可以使用 CMAKE_BUILD_TYPE 来设置 配置类型：

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

现在，debug 和 release build 都完成了， 我们可以使用 自定义的配置文件 来 打包 这2个 build 到 一个单独的release。

在step12目录，创建一个 文件，叫 MultiCPackConfig.cmake。 在这个文件中，先include cmake创建的 默认的配置文件。

接下来，使用 CPACK_INSTALL_CMAKE_PROJECTS 变量 来指定 哪个project 被install。 这个例子中，我们希望 install debug 和 release。

MultiCPackConfig.cmake

```
include("release/CPackConfig.cmake")
set(CPACK_INSTALL_CMAKE_PROJECTS
    "debug;Tutorial;ALL;/"
    "release;Tutorial;ALL;/"
    )
```

在step12目录中，执行 cpack 命令，通过 config 来指定我们自己的配置文件

```
cpack --config MultiCPackConfig.cmake
```

end

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========