← CMake Tutorial – Chapter 5: Functionally Improved Testing

CMake Tutorial – Chapter 6: Realistically Getting a Boost

Posted on 2015-03-03 by John Lamp

Contents

Introduction

Now that we have our testing simplified and automated we have a great foundation upon which to build our amazing command line To Do list app. What's that? You say that an awesome To Do app allows you to add items to your list? Indeed it does, and more! But wait, let's not get ahead of ourselves. We need to be able to accept and parse command line options if this app is to be of any use at all.

Archives
March 2015
July 2013
May 2013
March 2013

Categories
CMake Tutorial

Meta
Log in
Entries RSS
Comments RSS
WordPress.org

I know what you are thinking now: parsing command line options is a drag and who likes parsing stuff anyway? Well we are in luck as the Boost Program Options library will do all the hard work for us. All we need to do is rewrite our main function to be something useful, let the library do the parsing and our app will be on it's way to the top 10 list. Okay, I might be exaggerating that last one.

Boosting the Command Line

Okay, that section title may be a little over the top. Our main function has languished while we set up testing and streamlined our CMake. Now it's time to turn attention back to it and what we find is that it needs to be gutted and re-done, much like an old kitchen. Since we have better tests we don't need the one in main anymore. We will update main to have two command line options: `--add`, which will add a new entry to the to do list, and `--help`, which will do what you'd expect.

main.cc

```
 1∞#include <iostream>
 2∞  using std::cerr;
 3∞  using std::cout;
 4∞  using std::endl;
 5∞#include <string>
 6∞  using std::string;
 7∞
 8∞#include <boost/program_options.hpp>
 9∞  namespace po = boost::program_options;
10∞
11∞#include "ToDoCore/ToDo.h"
12∞  using ToDoCore::ToDo;
13∞
14∞int main(
15∞    int    argc,
16∞    char** argv
17∞)
```

```
{
    po::options_description desc("Options");
    desc.add_options()
        ("help,h", "display this help")
        ("add,a", po::value< string >(), "add a new entry to the To Do list")
        ;

    bool parseError = false;
    po::variables_map vm;
    try
    {
        po::store(po::parse_command_line(argc, argv, desc), vm);
        po::notify(vm);
    }
    catch (po::error& error)
    {
        cerr << "Error: " << error.what() << "\n" << endl;
        parseError = true;
    }

    if (parseError || vm.count("help"))
    {
        cout << "todo:  A simple To Do list program" << "\n";
        cout                                        << "\n";
        cout << "Usage:"                            << "\n";
        cout << "  " << argv[0] << " [options]"      << "\n";
        cout                                        << "\n";
        cout << desc                                << "\n";

        if (parseError)
        {
            return 64;
        }
        else
        {
```

```
53∞            return 0;
54∞        }
55∞    }
56∞
57∞
58∞    ToDo list;
59∞
60∞    list.addTask("write code");
61∞    list.addTask("compile");
62∞    list.addTask("test");
63∞
64∞    if (vm.count("add"))
65∞    {
66∞        list.addTask(vm["add"].as< string >());
67∞    }
68∞
69∞    for (size_t i = 0; i < list.size(); ++i)
70∞    {
71∞        cout << list.getTask(i) << "\n";
72∞    }
73∞    return 0;
74∞}
```

Boost Program Options makes it easier to parse command line options than it would be to do it by hand. Now that we have the required `--help` option and the `--add` our app is a bit more useful.

There's a new problem now. How will we link our app against Boost? As it turns out CMake has a command for finding things like Boost: the `find_package()` command. Let's see how it works.

CMakeLists.txt — **New or modified lines in bold.**

```
1∞cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
2∞set(CMAKE_LEGACY_CYGWIN_WIN32 0)
3∞
```

```
project("To Do List")

list(APPEND CMAKE_MODULE_PATH ${CMAKE_SOURCE_DIR}/cmake/Modules)

enable_testing()
include(gmock)


if (NOT DEFINED     BOOST_ROOT        AND
    NOT DEFINED ENV{BOOST_ROOT}       AND
    NOT DEFINED     BOOST_INCLUDEDIR  AND
    NOT DEFINED ENV{BOOST_INCLUDEDIR} AND
    NOT DEFINED     BOOST_LIBRARYDIR  AND
    NOT DEFINED ENV{BOOST_LIBRARYDIR})
    if (APPLE)
        set(BOOST_ROOT "../../../boost/boost_1_54_0/mac")
    elseif (WIN32)
        set(BOOST_INCLUDEDIR "C:/local/boost_1_55_0")
        set(BOOST_LIBRARYDIR "C:/local/boost_1_55_0/lib32-msvc-10.0")
    endif()
endif()
if (APPLE OR WIN32)
    set(Boost_USE_STATIC_LIBS TRUE)
endif()
find_package(Boost 1.32 REQUIRED COMPONENTS program_options)
include_directories(SYSTEM ${Boost_INCLUDE_DIRS})

if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU" OR
    "${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
    set(warnings "-Wall -Wextra -Werror")
elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
    set(warnings "/W4 /wd4512 /WX /EHsc")
        # Disabled Warnings:
        #   4512 "assignment operator could not be generated"
        #        This warning provides no useful information and will occur in
```

```
#         well formed programs.
#         <http://msdn.microsoft.com/en-us/library/hsyx7kbz.aspx>
endif()
if (NOT CONFIGURED_ONCE)
    set(CMAKE_CXX_FLAGS "${warnings}"
        CACHE STRING "Flags used by the compiler during all build types." FORCE)
    set(CMAKE_C_FLAGS   "${warnings}"
        CACHE STRING "Flags used by the compiler during all build types." FORCE)
endif()


include_directories(${CMAKE_CURRENT_SOURCE_DIR})

add_subdirectory(ToDoCore)

add_executable(toDo main.cc)
target_link_libraries(toDo toDoCore ${Boost_LIBRARIES})


set(CONFIGURED_ONCE TRUE CACHE INTERNAL
    "A flag showing that CMake has configured at least once.")
```

Source

`find_package(Boost 1.32 REQUIRED COMPONENTS program_options)`

This command searches for Boost, both the headers and the boost_program_options library, and then defines variables that indicate whether or not Boost has been found and if so describe the locations of the libraries and header files.

`include_directories(SYSTEM ${Boost_INCLUDE_DIRS})`

Add the paths to Boost's include files to the compiler's include search paths.

By using the `SYSTEM` argument CMake will tell the compiler, if possible, that these paths contain system include files. Oftentimes the compiler will ignore warnings from files found in system include paths.

The `SYSTEM` option does not have an effect with all generators. When using the Visual Studio 10 or the Xcode generators neither Visual Studio nor Xcode appear to treat system include paths any differently than regular include paths. This can make a big difference when compiler flags are set to treat warnings as errors.

```
target_link_libraries(toDo ${Boost_LIBRARIES} toDoCore)
```

This links our little app, toDo, with the Boost libraries. In this case just `boost_program_options` since that's the only compiled library we requested. It also links toDo with our toDoCore library. Naturally we need this as that library implements all of our to do list functionality.

```
find_package(package [version [EXACT]] [REQUIRED] [COMPONENTS components…])
```

`package`

The name of the package to find, e.g. `Boost`. This name is case sensitive.

[version]

The desired version of the package.

`[EXACT]`

Match the version of the package exactly instead of accepting a newer version.

`[REQUIRED]`

Specifying this option causes CMake's configure step to fail if the package cannot be found.

`[COMPONENTS components…]`

Some libraries, like Boost, have optional components. The `find_package()` command will only search for these components if they have been listed as arguments when the command is called.

`find_package()` documentation (2014-11-14)

How to Use FindBoost

We glossed over how to use FindBoost before and actually we glossed over how `find_package()` really works. Naturally CMake can't know how to find any arbitrary package. So `find_package()`, as invoked above, actually loads a CMake Module file called `FindBoost.cmake` which does the actual work of

finding Boost. CMake installations come with a good complement of Find Modules. CMake searches for `FindBoost.cmake` just as it would any module included using the `include()` command.

The documentation for it can be obtained using the command `cmake --help-module FindBoost`.

`set(BOOST_ROOT "../../../boost/boost_1_54_0/mac")`

FindBoost uses the value of `BOOST_ROOT` as a hint for where to look. It will search in `BOOST_ROOT` as well as the standard places to look for libraries. In this example I did not install Boost in a standard location on my Mac so I needed to tell FindBoost where to look.

`set(BOOST_INCLUDEDIR "C:/local/boost_1_55_0")`

If your installation of boost is not stored in the "normal" folders, i.e. `include` and `lib`, you will need to specify the directory that contains the include files separately. Since libraries don't seem to have a standard installation location on Windows as they do on Linux we needed to tell FindBoost where Boost's header files are. Usually when providing `BOOST_INCLUDEDIR` `BOOST_ROOT` isn't needed. If you are using any of Boost's compiled libraries you will also need `BOOST_LIBRARYDIR`.

`set(BOOST_LIBRARYDIR "C:/local/boost_1_55_0/lib32-msvc-10.0")`

The same as `BOOST_INCLUDEDIR`, if specifying `BOOST_ROOT` doesn't find the libraries then you will have to specify the `BOOST_LIBRARYDIR`.

`set(Boost_USE_STATIC_LIBS TRUE)`

By default FindBoost provides the paths to dynamic libraries, however you can set `Boost_USE_STATIC_LIBS` to true so that FindBoost will provide the paths to the static libraries instead.

We want to use the static libraries on Mac OS X (`APPLE`) because when Boost is installed on the Mac the dynamic libraries are not configured properly and our app would not run if we were to link against them.

On Windows we are linking with static libraries so Visual Studio will look for the static Boost libraries. Since FindBoost normally provides the paths to Boost's dynamic libraries linking would fail. By specifying that we want the static libraries linking will succeed and we can use our new command line arguments.

There are several other variables that affect how FindBoost works, but they aren't needed as often. Consult the documentation for more information.

FindBoost documentation (2015-03-02)

`include_directories(SYSTEM ${Boost_INCLUDE_DIRS})`

We add the paths to where the Boost header files are. These assume that your include directives are of the canonical form `#include <boost/…>`. `Boost_INCLUDE_DIRS` is set for us by FindBoost.

`target_link_libraries(toDo ${Boost_LIBRARIES} toDoCore)`

The paths to all of the boost libraries we requested, i.e. `program_options`, are provided by FindBoost in the variable `Boost_LIBRARIES`. We simply link against the list of libraries provided.

FindBoost defines several other variables, which are listed in its documentation. The most important one, not used here, is `Boost_FOUND`. If Boost has been found then `Boost_FOUND` will be true, otherwise it will be false. Since we specified that Boost was `REQUIRED` we know that `Boost_FOUND` must be true otherwise CMake's configuration step would have failed. If Boost were not `REQUIRED` then `Boost_FOUND` would be an extremely important variable.

If we had chosen not to require Boost but not changed anything else in our `CMakeLists.txt` we would run into trouble if Boost had not been found. You would expect that our code wouldn't compile because an include file could not be found. As it turns out you won't actually get that far. FindBoost will set `Boost_INCLUDE_DIRS` to a value indicating that Boost was not found. Because of this the CMake configure step will fail because we use that variable as an include directory. Since CMake checks this for us we need to remember to be careful when using optional packages.

Choosing a Root

Typically `BOOST_ROOT` should be the directory that contains the `include` and `lib` directories in which you will find boost. Remember the boost headers will be inside a `boost` directory. As you might notice this is the standard layout used on Unix and Linux. When the headers and libraries are not arranged this way, as is likely on Windows, the `BOOST_INCLUDEDIR` and `BOOST_LIBRARYDIR` should be used instead.

So right now you are probably wondering what use FindBoost really is if I had to specify the root, or worse the include and library directories. Well there are a few reasons:

- Most importantly if Boost has been installed in a standard location it would have been found without any information being provided.
- It will check that the Boost it finds is the desired version, 1.32 or greater in this case. Not all finders actually check version, but when available this feature is very useful as incorrect library versions are caught immediately rather than later through potentially confusing compile errors.
- In the case of Boost the finder will ensure the desired libraries are found. Since approximately 90% of the Boost libraries are header only some installs only include the headers and none of the compiled libraries.
- Lastly even though I specified my non-standard install locations for Boost in the `CMakeLists.txt` you needn't install it there. Regardless FindBoost will still find Boost if you have it installed in a standard location. Additionally you can set your own location using by setting the `BOOST_ROOT` variable using the `-D` command line option of `cmake` or by setting it using the GUI or curses interface. Perhaps most conveniently you can set the `BOOST_ROOT` environment variable and not need to tell CMake separately. This, of course, applies to the `BOOST_INCLUDEDIR` and `BOOST_LIBRARYDIR` variables, too.

So this leaves one question: does it make sense to set `BOOST_ROOT` in the `CMakeLists.txt`?

If you are the only one working on the project then it will certainly be easier to set it in the `CMakeLists.txt`, although you will have to do this for every project. Setting the environmental variable might be easier.

If you work on a team whose development machines are all configured similarly, or should be, then setting `BOOST_ROOT` in the `CMakeLists.txt` is a good idea because it simplifies things for most developers and therefore provides and incentive for all developers to use the standard configuration.

Now if you work with a disparate group of people, say on an free/open source project, it makes less

sense to set `BOOST_ROOT` in the `CMakeLists.txt` as there is likely no notion of a standard development environment.

Finding Packages

Since CMake ships with a reasonable number of Find modules there's a good chance that whatever you want to find can be found by simply using the `find_package` command. While you should review the documentation for that particular module there are some variables that you can expect to be defined.

`Package_FOUND`
: This variable indicates whether or not the package has been found.

`Package_INCLUDE_DIRS`
: The include directories for that particular package. This variable should be passed to the `include_directories()` command.

`Package_LIBRARIES`
: The full paths to this package's libraries. This variable should be passed to the `target_link_libraries()` command.

`Package_DEFINITIONS`
: Definitions required to compile code that uses this package. This should be passed to the `add_definitions()` command.

Documentation Found

As mentioned above you can get the documentation for FindBoost by using the `cmake` command. While this is somewhat convenient the terminal is not always the best tool for reading documentation. There is a slightly more useful variant of the command: `cmake --help-module FindBoost file`. This allows you to read the documentation however you please.

There's another convenient command that will list all of the available modules: `cmake --help-modules`.

This will also provide some documentation for each. Again you can easily save this to a file with the command `cmake --help-modules file`.

If you have a Unix/Linux-like shell then you can easily get a list of all available Find modules.

```
 > cmake --version
cmake version 2.8.12.1
 > cmake --help-modules | grep -E "^  Find"
  FindALSA
  FindASPELL
  FindAVIFile
  FindArmadillo
  FindBISON
  FindBLAS
  FindBZip2
  FindBoost
  FindBullet
  FindCABLE
  FindCUDA
  FindCURL
  FindCVS
  FindCoin3D
  FindCups
  FindCurses
  FindCxxTest
  FindCygwin
  FindDCMTK
  FindDart
  FindDevIL
  FindDoxygen
  FindEXPAT
  FindFLEX
  FindFLTK
  FindFLTK2
```

```
FindFreetype
FindGCCXML
FindGDAL
FindGIF
FindGLEW
FindGLUT
FindGTK
FindGTK2
FindGTest
FindGettext
FindGit
FindGnuTLS
FindGnuplot
FindHDF5
FindHSPELL
FindHTMLHelp
FindHg
FindITK
FindIcotool
FindImageMagick
FindJNI
FindJPEG
FindJasper
FindJava
FindKDE3
FindKDE4
FindLAPACK
FindLATEX
FindLibArchive
FindLibLZMA
FindLibXml2
FindLibXslt
FindLua50
FindLua51
FindMFC
```

```
FindMPEG
FindMPEG2
FindMPI
FindMatlab
FindMotif
FindOpenAL
FindOpenGL
FindOpenMP
FindOpenSSL
FindOpenSceneGraph
FindOpenThreads
FindPHP4
FindPNG
FindPackageHandleStandardArgs
FindPackageMessage
FindPerl
FindPerlLibs
FindPhysFS
FindPike
FindPkgConfig
FindPostgreSQL
FindProducer
FindProtobuf
FindPythonInterp
FindPythonLibs
FindQt
FindQt3
FindQt4
FindQuickTime
FindRTI
FindRuby
FindSDL
FindSDL_image
FindSDL_mixer
FindSDL_net
```

```
FindSDL_sound
FindSDL_ttf
FindSWIG
FindSelfPackers
FindSquish
FindSubversion
FindTCL
FindTIFF
FindTclStub
FindTclsh
FindThreads
FindUnixCommands
FindVTK
FindWget
FindWish
FindX11
FindXMLRPC
FindZLIB
Findosg
FindosgAnimation
FindosgDB
FindosgFX
FindosgGA
FindosgIntrospection
FindosgManipulator
FindosgParticle
FindosgPresentation
FindosgProducer
FindosgQt
FindosgShadow
FindosgSim
FindosgTerrain
FindosgText
FindosgUtil
FindosgViewer
```

```
FindosgVolume
FindosgWidget
Findosg_functions
FindwxWidgets
FindwxWindows
```

This entry was tagged CMake, long, tutorial. Bookmark the permalink.

← CMake Tutorial – Chapter 5: Functionally Improved Testing

11 thoughts on “CMake Tutorial – Chapter 6: Realistically Getting a Boost”

2015-03-19 at 21:59:16

blaskovich said:

Awesome, great to see you’re at this again!

Reply

2015-04-08 at 06:32:49

Roger Wehage said:

John, you are a true pioneer. I purchased the CMake book about three years ago and learned almost nothing from it, which means that the “book” has been collecting cobwebs on my Macintosh. Now that I really must use it, I debated purchasing a newer version of the book, hoping that it has been improved. But according to the reviews on Amazon, the “book” is still terrible; and the CMake website hasn’t been of much use either. Imagine