← CMake Tutorial – Chapter 1: Getting Started

CMake Tutorial – Chapter 3: GUI Tool →

CMake Tutorial – Chapter 2: IDE Integration

Posted on 2013-03-28 by John Lamp

Contents

Introduction

Now that we are familiar with CMake I will make good on CMake's promise of flexibility. I said before

Archives
March 2015
July 2013
May 2013
March 2013

Categories
CMake Tutorial

Meta
Log in
Entries RSS
Comments RSS
WordPress.org

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:
Source

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`

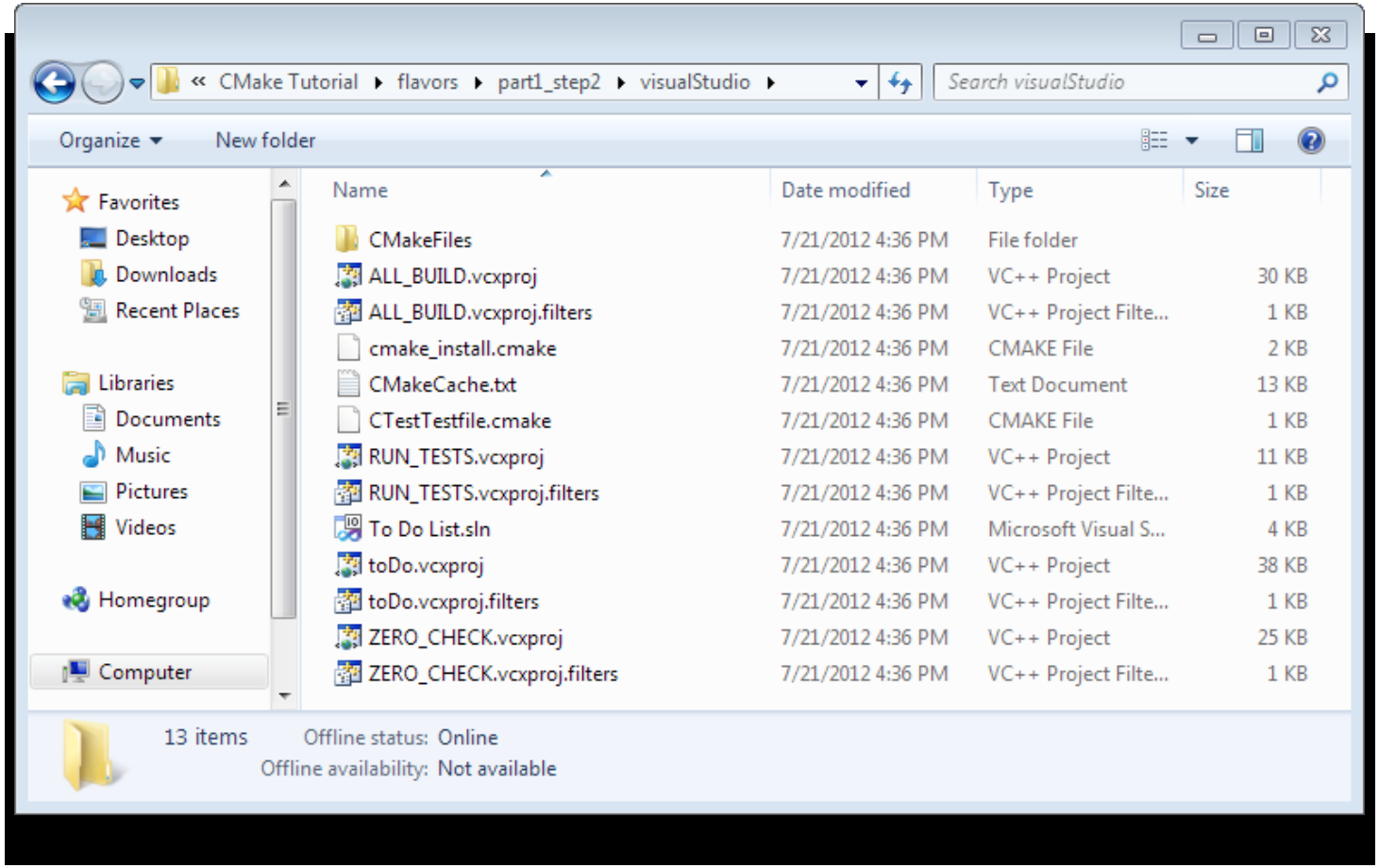
Generating a Visual Studio solution is simple, we just have to use a Visual Studio generator when we invoke CMake.

```
 > 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_s
```

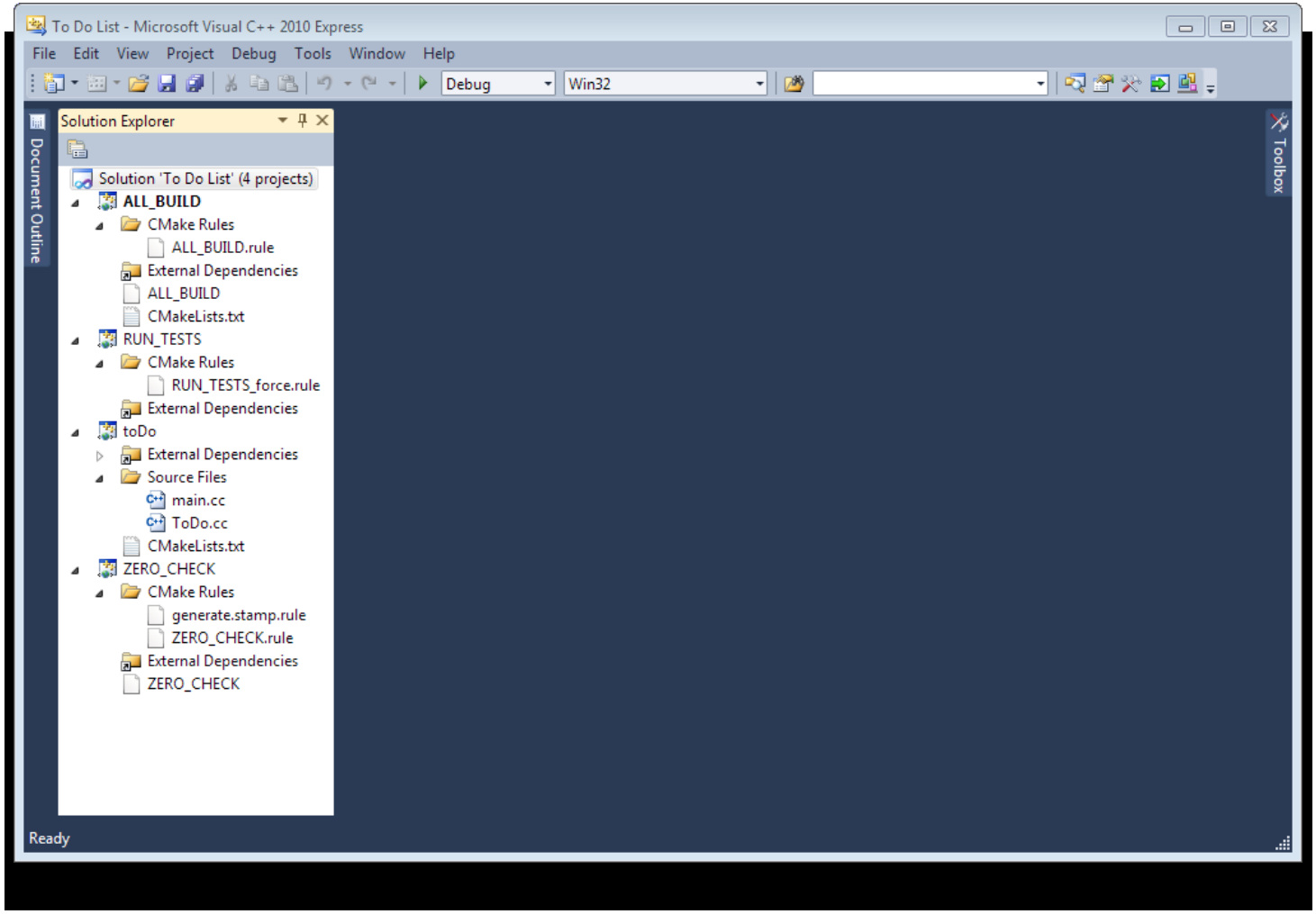
It is important to note that there are different generators for different versions of Visual Studio, so you will have to make sure that you chose the generator most appropriate for your Visual Studio install. CMake's output is actually a lot shorter than when we first ran it. You will notice that CMake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

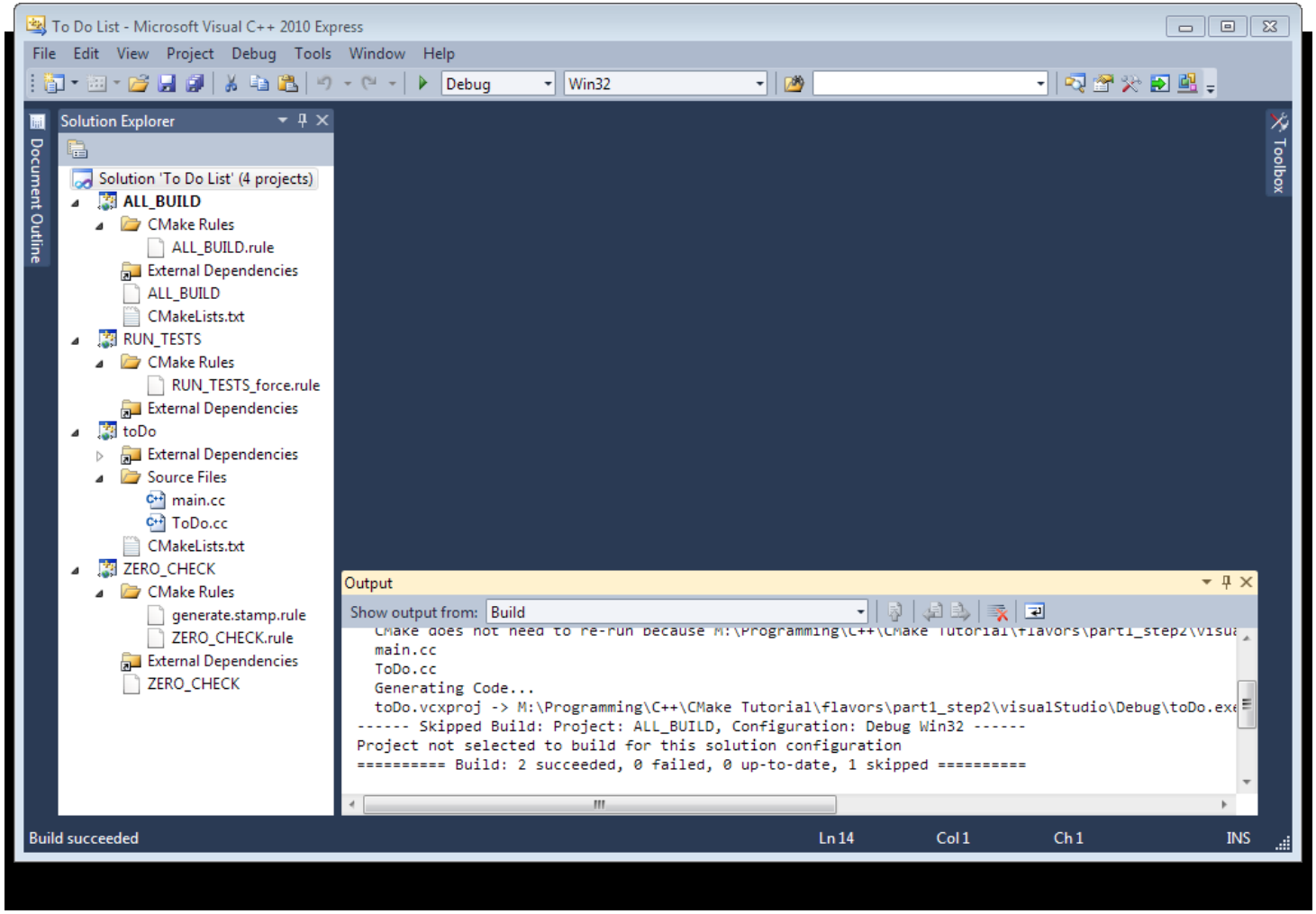
This is the little command line tool we are building. It corresponds to the `add_executable` command we have in our `CMakeLists.txt`.

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.

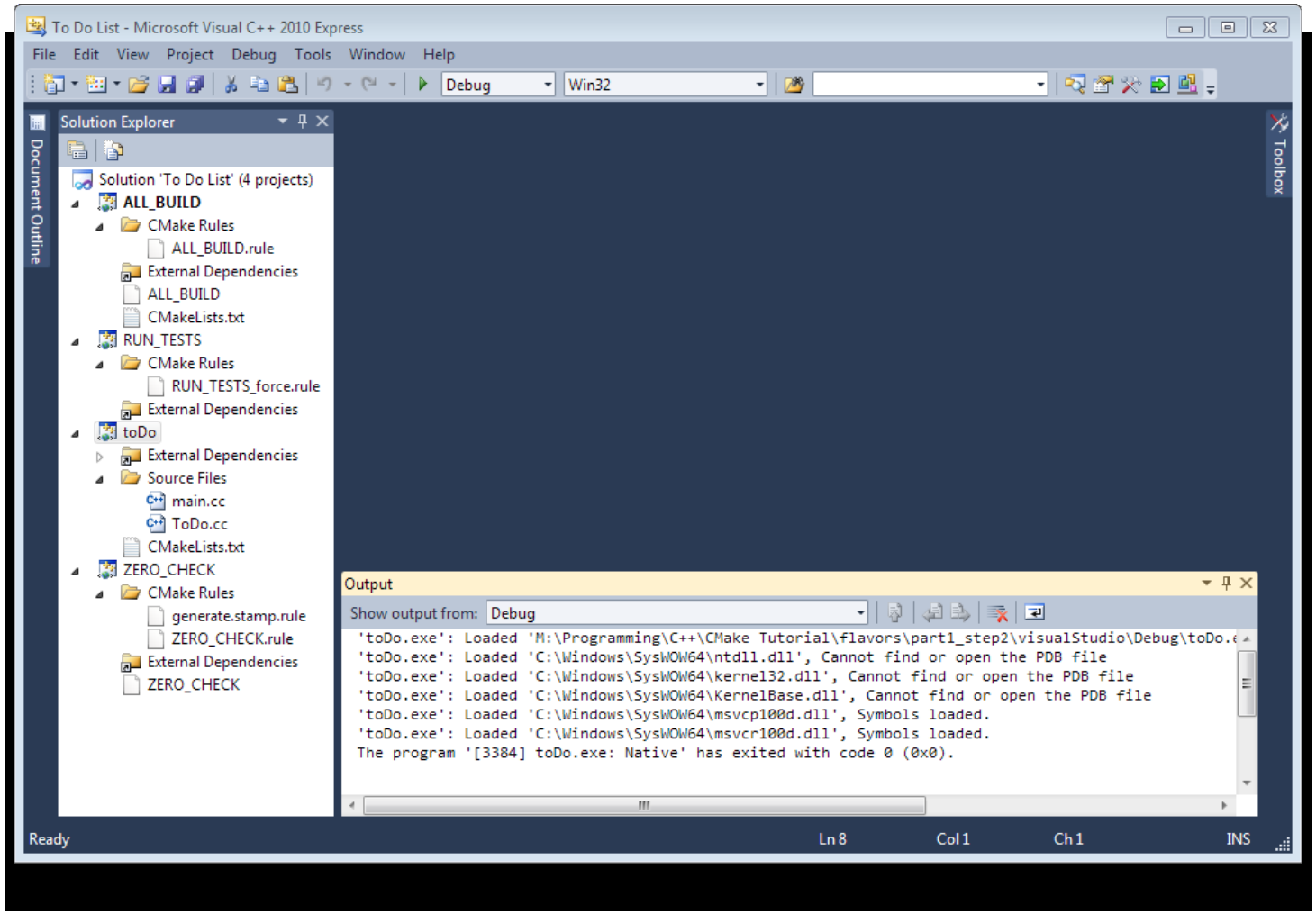
Let's try and build and see what happens.



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.
Project "M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\ALL_BUILD.
PrepareForBuild:
  Creating directory "Win32\Debug\ZERO_CHECK\".
  Creating directory "Debug\".
InitializeBuildStatus:
  Creating "Win32\Debug\ZERO_CHECK\ZERO_CHECK.unsuccessfulbuild" because "AlwaysCreate
CustomBuild:
  Checking Build System
  CMake does not need to re-run because M:/Programming/C++/CMake Tutorial/flavors/part
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\visualStu
Project "M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\ALL_BUILD.
PrepareForBuild:
  Creating directory "toDo.dir\Debug\".
InitializeBuildStatus:
  Creating "toDo.dir\Debug\toDo.unsuccessfulbuild" because "AlwaysCreate" was specifie
CustomBuild:
```

```
  Building Custom Rule M:/Programming/C++/CMake Tutorial/flavors/part1_step2/CMakeList
  CMake does not need to re-run because M:\Programming\C++\CMake Tutorial\flavors\part
ClCompile:
  C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\CL.exe /c /Zi /nologo /W3
cl : Command line warning D9035: option 'GX' has been deprecated and will be removed i
cl : Command line warning D9036: use 'EHsc' instead of 'GX' [M:\Programming\C++\CMake
cl : Command line warning D9035: option 'GZ' has been deprecated and will be removed i
cl : Command line warning D9036: use 'RTC1' instead of 'GZ' [M:\Programming\C++\CMake
  main.cc
  ToDo.cc
  Generating Code...
ManifestResourceCompile:
  C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\rc.exe /nologo /fo"toDo.dir\
Link:
  C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\link.exe /ERRORREPORT:QUE
  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
  C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\rc.exe /nologo /fo"toDo.dir\
LinkEmbedManifest:
  C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\link.exe /ERRORREPORT:QUE
  toDo.dir\Debug\main.obj
  toDo.dir\Debug\ToDo.obj  /machine:X86 /debug
  toDo.vcxproj -> M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\D
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\visualStu
PrepareForBuild:
  Creating directory "Win32\Debug\ALL_BUILD\".
InitializeBuildStatus:
  Creating "Win32\Debug\ALL_BUILD\ALL_BUILD.unsuccessfulbuild" because "AlwaysCreate"
CustomBuild:
  Building Custom Rule M:/Programming/C++/CMake Tutorial/flavors/part1_step2/CMakeList
```

```
  CMake does not need to re-run because M:\Programming\C++\CMake Tutorial\flavors\part
  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\visualStu
Build succeeded.
"M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\ALL_BUILD.vcxproj"
"M:\Programming\C++\CMake Tutorial\flavors\part1_step2\visualStudio\toDo.vcxproj" (def
(ClCompile target) ->
  cl : Command line warning D9035: option 'GX' has been deprecated and will be removed
  cl : Command line warning D9036: use 'EHsc' instead of 'GX' [M:\Programming\C++\CMak
  cl : Command line warning D9035: option 'GZ' has been deprecated and will be removed
  cl : Command line warning D9036: use 'RTC1' instead of 'GZ' [M:\Programming\C++\CMak
    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

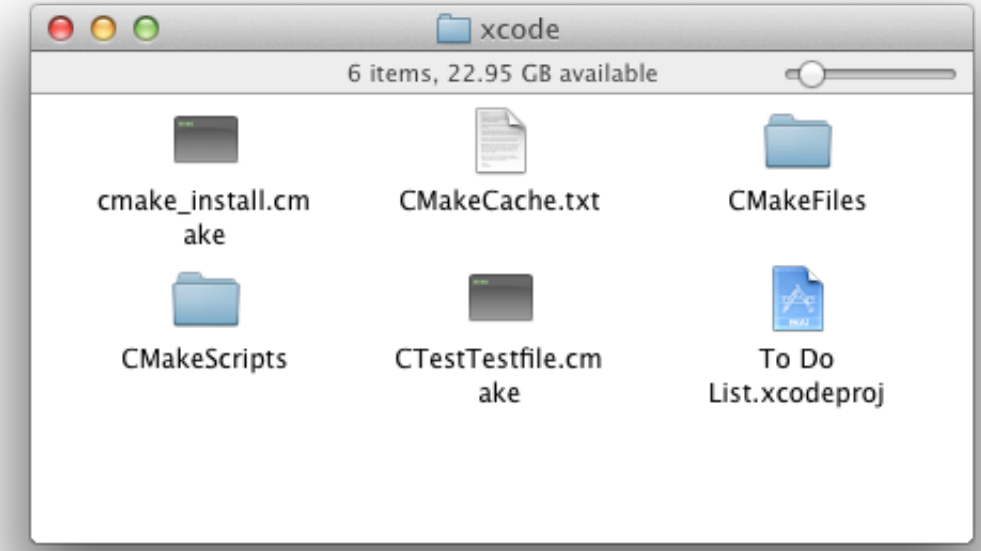
Mac OS X

Xcode Version 4.1 Build 4B110 was used.

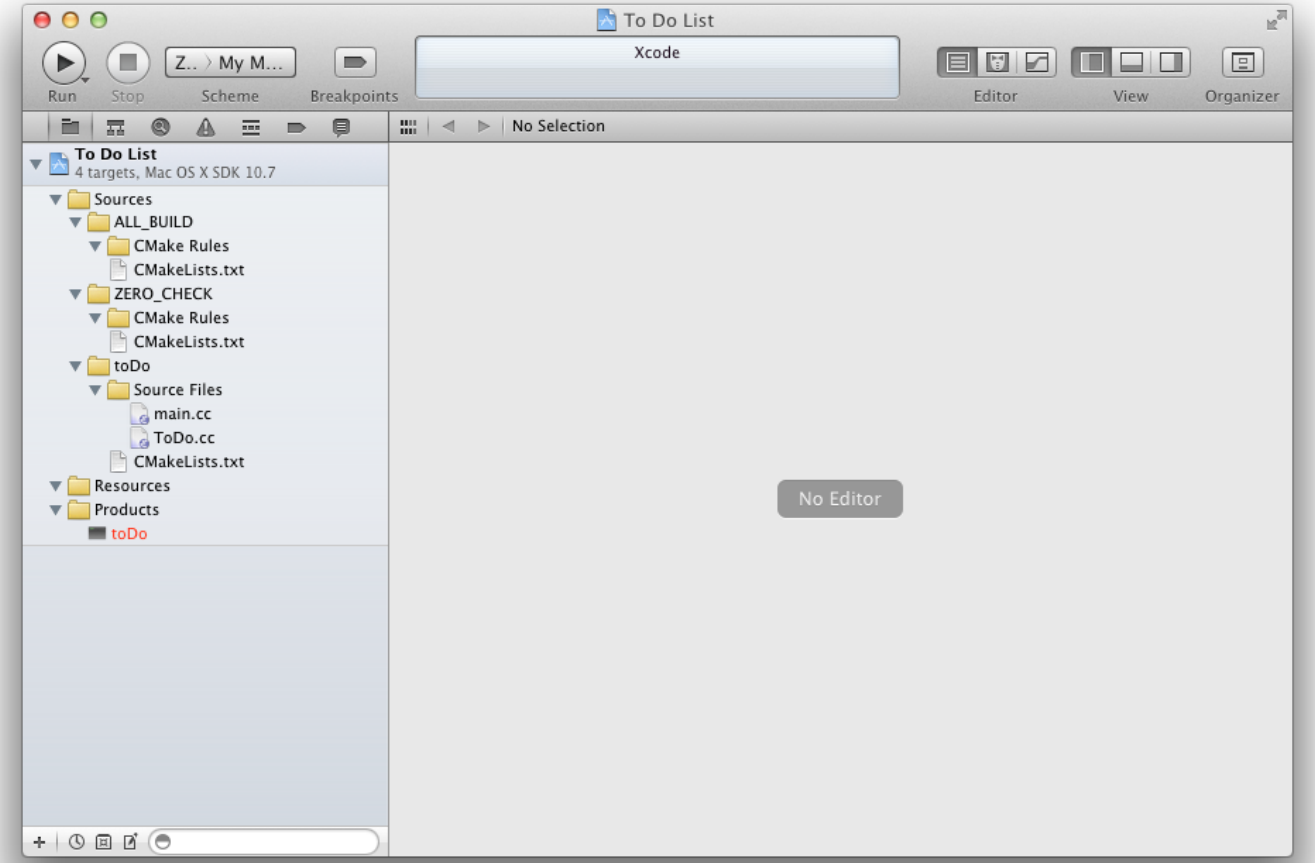
Generating an Xcode project is very similar to generating any other project:

```
 > 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
```

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:

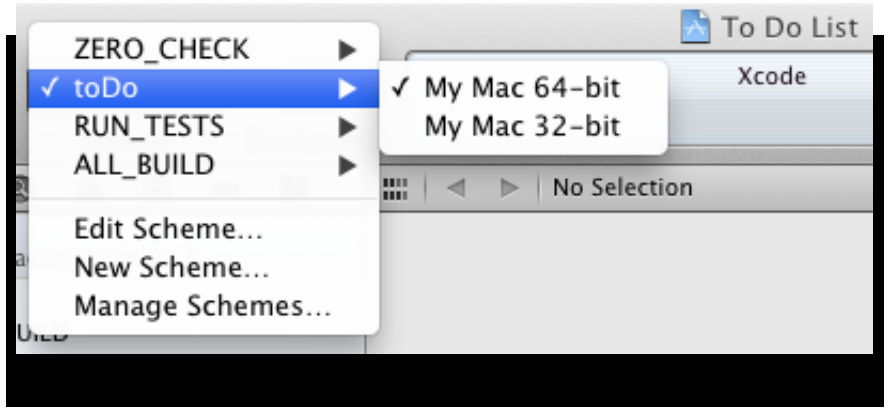


Most of these files will look familiar if you had looked at what files CMake generated before. The most important file is `To Do List.xcodeproj`. Note that the project file is named after the project command in `CMakeLists.txt`. If spaces in file names cause trouble in your environment then you will 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.

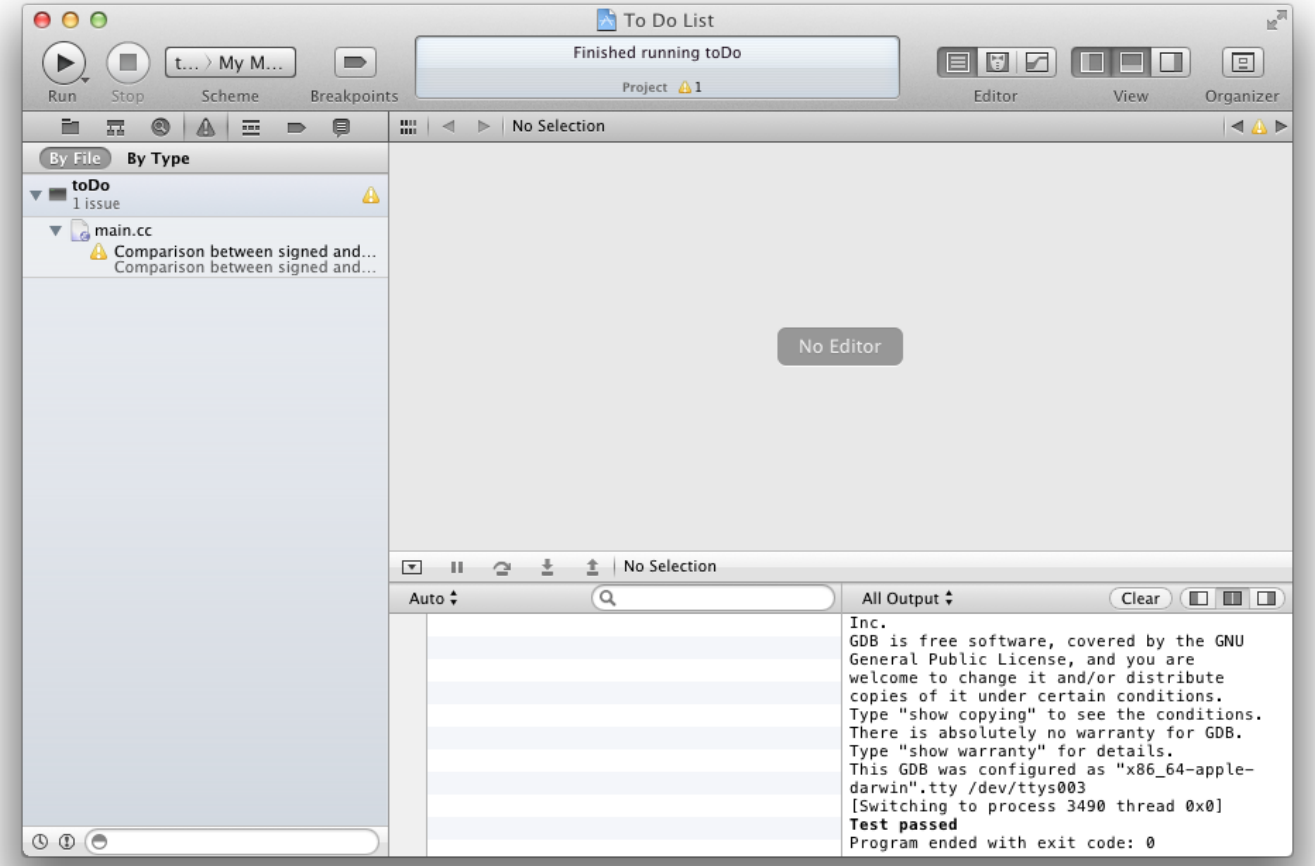
RUN_TESTS

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 CONFIGURA
Check dependencies
PhaseScriptExecution "CMake Rules" "xcode/To Do List.build/Debug/ZERO_CHECK.build/Scri
    cd "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2"
    /bin/sh -c "\"/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step
echo ""
make -f /Volumes/Documents/Programming/C++/CMake\ Tutorial/flavors/part1_step2/xcode/C
make[1]: `CMakeFiles/cmake.check_cache' is up to date.
=== BUILD NATIVE TARGET toDo OF PROJECT To Do List WITH THE DEFAULT CONFIGURATION (Deb
Check dependencies
CompileC "xcode/To Do List.build/Debug/toDo.build/Objects-normal/x86_64/ToDo.o" ToDo.c
    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-
CompileC "xcode/To Do List.build/Debug/toDo.build/Objects-normal/x86_64/main.o" main.c
    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-
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/main.cc: In func
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/main.cc:34:   in
/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2/main.cc:58: warn
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.
PhaseScriptExecution "CMake PostBuild Rules" "xcode/To Do List.build/Debug/toDo.build/
    cd "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2"
    /bin/sh -c "\"/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step
echo "Depend check for xcode"
Depend check for xcode
cd /Volumes/Documents/Programming/C++/CMake\ Tutorial/flavors/part1_step2/xcode && mak
make[1]: Nothing to be done for `PostBuild.toDo.Debug'.
=== BUILD AGGREGATE TARGET ALL_BUILD OF PROJECT To Do List WITH THE DEFAULT CONFIGURAT
Check dependencies
PhaseScriptExecution "CMake Rules" "xcode/To Do List.build/Debug/ALL_BUILD.build/Scrip
    cd "/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step2"
    /bin/sh -c "\"/Volumes/Documents/Programming/C++/CMake Tutorial/flavors/part1_step
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

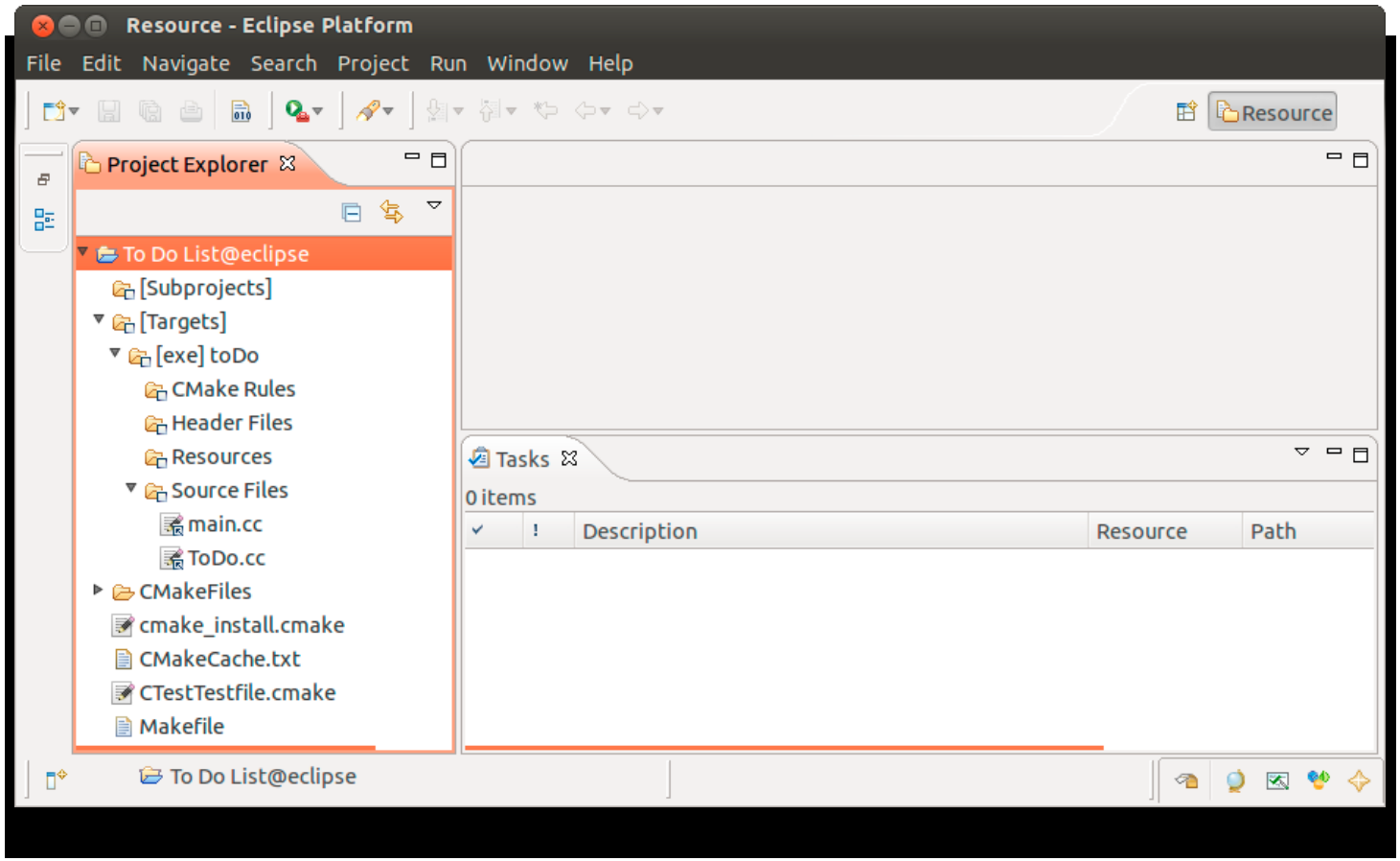
Eclipse Indigo Version 3.7.2 Build I20110613-1736 was used.

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_E
-- 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
```

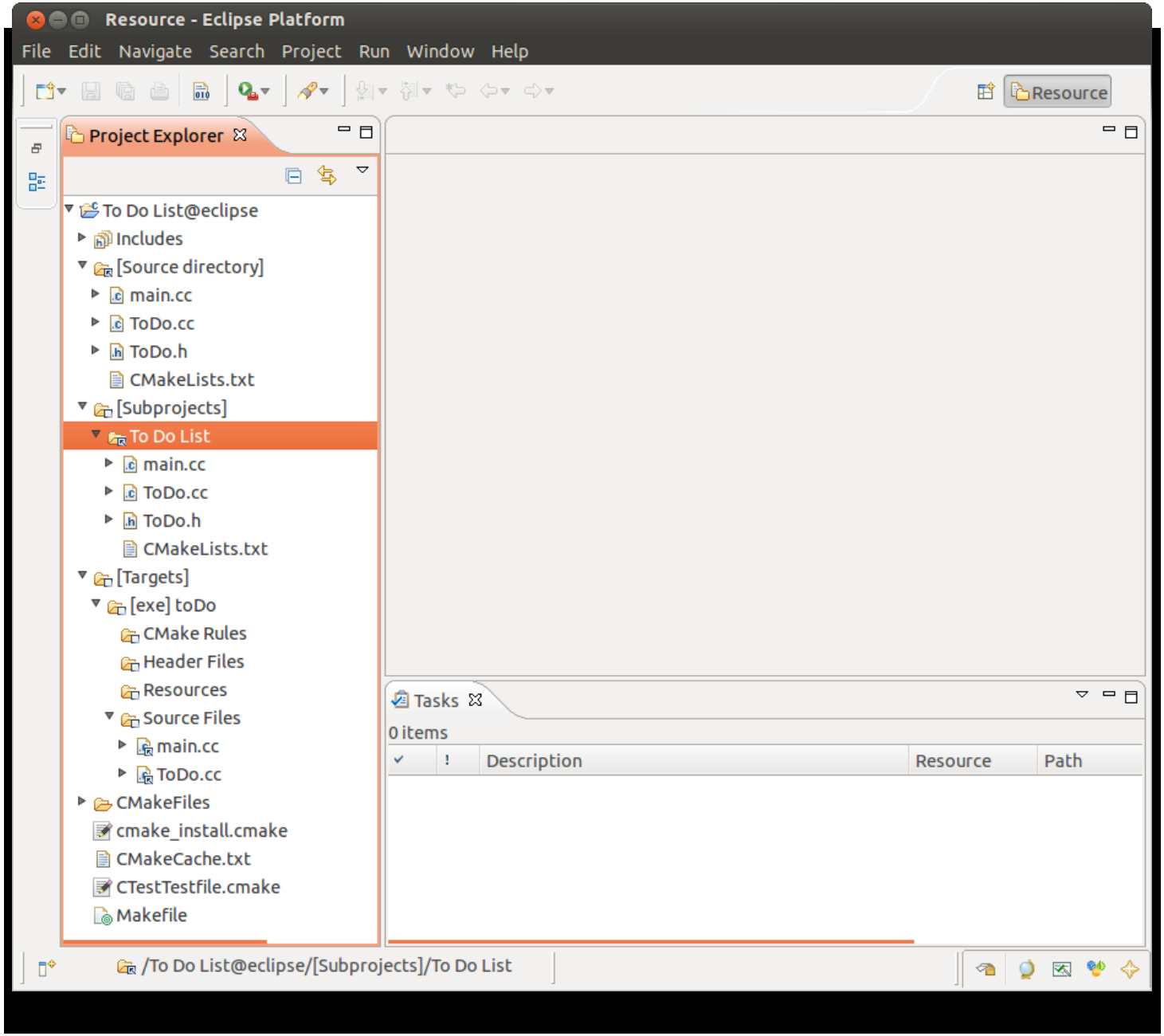
Well perhaps it isn't actually that simple. CMake warns us that Eclipse doesn't like the build directory being a subdirectory of the source directory. As you can see it created the `.project` and `.cproject` files required by Eclipse CDT.



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.

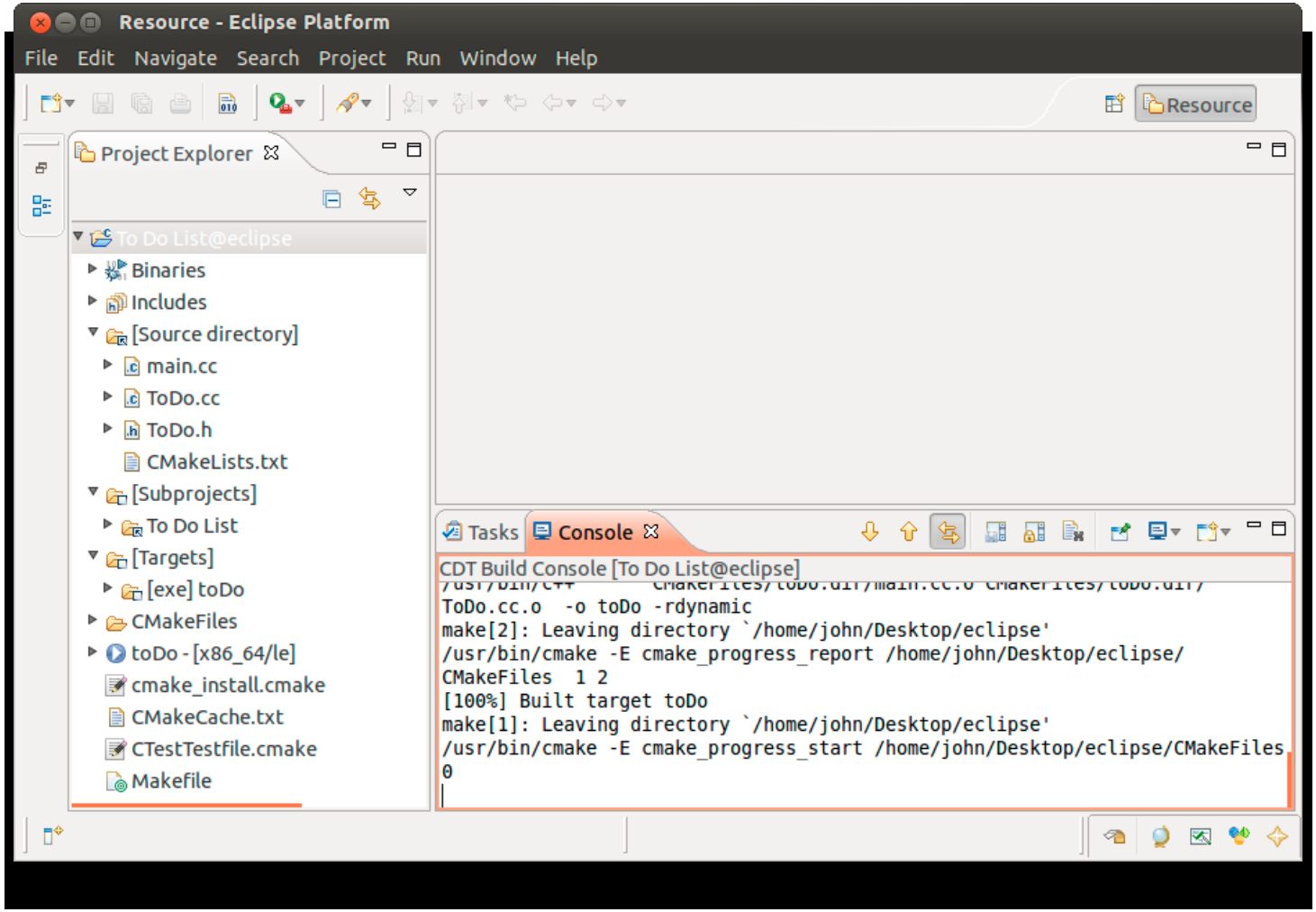
```
 > cd ..
 > mkdir eclipse
 > cd eclipse
 > cmake -G "Eclipse CDT4 - Unix Makefiles" ../part1_step2/
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Could not determine Eclipse version, assuming at least 3.6 (Helios). Adjust CMAKE_E
-- Check for working C compiler: /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.


Resource - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Resource
Project Explorer
To Do List@eclipse
Includes
[Source directory]
main.cc
ToDo.cc
ToDo.h
CMakeLists.txt
[Subprojects]
To Do List
main.cc
ToDo.cc
ToDo.h
CMakeLists.txt
[Targets]
[exe] toDo
CMake Rules
Header Files
Resources
Source Files
main.cc
ToDo.cc
CMakeFiles
cmake_install.cmake
CMakeCache.txt
CTestTestfile.cmake
Makefile
Tasks
0 items
Description
Resource
Path
/To Do List@eclipse/[Subprojects]/To Do List

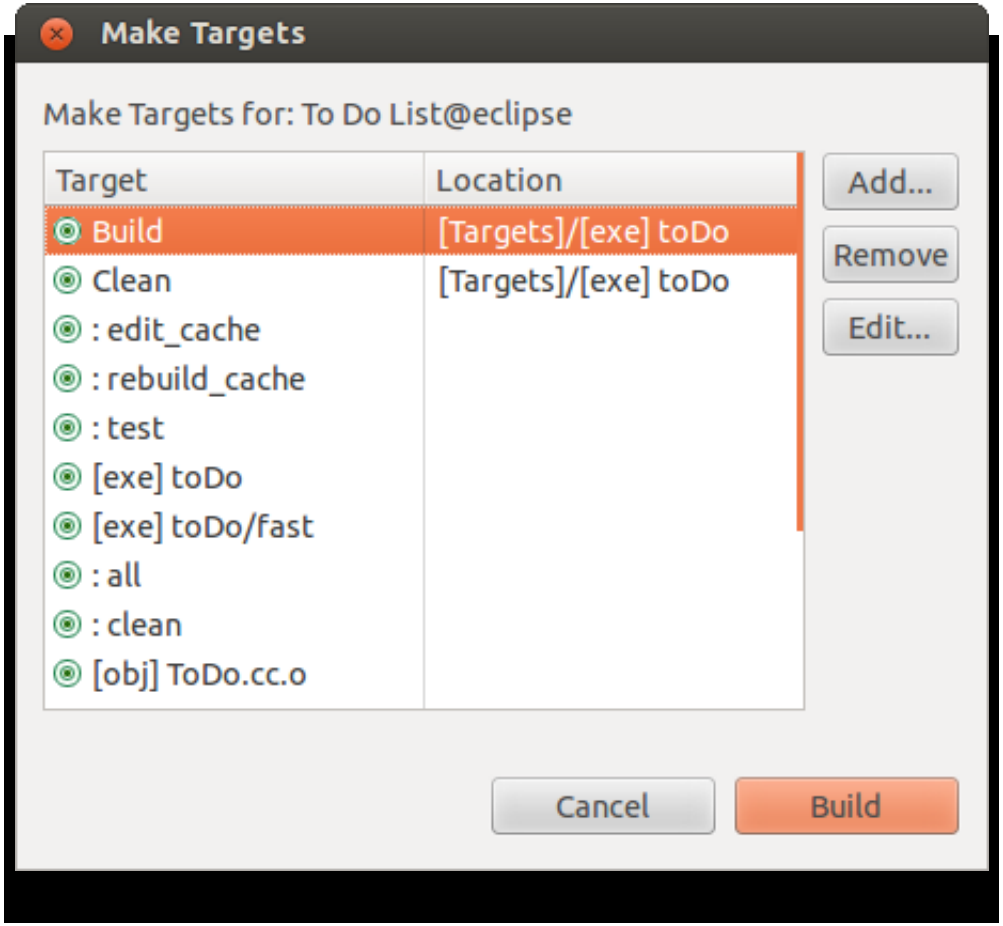
The project looks a lot better this time. If you are familiar with Eclipse you may know that it only supports one target per project whereas CMake supports many. In fact managing builds of complex source trees is one of CMake's strengths. These seem to be at odds with each other. If you looked closely 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.

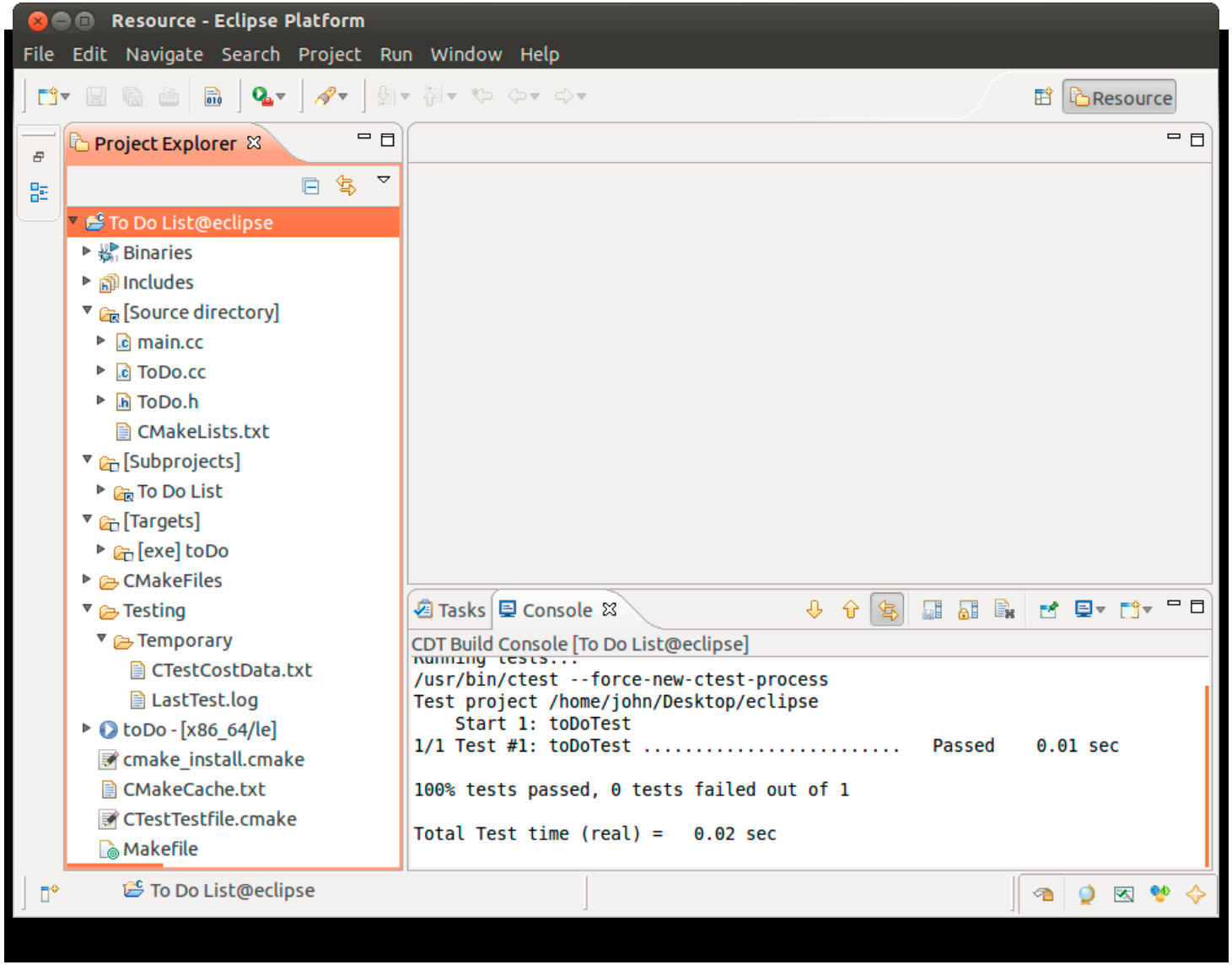


It still builds fine and as you can see Eclipse uses make to do the building. Conveniently the binary executable “toDo” is added to the project so it can easily be run or debugged from within Eclipse.

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)

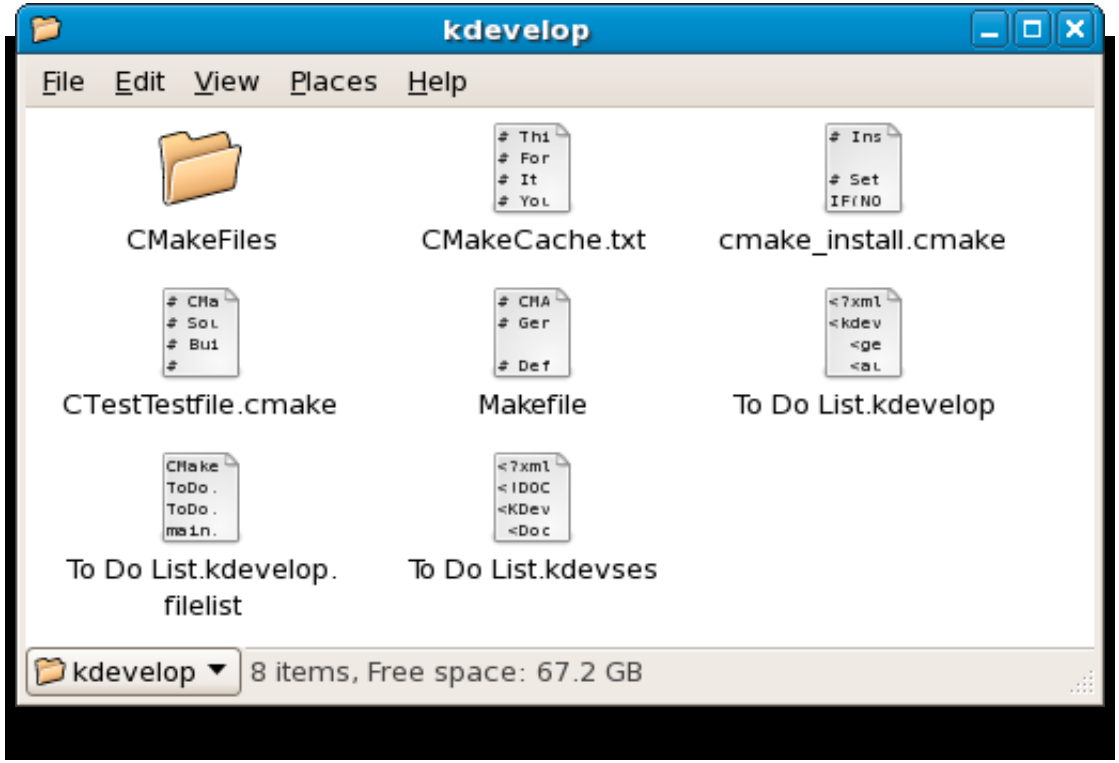
KDevelop Version 3.3.4 was used.

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.

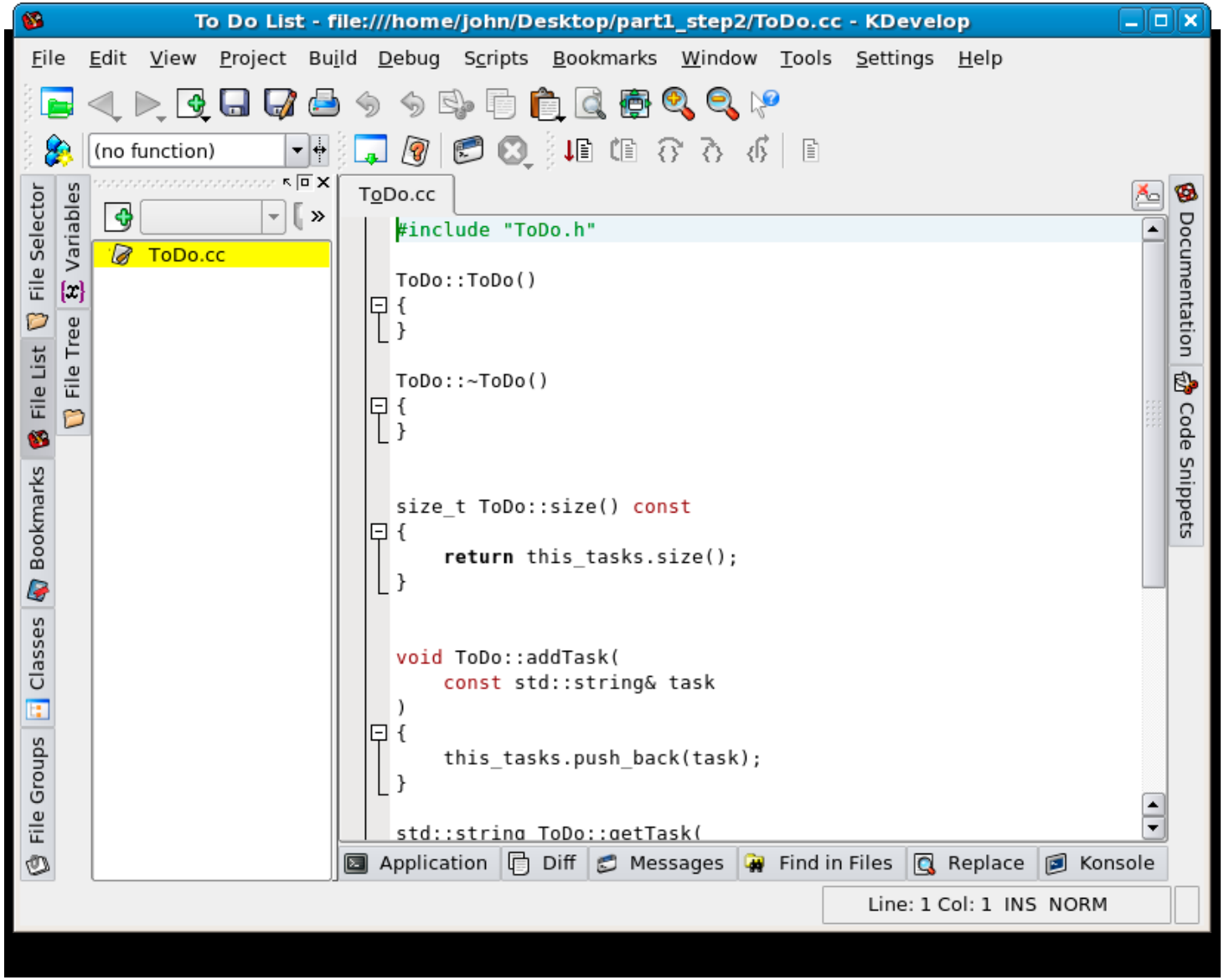
```
 > mkdir kdevelop
 > cd kdevelop
 > cmake -G "KDevelop3" ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler 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
```

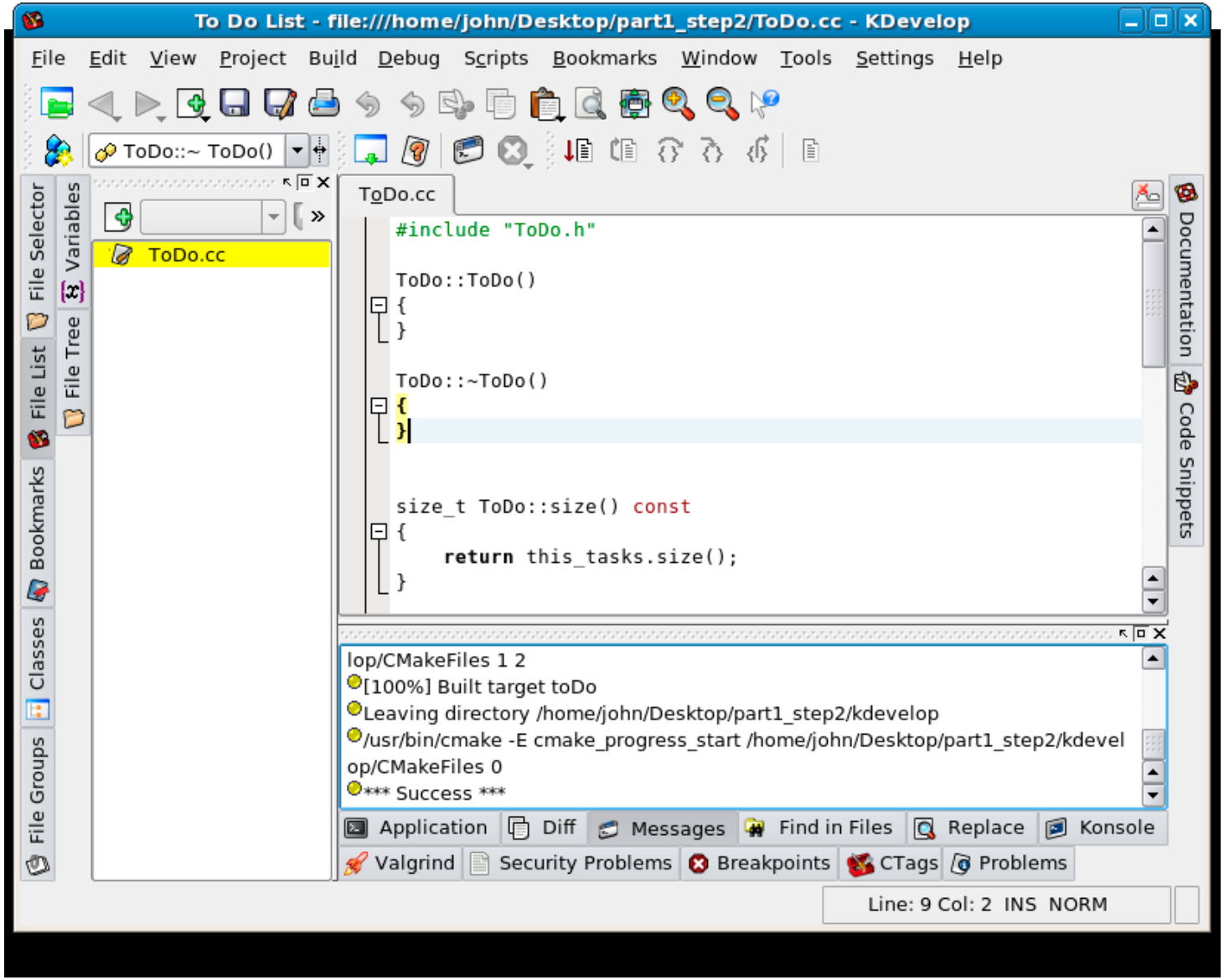
The output looks similar to the first time we ran it. It produces a few extra files for KDevelop, though.



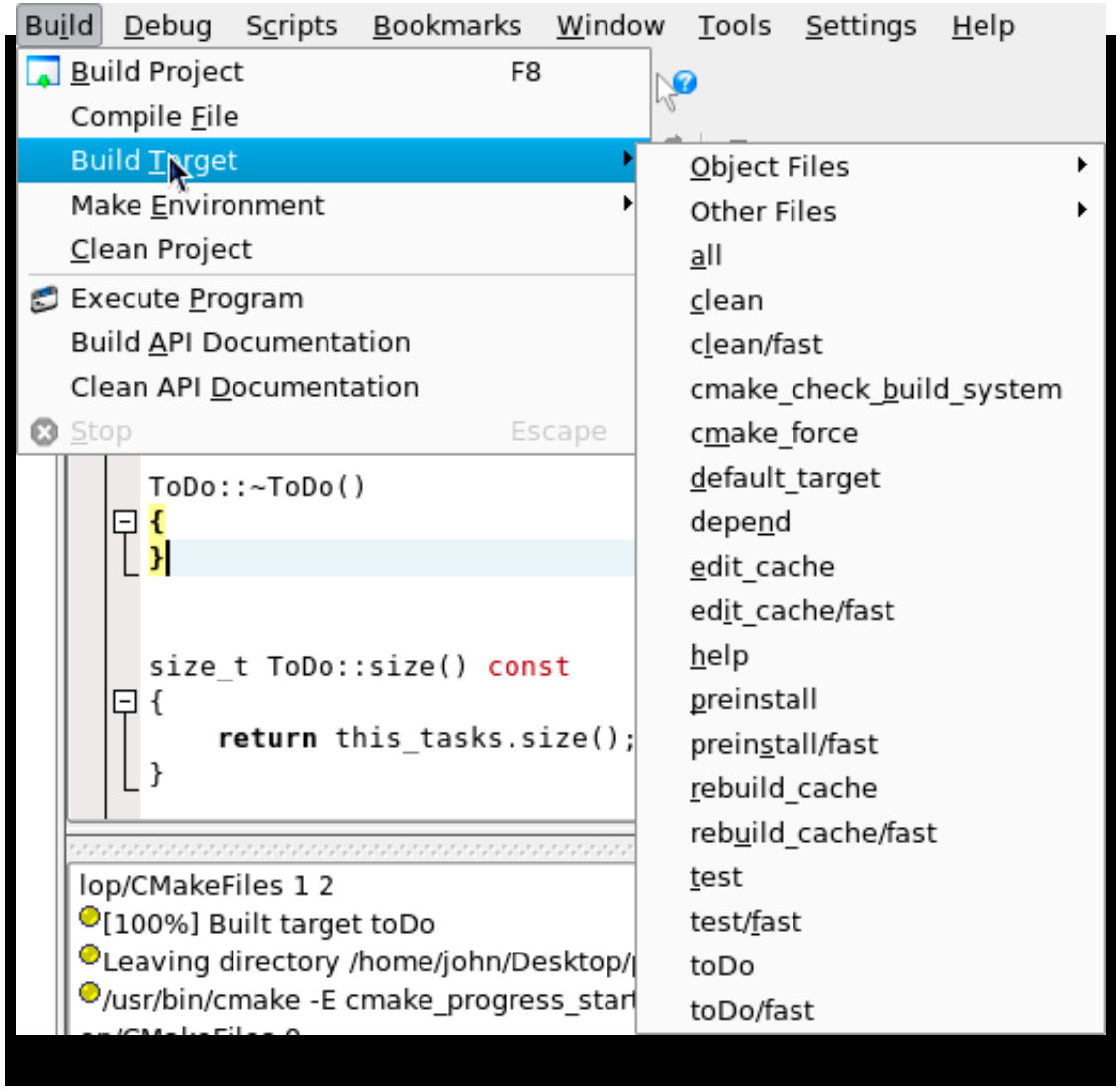
The KDevelop 3 project file, `To Do List.kdevelop` is the most important of the new files. You will notice that CMake still created a `Makefile`. KDevelop's Makefile support, however, is quite good. Let's see the project.



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.



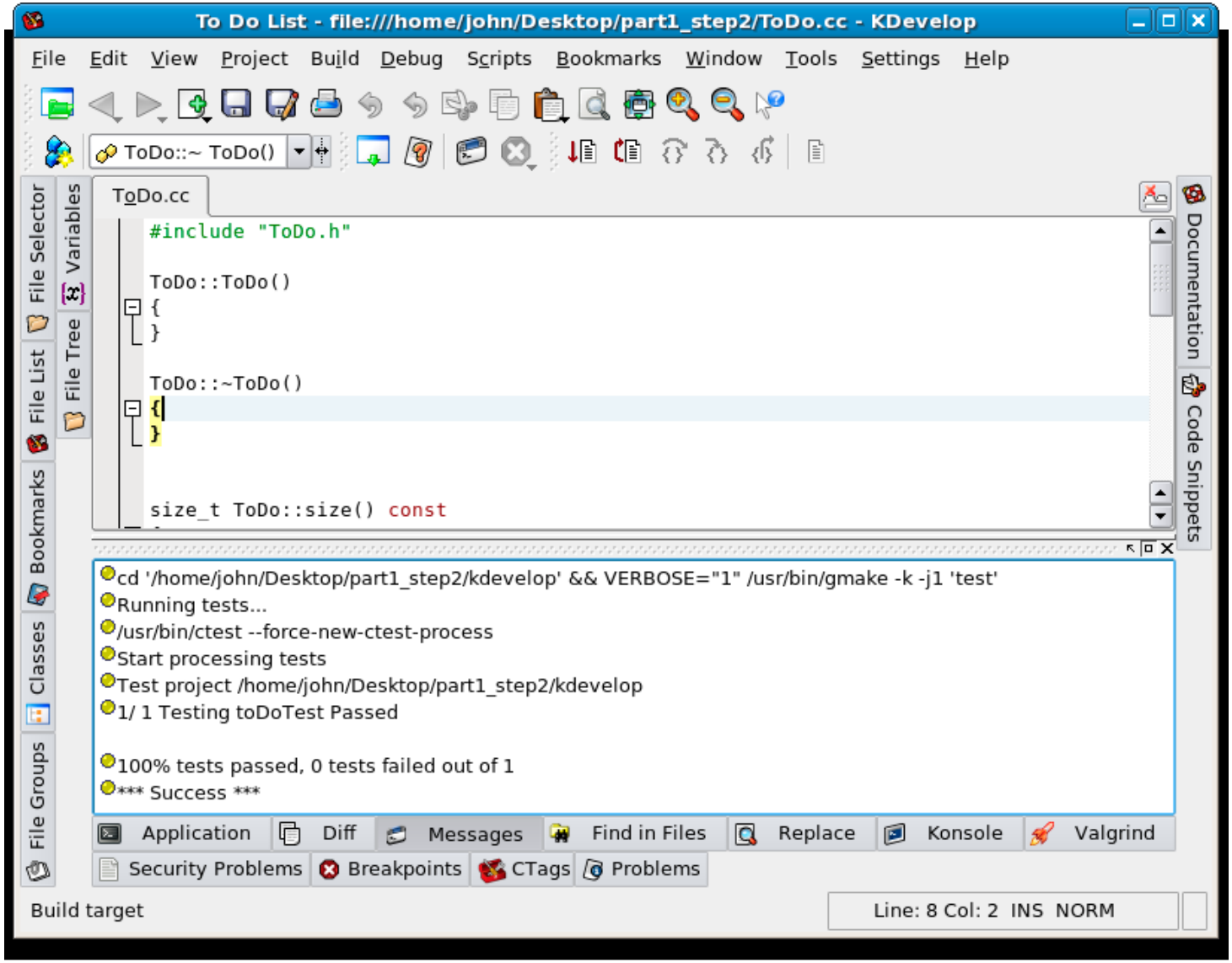
Of course it still builds. We are using the same `Makefile` as we originally did. The only difference this time is that KDevelop is running make for us. Thanks to KDevelop's Makefile support we can actually build any target we want.



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.

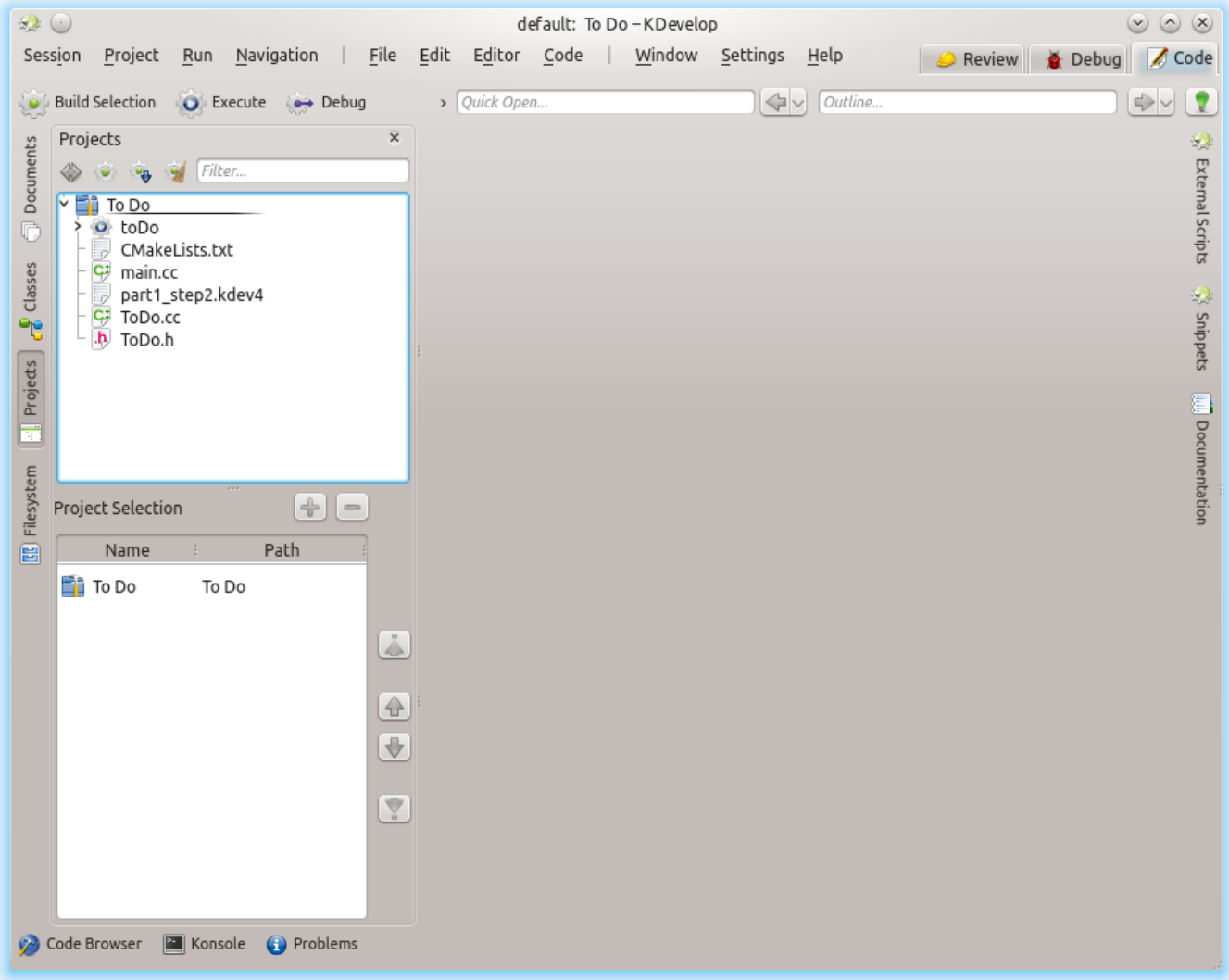
```
 > cd kdevelop
 > make
[ 50%] Building CXX object CMakeFiles/toDo.dir/main.cc.o
[100%] Building CXX object CMakeFiles/toDo.dir/ToDo.cc.o
Linking CXX executable toDo
[100%] Built target toDo
```

CMake Support (KDevelop4)

KDevelop Version 4.3.1 was used.

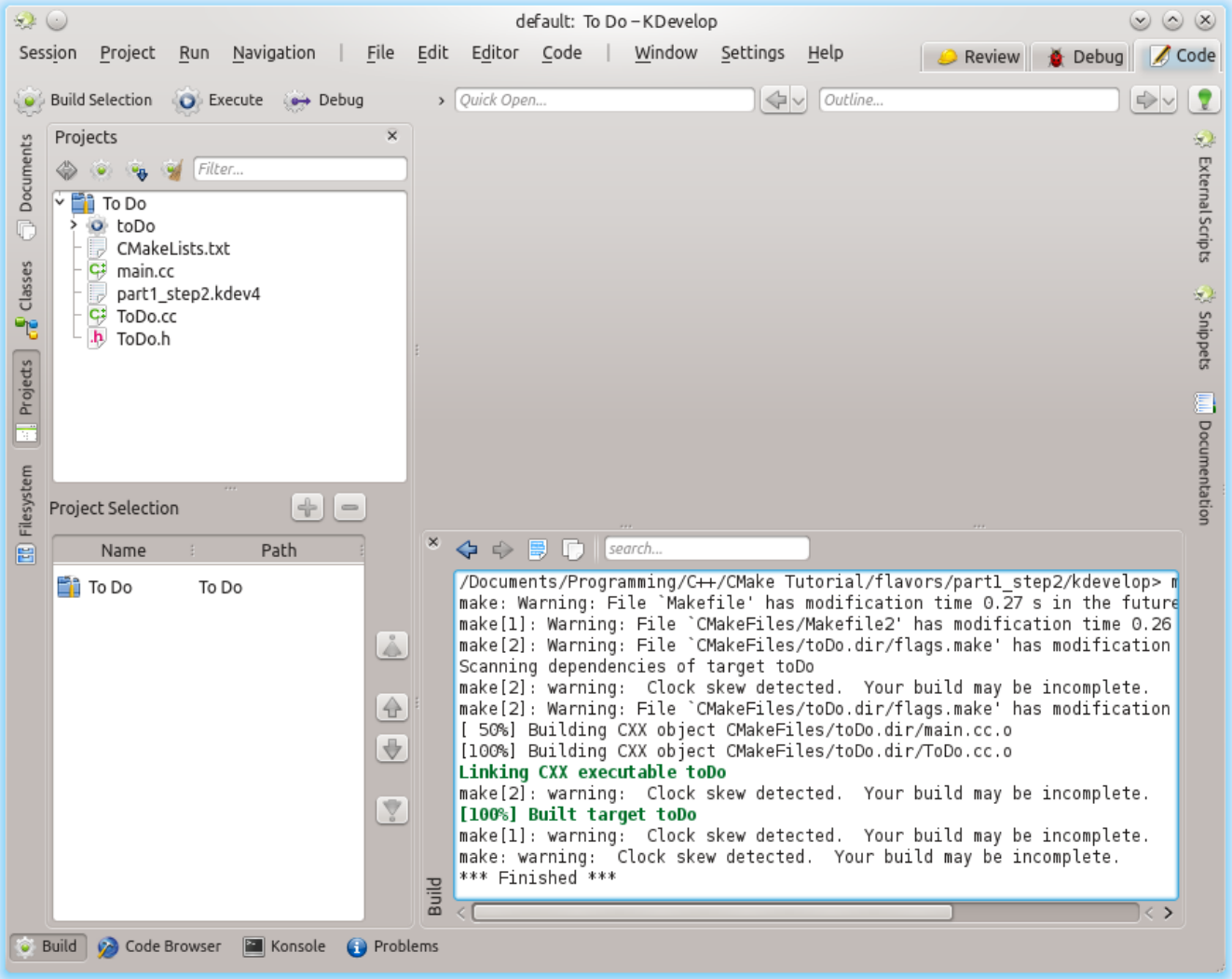
KDevelop 4 has built-in support for CMake projects. So rather than use a generator to make a new project file as was done in the previous examples we instead simply open the CMake project with KDevelop.

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.

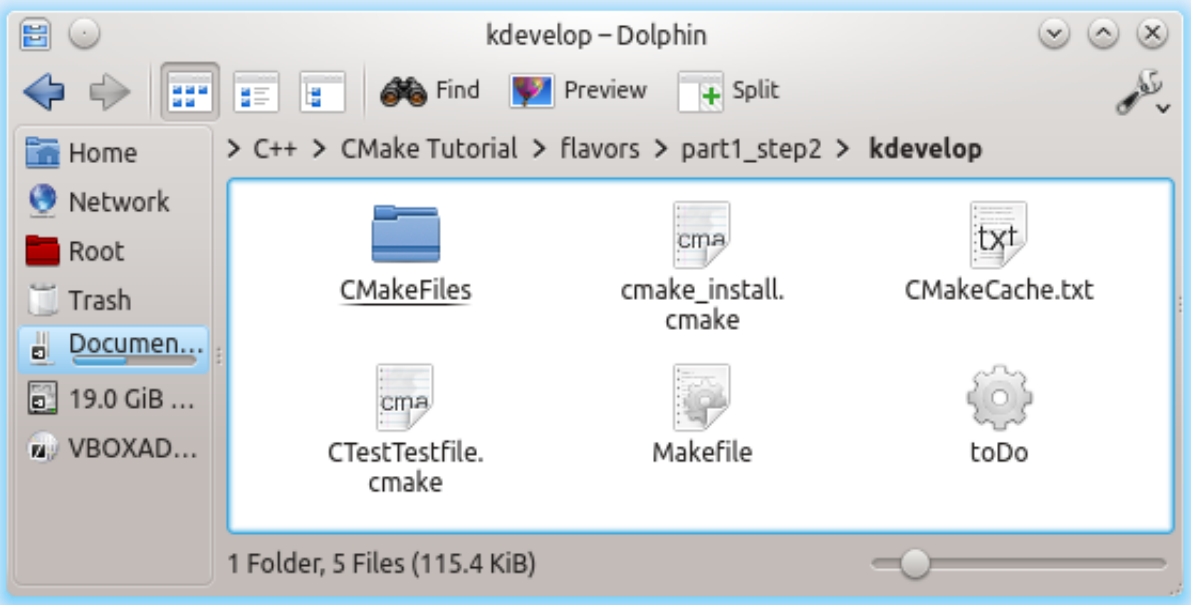


The file list shows all files that are actually in the project directory. Conveniently this include `ToDo.h`. However you may also notice a `kdev4` project file. While KDevelop4 supports CMake, including out of source builds, it does put a project file in your source directory. Although since it is only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

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

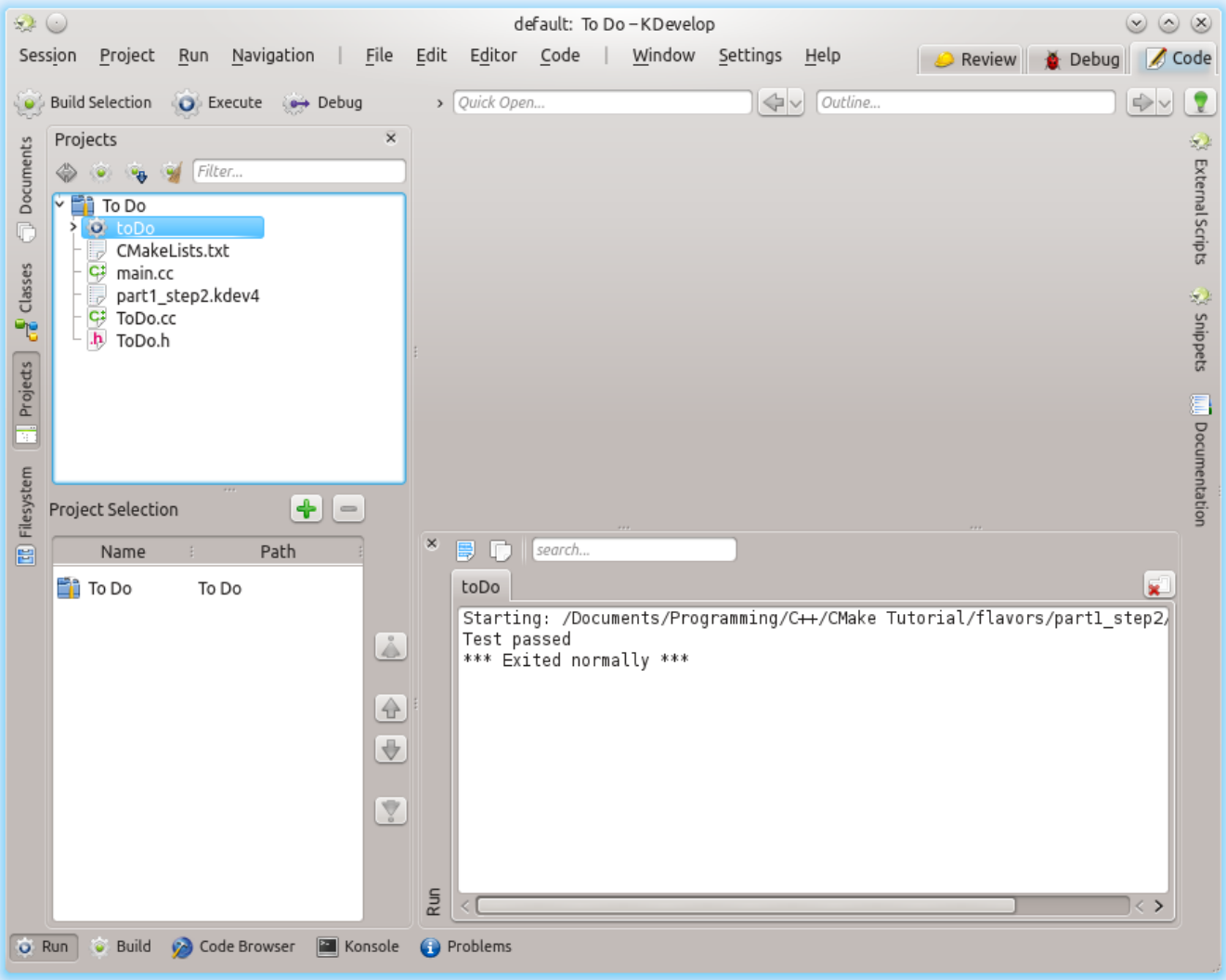


You will notice that KDevelop still uses make to build the project. The main difference here is that KDevelop also runs CMake for you. These are the files it created:



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
```

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

← CMake Tutorial – Chapter 1: Getting Started

CMake Tutorial – Chapter 3: GUI Tool →

7 thoughts on “CMake Tutorial – Chapter 2: IDE Integration”

2014-11-09 at 19:55:15

mayers said:

1-How do I run the RUN_TESTS from Visual Studio? In other example(not a VS example) you have shown running “make test” in a console will run all the specified test and create a Testing folder with the test run results.