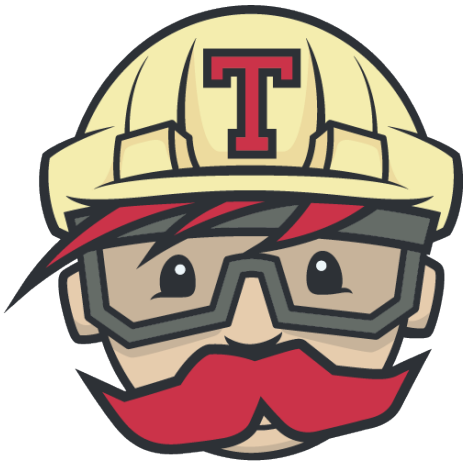
Travis C++ tutorial

Richèl Bilderbeek

March 14, 2016



Contents

1 Introduction

This is a Travis C++ tutorial, version 0.1

1.1 License

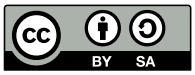


Figure 1: Creative Commons license 4.0

1.2 Continuous integration

Collaboration can be scary: the other(s)[1] may break the project worked on. The project can be of any type, not only programming, but also collaborative writing.

A good first step ensuring a pleasant experience is to use a version control system. A version control system keeps track of the changes in the project and allows for looking back in the project history when something has been broken.

The next step is to use an online version control repository, which makes the code easily accessible for all contributors. The online version control repository may also offer additional collaborative tools, like a place where to submit bug reports, define project milestones and allowing external people to submit requests, bug reports or patches.

Up until here, it is possible to submit a change that breaks the build.

A continuous integration tools checks what is submitted to the project and possibly rejects it when it does not satisfy the tests and/or requirements of the project. Instead of manually proofreading and/or testing the submission and

[1] if not you

mailing the contributor his/her addition is rejected is cumbersome at least. A continuous integration tool will do this for you.

Now, if someone changes you project, you can rest assured that his/her submission does not break the project. Enjoy!

1.3 Tool used

git git is a version control system. It tracks the changes made in the project and allows for viewing the project its history.

GitHub GitHub is a site where git repositories are hosted. It gives a git project a website where the files can be viewed. Next to this, there is a project page for issues like bug reports and feature requests.

Travis CI Travis CI is a continuous integration (hence the 'CI' in its name) tool that plays well with GitHub. It is activated when someone uploads his/her code to the GitHub.

Boost Boost is a collection of C++ libraries.

Boost.Test Boost.Test is a C++ testing framework within the Boost libraries.

gcov gcov is a GNU tool to measur the code coverage of (among others) C++ code. It can be actived from a Travis script.

Codecov Codecov is a tool to display a gcov code coverage result, that plays well with GitHub. It can be actived from a Travis script.

gprof gprof is a GNU tool to profile (among others) C++ code. It can be actived from a Travis script.

1.4 Feedback

This tutorial is not intended to be perfect yet. For that, I need help and feedback from the community. All referenced feedback is welcome, as well as any constructive feedback.

2 Setting up the basic build

The basic build is more than just a collection of files. It needs to be set up. This chapter shows how to do so.

- Create a GitHub online
- Bring the git repository to your local computer

- Create a Qt Creator project
- Create the build bash scripts

2.1 Create a GitHub online

What is GitHub? GitHub is a site that creates websites around projects. It is said to host the project. This project contains one, but usually a collection of files, which is called a repository. GitHub also keeps track of the history of the project, which is also called version control. GitHub uses git as a version control software. In short: GitHub hosts git repositories.

Register Before you can create a new repository, you must register
where git repositories are hosted. It gives a git project a website where the files can be viewed. Next to this, there is a project page for issues like bug reports and feature requests.

Creating a repository

2.2 Bring the git repository to your local computer

2.3 Create a Qt Creator project

2.4 Create the build bash scripts

3 The basic build

The basic build has the following specs:

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++98
- Libraries: STL only
- Code coverage: none
- Source: one single file, main.cpp

First I will show the single file this build is about:

Algorithm 1 main.cpp

```
#include <iostream>

int main() {
  std::cout << "Hello world\n";
}
```

All the code does is display the text 'Hello world', which is a traditional start for many programming languages. The code is written in C++98. It does not use features from the newer C++ standards. It will not compile under plain C.

This single file is compiled with qmake from the following Qt Creator project file:

Algorithm 2 travis_qmake_gcc_cpp98.pro

```
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle qt
SOURCES += main.cpp
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror
```

This Qt Creator project file has a typical setup for a standard console application, except that, in the last line, the warning level is set the the highest level, even making a warning break the build. This forces collaborators to write tidy code.

The bash build script to build this is:

Algorithm 3 build.sh

```
qmake
make
./travis_qmake_gcc_cpp98
```

This build script calls 'qmake' to create a makefile. Then 'make' is called to compile the makefile. Finally, the created executable 'travis_qmake_gcc_cpp98' is run. There is a potential error in the first and last step: the Qt Creator project file may be incorrect, or the executable will crash, possibly due to a failed test.

Setting up Travis is done by the following .travis.yml[2] file:

Algorithm 4 .travis.yml

```
language: cpp
compiler: gcc
script: ./build.sh
```

This .travis.yml file has the following elements:

- `language: cpp`

  The main programming language of this project is C++

[2] the filename starts with a dot. This means it is a hidden file

- `compiler: gcc`

  The C++ code will be compiled by GCC

- `script: ./travis_qmake_gcc_cpp98`

  The script that Travis will run, which is running the generated executable called 'travis_qmake_gcc_cpp98'.

4 Extending the build by one step

The following chapter describe how to extend the build in one direction. These are:

- Use of C++11: see chapter 4.1
- Use of C++14: see chapter 4.2
- Use of Boost: see chapter 4.3
- Use of Boost.Test: see chapter 4.4
- Use of gcov: see chapter 4.5
- Use of gprof: see chapter 4.6

4.1 Use of C++11

In this example, the basic build (chapter 3) is extended by using C++11.
The chapter has the following specs:

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++11
- Libraries: STL only
- Code coverage: none
- Source: one single file, main.cpp

The single C++ source file used is:

Algorithm 5 main.cpp

```
#include <iostream>

void f() noexcept {
  std::cout << "Hello world\n";
}

int main() { f(); }
```

All the file does is to create an empty graph, from the Boost.Graph library. It will not compile without the Boost libraries absent.

This single file is compiled with qmake from the following Qt Creator project file:

Algorithm 6 travis_qmake_gcc_cpp98.pro

```
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle qt
SOURCES += main.cpp
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

QMAKE_CXX = g++-5
QMAKE_LINK = g++-5
QMAKE_CC = gcc-5
QMAKE_CXXFLAGS += -std=c++11
```

The Qt Creator project file has the same lines as the basic project in chapter 3.

The bash build script to build this, run this and measure the code coverage:

Algorithm 7 build.sh

```
qmake
make
./travis_qmake_gcc_cpp11
```

The bash script has the same lines as the basic project in chapter 3.

Setting up Travis is done by the following .travis.yml:

Algorithm 8 .travis.yml

```
sudo: true
language: cpp
compiler: gcc
before_install:
  - sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test
  - sudo apt-get update -qq
install: sudo apt-get install -qq g++-5
script: ./build.sh
```

This .travis.yml file has one new feature:

- addons:
    apt:
      packages: libboost-all-dev

  This makes Travis aware that you want to use the aptitude package 'libboost-all-dev'. Note that this code cannot be put on one line: it has to be indented similar to this

4.2 Use of C++14

4.3 Adding the Boost libraries

In this example, the basic build (chapter 3) is extended by also using the Boost libraries.

The chapter has the following specs:

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++98
- Libraries: STL and Boost
- Code coverage: none
- Source: one single file, main.cpp

The single C++ source file used is:

Algorithm 9 main.cpp

```
#include <boost/graph/adjacency_list.hpp>

int main() {
  const boost::adjacency_list<> g;
}
```

All the file does is to create an empty graph, from the Boost.Graph library. It will not compile without the Boost libraries absent.

This single file is compiled with qmake from the following Qt Creator project file:

Algorithm 10 travis_qmake_gcc_cpp98.pro

```
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle qt
SOURCES += main.cpp
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror
```

The Qt Creator project file has the same lines as the basic project in chapter 3.

The bash build script to build this, run this and measure the code coverage:

Algorithm 11 build.sh

```
qmake
make
./travis_qmake_gcc_cpp98_boost
```

The bash script has the same lines as the basic project in chapter 3.

Setting up Travis is done by the following .travis.yml:

Algorithm 12 .travis.yml

```
language: cpp
compiler: gcc
addons:
  apt:
    packages: libboost-all-dev
script: ./build.sh
```

This .travis.yml file has one new feature:

- addons:
    apt:
      packages: libboost-all-dev

  This makes Travis aware that you want to use the aptitude package 'libboost-all-dev'. Note that this code cannot be put on one line: it has to be indented similar to this

4.4 Adding a testing framework

Adding only a testing framework does not work: it will not compile in C++98. Instead, this is covered in chapter 5.1.

4.5 Adding code coverage

In this example, the basic build (chapter 3) is extended by calling gcov and using codecov to show the code coverage.

The basic build has the following specs:

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++98
- Libraries: STL only
- Code coverage: none
- Source: one single file, main.cpp

The single C++ source file used is:

Algorithm 13 main.cpp

```
#include <iostream>

int main(int argc, char* argv[])
{
  if (argc >= 1) {
    std::cout << argv[0] << '\n';
  }
  else {
    std::cout << "I will never be called\n";
  }
}
```

This file openly contains some dead code, so we expect to observe a code coverage less than 100%.

This single file is compiled with qmake from the following Qt Creator project file:

Algorithm 14 travis_qmake_gcc_cpp98.pro

```
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle qt
SOURCES += main.cpp
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

# gcov
QMAKE_CXXFLAGS += -fprofile-arcs -ftest-coverage
LIBS += -lgcov
```

The Qt Creator project file has two new lines. The first of those adds two compiler flags, which cause the code to be compiled in such a way to gcov can work with it. The second line links the gcov library to the project.

The bash build script to build this, run this and measure the code coverage:

Algorithm 15 build.sh

```
qmake
make
./travis_qmake_gcc_cpp98_gcov
gcov main.cpp
cat main.cpp.gcov
```

The new step is after having run the executable, where gcov is run on the only source file. The text 'gcov' has generated is then shown using 'cat'.

Setting up Travis is done by the following .travis.yml:

Algorithm 16 .travis.yml

```
sudo: true
language: cpp
compiler: gcc
before_install: sudo pip install codecov
script: ./build.sh
after_success: codecov
```

This .travis.yml file has some new features:

- sudo: true

  Travis will give super user rights to the script. This will slow the build time, but it is inevitable for the next step

- `before_install: sudo pip install codecov`

  Travis will use pip to install codecov using super user rights

- `after_success: codecov`

  After the script has run successfully, codecov is called

4.6 Adding profiling

5 Extending the build by multiple steps

5.1 C++11, Boost and Boost.Test

Adding only a testing framework does not work: it will not compile in C++98. Instead, this is covered in

In this example, the basic build (chapter 3) is extended by calling gcov and using codecov to show the code coverage.

The basic build has the following specs:

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++11
- Libraries: STL and Boost, demonstrating Boost.Test
- Code coverage: none
- Source: multiple files: main.cpp, my_function.h, my_function.cpp, test_my_function.cpp

This project use four types of builds and each of these produces a different file. These files are used to let Travis check if the executable were built correctly. An overview of builds and files are shown in table 1:

	Exe	Test
Debug	exe_debug.txt	test_debug.txt
Release	exe_release.txt	test_debug.txt

Table 1: Types of builds and the file they will create

5.1.1 The function

First the function that is (1) tested by the test build (2) called by the real build, is shown here:

Algorithm 17 my_function.h

```
#ifndef MY_FUNCTIONS_H
#define MY_FUNCTIONS_H

int add(const int i, const int j) noexcept;

#endif // MY_FUNCTIONS_H
```

This header file has the #include guards and the declaration of the function 'add'. It takes two integer values as an argument and returns an int.

Its definition is shown here:

Algorithm 18 my_function.cpp

```
#include "my_functions.h"

int add(const int i, const int j) noexcept
{
  return i + j;
}
```

Perhaps it was expected that 'add' adds the two integers

5.1.2 Test build

The test build' is the build that tests the function. It does not have a 'main.cpp' as the exe build has, but uses 'test_my_functions.cpp' as its main source file. This can be seen in the Qt Creator project file:

Algorithm 19 travis_qmake_gcc_cpp11_boost_test_test.pro

```
CONFIG += console debug_and_release
CONFIG -= app_bundle
QT -= core gui
TEMPLATE = app

CONFIG(release, debug|release) {
  DEFINES += NDEBUG
}

QMAKE_CXX = g++-5
QMAKE_LINK = g++-5
QMAKE_CC = gcc-5
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror -std=c
    ++11

SOURCES += \
    my_functions.cpp

HEADERS += \
    my_functions.h

SOURCES += test_my_functions.cpp

LIBS += -lboost_unit_test_framework
```

Note how this Qt Creator project file links to the Boost unit test framework. Its main source file is shown here:

Algorithm 20 test_my_functions.cpp

```
#include <fstream>
#include "my_functions.h"


#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE my_test_module
#include <boost/test/unit_test.hpp>

BOOST_AUTO_TEST_SUITE(my_functions)

BOOST_AUTO_TEST_CASE(add_works)
{
  #ifndef NDEBUG
  std::ofstream f("test_debug.txt");
  #else
  std::ofstream f("test_release.txt");
  #endif

  BOOST_CHECK(add(1, 1) == 2);
  BOOST_CHECK(add(1, 2) == 3);
  BOOST_CHECK(add(1, 3) == 4);
  BOOST_CHECK(add(1, 4) == 5);

  f << "OK\n";
}

BOOST_AUTO_TEST_SUITE_END()
```

It uses the Boost.Test framework to automatically generate a test suites. Next to testing the function 'add', also a file is created, so Travis can verify there has been built both a debug and release mode.

5.1.3 Exe build

The 'exe' build' is the build that uses the function.

Algorithm 21 main.cpp

```
#include "my_functions.h"
#include <fstream>

int main()
{
  #ifndef NDEBUG
  std::ofstream f("exe_debug.txt");
  #else
  std::ofstream f("exe_release.txt");
  #endif
  f << add(40,2) << '\n';
}
```

Next to using the function 'add', also a file is created, so Travis can verify there has been built both a debug and release mode.

This single file is compiled with qmake from the following Qt Creator project file:

Algorithm 22 travis_qmake_gcc_cpp11_boost_test_exe.pro

```
CONFIG += console debug_and_release
CONFIG -= app_bundle
QT -= core gui
TEMPLATE = app

CONFIG(release, debug|release) {
  DEFINES += NDEBUG
}

QMAKE_CXX = g++-5
QMAKE_LINK = g++-5
QMAKE_CC = gcc-5
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror -std=c++11

SOURCES += main.cpp \
    my_functions.cpp

HEADERS += \
    my_functions.h
```

Note how this Qt Creator project file does not link to the Boost unit test framework.

5.1.4 Build script

The bash build script to build this, run this and measure the code coverage:

Algorithm 23 build.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp11_boost_test_exe.pro
make debug
./travis_qmake_gcc_cpp11_boost_test_exe
if [ ! -f exe_debug.txt ]
then
  echo "ERROR: Cannot find exe_debug.txt"
  exit 1
fi

qmake travis_qmake_gcc_cpp11_boost_test_exe.pro
make release
./travis_qmake_gcc_cpp11_boost_test_exe
if [ ! -f exe_release.txt ]
then
  echo "ERROR: Cannot find exe_release.txt"
  exit 1
fi

qmake travis_qmake_gcc_cpp11_boost_test_test.pro
make debug
./travis_qmake_gcc_cpp11_boost_test_test
if [ ! -f test_debug.txt ]
then
  echo "ERROR: Cannot find test_debug.txt"
  exit 1
fi

qmake travis_qmake_gcc_cpp11_boost_test_test.pro
make release
./travis_qmake_gcc_cpp11_boost_test_test
if [ ! -f test_release.txt ]
then
  echo "ERROR: Cannot find test_release.txt"
  exit 1
fi
```

The new step is after having run the executable, where gcov is run on the only source file. The text 'gcov' has generated is then shown using 'cat'.

5.1.5 Travis script

Setting up Travis is done by the following .travis.yml:

Algorithm 24 .travis.yml

```
sudo: true
language: cpp
compiler: gcc
addons:
  apt:
    packages: libboost-all-dev
before_install:
  - sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test
  - sudo apt-get update -qq
install: sudo apt-get install -qq g++-5
script: ./build.sh
```

This .travis.yml file has some new features:

- `sudo: true`

  Travis will give super user rights to the script. This will slow the build time, but it is inevitable for the next step

- `before_install: sudo pip install codecov`

  Travis will use pip to install codecov using super user rights

- `after_success: codecov`

  After the script has run successfully, codecov is called