这记录了旧版本的 CMake。单击此处查看最新版本。或者，从上面的下拉菜单中选择一个版本。

CMake 教程

内容

CMake 教程提供了一个分步指南，涵盖 CMake 帮助解决的常见生成系统问题。查看示例项目中所有主题如何协同工作会很有帮助。示例的教程文档和源代码可以在 CMake 源代码树的目录中找到。每个步骤都有自己的子目录，其中包含可用作起点的代码。教程示例是渐进的，因此每个步骤都为上一步提供了完整的解决方案。 Help/guide/tutorial

基本起点（步骤 1）

最基本的项目是从源代码文件构建的可执行文件。对于简单项目，需要三行文件。这将是我们教程的起点。在目录中创建一个文件，如下所示： CMakeLists.txt CMakeLists.txt Step1

```
cmake_minimum_required(VERSION 3.10)

# set the project name
project(Tutorial)

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

请注意，此示例在文件中使用小写命令。CMake 支持上部、下部和混合大小写命令。的源代码在目录中提供，可用于计算数字的平方根。 CMakeLists.txt tutorial.cxx Step1

添加版本号和配置的报头文件

我们将添加的第一个功能是提供我们的可执行文件，并提供了一个版本号。虽然我们可以在源代码中专门做到这一点，但使用提供了更大的灵活性。 CMakeLists.txt

首先，修改文件以使用 `project()` 命令设置项目名称和版本号。 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

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

然后，配置一个头文件以将版本号传递给源代码：

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

由于配置的文件将写入二进制树，因此我们必须将该目录添加到要搜索的包含文件的路径列表中。将以下行添加到文件末尾： CMakeLists.txt

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

使用您最喜爱的编辑器，在源目录中创建包含以下内容的内容： TutorialConfig.h.in

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

当 CMake 配置此头文件时，将替换 的 和 的值。 @Tutorial_VERSION_MAJOR@ @Tutorial_VERSION_MINOR@

下一个修改以包括配置的标头文件 。 tutorial.cxx TutorialConfig.h

最后，让我们按如下更新来打印版本号： tutorial.cxx

```
  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;
  }
```

指定C++标准

接下来，让我们通过 C++，将一些 C++11 功能添加到项目中。同时，删除 。 atof std::stod tutorial.cxx #include <cstdlib>

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

我们需要在 CMake 代码中明确说明它应该使用正确的标志。在 CMake 中启用特定C++的最简单方法是使用CMAKE_CXX_STANDARD变量。对于本教程，将CMAKE_CXX_STANDARD中的变量设置为 11，CMAKE_CXX_STANDARD_REQUIRED True： CMakeLists.txt

```
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)
```

生成和测试

运行cmake可执行文件或cmake-gui来配置项目，然后使用所选的生成工具生成它。

例如，从命令行，我们可以导航到 CMake 源代码树的目录并运行以下命令： Help/guide/tutorial

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

导航到生成教程的目录（可能是制作目录或调试或发布生成配置子目录），并运行以下命令：

```
Tutorial 4294967296
Tutorial 10
Tutorial
```

添加库（步骤 2）

现在，我们将向项目添加一个库。此库将包含我们自己的实现，用于计算数字的平方根。然后，可执行文件可以使用此库而不是编译器提供的标准平方根函数。

对于本教程，我们将将库放入名为 的子目录中。此目录已包含标头文件 和源文件 。源文件有一个函数调用，它提供与编译器函数类似的功能。 MathFunctions MathFunctions.h mysqrt.cxx mysqrt sqrt

将以下一行文件添加到目录中： CMakeLists.txt MathFunctions

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

为了利用新库，我们将在顶级文件中添加 add_subdirectory () 调用，以便构建库。我们将新库添加到可执行文件，并添加为包含目录，以便可以找到头文件。顶级文件的最后几行现在应看起来像： CMakeLists.txt MathFunctions mqsqrt.h CMakeLists.txt

```
# 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"
                           )
```

现在，让我们使 Math 函数库成为可选的。虽然对于本教程，真的没有必要这样做，对于较大的项目，这是常见的情况。第一步是向顶级文件添加一个选项。 CMakeLists.txt

```
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)
```

此选项将显示在 cmake-gui和ccmake中，默认值为 ON，用户可以更改。此设置将存储在缓存中，以便用户在每次在生成目录上运行 CMake 时都不需要设置值。

下一个更改是使构建和链接 Math 函数库成为条件。为此，我们将顶级文件的末尾更改为如下所示： CMakeLists.txt

```
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}
                           )
```

请注意使用 变量收集任何可选库，以稍后链接到可执行文件。该变量同样用于可选的头文件。这是处理许多可选组件时的经典方法，我们将在下一步介绍现代方法。 EXTRA_LIBS EXTRA_INCLUDES

对源代码的相应更改相当简单。首先，在 中，如果需要，请包括标头： tutorial.cxx MathFunctions.h

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

然后，在同一文件中，控制使用哪个平方根函数： USE_MYMATH

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

由于源代码现在需要，我们可以添加它与以下行： USE_MYMATH TutorialConfig.h.in

```
#cmakedefine USE_MYMATH
```

练习：为什么在 选项 之后配置 ？ 如果我们把两者颠倒过来会怎么样？ TutorialConfig.h.in USE_MYMATH

运行cmake可执行文件或cmake-gui来配置项目，然后使用所选的生成工具生成它。然后运行构建的教程可执行文件。

使用ccmake可执行文件或cmake-gui更新 的值。重新生成并再次运行本教程。哪个函数提供更好的结果， sqrt 或 mysqrt? USE_MYMATH

添加库的使用要求（步骤 3）

使用要求可以更好地控制库或可执行文件的链接，并包括行，同时可以更好地控制 CMake 内目标的传递属性。利用使用要求的主要命令是：

- target_compile_definitions()
- target_compile_options()
- target_include_directories()
- target_link_libraries()

让我们从添加库（步骤2）中重构代码，以使用现代的使用要求的 CMake 方法。我们首先指出，任何链接到 Math 函数的人都需要包括当前源目录，而 Math 函数本身则不需要。因此，这可以成为一种使用要求。 INTERFACE

记住意味着消费者需要的东西，但生产者不需要。将以下行添加到 的末尾： INTERFACE MathFunctions/CMakeLists.txt

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

现在，我们已经指定了 Math 函数的使用要求，我们可以安全地从顶层中删除变量的使用，这里： EXTRA_INCLUDES CMakeLists.txt

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

这里：

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

完成操作后， 运行 cmake可执行文件或 cmake-gui 来配置项目，然后使用所选的生成工具或从生成目录生成它。 cmake --build .

安装和测试（步骤 4）

现在，我们可以开始向项目添加安装规则和测试支持。

安装规则

安装规则相当简单：对于 Math 函数，我们要安装库和标头文件;对于应用程序，我们要安装可执行文件和配置的标头。

因此，最后我们添加： MathFunctions/CMakeLists.txt

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

在顶级的末尾，我们添加：CMakeLists.txt

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

这就是创建教程的基本本地安装所需的全部。

运行cmake可执行文件或cmake-gui来配置项目，然后使用所选的生成工具生成它。使用cmake 命令（在3.15 中引入，旧版本的 CMake 必须使用）从命令行运行安装步骤，或从 IDE 生成目标。这将安装相应的头文件、库和可执行文件。 installmake install INSTALL

CMake CMAKE_INSTALL_PREFIX用于确定文件安装位置的根目录。如果使用自定义安装目录，可以通过 参数给出。对于多配置工具，请使用 参数指定配置。 cmake --install--prefix--config

验证已安装的教程是否运行。

测试支持

接下来，让我们测试我们的应用程序。在顶级文件的末尾，我们可以启用测试，然后添加一些基本测试来验证应用程序是否正常工作。 CMakeLists.txt

```
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")
```

第一个测试只是验证应用程序是否运行，不会 segfault 或以其他方式崩溃，并且具有零返回值。这是 CTest 测试的基本形式。

下一个测试使用PASS_REGULAR_EXPRESSION属性来验证测试的输出是否包含某些字符串。在这种情况下，验证在提供错误数量的参数时是否打印了使用消息。

最后，我们有一个函数，用于运行应用程序，并验证计算的平方根是否适合给定的输入。对于 的每次调用，另一个测试将添加到项目中，其名称、输入和基于传递的参数的预期结果。`do_test` `do_test`

重新生成应用程序，然后将 cd 更新到二进制目录并运行`ctest`可执行文件： 和 。对于多配置生成器（例如 Visual Studio），必须指定配置类型。例如，若要在调试模式下运行测试，请使用生成目录（而不是调试子目录！或者，从 IDE 生成目标。`ctest -N` `ctest -VV` `ctest -C Debug -VV` `RUN_TESTS`

添加系统反省（步骤 5）

让我们考虑将一些代码添加到我们的项目中，这些代码取决于目标平台可能不具有的功能。对于此示例，我们将添加一些代码，这些代码取决于目标平台是否具有 和 函数。当然，几乎每个平台都有这些功能，但本教程假定它们并不常见。`log` `exp`

如果平台有，然后我们将使用它们来计算函数中的平方根。我们首先使用顶级的`CheckSymbolExists`模块测试这些功能的可用性。我们将在 中使用 新定义，因此请确保在配置该文件之前设置它们。`log` `exp` `mysqrt` `CMakeLists.txt` `TutorialConfig.h.in`

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

现在，让我们添加这些定义，以便我们可以使用它们从： `TutorialConfig.h.in` `mysqrt.cxx`

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

If and are available on the system, then we will use them to compute the square root in the function. Add the following code to the function in (don’ t forget the before returning the result!):`log` `exp` `mysqrt` `mysqrt` `MathFunctions/mysqrt.cxx` `#endif`

```
#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;
```

We will also need to modify to include .`mysqrt.cxx` `cmath`

```
#include <cmath>
```

Run the `cmake` executable or the `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 and , even if we think they should be available. We should realize quickly that we have forgotten to include in .`log` `exp` `TutorialConfig.h` `mysqrt.cxx`

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

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

进行此更新后，继续生成项目并运行构建的教程可执行文件。如果 和 仍在使用中，请从生成目录打开生成的文件。也许它们在当前系统中不可用？ log exp TutorialConfig.h

哪个函数现在提供更好的结果， sqrt 或 mysqrt?

指定编译定义

除了在中，我们还有一个更好的地方来保存 和 价值吗？让我们尝试使用 target_compile_definitions。 HAVE_LOG HAVE_EXP TutorialConfig.h

首先，从 中删除 定义。我们不再需要从 或 额外的包括。 TutorialConfig.h.in TutorialConfig.h mysqrt.cxx MathFunctions/CMakeLists.txt

接下来，我们可以将 检查和 移动到 ，然后将这些值指定为编译定义。 HAVE_LOG HAVE_EXP MathFunctions/CMakeLists.txt PRIVATE

```
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()
```

进行这些更新后，继续并再次生成项目。运行构建的教程可执行文件，并验证结果是否与本步骤中之前的结果相同。

添加自定义命令和生成的文件（步骤 6）

假设，为了本教程的目的，我们决定我们从来不想使用平台和函数，而是要生成一个在函数中使用的预计算值表。在本节中，我们将创建表作为生成过程的一部分，然后将该表编译到我们的应用程序中。 log exp mysqrt

首先，让我们删除 中 和 函数的检查。然后从 中删除 的支票。同时，我们可以删除 。 log exp MathFunctions/CMakeLists.txt HAVE_LOG HAVE_EXP mysqrt.cxx #include <cmath>

在子目录中，提供了名为"新源文件"来生成表。 MathFunctions MakeTable.cxx

查看文件后，我们可以看到表作为有效的代码生成C++输出文件名作为参数传递。

下一步是将适当的命令添加到文件中以生成 MakeTable 可执行文件，然后作为生成过程的一部分运行它。需要几个命令来完成这一任务。 MathFunctions/CMakeLists.txt

首先，在 的顶部添加 的可执行文件作为任何其他可执行文件将被添加。 MathFunctions/CMakeLists.txt MakeTable

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

然后，我们添加一个自定义命令，指定如何通过运行 MakeTable 进行生产。 Table.h

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

接下来，我们必须让CMake知道，这取决于生成的文件。这是通过将生成的 添加到库 Math 函数的源列表中完成。 mysqrt.cxx Table.h Table.h

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

我们还必须将当前二进制目录添加到包含目录列表中，以便可以由 找到和包含。 Table.h mysqrt.cxx

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

现在，让我们使用生成的表。首先，修改为包括 。接下来，我们可以重写 mysqrt 函数以使用表： mysqrt.cxx Table.h

```
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;
}
```

运行`cmake`可执行文件或`cmake-gui`来配置项目，然后使用所选的生成工具生成它。

生成此项目时，它将首先生成可执行文件。然后，它将运行以产生 。最后，它将编译，其中包括生成 Math 函数库。 MakeTable MakeTable Table.h mysqrt.cxx Table.h

运行教程可执行文件并验证它是否正在使用该表。

构建安装程序（步骤 7）

接下来，假设我们希望将项目分发给其他人，以便他们可以使用它。我们希望在各种平台上提供二进制和源分发。这与之前在安装和测试（步骤4）中安装的安装略有不同，我们在安装从源代码构建的二进制文件。在此示例中，我们将构建支持二进制安装和包管理功能的安装包。为此，我们将使用 CPack 创建特定于平台的安装程序。具体来说，我们需要在顶层文件的底部添加几行。 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}")
include(CPack)
```

这就是它的所有。我们首先包括安装要求系统图书馆。此模块将包括项目当前平台所需的任何运行时库。接下来，我们将一些 CPack 变量设置为存储此项目的许可证和版本信息的位置。版本信息在本教程的较早版本中设置，并且 已包含在此步骤的顶级源目录中。 license.txt

最后，我们包括CPack模块，它将使用这些变量和当前系统的一些其他属性来设置安装程序。

下一步是以通常的方式生成项目，然后运行 cpack 可执行文件。若要生成二进制分发，请从二进制目录运行：

```
cpack
```

若要指定生成器，请使用 选项。对于多配置生成，请使用 指定配置。例如： -G-C

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

若要创建源分发，请键入：

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

或者，运行或右键单击目标，然后从 IDE 单击。 make package Package Build Project

运行在二进制目录中找到的安装程序。然后运行已安装的可执行文件并验证其是否有效。

添加对仪表板的支持（步骤 8）

添加对将测试结果提交到仪表板的支持非常简单。我们已经在测试支持中为我们的项目定义了一些测试。现在，我们只需要运行这些测试，然后将它们提交到仪表板。为了包括对仪表板的支持，我们在顶级 中包括 CTest 模块。 CMakeLists.txt

取代：

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

带：

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

CTest模块将自动调用，因此我们可以将其从 CMake 文件中删除。enable_testing()

我们还需要在顶级目录中创建一个文件，您可以在其中指定项目的名称以及提交仪表板的位置。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可执行文件在运行时将读取此文件。若要创建一个简单的仪表板，您可以运行 cmake可执行文件或 cmake-gui来配置项目，但尚不生成它。相反，将目录更改为二进制树，然后运行：

```
ctest [-VV] - D 实验
```

请记住，对于多配置生成器（例如 Visual Studio），必须指定配置类型：

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

或者，从 IDE 生成目标。Experimental

ctest可执行文件将生成和测试项目，并将结果提交到 Kitware 的公共仪表板：https://my.cdash.org/index.php?project=CMakeTutorial。

混合静态和共享（步骤 9）

在本节中，我们将介绍如何使用 BUILD_SHARED_LIBS变量来控制 add_library() 的默认行为，并允许控制不显式类型 （， 或） 的库的构建方式。STATIC SHARED MODULE OBJECT

为了做到这一点，BUILD_SHARED_LIBS添加到顶层。我们使用option()命令，因为它允许用户选择值是开还是关。CMakeLists.txt

接下来，我们将重构 Math 函数，使之成为一个真正的库，该库使用 或 封装，而不是要求调用代码执行此逻辑。这也意味着不会控制构建 Math 函数，而是将控制此库的行为。mysqrt sqrt USE_MYMATH

第一步是将顶层的起始部分更新为：CMakeLists.txt

```
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)
```

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

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

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

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

```
#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 中进行一些更改，以便它不再使用： `tutorial.cxx` `USE_MYMATH`

1. 始终包括`MathFunctions.h`
2. 始终使用`mathfunctions::sqrt`
3. 不包括 cmath

最后，使用 dll 导出的更新定义： `MathFunctions/MathFunctions.h`

```
#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);
}
```

此时，如果生成所有内容，您会注意到链接失败，因为我们正在将不带位置独立代码的静态库与具有位置独立代码的库组合。 解决此解决方案是显式将 SqrtLibrary `POSITION_INDEPENDENT_CODE`的`目标`属性设置为 True，无论生成类型如何。

```
  # 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)
```

练习：我们修改为使用 dll 导出定义。使用 CMake 文档，您能否找到一个帮助器模块来简化此操作？ MathFunctions.h

添加生成器表达式（步骤 10）

在生成系统期间计算生成器表达式，以生成特定于每个生成配置的信息。

生成器表达式在许多目标属性的上下文中是允许的，例如 LINK_LIBRARIES、INCLUDE_DIRECTORIESCOMPILE_DEFINITIONS 等。当使用命令填充这些属性时，例如 target_link_libraries()，target_include_directories()，target_compile_definitions() 等属性时，也可以使用它们。

生成器表达式可用于启用条件链接、编译时使用的条件定义、条件包括目录等。这些条件可能基于生成配置、目标属性、平台信息或任何其他可查询信息。

有不同类型的生成器表达式，包括逻辑表达式、信息表达式和输出表达式。

逻辑表达式用于创建条件输出。基本表达式是 0 和 1 表达式。A 将产生空字符串，并产生"..."的内容。它们也可以嵌套。 $<0:...><1:...>

生成器表达式的常见用法是有条件地添加编译器标志，例如语言级别或警告的标志。一个很好的模式是将此信息与允许此信息传播的目标关联。让我们首先构造一个目标，并指定所需的C++标准级别，而不是使用CMAKE_CXX_STANDARD。 INTERFACE INTERFACE 11

因此，以下代码：

```
# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

将替换为：

```
add_library(tutorial_compiler_flags INTERFACE)
target_compile_features(tutorial_compiler_flags INTERFACE cxx_std_11)
```

接下来，我们添加项目所需的编译器警告标志。由于警告标志因编译器而异，我们使用生成器表达式来控制要应用给定语言和一组编译器 ID 的标志，如下所示： COMPILE_LANG_AND_ID

```
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}:$<BUILD_INTERFACE:-Wall;-Wextra;-Wshadow;-Wformat=2;-Wunused>>"
  "$<${msvc_cxx}:$<BUILD_INTERFACE:-W3>>"
)
```

对此，我们看到警告标志封装在条件中。这样做是以便我们已安装项目的使用者不会继承我们的警告标志。 BUILD_INTERFACE

练习：修改，以便所有目标都 `target_link_libraries()` 调用。

`MathFunctions/CMakeLists.txt` `tutorial_compiler_flags`

添加导出配置（步骤 11）

在本教程的安装和测试（步骤 4）期间，我们添加了 CMake 安装项目库和标头的能力。在构建安装程序（步骤 7）期间，我们添加了打包此信息以便将其分发给其他人的能力。

下一步是添加必要的信息，以便其他 CMake 项目可以使用我们的项目，包括从生成目录、本地安装还是打包时。

第一步是更新我们的 `install(TARGETS)` 命令，不仅指定一个关键字生成并安装包含代码的 CMake 文件，以从安装树导入安装命令中列出的所有目标。因此，让我们继续并显式通过更新中的命令来显式显示 Math 函数库，以使其看起来像：

`DESTINATION` `EXPORT` `EXPORT` `EXPORT` `install` `MathFunctions/CMakeLists.txt`

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

现在，我们已导出 Math 函数，我们还需要显式安装生成的文件。这是通过将以下内容添加到顶层的底部完成的：`MathFunctionsTargets.cmake` `CMakeLists.txt`

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

此时，您应该尝试运行 CMake。如果一切都设置正确，你会看到，CMake将生成一个错误，如下所示：

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

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

which is prefixed in the source directory.
```

CMake 试图说的是，在生成导出信息时，它将导出一条与当前计算机本质上相关的路径，并且在其他计算机上无效。为此的解决方案是更新 Math 函数 `target_include_directories()` 以了解在从生成目录中和安装/包使用时需要不同的位置。这意味着将 `target_include_directories()` 调用转换为 Math 函数的外观：`INTERFACE`

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

更新后，我们可以重新运行 CMake 并验证它不再警告。

此时，我们已正确打包所需的目标信息，但我们仍然需要生成一个，以便 CMake `find_package()` 命令可以找到我们的项目。因此，让我们继续，并添加一个新的文件到项目的顶层调用以下内容：`MathFunctionsConfig.cmake` `Config.cmake.in`

```
@PACKAGE_INIT@

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

然后，要正确配置和安装该文件，请将以下内容添加到顶层的底部：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
  )
# generate the version file for the config file
write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake"
  VERSION "${Tutorial_VERSION_MAJOR}.${Tutorial_VERSION_MINOR}"
  COMPATIBILITY AnyNewerVersion
)

# install the configuration file
install(FILES
  ${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake
  DESTINATION lib/cmake/MathFunctions
  )
```

此时，我们已为项目生成可重新可拆卸的 CMake 配置，可以在安装或打包项目后使用。如果我们希望我们的项目也从生成目录使用，我们只需将以下内容添加到顶层的底部：CMakeLists.txt

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

通过此导出调用，我们现在生成 一个 ， 允许生成目录中配置的 由其他项目使用，而无需安装它。 Targets.cmake MathFunctionsConfig.cmake

打包调试和发布（步骤 12）

注：此示例适用于单配置生成器，不适用于多配置生成器（例如 Visual Studio）。

默认情况下，CMake 的模型是生成目录仅包含单个配置，即调试、发布、MinSizeRel 或 RelWithDebInfo。但是，可以设置 CPack 来捆绑多个生成目录并构造一个包含同一项目的多个配置的包。

首先，我们希望确保调试和发布版本对要安装的可执行文件以及库使用不同的名称。让我们使用 *d* 作为调试可执行文件以及库的后缀。

设置CMAKE_DEBUG_POSTFIX在顶级文件的开头附近设置：CMakeLists.txt

```
set(CMAKE_DEBUG_POSTFIX d)

add_library(tutorial_compiler_flags INTERFACE)
```

教程可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)
```

我们还要将版本编号添加到 Math 函数库中。在 中设置 版本和 SOVERSION 属性：
MathFunctions/CMakeLists.txt

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

从目录中创建和子目录。布局将看起来像： Step12 debug release

```
- Step12
   - debug
   - release
```

现在我们需要设置调试和发布版本。我们可以使用CMAKE_BUILD_TYPE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，调试和发布版本都已完成，我们可以使用自定义配置文件将两个生成打包到单个版本中。在目录中，创建一个名为 的文件。在此文件中，首先包括 cmake 可执行文件创建的默认配置文件。 Step12 MultiCPackConfig.cmake

接下来，使用 变量指定要安装的项目。在这种情况下，我们希望同时安装调试和发布。
CPACK_INSTALL_CMAKE_PROJECTS

```
include("release/CPackConfig.cmake")

set(CPACK_INSTALL_CMAKE_PROJECTS
    "debug;Tutorial;ALL;/"
    "release;Tutorial;ALL;/"
    )
```

从目录中，运行 cpack指定我们的自定义配置文件，选项： Step12 config

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