Travis C++ tutorial

Richèl Bilderbeek

March 20, 2016

Contents

1 Introduction

This is a Travis C++ tutorial, version 0.2.

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.

[1]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 Tutorial style

This tutorial is aimed at the beginner.

Introduction of new terms and tools All terms and tools are introduced shortly once, by a 'What is' paragraph. This allows a beginner to have a general idea about what the term/tool is, without going in-depth. Also, this allows for those more knowledgeable to skim the paragraph.

Repetitiveness To allow skimming, most chapters follow the same structure. Sometimes the exact same wording is used. This is counteracted by referring to earlier chapters.

1.4 This tutorial

This tutorial is available online at `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

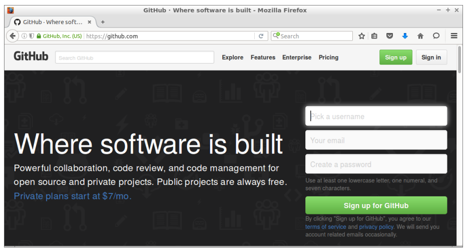
1.5 Acknowledgements

These people contributed to this tutorial:

- Kevin Ushey, for getting Rcpp11 and C++11 to work

1.6 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.



Where software is built

Powerful collaboration, code review, and code management for open source and private projects. Public projects are always free. Private plans start at $7/mo.

Use at least one lowercase letter, one numeral, and seven characters.

By clicking "Sign up for GitHub", you agree to our terms of service and privacy policy. We will send you account related emails occasionally.

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

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`.

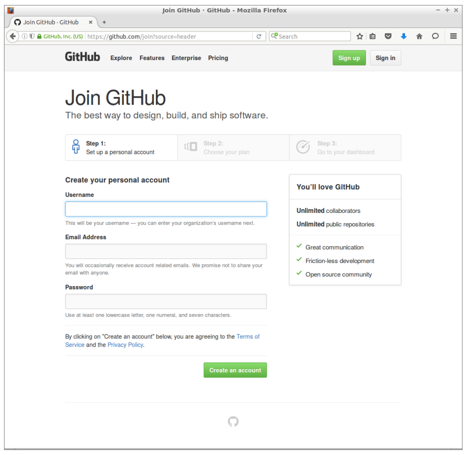
Register Before you can create a new repository, you must register. Registration is free for open source projects, with an unlimited[2] amount of public repositories.

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).

[2]the maximum I have observed is a person that has 350 repositories


Join GitHub · GitHub - Mozilla Firefox
https://github.com/join?source=header
GitHub
Explore
Features
Enterprise
Pricing
Sign up
Sign in
Create an account

Join GitHub

The best way to design, build, and ship software.

Step 1: Set up a personal account

Step 2: Choose your plan

Step 3: Go to your dashboard

Create your personal account

Username

This will be your username — you can enter your organization's username next.

Email Address

You will occasionally receive account related emails. We promise not to share your email with anyone.

Password

Use at least one lowercase letter, one numeral, and seven characters.

By clicking on "Create an account" below, you are agreeing to the Terms of Service and the Privacy Policy.

You'll love GitHub

Unlimited collaborators

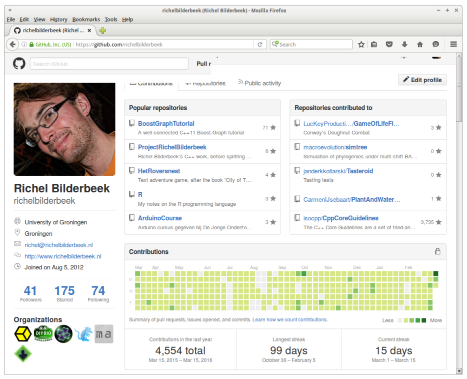
Unlimited public repositories

✓ Great communication

✓ Friction-less development

✓ Open source community

Figure 3: The join GitHub page


richelbilderbeek (Richel Bilderbeek) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://github.com/richelbilderbeek
Search GitHub
Pull r
Contributions
Repositories
Public activity
Edit profile

Richel Bilderbeek

richelbilderbeek

University of Groningen

Groningen

richel@richelbilderbeek.nl

http://www.richelbilderbeek.nl

Joined on Aug 5, 2012

41 Followers · 175 Starred · 74 Following

Organizations

Popular repositories

BoostGraphTutorial A well-connected C++11 Boost.Graph tutorial	71 ★
ProjectRichelBilderbeek Richel Bilderbeek's C++ work, before splitting …	8 ★
HetRoversnest Text adventure game, after the book 'City of T…	4 ★
R My notes on the R programming language	3 ★
ArduinoCourse Arduino cursus gegeven bij De Jonge Onderzo…	3 ★

Repositories contributed to

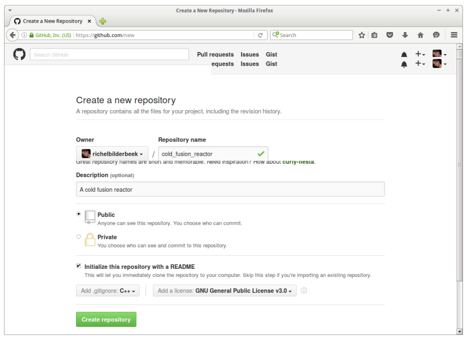
LucKeyProducti…/GameOfLifeFi… Conway's Doughnut Combat	3 ★
macroevolution/simtree Simulation of phylogenies under multi-shift BA…	0 ★
janderkkotlarski/Tasteroid Tasting tests	0 ★
CarmenIJsebaart/PlantAndWater…	1 ★
isocpp/CppCoreGuidelines The C++ Core Guidelines are a set of tried-an…	9,795 ★

Contributions

Summary of pull requests, issues opened, and commits. Learn how we count contributions.

Contributions in the last year	Longest streak	Current streak
4,554 total	99 days	15 days
Mar 15, 2015 – Mar 15, 2016	October 30 – February 5	March 1 – March 15

Figure 4: A GitHub profile page



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: richelbilderbeek / Repository name: cold_fusion_reactor ✓

Great repository names are short and memorable. Need inspiration? How about **curly-fiesta**.

Description (optional): A cold fusion reactor

[✓ Public] Anyone can see this repository. You choose who can commit.

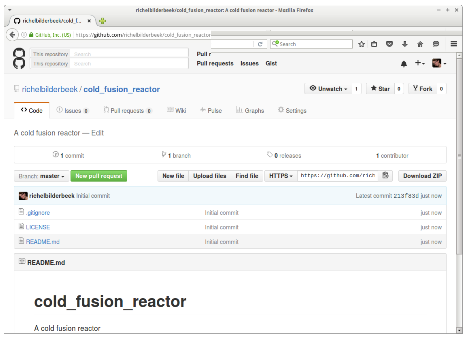
Private: You choose who can see and commit to this repository.

[✓ Initialize this repository with a README] This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: C++

Add a license: GNU General Public License v3.0

Figure 5: Create a GitHub repository



richelbilderbeek/cold_fusion_reactor: A cold fusion reactor - Mozilla Firefox

https://github.com/richelbilderbeek/cold_fusion_reactor

richelbilderbeek / cold_fusion_reactor

A cold fusion reactor — Edit

1 commit | 1 branch | 0 releases | 1 contributor

richelbilderbeek Initial commit — Latest commit 213f83d just now

.gitignore	Initial commit	just now
LICENSE	Initial commit	just now
README.md	Initial commit	just now

README.md

cold_fusion_reactor

A cold fusion reactor

Figure 6: Created a GitHub repository

Figure 7: git logo

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]
```

Replace '[your_name]' and '[your_repository]' by your GitHub username and the repository name. For example, to download this tutorial its repsoitory:

```
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 Project will have some defaults: GCC.

What is a Qt Creator project file? A Qt Creator project file contains the information how a Qt Creator project must be built. It commonly has the .pro file extension.

What is qmake? qmake is a tool to create makefiles.

What is GCC? GCC, the GNU Compiler Collection, is a collection of compilers, among other, the C++ compiler called g++.

What is g++? g++ is the C++ compiler that is part of the GCC.

What is C++98? C++98 is the first C++ standard in 1998.

What is the STL? The STL, the Standard Template Library, is the C++ standard library.

2.4 Create the build bash scripts

What is bash? 'bash' is a shell scripting language

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
- 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 -Weffc++ -Werror`

  The project is checked with all warnings ('-Wall'), with extra warnings ('-Wextra') and with the Effective C++ advices ('-Weffc++') enforced. A warning is treated as an error ('-Werror'). 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[3] 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 the GCC

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

- `script: ./build.sh`

  The script that Travis will run. In this case, it will execute the 'build.sh' bash script.

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 newer versions of C++.

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

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

- QMAKE_LINK = g++–5

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

- QMAKE_CC = g++5

  Set the 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:`

  The following events will take place before installation

- `- sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test`

  A new apt repository is added. The '-y' explicitly states that we are sure we want to do this. Without the '-y' flag, Travis will be prompted if it is sure it wants to add this repository. This would break the build.

- `- sudo apt-get update -qq`

  After adding the new apt repository, then the current repositories need to be updated updated. The '-qq' means that this happens quietly; with the least amount of output.

- `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 C++11 build in chapter 4.1, except for that it uses one different QMAKE_CXXFLAGS item:

- QMAKE_CXXFLAGS += -std=c++14

  Compile under C++14

The bash build script to build and run this:

Algorithm 11 build.sh

```
qmake
make
./travis_qmake_gcc_cpp14
```

The bash script has the same lines as the C++11 build in chapter 4.1

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 is the same as the C++11 build in chapter 4.1

Figure 8: Boost logo

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
- 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 only compile when the Boost libraries are present.

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 and run this:

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

Figure 9: Codecov logo

Build overview This will be a more complex build, consisting of two projects:

- The regular project that just runs the code
- The project that measures code coverage

The filenames are shown in this figure:

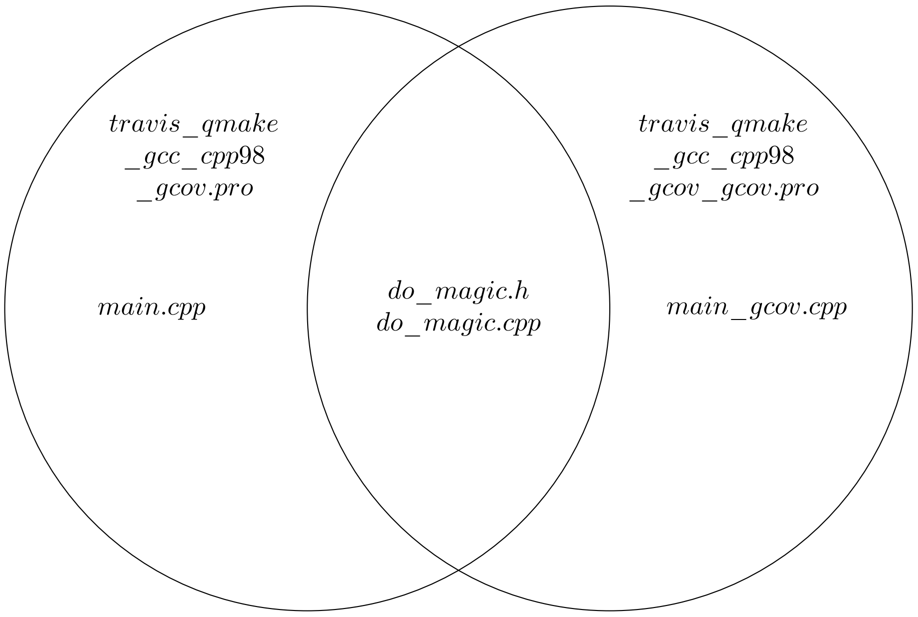

travis_qmake_gcc_cpp98_gcov.pro
main.cpp
do_magic.h
do_magic.cpp
travis_qmake_gcc_cpp98_gcov_gcov.pro
main_gcov.cpp

Figure 10: Venn diagram of the files uses in this build

Specifications The basic build has the following specs:

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

- C++ version: C++98
- Libraries: STL only
- Code coverage: yes
- Source: multiple files

Common files Both builds use the following code:

Algorithm 17 do_magic.h

```
#ifndef DO_MAGIC_H
#define DO_MAGIC_H

int do_magic(const int x);

#endif // DO_MAGIC_H
```

And its implementation:

Algorithm 18 do_magic.cpp

```
#include "do_magic.h"

int do_magic(const int x)
{
  if (x == 42)
  {
    return 42;
  }
  if (x == 314)
  {
    return 314;
  }
  return x * 2;
}
```

Normal build main function The C++ source file used by the normal build is:

Algorithm 19 main.cpp

```
#include "do_magic.h"
#include <iostream>

int main() {
  std::cout << do_magic(123) << '\n';
}
```

Normal build Qt Crator project file This normal is compiled with qmake from the following Qt Creator project file:

Algorithm 20 travis_qmake_gcc_cpp98_gcov.pro

```
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle qt
SOURCES += main.cpp do_magic.cpp
HEADERS += do_magic.h

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

Code coverage main function The C++ source file used by the normal build is:

Algorithm 21 main.cpp

```
#include "do_magic.h"

int main()
{
  if (do_magic(2) != 4) return 1;
  if (do_magic(42) != 42) return 1;
  //Forgot to test do_magic(314)
}
```

Code coverage build Qt Crator project file This normal is compiled with qmake from the following Qt Creator project file:

Algorithm 22 travis_qmake_gcc_cpp98_gcov.pro

```
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle qt
SOURCES += main_gcov.cpp do_magic.cpp
HEADERS += do_magic.h
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

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

The Qt Creator project file has two new lines:

- `QMAKE_CXXFLAGS += -fprofile-arcs -ftest-coverage`

  Let the C++ compiler add coverage information

- `LIBS += -lgcov`

  Link against the gcov library

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

Algorithm 23 build.sh

```
#!/bin/bash
echo "Normal run"
qmake travis_qmake_gcc_cpp98_gcov.pro
make
./travis_qmake_gcc_cpp98_gcov
./clean.sh
echo "Coverage run"
qmake travis_qmake_gcc_cpp98_gcov_gcov.pro
make
./travis_qmake_gcc_cpp98_gcov_gcov
gcov main_gcov.cpp
gcov do_magic.cpp
cat main_gcov.cpp.gcov
cat do_magic.cpp.gcov
```

The new step is after having run the executable,

- `gcov main_gcov.cpp`

Let gcov create a coverage report

- `cat main_gcov.cpp.gcov`

  Show the file 'main.cpp.gcov', which contains the coverage of 'main.cpp'

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

Algorithm 24 .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

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

Figure 11: R logo

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: multiple files

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

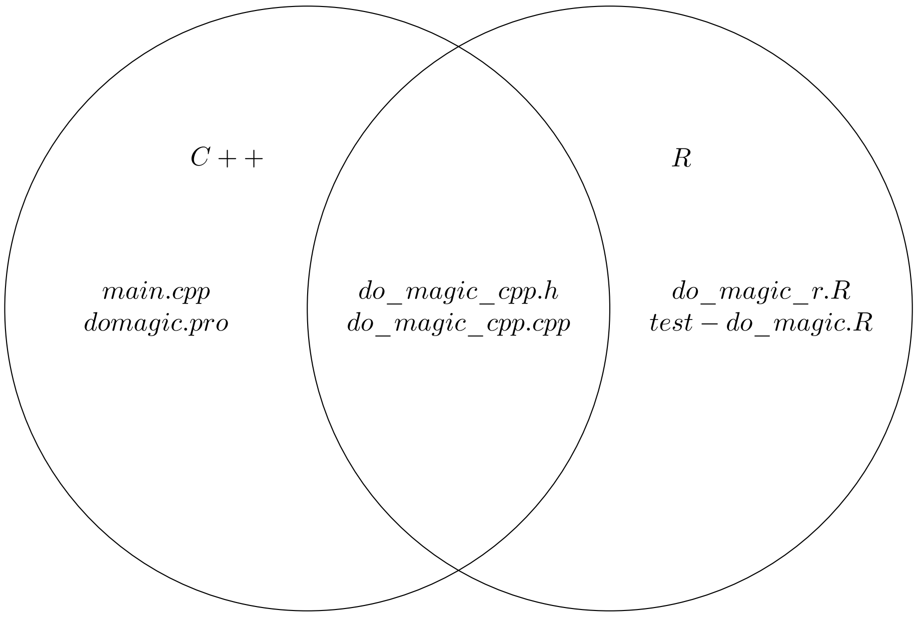


Figure 12: Venn diagram of the files uses in this build

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 25 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 26 src/do_magic_cpp.cpp

```
#include "do_magic_cpp.h"

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

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 27 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 28 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/richel/R/i686-pc-linux-gnu-library/3.2/Rcpp/include
#INCLUDEPATH += /home/p230198/R/x86_64-pc-linux-gnu-library/3.2/Rcpp11/include/Rcpp
#INCLUDEPATH += /home/richel/R/i686-pc-linux-gnu-library/3.2/Rcpp11/include/Rcpp.h

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 29 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 30 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 31 build_cpp.sh

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

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 32 .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

4.8 Adding the SFML library

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

What is SFML? SFML ('Simple and Fast Multimedia Library') is a library vey suitable for 2D game development

Figure 13: SFML logo

Specifications

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

The single C++ source file used is:

Algorithm 33 main.cpp

```
#include <SFML/Graphics.hpp>

int main()
{
  ::sf::RectangleShape shape(::sf::Vector2f(100.0,250.0))
      ;
  if (shape.getSize().x < 50) return 1;
}
```

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

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

Algorithm 34 travis_qmake_gcc_cpp98_sfml.pro

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

SOURCES += main.cpp

QMAKE_CXXFLAGS += -Wall -Wextra -Werror

LIBS += -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio
```

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_cpp98_sfml
```

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

```
language: cpp
compiler: gcc
sudo: true

before_install:
  - sudo apt-add-repository ppa:sonkun/sfml-development -y
  - sudo apt-get update -qq

install:
  - sudo apt-get install libsfml-dev

script:
 - ./build.sh
```

This .travis.yml file has one new feature:

- `install: sudo apt-get install libsfml-dev`

  This makes Travis install the needed package.

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 37 main.cpp

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

int f() noexcept {
  boost::adjacency_list<> g;
  boost::add_vertex(g);
  return boost::num_vertices(g);
}

int main() {
  if (f() != 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 38 travis_qmake_gcc_cpp11_boost.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 and run this:

Algorithm 39 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 40 .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 ...

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 41 main.cpp

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

auto f() noexcept
{
  boost::adjacency_list<> g;
  boost::add_vertex(g);
  return boost::num_vertices(g);
}

int main() {
  if (f() != 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 42 travis_qmake_gcc_cpp14_boost.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 and run this:

Algorithm 43 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 44 .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 ...

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

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.

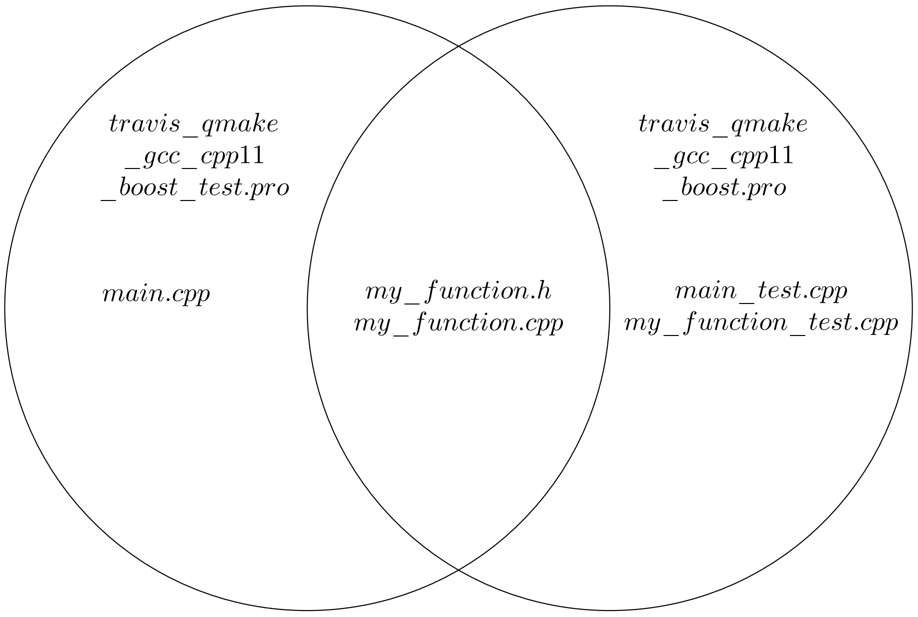

travis_qmake _gcc_cpp11 _boost_test.pro
main.cpp
my_function.h
my_function.cpp
travis_qmake _gcc_cpp11 _boost.pro
main_test.cpp
my_function_test.cpp

Figure 14: Venn diagram of the files uses in this build

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 45 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 46 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 47 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

HEADERS += my_functions.h
SOURCES += my_functions.cpp \
     main_test.cpp \
     my_functions_test.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 48 main_test.cpp

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE my_functions_test_module
#include <boost/test/unit_test.hpp>
#include <fstream>

BOOST_AUTO_TEST_CASE(inform_travis)
{
  #ifndef NDEBUG
  std::ofstream f("test_debug.txt");
  #else
  std::ofstream f("test_release.txt");
  #endif
  f << "OK\n";
}

//No main needed, BOOST_TEST_DYN_LINK creates it
```

It uses the Boost.Test framework to automatically generate a main function and test suite. An empty file is created, so Travis can verify there has been built both a debug and release mode.

Its main testing file file is shown here:

Algorithm 49 my_functions_test.cpp

```
#include <boost/test/unit_test.hpp>
#include "my_functions.h"

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);
}
```

It tests the function 'add'.

6.1.3 Exe build

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

Algorithm 50 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 51 travis_qmake_gcc_cpp11_boost_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 += 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, test and run this:

Algorithm 52 build.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp11_boost_test.pro
make debug
./travis_qmake_gcc_cpp11_boost_test
if [ ! -f exe_debug.txt ]
then
  echo "ERROR: Cannot find exe_debug.txt"
  exit 1
fi

qmake travis_qmake_gcc_cpp11_boost_test.pro
make release
./travis_qmake_gcc_cpp11_boost_test
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
```

In this script both projects are compiled in both debug and release mode. All four exectables are run.

6.1.5 Travis script

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

Algorithm 53 .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 ...

Index