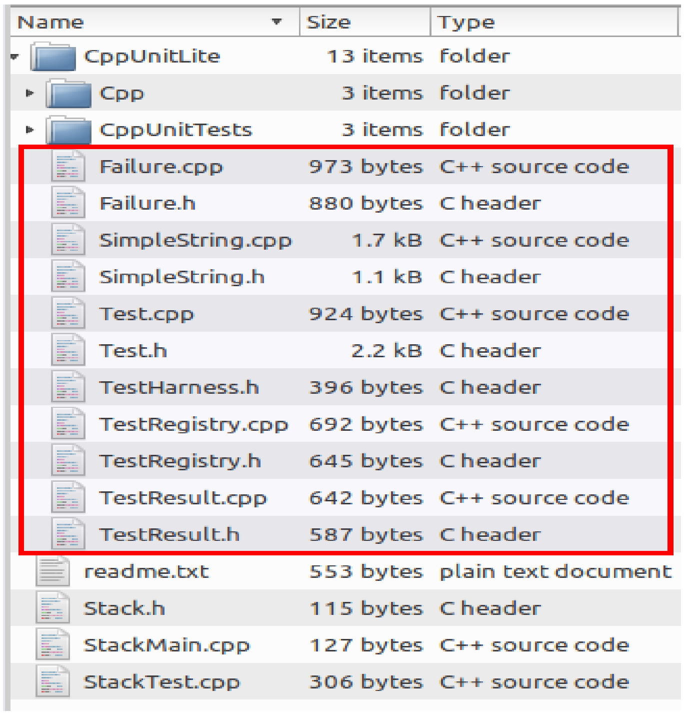
Building And Integrating CppUnitLite in Eclipse on Linux

.

If you are familiar with CppUnit, CppUnitLite is – as the website mentions – more barebones, lighter, and more portable as it avoids using some C++ features such as exceptions, and templates.

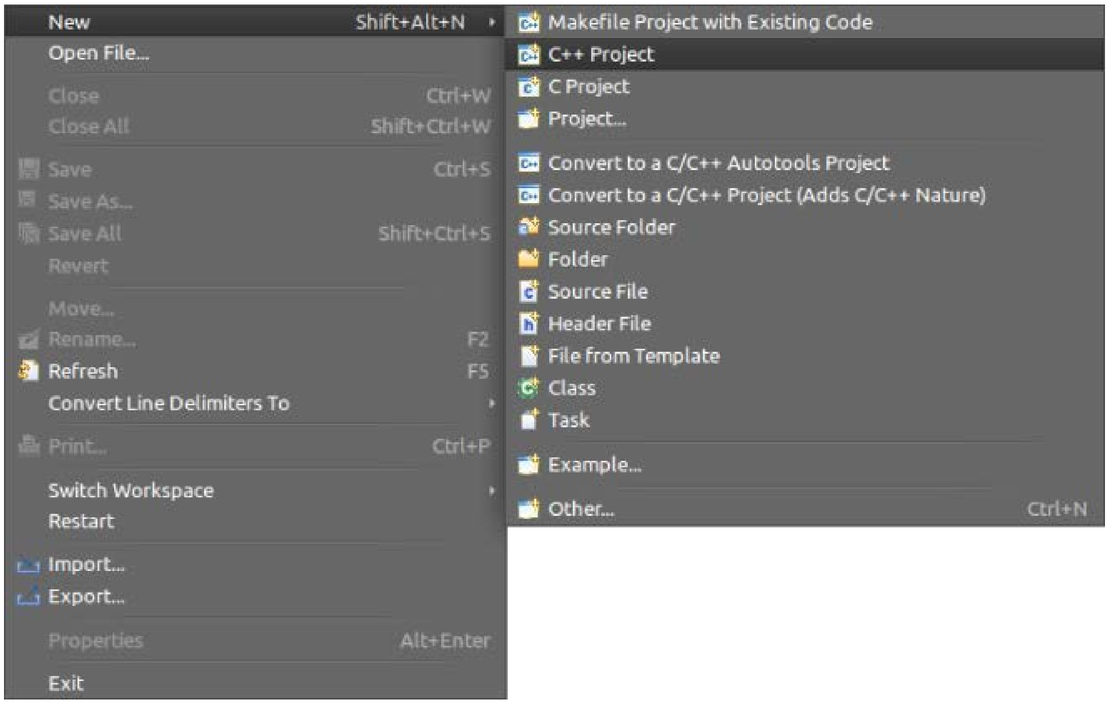
We will build it as a static library, so that we can then link into any of our projects that we would like to write unit tests for later.

First, we need to get the source code form the CppUnitLite website. Go there and get it. Once you do, extract the zip file. You'll find that it contains a lot of files and a couple of folders. We only need a sub-set of those, specifically the folders in the CppUnitLite sub-folder highlighted in the below screen shot.

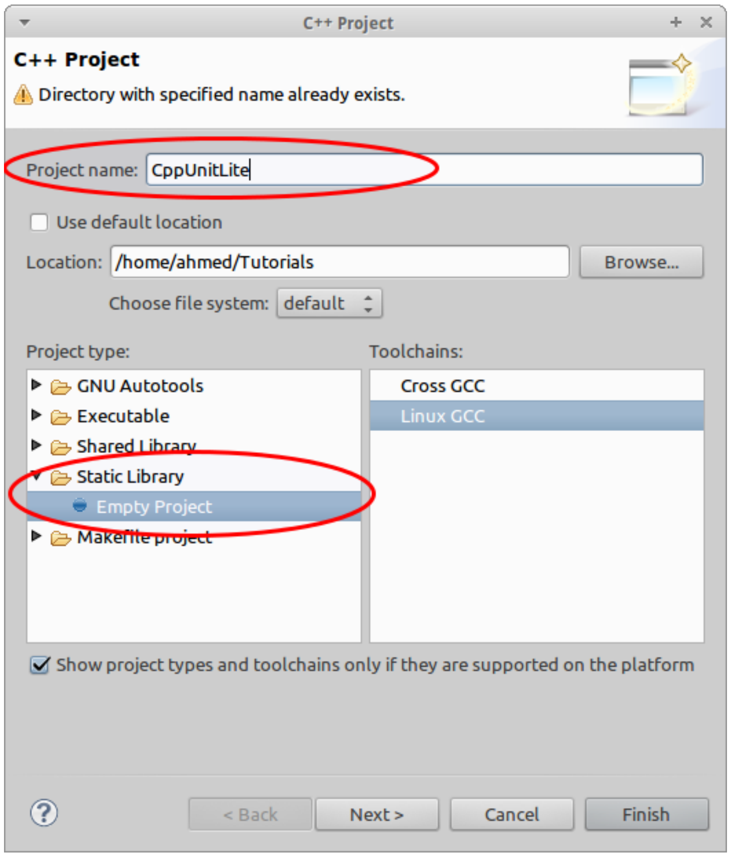


Name	Size	Type
CppUnitLite	13 items	folder
Cpp	3 items	folder
CppUnitTests	3 items	folder
Failure.cpp	973 bytes	C++ source code
Failure.h	880 bytes	C header
SimpleString.cpp	1.7 kB	C++ source code
SimpleString.h	1.1 kB	C header
Test.cpp	924 bytes	C++ source code
Test.h	2.2 kB	C header
TestHarness.h	396 bytes	C header
TestRegistry.cpp	692 bytes	C++ source code
TestRegistry.h	645 bytes	C header
TestResult.cpp	642 bytes	C++ source code
TestResult.h	587 bytes	C header
readme.txt	553 bytes	plain text document
Stack.h	115 bytes	C header
StackMain.cpp	127 bytes	C++ source code
StackTest.cpp	306 bytes	C++ source code

Now go ahead and open up Eclipse. Select File > New > C++ Project.



From the C++ Project Window, name your project **CppUnitLite**, and make sure you selected a **Static Library** project type. Now click on **Next** >.


C++ Project

C++ Project

Directory with specified name already exists.

Project name: CppUnitLite

Use default location

Location: /home/ahmed/Tutorials

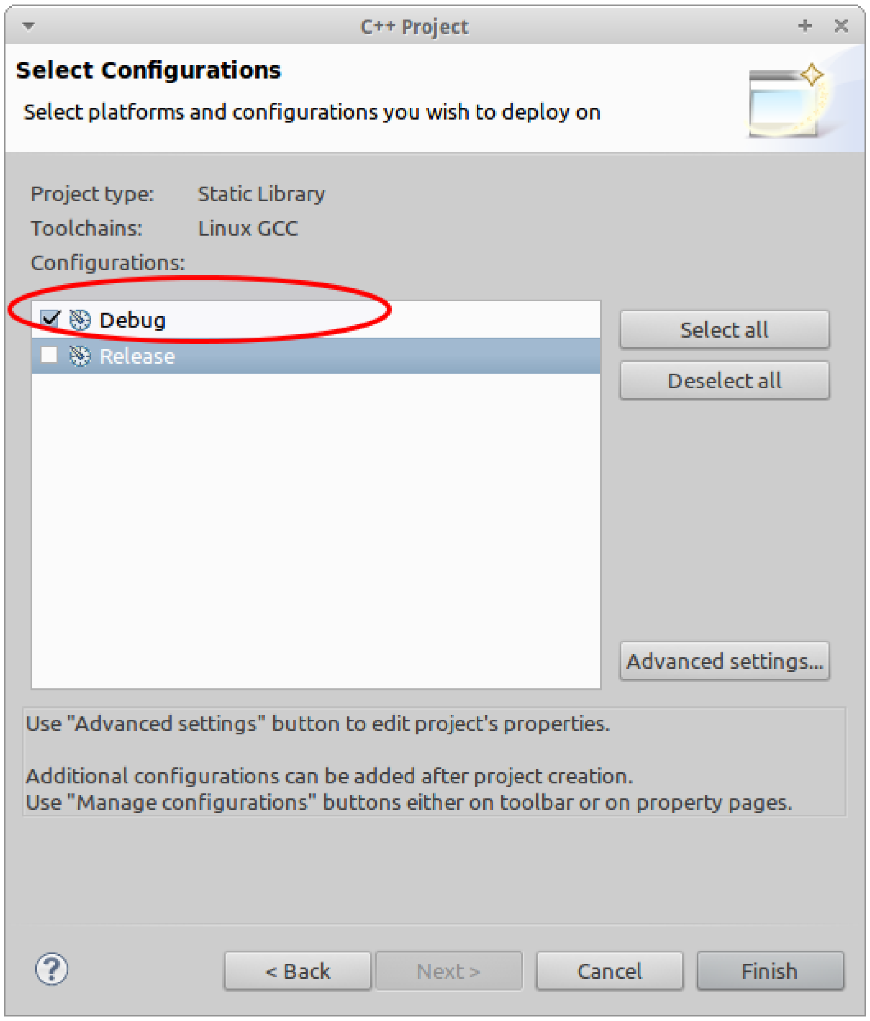
Choose file system: default

Project type:
- GNU Autotools
- Executable
- Shared Library
- Static Library
  - Empty Project
- Makefile project

Toolchains:
- Cross GCC
- Linux GCC

Show project types and toolchains only if they are supported on the platform

For the build configuration, you can just select **Debug** for now. Then click **Finish**.


C++ Project

Select Configurations

Select platforms and configurations you wish to deploy on

Project type: Static Library

Toolchains: Linux GCC

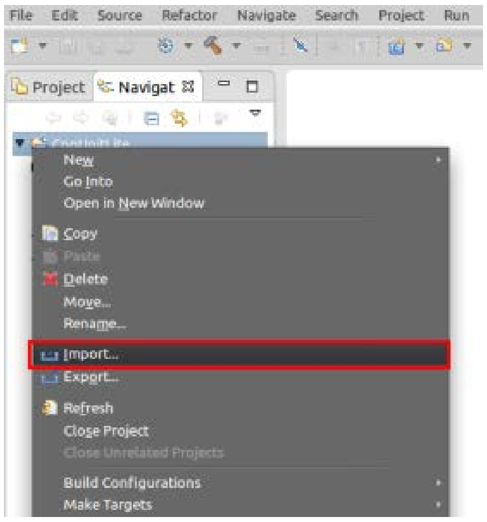
Configurations:

- [✓] Debug
- [ ] Release

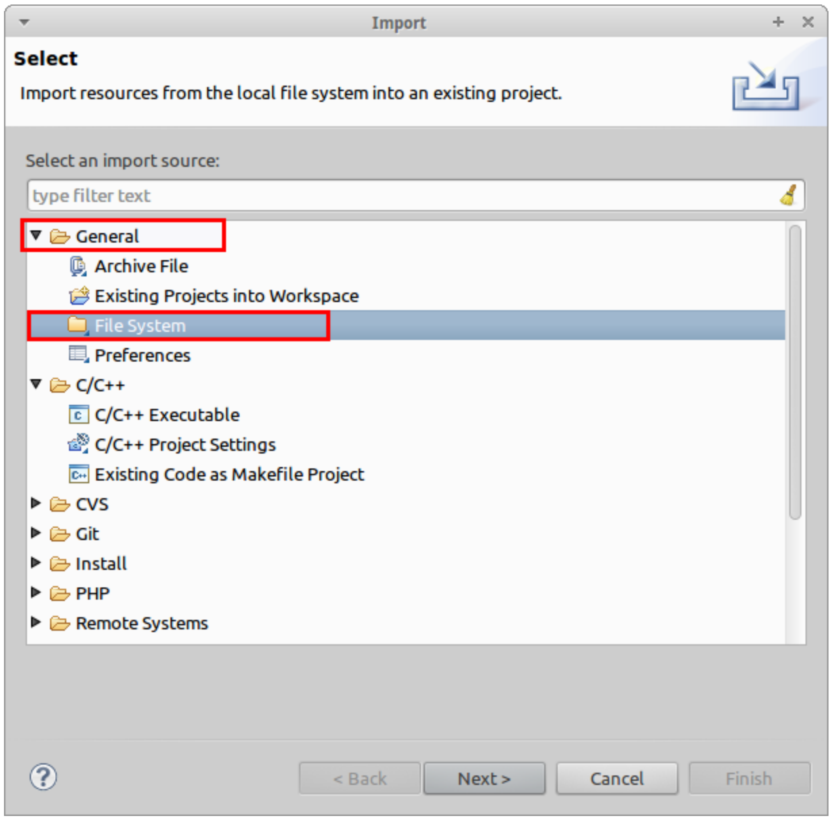
Use "Advanced settings" button to edit project's properties.

Additional configurations can be added after project creation.
Use "Manage configurations" buttons either on toolbar or on property pages.

You are now ready to import the source files. Right-click the project and click **Import …**from the menu.



From the Import window, select **General > File System** as the import source. Then click **Next >**.



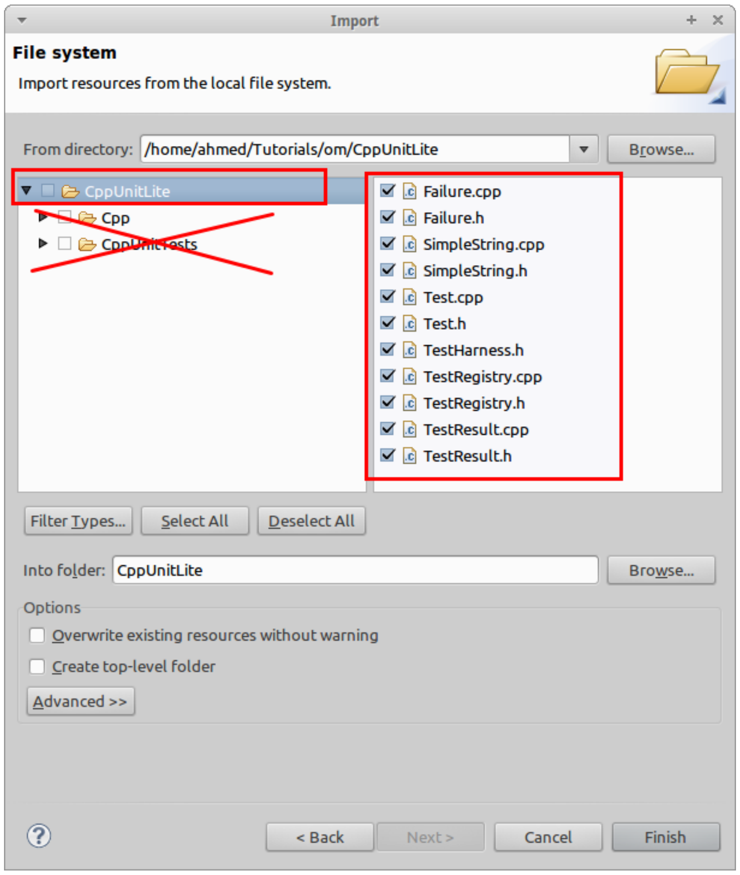
Import

Select

Import resources from the local file system into an existing project.

Select an import source:

From the next window, click on the **Browse** button, and navigate to the folder where the needed CppUnitLite sources (highlighted at the top of this post) are located. Now select only the needed files. Make sure **NOT** to select the two sub-folders Cpp and CppUnitTests. When you’re done, click **Finish**.


Import

File system

Import resources from the local file system.

From directory: /home/ahmed/Tutorials/om/CppUnitLite

- CppUnitLite
  - ~~Cpp~~
  - ~~CppUnitTests~~

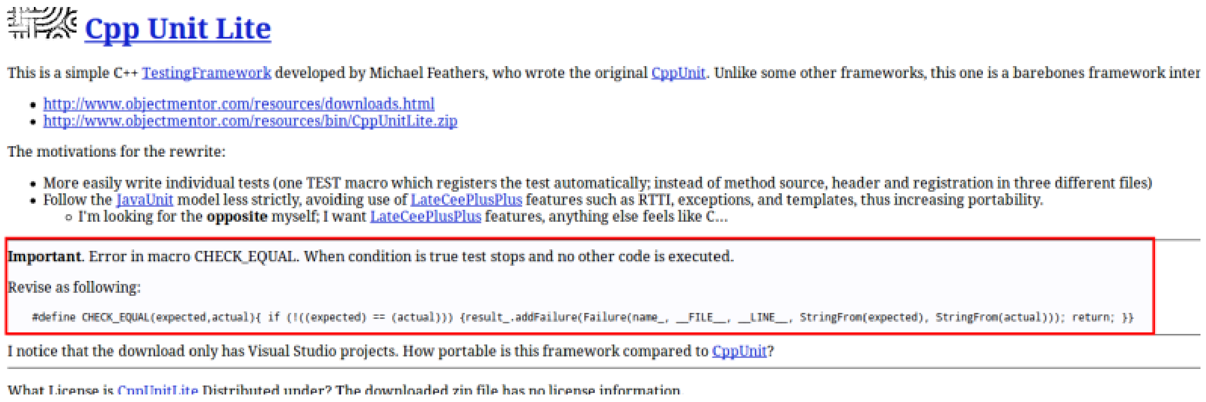
- [✓] Failure.cpp
- [✓] Failure.h
- [✓] SimpleString.cpp
- [✓] SimpleString.h
- [✓] Test.cpp
- [✓] Test.h
- [✓] TestHarness.h
- [✓] TestRegistry.cpp
- [✓] TestRegistry.h
- [✓] TestResult.cpp
- [✓] TestResult.h

Into folder: CppUnitLite

Options

- [ ] Overwrite existing resources without warning
- [ ] Create top-level folder

Now before we build, I hope you noticed on their website that they have this important note about a bug in the **CHECK_EQUAL** macro, and the revised one has been provided. This macro is located in the file **Test.h**. Copy the correct one from the website, and paste over the wrong one in the file.



Cpp Unit Lite

This is a simple C++ TestingFramework developed by Michael Feathers, who wrote the original CppUnit. Unlike some other frameworks, this one is a barebones framework inter

- http://www.objectmentor.com/resources/downloads.html
- http://www.objectmentor.com/resources/bin/CppUnitLite.zip

The motivations for the rewrite:

- More easily write individual tests (one TEST macro which registers the test automatically; instead of method source, header and registration in three different files)
- Follow the JavaUnit model less strictly, avoiding use of LateCeePlusPlus features such as RTTI, exceptions, and templates, thus increasing portability.
  - I'm looking for the **opposite** myself; I want LateCeePlusPlus features, anything else feels like C...

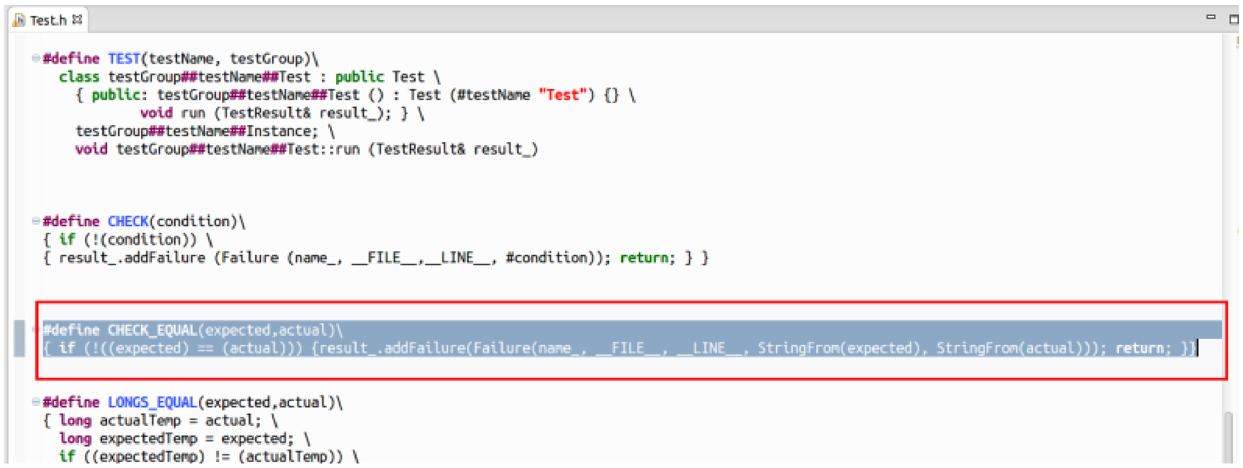
Important. Error in macro CHECK_EQUAL. When condition is true test stops and no other code is executed.

Revise as following:

```
#define CHECK_EQUAL(expected,actual){ if (!((expected) == (actual))) {result_.addFailure(Failure(name_, __FILE__, __LINE__, StringFrom(expected), StringFrom(actual))); return; }}
```

I notice that the download only has Visual Studio projects. How portable is this framework compared to CppUnit?

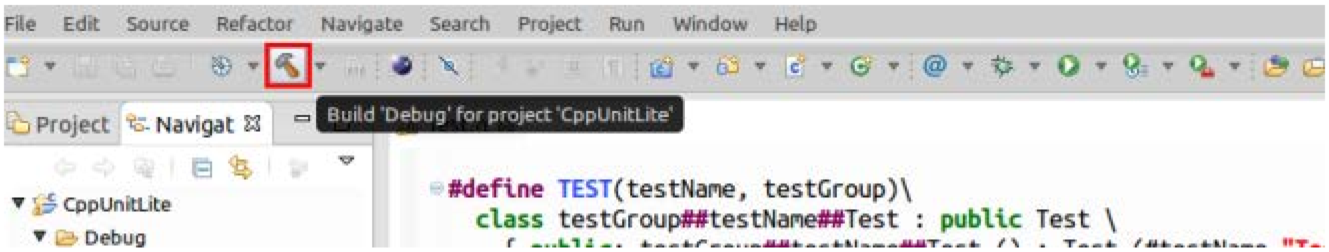
What License is CppUnitLite Distributed under? The downloaded zip file has no license information



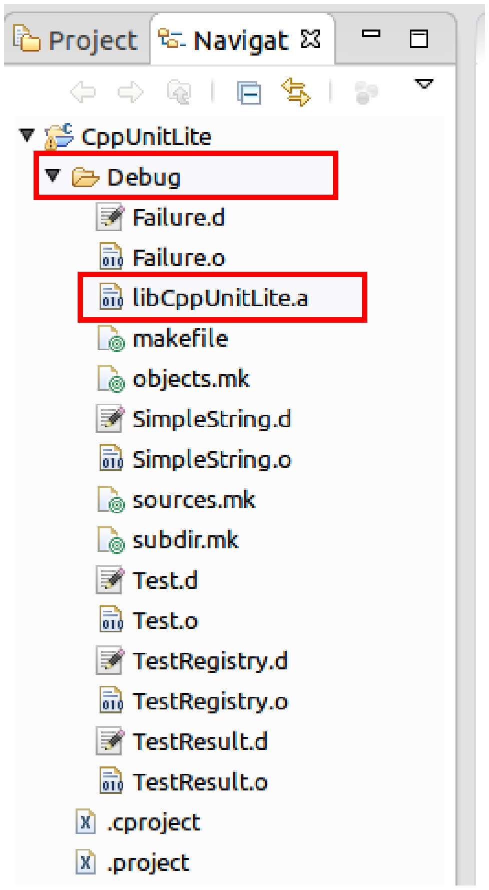
Test.h

```
#define TEST(testName, testGroup)\
  class testGroup##testName##Test : public Test \
    { public: testGroup##testName##Test () : Test (#testName "Test") {} \
            void run (TestResult& result_); } \
    testGroup##testName##Instance; \
    void testGroup##testName##Test::run (TestResult& result_)


#define CHECK(condition)\
{ if (!(condition)) \
{ result_.addFailure (Failure (name_, __FILE__,__LINE__, #condition)); return; } }


#define CHECK_EQUAL(expected,actual)\
{ if (!((expected) == (actual))) {result_.addFailure(Failure(name_, __FILE__, __LINE__, StringFrom(expected), StringFrom(actual))); return; }}

#define LONGS_EQUAL(expected,actual)\
{ long actualTemp = actual; \
  long expectedTemp = expected; \
  if ((expectedTemp) != (actualTemp)) \
```

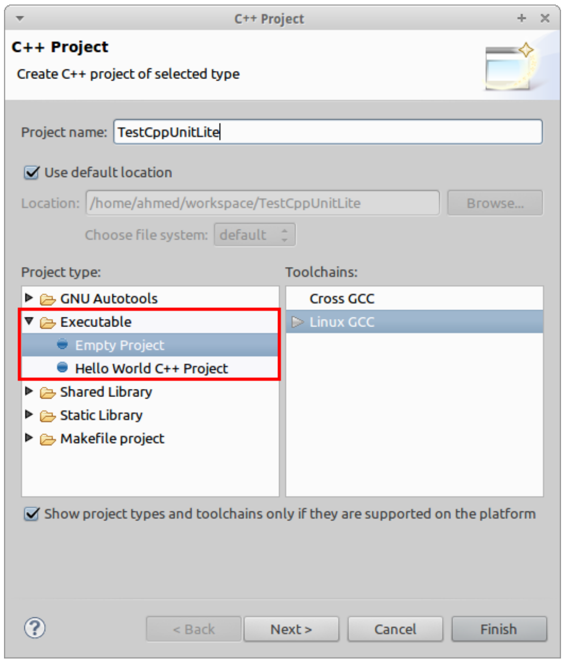
Now **Build** the project. This will create a sub-folder named **Debug** under your project. If you take a look inside it, you’ll find a file named **libCppUnitLite.a**. This is the static library file that you would want to link in your projects later.



```
#define TEST(testName, testGroup)\
  class testGroup##testName##Test : public Test \
```



Now we're done with building the library, and we need to experiment with it to write some simple unit tests. For demonstration purposes, we will create a new project, import the library, and the header files of CppUnitLite there, and write some simple unit tests just to show how things work. Create a new C++ project, and make sure the project type is**Executable** this time. Name the project whatever you want. I named it **TestCppUnitLite**. Now click **Finish**.


C++ Project

C++ Project

Create C++ project of selected type

Project name: TestCppUnitLite

☑ Use default location

Location: /home/ahmed/workspace/TestCppUnitLite

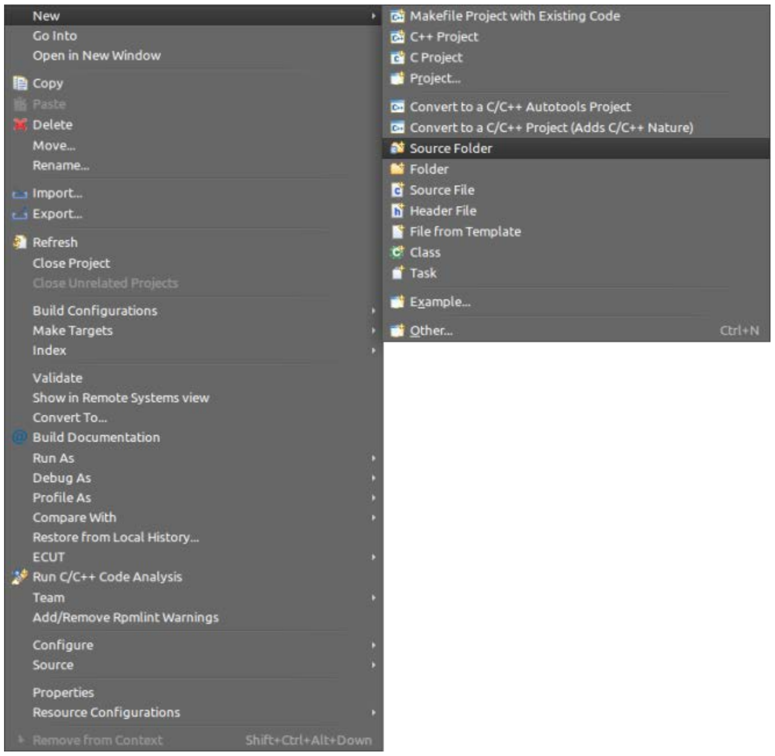
Choose file system: default

Project type: Executable > Empty Project

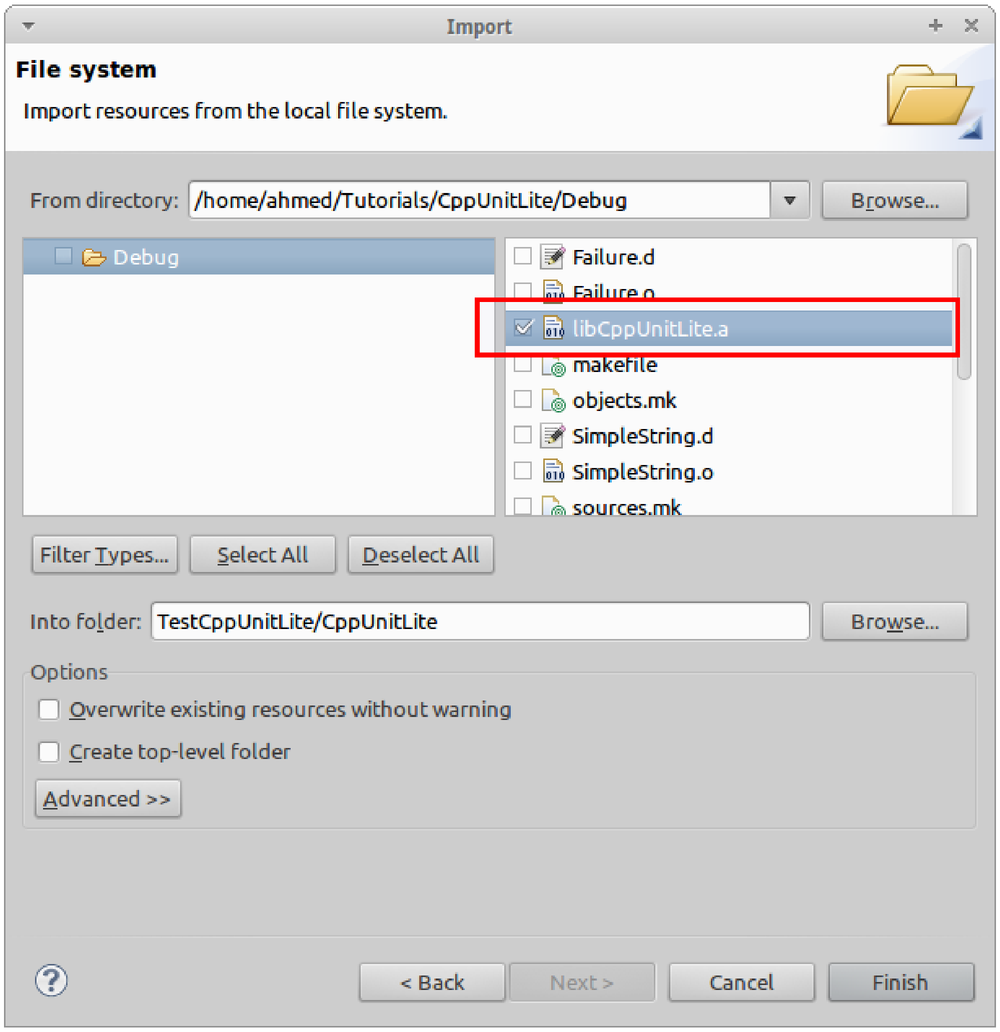
Toolchains: Linux GCC

☑ Show project types and toolchains only if they are supported on the platform

We will create a sub-folder under this project in which we will import the CppUnitLite library and header files shortly. Right-click on the project name and select **New > Source Folder**. Name the folder CppUnitLite.



Now right-click on that newly created folder, and select **Import**. From the **Import** window, as shown above, select **General > File System** as the import source and click **Next >**. Click the **Browse** button and navigate to the folder where we built the library. Once you do that select only the library file (**libCppUnitLite.a**) from the list and click finish.



Import

File system

Import resources from the local file system.

From directory: /home/ahmed/Tutorials/CppUnitLite/Debug

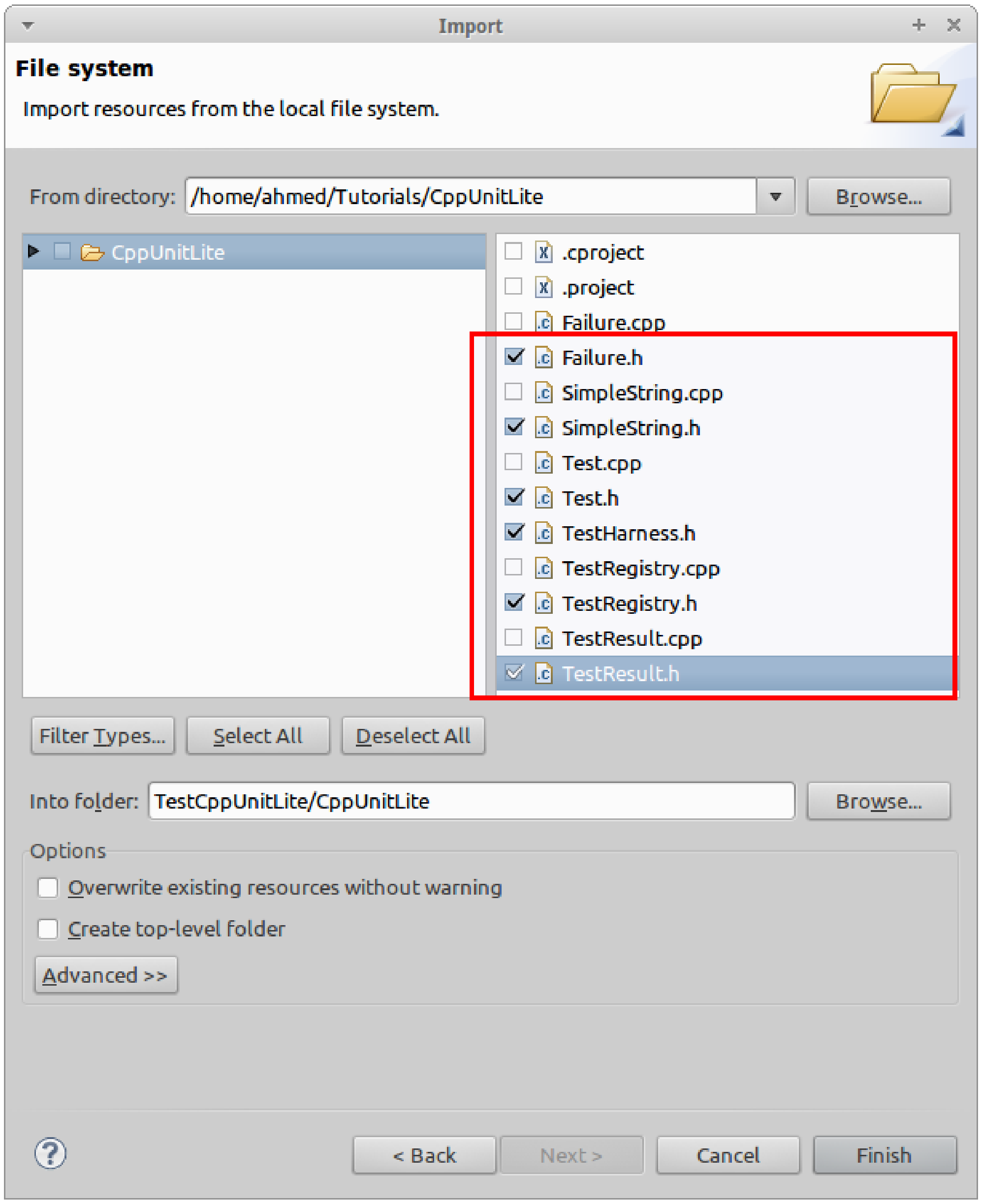
Debug

☐ Failure.d
☐ Failure.o
☑ libCppUnitLite.a
☐ makefile
☐ objects.mk
☐ SimpleString.d
☐ SimpleString.o
☐ sources.mk

Into folder: TestCppUnitLite/CppUnitLite

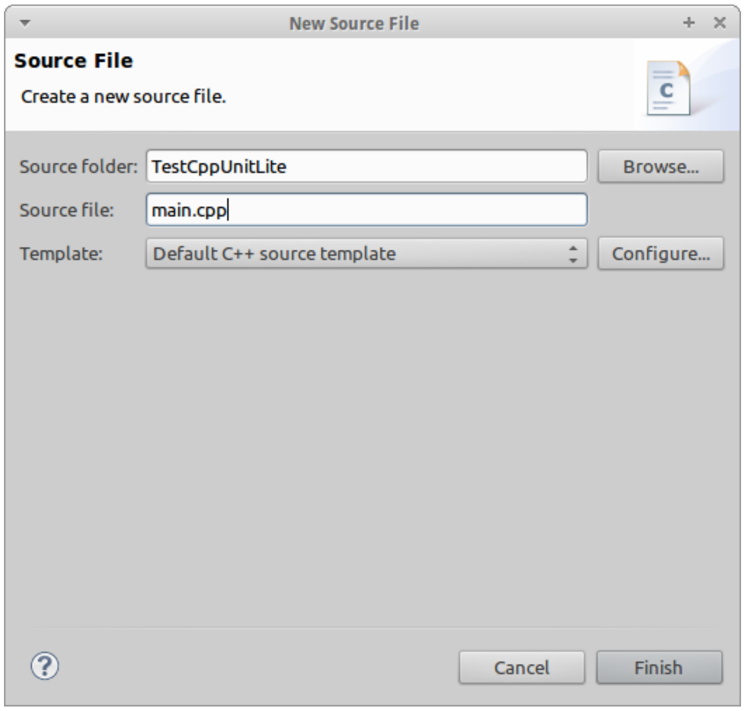
Options
☐ Overwrite existing resources without warning
☐ Create top-level folder

Repeat exactly the same above mentioned process again to import the header files. Just browse to where they are and import them into the same folder. You'll need to import 6 header files namely **Test.h, TestHarness.h, TestResult.h, TestRegistry.h, SimpleString.h, and Failure.h**. Then click **Finish**.


Import
File system
Import resources from the local file system.
From directory: /home/ahmed/Tutorials/CppUnitLite
Browse...
CppUnitLite
Filter Types...
Select All
Deselect All
Into folder: TestCppUnitLite/CppUnitLite
Browse...
Options
Overwrite existing resources without warning
Create top-level folder
Advanced >>
< Back
Next >
Cancel
Finish

- [ ] .cproject
- [ ] .project
- [ ] Failure.cpp
- [x] Failure.h
- [ ] SimpleString.cpp
- [x] SimpleString.h
- [ ] Test.cpp
- [x] Test.h
- [x] TestHarness.h
- [ ] TestRegistry.cpp
- [x] TestRegistry.h
- [ ] TestResult.cpp
- [x] TestResult.h

Now it’s time to write some simple code. Right-click on the project name, and click **New** > **Source File**. Name the file **main.cpp** and click **Finish**.


New Source File

Source File

Create a new source file.

Source folder: TestCppUnitLite

Source file: main.cpp

Template: Default C++ source template

All you need to write is the following code. It's kind of like a boiler-plate code to start running the tests and reporting the results.

```
#include "CppUnitLite/TestHarness.h"
int main()
{
    TestResult tr;
    TestRegistry::runAllTests(tr);
    return 0;
}
```

At this moment there isn't any tests to run, so we need to write some very simple tests just to show how things work. We will create a new **source file** and we will name it **tests.cpp**. In this newly created file we will write some tests, some of them are intended to fail just to show how CppUnitLite reports a test failure. Notice that CppUnitLite used predefined macros to enable us to write these simple unit tests easily. Copy and paste the following code into **tests.cpp**.

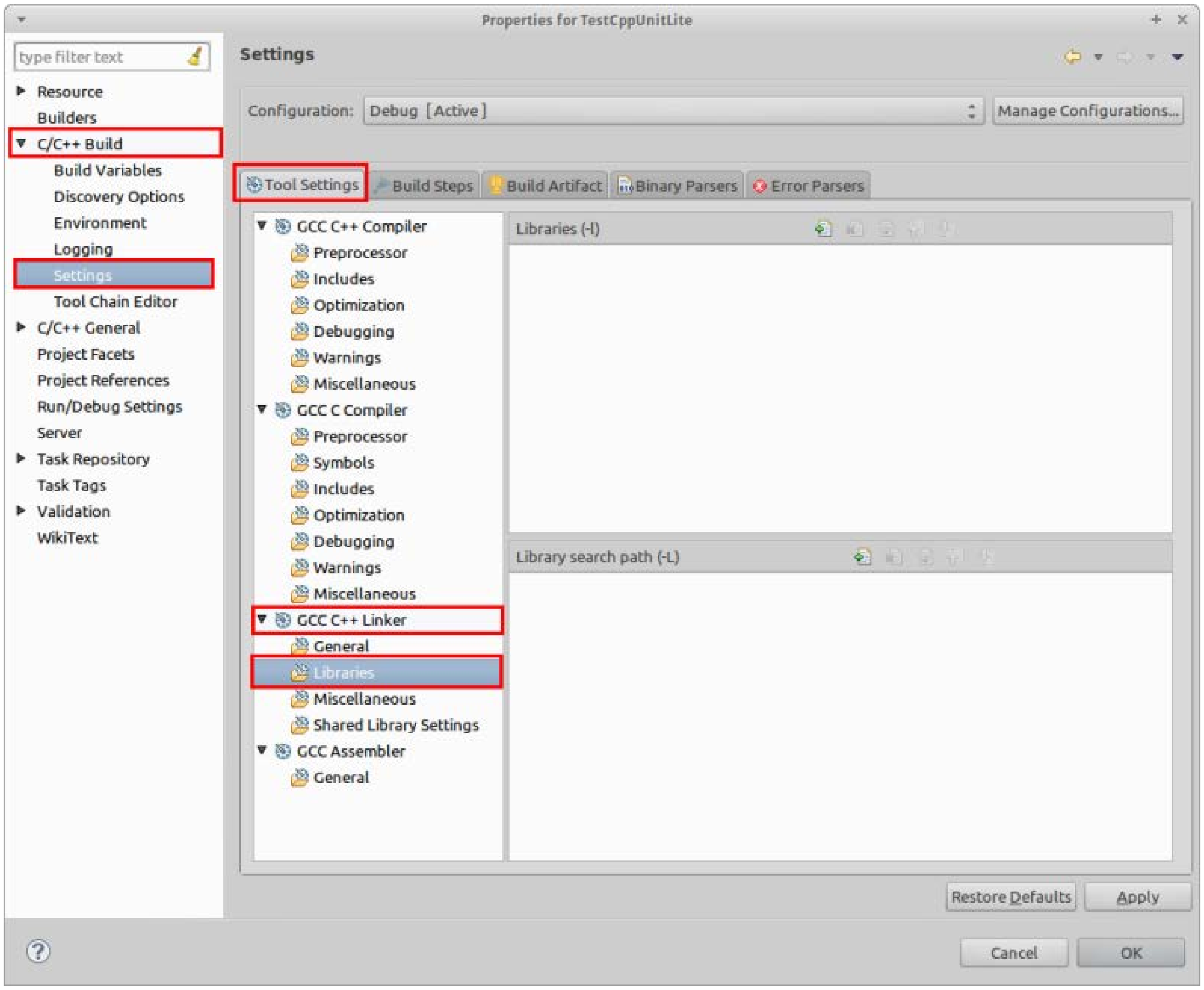
```
#include "CppUnitLite/TestHarness.h"
TEST(EqualitySuccess, EqualityGroup)
{
    int a = 5;
    CHECK(a == 5);
}
TEST(EqualityFailure, EqualityGroup)
{
    int a = 6;
    CHECK(a == 5);
}
TEST(GreaterSuccess, GreaterGroup)
{
    int a = 50;
    CHECK(a > 30);
}
TEST(GreaterFailure, GreaterGroup)
{
    int a = 50;
    CHECK(a > 100);
}
```

For any test you write, First you'll need to include the **TestHarness.h** header file, which includes all of the others for you. Next you'll need to use the **TEST** macro. the **TEST** macro takes two arguments, the first is the test name, the second is the test group. This is useful when you want to create multiple tests that belong to a single group. As you can see above I have to test groups, each of which has two tests, one that is intended to succeed, and the other is intended to fail.

I'm using the **CHECK** macro for all of my tests above, but you have others too that you can use such as **CHECK_EQUAL**, **LONGS_EQUAL**, .. etc. They're defined in the header file **Test.h**.

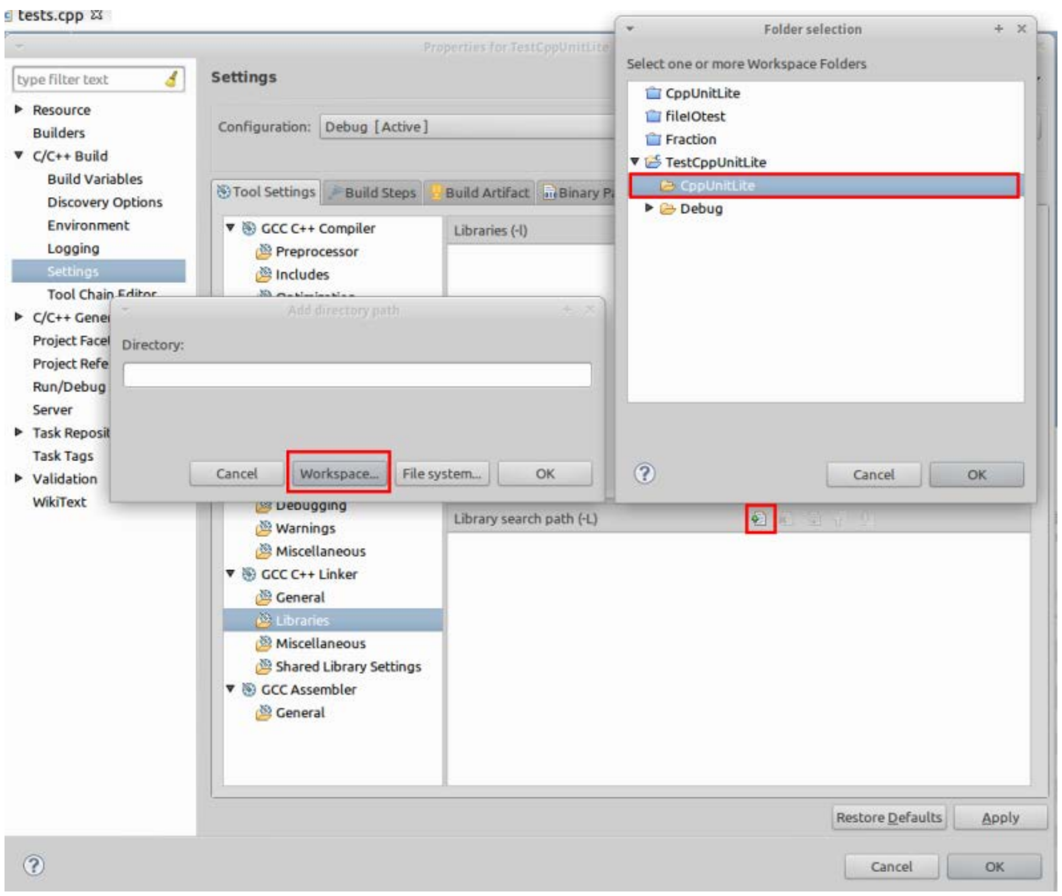
Now we need to build and run those tests to see the result. But **WAIT!!** If you try to build now, you'll fail as the linker doesn't know how to link to the library file. You must specify that yourself.

Right-click the project name and select **Properties**. From the Properties window, go to C/C++ Build > Settings. From the Tool Settings tab, under GCC C++ Linker > Libraries, you'll need to do two things.



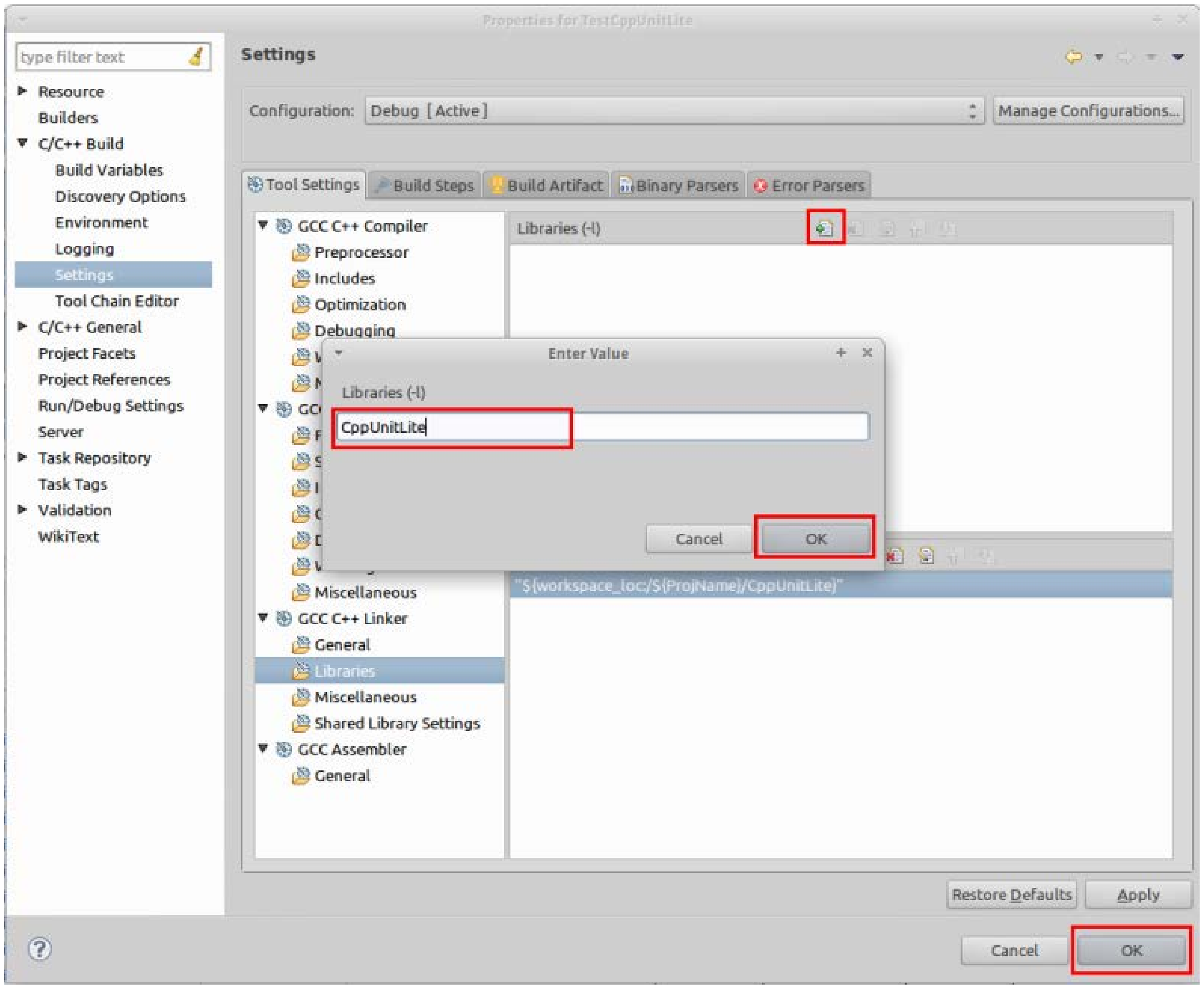
Configuration: Debug [Active]

You’ll need to add the library search path. Click on the **Add…** button next to **Library Search Path (-L)** and click on the **Workspace** button, and select the folder where we imported the library file. click Ok, and then Ok again.



Configuration: Debug [Active]

Next you'll need to specify the name of the library. From the same window click on the**Add…** button next to **Libraries (-l)** and type **CppUnitLib**. Remember our library file is name**libCppUnitLite.a**, but GCC C++ Linker doesn't need the **lib** or **.a** parts of the name. So if you named your library **libMonkey.a**, just type **Monkey** when you add the library :). Click Ok and then Ok again to exit the project Properties window.



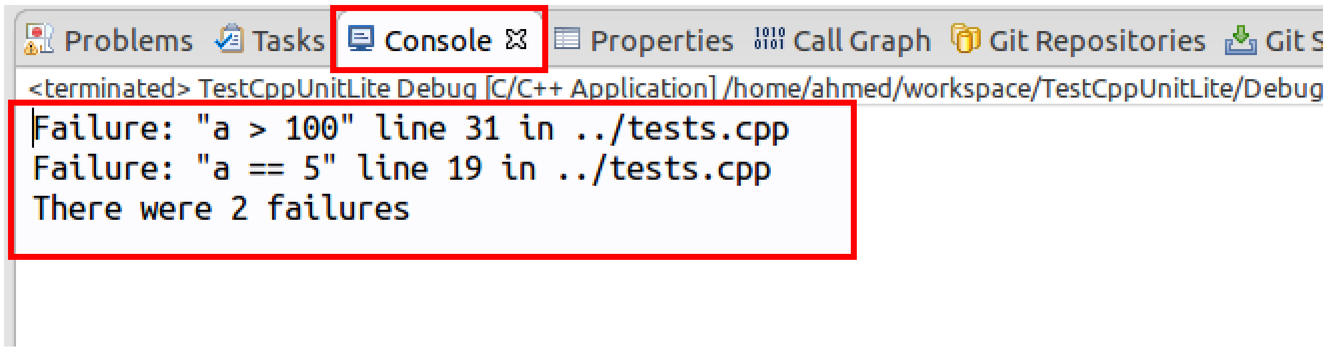
Configuration: Debug [Active]

Libraries (-l)

Libraries (-l): CppUnitLite

"${workspace_loc:/${ProjName}/CppUnitLite}"

Now everything is ready to build and run. Build the project and run it as a C++ Application, and see the output on the console window. You'll see that it reports the two intentional failures, what conditions that failed, in which files they are, and in which lines as well.



```
<terminated> TestCppUnitLite Debug [C/C++ Application] /home/ahmed/workspace/TestCppUnitLite/Debug
Failure: "a > 100" line 31 in ../tests.cpp
Failure: "a == 5" line 19 in ../tests.cpp
There were 2 failures
```

Source : http://webinstitute.wordpress.com/2013/02/15/building-and-integrating-cppunitlite-in-eclipse-on-linux/