JohnLamp.net

Search …

Home CMake Tutorial

← CMake Tutorial – Chapter 2: IDE Integration

CMake Tutorial – Chapter 4: Libraries and Subdirectories →

CMake Tutorial – Chapter 3: GUI Tool

Posted on 2013-03-28 by John Lamp

Contents

Introduction

Although when we looked at IDE projects generated by CMake we still used the command line. You can also use the CMake GUI to generate and configure projects. This can be convenient if you don't like

Archives
March 2015
July 2013
May 2013
March 2013

Categories
CMake Tutorial

Meta
Log in
Entries RSS
Comments RSS
WordPress.org

the command line, however it can be even more useful than that.

CMake stores a lot of configuration settings in the project's cache. This cache can be viewed and edited using the CMake GUI. This can be quite useful for seeing how a project is configured as the settings are presented in a nice list. You can also change these values so you can set your build type to "Release" to make a release build or you can add specific compiler flags.

First Fix a Warning

In chapter 2 when covering the Xcode generator I said that I'd fix the warning we saw later. Well it looks like later has come. The first thing we need to do is give the compiler some more flags so that we can reproduce the warning.

CMakeLists.txt New or modified lines in bold.

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
set(CMAKE_LEGACY_CYGWIN_WIN32 0)

project("To Do List")

enable_testing()


if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU" OR
    "${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
    set(warnings "-Wall -Wextra -Werror")
elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
    set(warnings "/W4 /WX /EHsc")
endif()
set(CMAKE_C_FLAGS   "${CMAKE_C_FLAGS} ${warnings}")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${warnings}")

add_executable(toDo main.cc
                    ToDo.cc)

add_test(toDoTest toDo)
```

`if(...), elseif(...), else(), endif()`

While everything in CMake looks like a function call control flow is available. Its `if` syntax is rather strange so be sure to keep the documentation (2013-01-07) handy. The arguments passed to `else()` and `endif()` are ignored, but they can be useful for documentation purposes.

`CMAKE_<LANG>_COMPILER_ID`

These variables identify the type of compiler being used. Here we are using it to be able to pass different flags to different compilers as needed. Since Clang accepts the same arguments as GCC I grouped them together. A list of possible values is provided by the documentation (2013-01-07). Obviously my if statement is not exhaustive as it only covers the 3 compilers I have readily available.

`set(variableName value…)`

Set a variable with the given name to a particular value or list of values. (Lists will be covered later)

set() documentation (2013-03-26)

`CMAKE_<LANG>_FLAGS`

These variables store the flags that will be passed to the compiler for all build types. In this particular case we wanted to add some flags that control warnings. (Build types will be covered later in this chapter.)

Note: This variable is a string containing all of the flags separated by spaces; it is **not** a list.

In this case we are turning on most warnings and having the compiler treat them as errors. (This is, in fact, Microsoft's suggestion for all new projects.) Since we only want to add these options we append them to the end of the existing flags string.

CMake does offer some string functions, but not for something as simple as appending to an existing string.

A few notes about MSVC: The `/EHsc` flag enables complete C++ exception handling which is required by `iostream`. (`/EH` documentation 2013-04-13) More importantly is that CMake will convert Unix-style flags to Microsoft-style flags automatically for you. So we could have used `"-W4 -WX -EHsc"` instead and it would have worked. This means that any common flags do not need to be defined separately for MSVC. I would, however, recommend always using Microsoft-style flags for MSVC specific flags. Then not only is it obvious that they are MSVC flags, but they are also easier to look up since you won't have to remember to translate them yourself.

Now if we build not only should we see more warnings and since they are being treated as errors they should also prevent the build from completing. Since warnings usually point to potential problems I always set up my `CMakeLists.txt` to enable stricter warnings and treat them as errors. Developing this way can be a bit annoying, but in the long run it will lead to cleaner code and, in theory, fewer defects.

```
 > mkdir build
 > cd build
 > cmake -G "Unix Makefiles" ..
-- The C compiler identification is Clang 4.1.0
-- The CXX compiler identification is Clang 4.1.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Volumes/Documents/Programming/C++/CMake Tutorial
 > make
Scanning dependencies of target toDo
[ 50%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
```

```
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/mac/part3/main.cc:22:12: err
      unused parameter 'argc' [-Werror,-Wunused-parameter]
    int    argc,
           ^
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/mac/part3/main.cc:23:12: err
      unused parameter 'argv' [-Werror,-Wunused-parameter]
    char** argv
           ^
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/mac/part3/main.cc:58:19: err
      comparison of integers of different signs: 'const unsigned long' and
      'const int' [-Werror,-Wsign-compare]
    if (testValue != expectedValue)
        ~~~~~~~~~ ^  ~~~~~~~~~~~~~
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/mac/part3/main.cc:34:15: not
      in instantiation of function template specialization
      'equalityTest<unsigned long, int>' requested here
    result |= EXPECT_EQUAL(list.size(), 3);
              ^
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/mac/part3/main.cc:8:36: note
      expanded from macro 'EXPECT_EQUAL'
#define EXPECT_EQUAL(test, expect) equalityTest( test,  expect, \
                                   ^

3 errors generated.
make[2]: *** [CMakeFiles/toDo.dir/main.cc.o] Error 1
make[1]: *** [CMakeFiles/toDo.dir/all] Error 2
make: *** [all] Error 2
```

This time CMake found Clang and with our new flags we have 3 errors. (Rather nice errors, actually.) These errors are actually simple to fix, so lets fix them before we move on.

main.cc New or modified lines in bold.

```
#include <iostream>
  using std::cerr;
```

```
  using std::cout;
  using std::endl;

#include "ToDo.h"

#define EXPECT_EQUAL(test, expect) equalityTest( test,  expect, \
                                                 #test, #expect, \
                                                 __FILE__, __LINE__)

template < typename T1, typename T2 >
int equalityTest(const T1    testValue,
                 const T2    expectedValue,
                 const char* testName,
                 const char* expectedName,
                 const char* fileName,
                 const int   lineNumber);


int main(
    int,
    char**
)
{
    int result = 0;

    ToDo list;

    list.addTask("write code");
    list.addTask("compile");
    list.addTask("test");

    result |= EXPECT_EQUAL(list.size(),     size_t(3));
    result |= EXPECT_EQUAL(list.getTask(0), "write code");
    result |= EXPECT_EQUAL(list.getTask(1), "compile");
    result |= EXPECT_EQUAL(list.getTask(2), "test");

    if (result == 0)
    {
        cout << "Test passed" << endl;
    }

    return result;
}


template < typename T1, typename T2 >
int equalityTest(
    const T1    testValue,
    const T2    expectedValue,
    const char* testName,
    const char* expectedName,
    const char* fileName,
    const int   lineNumber
)
{
    if (testValue != expectedValue)
    {
        cerr << fileName << ":" << lineNumber << ": "
             << "Expected " << testName << " "
             << "to equal " << expectedName << " (" << expectedValue << ") "
             << "but it was (" << testValue << ")" << endl;

        return 1;
    }
    else
    {
        return 0;
    }
}
```

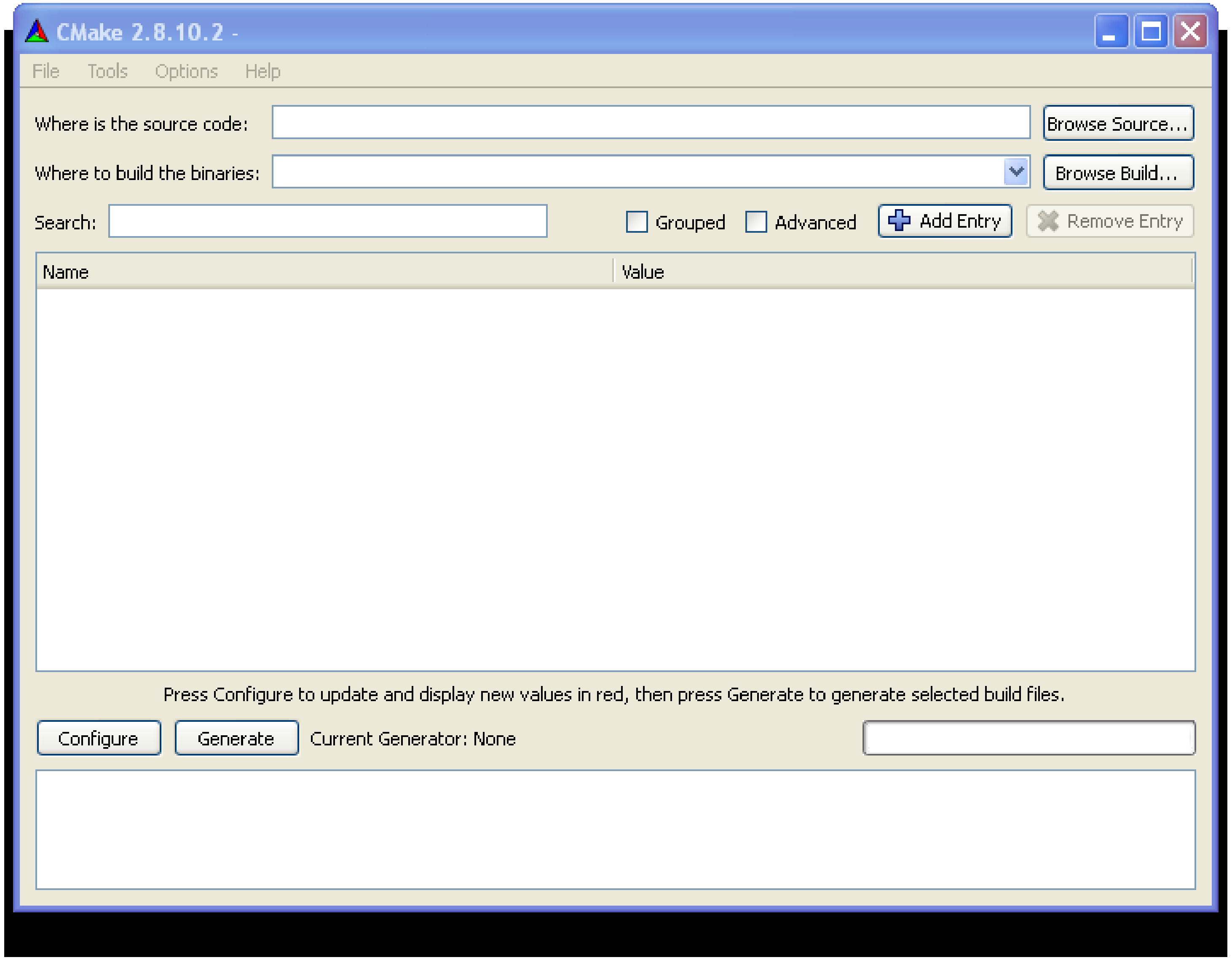
Source

They were rather simple errors to fix. The simplest solution to unused function parameters is to delete their names leaving only the types, if it's temporary just comment them out. This documents both for other people and the compiler that the parameters aren't being used. The last error is caused by literal numbers defaulting to being `int`s. If we construct a `size_t` the problem is fixed.

CMake GUI

Generating Our Project

The CMake GUI allows one to easily run CMake without having to use the command line. It also makes it easier to set or change specific options, which we will explore.



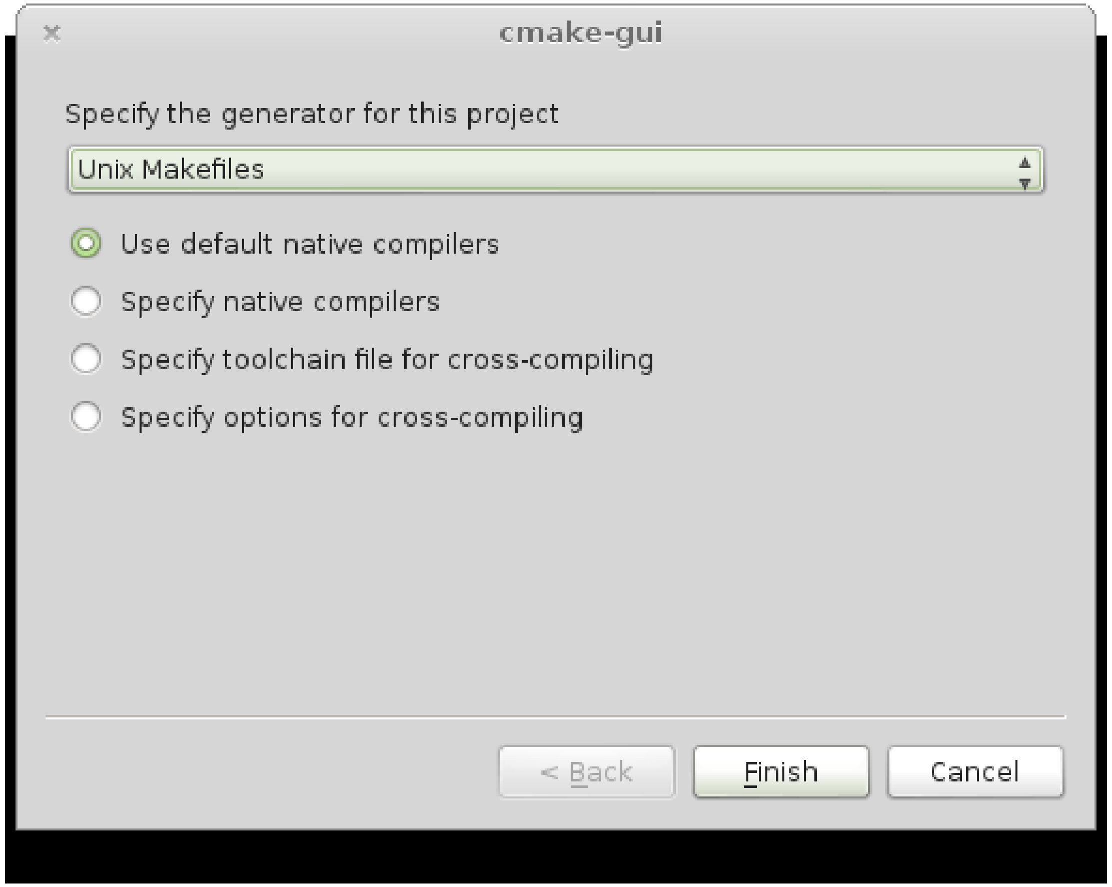
Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: None

The first two entries should be familiar, but more explicit than what we saw earlier. To relate to the command line we were using:
`cd <Where to build the binaries>; cmake <Where is the source code>`. That command line also

configures and generates, which you would do using the “Configure” and “Generate” buttons, of course. The bulk of the window is for variables, which are only visible once you have configured.

It isn’t quite that simple, though. Once you pick your source and build directories and then click “Configure” CMake will ask you about which generator you want to use and more.


cmake-gui

Specify the generator for this project

Unix Makefiles

[✓ Use default native compilers]

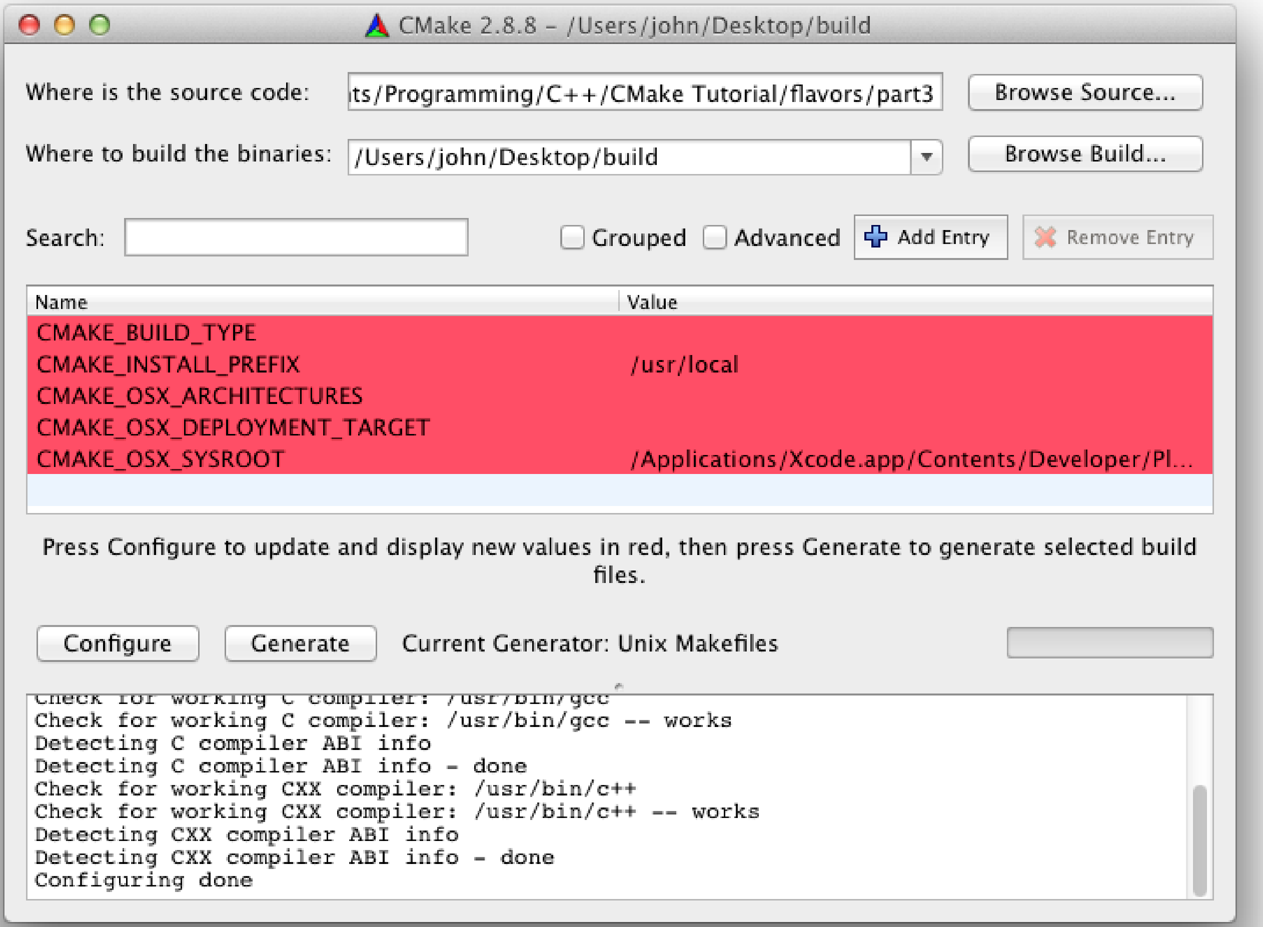
Specify native compilers

Specify toolchain file for cross-compiling

Specify options for cross-compiling

The displayed options are the typical ones used so far during this tutorial. Generate Unix Makefiles and use the default native compilers. A different generator can be chosen from the list rather than having to carefully type it, which can be handy. The other options allow you to specify which compiler to use, a topic that will be covered later. Clicking “Finish” will then actually configure.

Note: This step can only be done the first time, so if you want to use a different generator (or compiler) you will have to start over with an empty build directory.



Where is the source code: ts/Programming/C++/CMake Tutorial/flavors/part3

Where to build the binaries: /Users/john/Desktop/build

Name	Value
CMAKE_BUILD_TYPE	
CMAKE_INSTALL_PREFIX	/usr/local
CMAKE_OSX_ARCHITECTURES	
CMAKE_OSX_DEPLOYMENT_TARGET	
CMAKE_OSX_SYSROOT	/Applications/Xcode.app/Contents/Developer/Pl...

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Unix Makefiles

```
Check for working C compiler: /usr/bin/gcc
Check for working C compiler: /usr/bin/gcc -- works
Detecting C compiler ABI info
Detecting C compiler ABI info - done
Check for working CXX compiler: /usr/bin/c++
Check for working CXX compiler: /usr/bin/c++ -- works
Detecting CXX compiler ABI info
Detecting CXX compiler ABI info - done
Configuring done
```

Notice that the bottom section displays the same output the `cmake` command displays when configuring.

There are also now some variables displayed in the central portion of the window. In this example most are specific to Mac OS X. The variables' values can easily be changed by double clicking in the "Value" field and entering a new value.

`CMAKE_BUILD_TYPE`

This variable controls the type of build to be done. The possible values are empty, `Debug`, `Release`, `RelWithDebInfo`, and `MinSizeRel`. The values' meanings are relatively obvious. Based upon the value of this variable CMake will set the compiler flags appropriately. This is done by adding the value of the variable `CMAKE_<LANG>_FLAGS_<BUILD_TYPE>` to `CMAKE_<LANG>_FLAGS`. By setting these variables appropriately you can control the compiler flags for the various types of builds.

Note: This variable is not available with all generators. Some IDE generators create non-Makefile projects, e.g. Visual Studio, in which case the build type is handled by the IDE itself.

CMAKE_BUILD_TYPE Documentation 2013-01-20

CMAKE_INSTALL_PREFIX

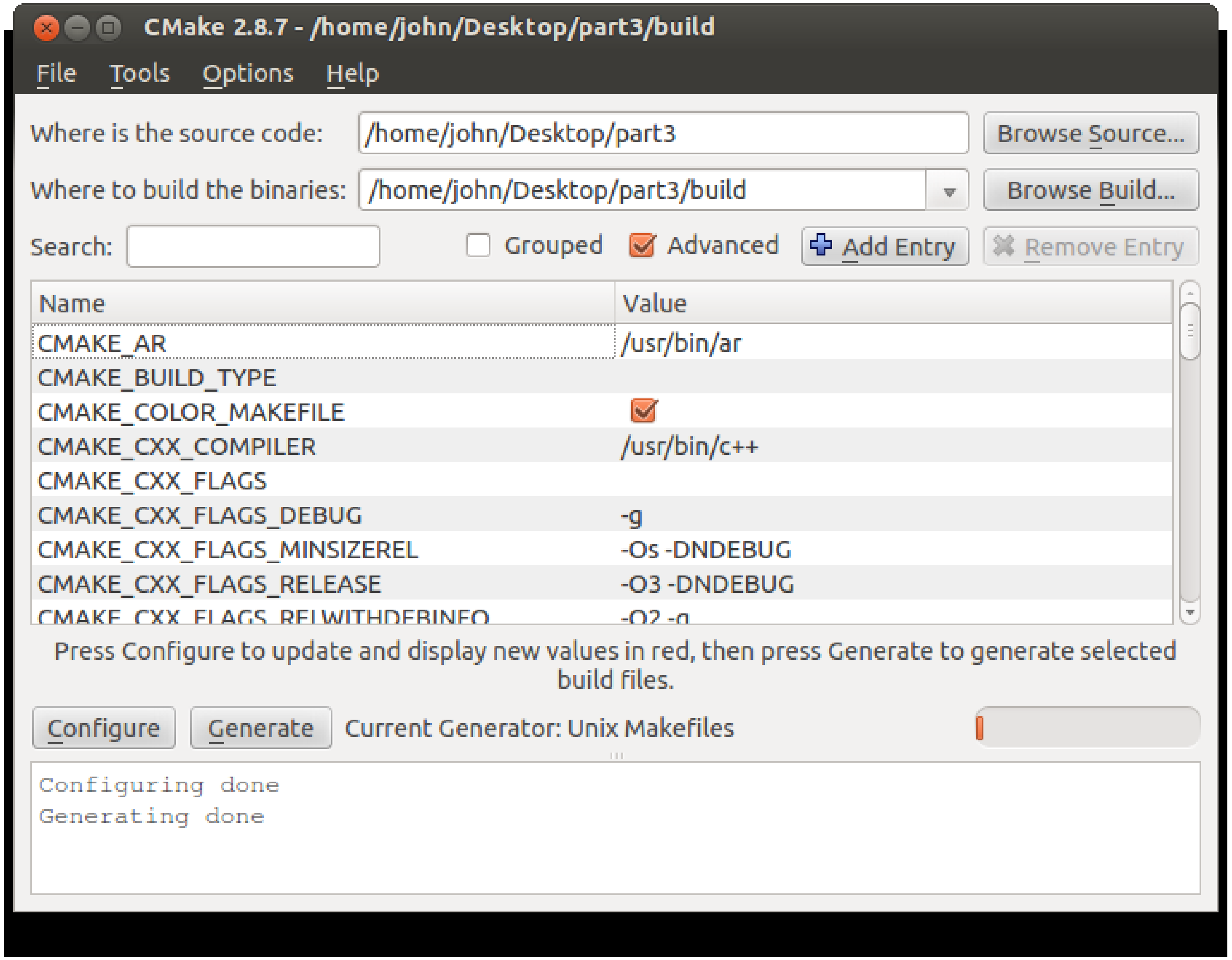
CMake can create an install target which will be covered in a future chapter. This prefix can be set to control where things are installed. It is similar to the `--prefix` argument for `configure` scripts.

However if you are curious: CMAKE_INSTALL_PREFIX Documentation 2013-01-20

Simply click "Configure" again as directed. Clicking "Generate" will then generate our Makefile so we can build.

CMake Cache

If you check the "Advanced" box all cache variables will be listed.



Where is the source code: /home/john/Desktop/part3

Where to build the binaries: /home/john/Desktop/part3/build

Search: ☐ Grouped ☑ Advanced

Name	Value
CMAKE_AR	/usr/bin/ar
CMAKE_BUILD_TYPE	
CMAKE_COLOR_MAKEFILE	✓
CMAKE_CXX_COMPILER	/usr/bin/c++
CMAKE_CXX_FLAGS	
CMAKE_CXX_FLAGS_DEBUG	-g
CMAKE_CXX_FLAGS_MINSIZEREL	-Os -DNDEBUG
CMAKE_CXX_FLAGS_RELEASE	-O3 -DNDEBUG
CMAKE_CXX_FLAGS_RELWITHDEBINFO	-O2 -g

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Unix Makefiles

```
Configuring done
Generating done
```

CMake stores variables that need to be persistent in the cache. These include things such as the path to your compiler and the flags for the compiler. Naturally one should be careful when editing variables in the cache.

You will notice that the compiler flags we added earlier do not appear in the cache. While this might be a good idea as it forces those options to always be used it really isn't correct. We can tell `set()` to put the variable in the cache, however it's not that simple. Either the cache will never be updated or our options will be appended *every* time CMake configures.

The following should do the trick:

CMakeLists.txt New or modified lines in bold.

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
set(CMAKE_LEGACY_CYGWIN_WIN32 0)

project("To Do List")

enable_testing()


if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU" OR
    "${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
    set(warnings "-Wall -Wextra -Werror")
elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
    set(warnings "/W4 /WX /EHsc")
endif()
if (NOT CONFIGURED_ONCE)
    set(CMAKE_CXX_FLAGS "${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()


add_executable(toDo main.cc
                    ToDo.cc)

```

```
add_test(toDoTest toDo)


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

Source

`if (NOT CONFIGURED_ONCE)`

In CMake an undefined variable evaluates to false. Because of this we can use `CONFIGURED_ONCE` as a flag to determine if CMake has configured this project at least once.
Defined variables that are empty or contain `0`, `N`, `NO`, `OFF`, `FALSE`, `NOTFOUND` or `variable-NOTFOUND` are also considered false.

`set(CMAKE_CXX_FLAGS “${warnings}” CACHE STRING “Flags used by the compiler during all build types.” FORCE)`

Initialize the value of `CMAKE_CXX_FLAGS` to be the desired warning flags. The syntax for this form of the `set` command is explained below. Two things to note:

1. The docstring is exactly what CMake uses by default. When overriding built-in CMake variables be sure to use the same docstring as it does to avoid confusion.
2. We need to force this value to be stored in the cache because the built-in variables are present in the cache even before the first time our project is configured. This is why we need the `CONFIGURED_ONCE` variable.

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

Set `CONFIGURED_ONCE` to true and store it in the cache since by now configuration is complete. We don’t need to force this as `CONFIGURED_ONCE` is not present in the cache.

A new form of the `set` command was used this time to store variables in the CMake project’s cache. It is explained here and also in CMake’s documentation (2013-01-29)

`set(variableName value … CACHE type docstring [FORCE])`

This form of the `set` function allows you to store a variable in CMake's cache. The cache is both global and persistent. For both of these reasons it can be quite useful and should be used carefully. The other important thing about the cache is that users can, for the most part, edit it. The `CACHE` flag is a literal that tells CMake you want to store this variable in the cache.

`type`
: The type of value being stored in the cache. Possible values:

  `FILEPATH`
  : A path to a file. In the CMake GUI a file chooser dialog may be used.

  `PATH`
  : A path to a directory. In the CMake GUI a directory chooser dialog may be used.

  `STRING`
  : An arbitrary string.

  `BOOL`
  : A boolean on/off value. In the CMake GUI a checkbox will be used.

  `INTERNAL`
  : A value of any type with no GUI entry. This is useful for persistent, global variables.

`docstring`
: A string that describes the purpose of the variable. If only specific values are allowed list them here as the user will see this string in the CMake GUI as a tool tip.

`FORCE` (optional)
: Force this entry to be set in the cache. Normally if a variable already exists in the cache future attempts to set it will be ignored unless `FORCE` is the last argument. Please note that setting a variable in the cache is dependent on the variable already being in the cache not on its emptiness. Because of this and the fact that many of the CMake variables exist in the cache before your `CMakeLists.txt` is processed you need to test for the first configuration as done above.

CMake Curses Interface

Introducing `ccmake`

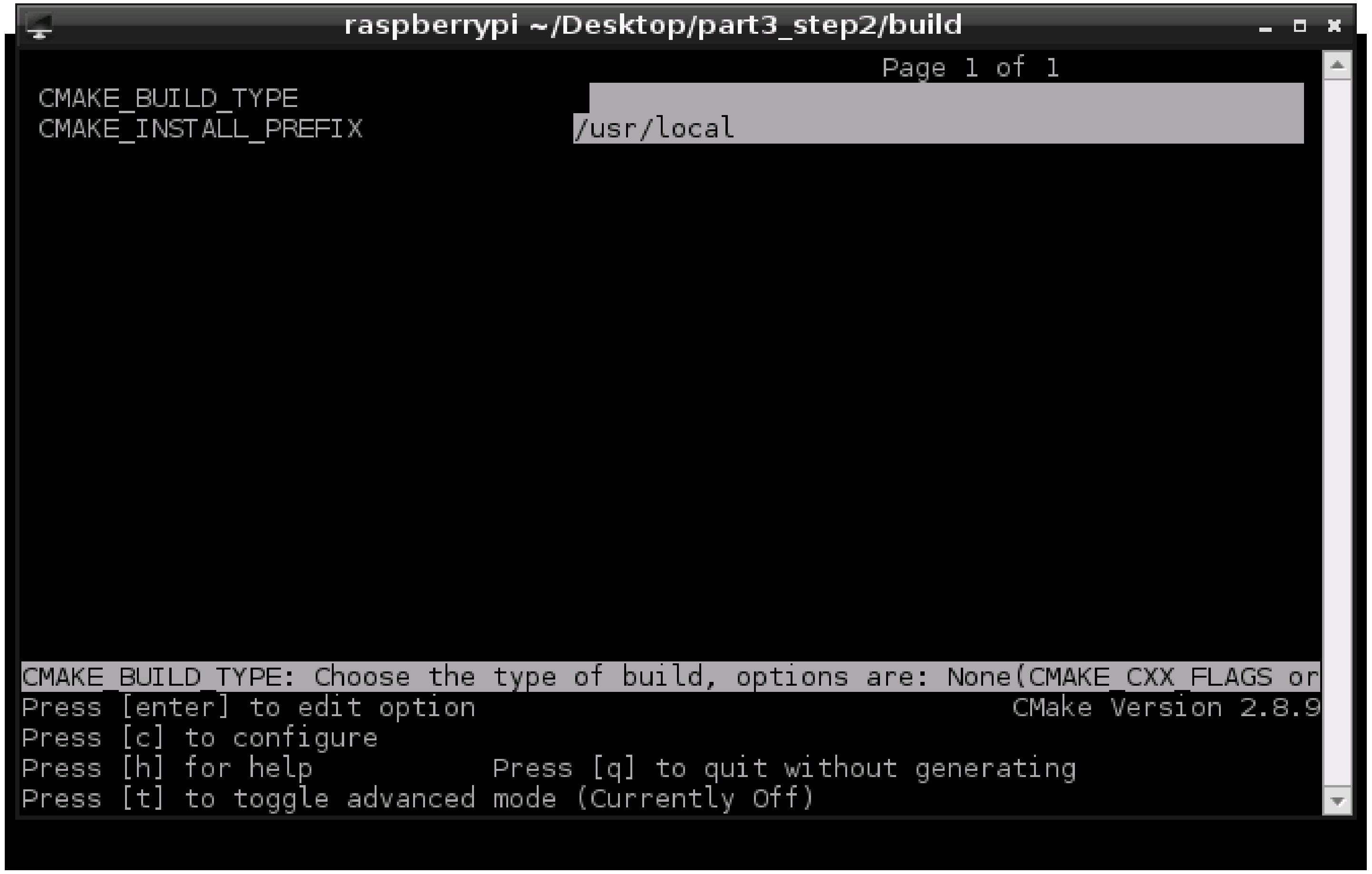
CMake also includes a command line curses-based interface, `ccmake`. It provides equivalent functionality to that of the GUI. Most installations include this tool, although not all. The `ccmake` tool can be used both to create a CMake build or edit the cache of one. To create a new build it is used very similarly to `cmake`:

```
ccmake path-to-source
```

Naturally editing a build's cache is quite similar:

```
ccmake path-to-existing-build
```

For the most part this tool is very much like the GUI except, of course, its interactions are all keyboard based. It can be useful if you often connect to your build machine via an ssh session or you don't want the dependency of Qt, which the GUI requires.



```
                                                Page 1 of 1
 CMAKE_BUILD_TYPE
 CMAKE_INSTALL_PREFIX            /usr/local









CMAKE_BUILD_TYPE: Choose the type of build, options are: None(CMAKE_CXX_FLAGS or
Press [enter] to edit option                                CMake Version 2.8.9
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

The main difference between this tool and the GUI is that it won’t walk you through setting up a build, you have to provide paths on the command line. Besides that its features are mostly the same. Of course, instead of clicking the “Configure” and “Generate” buttons you would use the `c` and `g` keys.

Useful Makefile Targets

There are two built-in make targets that CMake provides that are useful for managing the cache. These are especially useful if you work from the command line a lot.

`make rebuild_cache`
: This target re-runs CMake for your build having the same effect as `cmake .`, this can be handy, though, if you have multiple versions of CMake installed or don't have `cmake` in your path as this target knows the path to the `cmake` that was originally used to generate the build.

`make edit_cache`
: Very similar to the above target except this one runs the appropriate `ccmake`, or `cmake-gui` if `ccmake` isn't installed. The reasons for this being useful are the same, too.

Most of the time these targets aren't used, but as they can be handy it's good to know about them.

There is one last Makefile target that is useful, especially on larger projects: `make help`. This prints a list of targets provided by the Makefile. This can be convenient if you only want to build specific targets but aren't sure how they were named.

Revision History

Version	Date	Comment
1	2013-03-28	Original version.
2	2013-04-14	Updated MSVC compiler flags and added note about automatic flag-style conversion.
3	2013-07-14	Added line numbers and indication of changes to code sample. Added a link to the section on lists.

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

← CMake Tutorial – Chapter 2: IDE Integration

CMake Tutorial – Chapter 4: Libraries and Subdirectories →