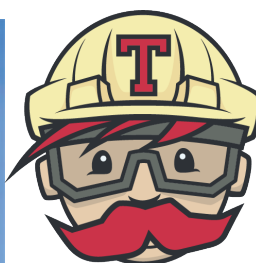


# Travis C++ tutorial

Richèl Bilderbeek

March 16, 2016



## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	License . . . . .	2
1.2	Continuous integration . . . . .	2
1.3	Tool used . . . . .	3
1.4	This tutorial . . . . .	4
1.5	Feedback . . . . .	4
<b>2</b>	<b>Setting up the basic build</b>	<b>4</b>
2.1	Create a GitHub online . . . . .	4
2.2	Bring the git repository to your local computer . . . . .	5
2.3	Create a Qt Creator project . . . . .	8
2.4	Create the build bash scripts . . . . .	8
<b>3</b>	<b>The basic build</b>	<b>8</b>
<b>4</b>	<b>Extending the build by one step</b>	<b>12</b>
4.1	Use of C++11 . . . . .	12
4.2	Use of C++14 . . . . .	14
4.3	Adding the Boost libraries . . . . .	16
4.4	Adding a testing framework . . . . .	18
4.5	Adding code coverage . . . . .	18
4.6	Adding profiling . . . . .	21
4.7	Adding Rcpp . . . . .	21

4.7.1	C++ and R: the C++ function . . . . .	22
4.7.2	C++: main source file . . . . .	23
4.7.3	C++: Qt Creator project file . . . . .	23
4.7.4	R: the R function . . . . .	24
4.7.5	R: The R tests . . . . .	25
4.7.6	The build script . . . . .	25
4.7.7	The Travis script . . . . .	25
<b>5</b>	<b>Extending the build by two steps</b>	<b>26</b>
5.1	C++11 and Boost libraries . . . . .	26
5.2	C++14 and Boost libraries . . . . .	28
<b>6</b>	<b>Extending the build by multiple steps</b>	<b>29</b>
6.1	C++11, Boost and Boost.Test . . . . .	29
6.1.1	The function . . . . .	30
6.1.2	Test build . . . . .	30
6.1.3	Exe build . . . . .	32
6.1.4	Build script . . . . .	33
6.1.5	Travis script . . . . .	34

## 1 Introduction

This is a Travis C++ tutorial, version 0.1

### 1.1 License

This tutorial is licensed under Creative Commons license 4.0. All C++ code is licensed under GPL 3.0.



Figure 1: Creative Commons license 4.0

### 1.2 Continuous integration

Collaboration can be scary: the other(s)<sup>1</sup> 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.

---

<sup>1</sup>if not you

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 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. git is discussed in more detail in chapter [TODO].

**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. GitHub is discussed in more detail in chapter [TODO].

**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. Travis CI is discussed in more detail in chapter [TODO].

**Boost** Boost is a collection of C++ libraries. Boost is discussed in more detail in chapter [TODO].

**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. gcov is discussed in more detail in chapter [TODO].

**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. Codecov is discussed in more detail in chapter [TODO].

**gprof** gprof is a GNU tool to profile (among others) C++ code. It can be actived from a Travis script. gprof is discussed in more detail in chapter [TODO].

**Rcpp** Rcpp is an R package that allows mixing R and C++ code. Rcpp is discussed in more detail in chapter [TODO].

**Rcpp11** Rcpp is an R package that allows mixing R and C++11 code. Rcpp11 is discussed in more detail in chapter [TODO].

## 1.4 This tutorial

This tutorial is available online at [https://github.com/richelbilderbeek/travis\\_cpp\\_tutorial](https://github.com/richelbilderbeek/travis_cpp_tutorial). Of course, it is checked by Travis that

- all the setups described work
- this document can be converted to PDF. For this it needs the files from all of these setups

## 1.5 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.

Figure 2 shows the GitHub homepage, <https://github.com>.

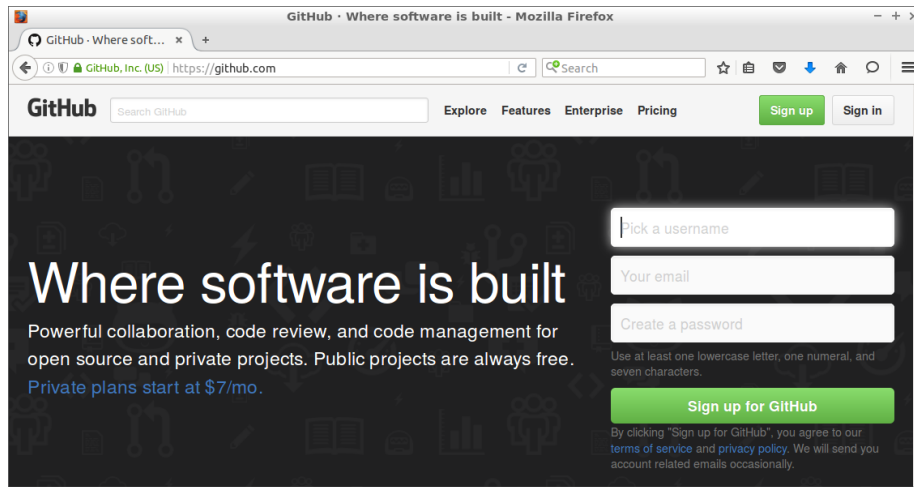


Figure 2: The GitHub homepage, <https://github.com>

**Register** Before you can create a new repository, you must register. Registration is free for open source projects, with an unlimited<sup>2</sup> amount of public repositories.

From the GitHub homepage, <https://github.com> (see figure 2), click the top right button labeled 'Sign up'. This will take you to the 'Join GitHub' page (see figure 3).

Filling this in should be as easy. After filling this in, you are taken to your GitHub profile page (figure 4).

**Creating a repository** From your GitHub profile page (figure 4), click on the plus ('Create new ...') at the top right, then click 'New repository' (figure 5).

Do check 'Initialize this repository with a README', add a .gitignore with 'C++' and add a licence like 'GPL 3.0'.

You have now created your own online version controlled repository (figure 6)!

## 2.2 Bring the git repository to your local computer

**What is git?** git is a version control system.

**Using git** Go to the terminal and type the following line to download your repository:

```
git clone https://github.com/[your_name]/[your_repository]
```

---

<sup>2</sup>the maximum I have observed is a person that has 350

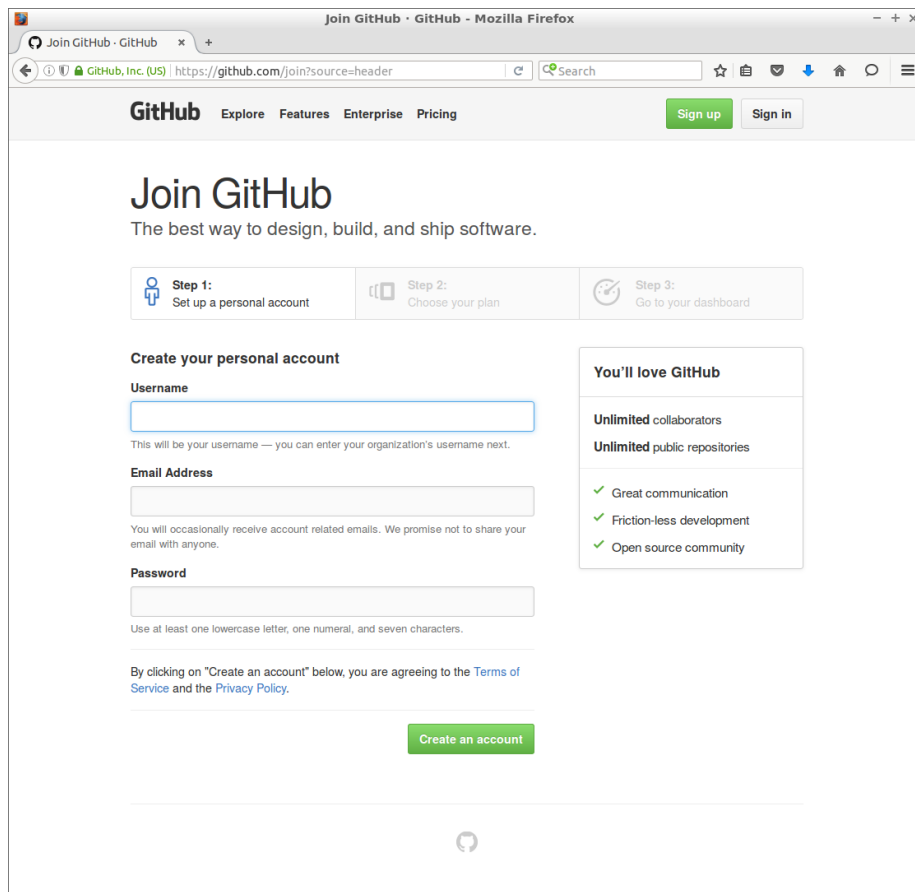


Figure 3: The join GitHub page

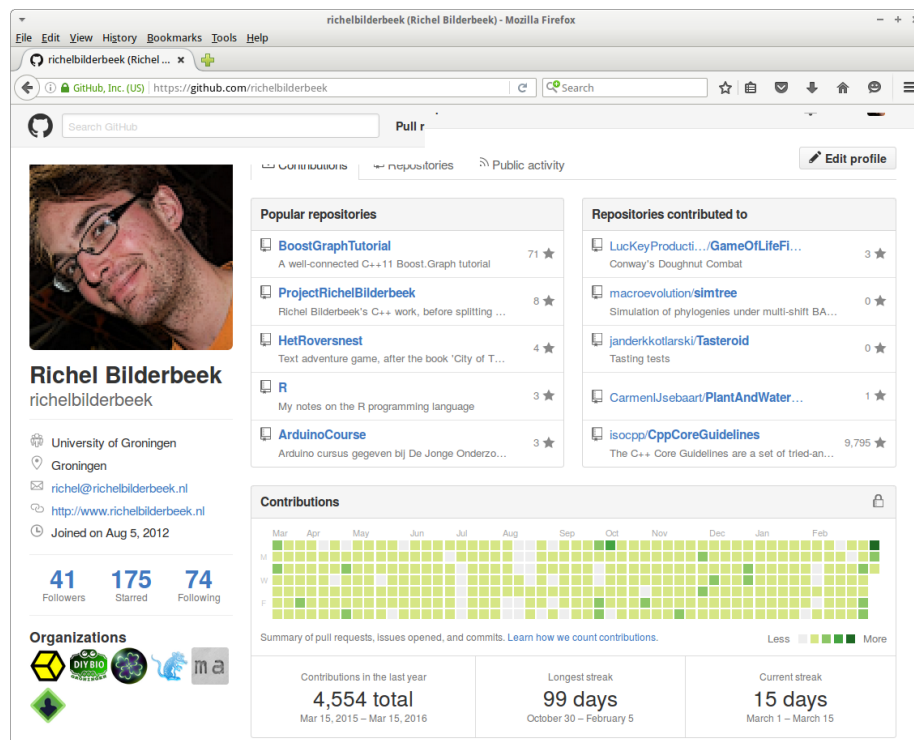


Figure 4: A GitHub profile page

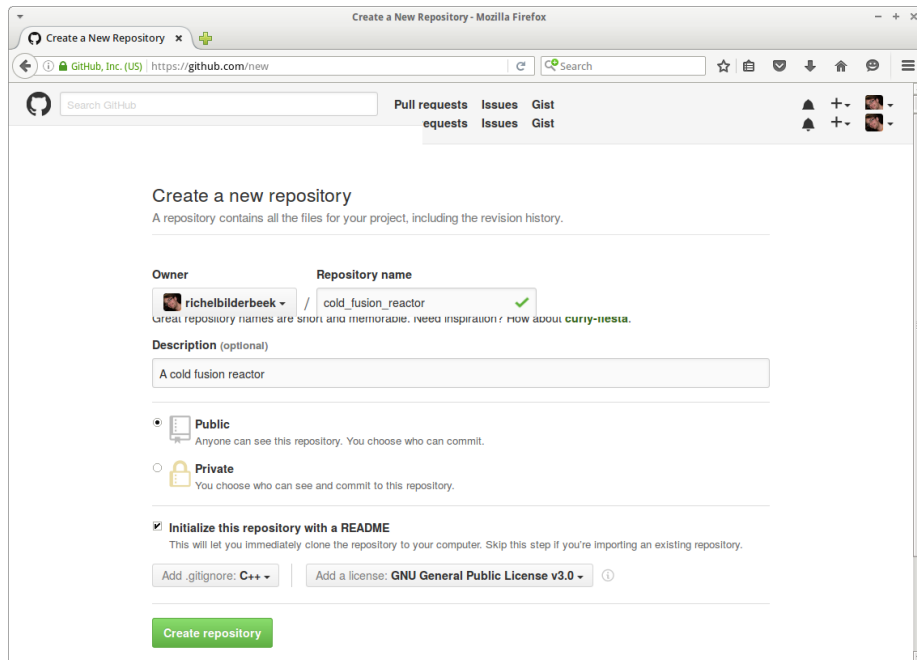


Figure 5: Create a GitHub repository

Replace '[your\_name]' and '[your\_repository]' by your GitHub username and the repository name. For example, to download this tutorial:

```
git clone https://github.com/richelbilderbeek/travis_cpp_tutorial
```

A new folder called '[your\_repository]' is created where you should work in.

## 2.3 Create a Qt Creator project

**What is Qt Creator?** Qt Creator is a C++ IDE

**Creating a new 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



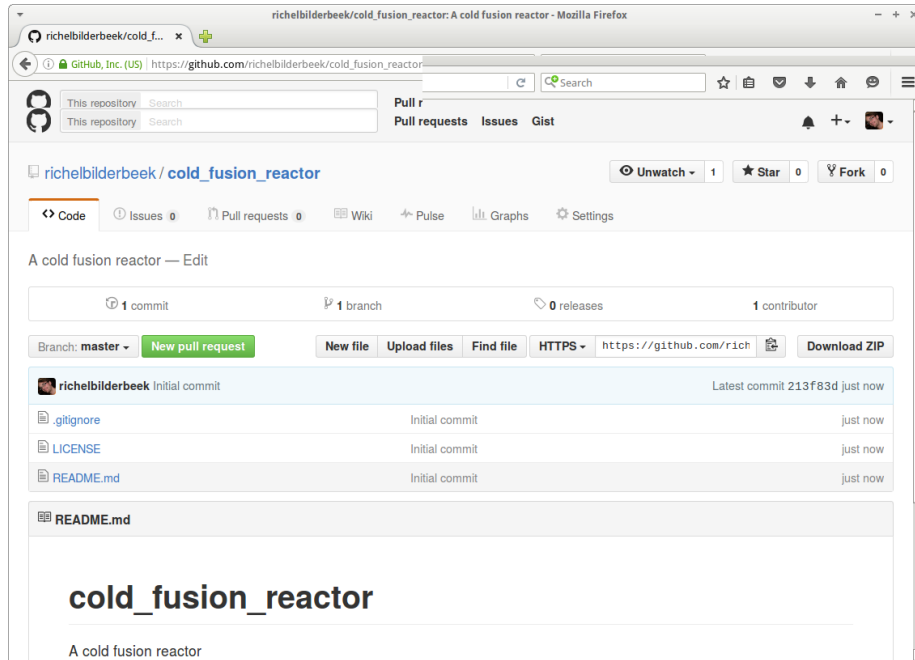


Figure 6: Created a GitHub repository



Figure 7: git logo

- Libraries: STL only
- Code coverage: none
- Source: one single file, main.cpp
- Functionality: Show the text 'Hello world' on screen

First I will show the single C++ 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. In more details:

- `#include <iostream>`  
Read a header file called 'iostream'
- `int main() { /* your code */ }`

The 'main' function is the starting point of a C++ program. Its body is between curly braces

- `std::cout << "Hello world\n";`

Show the text 'Hello world' on screen and go to the next line

The code is written in C++98. It does not use features from the newer C++ standards, but can be compiled under these newer 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 the following elements

- `TEMPLATE = app`  
`CONFIG += console`  
`CONFIG -= app_bundle qt`

This is a typical setup for a standard console application

- `SOURCES += main.cpp`

The file 'main.cpp' is a source file, that has to be compiled

- `QMAKE_CXXFLAGS += -Wall -Wextra -Werror`

The project is checked for all warnings. A warning is treated as an error.  
This forces you to write tidy code.

The bash build script to build this is:

---

**Algorithm 3** build.sh

---

```
#!/bin/bash
qmake
make
./travis_qmake_gcc_cpp98
```

---

This build script calls

- `#!/bin/bash`

This line indicates the script is a bash script

- `qmake`

'qmake' is called to create a makefile

- `make`

'make' is called to compile the makefile

- `./travis_qmake_gcc_cpp98`

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<sup>3</sup> file:

---

<sup>3</sup>the filename starts with a dot. This means it is a hidden 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++

- `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
- Use of Rcpp: see chapter 4.7

### 4.1 Use of C++11

In this example, the basic build (chapter 3) is extended by using C++11.

**What is C++11?** C++11 is a C++ standard

**Specifications** 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(); }
```

---

This is a C++11 version of a 'Hello world' program. The keyword 'noexcept' does not exist in C++98 and it will fail to compile. This code will compile under C++14.

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

---

**Algorithm 6** travis\_qmake\_gcc\_cpp11.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, except for:

- `QMAKE_CXX = g++-5`  
`QMAKE_LINK = g++-5`  
`QMAKE_CC = g++-5`

Set the C++ compiler, linker and C compiler to use g++ version 5, which is a newer version than currently used by default

- `QMAKE_CXXFLAGS += -std=c++11`

Compile under C++11

The bash build script to build and run this:

---

**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 some new features:

- `before_install`:
  - `sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test`
  - `sudo apt-get update -qq`

A new apt repository is added. Then the current apt repository is updated

- `install: sudo apt-get install -qq g++-5`

Install g++-5, which is a newer version of GCC than is installed by default

## 4.2 Use of C++14

In this example, the basic build (chapter 3) is extended by using C++14.

**What is C++14?** C++14 is a C++ standard.

### Specifications

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++14
- Libraries: STL only
- Code coverage: none
- Source: one single file, main.cpp

The single C++ source file used is:

---

**Algorithm 9** main.cpp

---

```
#include <iostream>

auto f() noexcept {
    return "Hello_world\n";
}

int main() {
    std::cout << f();
}
```

---

This is a simple C++14 program that will not compile under C++11.

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

---

**Algorithm 10** travis\_qmake\_gcc\_cpp14.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++14
```

---

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_cpp14
```

---

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

---

```
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

### 4.3 Adding the Boost libraries

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

**What is Boost?** Boost is a collection of C++ libraries

#### Specifications

- Build system: qmake
- C++ compiler: gcc





Figure 8: Boost logo

- 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 13** 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 14** travis\_qmake\_gcc\_cpp98\_boost.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 15** 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 16** `.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 6.1.

## 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.

**What is `gcov`?** `gcov` is a tool that works with GCC to analyse code coverage

**What is `Codecov`?** `Codecov` works nice with GitHub and give nicer reports

**Specifications** The basic build has the following specs:

- Build system: `qmake`
- C++ compiler: `gcc`
- C++ version: C++98
- Libraries: STL only
- Code coverage: none



Figure 9: Codecov logo

- Source: one single file, main.cpp

The single C++ source file used is:

---

**Algorithm 17** 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%.

There are multiple Qt Creator project files:

- the true executable: \_exe
- the executable that measures code coverage

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

---

**Algorithm 18** travis\_qmake\_gcc\_cpp98\_gcov.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 19** 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 20** .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



Figure 10: R logo

- `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

## 4.7 Adding Rcpp

In this example, the basic build (chapter 3) is extended by also using the Rcpp library/package.

**What is R?** R is a programming language.

**What is Rcpp?** Rcpp is a package that allows to call C++ code from R

**Specifications** The chapter has the following specs:

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

The build will be complex: I will show the C++ build and the R build separately

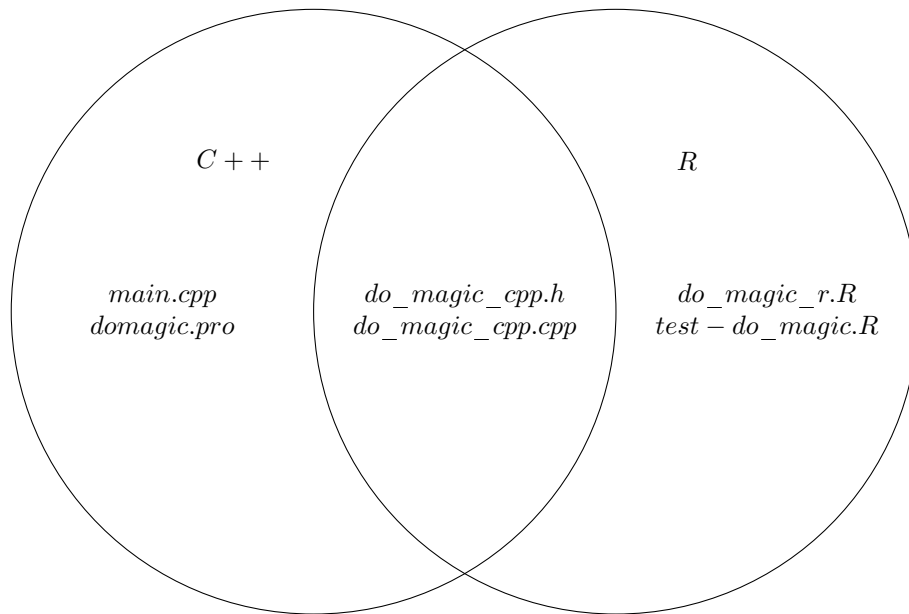


Figure 11: Venn diagram of the files uses in this build

#### 4.7.1 C++ and R: the C++ function

Both C++ and R use this function. It is called 'do\_magic\_cpp'. It is declared in the header file 'do\_magic\_cpp.h', as shown here:

---

**Algorithm 21** src/do\_magic\_cpp.h

---

```
#ifndef DO_MAGIC_CPP_H
#define DO_MAGIC_CPP_H

int do_magic_cpp(const int x);

#endif // DO_MAGIC_CPP_H
```

---

The header file consists solely of #include guards and the declaration of the function 'do\_magic\_cpp'.

The function 'do\_magic\_cpp' is implemented in the implementation file 'do\_magic\_cpp.cpp', as shown here:

---

**Algorithm 22** src/do\_magic\_cpp.cpp

---

```
#include "do_magic_cpp.h"

#include <Rcpp.h>

using namespace Rcpp;

// ' Does magic
// ' @param x Input
// ' @return Magic value
// ' @export
// [[Rcpp::export]]
int do_magic_cpp(const int x)
{
    return x * 2;
}

// ' @useDynLib domagic
// ' @importFrom Rcpp sourceCpp
```

---

This implementation file has gotten rather elaborate, thanks to Rcpp and documentation. This is because it has to be callable from both C++ and R and satisfy the requirement from both languages.

#### 4.7.2 C++: main source file

The C++ program has a normal main function:

---

**Algorithm 23** main.cpp

---

```
#include "do_magic_cpp.h"

int main()
{
    if (do_magic_cpp(2) != 4) return 1;
}
```

---

All it does is a simple test of the 'do\_magic\_cpp' function.

#### 4.7.3 C++: Qt Creator project file

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

---

**Algorithm 24** domagic.pro

---

```
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle
CONFIG -= qt

INCLUDEPATH += src
INCLUDEPATH += /home/p230198/R/x86_64-pc-linux-gnu-library/3.2/Rcpp/include
#INCLUDEPATH += /home/p230198/R/x86_64-pc-linux-gnu-library/3.2/Rcpp11/include/Rcpp
INCLUDEPATH += /usr/share/R/include/

SOURCES += \
    src/do_magic_cpp.cpp \
    main.cpp

HEADERS += \
    src/do_magic_cpp.h

LIBS += -lR
```

---

The name of the Qt Creator project file is 'domagic' as it follows the same naming as the R project. It add the R and Rcpp and src folders to its include path and links to R.

#### 4.7.4 R: the R function

The R function 'do\_magic\_r' calls the C++ function 'do\_magic\_cpp':

---

**Algorithm 25** R/do\_magic\_r.R

---

```
#' Does magic
#' @param x Input
#' @return Magic value
#' @export
#' @useDynLib domagic
#' @importFrom Rcpp sourceCpp
do_magic_r <- function(x) {
  return(do_magic_cpp(x))
}
```

---

Next to this, it is just Roxygen2 documentation



#### 4.7.5 R: The R tests

R allows for easy testing using the 'testthat' package. A test file looks as such:

---

**Algorithm 26** tests/testthat/test-do\_magic\_r.R

---

```
context("do_magic")

test_that("basic use", {
  expect_equal(do_magic_r(2), 4)
  expect_equal(do_magic_r(3), 6)
  expect_equal(do_magic_r(4), 8)

  expect_equal(domagic::do_magic_cpp(2), 4)
  expect_equal(domagic::do_magic_cpp(3), 6)
  expect_equal(domagic::do_magic_cpp(4), 8)
})
```

---

The tests call both the R and C++ functions with certain inputs and checks if the output matches the expectations.

#### 4.7.6 The build script

The bash build script is empty.

---

**Algorithm 27** build.sh

---

```
#!/bin/bash
echo "Nothing_to_be_done"
```

---

It is empty, because we set Travis CI to do the testing from R its point of view. The C++ code has to be tested manually. I do not know how to do both, if you do know, please send me an email.

#### 4.7.7 The Travis script

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

---

**Algorithm 28** .travis.yml

---

```
language: r
warnings_are_errors: true
sudo: required
```

---

This .travis.yml file is completely different than others:

- `language: r`

Travis has to work with R code

- `warnings_are_errors: true`

Travis will give an error when a warning is emitted. This enforces clean coding

- `sudo: required`

Travis will need sudo. This will slow down the build

## 5 Extending the build by two steps

The following chapter describe how to extend the build in two directions. These are:

- Use of C++11 and Boost: see chapter 5.1
- Use of C++14 and Boost: see chapter 5.2

### 5.1 C++11 and 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++11
- Libraries: STL and Boost
- Code coverage: none
- Source: one single file, main.cpp

The single C++ source file used is:

---

**Algorithm 29** main.cpp

---

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

int main() {
    boost::adjacency_list<> g;
    boost::add_vertex(g);
    if (boost::num_vertices(g) != 1) return 1;
}
```

---

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 30** travis\_qmake\_gcc\_cpp11\_boost.pro

---

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 31** build.sh

---

```
#!/bin/bash
qmake
make
./travis_qmake_gcc_cpp11_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 32** .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
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

## 5.2 C++14 and 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++14
- Libraries: STL and Boost
- Code coverage: none
- Source: one single file, main.cpp

The single C++ source file used is:

---

**Algorithm 33** main.cpp

---

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

int main() {
    boost::adjacency_list<> g;
    boost::add_vertex(g);
    if (boost::num_vertices(g) != 1) return 1;
}
```

---

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 34** travis\_qmake\_gcc\_cpp14\_boost.pro

---

---

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 35** build.sh

---

```
#!/bin/bash
qmake
make
./travis_qmake_gcc_cpp14_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 36** `.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
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

## 6 Extending the build by multiple steps

### 6.1 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

### 6.1.1 The function

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

---

**Algorithm 37** 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 38** 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

### 6.1.2 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 39** 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 -Werror -std=c
                ++11

HEADERS += my_functions.h
SOURCES += my_functions.cpp 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 40** 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) {
        BOOST_CHECK(add(1, 1) == 2);
        BOOST_CHECK(add(1, 2) == 3);
        BOOST_CHECK(add(1, 3) == 4);
        BOOST_CHECK(add(1, 4) == 5);

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

### 6.1.3 Exe build

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



---

**Algorithm 41** 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 42** 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.

#### 6.1.4 Build script

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

---

**Algorithm 43** 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'.

#### 6.1.5 Travis script

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

---

**Algorithm 44** .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

## Index

C++98, 10

Hello world, 10

make, 11

Makefile, 11

qmake, 11

Qt Creator project file, 10