CMake Tutorial

JohnLamp.net

Table of Contents

Introduction

What is CMake?

According to CMake's creators, Kitware, CMake is an open-source cross platform build system. This is not completely accurate as CMake is not actually a build system. What CMake provides is an easy way to build C/C++ projects across platforms. The reason I say that CMake isn't a build system is because it doesn't actually build software. "A build system that *doesn't* build software?" you ask. Yes; what CMake does is generate a configuration for your existing build system, e.g. Make. This allows CMake to focus on things that most build systems don't; such as cross platform configuration, dependency calculation, testing, packaging, and installation.

Why CMake?

By not being a true build system, per se, CMake allows for a more flexible development environment as it can generate Makefiles or projects for a variety of IDEs. This allows developers to easily work on different platforms using different tools since one can build using Microsoft's Visual Studio on Windows or with GNU Make on Linux just as easily.

CMake also includes tools for finding libraries, e.g. boost, and the ability to easily include external projects in your build. These two features, in particular, make it much simpler to build projects that have external dependencies and by using the find tools rather than hard coding paths it is much easier for new developers to get started on an existing project.

Included with CMake is CTest, a test driver program. Both work together to make it easy to run a project's test programs and/or scripts. When you configure your project you specify how to run your tests and CMake generates a configuration for CTest. CTest will run all of your tests and provide a summary of which ones passed and which ones failed. In addition it logs the output of all the tests it ran. Optionally CTest can be directed to run only specific tests or skip specific tests, perhaps the slow ones. While it may not be a continuous build system you have most of the components provided.

In addition to setting up a build CMake can also create an install target that will install the outputs of your project in the appropriate locations. Once you have configured your project to be installed you can also package your project using the included CPack utility. A variety of packages can be created including tar files, zip files, or an installer.

Acknowledgements

I would like to thank the following people for their help and contributions to this tutorial. Without them it would not exist.

Devin Ronge

This tutorial would not exist without Devin, he suggested I write it and motivated me to start. Despite being primarily a C# and JavaScript developer Devin has read every word of this tutorial at least once. Thanks to him you get a better written tutorial than you would had he not proof-read it first.

Steve Rieman

As a C++ developer who actively uses CMake Steve has provided a technical review of the sample code in addition to a review of the prose. He has also provided numerous ideas for the contents of this tutorial.

Chapter 1: Getting Started

Introduction

In this chapter we start by installing CMake. Like most open source software the best way to do this depends on your platform and how you usually do things. Once we have CMake installed we create a simple project. Perhaps it's a little fancier than "hello world" but not much. We finish up with the test support built into CMake.

I won't cover any particular aspect of CMake in great detail yet. That will be left for future chapters. However, after this chapter you will know enough to build simple programs with CMake and run simple tests with CTest.

Installation

Windows

Download and Install
: Download the installer from the CMake website (2012-06-02). Run the installer and follow its steps. Be sure to add CMake to the system `PATH` so that you can use it from the command line. Add it for the current or all users as appropriate.

  This provides both the `cmake` command and the CMake GUI (`cmake-gui`) but not the curses interface (`ccmake`).

Cygwin
: CMake can, of course, be installed as part of Cygwin. Even if you don't already have Cygwin installed you may want to as it provides a Linux-like environment natively in Windows. This way common Linux tools and utilities can be available. Also most of this tutorial is done in a Linux-like environment, so with Cygwin installed it will be easier to follow along.

  Download Cygwin's `setup.exe` from their website (2012-06-02). Run `setup.exe`. Follow its steps until you can select packages, then either chose to install all packages or just CMake. To install all packages click the word "Default" next to "All" until it reads "Install". If you don't want to install everything click the word "Default" next to "Devel" until it reads "Install"; this will install just the development tools. If you chose to install all packages the install will take a a few hours, but even just installing the development tools will take at least half an hour. After the installer has finished the Cygwin environment can then be accessed via the `Cygwin Terminal` which can be found in the Start Menu.

  This provides the `cmake` command and the curses interface (`ccmake`) but not the CMake GUI.

Mac OS X

Download and Install
: Download the disk image from the CMake website (2012-06-02). Pick the correct download for whichever version of OS X you are using. Use the installer and follow its directions. It will ask if you want it to make the command line tools available in your path by creating symbolic links, have it do so.

  This provides the `cmake` command, the CMake GUI (`CMake.app`), and the curses interface (`ccmake`).

Homebrew
: If you already have homebrew installed you can simply install CMake with the command `brew install cmake`.

  This provides the `cmake` command and the curses interface (`ccmake`) but **not** the CMake GUI.

Linux

Ubuntu (Debian)
: The simplest way to install CMake is via the command line: `sudo apt-get install cmake`. However, searching for CMake in the Ubuntu Software Center or in the Synaptic Package Manager, depending upon your Ubuntu version, will find the `cmake` package. If your Ubuntu install doesn't include X or you primarily use ssh sessions you will also want to install the `cmake-curses-gui` package. Again this is simplest with the command `sudo apt-get install cmake-curses-gui`, but either GUI interface can be used instead.

This provides the `cmake` command and the CMake GUI (`cmake-gui`). The second, optional, package provides the curses interface (`ccmake`).

Red Hat/CentOS

To install CMake via the command line is straightforward. First use `yum search cmake` to find the correct package to install. On a 64 bit install it would be `cmake.x86_64`. Use whichever package your search found when installing: `sudo yum install cmake.x86_64`. If `sudo` is not setup use `su` first and then run `yum install cmake.x86_64`.

This provides the `cmake` command and the curses interface (`ccmake`), but not the CMake GUI.

Fedora

Either the command line or the Add/Remove Software GUI can be used. In the GUI simply search for cmake and install at least the `cmake` module. If you desire the CMake GUI as well install the `cmake-gui` module. From the command line use `sudo yum install cmake` and `sudo yum install cmake-gui`, if you desire the GUI as well.

This provides the `cmake` command and the curses interface (`ccmake`). The second, optional, package provides the CMake GUI (`cmake-gui`).

Source

As CMake is an open source tool you can, of course, download the source code and build it yourself. However, that is outside the scope of this tutorial.

Hands On

For this tutorial we will create a To Do List program. Naturally our focus will be on CMake more than the actual code and its functionality. Most examples will be done using the command line generating Makefiles. CMake can be used with a GUI (chapter 3) and also generate projects for many IDEs (chapter 2).

Diving In

Just as any IDE has project files or Make has Makefiles CMake has `CmakeLists.txt` files. These describe your project to CMake and affect its output. They are fairly simple especially compared to Makefiles. Here's our first `CMakelists.txt`:

CMakeLists.txt

```
1 project("To Do List")
2
3
4 add_executable(toDo main.cc
5                   ToDo.cc)
```

`project(name)`

The `project` command names your project. Optionally you can specify what language the project supports, any of `CXX`, `C`, `JAVA`, or `FORTRAN`. CMake defaults to `C` and `CXX` so if you do not have compilers for C++ installed you may need to specify the language supported so that CMake doesn't search for it.

Note: If your project name contains spaces it must be surrounded by quotes.

project() documentation (2013-03-26)

`add_executable(target sources…)`

This command tells CMake you want to make an executable and adds it as a target. The first argument is the name of the executable and the rest are the source files. You may notice that header files aren't listed. CMake handles dependencies automatically so headers don't need to be listed.

add_executable() documentation (2013-03-26)

Of course we need some source code to build, so we will start with the simplest skeleton possible:

main.cc

```
#include "ToDo.h"

int main(
    int    argc,
    char** argv
)
{
    ToDo list;

    return 0;
}
```

ToDo.h

```
#ifndef TODO_H
#define TODO_H

class ToDo
{
public:
    ToDo();
    ~ToDo();
};

#endif // TODO_H
```

ToDo.cc

```
#include "ToDo.h"


ToDo::ToDo()
{
}

ToDo::~ToDo()
{
}
```

CMake's documentation strongly suggests that out-of-source builds be done rather than in-source builds. I agree as it makes it much easier to convince yourself that your build has really been cleaned since you can simply delete the build folder and start over. Building with CMake is actually rather simple, so we will charge ahead:

```
 > mkdir build
 > cd build
 > cmake -G "Unix Makefiles" ..
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler identification is GNU 4.2.1
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Checking whether C compiler supports OSX deployment target flag
-- Checking whether C compiler supports OSX deployment target flag - yes
-- 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
-- Checking whether CXX compiler has -isysroot
-- Checking whether CXX compiler has -isysroot - yes
-- Checking whether CXX compiler supports OSX deployment target flag
-- Checking whether CXX compiler supports OSX deployment target flag - yes
-- 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/flavors/part1_step1/build
 > ls
CMakeCache.txt  Makefile
CMakeFiles  cmake_install.cmake
 > make
Scanning dependencies of target toDo
[ 50%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
[100%] Building CXX object CMakeFiles/toDo.dir/ToDo.cc.o
Linking CXX executable toDo
[100%] Built target toDo
```

Note: If you are using Cygwin you may see a warning. Don't worry about it, we will take care of that shortly.

`mkdir build`

Create the directory in which to build our application. In this example it is a subdirectory of our source directory, but it could be anywhere. With our build happening outside of the source tree we can easily clean up by simply removing the build directory.

`cd build`

Change into the build directory to work from there.

`cmake -G "Unix Makefiles" ..`

Use CMake to setup a build using Unix Makefiles.

`-G <generator name>`

This allows us to tell CMake what kind of project file it should generate. In this example I wanted to use a Makefile. Which generators are available depends on your platform, use `cmake --help` to list them. Other generators will be covered in the next chapter.

<path to source>

The path to the source code. When doing out-of-source builds as is recommended the source code could be anywhere relative to the build directory. This path should be to the directory containing your top level `CMakeLists.txt`. In this example the source is in the parent directory so the path is '..'.

`ls`

CMake generates several files which should not be edited by hand. `Makefile` is the most important one to us as we use it to build our project. `CMakeCache.txt` is important to CMake as it stores a variety of information and settings for the project. Again you shouldn't touch this, however if unexpected problems arise this file probably is the cause; the best option then is to delete your build folder and have CMake regenerate.

`make`

Run `make` to build our target executable. Since we chose "Unix Makefiles" as our generator CMake created a Makefile for us.

CMake does all the hard work of making sure your environment has everything you need and sets up a project file, in this case a Makefile. You will notice that the Makefile created by CMake is quite fancy and has nice color output. If you are used to Make you will notice that this Makefile suppresses the standard output. While this provides a neater and cleaner experience it can make debugging more difficult as you can't check the flags passed to the compiler, etc. Before you start worrying you can get all of that output by running `make VERBOSE=1`.

```
 > cd build
 > make VERBOSE=1
/usr/local/Cellar/cmake/2.8.8/bin/cmake -H"/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step1" -B"/Volume
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_progress_start "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/p
make -f CMakeFiles/Makefile2 all
make -f CMakeFiles/toDo.dir/build.make CMakeFiles/toDo.dir/depend
cd "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step1/build" && /usr/local/Cellar/cmake/2.8.8/bin/cmake
Dependee "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step1/build/CMakeFiles/toDo.dir/DependInfo.cmake"
Dependee "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step1/build/CMakeFiles/CMakeDirectoryInformation.c
Scanning dependencies of target toDo
make -f CMakeFiles/toDo.dir/build.make CMakeFiles/toDo.dir/build
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_progress_report "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/
[ 50%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
/usr/bin/c++     -o CMakeFiles/toDo.dir/main.cc.o -c "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step1/
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_progress_report "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/
[100%] Building CXX object CMakeFiles/toDo.dir/ToDo.cc.o
/usr/bin/c++     -o CMakeFiles/toDo.dir/ToDo.cc.o -c "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step1/
Linking CXX executable toDo
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_link_script CMakeFiles/toDo.dir/link.txt --verbose=1
/usr/bin/c++    -Wl,-search_paths_first -Wl,-headerpad_max_install_names   CMakeFiles/toDo.dir/main.cc.o CMakeFiles/toDo.dir
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_progress_report "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/
[100%] Built target toDo
/usr/local/Cellar/cmake/2.8.8/bin/cmake -E cmake_progress_start "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/p
```

You can see that the makefile created by CMake is very precise and detailed. As such if anything moves you will have to run `cmake` again.

Simple Improvements

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
4  project("To Do List")
5
6  enable_testing()
7
8
9  add_executable(toDo main.cc
10                    ToDo.cc)
11
12 add_test(toDoTest toDo)
```

`cmake_minimum_required(VERSION` *version* `[FATAL_ERROR])`

This command specifies the minimum version of CMake that can be used with `CMakeLists.txt` file. The first argument must be `VERSION` verbatim. The next is the minimum version of CMake that can be used. The last is optional, but should be included, it must be `FATAL_ERROR` verbatim. It is recommended that this command be used in all top level `CMakeLists.txt`. If you aren't sure what version to set use the version of CMake you have installed.

cmake_minimum_required() documentation (2013-03-26)

`set(CMAKE_LEGACY_CYGWIN_WIN32 0)`

This gets rid of the warning you would have seen earlier if you were using Cygwin. If you aren't using Cygwin then it has no effect at all.

This tells CMake not to define `WIN32` when building with Cygwin. This is the preferred option and for us it doesn't make a difference either way so we will use the recommended setting.

`enable_testing()`

Enables testing for this CMake project. This should only be used in top level `CMakeLists.txt`. The main thing this does is enable the `add_test()` command.

enable_testing() documentation (2013-03-26)

`add_test(testname executable [arg1 ...])`

This command only does something if the `enable_testing()` has already been run, otherwise it does nothing. This adds a test to the current directory that will be run by CTest. The executable can be anything, so it could be a test program, e.g. a unit test created with something like Google Test, a script, or any other test imaginable. *Note:* Tests are not run automatically and if your test program is built as part of your project the test target will not ensure it is up to date. It is best to build all other targets before running the test target.

add_test() documentation (2013-03-26)

Perhaps I lied. One can easily argue that introducing the `add_test()` command is not a simple improvement. And they would probably be right, however, it is an important improvement. Testing will be explored further later in this tutorial.

Naturally we need some more code to go with this, so here goes:

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    argc,
    char** argv
)
{
    int result = 0;

    ToDo list;

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

    result |= EXPECT_EQUAL(list.size(),     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;
    }
}
```

ToDo.h **New or modified lines in bold.**

```
#ifndef TODO_H
#define TODO_H

#include <string>
#include <vector>


class ToDo
{
public:
    ToDo();
    ~ToDo();

    size_t size() const;

    void addTask(const std::string& task);
    std::string getTask(size_t index) const;

private:
    std::vector< std::string > this_tasks;
};

#endif // TODO_H
```

ToDo.cc **New or modified lines in bold.**

```
#include "ToDo.h"


ToDo::ToDo()
{
}

ToDo::~ToDo()
{
}


size_t ToDo::size() const
{
    return this_tasks.size();
}


void ToDo::addTask(
    const std::string& task
)
{
    this_tasks.push_back(task);
}

std::string ToDo::getTask(
    size_t index
) const
{
    if (index < this_tasks.size())
    {
        return this_tasks[index];
    }
    else
    {
        return "";
    }
}
```

Whew! That was not simple at all. Hopefully some of you are wondering why I didn't use a test framework. Later we will, but had we done so now we would have gotten further ahead of ourselves than we already have.

Building is exactly the same as before. In fact if you modified the files you had used before you simply need to run `make` again. The Makefile created by CMake will automatically run `cmake` again if you modify your `CMakeLists.txt`. So let's run our test:

```
 > mkdir build
 > cd build
 > cmake -G "Unix Makefiles" ..
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler identification is GNU 4.2.1
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Checking whether C compiler supports OSX deployment target flag
-- Checking whether C compiler supports OSX deployment target flag - yes
-- 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
-- Checking whether CXX compiler has -isysroot
-- Checking whether CXX compiler has -isysroot - yes
-- Checking whether CXX compiler supports OSX deployment target flag
-- Checking whether CXX compiler supports OSX deployment target flag - yes
-- 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/flavors/part1_step2/build
 > make
Scanning dependencies of target toDo
[ 50%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
[100%] Building CXX object CMakeFiles/toDo.dir/ToDo.cc.o
Linking CXX executable toDo
[100%] Built target toDo
 > make test
Running tests...
Test project /Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/build
    Start 1: toDoTest
1/1 Test #1: toDoTest .........................   Passed    0.01 sec
100% tests passed, 0 tests failed out of 1
Total Test time (real) =   0.03 sec
 > ls Testing
Temporary
 > ls Testing/Temporary
CTestCostData.txt LastTest.log
 > cat Testing/Temporary/LastTest.log
Start testing: Jul 16 22:00 EDT
----------------------------------------------------------
1/1 Testing: toDoTest
1/1 Test: toDoTest
Command: "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/build/toDo"
Directory: /Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/build
"toDoTest" start time: Jul 16 22:00 EDT
Output:
----------------------------------------------------------
Test passed
<end of output>
Test time =   0.01 sec
----------------------------------------------------------
Test Passed.
"toDoTest" end time: Jul 16 22:00 EDT
"toDoTest" time elapsed: 00:00:00
----------------------------------------------------------
End testing: Jul 16 22:00 EDT
 > cat Testing/Temporary/CTestCostData.txt
toDoTest 1 0.00976491
---
```

As mentioned earlier building with CMake is the same as it was before.

`make test`

The `enable_testing()` function we added to our `CMakeLists.txt` adds the "test" target to our Makefile. Making the "test" target will run CTest which will, in turn, run all of our tests. In our case just the one.

When CTest runs our tests it prints an abbreviated output that just provides the status of each of our tests. It then finishes up with a summary of all tests.

`Testing/Temporary/LastTest.log`

This file is created by CTest whenever it is run. It contains much more detail than the terminal output of CTest shows. Most importantly it contains the output of the tests. This is where you will want to look whenever a test fails.

`Testing/Temporary/CTestCostData.txt`

This file contains the time, in seconds, taken to run each test.

CMake along with CTest makes it easy to run our tests. CTest has many other features which will be presented later in this tutorial. There are, however, a few drawbacks to running our tests this way but we will leave those for later, too.

Chapter 2: IDE Integration

Introduction

Now that we are familiar with CMake I will make good on CMake's promise of flexibility. I said before that CMake could create projects for various IDE's and in this chapter we will do so. This is one of CMake's greatest strengths as it allows for very diverse development environments while working on the same project. It also makes it possible for you to take advantage of all available tools. If, for example, you prefer to work in Emacs or Vim and build with Make you could still create an IDE project and take advantage of its refactoring tools.

By now some of you have looked at the scroll bar and noticed that this chapter is rather long. Don't worry I don't expect you to read all of it and there are a lot of pictures. I present several IDEs but assume that you will only read the ones that are useful to you.

Please remember that CMake has more generators than those presented here. To list all of the available generators for your install use the command `cmake --help`. Most available generators are listed in the CMake documentation (2012-07-08).

We will use the same code as we had at the end of the first chapter. It can be downloaded again here:

Visual Studio

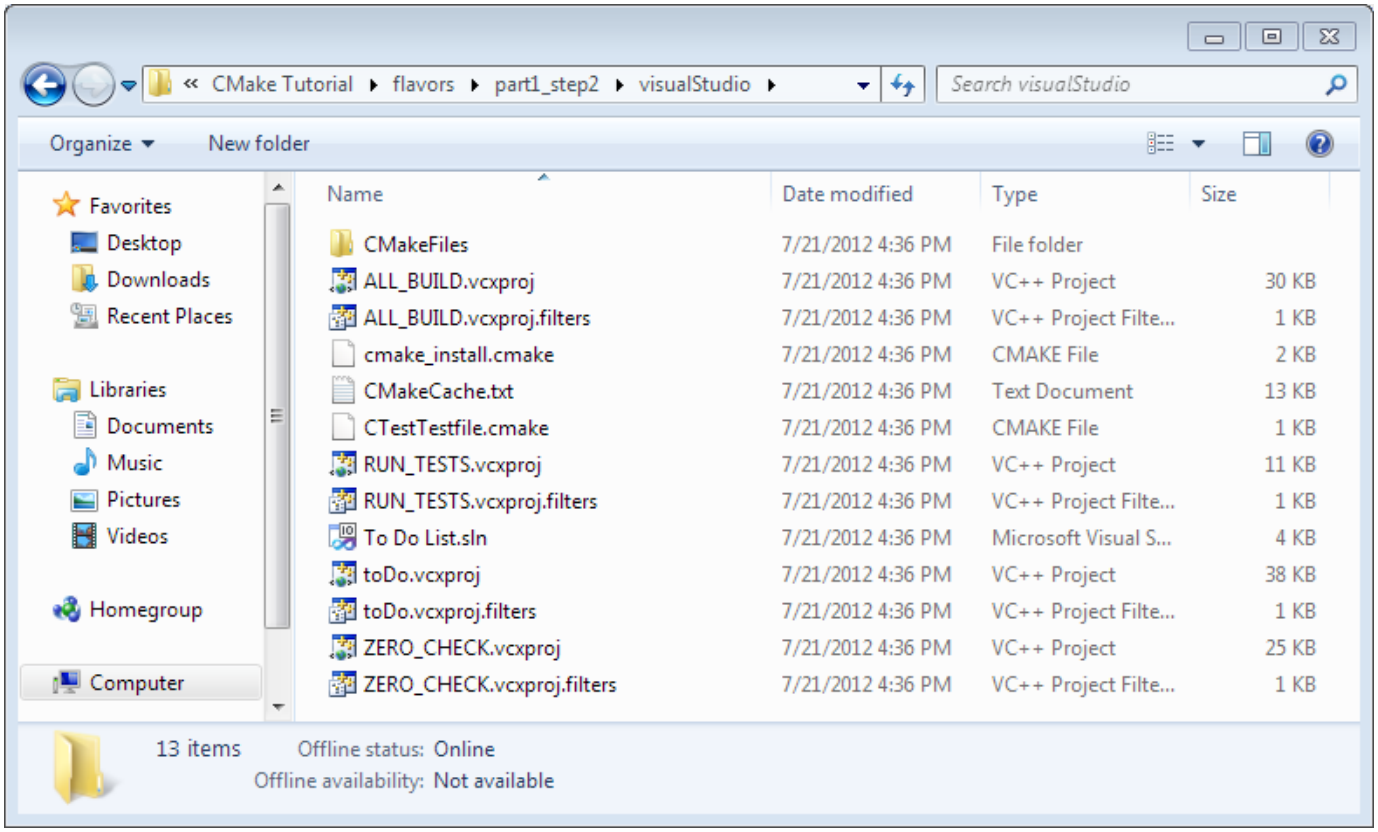
Visual Studio 2010 Express Version 10.0.30319.1 RTMTel was used.
Visual Studio 2010 Professional Version 10.0.30319.1 RTMRel was used for `MSBuild`

Generating a Visual Studio solution is simple, we just have to use a Visual Studio generator when we invoke CMake.

```
> mkdir visualStudio
> cd visualStudio
> cmake -G "Visual Studio 10" ..
-- Check for working C compiler using: Visual Studio 10
-- Check for working C compiler using: Visual Studio 10 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler using: Visual Studio 10
-- Check for working CXX compiler using: Visual Studio 10 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: M:/Programming/C++/CMake Tutorial/flavors/part1_step2/visualStudio
```

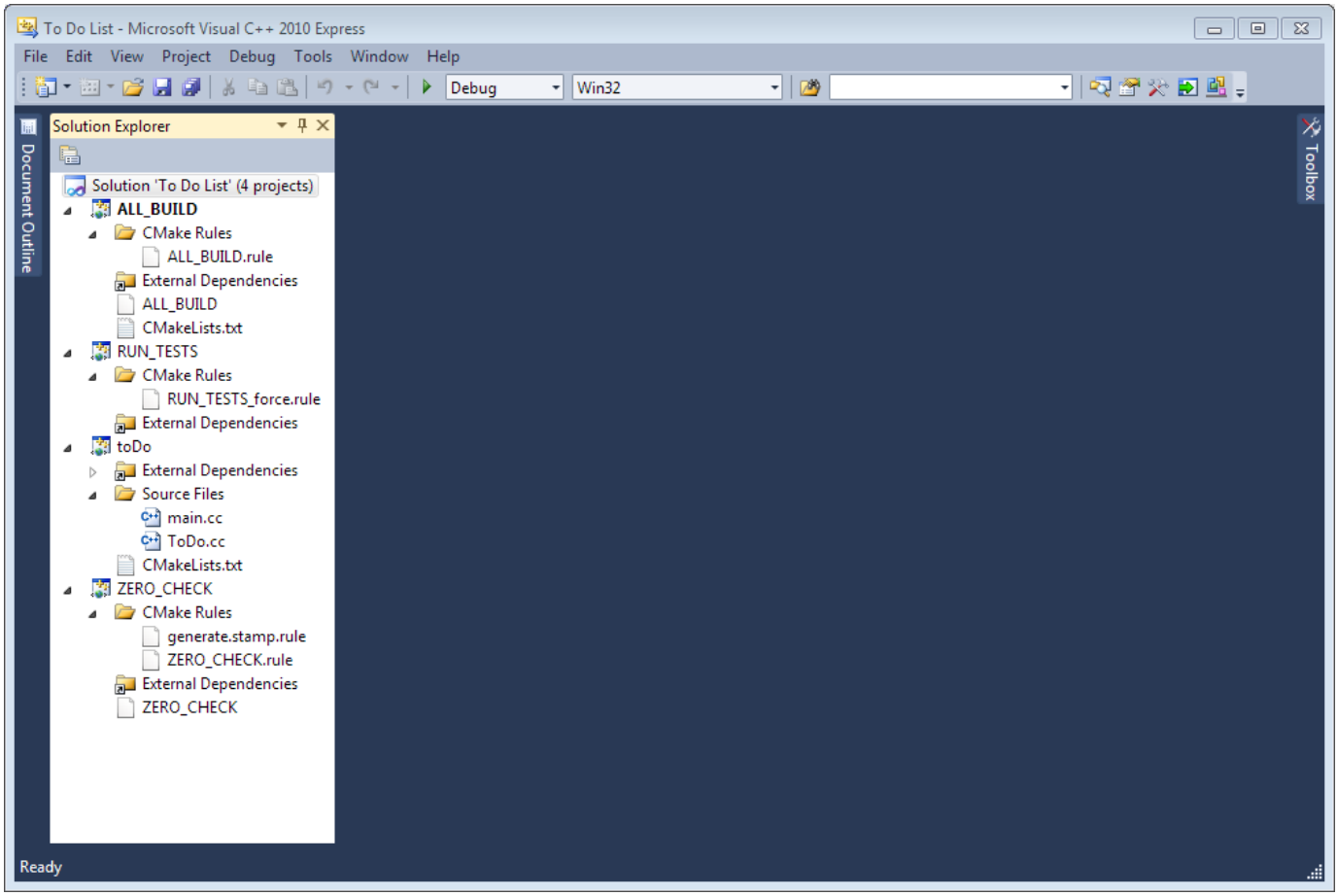
It is important to note that there are different generators for different versions of Visual Studio, so you will have to make sure that you chose the generator most appropriate for your Visual Studio install. CMake's output is actually a lot shorter than when we first ran it. You will notice that CMake uses Visual Studio to compile rather than interacting directly with the compiler.

Of course we still did an out-of-source build so the Visual Studio project files will not clutter your source tree. This is what CMake created:



As you can see CMake created several Visual Studio files. The one we really care about is `To Do List.sln`, as you can see this is named after our CMake project. If file names containing spaces cause problems for you, or are inconvenient, then you will want to make sure your project names do not contain spaces. Let's see what kind of solution CMake created.

Note: When you open the solution Visual Studio may display a Security Warning because it doesn't trust the projects. This seems to be caused by CMake creating them not Visual Studio. You can just click "OK".



The generated solution is a bit more complicated than what you would have created by hand. There are 3 more projects than you would have expected since we are only building one executable and nothing else. Each project does, however, have a purpose:

ALL_BUILD

This project builds all of the targets that are defined in the `CMakeLists.txt`. Since we only have one in ours it is a bit redundant.

RUN_TESTS

Building this project runs CTest in much the same way that `make test` did. It creates the same output files, too. CTest's output is also displayed in the Output Window. Just as before this does not depend on any of your targets, so if your tests depend on any targets be sure to build them first.

toDo

This is the little command line tool we are building. It corresponds to the `add_executable` command we have in our `CMakeLists.txt`.

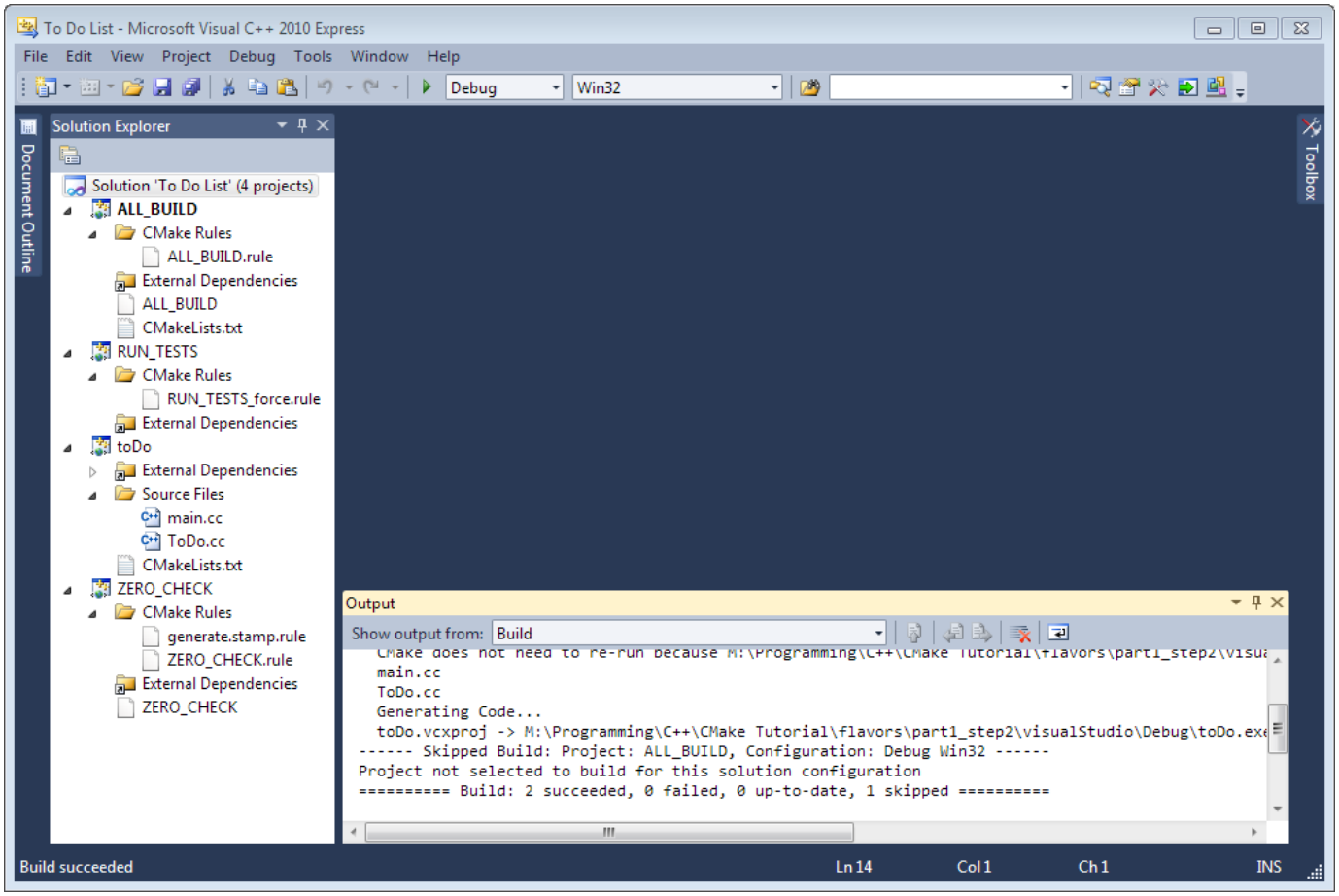
ZERO_CHECK

This is a rather oddly named project. It's purpose is to make sure that the Visual Studio solution and its projects are all up to date. If you modify the `CMakeLists.txt` this project will update your Visual Studio solution. All other projects depend on this one so you don't have to build it manually. Unfortunately when the solution and projects are updated by this Visual Studio will, for each one updated, ask you if you want to reload it, which can get a bit annoying.

If you look at the "toDo" project you will notice that it only contains the `.cc` files. This is because those are the only files listed in the `CMakeLists.txt` for the toDo target. If you were to add `ToDo.h` to the toDo target it would appear in

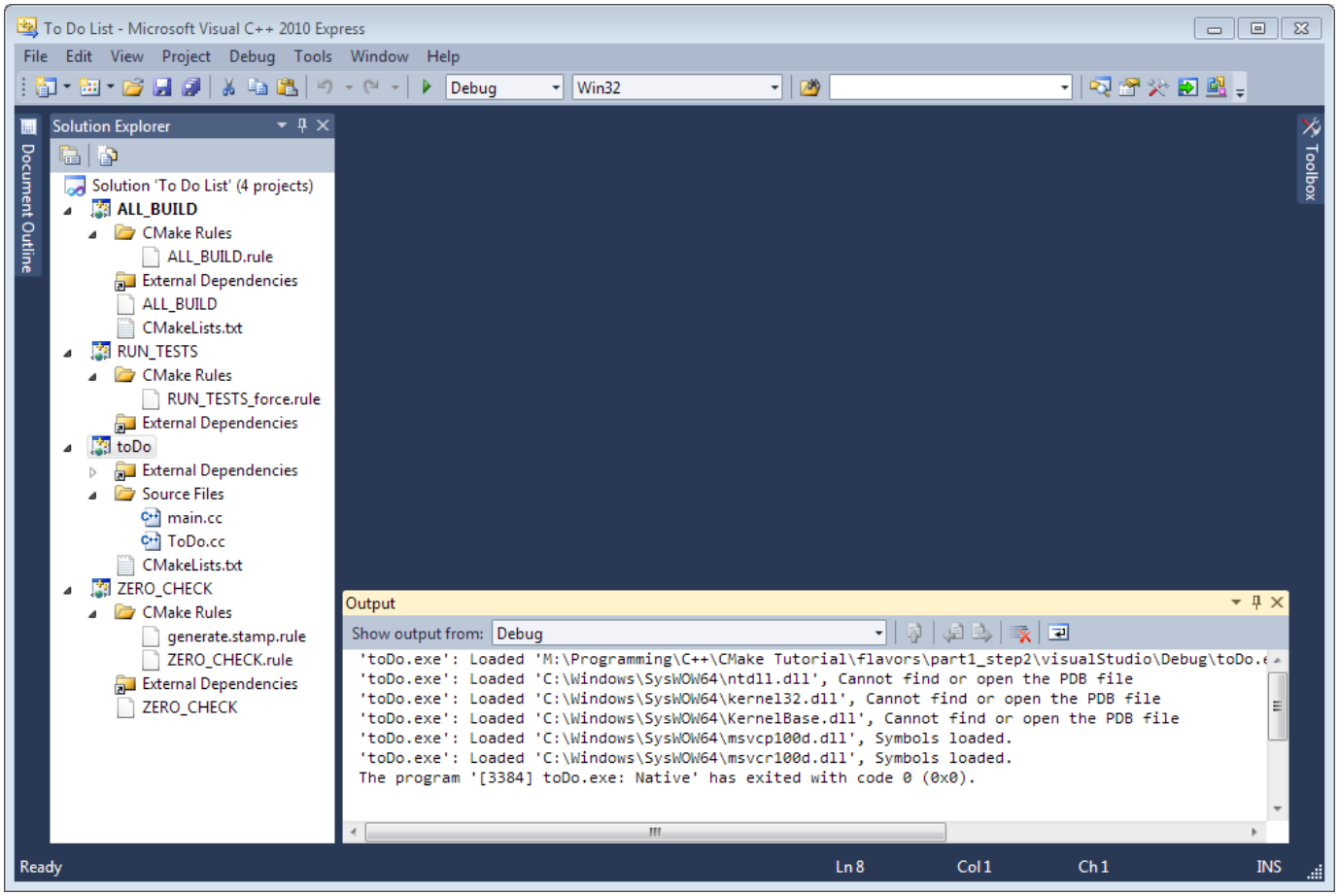
the "toDo" project in Visual Studio.

Let's try and build and see what happens.



I used the "Start Debugging" button on the toolbar which tried to debug the "ALL_BUILD" project. So while it successfully built toDo.exe it was not run since "ALL_BUILD" does not produce any outputs, much less an executable. So if you want to actually debug "toDo" you will have to explicitly pick that project. If we explicitly debug the "toDo" project we get what we were expecting.

Note: If you set the project you want to debug as the "StartUp" project Visual Studio will debug it when you click the "Start Debugging" button. You can recognize the "StartUp" project as its name will be bold. To do this right click on the project and pick "Set as StartUp Project".



Unfortunately our program is run in a command window that closes as soon as our program completes, so we don't get to see its output. However the Output Window in Visual Studio shows that toDo exited with a code of 0 which means our test still passes. So everything works fine in Visual Studio.

If you need to be able to build from the command line either because you prefer to or for an automated build process you can use the `MSBuild` command.

Note: `MSBuild` does not appear to be included with Visual Studio Express, but only Visual Studio Professional.

```
 > cd visualStudio
 > MSBuild ALL_BUILD.vcxproj
Microsoft (R) Build Engine Version 4.0.30319.1
[Microsoft .NET Framework, Version 4.0.30319.269]
Copyright (C) Microsoft Corporation 2007. All rights reserved.
Build started 7/22/2012 1:18:41 AM.
Project "M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\ALL_BUILD.vcxproj" on node 1 (default targets).
Project "M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\ALL_BUILD.vcxproj" (1) is building "M:\Programmi
PrepareForBuild:
  Creating directory "Win32\Debug\ZERO_CHECK\".
  Creating directory "Debug\".
InitializeBuildStatus:
  Creating "Win32\Debug\ZERO_CHECK\ZERO_CHECK.unsuccessfulbuild" because "AlwaysCreate" was specified.
CustomBuild:
  Checking Build System
  CMake does not need to re-run because M:/Programming/C++/CMake Tutorial/flavors/part1_step2/visualStudio/CMakeFiles/genera
FinalizeBuildStatus:
  Deleting file "Win32\Debug\ZERO_CHECK\ZERO_CHECK.unsuccessfulbuild".
  Touching "Win32\Debug\ZERO_CHECK\ZERO_CHECK.lastbuildstate".
Done Building Project "M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\ZERO_CHECK.vcxproj" (default targe
Project "M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\ALL_BUILD.vcxproj" (1) is building "M:\Programmi
PrepareForBuild:
```

```
  Creating directory "toDo.dir\Debug\".
InitializeBuildStatus:
  Creating "toDo.dir\Debug\toDo.unsuccessfulbuild" because "AlwaysCreate" was specified.
CustomBuild:
  Building Custom Rule M:/Programming/C++/CMake Tutorial/flavors/part1_step2/CMakeLists.txt
  CMake does not need to re-run because M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\CMakeFiles\genera
ClCompile:
  C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\CL.exe /c /Zi /nologo /W3 /WX- /Od /Ob0 /Oy- /D WIN32 /D _WINDO
cl : Command line warning D9035: option 'GX' has been deprecated and will be removed in a future release [M:\Programming\C++
cl : Command line warning D9036: use 'EHsc' instead of 'GX' [M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStu
cl : Command line warning D9035: option 'GZ' has been deprecated and will be removed in a future release [M:\Programming\C++
cl : Command line warning D9036: use 'RTC1' instead of 'GZ' [M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStu
  main.cc
  ToDo.cc
  Generating Code...
ManifestResourceCompile:
  C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\rc.exe /nologo /fo"toDo.dir\Debug\toDo.exe.embed.manifest.res" toD
Link:
  C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\link.exe /ERRORREPORT:QUEUE /OUT:"M:\Programming\C++\CMake Tutc
  toDo.dir\Debug\main.obj
  toDo.dir\Debug\ToDo.obj  /machine:X86 /debug
Manifest:
  C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\mt.exe /nologo /verbose /out:"toDo.dir\Debug\toDo.exe.embed.manife
  C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\rc.exe /nologo /fo"toDo.dir\Debug\toDo.exe.embed.manifest.res" toD
LinkEmbedManifest:
  C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\link.exe /ERRORREPORT:QUEUE /OUT:"M:\Programming\C++\CMake Tutc
  toDo.dir\Debug\main.obj
  toDo.dir\Debug\ToDo.obj  /machine:X86 /debug
  toDo.vcxproj -> M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\Debug\toDo.exe
FinalizeBuildStatus:
  Deleting file "toDo.dir\Debug\toDo.unsuccessfulbuild".
  Touching "toDo.dir\Debug\toDo.lastbuildstate".
Done Building Project "M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\toDo.vcxproj" (default targets).
PrepareForBuild:
  Creating directory "Win32\Debug\ALL_BUILD\".
InitializeBuildStatus:
  Creating "Win32\Debug\ALL_BUILD\ALL_BUILD.unsuccessfulbuild" because "AlwaysCreate" was specified.
CustomBuild:
  Building Custom Rule M:/Programming/C++/CMake Tutorial/flavors/part1_step2/CMakeLists.txt
  CMake does not need to re-run because M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\CMakeFiles\genera
  Build all projects
FinalizeBuildStatus:
  Deleting file "Win32\Debug\ALL_BUILD\ALL_BUILD.unsuccessfulbuild".
  Touching "Win32\Debug\ALL_BUILD\ALL_BUILD.lastbuildstate".
Done Building Project "M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\ALL_BUILD.vcxproj" (default target
Build succeeded.
"M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\ALL_BUILD.vcxproj" (default target) (1) ->
"M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\toDo.vcxproj" (default target) (3) ->
(ClCompile target) ->
  cl : Command line warning D9035: option 'GX' has been deprecated and will be removed in a future release [M:\Programming\C
  cl : Command line warning D9036: use 'EHsc' instead of 'GX' [M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualS
  cl : Command line warning D9035: option 'GZ' has been deprecated and will be removed in a future release [M:\Programming\C
  cl : Command line warning D9036: use 'RTC1' instead of 'GZ' [M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualS
    4 Warning(s)
    0 Error(s)
Time Elapsed 00:00:05.49
```

`MSBuild ALL_BUILD.vcxproj`

The `MSBuild` tool requires the project to build as a command line argument. In this case I built everything. As you can see its output is rather verbose. (Also it seems the projects created by CMake could use some updating.) reference, command line reference (2012-07-22)

Xcode

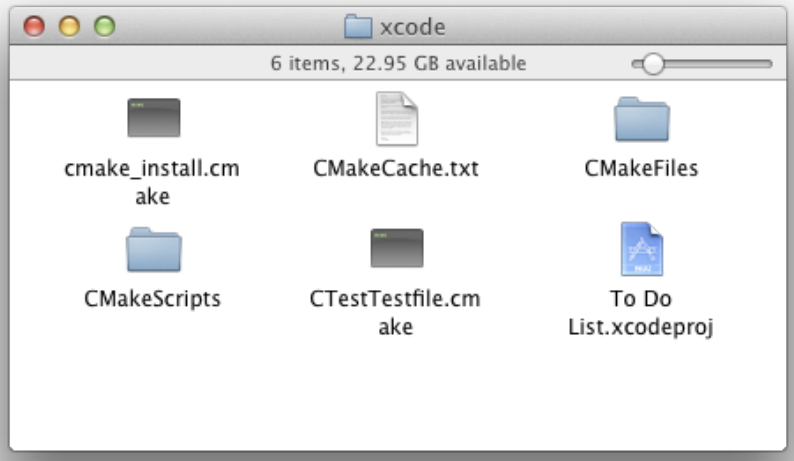
Mac OS X

Xcode Version 4.1 Build 4B110 was used.

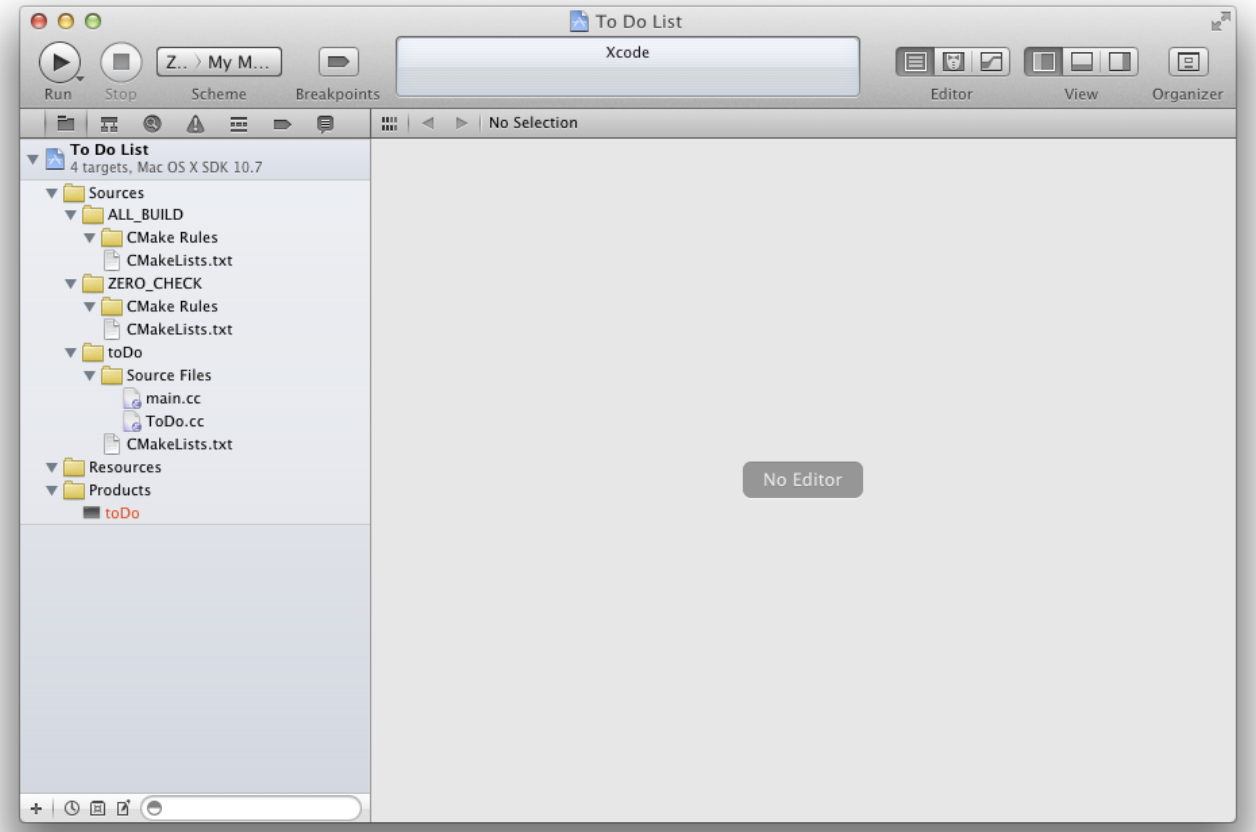
Generating an Xcode project is very similar to generating any other project:

```
> mkdir xcode
> cd xcode
> cmake -G "Xcode" ..
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler identification is GNU 4.2.1
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Checking whether C compiler supports OSX deployment target flag
-- Checking whether C compiler supports OSX deployment target flag - yes
-- Check for working C compiler using: Xcode
-- Check for working C compiler using: Xcode -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Checking whether CXX compiler has -isysroot
-- Checking whether CXX compiler has -isysroot - yes
-- Checking whether CXX compiler supports OSX deployment target flag
-- Checking whether CXX compiler supports OSX deployment target flag - yes
-- Check for working CXX compiler using: Xcode
-- Check for working CXX compiler using: Xcode -- 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/flavors/part1_step2/xcode
```

If you look closely you will notice that most of CMake's output looks the same as it did when we first ran it. In fact the only major difference is that CMake doesn't directly interact with the compiler, it uses Xcode instead. We are still doing an out-of-source build so even the Xcode project won't clutter your source tree. CMake created the following files:

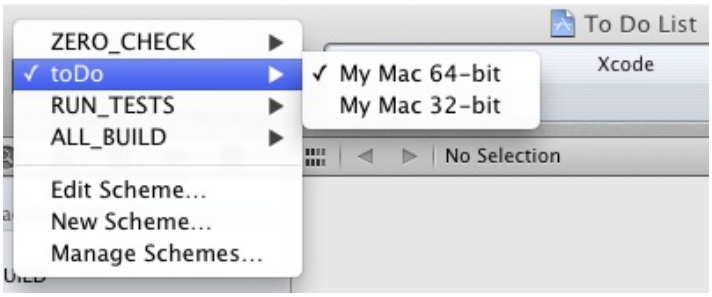


Most of these files will look familiar if you had looked at what files CMake generated before. The most important file is `To Do List.xcodeproj`. Note that the project file is named after the project command in `CMakeLists.txt`. If spaces in file names cause trouble in your environment then you will want to ensure your project names have no spaces. Now let's take a look at the project CMake created for us.



The project is not as neat as one you would have made by hand. Most conspicuously `ToDo.h` is missing. This is because CMake doesn't actually know about it. However because it is in the same directory as `ToDo.cc` the compiler will still find it. If you were to include `ToDo.h` in the `add_executable()` command then it would be included in the Xcode project. Both Xcode and CMake know not to compile header files so there would be no actual effect on the build.

You will notice the extra folders "ALL_BUILD" and "ZERO_CHECK", these actually correspond to particular Xcode targets created by CMake. These are the targets created by CMake:



ZERO_CHECK

This oddly named target checks your `CMakeLists.txt` and updates your project as needed. Just as with the generated Makefile.

toDo

This is our executable as specified by the `add_executable()` command. This will build our little command line

tool.

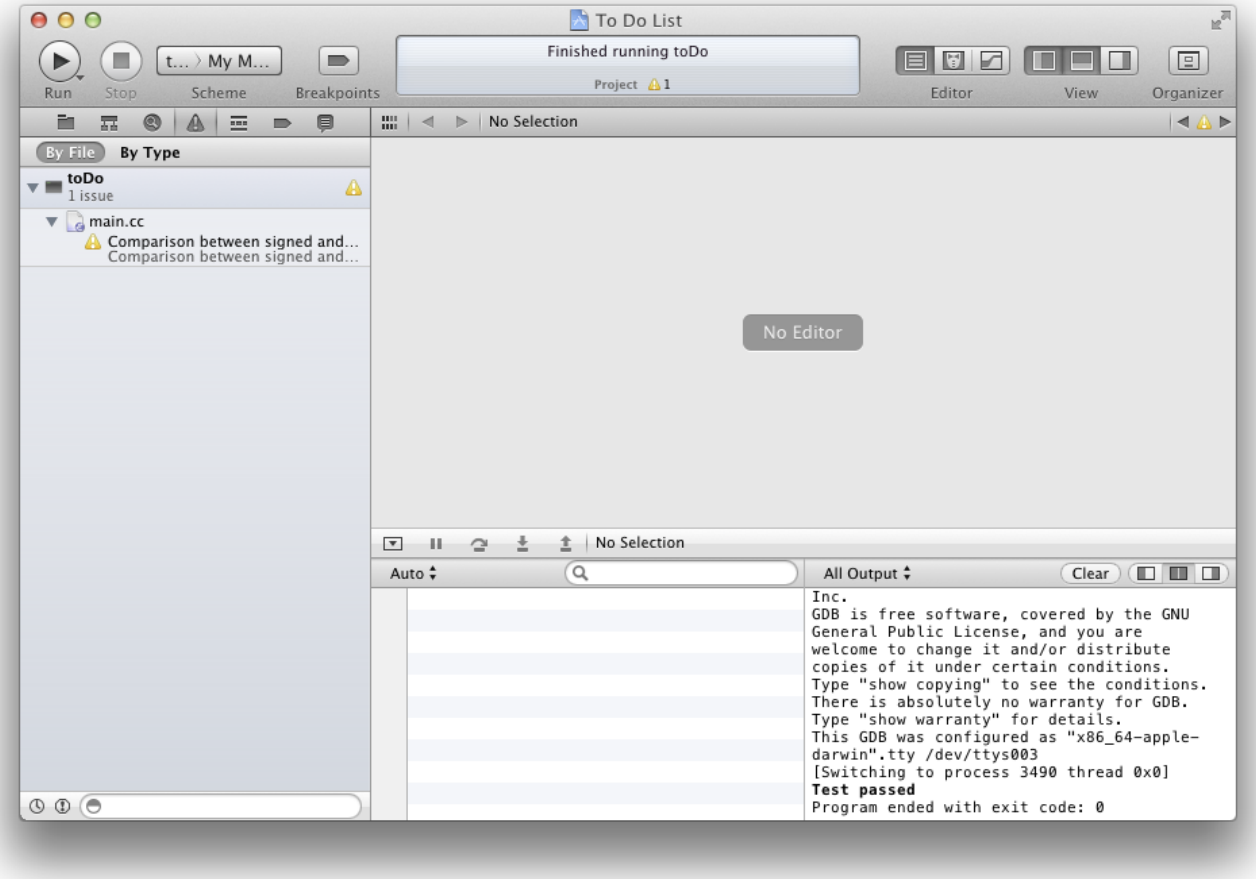
RUN_TESTS

This runs CTest just as `make test` did before. It produces the same output files as before, too. CTest's output, however, is not displayed, but it can be found using the Log Navigator. Also as before it does not depend on any other targets, e.g. "toDo," even if a test does.

ALL_BUILD

This builds all targets except "RUN_TESTS" just as `make` did before. Since we only specified one target, "toDo," this target is redundant, but if we had specified other targets, say another executable, this would build them all.

Let's build toDo and see what output Xcode produces.



The "Run" button in Xcode builds and then runs the target. The build succeeded and the test still passes; so far everything works fine in Xcode. You will notice, though, that we now have a warning. If you were to look in Xcode you will find that `-Wmost`, `-Wno-four-char-constants`, and `-Wno-unknown-pragmas` are passed to gcc by Xcode. Our `CMakeLists.txt` doesn't pass any additional options to the compiler so when we were using the Makefile generator we were using gcc's default settings. For now don't worry about the warning, we will get to that in chapter 3.

Now if you prefer to work from the command line but must use Xcode you can use the `xcodebuild` tool provided by Apple.

```
 > cd xcode
 > xcodebuild -list
Information about project "To Do List":
    Targets:
        ALL_BUILD
        RUN_TESTS
        ZERO_CHECK
        toDo
    Build Configurations:
        Debug
        Release
        MinSizeRel
        RelWithDebInfo
    If no build configuration is specified "Debug" is used.
 > xcodebuild
=== BUILD AGGREGATE TARGET ZERO_CHECK OF PROJECT To Do List WITH THE DEFAULT CONFIGURATION (Debug) ===
Check dependencies
PhaseScriptExecution "CMake Rules" "xcode/To Do List.build/Debug/ZERO_CHECK.build/Script-1D0B6873874D4ED8AF14DE31.sh"
    cd "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2"
    /bin/sh -c "\"/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/xcode/To Do List.build/Debug/ZERO_CH
echo ""
make -f /Volumes/Documents/Programming/C++/CMake\ Tutorial/flavors/part1_step2/xcode/CMakeScripts/ReRunCMake.make
make[1]: `CMakeFiles/cmake.check_cache' is up to date.
=== BUILD NATIVE TARGET toDo OF PROJECT To Do List WITH THE DEFAULT CONFIGURATION (Debug) ===
Check dependencies
CompileC "xcode/To Do List.build/Debug/toDo.build/Objects-normal/x86_64/ToDo.o" ToDo.cc normal x86_64 c++ com.apple.compiler
    cd "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2"
    setenv LANG en_US.US-ASCII
    /Developer/usr/bin/llvm-gcc-4.2 -x c++ -arch x86_64 -fmessage-length=0 -pipe -Wno-trigraphs -fpascal-strings -O0 "-DCMAK
CompileC "xcode/To Do List.build/Debug/toDo.build/Objects-normal/x86_64/main.o" main.cc normal x86_64 c++ com.apple.compiler
    cd "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2"
    setenv LANG en_US.US-ASCII
    /Developer/usr/bin/llvm-gcc-4.2 -x c++ -arch x86_64 -fmessage-length=0 -pipe -Wno-trigraphs -fpascal-strings -O0 "-DCMAK
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/main.cc: In function 'int equalityTest(T1, T2, const c
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/main.cc:34:   instantiated from here
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/main.cc:58: warning: comparison between signed and uns
Ld xcode/Debug/toDo normal x86_64
    cd "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2"
    setenv MACOSX_DEPLOYMENT_TARGET 10.7
    /Developer/usr/bin/llvm-g++-4.2 -arch x86_64 -isysroot /Developer/SDKs/MacOSX10.7.sdk "-L/Volumes/Documents/Programming/
PhaseScriptExecution "CMake PostBuild Rules" "xcode/To Do List.build/Debug/toDo.build/Script-01429AA71A364B6AAE9CB89B.sh"
    cd "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2"
    /bin/sh -c "\"/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/xcode/To Do List.build/Debug/toDo.bu
echo "Depend check for xcode"
Depend check for xcode
cd /Volumes/Documents/Programming/C++/CMake\ Tutorial/flavors/part1_step2/xcode && make -C /Volumes/Documents/Programming/C+
make[1]: Nothing to be done for `PostBuild.toDo.Debug'.
=== BUILD AGGREGATE TARGET ALL_BUILD OF PROJECT To Do List WITH THE DEFAULT CONFIGURATION (Debug) ===
Check dependencies
PhaseScriptExecution "CMake Rules" "xcode/To Do List.build/Debug/ALL_BUILD.build/Script-48A6EF12B1004D59A240CCC6.sh"
    cd "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2"
    /bin/sh -c "\"/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/xcode/To Do List.build/Debug/ALL_BUI
echo ""
echo Build\ all\ projects
Build all projects
** BUILD SUCCEEDED **
```

`xcodebuild -list`

This lists all the targets and all the build configurations set up in the Xcode project. Xcode, by default, uses the `xcodeproj` file in the current directory if there is only one, which is the case when using CMake. (man page 2012-07-17)

`xcodebuild`

`xcodebuild` assumes the first target if none is provided on the command line, much like `make`. Conveniently CMake made `ALL_BUILD` the first target. As you can see this builds everything and is a lot more verbose than the makefile created by CMake.

iOS

While cross-compiling will not be covered until later you can build for iOS using CMake and the Xcode generator. There is a Google Code Project specifically for this: ios-cmake (2012-07-09).

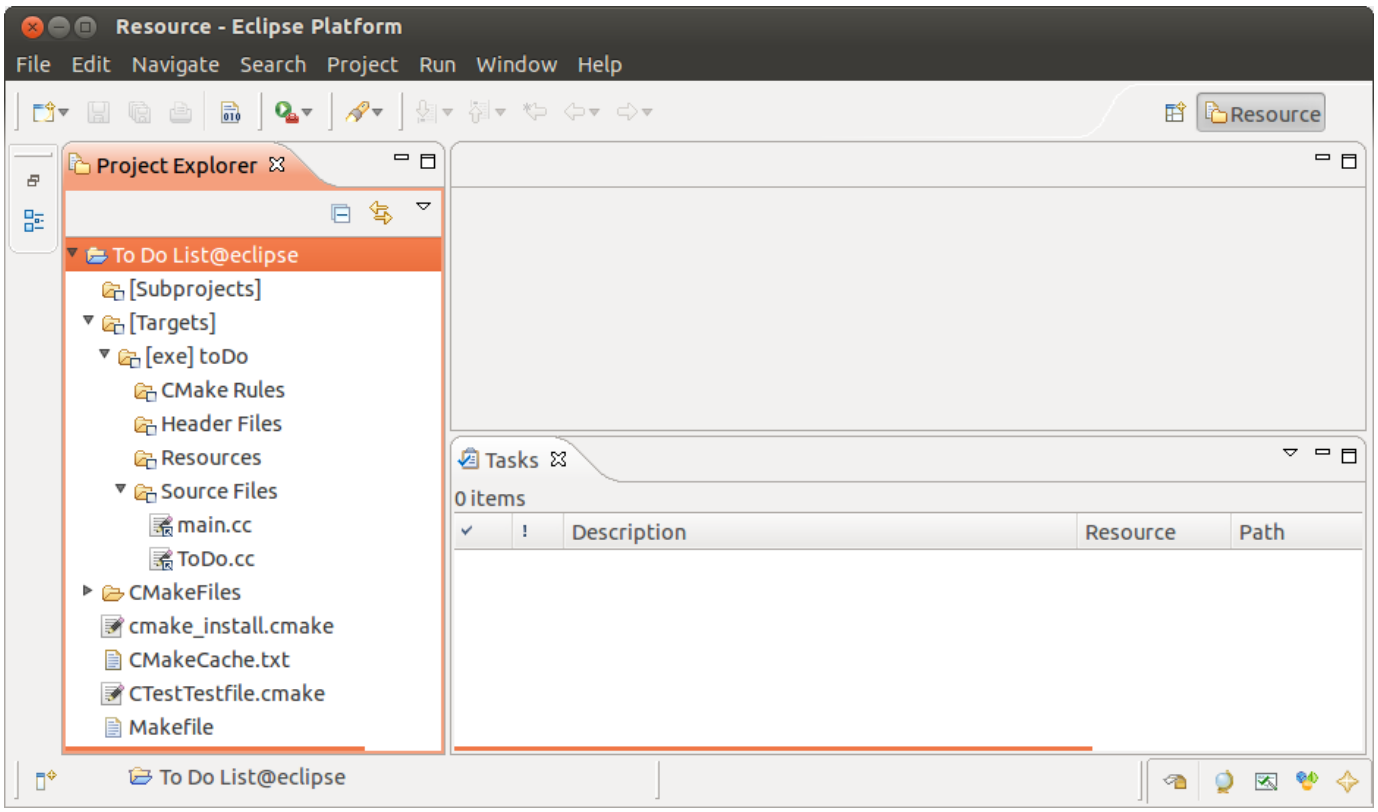
Eclipse CDT4

Eclipse Indigo Version 3.7.2 Build I20110613-1736 was used.

If you want to use Eclipse you simply need to tell CMake so when you generate your project files.

```
 > mkdir eclipse
 > cd eclipse
 > cmake -G "Eclipse CDT4 - Unix Makefiles" ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Could not determine Eclipse version, assuming at least 3.6 (Helios). Adjust CMAKE_ECLIPSE_VERSION if this is wrong.
-- 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
CMake Warning in CMakeLists.txt:
  The build directory is a subdirectory of the source directory.
  This is not supported well by Eclipse.  It is strongly recommended to use a
  build directory which is a sibling of the source directory.
-- Generating done
-- Build files have been written to: /home/john/Desktop/part1_step2/eclipse
 > ls -A
CMakeCache.txt  cmake_install.cmake  CTestTestfile.cmake  .project
CMakeFiles      .cproject            Makefile
```

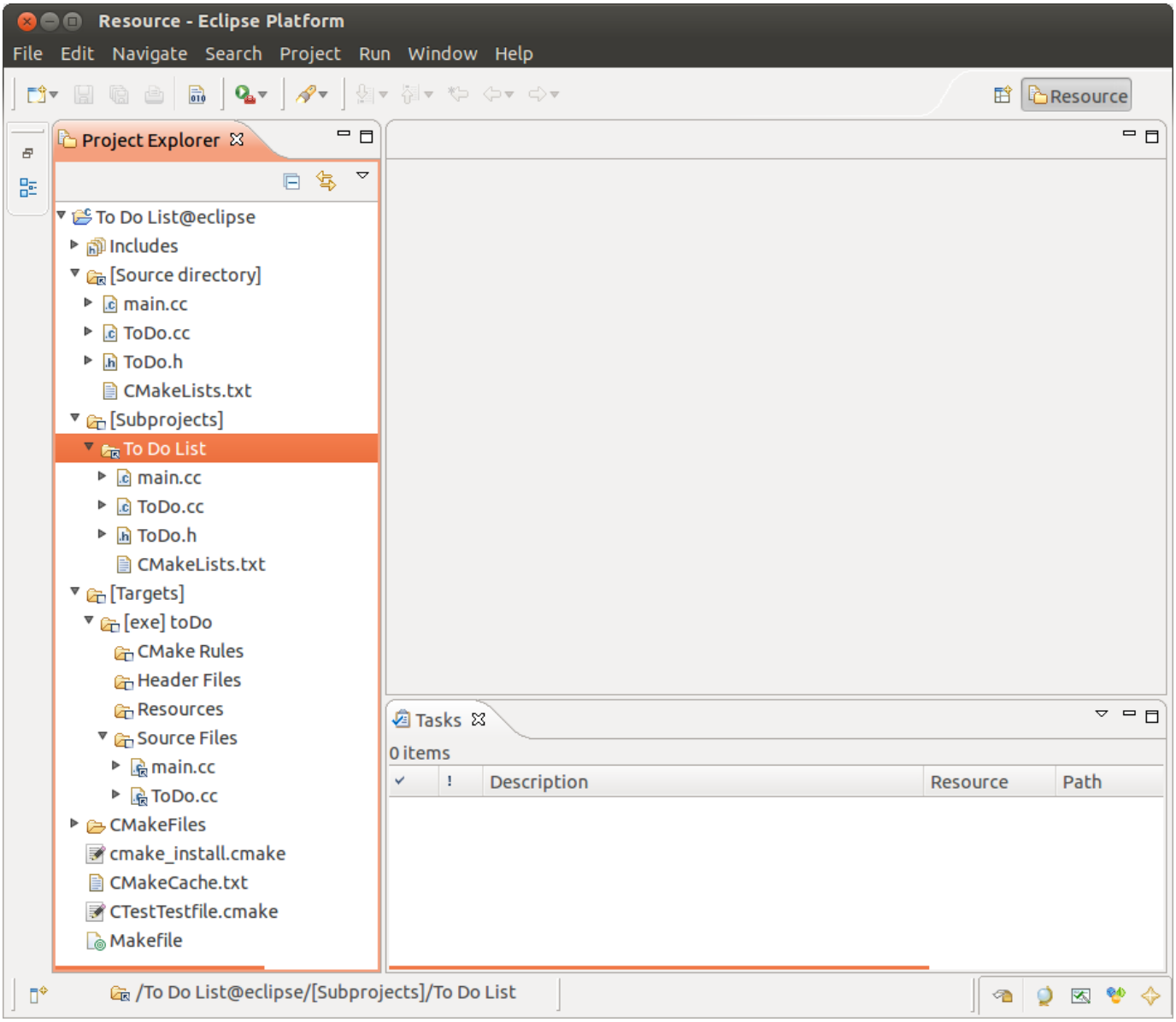
Well perhaps it isn't actually that simple. CMake warns us that Eclipse doesn't like the build directory being a subdirectory of the source directory. As you can see it created the `.project` and `.cproject` files required by Eclipse CDT.



The project looks okay, however it isn't. Certain aspects of the project will not function properly. So we will learn from our mistake and follow CMake's advice.

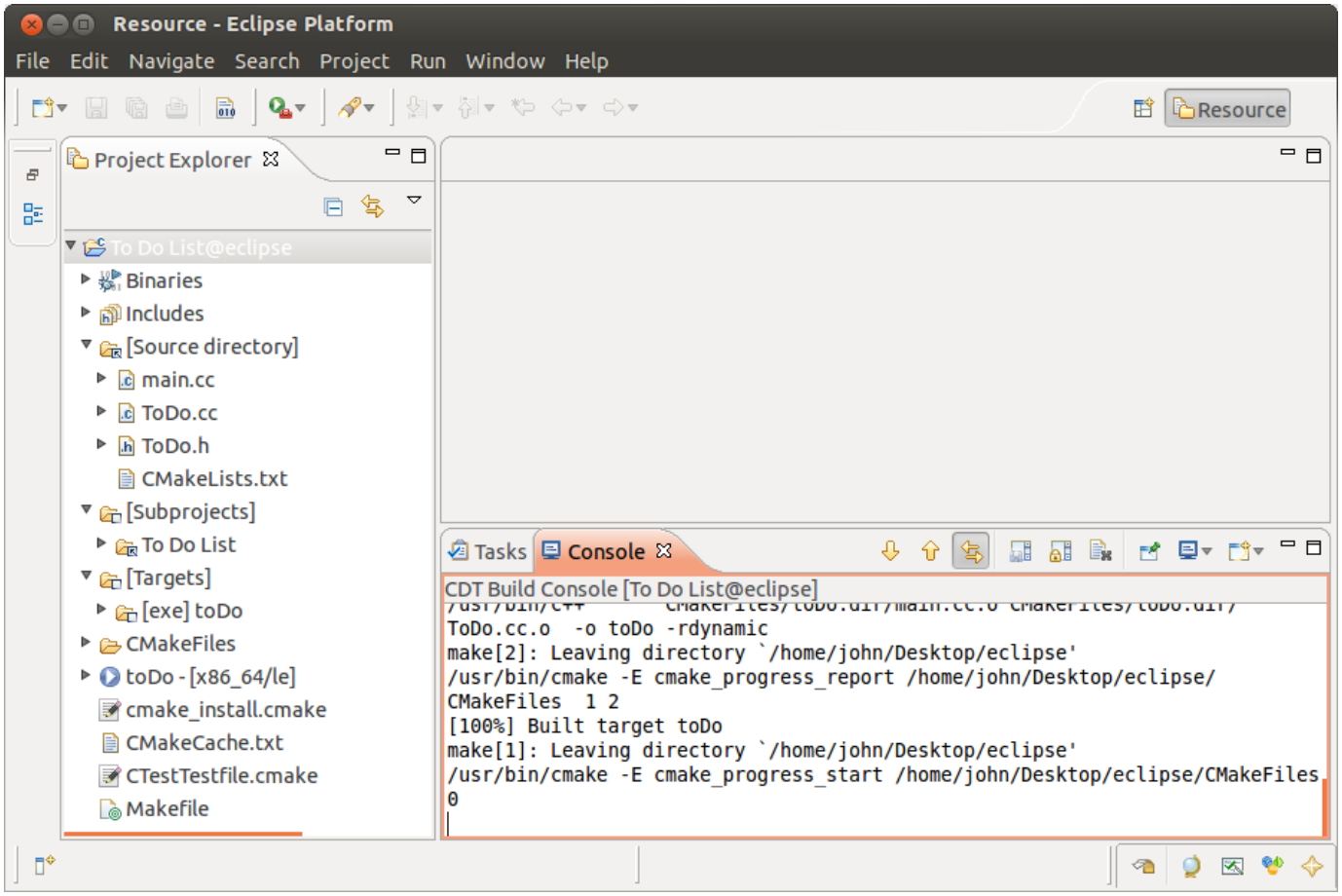
```
> cd ..
> mkdir eclipse
> cd eclipse
> cmake -G "Eclipse CDT4 - Unix Makefiles" ../part1_step2/
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Could not determine Eclipse version, assuming at least 3.6 (Helios). Adjust CMAKE_ECLIPSE_VERSION if this is wrong.
-- 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
-- Generating done
-- Build files have been written to: /home/john/Desktop/eclipse
> ls -a
.   CMakeCache.txt  cmake_install.cmake  CTestTestfile.cmake  .project
..  CMakeFiles      .cproject            Makefile
```

CMake's output looks the same, save for the lack of a warning, and it also created the same files as before. The project should work fine this time. Let's have a look.



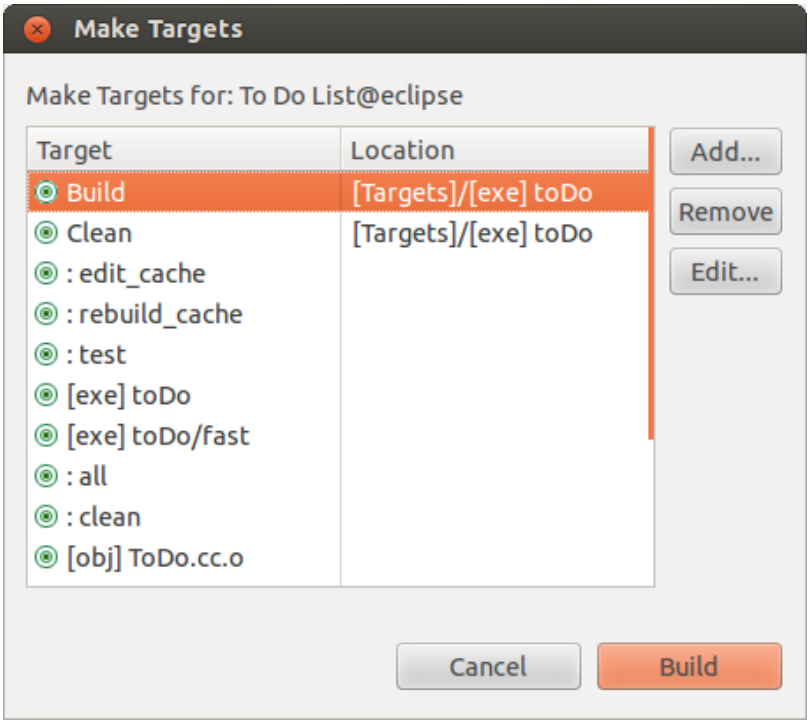
The project looks a lot better this time. If you are familiar with Eclipse you may know that it only supports one target per project whereas CMake supports many. In fact managing builds of complex source trees is one of CMake's strengths. These seem to be at odds with each other. If you looked closely before you would have noticed that CMake created a Makefile and created a Makefile project for Eclipse. This allows CMake to support multiple targets *and* work with Eclipse. The "[Subprojects]" folder lists every CMake project included, in our case there's just one. Similarly the "[Targets]" folder lists all of the targets defined in your `CmakeLists.txt`. If you looked at any of the other IDE projects generated by CMake you may be surprised to see `ToDo.h` included. That is because the Eclipse project includes some virtual folders which display whatever files happened to be in the corresponding directory.

Let's try building our project and see if it still works.

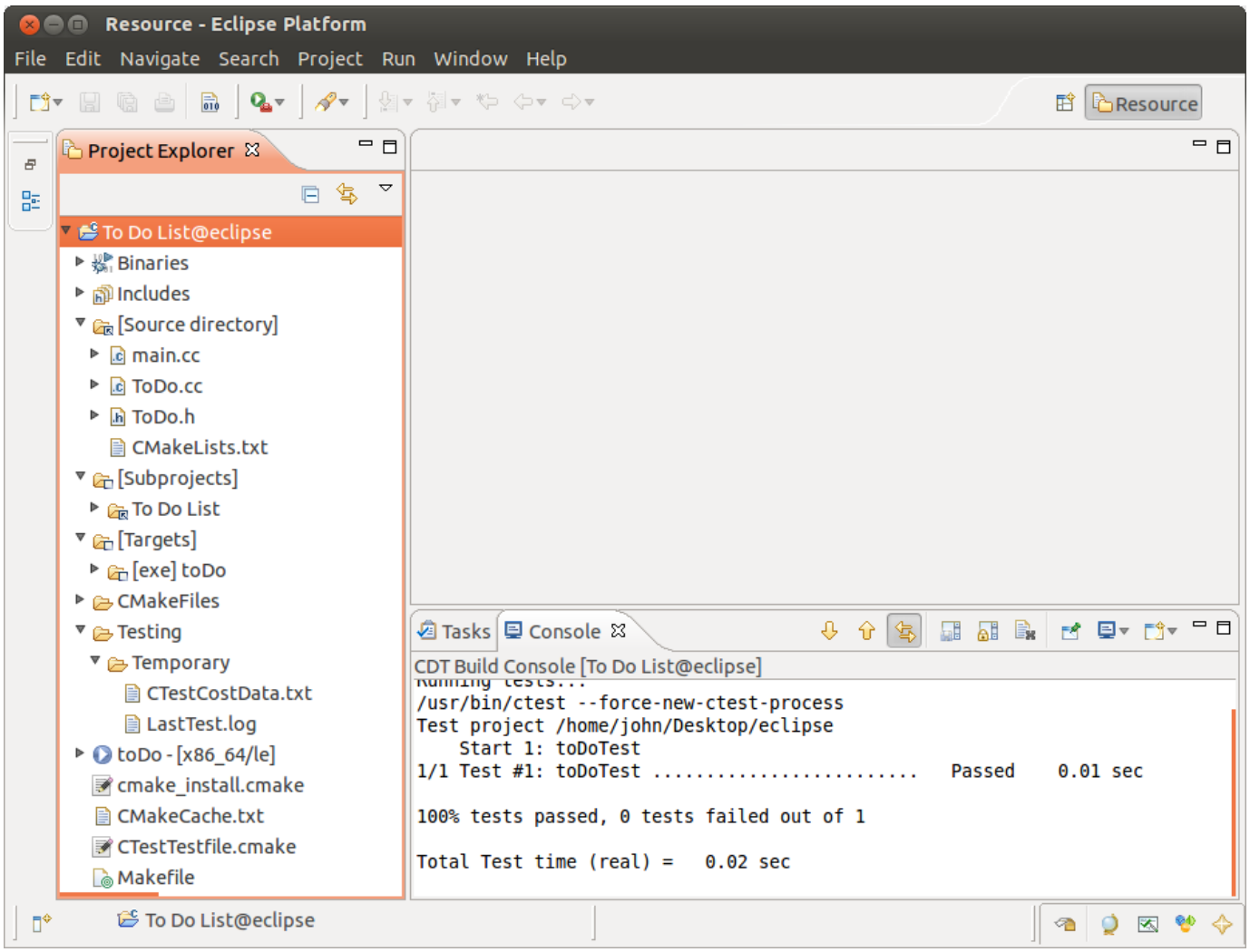


It still builds fine and as you can see Eclipse uses make to do the building. Conveniently the binary executable "toDo" is added to the project so it can easily be run or debugged from within Eclipse.

Eclipse supports Makefiles rather well so you can get it to build any of the available targets. Eclipse provides a convenient list.



The default is, of course, to build all targets. "[exe] toDo" is, of course our tiny example program. However there is also "[exe] toDo/fast" which has an intriguing name. The difference between the two is that the "fast" version doesn't check if the `CmakeLists.txt` has changed or recalculate toDo's dependencies. It also doesn't calculate completion percentages. If you are sure that none of those have changed using a "fast" target can speed up your build a bit. However, the most interesting target here is ": test" which will run CTest just as `make test` did before. CTest's output is displayed in the Build Console and the Testing directory is added to the project.



As you can see the test still passes so everything works in Eclipse.

If you desire to still build your project from the command line it is actually quite easy because CMake created Makefiles. So you can build just as you did before.

```
> cd ../eclipse
> make
[ 50%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
[100%] Building CXX object CMakeFiles/toDo.dir/ToDo.cc.o
Linking CXX executable toDo
[100%] Built target toDo
```

KDevelop

For KDevelop 3 CMake will generate a project for you to use. KDevelop 4, however, has native CMake support making that step unnecessary.

Generated (KDevelop3)

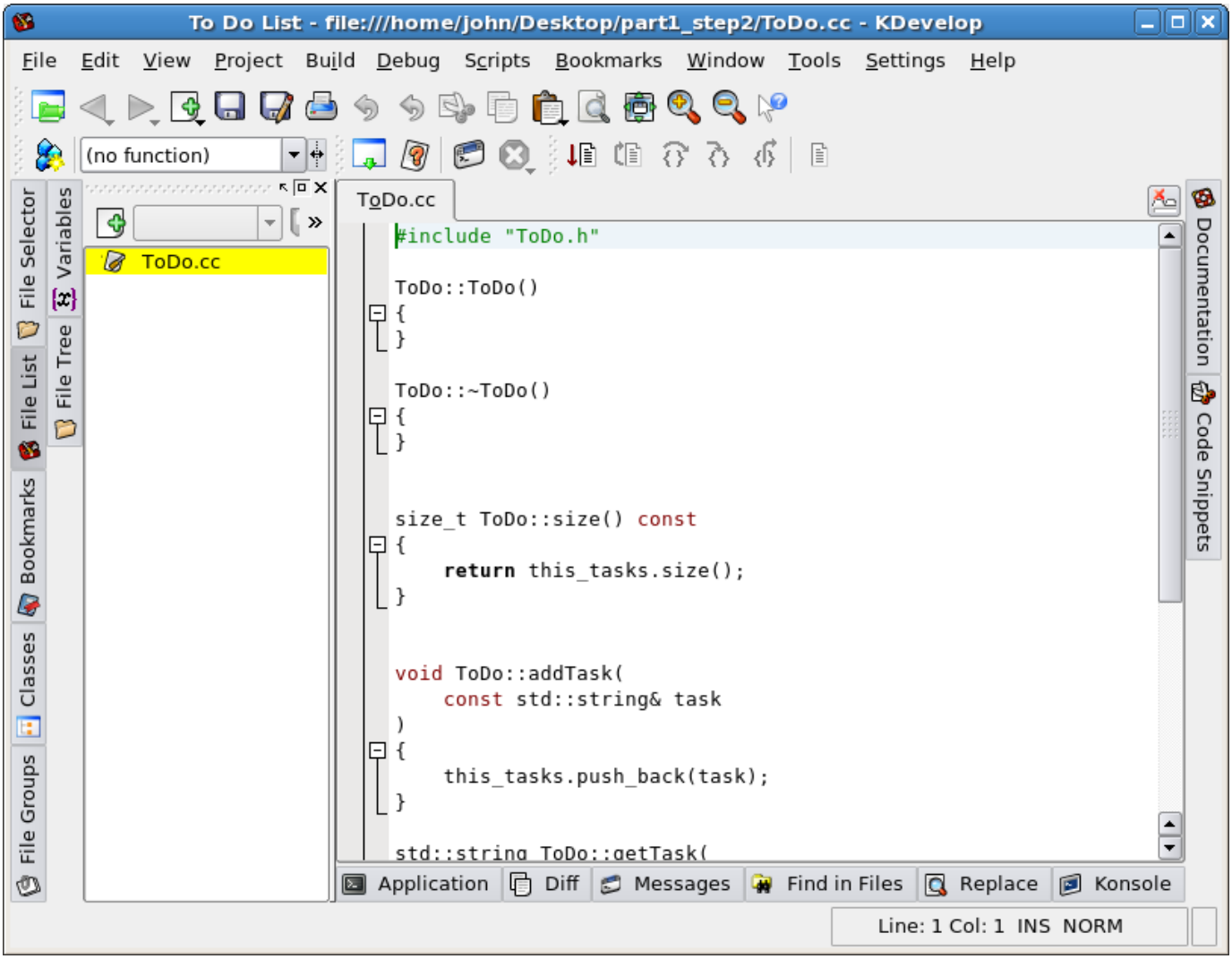
KDevelop Version 3.3.4 was used.

If you want CMake to create a KDevelop project for you specify the "KDevelop3" generator. There is also a "KDevelop3 - Unix Makefiles" which generates the same exact files, so save yourself the typing.

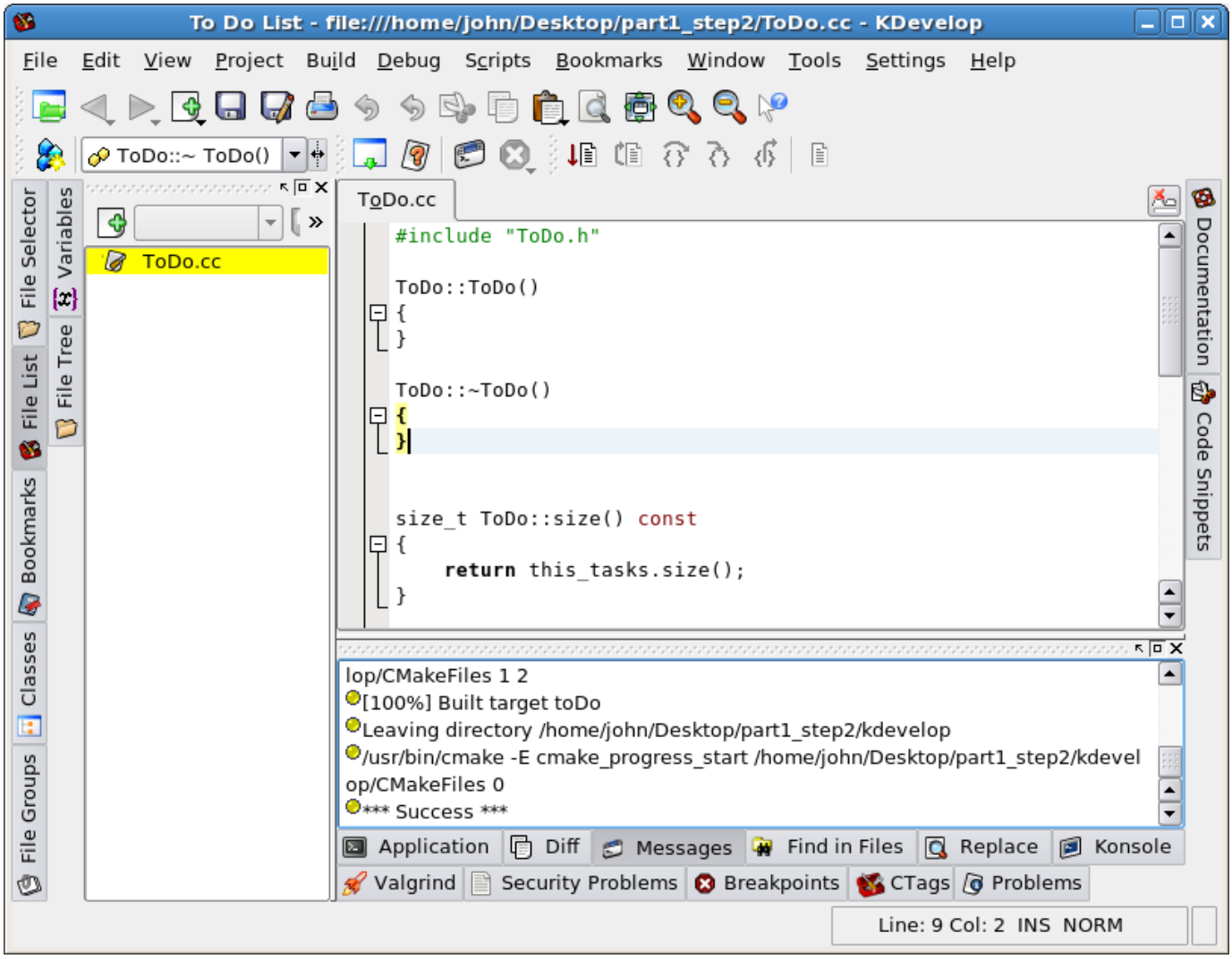
```
> mkdir kdevelop
> cd kdevelop
> cmake -G "KDevelop3" ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler 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
-- Generating done
-- Build files have been written to: /home/john/Desktop/part1_step2/kdevelop
```

The output looks similar to the first time we ran it. It produces a few extra files for KDevelop, though.

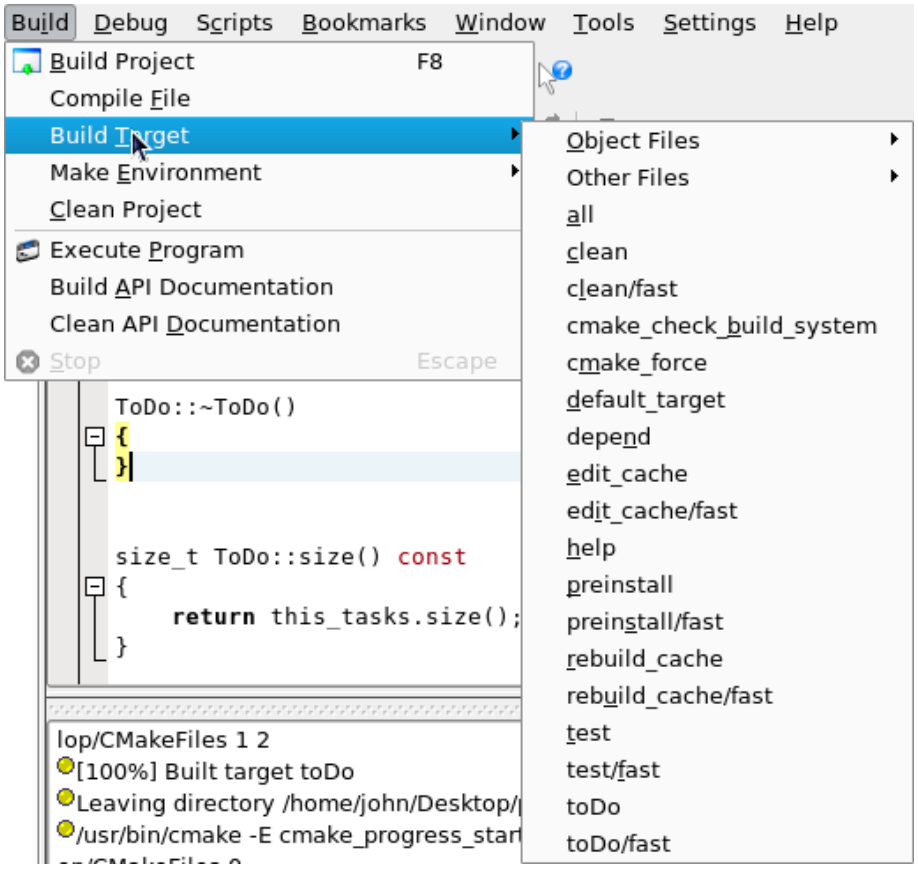
The KDevelop 3 project file, `To Do List.kdevelop` is the most important of the new files. You will notice that CMake still created a `Makefile`. KDevelop's Makefile support, however, is quite good. Let's see the project.



Oddly the "File List" only displays `ToDo.cc` even though we would expect it to also include `main.cc`. The "File Selector" shows all of the files in your source directory. Let's see if we can still build.

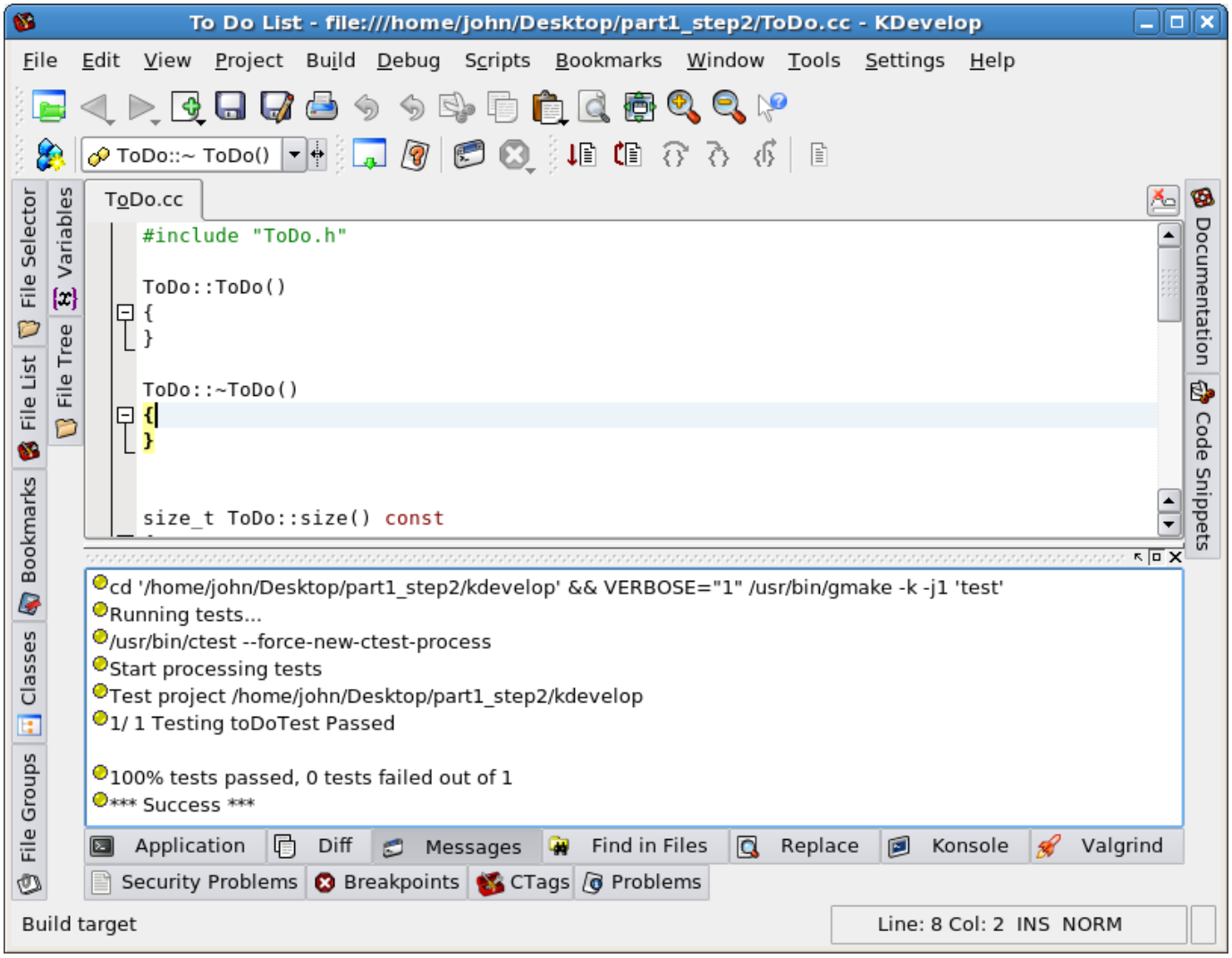


Of course it still builds. We are using the same `Makefile` as we originally did. The only difference this time is that KDevelop is running make for us. Thanks to KDevelop's Makefile support we can actually build any target we want.



By default KDevelop builds the target "all" which does exactly what you'd expect, it builds everything. There are a few targets that end with "/fast". These "fast" targets skip some steps to save time, so be careful when using them. Dependency calculation and checking the `CMakeLists.txt` file for changes are skipped; also completion percentages aren't printed. While these will build faster than the regular targets if there are any changes that require dependencies to be recalculated or any `CMakeLists.txt` have been changed you results will not be what you expected.

Currently the most interesting target is "test". Building this target is, of course, the same as running `make test`.



Our test still passes. Don't lie, I know you had doubts. CTest's output is displayed in the Messages panel. Just as before CTest creates the same files, too.

If you wanted to build from the command line it's quite simple since we have a `Makefile` just as before.

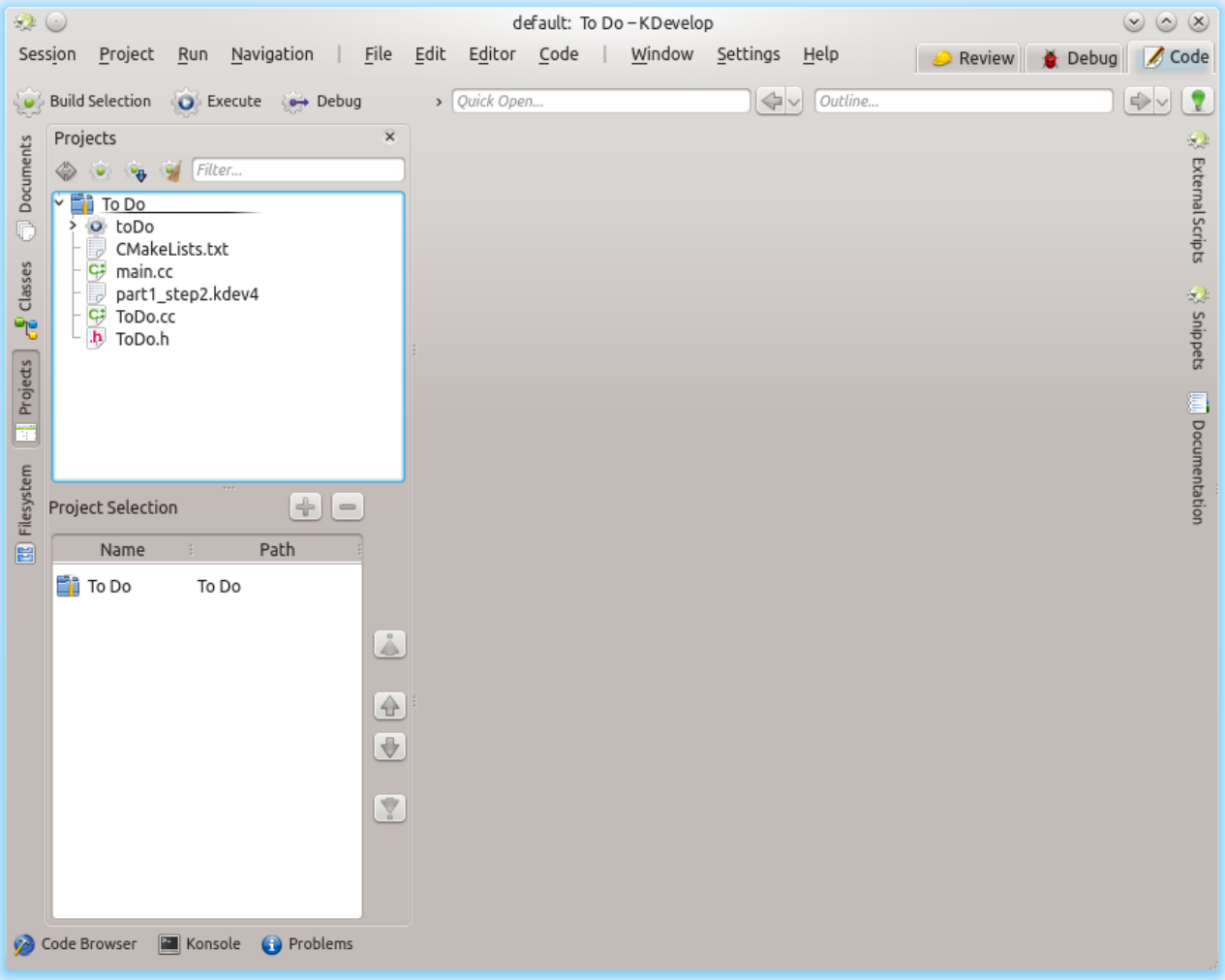
```
> cd kdevelop
> make
[ 50%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
[100%] Building CXX object CMakeFiles/toDo.dir/ToDo.cc.o
Linking CXX executable toDo
[100%] Built target toDo
```

CMake Support (KDevelop4)

KDevelop Version 4.3.1 was used.

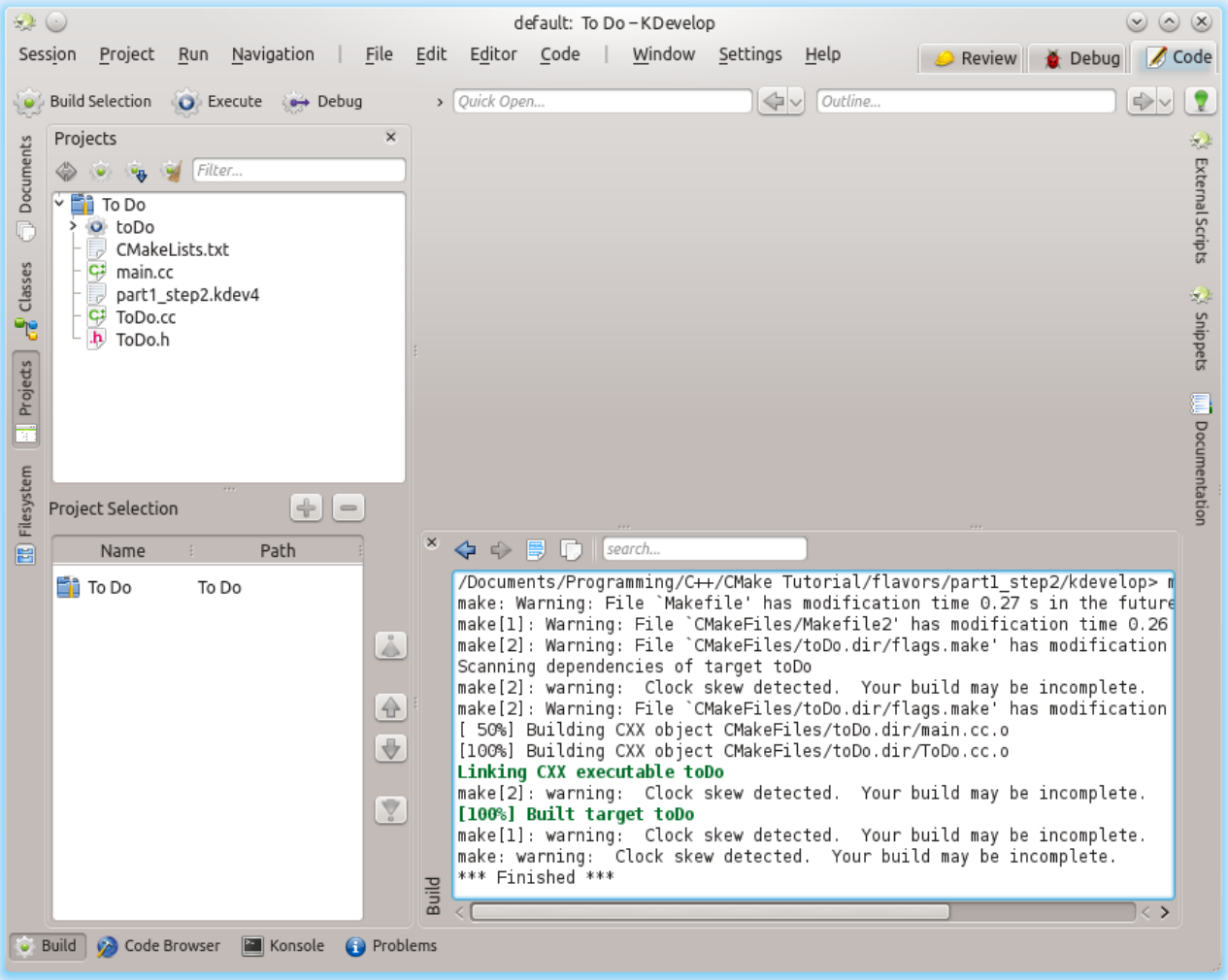
KDevelop 4 has built-in support for CMake projects. So rather than use a generator to make a new project file as was done in the previous examples we instead simply open the CMake project with KDevelop.

After launching KDevelop 4 choose "Open / Import Project..." from the "Project Menu" and follow the steps of the import process. First you will have to find you `CmakeLists.txt` file. KDevelop will treat it as your project file. Next it will ask for a project name and build system. It will infer both and likely be correct. Lastly it will configure your build directory and CMake binary. Again the defaults are probably sufficient. After that you will get to see your project.

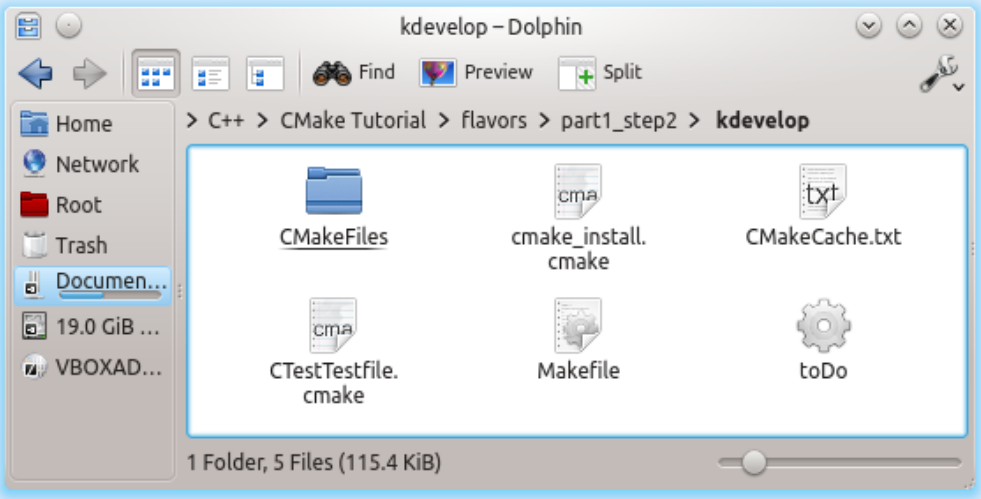


The file list shows all files that are actually in the project directory. Conveniently this include `ToDo.h`. However you may also notice a `kdev4` project file. While KDevelop4 supports CMake, including out of source builds, it does put a project file in your source directory. Although since it is only one file it is easy to clean up (or have git ignore).

Building is, of course, as simple as clicking the "Build Selection" button.



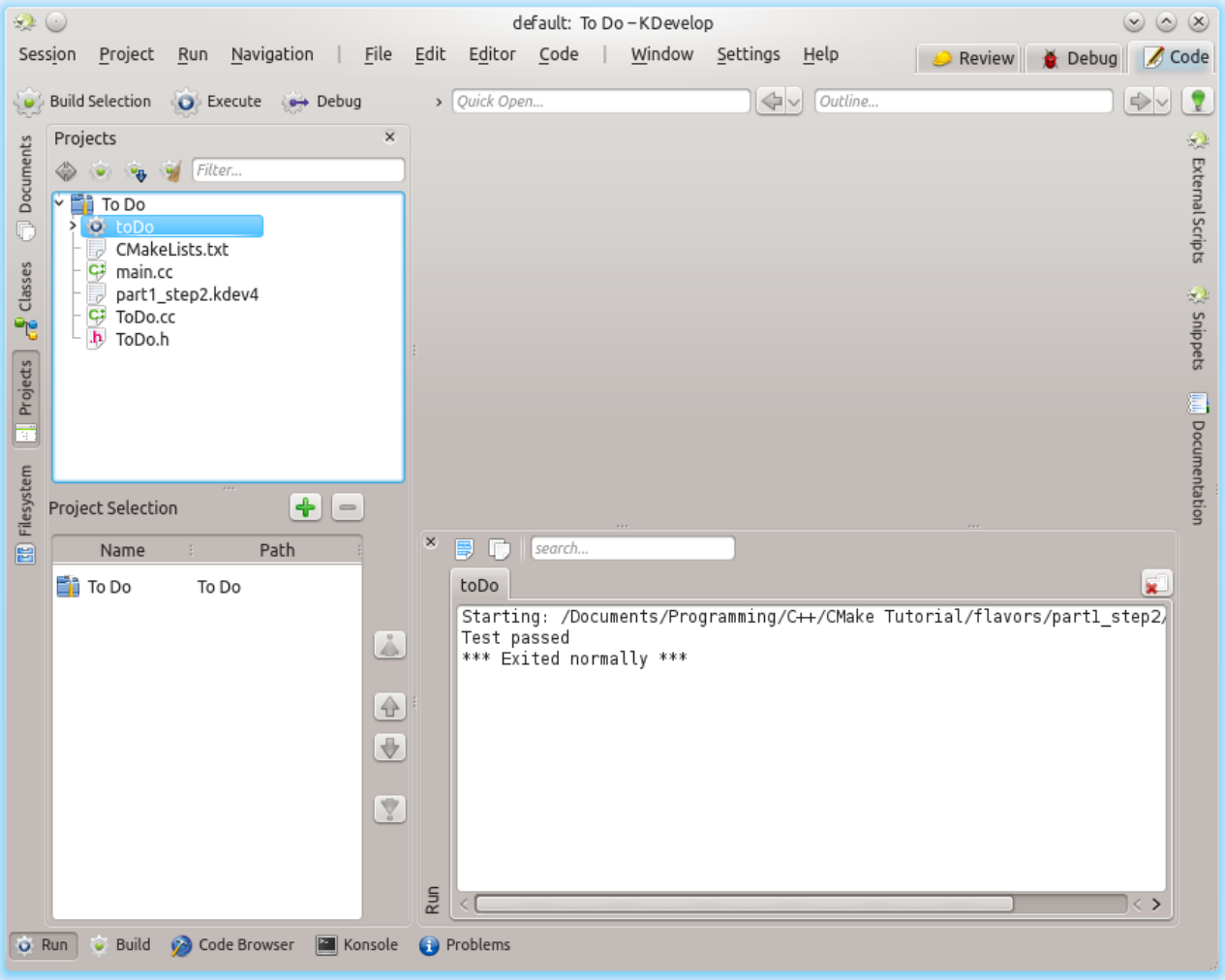
You will notice that KDevelop still uses make to build the project. The main difference here is that KDevelop also runs CMake for you. These are the files it created:



Exactly the files you should have expected.

Now if I wanted to run our little program the "Execute" button doesn't seem to work, it merely displays an error. However if I right-click on the "toDo" entry under the project and pick "Execute As..." > "Native Application" it

runs fine.



Unfortunately I cannot find a way to run the tests from within KDevelop. As it does create a Makefile project the tests can be manually run from the command line. That seems like an ugly work-around, though.

```
 > cd kdevelop
 > make test
Running tests...
Test project /Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/kdevelop
    Start 1: toDoTest
1/1 Test #1: toDoTest .........................   Passed    0.01 sec
100% tests passed, 0 tests failed out of 1
Total Test time (real) =   0.05 sec
```

Since this is a Makefile project you can easily build from the command line using `make`.

```
 > cd kdevelop
 > make
[ 50%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
[100%] Building CXX object CMakeFiles/toDo.dir/ToDo.cc.o
Linking CXX executable toDo
[100%] Built target toDo
```

Chapter 3: GUI Tool

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 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.**

```
1  cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
2  set(CMAKE_LEGACY_CYGWIN_WIN32 0)
3
4  project("To Do List")
5
6  enable_testing()
7
8
9  if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU" OR
10     "${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang")
11     set(warnings "-Wall -Wextra -Werror")
12 elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
13     set(warnings "/W4 /WX /EHsc")
14 endif()
15 set(CMAKE_C_FLAGS   "${CMAKE_C_FLAGS} ${warnings}")
16 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${warnings}")
17
18 add_executable(toDo main.cc
19                    ToDo.cc)
20
21 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/flavors/mac/part3/build
 > 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: error:
      unused parameter 'argc' [-Werror,-Wunused-parameter]
    int    argc,
           ^
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/mac/part3/main.cc:23:12: error:
      unused parameter 'argv' [-Werror,-Wunused-parameter]
    char** argv
           ^
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/mac/part3/main.cc:58:19: error:
      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: note:
      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
```

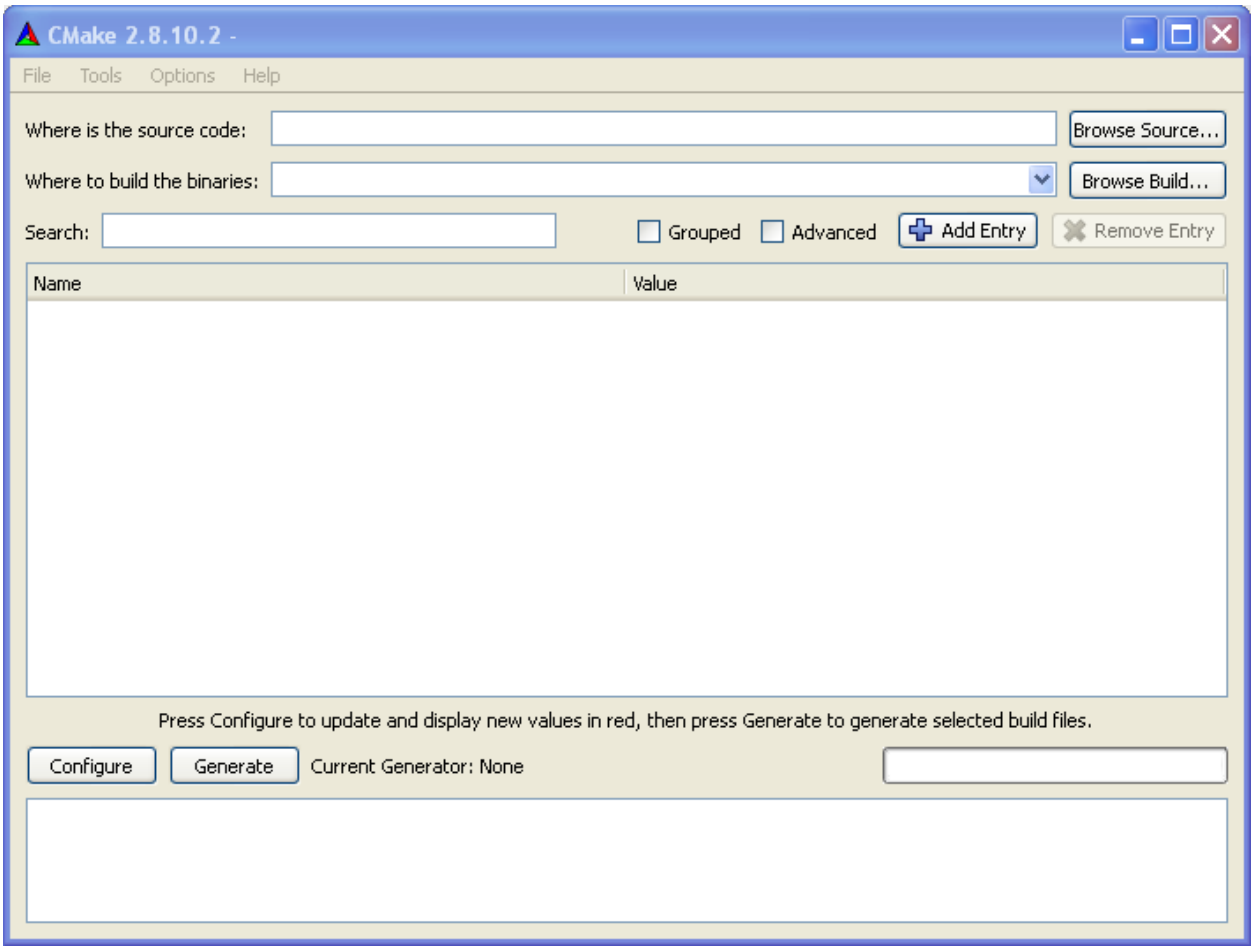
```
68     {
69         return 0;
70     }
71 }
```

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

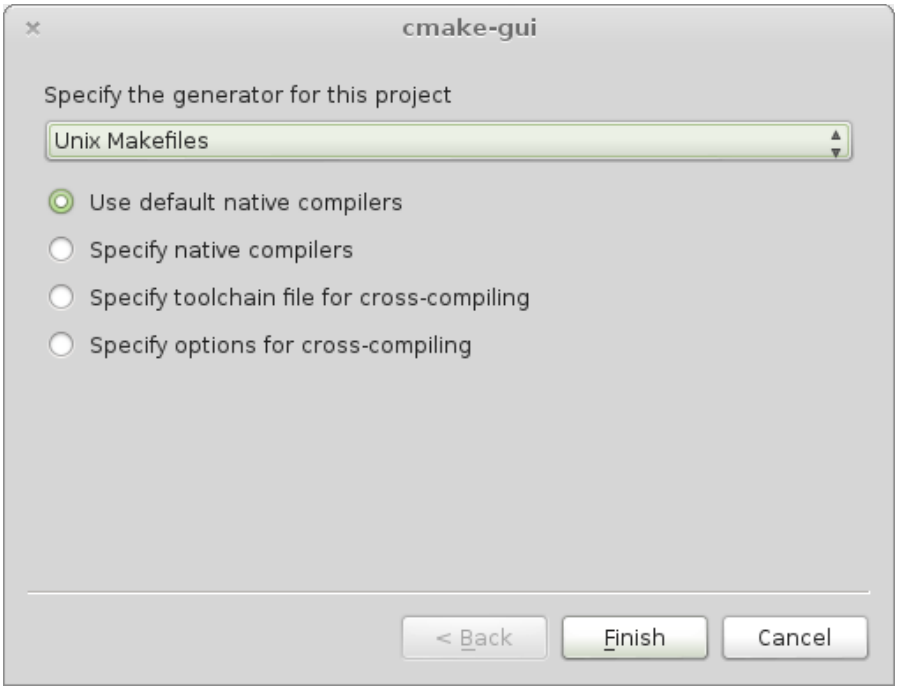
Generating Our Project

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.



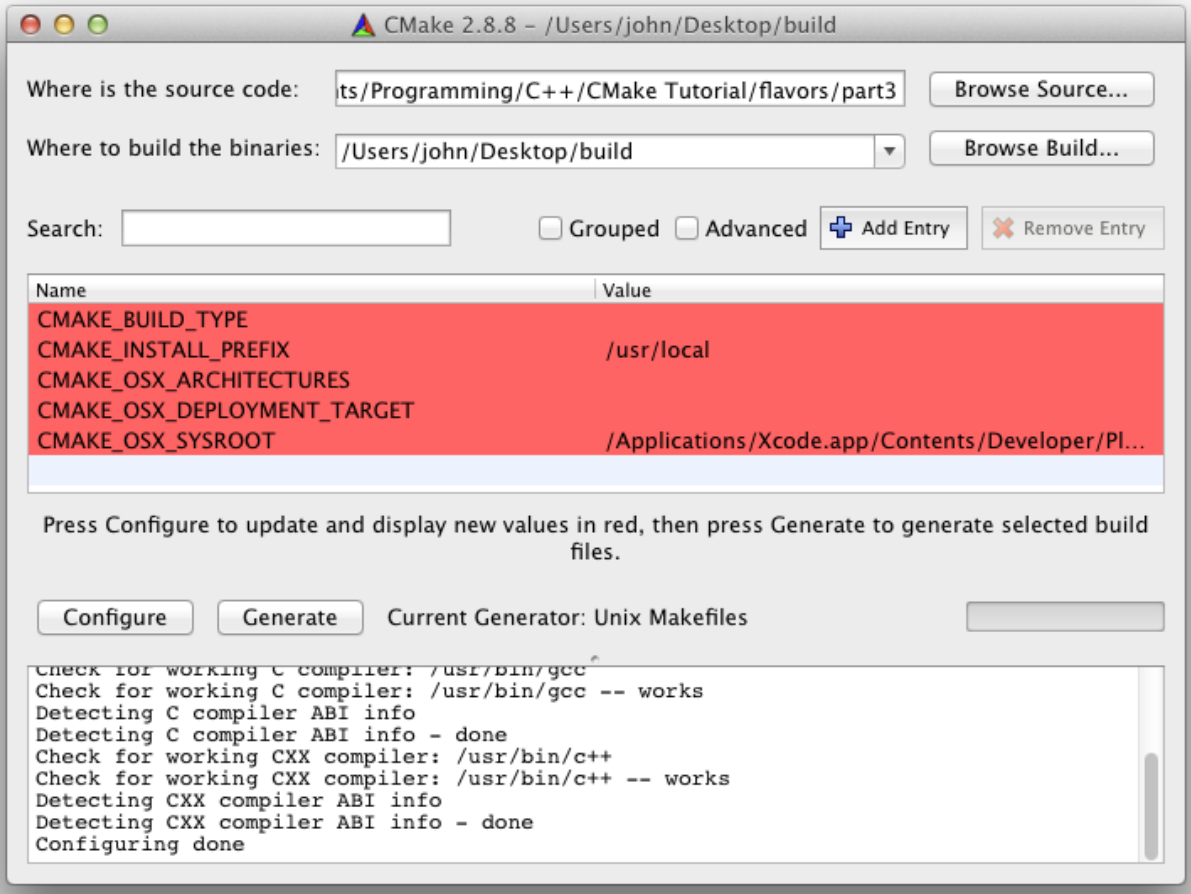
The first two entries should be familiar, but more explicit than what we saw earlier. To relate to the command line we were using: `cd <Where to build the binaries>; cmake <Where is the source code>`. That command line also configures and generates, which you would do using the "Configure" and "Generate"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



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.



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

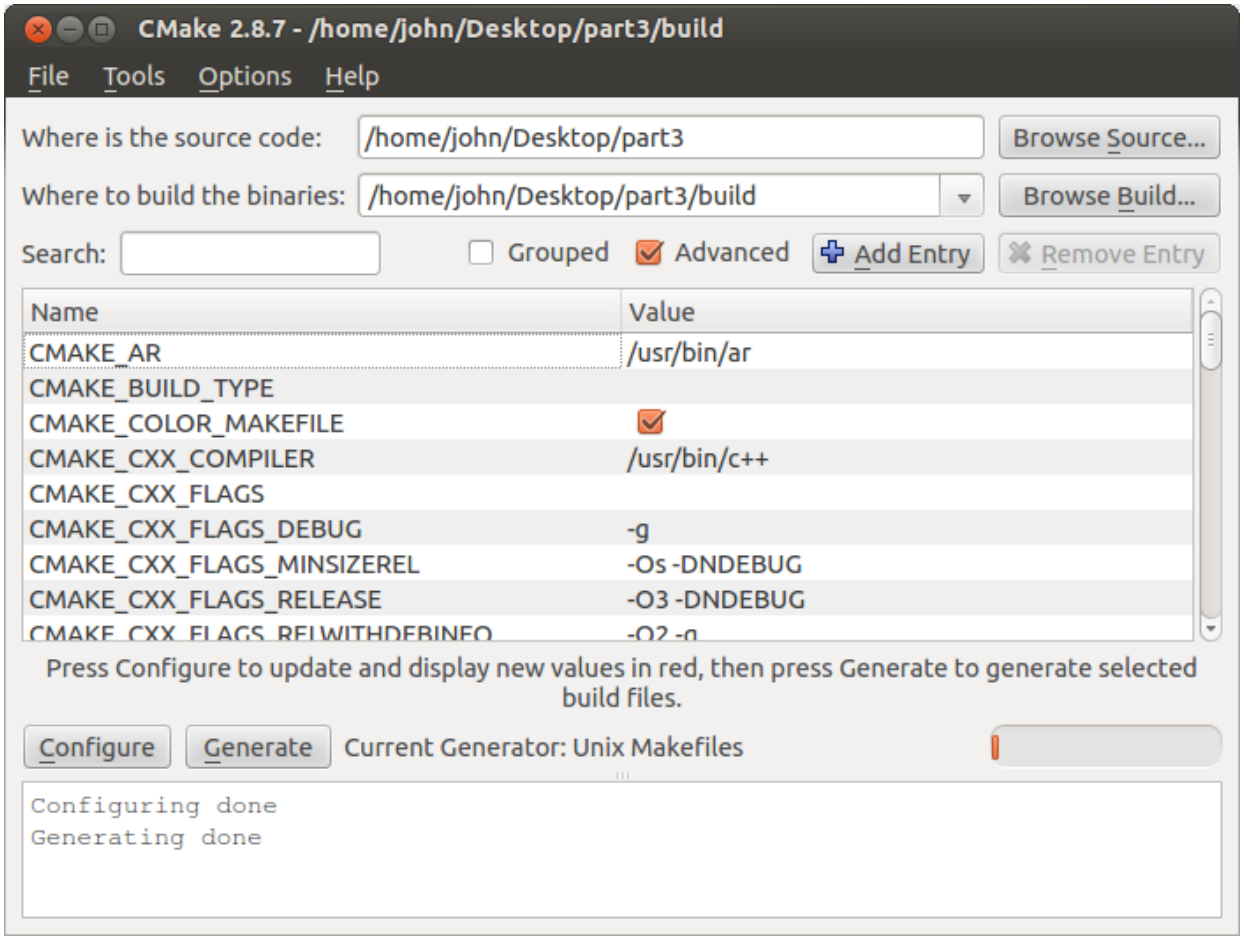
CMake can create an install target which will be covered in a future chapter. This prefix can be set to control where things are installed. It is similar to the `--prefix` argument for `configure` scripts.

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.



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.")
```

`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`

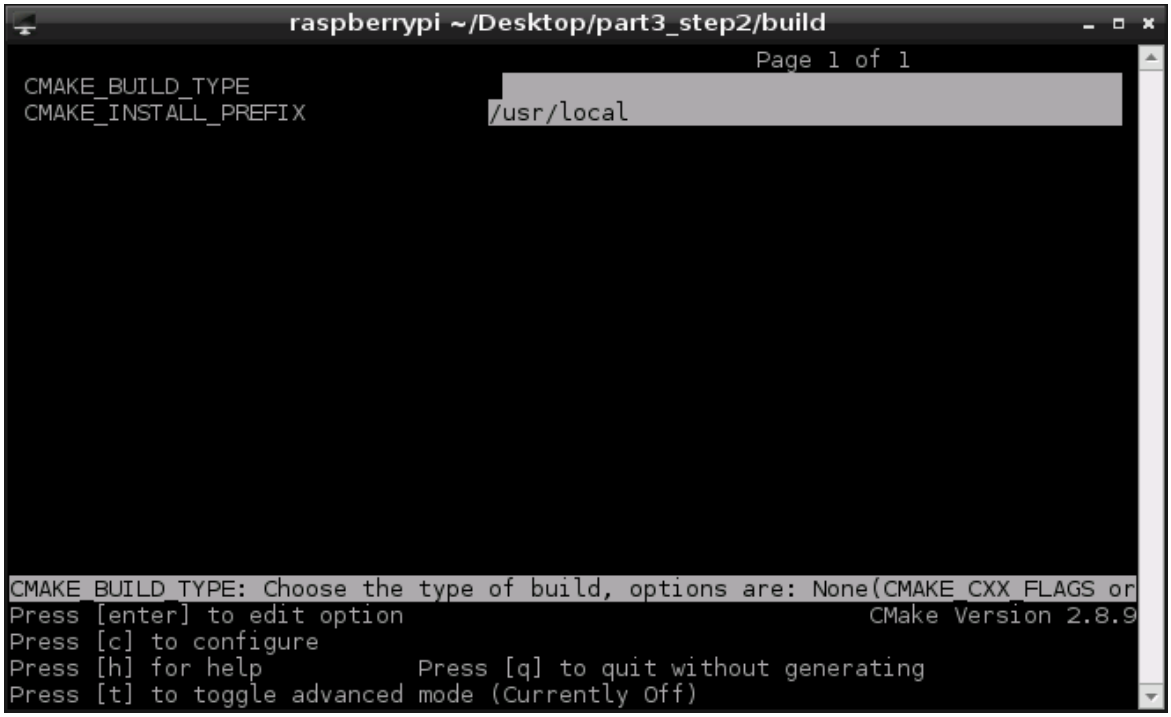
CMake also includes a command line curses-based interface, `ccmake`. It provides equivalent functionality to that of the GUI. Most installations include this tool, although not all. The `ccmake` tool can be used both to create a CMake build or edit the cache of one. To create a new build it is used very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



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.

Chapter 4: Libraries and Subdirectories

Introduction

So far our project is rather simple. A real project would be more complicated than the one we've created. Let's add subdirectories, libraries, and proper unit tests to make our project more realistic.

In this chapter we will split up our project to have a library which we can put in a subdirectory. Then we will use Google Test and Google Mock to add a more realistic unit test.

The Library in a Subdirectory

We will make the ToDo class its own library, and put it in a subdirectory. Even though it is a single source file making it a library actually has one significant advantage. CMake will compile source files once for each target that includes them. So if the ToDo class is used by our command line tool, a unit test, and perhaps a GUI App it would be compiled three times. Imagine if we had a collection of classes instead of just one. This results in a lot of unnecessary compilation.

There were some minor changes to the C++, grab the files here:
(`CMakeLists.txt` listed below)

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()


include_directories(${CMAKE_CURRENT_SOURCE_DIR})

add_subdirectory(ToDoCore)

add_executable(toDo main.cc)
target_link_libraries(toDo toDoCore)

add_test(toDoTest toDo)


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

So now our executable "toDo" only depends on the file "main.cc" and the new library "toDoCore". Our project also has a new subdirectory "ToDoCore".

`include_directories(`*`directories`*`)`

Add *`directories`* to the end of this directory's include paths. We didn't need this before because all of our files were in the same directory.

`include_directories()` documentation (2013-04-20)

`CMAKE_CURRENT_SOURCE_DIR`

The full path to the source directory that CMake is currently processing.

`CMAKE_CURRENT_SOURCE_DIR` documentation (2013-04-20)

`add_subdirectory(source_dir)`

Include the directory *`source_dir`* in your project. This directory *must* contain a `CMakeLists.txt` file.

Note: We're omitting the optional second parameter. This only works with subdirectories of the current directory. We will see how to add external directories later.

`add_subdirectory` documentation (2013-04-20)

`target_link_libraries(target library…)`

Specify that *`target`* needs to be linked against one or more libraries. If a library name matches another target dependencies are setup automatically so that the libraries will be built first and *`target`* will be updated whenever any of the libraries are.

If the *`target`* is an executable then it will be linked against the listed libraries.

If the target is a library then its dependency on these libraries will be recorded. Then when something else links against *`target`* it will also link against *`target`*'s dependencies. This makes it much easier to handle a library's dependencies since you only have to define them once when you define library itself.

For the moment we are using the simplest form of this command. For more information see the documentation (2013-04-20).

When describing `add_subdirectory()` I stated that the subdirectory must contain a `CMakeLists.txt` file. So here's the new file.

ToDoCore/CMakeLists.txt

```
1 add_library(toDoCore ToDo.cc)
```

Conveniently this file is rather simple.

`add_library(target [STATIC | SHARED | MODULE] sources…)`

This command creates a new library *`target`* built from *`sources`*. As you may have noticed this command is very similar to `add_executable`.

With `STATIC`, `SHARED`, and `MODULE` you can specify what kind of library to build. `STATIC` libraries are archives of object files that are linked directly into other targets. `SHARED` libraries are linked dynamically and loaded at runtime. `MODULE` libraries are plug-ins that aren't linked against but can be loaded dynamically at runtime.

If the library type is not specified it will be either `STATIC` or `SHARED`. The default type is controlled by the `BUILD_SHARED_LIBS` variable. By default static libraries are created.

`add_library()` documentation (2013-04-20)

Testing – for Real

We have a rudimentary test but if we were really developing software we'd write a real test using a real testing framework. As mentioned earlier we will use Google Test 1.6.0 and Google Mock 1.6.0. Conveniently they include their own `CMakeLists.txt` files, which makes them easy for us to use.

First the test:

ToDoCore/unit_test/ToDoTest.cc

```
1 #include "ToDoCore/ToDo.h"
2
3 #include <string>
4   using std::string;
5
6 #include <gmock/gmock.h>
7   using ::testing::Eq;
8 #include <gtest/gtest.h>
```

```
#include <gtest/gtest.h>
  using ::testing::Test;


namespace ToDoCore
{
namespace testing
{
    class ToDoTest : public Test
    {
    protected:
        ToDoTest(){}
        ~ToDoTest(){}

        virtual void SetUp(){}
        virtual void TearDown(){}


        ToDo list;

        static const size_t taskCount = 3;
        static const string tasks[taskCount];
    };

    const string ToDoTest::tasks[taskCount] = {"write code",
                                               "compile",
                                               "test"};


    TEST_F(ToDoTest, constructor_createsEmptyList)
    {
        EXPECT_THAT(list.size(), Eq(size_t(0)));
    }

    TEST_F(ToDoTest, addTask_threeTimes_sizeIsThree)
    {
        list.addTask(tasks[0]);
        list.addTask(tasks[1]);
        list.addTask(tasks[2]);

        EXPECT_THAT(list.size(), Eq(taskCount));
    }

    TEST_F(ToDoTest, getTask_withOneTask_returnsCorrectString)
    {
        list.addTask(tasks[0]);

        ASSERT_THAT(list.size(),     Eq(size_t(1)));
        EXPECT_THAT(list.getTask(0), Eq(tasks[0]));
    }

    TEST_F(ToDoTest, getTask_withThreeTasts_returnsCorrectStringForEachIndex)
    {
        list.addTask(tasks[0]);
        list.addTask(tasks[1]);
        list.addTask(tasks[2]);

        ASSERT_THAT(list.size(),     Eq(taskCount));
        EXPECT_THAT(list.getTask(0), Eq(tasks[0]));
        EXPECT_THAT(list.getTask(1), Eq(tasks[1]));
        EXPECT_THAT(list.getTask(2), Eq(tasks[2]));
    }

} // namespace testing
} // namespace ToDoCore
```

This is a rather simple test, but `ToDo` is still a rather simple class. It may look strange if you are unfamiliar with Google Test, taking a look at Google Test Primer may be helpful. I also use a little functionality from Google Mock so Google Mock for Dummies may also be useful.

Now we need to build the test:

ToDoCore/CMakeLists.txt **New or modified lines in bold.**

```
1 add_library(toDoCore ToDo.cc)
2
3 add_subdirectory(unit_test)
```

ToDoCore/unit_test/CMakeLists.txt

```
1 set(GMOCK_DIR "../../../../../gmock/gmock-1.6.0"
2     CACHE PATH "The path to the GoogleMock test framework.")
3
4 if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
5     # force this option to ON so that Google Test will use /MD instead of /MT
6     # /MD is now the default for Visual Studio, so it should be our default, too
7     option(gtest_force_shared_crt
8            "Use shared (DLL) run-time lib even when Google Test is built as static lib."
9            ON)
10 elseif (APPLE)
11     add_definitions(-DGTEST_USE_OWN_TR1_TUPLE=1)
12 endif()
13 add_subdirectory(${GMOCK_DIR} ${CMAKE_BINARY_DIR}/gmock)
14
15 include_directories(SYSTEM ${GMOCK_DIR}/gtest/include
16                            ${GMOCK_DIR}/include)
17
18
19 add_executable(ToDoTest ToDoTest.cc)
20 target_link_libraries(ToDoTest toDoCore
21                                gmock_main)
22
23 add_test(ToDoTest ToDoTest)
```

First we add the Google Mock directory to our project then we add our test. The path to Google Mock is stored in a cached variable so that you can easily set it to the correct value either from the command line or via one of the GUIs. There are several potential problems with that line but we will worry about those later, for now it's good enough. Okay I oversimplified a little. We don't just add the Google Mock directory, we also work around some OS-specific problems.

When using Visual Studio to build our test we would run into a problem. Even when building static libraries, CMake's default, MSVC defaults to linking against the multi-threaded, DLL-specific version of the standard library. By default Google Test overrides this so that the non-DLL version of the multi-threaded standard library is used. Then when our test links against both `toDoCore` and `gmock_main` the linker will output a large number of errors since we would be linking against two different copies of the standard library. To avoid this problem we force Google Test to use the DLL-specific version to match Visual Studio's default by setting the `gtest_force_shared_crt` option to `ON`. See Microsoft C/C++ Compiler Run-Time Library.

The second problem occurs on newer version of Mac OS X which default to using a different standard library that fully supports C++11. GTest uses the `tuple` class from the draft TR1 standard and therefore looks for it in the `std::tr1` namespace. The `tr1` namespace is not present in the C++11 standard library that Apple uses so GTest cannot find it and won't compile. We fix this by telling GTest to use its own `tuple` implementation.

`add_subdirectory(`*`source_dir [binary_dir]`*`)`

> Add the directory *`source_dir`* to the current project with *`binary_dir`* as its corresponding binary output directory. When adding a directory that is a subdirectory of the current directory CMake will automatically determine what the binary output directory should be, making the second argument optional. However if you add a directory that isn't a subdirectory you need to specify the binary output directory.
>
> `add_subdirectory` documentation (2013-04-20)

`CMAKE_BINARY_DIR`

> This variable holds the path to the top level binary output directory, i.e. the directory in which you ran the `cmake` command or the path you chose for "Where to build the binaries" in the GUI.
>
> `CMAKE_BINARY_DIR` documentation (2013-04-27)

`include_directories([AFTER|BEFORE] [SYSTEM] `*`directory…`*`)`

> `AFTER|BEFORE`

Specify whether or not these include directories should be appended or prepended to the list of include directories. If omitted then the default behavior is used.

By default directories are appended to the list. This behavior can be changed by setting `CMAKE_INCLUDE_DIRECTORIES_BEFORE` to `TRUE`.

`SYSTEM`

Specify that these directories are system include directories. This only has an affect on compilers that support the distinction. This can change the order in which the compiler searches include directories or the handling of warnings from headers found in these directories.

`directory…`

The directories to be added to the list of include directories.

`include_directories()` documentation (2013-04-20)

`option(`*`name docstring [initialValue]`*`)`

Provide a boolean option to the user. This will be displayed in the GUI as a checkbox. Once created the value of the option can be accessed as the variable *`name`*. The *`docstring`* will be displayed in the GUI to tell the user what this option does. If no initial value is provided it defaults to OFF.

While this boolean option is stored in the cache and accessible as a variable you cannot override the *`initialValue`* by setting a variable of the same name beforehand, not even by passing a `-D` command line option to CMake. Which is why we have to define the option ourselves before Google Test does.

`option()` documentation (2013-05-3)

`add_definitions(`*`flags…`*`)`

Add preprocessor definitions to the compiler command line for targets in the current directory and those below it. While this command is intended for adding definitions you still need to precede them with `-D`.

Because this command modifies the `COMPILE_DEFINITIONS` directory property it affects *all* targets in the directory, even those that were defined **before** this command was used. If this is not the desired effect then modifying the `COMPILE_DEFINITIONS` property of particular targets or source files will work better. (Properties are introduced below.)

`add_definitions()` documentation (2014-09-28)

`COMPILE_DEFINITIONS` directory property documentation (2014-09-28)

`COMPILE_DEFINITIONS` target property documentation (2014-09-28)

`COMPILE_DEFINITIONS` source file property documentation (2014-09-28)

Let's go ahead and try out our new test!

```
 > mkdir build
 > cd build
 > cmake -G "Unix Makefiles" ..
-- The C compiler identification is Clang 4.2.0
-- The CXX compiler identification is Clang 4.2.0
-- 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
-- Found PythonInterp: /usr/local/bin/python (found version "2.7.3")
-- Looking for include file pthread.h
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /Documents/Programming/C++/CMake Tutorial/flavors/part4_step2/build
 > make
Scanning dependencies of target toDoCore
[ 14%] Building CXX object ToDoCore/CMakeFiles/toDoCore.dir/ToDo.cc.o
Linking CXX static library libtoDoCore.a
[ 14%] Built target toDoCore
Scanning dependencies of target toDo
[ 28%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
Linking CXX executable toDo
[ 28%] Built target toDo
Scanning dependencies of target gtest
[ 42%] Building CXX object gmock/gtest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
In file included from /Documents/Programming/C++/gmock/gmock-1.6.0/gtest/src/gtest-all.cc:42:
In file included from /Documents/Programming/C++/gmock/gmock-1.6.0/gtest/src/gtest.cc:132:
/Documents/Programming/C++/gmock/gmock-1.6.0/gtest/src/gtest-internal-inl.h:206:8: error:
      private field 'pretty_' is not used [-Werror,-Wunused-private-field]
  bool pretty_;
       ^
1 error generated.
make[2]: *** [gmock/gtest/CMakeFiles/gtest.dir/src/gtest-all.cc.o] Error 1
make[1]: *** [gmock/gtest/CMakeFiles/gtest.dir/all] Error 2
make: *** [all] Error 2
```

Oh noes! Newer versions of Clang have some pretty strict warnings and we have just run afoul of one. So we have a problem: we want to use strict compiler settings to ensure we write good code but we also don't want to go changing Google Test. As it turns out CMake actually provides us the flexibility we need to disable warnings for just the `gtest` target.

This is a capability that can easily be abused. In the case of Google Test we didn't write it and we know, or at least assume, that it works fine. Because of that we don't care about any warnings we might find in Google Test's code. We need to be careful not to use this feature to allow ourselves to write poor code.

ToDoCore/unit_test/CMakeLists.txt **New or modified lines in bold.**

```
1  set(GMOCK_DIR "../../../../../gmock/gmock-1.6.0"
2      CACHE PATH "The path to the GoogleMock test framework.")
3
4  if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
5      # force this option to ON so that Google Test will use /MD instead of /MT
6      # /MD is now the default for Visual Studio, so it should be our default, too
7      option(gtest_force_shared_crt
8             "Use shared (DLL) run-time lib even when Google Test is built as static lib."
9             ON)
10 elseif (APPLE)
11     add_definitions(-DGTEST_USE_OWN_TR1_TUPLE=1)
12 endif()
13 add_subdirectory(${GMOCK_DIR} ${CMAKE_BINARY_DIR}/gmock)
14 set_property(TARGET gtest APPEND_STRING PROPERTY COMPILE_FLAGS " -w")
15
16 include_directories(SYSTEM ${GMOCK_DIR}/gtest/include
17                            ${GMOCK_DIR}/include)
18
19
20 add_executable(ToDoTest ToDoTest.cc)
21 target_link_libraries(ToDoTest toDoCore
22                               gmock_main)
23
24 add_test(ToDoTest ToDoTest)
```

`set_property(TARGET gtest APPEND_STRING PROPERTY COMPILE_FLAGS " -w")`

There are a variety of things that have properties in CMake, in this case we are interested in a target's properties. Each target can have it's own compiler flags in addition the ones set in `CMAKE_<LANG>_FLAGS`. Here we append " `-w`" to `gtest`'s `COMPILE_FLAGS`. The flag "`-w`" disables all warnings for both GCC and Clang. When compiling with MSVC the "`-w`" will be automatically converted to "`/w`" which has the same function. (Although it will warn that "`/w`" is overriding "`/W4`")

`COMPILE_FLAGS` documentation (2013-04-28)

GCC Warning Options (2013-04-28), currently these work for Clang too.

Microsoft C/C++ Compiler Warning Level (2013-04-28)

`set_property(TARGET` *`target_name…`* `[APPEND|APPEND_STRING] PROPERTY` *`name value…`*`)`

`TARGET`

Specify that we want to set the property of a target. Several other types of things have properties you can set. For the moment we are only going to deal with targets, but the concept is the same for the rest.

`target_name…`

The name of the target whose property you want to set. You can list multiple targets and all will have the property set the same way for each.

`[ APPEND | APPEND_STRING ]`

Append to the property's existing value instead of setting it. `APPEND` appends to the property as a list. `APPEND_STRING` appends to the property as a string.

Note: Do not provide a multiple values when using `APPEND_STRING` as the results will not be what you expect.

Don't worry about lists we will cover them in the next chapter.

`PROPERTY`

`name`

The name of the property you want to set. See Properties on Targets.

`value…`

The value to set for the property. If multiple values are provided they are treated as a list. Only provide one value if also using `APPEND_STRING`.

Don't worry about lists yet.

`set_property()` documentation (2013-04-28)

Let's give this version a try.

```
 > mkdir build
 > cd build
 > cmake -G "Unix Makefiles" ..
-- The C compiler identification is Clang 4.2.0
-- The CXX compiler identification is Clang 4.2.0
-- 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
-- Found PythonInterp: /usr/local/bin/python (found version "2.7.3")
-- Looking for include file pthread.h
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /Documents/Programming/C++/CMake Tutorial/flavors/part4_step3/build
 > make
Scanning dependencies of target toDoCore
[ 14%] Building CXX object ToDoCore/CMakeFiles/toDoCore.dir/ToDo.cc.o
Linking CXX static library libtoDoCore.a
[ 14%] Built target toDoCore
Scanning dependencies of target toDo
[ 28%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
Linking CXX executable toDo
[ 28%] Built target toDo
Scanning dependencies of target gtest
[ 42%] Building CXX object gmock/gtest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
Linking CXX static library libgtest.a
[ 42%] Built target gtest
Scanning dependencies of target gmock
[ 57%] Building CXX object gmock/CMakeFiles/gmock.dir/src/gmock-all.cc.o
Linking CXX static library libgmock.a
[ 57%] Built target gmock
Scanning dependencies of target gmock_main
[ 71%] Building CXX object gmock/CMakeFiles/gmock_main.dir/src/gmock_main.cc.o
Linking CXX static library libgmock_main.a
[ 71%] Built target gmock_main
Scanning dependencies of target ToDoTest
[ 85%] Building CXX object ToDoCore/unit_test/CMakeFiles/ToDoTest.dir/ToDoTest.cc.o
Linking CXX executable ToDoTest
[ 85%] Built target ToDoTest
Scanning dependencies of target gtest_main
[100%] Building CXX object gmock/gtest/CMakeFiles/gtest_main.dir/src/gtest_main.cc.o
Linking CXX static library libgtest_main.a
[100%] Built target gtest_main
 > make test
Running tests...
Test project /Documents/Programming/C++/CMake Tutorial/flavors/part4_step3/build
    Start 1: ToDoTest
1/1 Test #1: ToDoTest .........................   Passed    0.00 sec
100% tests passed, 0 tests failed out of 1
Total Test time (real) =   0.01 sec
 > ToDoCore/unit_test/ToDoTest
Running main() from gmock_main.cc
[==========] Running 4 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 4 tests from ToDoTest
[ RUN      ] ToDoTest.constructior_createsEmptyList
[       OK ] ToDoTest.constructior_createsEmptyList (0 ms)
```

```
[ RUN      ] ToDoTest.addTask_threeTimes_sizeIsThree
[       OK ] ToDoTest.addTask_threeTimes_sizeIsThree (0 ms)
[ RUN      ] ToDoTest.getTask_withOneTask_returnsCorrectString
[       OK ] ToDoTest.getTask_withOneTask_returnsCorrectString (0 ms)
[ RUN      ] ToDoTest.getTask_withThreeTasts_returnsCorrectStringForEachIndex
[       OK ] ToDoTest.getTask_withThreeTasts_returnsCorrectStringForEachIndex (1 ms)
[----------] 4 tests from ToDoTest (1 ms total)
[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 4 tests.
```

Yay! Everything works now and our test passes, too.

Next we will focus on how we could add more unit tests (if we had more units) without duplicating the work we've done here. Also we will make it so that our unit tests are automatically run as needed whenever we build.

Chapter 5: Functionally Improved Testing

Introduction

Last time we added a nice unit test and then set up CMake to build it, of course, and add it to the list of tests that CTest will run. This is great, now we can run `cmake` then use `make` and `make test` to test our project. Now it's time to build on our success because we certainly aren't done yet.

The main problem we need to tackle is that there are currently 3 steps to creating a test program:

1. add the executable target
2. link the executable against the "gmock_main" library
3. add the test to CTest's list of tests

That's 3 steps too many. If you are thinking that 3 steps aren't too many remember that any project of a useful size will have a rather large number of unit tests, each of which will require these same 3 steps – that's a lot of repetition. As programmers we should not repeat ourselves, and we shouldn't slack off just because we are merely setting up our build system. What we want is the ability to add a new test in a single step. Writing the test is hard enough, building and running it should be easy.

Lucky for us CMake offers the ability to write functions. So we will start by writing a function that combines these 3 steps so that only one step will be needed. Once we have the function we will improve it further taking advantage of the fact that we will only have to write said improvements once.

A Simple Function

We have 3 simple steps to encapsulate in a function, that should be simple, right?

ToDoCore/unit_test/CMakeLists.txt **New or modified lines in bold.**

```
set(GMOCK_DIR "../../../../../gmock/gmock-1.6.0"
    CACHE PATH "The path to the GoogleMock test framework.")

if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
    # force this option to ON so that Google Test will use /MD instead of /MT
    # /MD is now the default for Visual Studio, so it should be our default, too
    option(gtest_force_shared_crt
           "Use shared (DLL) run-time lib even when Google Test is built as static lib."
           ON)
elseif (APPLE)
    add_definitions(-DGTEST_USE_OWN_TR1_TUPLE=1)
endif()
add_subdirectory(${GMOCK_DIR} ${CMAKE_BINARY_DIR}/gmock)
set_property(TARGET gtest APPEND_STRING PROPERTY COMPILE_FLAGS " -w")

include_directories(SYSTEM ${GMOCK_DIR}/gtest/include
                           ${GMOCK_DIR}/include)


#
# add_gmock_test(<target> <sources>...)
#
#  Adds a Google Mock based test executable, <target>, built from <sources> and
#  adds the test so that CTest will run it. Both the executable and the test
#  will be named <target>.
#
function(add_gmock_test target)
    add_executable(${target} ${ARGN})
    target_link_libraries(${target} gmock_main)

    add_test(${target} ${target})
endfunction()


add_gmock_test(ToDoTest ToDoTest.cc)
target_link_libraries(ToDoTest toDoCore)
```

I like to put comments before my functions that show how they should be called and explain what they do.

`function(add_gmock_test target)`

Start the definition of the function `add_gmock_test` with one required parameter `target`.

Inside the function its first argument is available as the variable `target` and the rest of the arguments are available in a list stored in the variable `ARGN`. CMake will allow you to pass more arguments to a function than the number of parameters it defined. It is up to the writer of the function to handle all of them, validate them and produce an error if they aren't correct, or merely ignore them. In this case we are just passing them all on to the command `add_executable()`.

Also available is the variable `ARGC` which holds the count of all arguments passed to the function, both ones matching parameters and any extras. Additionally each argument can be accessed via the variables `ARGV0`, `ARGV1`, ... `ARGVN`. As if that weren't enough ways to access function arguments all arguments are also available as a list stored in the variable `ARGV`. This affords a lot of flexibility but can make argument validation and handling difficult.

`function()` documentation (2013-06-01)

`endfunction()`

Ends the definition of a function. As I've said before CMake's syntax is a bit strange. You can pass the name of the function as an argument to this command, but it is not required. If you do it should match otherwise CMake will print a warning when configuring. I think it's easier to read if no arguments are passed to `endfunction()` and functions shouldn't be long enough that a reminder of what function is being ended is needed.

`endfunction()` documentation (2013-06-01)

`add_gmock_test(ToDoTest ToDoTest.cc)`

Now we use the function we just wrote to add our Google Mock based test. With the function written it is now much simpler as we don't need to write out the three separate commands every time.

```
target_link_libraries(ToDoTest toDoCore)
```

We still have to link our test with the "toDoCore" library. Since this is specific to this test and not all tests it wouldn't make sense to include this in our function.

Commands and Functions and Macros! Oh my!

So far we have seen several CMake commands and now even written a function! You may wonder what the difference is between a command and a function. Simply put commands are built into CMake and functions are written using CMake's language. While some commands behave quite similarly to functions, e.g. `add_executable`, some others behave in ways that cannot be mimicked using functions or macros, e.g. `if()` and `function()`.

Macros, on the other hand, are similar to functions in that they are written the same and offer all of the same ways for accessing arguments. However, macros don't have their own scope and rather than dereferencing arguments when run arguments are replaced instead. The first difference is what makes macros both useful and dangerous, the second is more subtle and can make working with lists difficult. (Yes, I know. I haven't talked about lists yet.)

You can't add commands, but you can create functions and macros. As a rule of thumb do not use a macro unless absolutely necessary, then you will avoid many problems.

Scope

Scope is interesting in CMake and can occasionally be confusing. There's local scope, directory scope, global scope, and cache scope. As with most languages things are inherited from enclosing scopes. For example if you were to set `someVariable` to "some value" and then call `someFunction()` inside the function dereferencing `someVariable` would yield "some value".

Local Scope
: This refers to the most narrow scope at a given location. So the current function or directory if not inside a function. Note that conditionals, loops, and macros do not create a new scope, which is important to remember. When you set a variable this is the scope that is affected.

Parent Scope
: The scope enclosing the current local scope. For example the scope that called the current function or the directory that executed the most recent `add_subdirectory()` command. This is important because the `set()` command can be used to set variables in the parent scope. In fact this is the only way to return values from a function.

```
set(variable values… PARENT_SCOPE)
```

`set()` documentation (2013-06-01)

Directory Scope
: This is the scope of the current directory being processed by CMake which is used by directory properties and macros. The confusing thing is that some commands affect directory properties, such as `add_definitions()` and `remove_definitions()`. Many of these properties affect the targets created within this directory scope but only take effect when generating. So if you create a target and then use the `add_definitions()` command those definitions will apply to the target created previously. It is less confusing if things that affect directory scope are done before creating any targets in that directory. Also do not mix setting directory properties and creating targets inside a function, either use separate functions or set the corresponding target property.

Global Scope
: As expected anything defined with global scope is accessible from within any local scope. Targets, functions, and global properties all have global scope. For this reason all targets must have unique names. (Strictly speaking this isn't true, however not all generators can handle multiple targets with the same name. For maximum compatibility it is best to ensure all targets have unique names.) Functions, on the other hand, can be redefined at will, but that is generally not a good idea.

Cache Scope
: This is similar to global scope, however only variables can be stored in the cache. In addition the cache persists between CMake configure runs. As we have already seen some cached variables can also be edited using the CMake GUI or the `ccmake` tool.

Let's Include Some Organization

There's two issues with what we have now. First we've combined settings and functions for unit testing as well as an actual target. Second burying the inclusion of Google Mock this deep in our project makes it difficult to use a relative path. If you were to set the path to Google Mock on the command line using `cmake -DGMOCK_DIR=somePath` you would expect the path to be relative to the top project directory rather than two directories deeper. We can fix both of these problems at the same time.

We will refactor the code related to Google Mock into a separate file. Which will resolve problem one. Then we will include our new file from the top `CMakeLists.txt` file, which will address problem two. The question is where to put this new file and what to call it? In CMake files like these are called modules. Cmake comes with many which are stored in a directory called "Modules". Many software projects, on the other hand, store CMake related code in a directory called "cmake", a logical name, sometimes this is done out of necessity (e.g. if using ClearCase). I think we shall put the file in `cmake/Modules`. As for the name since we consistently used `gmock` or `GMOCK` let's go with `gmock.cmake`.

cmake/Modules/gmock.cmake

```
set(GMOCK_DIR "../../../gmock/gmock-1.6.0"
    CACHE PATH "The path to the GoogleMock test framework.")

if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
    # force this option to ON so that Google Test will use /MD instead of /MT
    # /MD is now the default for Visual Studio, so it should be our default, too
    option(gtest_force_shared_crt
           "Use shared (DLL) run-time lib even when Google Test is built as static lib."
           ON)
elseif (APPLE)
    add_definitions(-DGTEST_USE_OWN_TR1_TUPLE=1)
endif()
add_subdirectory(${GMOCK_DIR} ${CMAKE_BINARY_DIR}/gmock)
set_property(TARGET gtest APPEND_STRING PROPERTY COMPILE_FLAGS " -w")

include_directories(SYSTEM ${GMOCK_DIR}/gtest/include
                           ${GMOCK_DIR}/include)


#
# add_gmock_test(<target> <sources>...)
#
#  Adds a Google Mock based test executable, <target>, built from <sources> and
#  adds the test so that CTest will run it. Both the executable and the test
#  will be named <target>.
#
function(add_gmock_test target)
    add_executable(${target} ${ARGN})
    target_link_libraries(${target} gmock_main)

    add_test(${target} ${target})

endfunction()
```

If you look closely the only change to this code you'll notice is that the default value for `GMOCK_DIR` has two fewer parent directories in it. It is now relative to the top of our project as one would expect.

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")

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

enable_testing()
include(gmock)


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()


include_directories(${CMAKE_CURRENT_SOURCE_DIR})

add_subdirectory(ToDoCore)

add_executable(toDo main.cc)
target_link_libraries(toDo toDoCore)


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

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

Lists, finally! Okay not quite yet. Here we append the "Modules" directory we created to CMake's module path. This is the path CMake searches when you include a module.

We set the include path because, in the future, we might want to include modules from other `CMakeLists.txt` in other directories. This allows us to include them without having to specify the full path every time.

`include(gmock)`

This includes the new module we created. When used this way CMake searches the module path for the file `gmock.cmake` and when it finds the file it is included. These includes are much like those done by the C preprocessor. The code in the included file executes in the same scope as the file that included it.

`list(APPEND *list elements…*)`

Appends the elements to the list stored in the variable named *`list`*. That's correct, you pass in the *name* of the list to be updated, you do not dereference it.

`list()` documentation (2013-06-04)

`CMAKE_MODULE_PATH`

When including modules CMake searches for the requested module in the paths in this list. If this list is exhausted then CMake will look in the directory containing the default modules that come with CMake. Because these paths need to work anywhere in the build tree they must be absolute paths. Since this is a list the `list()` command should be used to manipulate it.

`CMAKE_MODULE_PATH` documentation (2013-06-04)

`include(*module* | *file*)`

Include the module or file in the current file being processed. If a *module* name is provided CMake will search for the file *`module`*`.cmake` and included it if found. Alternatively if a *`file`* name is provided CMake will include that file directly; no module path searching is required. If the file cannot be included either because

it doesn't exist or wasn't found CMake will issue a warning, but will continue processing.

`include()` documentation (2013-06-04)

ToDoCore/unit_test/CMakeLists.txt

```
1 add_gmock_test(ToDoTest ToDoTest.cc)
2 target_link_libraries(ToDoTest toDoCore)
```

This file has gone on a serious diet. After moving all general code for unit testing with Google Mock into `gmock.cmake` this file became quite simple.

Lists!

At long last! You've been teased by lists for 2 chapters now, and most of this one too. It is high time we discussed lists.

CMake has two data structures built in: strings and lists. Well, strictly speaking that isn't completely true; lists are semicolon delimited strings. So an empty string is also an empty list and a regular string is a list with only one item. The simplest way to make a list is `set(myList a b c)` which is exactly the same as `set(myList a;b;c)`. However `set(myList "a;b;c")` creates a list with just one item. If a string begins with `"` it is treated as a string literal and any spaces or quotes remain a part of that string rather than causing it to be split into several list items.

Lists are important to understand not just because they are useful but also because all arguments to commands, functions, and macros are processed as a list. So just as `set(myList a b c)` is the same as `set(myList a;b;c)` so too is `set(myList;a;b;c)`. When CMake processes the call to the `set()` command it collects all of the arguments into a single list. This list (`ARGV`) is the separated into the first argument, the variable name (`myList`), and the rest of the items, the values (`a;b;c`). This can cause trouble if you pass a quoted string containing semicolons to a function that then passes it to another function without quoting it as your string will become a list.

While you can create list with `set(myList a b c)` I'd strongly recommend using `list(APPEND myList a b c)`. Using the `list()` command shows that you are using the variable `myList` as a list. Naturally the `list()` command allows you to do other things with lists.

`list()` documentation (2013-06-04)

Auto Play

Well really automatic test running. So far in my experience it takes significantly less time to run unit tests than it does to build them. For this reason I think it is beneficial to run your unit tests every time they are built. This also has the side effect of stopping your build if the unit test fails.

cmake/Modules/gmock.cmake **New or modified lines in bold.**

```
set(GMOCK_DIR "../../../gmock/gmock-1.6.0"
    CACHE PATH "The path to the GoogleMock test framework.")

if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
    # force this option to ON so that Google Test will use /MD instead of /MT
    # /MD is now the default for Visual Studio, so it should be our default, too
    option(gtest_force_shared_crt
           "Use shared (DLL) run-time lib even when Google Test is built as static lib."
           ON)
elseif (APPLE)
    add_definitions(-DGTEST_USE_OWN_TR1_TUPLE=1)
endif()
add_subdirectory(${GMOCK_DIR} ${CMAKE_BINARY_DIR}/gmock)
set_property(TARGET gtest APPEND_STRING PROPERTY COMPILE_FLAGS " -w")

include_directories(SYSTEM ${GMOCK_DIR}/gtest/include
                           ${GMOCK_DIR}/include)


#
# add_gmock_test(<target> <sources>...)
#
#  Adds a Google Mock based test executable, <target>, built from <sources> and
#  adds the test so that CTest will run it. Both the executable and the test
#  will be named <target>.
#
function(add_gmock_test target)
    add_executable(${target} ${ARGN})
    target_link_libraries(${target} gmock_main)

    add_test(${target} ${target})

    add_custom_command(TARGET ${target}
                       POST_BUILD
                       COMMAND ${target}
                       WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
                       COMMENT "Running ${target}" VERBATIM)
endfunction()
```

```
add_custom_command(TARGET ${target}
                   POST_BUILD
                   COMMAND ./${target}
                   WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
                   COMMENT "Running ${target}" VERBATIM)
```

We use the `add_custom_command()` command to run each test after each time it is built. Here we simply run the test and if it fails the build will stop. However if you were to build again immediately the failed test would **not** be run again and the build will continue. Fixing that will be left for later.

```
add_custom_command(TARGET target
                   PRE_BUILD | PRE_LINK | POST_BUILD
                   COMMAND command [arguments…]
                   [COMMAND command2 [arguments…] …]
                   [WORKING_DIRECTORY directory]
                   [COMMENT comment] [VERBATIM])
```

target

The name of the target to which we are adding the custom command.

PRE_BUILD | PRE_LINK | POST_BUILD

When to run the custom command. PRE_BUILD will run the command before any of the target's other dependencies. PRE_LINK runs the command after all other dependencies. Lastly POST_BUILD runs the command after the target has been built.

Note: the PRE_BUILD option only works with Visual Studio 7 or newer. For all other generators it is treated as PRE_LINK instead.

COMMAND *command [arguments…]*

The command to run and any arguments to be passed to it. If *command* specifies an executable

target, i.e. one created with the `add_executable()` command, the location of the actual built executable will replace the name; additionally a target level dependency will be added so that the executable target will be built before this custom command is run.

Note: target level dependencies merely control the order in which targets are build. If a target level dependency is rebuilt this command will not be re-run.

Any number of commands can be listed using this syntax and they will all be run in order each time.

`[ WORKING_DIRECTORY directory ]`

Specify the working directory from which the listed commands will be run.

`[ COMMENT comment ]`

Provide a comment that will be displayed before the listed commands are run.

`[ VERBATIM ]`

This argument tells CMake to ensure that the commands and their arguments are escaped appropriately for whichever build tool is being used. If this argument is omitted the behavior is platform and tool specific. Therefore it is **strongly** recommended that you always provide the `VERBATIM` argument.

`add_custom_command()` documentation (2013-06-15)

Now it's time to see our hard work in action.

```
 > mkdir build
 > cd build
 > cmake -G "Unix Makefiles" ..
-- The C compiler identification is Clang 4.2.0
-- The CXX compiler identification is Clang 4.2.0
-- 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
-- Found PythonInterp: /usr/local/bin/python (found version "2.7.3")
-- Looking for include file pthread.h
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /Documents/Programming/CMake/CMake Tutorial/flavors/part5_step3/build
 > make
Scanning dependencies of target toDoCore
[ 14%] Building CXX object ToDoCore/CMakeFiles/toDoCore.dir/ToDo.cc.o
Linking CXX static library libtoDoCore.a
[ 14%] Built target toDoCore
Scanning dependencies of target toDo
[ 28%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
Linking CXX executable toDo
[ 28%] Built target toDo
Scanning dependencies of target gtest
[ 42%] Building CXX object gmock/gtest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
Linking CXX static library libgtest.a
[ 42%] Built target gtest
Scanning dependencies of target gmock
[ 57%] Building CXX object gmock/CMakeFiles/gmock.dir/src/gmock-all.cc.o
Linking CXX static library libgmock.a
[ 57%] Built target gmock
Scanning dependencies of target gmock_main
[ 71%] Building CXX object gmock/CMakeFiles/gmock_main.dir/src/gmock_main.cc.o
Linking CXX static library libgmock_main.a
[ 71%] Built target gmock_main
Scanning dependencies of target gtest_main
[ 85%] Building CXX object gmock/gtest/CMakeFiles/gtest_main.dir/src/gtest_main.cc.o
Linking CXX static library libgtest_main.a
[ 85%] Built target gtest_main
Scanning dependencies of target ToDoTest
[100%] Building CXX object ToDoCore/unit_test/CMakeFiles/ToDoTest.dir/ToDoTest.cc.o
Linking CXX executable ToDoTest
Running ToDoTest
Running main() from gmock_main.cc
[==========] Running 4 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 4 tests from ToDoTest
[ RUN      ] ToDoTest.constructor_createsEmptyList
[       OK ] ToDoTest.constructor_createsEmptyList (0 ms)
[ RUN      ] ToDoTest.addTask_threeTimes_sizeIsThree
[       OK ] ToDoTest.addTask_threeTimes_sizeIsThree (0 ms)
[ RUN      ] ToDoTest.getTask_withOneTask_returnsCorrectString
[       OK ] ToDoTest.getTask_withOneTask_returnsCorrectString (0 ms)
[ RUN      ] ToDoTest.getTask_withThreeTasts_returnsCorrectStringForEachIndex
[       OK ] ToDoTest.getTask_withThreeTasts_returnsCorrectStringForEachIndex (0 ms)
[----------] 4 tests from ToDoTest (0 ms total)
[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 4 tests.
[100%] Built target ToDoTest
```

It still works, just it's more automatic now.

Chapter 6: Realistically Getting a Boost

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.

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

```
#include <iostream>
  using std::cerr;
  using std::cout;
  using std::endl;
#include <string>
  using std::string;

#include <boost/program_options.hpp>
  namespace po = boost::program_options;

#include "ToDoCore/ToDo.h"
  using ToDoCore::ToDo;

int main(
    int    argc,
    char** argv
)
{
    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";
```

```
42          cout << "Usage:"                                << "\n";
43          cout << "   " << argv[0] << " [options]"        << "\n";
44          cout                                            << "\n";
45          cout << desc                                    << "\n";
46
47          if (parseError)
48          {
49              return 64;
50          }
51          else
52          {
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.**

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

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.")
```

`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_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
```