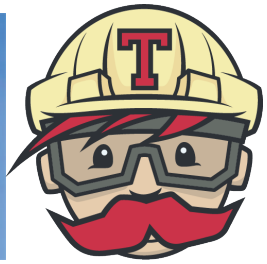
Travis C++ tutorial

Richèl Bilderbeek

April 17, 2016



Contents

1 Introduction

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

1.1 License

This tutorial is licensed under Creative Commons license 4.0.

Figure 1: Creative Commons license 4.0

All C++ code is licensed under GPL 3.0.

Figure 2: GPL 3.0

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.

[1] if not you

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 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.

From Travis to source Every build, I start from Travis CI its point of view: 'What do I have to do?'. Usually Travis CI has to call at least one build bash script. After describing the Travis file, I will show those build files. Those build files usually invoke Qt Creator project files, which in turn combine source files to executables. It may feel that the best is saved for last, but I'd disagree: this is a Travis tutorial. I also think it makes up for a better narrative, to go from big to small.

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 Collaboration

I welcome collaboration for this tutorial, especially in getting the scripts as clean as possible. If you want to help scraping off some lines, I will be happy to make you a collaborator of some GitHubs.

1.7 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 these projects. Each project contains at least one, but usually multiple files. These files can be put on your own hard disc, USB stick, or other storage devices. They could also be put at a central place, which is called a repository, so potentially others can also access these. GitHub is such a file 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 3 shows the GitHub homepage, `https://github.com`.

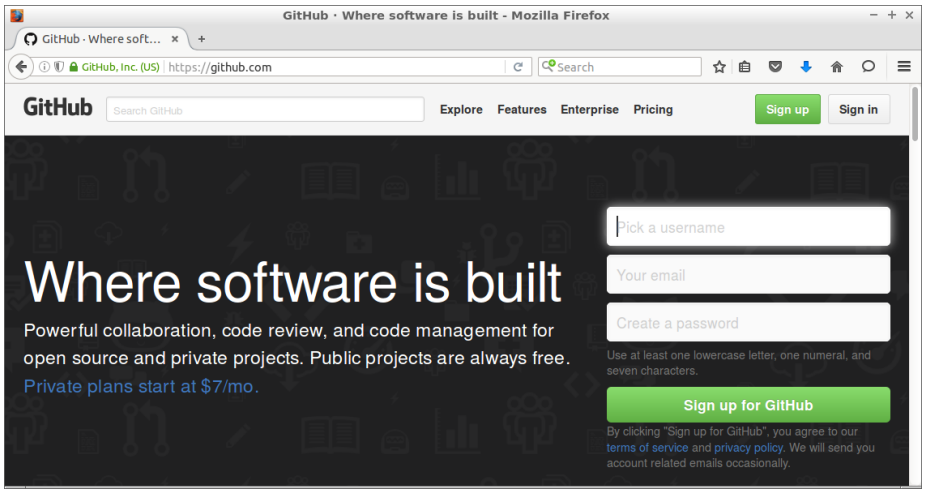


Figure 3: The GitHub homepage, `https://github.com`

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 3), click the top right button labeled 'Sign up'. This will take you to the 'Join GitHub' page (see figure 4).

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

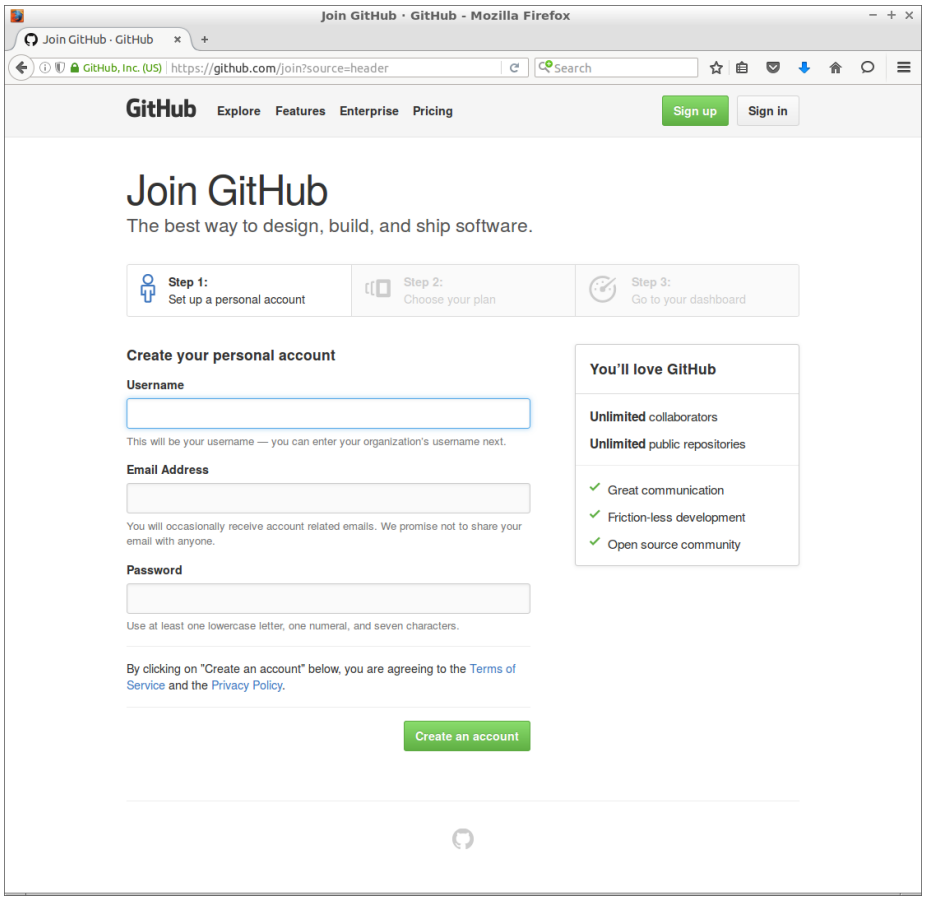


Figure 4: The join GitHub page

Filling this in should be as easy. After filling this in, you are taken to your GitHub profile page (figure 5).

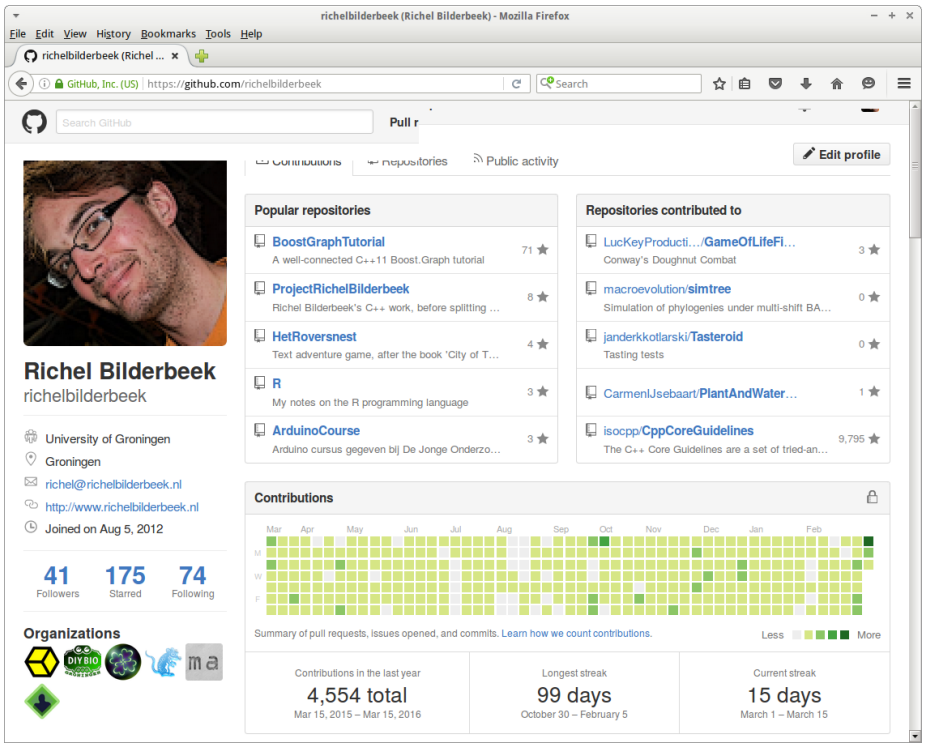


Figure 5: A GitHub profile page

Creating a repository From your GitHub profile page (figure 5), click on the plus ('Create new ...') at the top right, then click 'New repository' (figure 6).

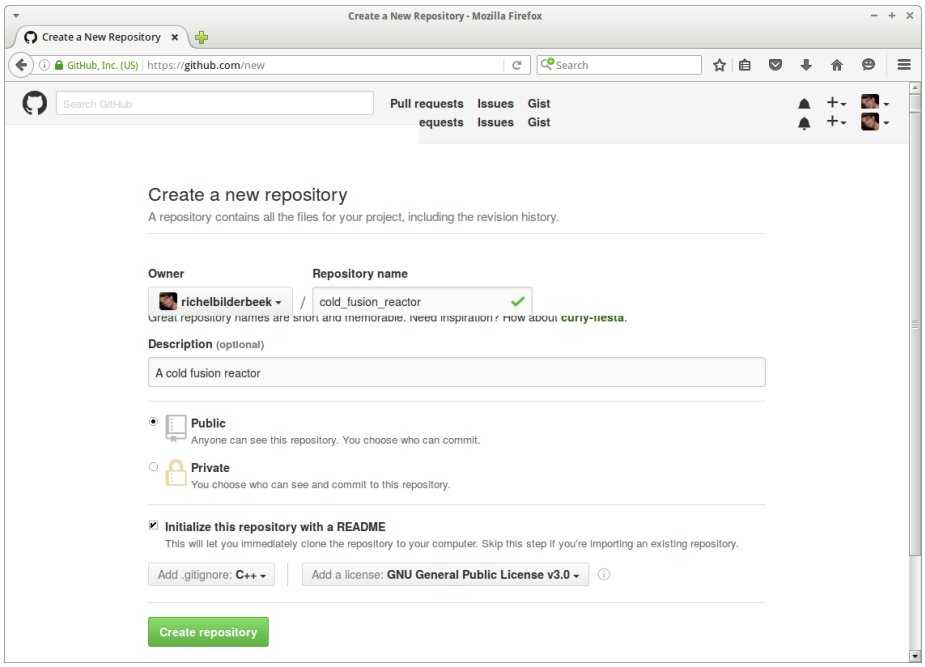


Figure 6: Create a GitHub repository

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

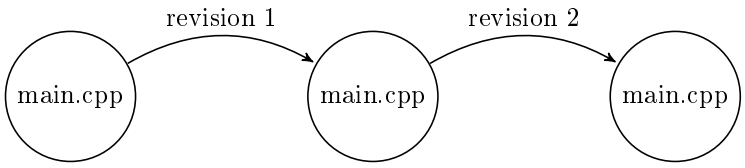


Figure 8: Multiple versions of main.cpp. git allows to always go back to each version of main

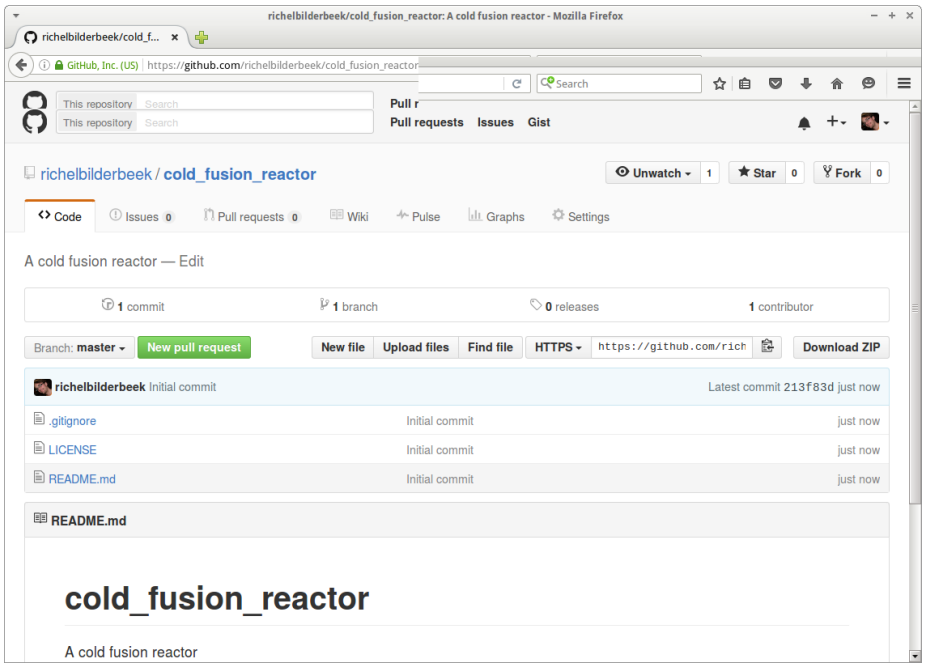


Figure 7: Created a GitHub repository

You have now created your own online version controlled repository (figure 7)!

2.2 Bring the git repository to your local computer

What is git? git is a version control system. It allows you keep a history of a file its content in time. It is the more convenient alternative of making copies before each modification.

Figure 9: git logo

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. A new folder called '[your_repository]' is created where you should work in. For example, to download this tutorial its repository to a folder called 'travis_cpp_tutorial':

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

2.3 Create a Qt Creator project

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

Figure 10: Qt creator logo

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.

Two big circles: 'C++ Project' and 'executable'
Within first circle: two smaller circles: .cpp and .h
Arrow from first to second circle with text 'compiler, linker'

Figure 11: Overview of converting a C++ project to an executable

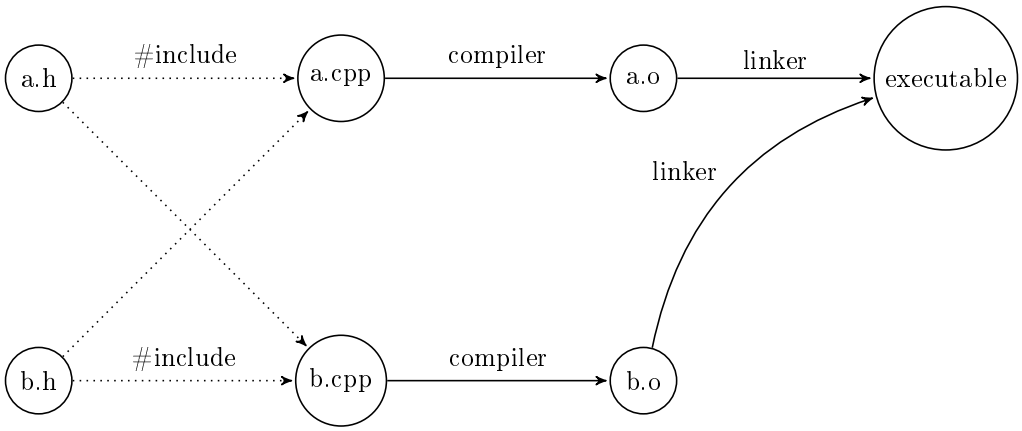


Figure 12: From files to executable. The compiler converts source (.cpp) files to object (.o) files. The linker uses these object files to create one executable

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

Two upper circles: '.pro' -> 'Makefile'
Two lower circles: '.cpp' and '.h', both -> to .pro, both dotted line to 'Makefile'

Figure 13: What qmake does

What is make? make is a tool that reads a makefile and creates an executable

'Makefile' –[make]> 'executable'

Figure 14: What make does

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

Figure 15: GCC logo

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

This basic build consists of a 'Hello World' program, written in C++98. It uses the Qt Creator default settings: Qt Creator will create a Qt Creator project file, which in turn will use GCC.

- What is a C++98 'Hello world' program? See chapter 3.1
- The Travis build file. See chapter 3.2
- The build script. See chapter 3.3
- The Qt Creator project file. See chapter 3.4
- The source file. See chapter 3.5

3.1 What is a C++98 'Hello world' program?

A 'Hello World' program shows the text 'Hello world' on the screen. It is a minimal program. Its purpose is to show that all machinery is in place to create an executable from C++ source code.

A listing of a 'Hello world' program is shown at algorithm 4. Here I go through each line:

- `#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

3.2 The Travis file

Travis CI is set up by a file called '.travis.yml'. The filename starts with a dot, which means it is a hidden file on UNIX systems. The extension 'yml' is an abbreviation of 'Yet another Markup Language'.

The '.travis.yml' file to build and run a 'Hello world' program looks like this:

Algorithm 1 .travis.yml

```
language: cpp
compiler: gcc
script:
  - ./build.sh
  - ./travis_qmake_gcc_cpp98
```

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 (What is GCC? See chapter 2.3)

- `script:`
  - `./build.sh`
  - `./travis_qmake_gcc_cpp98`

  The script that Travis will run. In this case, it will execute the 'build.sh' bash script, that should build the excutable. Then, the (hopefully) created executable called 'travis_qmake_gcc_cpp98' is run

This build script can fail in in two places:

1. The bash script can fail, which is discussed in chapter 3.3
2. The executable can return an error code. A 'Hello World' program is intended to return the error code for 'everything went fine'. Other programs in this tutorial return error codes depending on test cases. It may also be that dynamically linked libraries cannot be found, which crashes the program at startup

3.3 The build bash script

The bash build script used to build the executable of a 'Hello world' program looks like this:

Algorithm 2 build.sh

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

This build script calls:

- #!/bin/bash

  This line indicates the script is a bash script. The '#!', (also called the 'shebang') is a directive to use the executable at the absolute path following it. In this script, 'bash' is used, which resides in the '/bin' folder

- qmake

  'qmake' is called to create a makefile (What is 'qmake'? See chapter 2.3) from the only Qt Creator project file. In this build, the name of this project file is ommitted, as there is only one, but there are chapters in this tutorial where the project name is mentioned explicitly. Note that currently, qmake uses Qt4 (What is Qt4? see chapter 4.10.1)

- make

  'make' is called to compile the makefile (What is 'make'? See chapter 2.3). In this build, 'make' is called without any arguments, but there are chapters in this tutorial where 'make' is called with arguments

This bash script can fail in two places:

1. If the Qt Creator project file is incorrectly formed, 'qmake' will fail, and as it cannot create a valid makefile
2. If the Qt Creator project file is incomplete (for example: by omitting libraries), 'make' will fail. 'qmake' has created a makefile, after which 'make' finds out that it cannot create an executable with that makefile

3.4 Qt Creator project file

The following Qt Creator project file is used in this 'Hello world' build:

Algorithm 3 travis_qmake_gcc_cpp98.pro

```
SOURCES += main.cpp
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror
```

This Qt Creator project file has the following elements:

- 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++ [1] advices ('-Weffc++') enforced. A warning is treated as an error ('-Werror'). This forces you (and your collaborators) to write tidy code.

3.5 C++ source file

The single C++ source file used in this 'Hello world' build is:

Algorithm 4 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. See 3.1 for a line-by-line explanation. The code is written in C++98 (What is C++98? See chapter 2.3). 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.

4 Extending the build by one step

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

- Use a debug and release build: see chapter 4.1
- Use of C++11: see chapter 4.2
- Use of C++14: see chapter 4.3
- Use of Boost: see chapter 4.4
- Use of Boost.Test: see chapter 4.5
- Use of clang: see chapter 4.6
- Use of gcov and Codecov: see chapter 4.7

- Use of gprof: see chapter 4.8
- Use of Qt: see chapter 4.9
- Use of Qt4: see chapter 4.9
- Use of Qt5: see chapter 4.9
- Use of QTest: see chapter 4.12
- Use of Rcpp: see chapter 4.13
- Use of SFML: see chapter 4.14
- Use of Urho3D: see chapter 4.15
- Use of Wt: see chapter 4.16

4.1 Use of debug and release build

This example shows how to use Travis to create a debug and release build.

4.1.1 What are debug and release builds?

A debug build means that the executable is created in such a way that helps in debugging it. For example, assert statements are only present in debug builds.

A release build means that the executable is created in a way that allows it to run quicker and have a smaller file size. For example, assert statements are removed from the source code in a release build.

4.1.2 The Travis file

The Travis file has to do more things now, as it has to to create and run two different builds.

Here is how that looks like:

Algorithm 5 .travis.yml

```
language: cpp
compiler: gcc

script:
  - ./build_debug.sh
  - ./travis_qmake_gcc_cpp98_debug_and_release
  - ./clean.sh
  - ./build_release.sh
  - ./travis_qmake_gcc_cpp98_debug_and_release
```

This .travis.yml file is rather self-explanatory: it builds a debug version, and runs it. After cleaning up, it builds a release version and runs it.

4.1.3 The build bash scrips

Both build modes have their own build script. They are very similar to the one described in chapter 3.3:

Algorithm 6 build_debug.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp98_debug_and_release.pro
make debug
```

Algorithm 7 build_release.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp98_debug_and_release.pro
make release
```

The only difference is the added extra parameter to 'make', which is 'debug' for the debug build, and 'release' for the release build.

4.1.4 The Qt Creator project file

The Qt Creator project file has to allow for the two different builds. It does so as follows:

Algorithm 8 travis_qmake_gcc_cpp98_debug_and_release.pro

```
SOURCES += main.cpp

# Debug and release mode
CONFIG += console debug_and_release
CONFIG(release, debug|release) {
  DEFINES += NDEBUG
}
```

Next to setting 'main.cpp' as the only source file, these lines are new:

- `CONFIG += console debug_and_release`

  Create a debug and release makefiles

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

In the release makefile only, the preprocessor symbol 'NDEBUG' is #defined. This, among others, will remove all assert statements

4.1.5 The source files

This build uses a 'Hello world'-like program that shows and proves the mode in which it is built:

Algorithm 9 main.cpp

```
#include <cassert>
#include <iostream>

int main() {
  #ifdef NDEBUG
   std::cout << "Release mode" << '\n';
   assert(1==2);
  #else
   std::cout << "Debug mode" << '\n';
   assert(1+1==2);
  #endif
}
```

It will show in text the build type. Next to this, an assert is called. In release mode, the known-to-be-false assert statement is removed. In debug mode, the known-to-be-true assert statement is left in.

4.2 Use of C++11

In this example, the basic build (chapter 3) is extended by using C++11, instead of C++98.

4.2.1 What is C++11?

C++11 is the C++ standard formalized in 2011. Its working title was C++0x, as then it was assumed that the standard would be finished in 200x. C++11 is fully backwards compatible with C++98. One of the major new features of C++11 is the introduction of move semantics, which results in faster run-time code, by possibly reducing needless copies of objects.

In my examples, I typically use the C++11 'noexcept' keyword (What is noexcept? See chapter 4.2.2).

4.2.2 What is noexcept?

'noexcept' is a C++11 keyword. It is a modifier that specifies that a (member) function will not throw an exception. Would that function throw an exception anyhow, the program is terminated.

4.2.3 The Travis file

The default Travis CI setup is not sufficient to use C++11 (yet). Travis CI by default uses a LTS ('Long Term Stable') repository, as these is the most stable and reliable. The version of g++ in that repository is version 4.6.3, whuch does not support C++11. To use C++11, we will first add a fresher (less stable) repository. Then we can install g++-5,that does support C++11.

Here is how that looks like:

Algorithm 10 .travis.yml

```
sudo: require
language: cpp
compiler: gcc

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

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

script:
  - ./build.sh
  - ./travis_qmake_gcc_cpp11
```

This .travis.yml file has some new features:

- `sudo: require`

  For this build, we need super user rights. When you need super user rights, the build will be slower.

- `before_install:`

  The following events will take place before installation

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

  A new apt repository is added. The '–yes' explicitly states that we are sure we want to do this. Without the '–yes' 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

In the script, the code is built and then run.

4.2.4 The build bash scrips

The bash build script is identical to the basic build script, as described in chapter 3.3:

Algorithm 11 build.sh

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

4.2.5 The Qt Creator project file

The Qt Creator project file by default calls 'g++' with its default C++ standard. In this build, we will have to let it call g++-5 with the C++11 standard:

Algorithm 12 travis_qmake_gcc_cpp11.pro

```
# Project files
SOURCES += main.cpp

# Compile at high warning levels, a warning is an error
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

# C++11
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

Except for this, all is just the same.

4.2.6 The source files

This build uses a 'Hello world'-like program that uses C++11:

Algorithm 13 main.cpp

```
#include <iostream>

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

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

It will show the text 'Hello world' on screen.

The keyword 'noexcept' (What is noexcept? See chapter 4.2.2) does not exist in C++98 and it will fail to compile. This code will compile under newer versions of C++.

4.3 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 that was formalized in 2014. It is fully backwards compatible with C++11 and C++98. It does not have any major new features, and mostly extends C++11 features.

In my examples, I usually add digit seperators: instead of '1000', in C++14 one can write '1'000', using a single quote as a seperator. This will not compile in C++11.

4.3.1 The Travis file

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

Algorithm 14 .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
  - ./travis_qmake_gcc_cpp14
```

This .travis.yml file is the same as the C++11 build in chapter 4.2.

4.3.2 The build bash scrips

The bash build script to build and run this:

Algorithm 15 build.sh

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

The bash script is identical to the basic build script (see chapter 3.3)

4.3.3 The Qt Creator project files

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

Algorithm 16 travis_qmake_gcc_cpp14.pro

```
SOURCES += main.cpp

# Compile with high warning levels, a warning is an error
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

# C++14
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 4.2, except for that it uses one different QMAKE_CXXFLAGS item:

- QMAKE_CXXFLAGS += -std=c++14

  Compile under C++14

4.3.4 The source files

The single C++ source file used is:

Algorithm 17 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.

4.4 Adding Boost

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

4.4.1 What is Boost?

Boost is a collection of C++ libraries.

Figure 16: Boost logo

4.4.2 The Travis file

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

Algorithm 18 .travis.yml

```
language: cpp
compiler: gcc

addons:
  apt:
    packages: libboost-all-dev

script:
  - ./build.sh
  - ./travis_qmake_gcc_cpp98_boost
```

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

Using packages like this avoids using sudo, which speeds up the build. Not all packages can be used as such, however, but most are.

4.4.3 The build bash scrips

The bash build script to build and run this:

Algorithm 19 build.sh

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

The bash script is identical to the basic build script as in chapter 3.3.

4.4.4 The Qt Creator project files

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

Algorithm 20 travis_qmake_gcc_cpp98_boost.pro

```
SOURCES += main.cpp
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror
```

The Qt Creator project file has the same lines as the basic project in chapter 3.4.

4.4.5 The source files

The single C++ source file used is:

Algorithm 21 main.cpp

```
#include <iostream>
#include <boost/version.hpp>

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

All the file does is display the version of Boost on the screens. It will only compile when the Boost libraries are present.

Currently, on Travis CI, the default Boost version is 1.46.1.

4.5 Adding Boost.Test

Adding only a testing framework does not work: it will not compile in C++98. Instead, this is covered in chapter 5.4.

4.6 Use of clang

In this example, the basic build (chapter 3) is compiled by the clang compiler.

4.6.1 What is Clang?

clang is a C++ compiler

Figure 17: clang logo

4.6.2 The Travis file

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

Algorithm 22 .travis.yml

```
language: cpp
compiler: gcc

addons:
  apt:
    packages: clang

script:
  - ./build.sh
  - ./travis_qmake_clang_cpp98
```

This .travis.yml file uses the package clang (without needing sudo), compiles the program and then runs it.

4.6.3 The build bash scrip

The bash build script to build this:

Algorithm 23 build.sh

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

The bash script is identical to the basic bash script as described in chapter 3.3.

4.6.4 The Qt Creator project files

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

Algorithm 24 travis_qmake_clang_cpp98.pro

```
SOURCES += main.cpp

# Compile at a high warning level, a warning is an error
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

# clang
QMAKE_CXX = clang++
QMAKE_LINK = clang++
QMAKE_CC = clang
```

The Qt Creator project file.. except for:

- QMAKE_CXX = clang++

  Set the C++ compiler to use clang++

- QMAKE_LINK = clang++

  Set the C++ linker to use clang++

- QMAKE_CC = clang

  Set the C compiler to use clang

4.6.5 The source files

The single C++ source file used is:

Algorithm 25 main.cpp

```
#include <iostream>

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

This is just a 'Hello world' program, as discussed in detail in chapter .

4.7 Adding gcov and Codecov

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

4.7.1 What is gcov?

gcov is a tool that works with GCC to analyse code coverage

4.7.2 What is Codecov?

Codecov works nice with GitHub and give nicer reports

Figure 18: Codecov logo

Here is an example of a code coverage report, which is generated by this example:

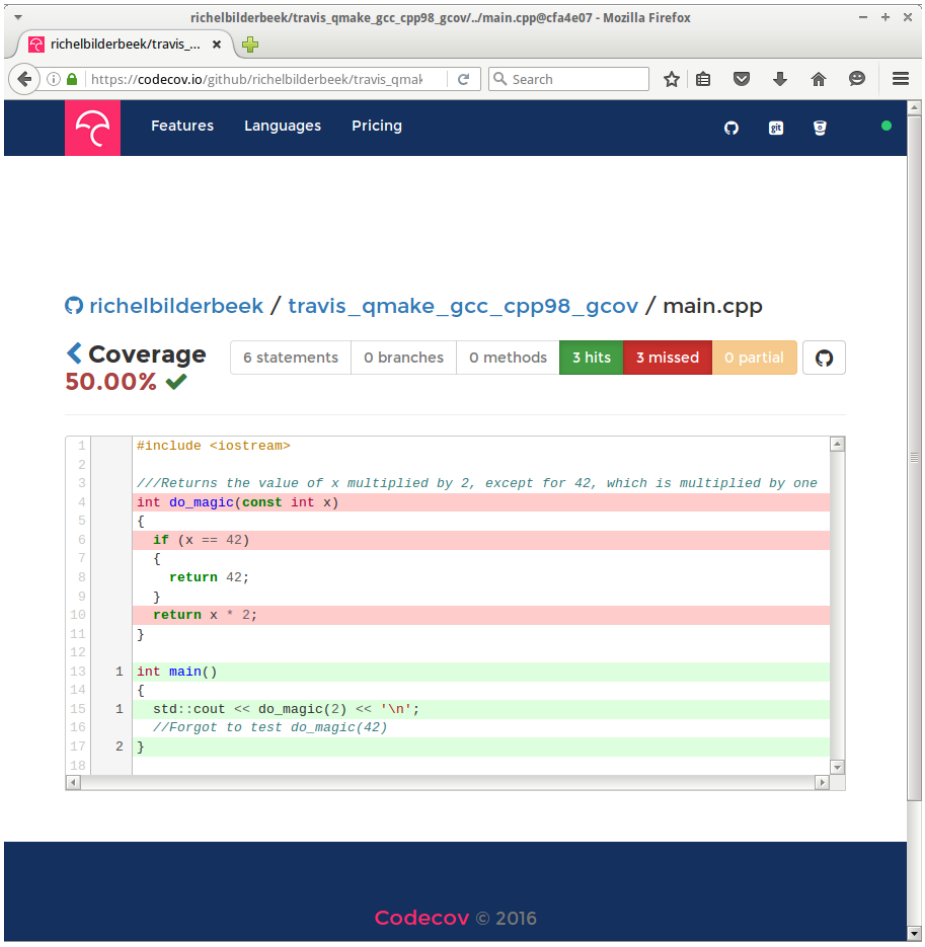


Figure 19: Codecov report of this build

4.7.3 The Travis file

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

Algorithm 26 .travis.yml

```
sudo: require
language: cpp
compiler: gcc

before_install:
 - sudo pip install codecov

script:
 - ./build.sh
 - ./travis_qmake_gcc_cpp98_gcov
 - ./get_code_cov.sh
 - codecov
```

This .travis.yml file has some new features:

- `sudo: require`

  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. It is temporary workaround to use sudo, as sudo should not be needed. This is seen as a bug, is known and solved. It just has to seep through to the Travis CI GNU/Linux distro.

- `after_success: codecov`

  After the script has run successfully, codecov is called

The code coverage performed in this build mismatches with the goals of code coverage. One of these goals is to test for unused ('dead') code. Code coverage fits better within a debug build, where all functions are tested with valid and invalid input. Chapter 6.1 shows a build in which code coverage is tested in debug mode only.

4.7.4 The build bash scrips

The bash build script to build this:

Algorithm 27 build.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp98_gcov.pro
make
```

The bash script is identical to the basic bash script as described in chapter 3.3.

The bash build to measure the code coverage:

Algorithm 28 get_code_cov.sh

```
#!/bin/bash
for filename in `find . | egrep '\.cpp'`;
do
  gcov -n -o . $filename > /dev/null;
done
```

This script uses gcov on all implementation files.
Going into a bit more detail on the new lines:

- ```
  for filename in `find . | egrep '\.cpp'`;
  do
    gcov -n -o . $filename > /dev/null;
  done
  ```

  Find all filenames (in this folder and its subfolder) that end with '.cpp'. For each of these filenames, let 'gcov' work on it. The '-n' flag denotes 'no output please'. Because there is still some output, this output is sent to the void of '/dev/null'. The '-o .' means that the object files are in the same folder as this script

4.7.5 The Qt Creator project files

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

Algorithm 29 travis_qmake_gcc_cpp98_gcov.pro

```
SOURCES += main.cpp

# Compile with a high warning level, a warning is an error
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

4.7.6 The source files

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

Algorithm 30 main.cpp

```
#include <iostream>

///Returns the value of x multiplied by 2,
/// except for 42, which is multiplied by one
int do_magic(const int x) {
  if (x == 42) {
    return 42;
  }
  return x * 2;
}

int main() {
  std::cout << do_magic(2) << '\n';
  //Forgot to test do_magic(42)
}
```

It defines a function called 'do_magic'. It is called for the value two, but not for the value 42. Due to this, we expect to see an incomplete code coverage. And this is indeed detected, as shown in figure 19.

4.8 Adding profiling

4.9 Adding the Qt library

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

What is Qt? Qt (pronounce 'cute') is a library to create C++ GUI's.

Figure 20: Qt logo

At this moment, there are two versions of Qt: Qt4 and Qt5. The GNU/Linux version Travis CI uses has Qt4. When this GNU/Linux distro changes, Qt5 will be the new (next) default.

4.10 Adding the Qt4 library

4.10.1 What is Qt4?

Qt4 is version 4 of the Qt library (What is Qt? see chapter 4.9).

4.10.2 The Travis file

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

Algorithm 31 .travis.yml

```
language: cpp
compiler: gcc

# Start virtual X server, from https://docs.travis-ci.com/user/gui-and-headless-browsers/
before_script:
  - "export DISPLAY=:99.0"
  - "sh -e /etc/init.d/xvfb start"
  - sleep 3 # give xvfb some time to start

script:
 - ./build.sh
 - ./travis_qmake_gcc_cpp98_qt4
```

This .travis.yml file starts xvfb before the script. In the script, it builds the code first, before running the resulting executable.

4.10.3 What is xvfb?

xvfb is the vitual X server.

4.10.4 The build bash scrips

The bash build script to build this:

Algorithm 32 build.sh

```
#!/bin/bash
qmake-qt4
make
```

The bash script is close to the bash script of the basic build (see chapter 3.3). Instead of calling 'qmake', however, it explicitly calls 'qmake-qt4'.

4.10.5 The Qt Creator project files

This project is compiled from the following Qt Creator project file:

Algorithm 33 travis_qmake_gcc_cpp98_qt4.pro

```
QT += core gui

# Cannot use -Weffc++ with Qt4
QMAKE_CXXFLAGS += -Wall -Wextra -Werror

SOURCES += main.cpp
SOURCES += my_dialog.cpp
FORMS   += my_dialog.ui
HEADERS += my_dialog.h

RESOURCES += travis_qmake_gcc_cpp98_qt4.qrc
```

The Qt Creator project file:

- QT += core gui

  To be able to use a GUI, one needs to add 'gui' (and keep 'core') defined

- QMAKE_CXXFLAGS += -Wall -Wextra -Werror

  When working with a Qt resource file, the '-Weffc++' flag will trigger a warning

- SOURCES += main.cpp
  HEADERS += my_dialog.h
  SOURCES += my_dialog.cpp
  FORMS   += my_dialog.ui

The files that, respectively, contain the main function definition, the declaration of 'my_dialog', the implementation of 'my_dialog' and the form of 'my_dialog'

- RESOURCES += travis_qmake_gcc_cpp98_qt4.qrc

  Use a resource file. This resource file contains the picture that is on the form.

4.10.6 The source files

This project uses multiple source files.

The main function is defined as such:

Algorithm 34 main.cpp

```
#include <QApplication>
#include "my_dialog.h"

int main(int argc, char *argv[])
{
  QApplication a(argc, argv);
  my_dialog d;
  d.show();
  return a.exec();
}
```

This is a standard implementation of the main function for a graphical Qt application.

qwfq

- ```
  #include <QApplication>
  #include "my_dialog.h"
  ```

  Read the headers of, respectively, the Qt QApplication class and our custom my_dialog class

- ```
  int main(int argc, char *argv[])
  ```

  This is one of the two official versions of main. This version takes into account the arguments supplied at startup of the application. For example, would this application be called with 'travis_qmake_gcc_cpp98_qt4 hello', the value of argc ('argument count') would be two and the array argv would be (thus) of size two with strings 'travis_qmake_gcc_cpp98_qt4' and 'hello'

- ```
  QApplication a(argc, argv);
  ```
  ```

Start the QApplication class

- `my_dialog d;`
  `d.show();`

  Create an instance of my_dialog and show it

- `return a.exec();`

  Start QApplication (which handles events for my_dialog) and return an error code depending on how the application is terminated.

The declaration of my_dialog looks like this:

Algorithm 35 my_dialog.h

```
#ifndef MY_DIALOG_H
#define MY_DIALOG_H

#include <QDialog>

namespace Ui {
  class my_dialog;
}

class my_dialog : public QDialog
{
  Q_OBJECT

public:
  explicit my_dialog(QWidget *parent = 0);
  ~my_dialog();

private:
  Ui::my_dialog *ui;
};

#endif // MY_DIALOG_H
```

This header file is completely generated by Qt Creator.

- ```
  #ifndef MY_DIALOG_H
  #define MY_DIALOG_H

  //...

  #endif // MY_DIALOG_H
  ```
  ```

This is an #include guard. An #include guard ensures that this file is read only once per compilation unit. Every header file should have these [REF], although '#pragma once' is also a fine solution.

- `#include <QDialog>`

  Read the Qt QDialog header file

- ```
  namespace Ui {
    class my_dialog;
  }
  ```

  A forward-declaration of a class called 'my_dialog' within the 'Ui' namespace. Forward-declarations intend to speed up compilation.

- ```
  class my_dialog : public QDialog
  {
    //...
  };
  ```

  Create a class called 'my_dialog' which is a derived class of the Qt 'QDialog' class

- Q_OBJECT

  Macro to signify that this class uses the Qt signal and slot mechanism

- ```
  public:
    explicit my_dialog(QWidget *parent = 0);
    ~my_dialog();
  ```

  Public constructor and destructor

- ```
  private:
    Ui::my_dialog *ui;
  ```

  The private user interface (which has only been forward-declared)

The implementation of my_dialog looks like this:

Algorithm 36 my_dialog.cpp

```
#include "my_dialog.h"
#include "ui_my_dialog.h"
#include <QTimer>

my_dialog::my_dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::my_dialog)
{
    ui->setupUi(this);
    QTimer * const timer(new QTimer(this));
    connect(timer,SIGNAL(timeout()),this,SLOT(close()));
    timer->setInterval(1000);
    timer->start();
}

my_dialog::~my_dialog()
{
    delete ui;
}
```

Most of this code is generated by Qt, except for the addition of a timer that closes the dialog after one second:

- ```
  #include "my_dialog.h"
  #include "ui_my_dialog.h"
  #include <QTimer>
  ```

  Read, respectively, the declaration of my_dialog, the declaration of the user interface of my_dialog, and the declaration of the Qt QTimer class

- ```
  my_dialog::my_dialog(QWidget *parent) :
      QDialog(parent),
      ui(new Ui::my_dialog)
  {
      //...
  }
  ```

  This is a standard constructor. The base class of my_dialog, QDialog, is called with the optional 'parent' argument. The user interface is instanciated.

- ```
  ui->setupUi(this);
  ```

  Set up the user interface of my_dialog
  ```

- ```
  QTimer * const timer(new QTimer(this));
  connect(timer,SIGNAL(timeout()),this,SLOT(close()));
  timer->setInterval(1000);
  timer->start();
  ```

  Create a timer, which will be deleted by this class. Connect its 'timeout' signal to the 'close' slot of this dialog. Set the interval of the timer to a thousand milliseconds and start it.

- ```
  my_dialog::~my_dialog()
  {
    delete ui;
  }
  ```

  A standard destructor, that deletes the user interface

4.11 Adding the Qt5 library

4.11.1 What is Qt5?

Qt5 is version 5 of the Qt library (What is Qt? see chapter 4.9).

4.11.2 The Travis file

Qt5 is not the default Qt version in the current Travis CI GNU/Linux distro.

Thanks to `http://stackoverflow.com/questions/25737062/travis-ci-for-a-qt5-project#25743300` for showing how install Qt5 on Travis CI:

Algorithm 37 .travis.yml

```
sudo: require
language: cpp
compiler: gcc

before_install:
  - sudo add-apt-repository --yes ppa:ubuntu-sdk-team/ppa
  - sudo apt-get update -qq

install:
  - sudo apt-get install qtbase5-dev qtdeclarative5-dev
  - sudo apt-get install libqt5webkit5-dev libsqlite3-dev
  - sudo apt-get install qt5-default qttools5-dev-tools

# Start virtual X server, from https://docs.travis-ci.com/user/gui-and-headless-browsers/
before_script:
  - "export DISPLAY=:99.0"
  - "sh -e /etc/init.d/xvfb start"
  - sleep 3 # give xvfb some time to start

script:
  - ./build.sh
  - ./travis_qmake_gcc_cpp98_qt5
```

This .travis.yml file is an extension of when adding the Qt4 library (chapter 4.10). The new lines are:

- `sudo add-apt-repository --yes ppa:ubuntu-sdk-team/ppa`

  Add an apt repository that has Qt5

- `sudo apt-get update -qq`

  Update the current apt repositories, to be able to find Qt5

- ```
  - sudo apt-get install qtbase5-dev qtdeclarative5-dev
  - sudo apt-get install libqt5webkit5-dev libsqlite3-dev
  - sudo apt-get install qt5-default qttools5-dev-tools
  ```

  Install all Qt5 apt packagess. I put these on three lines just for readability.

4.11.3 The build bash scrips

The bash build script to build this:
  ```

Algorithm 38 build.sh

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

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

4.11.4 The Qt Creator project files

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

Algorithm 39 travis_qmake_gcc_cpp98_qt5.pro

```
QT       += core gui widgets

# Use highest warning level, a warning is an error.
# Cannot use -Weffc++ with Qt5
QMAKE_CXXFLAGS += -Wall -Wextra -Werror

SOURCES += main.cpp

SOURCES += my_qt5_dialog.cpp
FORMS   += my_qt5_dialog.ui
HEADERS += my_qt5_dialog.h

RESOURCES += travis_qmake_gcc_cpp98_qt5.qrc
```

The Qt Creator project file is similar to the one needed for the Qt4 library (chapter 4.10), except for:

- QT += core gui widgets

  Add 'core', 'gui' and (new) 'widgets' to the Qt configuration. One of the differences between Qt4 and Qt5 is that part of what was 'gui' has been moved to 'widgets'.

4.11.5 The source files

This project uses multiple source files.

The main function is defined as such:

Algorithm 40 main.cpp

```
#include <QApplication>
#include "my_qt5_dialog.h"

int main(int argc, char *argv[])
{
  QApplication a(argc, argv);
  my_qt5_dialog d;
  d.show();
  return a.exec();
}
```

This main function definition is close to identical to that of when using Qt4 (see chapter 4.10).

The declaration of my_qt5_dialog is as such:

Algorithm 41 my_qt5_dialog.h

```
#ifndef MY_DIALOG_H
#define MY_DIALOG_H

#include <QDialog>

namespace Ui {
  class my_qt5_dialog;
}

class my_qt5_dialog : public QDialog
{
  Q_OBJECT

public:
  explicit my_qt5_dialog(QWidget *parent = 0);
  ~my_qt5_dialog();

private:
  Ui::my_qt5_dialog *ui;
};

#endif // MY_DIALOG_H
```

This header file is also close to identical to that of when using Qt4 (see chapter 4.10).

The implementation of my_qt5_dialog:

Algorithm 42 my_qt5_dialog.cpp

```
#include "my_qt5_dialog.h"
#include "ui_my_qt5_dialog.h"
#include <QTimer>

my_qt5_dialog::my_qt5_dialog(QWidget *parent) :
  QDialog(parent),
  ui(new Ui::my_qt5_dialog)
{
  ui->setupUi(this);
  QTimer * const timer(new QTimer(this));
  connect(
    timer,&QTimer::timeout,
    this, &my_qt5_dialog::close
  );
  timer->setInterval(1000);
  timer->start();
}

my_qt5_dialog::~my_qt5_dialog()
{
  delete ui;
}
```

This implementation file is also close to identical to that of when using Qt4 (see chapter 4.10), except for this line:

- ```
  connect(
    timer, &QTimer::timeout,
    this, &my_qt5_dialog::close
  );
  ```

  This is the Qt5 syntax of connecting QTimer its 'timeout' slot to my_qt5_dialog its 'close' slot. This syntax will not compile with Qt4. The new syntax has the benefit that during compilation it can be checked that the signals and slots exist (Qt4 emits a warning at runtime).

4.12 Adding QTest

One cannot use QTest without Qt. Because this thus takes two steps, this is covered in chapter 5.2.
  ```

4.13 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.

Figure 21: R logo

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

4.13.1 Build overview

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

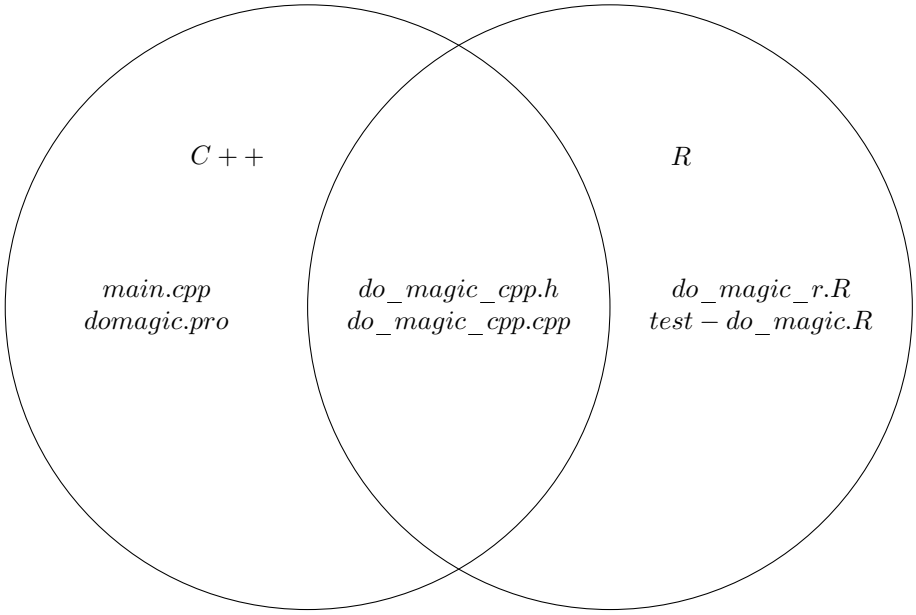


Figure 22: Venn diagram of the files uses in this build

4.13.2 The Travis file

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

Algorithm 43 .travis.yml

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

before_install:
  - sudo add-apt-repository -y ppa:marutter/rrutter # For R
  - sudo apt-get update -qq

install:
  - sudo apt-get install -qq r-base r-base-dev # For R
  - sudo apt-get install -qq lyx     # For pdflatex
  - sudo apt-get install -qq texlive # For pdflatex

script:
 # C++
 - ./build_cpp.sh
 - ./domagic
 # R wants all non-R files gone...
 - ./clean.sh
 - sudo Rscript install_r_packages.R
 - rm .gitignore
 - rm src/.gitignore
 - rm .travis.yml
 - rm -rf .git
 - rm -rf ..Rcheck
 # Now R is ready to go
 - R CMD check .

after_failure:
 # fatal error: Rcpp.h: No such file or directory
 - find / -name 'Rcpp.h'
 # R logs
 - cat /home/travis/build/richelbilderbeek/travis_qmake_gcc_cpp98_rcpp/..Rcheck/00install.ou
```

This .travis.yml file is longer than usual, as it both compiles and runs the C++ and R code.

4.13.3 The build bash scrips

The C++ build script:

Algorithm 44 build_cpp.sh

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

The bash script has the same lines as the basic project in chapter 3. This R build script installs the required R packages:

Algorithm 45 build_cpp.sh

```
install.packages("Rcpp", repos = "http://cran.uk.r-
    project.org")
install.packages("knitr", repos = "http://cran.uk.r-
    project.org")
install.packages("testthat", repos = "http://cran.uk.r-
    project.org")
install.packages("rmarkdown", repos = "http://cran.uk.r-
    project.org")
```

4.13.4 The Qt Creator project files

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

Algorithm 46 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 += /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.

4.13.5 The C++ and R source files

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 47 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 48 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.

4.13.6 The C++-only source files

The C++ program has a normal main function:

Algorithm 49 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.

4.13.7 The R-only source files

The R function 'do_magic_r' calls the C++ function 'do_magic_cpp':

Algorithm 50 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

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

Algorithm 51 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.

4.14 Adding the SFML library

In this example, the basic build (chapter 3) is extended by also using the SFML library. The result will be a simple graphical display as shown in figure 23:

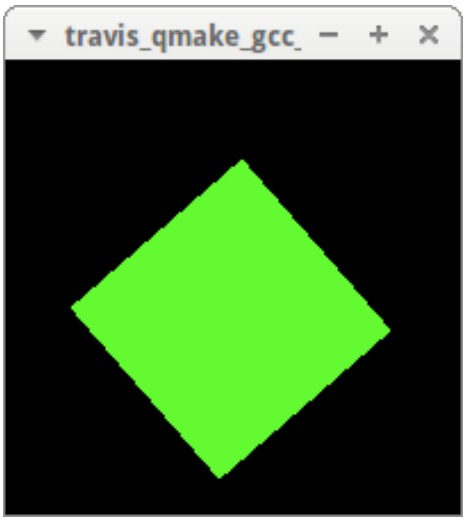


Figure 23: travis_qmake_gcc_cpp98_sfml screenshot

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

Figure 24: SFML logo

4.14.1 The Travis file

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

Algorithm 52 .travis.yml

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

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

install:
  - sudo apt-get install libsfml-dev

# Start virtual X server, from https://docs.travis-ci.com/user/gui-and-headless-browsers/
before_script:
  - "export DISPLAY=:99.0"
  - "sh -e /etc/init.d/xvfb start"
  - sleep 3 # give xvfb some time to start

script:
 - ./build.sh
 - ./travis_qmake_gcc_cpp98_sfml
```

This .travis.yml file has one new feature:

- `sudo apt-add-repository ppa:sonkun/sfml-development --yes`

  Add an apt repository for a fresh version of SFML

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

  This makes Travis install the needed package

4.14.2 The build bash scrips

The bash build script to build this:

Algorithm 53 build.sh

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

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

4.14.3 The Qt Creator project files

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

Algorithm 54 travis_qmake_gcc_cpp98_sfml.pro

```
SOURCES += main.cpp

# Compile with high warning levels, a warning is an error
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

# SFML
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, except for:

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

  Link to the SFML libraries

4.14.4 The source files

The single C++ source file used is:

Algorithm 55 main.cpp

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

int main() {
  sf::RenderWindow window(
    sf::VideoMode(200, 200),
    "travis_qmake_gcc_cpp98_sfml"
  );
  double angle = 0.0;

  while (window.isOpen()) {
    sf::Event event;
    while (window.pollEvent(event))
    {
      if (event.type == sf::Event::Closed)
      {
        window.close();
      }
    }
    window.clear(sf::Color::Black);

    sf::RectangleShape r(sf::Vector2f(100.0,100.0));
    r.setOrigin(sf::Vector2f(40.0,40.0));
    r.setPosition(100.0,100.0);
    r.rotate(angle);
    r.setFillColor(sf::Color(100, 250, 50));
    window.draw(r);

    window.display();

    angle += 0.01;
    if (angle > 100.0) break;
  }
}
```

It draws a rotating rectangle by incrementing the variable 'angle'. After this variable reaches a certain value, the application is terminated.

The reason the application is terminated, is because it must be run on Travis CI and thus terminate without user input.

4.15 Adding the Urho3D library

In this example, the basic build (chapter 3) is extended by also using the Urho3D library.

What is Urho3D? Urho3D is a library to create C++ 3D games.

Figure 25: Urho3D logo

4.15.1 Build overview

The files will work together to create the following 3D world:

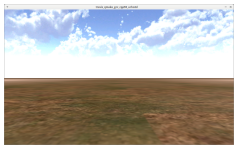


Figure 26: Screenshot of travis_qmake_gcc_cpp98_urho3d

4.15.2 The Travis file

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

Algorithm 56 .travis.yml

```
sudo: require
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
  - sudo apt-get install libx11-dev libxrandr-dev libasound2-dev libgl1-mesa-dev
  - sudo apt-get install libsdl1.2-dev libsdl-image1.2-dev libsdl-mixer1.2-dev libsdl-ttf2.(

addons:
  apt:
    sources:
    - boost-latest
    - ubuntu-toolchain-r-test
    packages:
    - gcc-5
    - g++-5
    - libboost1.55-all-dev

before_script:
  - ./build_urho3d.sh

script:
  - ./build.sh
```

This .travis.yml file has to do many things.
Note that we do not run the application.

4.15.3 The build bash scrips

The bash build script to build:

Algorithm 57 build.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp98_urho3d.pro
make
```

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

4.15.4 The Qt Creator project files

The files are compiled with qmake from the following Qt Creator project file:

Algorithm 58 travis_qmake_gcc_cpp98_urho3d.pro

```
SOURCES += \
    mastercontrol.cpp \
    inputmaster.cpp \
    cameramaster.cpp

HEADERS += \
    mastercontrol.h \
    inputmaster.h \
    cameramaster.h

# C++98
QMAKE_CXX = g++-5
QMAKE_LINK = g++-5
QMAKE_CC = gcc-5
QMAKE_CXXFLAGS += -Wall -Wextra -Werror

# Qt resources emit a warning
QMAKE_CXXFLAGS += -Wno-unused-variable

# Urho3D
INCLUDEPATH += \
    ../travis_qmake_gcc_cpp98_urho3d/Urho3D/include \
    ../travis_qmake_gcc_cpp98_urho3d/Urho3D/include/Urho3D/ThirdParty
LIBS += ../travis_qmake_gcc_cpp98_urho3d/Urho3D/lib/libUrho3D.a
LIBS += -lpthread -lSDL -ldl -lGL
```

The Qt Creator project file lists all source files, uses g++5, suppresses a warning , includes and links to multiple libraries.

4.15.5 The source files

The C++ source files are too big to show here. Their names are:

- cameramaster.h
- cameramaster.cpp
- inputmaster.h
- inputmaster.cpp
- mastercontrol.h

- mastercontrol.cpp

4.16 Adding the Wt library

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

What is Wt? Wt (pronounce 'witty') is a library to create C++ websites.

Figure 27: Wt logo

4.16.1 The Travis file

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

Algorithm 59 .travis.yml

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

This .travis.yml file has uses the package 'libboost-all-dev' and installs 'witty-dev'. It does not run the application.

4.16.2 The build bash scrips

The bash build script to build this:

Algorithm 60 build.sh

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

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

4.16.3 The Qt Creator project files

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

Algorithm 61 travis_qmake_gcc_cpp98_wt.pro

```
SOURCES += main.cpp

# Compile with high warning levels, a warning is an error
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

# Wt
LIBS += \
  -lboost_date_time \
  -lboost_filesystem \
  -lboost_program_options \
  -lboost_regex \
  -lboost_signals \
  -lboost_system
LIBS += -lwt -lwthttp
DEFINES += BOOST_SIGNALS_NO_DEPRECATION_WARNING
```

The Qt Creator project file has the same lines as the basic project in chapter 3, except for that it links to multiple libraries and suppresses a warning.

4.16.4 The source files

The single C++ source file used is:

Algorithm 62 main.cpp

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Weffc++"
#include <boost/program_options.hpp>
#include <boost/signals2.hpp>
#include <Wt/WApplication>
#include <Wt/WContainerWidget>
#include <Wt/WEnvironment>
#include <Wt/WPaintDevice>
#include <Wt/WPaintedWidget>
#include <Wt/WPainter>
#include <Wt/WPushButton>
#pragma GCC diagnostic pop

struct WtWidget : public Wt::WPaintedWidget
{
  WtWidget()
  {
    this->resize(32,32);
  }
  protected:
  void paintEvent(Wt::WPaintDevice *paintDevice)
  {
    Wt::WPainter painter(paintDevice);
    for (int y=0; y!=32; ++y)
    {
      for (int x=0; x!=32; ++x)
      {
        painter.setPen(
          Wt::WPen(
            Wt::WColor(
              ((x+0) * 8) % 256,
              ((y+0) * 8) % 256,
              ((x+y) * 8) % 256)));
        //Draw a line of one pixel long
        painter.drawLine(x,y,x+1,y);
        //drawPoint yiels too white results
        //painter.drawPoint(x,y);
      }
    }
  }
};

struct WtDialog : public Wt::WContainerWidget
{
  WtDialog()
  : m_widget(new WtWidget)
  {
    this->addWidget(m_widget);
  }
  private:
  WtDialog(const WtDialog&); //delete
  WtDialog& operator=(const WtDialog&); //delete
  WtWidget * const m_widget;
};
```

It starts a web server.

5 Extending the build by two steps

You will probably want to combine the single ingredients in the previous chapters. This will also result in more complex project setups. In this chapter, such setups will be described:

- Use of gcov in debug mode only: see chapter 5.1
- Use of Qt and QTest: see chapter
- Use of C++11 and Boost: see chapter 5.3
- Use of C++11 and Boost.Test: see chapter 5.4
- Use of C++14 and Boost: see chapter 5.12

5.1 Use of gcov in debug mode only

In this example, the C++98 build with gcov (chapter 4.7) is extended by using gcov in debug mode only.

5.1.1 Build overview

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

- A release version that just runs the code, assuming it to be correct
- A debug version that tests the code and measures code coverage

The filenames are shown in this figure:

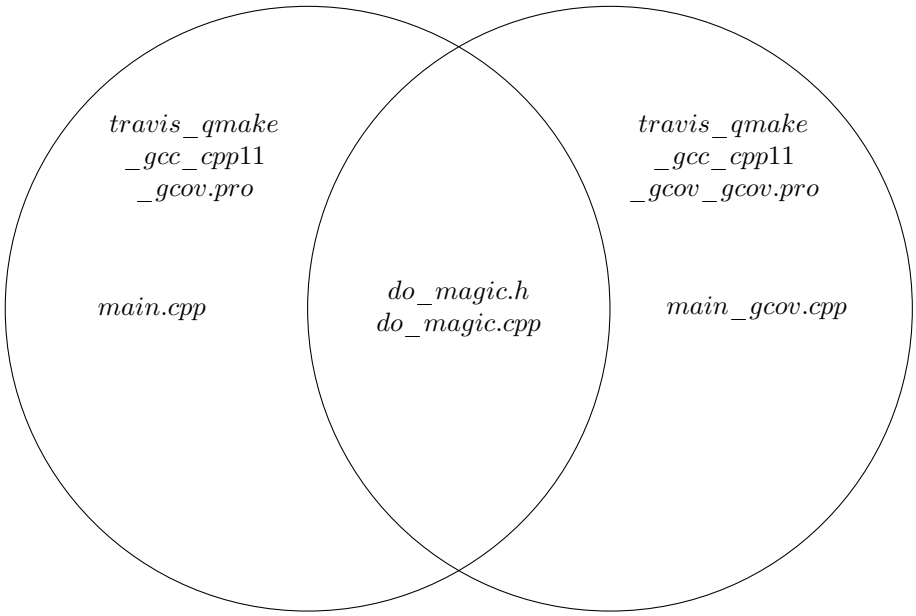


Figure 28: Venn diagram of the files uses in this build

5.1.2 The Travis file

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

Algorithm 63 .travis.yml

```
sudo: require
language: cpp
compiler: gcc
before_install: sudo pip install codecov
script:
  - ./build_debug.sh
  - ./travis_qmake_gcc_cpp98_debug_gcov_debug
  - ./get_code_cov.sh
  - codecov
  - ./clean.sh
  - ./build_release.sh
  - ./travis_qmake_gcc_cpp98_debug_gcov
```

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

5.1.3 The build bash scrips

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

Algorithm 64 build_debug.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp98_debug_gcov_debug.pro
make
```

The new step is ...
The bash build script to build this, run this and measure the code coverage:

Algorithm 65 build_release.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp98_debug_gcov.pro
make
```

This is ...

5.1.4 The Qt Creator project files

Release:

Algorithm 66 travis_qmake_gcc_cpp98_debug_gcov.pro

```
SOURCES += do_magic.cpp main.cpp
HEADERS += do_magic.h

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

Debug with gcov:

Algorithm 67 travis_qmake_gcc_cpp98_gcov.pro

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

5.1.5 The source files

Common files Both builds use the following code:

Algorithm 68 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 69 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;
}
```

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

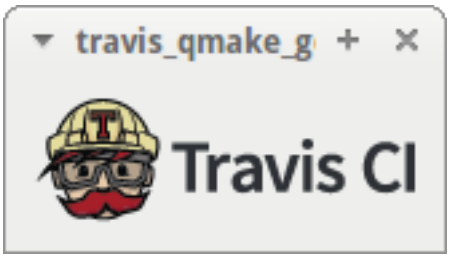


Figure 29: The application

Algorithm 70 main.cpp

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

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

Debug and gcov main function The C++ source file used by the normal build is:

Algorithm 71 main_gcov.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)
}
```

5.2 Qt and QTest

This build is about a Qt dialog that displays an image (using a Qt resource). When the key 'x' is pressed, it should close.

The release build is just that application.

The debug build tests if the application indeed closes upon a press of the 'x' key. Its primary output is test report. During the test, the dialog will show up shortly.

In this build, only one dialog is tested. For a build that has more dialogs, see chapter .

5.2.1 What is QTest?

QTest is the Qt testing framework

5.2.2 Do not use Boost.Test to test graphical Qt aplications

The Boost.Test library (see chapter 5.4) works great with console (that is: non-graphical) applications. But is is tedious to let it test graphical Qt classes.

Why is this tedious? Because Qt has its own Qt way, that works best in that way. QTest will process the QApplication event queue and have many privileges. Using Boost.Test will make you reponsible to do yourself what Qt normally does for you in the back, such as emptying the QApplication event queue. Next to this, you will have to make some member functions public (e.g. keyPressEvent) to allow your tests to use these.

5.2.3 The Travis file

Algorithm 72 .travis.yml

```
language: cpp
compiler: gcc

# Start virtual X server
before_script:
  - "export DISPLAY=:99.0"
  - "sh -e /etc/init.d/xvfb start"
  - sleep 3 # give xvfb some time to start

script:
 - ./build_test.sh
 - ./travis_qmake_gcc_cpp98_qt_qtest_test
 - ./build_normal.sh
```

Because this application uses graphics, we need to start a virtual X server on Travis CI (see `https://docs.travis-ci.com/user/gui-and-headless-browsers`), before the tests run.

In the script, the testing executable is created and run. The test results will be visible in Travis CI.

After the test, the normal executable is created. The normal executable is not run, as it requires user input. This means that on Travis CI, it would run forever, wouldn't Travis CI detect this and indicate a failure.

5.2.4 The build bash scrips

There need to be two bash scripts, one for building the testing executable, one for building the normal program. Both are as short as can be:

Algorithm 73 build_test.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp98_qt_qtest_test.pro
make
```

Algorithm 74 build_normal.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp98_qt_qtest.pro
make
```

5.2.5 The Qt Creator project files

There need to be two Qt Creator scripts, one for building the testing executable, one for building the normal program. Both are as short as can be. The only difference is that the testing project file uses 'QT += testlib'.

Test:

Algorithm 75 travis_qmake_gcc_cpp98_qt_qtest_test.pro

```
# Shared files
SOURCES += my_dialog.cpp
FORMS += my_dialog.ui
HEADERS += my_dialog.h
RESOURCES += travis_qmake_gcc_cpp98_qt_qtest.qrc

# Unique files
SOURCES += qtmain_test.cpp
SOURCES += my_dialog_test.cpp
HEADERS += my_dialog_test.h

# Qt
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

# QTest
QT += testlib
```

Normal:

Algorithm 76 travis_qmake_gcc_cpp98_qt_qtest.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

# Shared files
SOURCES += my_dialog.cpp
FORMS += my_dialog.ui
HEADERS += my_dialog.h
RESOURCES += travis_qmake_gcc_cpp98_qt_qtest.qrc

# Unique files
SOURCES += qtmain.cpp
```

5.2.6 The source files

The dialog This is the source of dialog:

Algorithm 77 my_dialog.h

```
#ifndef MY_DIALOG_H
#define MY_DIALOG_H

#include <QDialog>

namespace Ui { class my_dialog; }

class my_dialog : public QDialog {
  Q_OBJECT

public:
  explicit my_dialog(QWidget *parent = 0);
  ~my_dialog();

protected:
  void keyPressEvent(QKeyEvent *);

private:
  Ui::my_dialog *ui;
};

#endif // MY_DIALOG_H
```

The only added line, is the 'keyPressEvent'.

Algorithm 78 my_dialog.cpp

```
#include "my_dialog.h"
#include <QKeyEvent>
#include "ui_my_dialog.h"

my_dialog::my_dialog(QWidget *parent) :
  QDialog(parent),
  ui(new Ui::my_dialog) {
  ui->setupUi(this);
}

my_dialog::~my_dialog() {
  delete ui;
}

void my_dialog::keyPressEvent(QKeyEvent * e) {
  if (e->key() == Qt::Key_X) close();
}
```

Here we can see that when 'x' is pressed, the application will close.

The main function of the normal executable Most graphical Qt applications have this main function:

Algorithm 79 qtmain.cpp

```
#include <QApplication>
#include "my_dialog.h"

int main(int argc, char* argv[]) {
  QApplication a(argc, argv);
  my_dialog d;
  d.exec();
  return a.exec();
}
```

This main is given as default when creating a new graphical Qt application.

The main function of the testing executable The QTest framework collects all tests and calls these within a QTest-generated main function. This leaves us little left to write (which is awesome):

Algorithm 80 qtmain_test.cpp

```
#include <QtTest/QtTest>
#include "my_dialog_test.h"

QTEST_MAIN(my_dialog_test)
```

The class for the tests Here comes in the QTest architecture: for each test suite we will have to create a class:

Algorithm 81 my_dialog_test.h

```
#ifndef MY_DIALOG_TEST_H
#define MY_DIALOG_TEST_H

#include <QtTest/QtTest>

class my_dialog_test: public QObject
{
    Q_OBJECT
private slots:
    void close_with_x();
};

#endif // MY_DIALOG_TEST_H
```

Here we create a class called 'my_dialog_test'. The fit into the QTest framework each test suite

- must be a derived class from QObject
- the header file must include the 'QtTest' header file

where each member function is a tests.

The implementation of each test can be seen in the implementation file:

Algorithm 82 my_dialog_test.cpp

```
#include "my_dialog_test.h"
#include "my_dialog.h"

void my_dialog_test::close_with_x()
{
  my_dialog d;
  d.show();
  QVERIFY(d.isVisible());
  QTest::keyClick(&d, Qt::Key_X, Qt::NoModifier, 100);
  QVERIFY(d.isHidden());
}
```

The 'QVERIFY' macro is used by the QTest framework to do a single check, which will end up in the test report. The QTest has some priviliges, as it can directly click keys on the form, also when the 'keyPressEvent' isn't public.

5.3 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 83 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 84 travis_qmake_gcc_cpp11_boost.pro

```
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 85 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 86 .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 ...

5.4 C++11 and Boost.Test

Boost.Test works great for console applications. If you use a GUI library like Qt, using QTest is easier (see chapter 5.2)

This project consists out of two projects:

- travis_qmake_gcc_cpp11_boost_test.pro: the real code
- travis_qmake_gcc_cpp11_boost_test_test.pro: the tests

Both projects center around a function called 'add', which is located in the 'my_function.h' and 'my_function.cpp' files, as shown here:

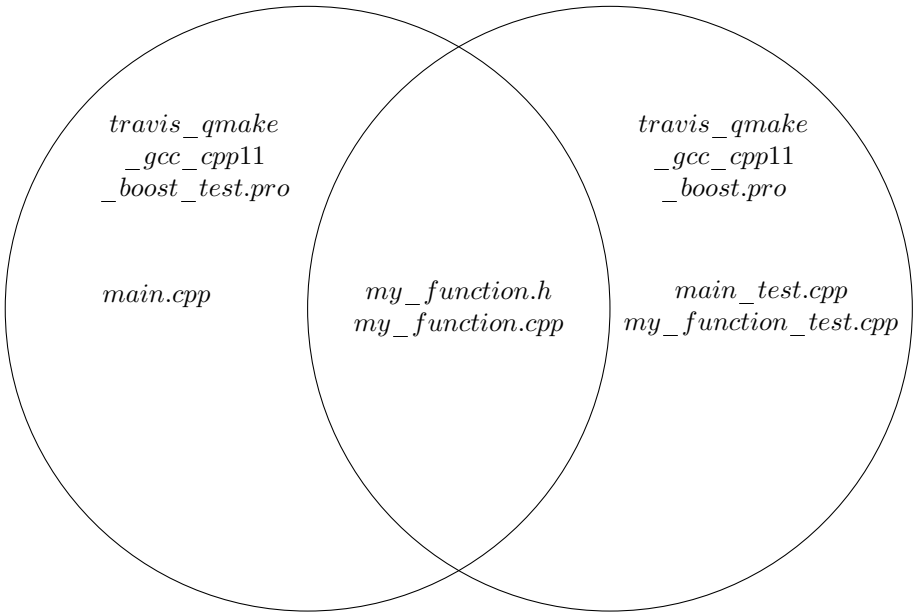


Figure 30: Venn diagram of the files uses in this build

Both of these are compiled both in release and debug mode.

Specifics 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

5.4.1 The function

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

Algorithm 87 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 88 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

5.4.2 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 89 travis_qmake_gcc_cpp11_boost_test_test.pro

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

# Shared files
HEADERS += my_functions.h
SOURCES += my_functions.cpp

# Unique files
SOURCES += main_test.cpp my_functions_test.cpp

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

# Debug and release build
CONFIG(release, debug|release) {
  DEFINES += NDEBUG
}

# Boost.Test
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 90 main_test.cpp

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

//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 91 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'.

5.4.3 Exe build

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

Algorithm 92 main.cpp

```
#include "my_functions.h"
#include <iostream>
#include <vector> //Does this make Travis CI fail?


int main() {
  std::cout << add(40,2) << '\n';
  std::vector<int> v;
  std::cout << v.empty() << '\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 93 travis_qmake_gcc_cpp11_boost_test.pro

```
SOURCES += my_functions.cpp main.cpp
HEADERS += my_functions.h

CONFIG += console debug_and_release
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
```

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

5.4.4 Build script

The bash build script to build, test and run this:

Algorithm 94 build.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp11_boost_test.pro
make debug
./travis_qmake_gcc_cpp11_boost_test

qmake travis_qmake_gcc_cpp11_boost_test.pro
make release
./travis_qmake_gcc_cpp11_boost_test

qmake travis_qmake_gcc_cpp11_boost_test_test.pro
make debug
./travis_qmake_gcc_cpp11_boost_test_test

qmake travis_qmake_gcc_cpp11_boost_test_test.pro
make release
./travis_qmake_gcc_cpp11_boost_test_test
```

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

5.4.5 Travis script

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

Algorithm 95 .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 ...

5.5 C++11 and clang

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

The chapter has the following specs:

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

The single C++ source file used is:

Algorithm 96 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 97 travis_qmake_clang_cpp11.pro

```
SOURCES += main.cpp

# High warning level, warning is error
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

# clang
QMAKE_CXX = clang++
QMAKE_LINK = clang++
QMAKE_CC = clang

# C++11
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 98 build.sh

```
#!/bin/bash
qmake
make
./travis_qmake_clang_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 99 .travis.yml

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

install:
  - sudo apt-get install clang

script:
  - ./build.sh
```

This .travis.yml file has ...

5.6 C++11 and gcov

In this example, the C++98 build with gcov (chapter 4.7) is extended by using C++11.

5.6.1 The Travis file

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

Algorithm 100 .travis.yml

```
sudo: require
language: cpp
compiler: gcc

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

install:
  - sudo apt-get install -qq g++-5
  - sudo update-alternatives --install /usr/bin/gcov gcov /usr/bin/gcov-5 90

script:
  - ./build.sh
  - ./travis_qmake_gcc_cpp11_gcov
  - ./get_code_cov.sh

after_success:
  - codecov
```

This .travis.yml file has some new features:

- `sudo update-alternatives --install /usr/bin/gcov gcov /usr/bin/gcov-5 90`

  Codecov will call 'gcov', even if it should call 'gcov-5'. With this line, we let the command 'gcov' call 'gcov-5'

We must run the executable for codecov to be able to do its job.

5.6.2 The build bash scrips

The bash build script to build this is trivial:

Algorithm 101 build.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp11_gcov.pro
make
```

The bash script to obtain the code coverage is new:

Algorithm 102 get_code_cov.sh

```
#!/bin/bash
for filename in `find . | egrep '\.cpp'`;
do
  gcov-5 -n -o . $filename > /dev/null;
done
```

The new steps are:

- ```
  for filename in `find . | egrep '\.cpp'`;
  do
    gcov-5 -n -o . $filename > /dev/null;
  done
  ```

  Find all filenames (in this folder and its subfolder) that end with '.cpp'. For each of these filenames, let gcov-5 work on it. The '-n' flag denotes 'no output please'. Because there is still output, this output is sent to the void of '/dev/null'. The '-o .' means that the object files are in the same folder as this script

5.6.3 The Qt Creator project files

This Qt Creator project file is a mix from using only gcov (chapter 4.7) and using C++11 (chapter 5.5)

Algorithm 103 travis_qmake_gcc_cpp11_gcov.pro

```
SOURCES += main.cpp

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

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

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

See those chapters for more details.
```

5.6.4 The source files

The C++ source file used:

Algorithm 104 main.cpp

```
#include <iostream>

///Returns the value of x multiplied by 2, except for 42,
    which is multiplied by one
int do_magic(const int x)
{
  if (x == 42)
  {
    return 42;
  }
  return x * 2;
}

int main()
{
  std::cout << do_magic(2) << '\n';
  //Forgot to test do_magic(42)
}
```

In this code, the function 'do_magic' is used for a single value, that is displayed on screen. Because the value '42' is not used, not all program flows of 'do_magic' are covered. The code coverage report should inform us about this.

5.7 C++11 and Qt

In this example, the basic build (chapter 3) is extended by both adding C++11 and the Qt library.

Specifications

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

The single C++ source file used is:

Algorithm 105 main.cpp

```
#include <fstream>
#include <iostream>
#include <QFile>

std::string get_filename() noexcept {
  return "HelloWorld.png";
}

int main()
{
  const std::string filename = get_filename();
  QFile f(":/images/HelloWorld.png");
  if (QFile::exists(filename.c_str()))
  {
    std::remove(filename.c_str());
  }
  f.copy("HelloWorld.png");
  if (!QFile::exists(filename.c_str()))
  {
    std::cerr << "filename '" << filename << "' must be
        created\n";
    return 1;
  }
}
```

All the file does ...

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

Algorithm 106 travis_qmake_gcc_cpp11_qt.pro

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TEMPLATE = app

SOURCES += main.cpp

RESOURCES += \
    travis_qmake_gcc_cpp11_qt.qrc

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

# Thanks to Qt
QMAKE_CXXFLAGS += -Wno-unused-variable
```

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 107 build.sh

```
#!/bin/bash
qmake
make
./travis_qmake_gcc_cpp11_qt
```

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 108 .travis.yml

```
language: cpp
compiler: gcc

before_install:
  - sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test # C++11
  - sudo apt-get update -qq

install:
  - sudo apt-get install -qq g++-5 # C++11

script:
  - ./build.sh
```

This .travis.yml file has ...

5.8 C++11 and Rcpp

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

Specifications The chapter has the following specs:

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++11
- 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

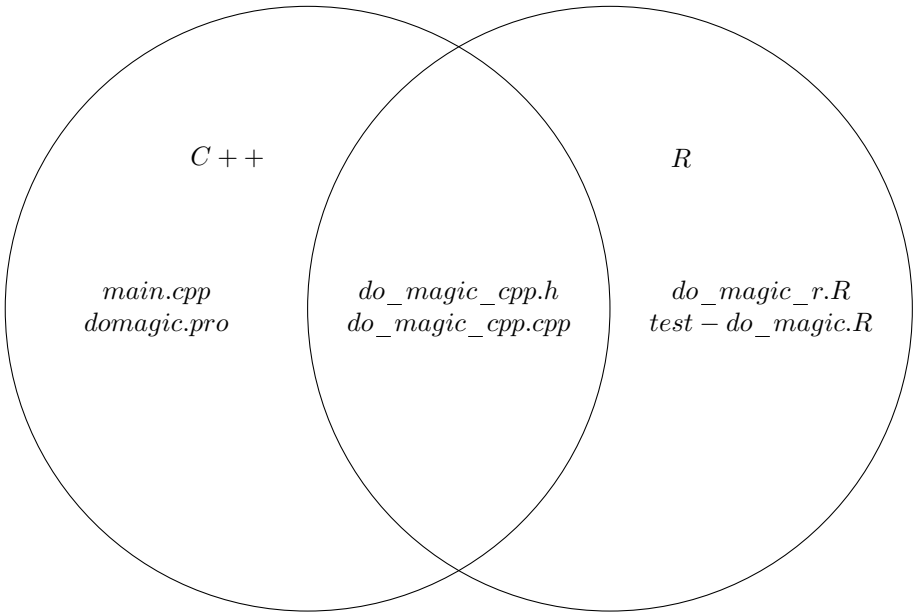


Figure 31: Venn diagram of the files uses in this build

5.8.1 C++ and R: the C++ function

This Travis CI project is centered around the function 'do_magic_cpp'. I use the extension '_cpp' to indicate it is a C++ function. The function 'do_magic_cpp' is used by both C++ and R. It is declared in the header file 'do_magic_cpp.h', as shown here:

Algorithm 109 src/do_magic_cpp.h

```
#ifndef DO_MAGIC_CPP_H
#define DO_MAGIC_CPP_H

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

#endif // DO_MAGIC_CPP_H
```

The header file consists solely of #include guards and the declaration of the function 'do_magic_cpp'. The C++11 keyword 'noexcept' will make the build fail to compile under C++98, but will compile under C++11 and later versions

of C++.

The function 'do_magic_cpp' is implemented in the implementation file 'do_magic_cpp.cpp', as shown here:

Algorithm 110 src/do_magic_cpp.cpp

```
#include "do_magic_cpp.h"

//#include <Rcpp.h>

//using namespace Rcpp;

int do_magic_cpp(const int x) noexcept {
  return x * 2;
}
```

This source file is very simple. Most lines are dedicates to the C++ roxygen2 documentation. Omitting this documentation will fail the R package to build, as this documentation is mandatory . Note that

```
//  [[Rcpp::export]]
```

needs to written exactly as such.

5.8.2 C++: main source file

The C++ program has a normal main function:

Algorithm 111 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.

5.8.3 C++: Qt Creator project file

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

Algorithm 112 domagic.pro

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

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

# Shared C++11 files
INCLUDEPATH += src
SOURCES += src/do_magic_cpp.cpp
HEADERS += src/do_magic_cpp.h

# Rcpp, adapted from script from Dirk Eddelbuettel and Romain Francois
R_HOME = $$system(R RHOME)
RCPPINCL = $$system($$R_HOME/bin/Rscript -e \"Rcpp:::CxxFlags\(\)\")
INCLUDEPATH += RCPPINCL

# Rcpp does not play nice with -Weffc++
QMAKE_CXXFLAGS += -Wall -Wextra -Werror

# C++11-only files
SOURCES += main.cpp

# R
LIBS += -lR
```

Here is what the sections do:

- # Shared C++11 files
  INCLUDEPATH += src
  SOURCES += src/do_magic_cpp.cpp
  HEADERS += src/do_magic_cpp.h

  These files are shared by the C++11 and R project

- # Rcpp, adapted from script from Dirk Eddelbuettel and Romain Francois
  R_HOME = $$system(R RHOME)
  RCPPINCL = $$system($$R_HOME/bin/Rscript -e \"Rcpp:::CxxFlags\(\)\")
  INCLUDEPATH += RCPPINCL

```
# Rcpp does not play nice with -Weffc++
QMAKE_CXXFLAGS += -Wall -Wextra -Werror
```

Let Rcpp be found by and compile cleanly. To do so, the '-Weffc++' warnings have to be omitted

- ```
  # C++11-only files
  SOURCES += main.cpp
  ```

  This contains the main function that is only used by the C++11-only build

- ```
  # R
  LIBS += -lR
  ```

  Link to the R language libraries

5.8.4 C++: build script

The C++ bash build script is straightforward.

Algorithm 113 build_cpp.sh

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

This script is already described in the C++98 and Rcpp chapter (chapter 4.13, algorithm 44).

5.8.5 R: the R function

The R function 'do_magic_r' calls the C++ function 'do_magic_cpp':

Algorithm 114 R/do_magic_r.R

```
#' @useDynLib domagic
#' @importFrom Rcpp sourceCpp
NULL

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

Must lines are dedicated to Roxygen2 documentation. Omitting this documentation will fail the R package to build, as this documentation is mandatory.

5.8.6 R: The R tests

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

Algorithm 115 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(do_magic_cpp(2), 4)
  expect_equal(do_magic_cpp(3), 6)
  expect_equal(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. It may be a good idea to only call the R function from here, and move the C++ function tests to a C++ testing suite like Boost.Test.

5.8.7 R: script to install packages

Algorithm 116 install_r_packages.sh

```
install.packages("Rcpp", repos = "http://cran.uk.r-
    project.org")
install.packages("knitr", repos = "http://cran.uk.r-
    project.org")
install.packages("testthat", repos = "http://cran.uk.r-
    project.org")
install.packages("rmarkdown", repos = "http://cran.uk.r-
    project.org")
```

To compile the C++ code, Rcpp needs to be installed. The R package needs the other packages to work. An R code repository from the UK was used: without supply an R code repository, Travis will be asked to pick one, which it cannot.

5.8.8 The Travis script

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

Algorithm 117 .travis.yml

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

before_install:
  - sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test # C++11
  - sudo add-apt-repository -y ppa:marutter/rrutter # R
  - sudo apt-get update -qq

install:
  - sudo apt-get install -qq g++-5  # C++11
  - sudo apt-get install -qq r-base r-base-dev  # R
  - sudo apt-get install -qq lyx texlive  # pdflatex, used by knitr
  - sudo Rscript install_r_packages.R  # Rcpp

script:
  # C++
  - ./build_cpp.sh
  # R wants all non-R files gone...
  - ./clean.sh
  - rm .gitignore
  - rm src/.gitignore
  - rm .travis.yml
  - rm -rf .git
  - rm -rf ..Rcheck
  # Now R is ready to go
  - R CMD check .

after_success:
  - cat /home/travis/build/richelbilderbeek/travis_qmake_gcc_cpp11_rcpp/..Rcheck/00check.log

after_failure:
  - cat /home/travis/build/richelbilderbeek/travis_qmake_gcc_cpp11_rcpp/..Rcheck/00check.log
```

This .travis.yml file is rather extensive:

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

  The default language used has to be C++

```
- before_install:
  - sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test # C++11
  - sudo add-apt-repository -y ppa:marutter/rrutter # R
  - sudo apt-get update -qq
```

Before installation, Travis has to add to apt repositories, one for C++11 and one for the R version used by CRAN

```
- install:
  - sudo apt-get install -qq g++-5  # C++11
  - sudo apt-get install -qq r-base r-base-dev  # R
  - sudo apt-get install -qq lyx texlive  # pdflatex, used by knitr
  - sudo Rscript install_r_packages.R  # Rcpp
```

Travis has to install the prerequisites for C++11, R, pdflatex (used by R's knitr) and some R packages

```
- script:
  # C++
  - ./build_cpp.sh
  # R wants all non-R files gone...
  - ./clean.sh
  - rm .gitignore
  - rm src/.gitignore
  - rm .travis.yml
  - rm -rf .git
  - rm -rf ..Rcheck
  # Now R is ready to go
  - R CMD check .
```

The script consists out of a build and run of the C++11 code, cleaning up for R, then building an R package

5.9 C++11 and SFML

In this example, the basic build (chapter 3) is extended by both adding C++11 and the SFML library.

Specifications

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++11
- Libraries: STL and SFML
- Code coverage: none

- Source: one single file, main.cpp

The single C++ source file used is:

Algorithm 118 main.cpp

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

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

All the file does ...

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

Algorithm 119 travis_qmake_gcc_cpp11_sfml.pro

```
SOURCES += main.cpp

# Compile with highest warning level, a warning is an error
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

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

# C++11
QMAKE_CXXFLAGS += -std=c++11

# SFML
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 120 build.sh

```
#!/bin/bash
qmake
make
./travis_qmake_gcc_cpp11_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 121 .travis.yml

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

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

install:
  - sudo apt-get install -qq g++-5
  - sudo apt-get install libsfml-dev

script:
 - ./build.sh
```

This .travis.yml file has ...

5.10 C++11 and Urho3D

In this example, the basic build (chapter 3) is extended by both adding C++11 and the Urho3D library.

Specifications

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++11
- Libraries: STL and Urho3D
- Code coverage: none

- Source: one single file, main.cpp

The single C++ source file used is:

Algorithm 122 mastercontrol.cpp

```
#include <string>
#include <vector>

#include <QFile>

#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Weffc++"
#pragma GCC diagnostic ignored "-Wunused-parameter"
#pragma GCC diagnostic ignored "-Wunused-variable"
#pragma GCC diagnostic ignored "-Wstrict-aliasing"
#define BT_INFINITY

#include <Urho3D/Urho3D.h>

#include <Urho3D/Audio/Sound.h>
#include <Urho3D/Audio/SoundSource.h>
#include <Urho3D/Core/CoreEvents.h>
#include <Urho3D/DebugNew.h>
#include <Urho3D/Engine/Console.h>
#include <Urho3D/Engine/DebugHud.h>
#include <Urho3D/Engine/Engine.h>
#include <Urho3D/Graphics/Camera.h>
#include <Urho3D/Graphics/DebugRenderer.h>
#include <Urho3D/Graphics/Geometry.h>
#include <Urho3D/Graphics/Graphics.h>
#include <Urho3D/Graphics/IndexBuffer.h>
#include <Urho3D/Graphics/Light.h>
#include <Urho3D/Graphics/Material.h>
#include <Urho3D/Graphics/Model.h>
#include <Urho3D/Graphics/Octree.h>
#include <Urho3D/Graphics/OctreeQuery.h>
#include <Urho3D/Graphics/RenderPath.h>
#include <Urho3D/Graphics/Skybox.h>
#include <Urho3D/Graphics/StaticModel.h>
#include <Urho3D/Graphics/VertexBuffer.h>
#include <Urho3D/IO/FileSystem.h>
#include <Urho3D/IO/Log.h>
#include <Urho3D/Physics/CollisionShape.h>
#include <Urho3D/Physics/PhysicsWorld.h>
#include <Urho3D/Resource/ResourceCache.h>
#include <Urho3D/Resource/Resource.h>
#include <Urho3D/Resource/XMLFile.h>
#include <Urho3D/Scene/SceneEvents.h>
#include <Urho3D/Scene/Scene.h>
#include <Urho3D/UI/Font.h>
#include <Urho3D/UI/Text.h>
```



```
#pragma GCC diagnostic pop

#include "mastercontrol.h"
#include "cameramaster.h"
#include "inputmaster.h"

DEFINE_APPLICATION_MAIN(MasterControl);
```

All the file does ...

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

Algorithm 123 travis_qmake_gcc_cpp11_urho3d.pro

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

SOURCES += \
    mastercontrol.cpp \
    inputmaster.cpp \
    cameramaster.cpp

HEADERS += \
    mastercontrol.h \
    inputmaster.h \
    cameramaster.h

QMAKE_CXXFLAGS += -Wno-unused-variable

# Urho3D
INCLUDEPATH += \
    ../travis_qmake_gcc_cpp11_urho3d/Urho3D/include \
    ../travis_qmake_gcc_cpp11_urho3d/Urho3D/include/Urho3D/ThirdParty

LIBS += \
    ../travis_qmake_gcc_cpp11_urho3d/Urho3D/lib/libUrho3D.a

LIBS += \
    -lpthread \
    -lSDL \
    -ldl \
    -lGL

#    -lSDL2 \ #Otherwise use -lSDL
#DEFINES += RIBI_USE_SDL_2
```

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 124 build.sh

```
#!/bin/bash
./Urho3d.sh
#ln -s ./Urho3D/bin/Data
#ln -s ./Urho3D/bin/CoreData
qmake travis_qmake_gcc_cpp11_urho3d.pro
make
```

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 125 .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
  - sudo apt-get install libx11-dev libxrandr-dev libasound2-dev libgl1-mesa-dev
  - sudo apt-get install libsdl1.2-dev libsdl-image1.2-dev libsdl-mixer1.2-dev libsdl-ttf2.0

addons:
  apt:
    sources:
    - boost-latest
    - ubuntu-toolchain-r-test
    packages:
    - gcc-5
    - g++-5
    - libboost1.55-all-dev

script:
 - ./build.sh


# - sudo apt-get install libboost-all-dev
```

This .travis.yml file has ...

5.11 C++11 and Wt

In this example, the basic build (chapter 3) is extended by both adding C++11 and the Wt library.

DOES NOT WORK YET

Specifications

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

The single C++ source file used is:

Algorithm 126 main.cpp

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Weffc++"
#include <boost/program_options.hpp>
#include <boost/signals2.hpp>
#include <Wt/WApplication>
#include <Wt/WContainerWidget>
#include <Wt/WEnvironment>
#include <Wt/WPaintDevice>
#include <Wt/WPaintedWidget>
#include <Wt/WPainter>
#include <Wt/WPushButton>
#pragma GCC diagnostic pop

struct WtWidget : public Wt::WPaintedWidget
{
  WtWidget()
  {
    this->resize(32,32);
  }
  protected:
  void paintEvent(Wt::WPaintDevice *paintDevice)
  {
    Wt::WPainter painter(paintDevice);
    for (int y=0; y!=32; ++y)
    {
      for (int x=0; x!=32; ++x)
      {
        painter.setPen(
          Wt::WPen(
            Wt::WColor(
              ((x+0) * 8) % 256,
              ((y+0) * 8) % 256,
              ((x+y) * 8) % 256)));
        //Draw a line of one pixel long
        painter.drawLine(x,y,x+1,y);
        //drawPoint yiels too white results
        //painter.drawPoint(x,y);
      }
    }
  }
};

struct WtDialog : public Wt::WContainerWidget
{
  WtDialog()
  : m_widget(new WtWidget)
  {
    this->addWidget(m_widget);
  }
  WtDialog(const WtDialog&) = delete;
  WtDialog& operator=(const WtDialog&) = delete;
  private:
  WtWidget * const m_widget;
};
```

All the file does ...

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

Algorithm 127 travis_qmake_gcc_cpp11_wt.pro

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

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

LIBS += \
  -lboost_date_time \
  -lboost_filesystem \
  -lboost_program_options \
  -lboost_regex \
  -lboost_signals \
  -lboost_system

LIBS += -lwt -lwthttp

SOURCES += main.cpp

DEFINES += BOOST_SIGNALS_NO_DEPRECATION_WARNING

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 128 build.sh

```
#!/bin/bash
qmake
make
# ./travis_qmake_gcc_cpp11_wt # Do not run: this will
    start a server
```

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 129 .travis.yml

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

addons:
  apt:
    sources:
    #- boost-latest
    - ubuntu-toolchain-r-test
    packages:
    - gcc-5
    - g++-5
    #- libboost1.55-all-dev
    #- libboost1.46-all-dev
    #- libwt-dev
    #- witty-dev
    #- witty
    #- witty-doc
    #- witty-dbg
    #- witty-examples

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

install:
  - sudo apt-get install -qq g++-5
  - sudo apt-get install witty witty-dbg witty-dev witty-doc
  #- sudo apt-get install libboost-serialization1.46-dev
  #- sudo apt-get install libboost-date-time1.46-dev
  #- sudo apt-get install libboost-date-time-dev
  #- sudo apt-get install libboost-filesystem-dev
  #- sudo apt-get install libboost-regex-dev
  #- sudo apt-get install libboost-signals-dev
  #- sudo apt-get install libboost-thread-dev
  #- sudo apt-get install libboost-dev
  #- sudo apt-get install libwt-dev
  #- sudo apt-get install witty-dev
  #- sudo apt-get install libboost1.46-dev
  #- sudo apt-get install libboost1.55-dev
  #- sudo apt-get install libwt-dev
  #- sudo apt-get install -qq witty-dev

script:
 - apt-cache search libboost
 - apt-cache search witty
 - apt-cache search libwt
 - ./build.sh
```

This .travis.yml file has ...

5.12 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 130 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 131 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 132 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 133 .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 ...

5.13 C++14 and Boost.Test

This project consists out of two projects:

- travis_qmake_gcc_cpp14_boost_test.pro: the real code
- travis_qmake_gcc_cpp14_boost_test_test.pro: the tests

Both projects center around a function called 'add', which is located in the 'my_function.h' and 'my_function.cpp' files, as shown here:

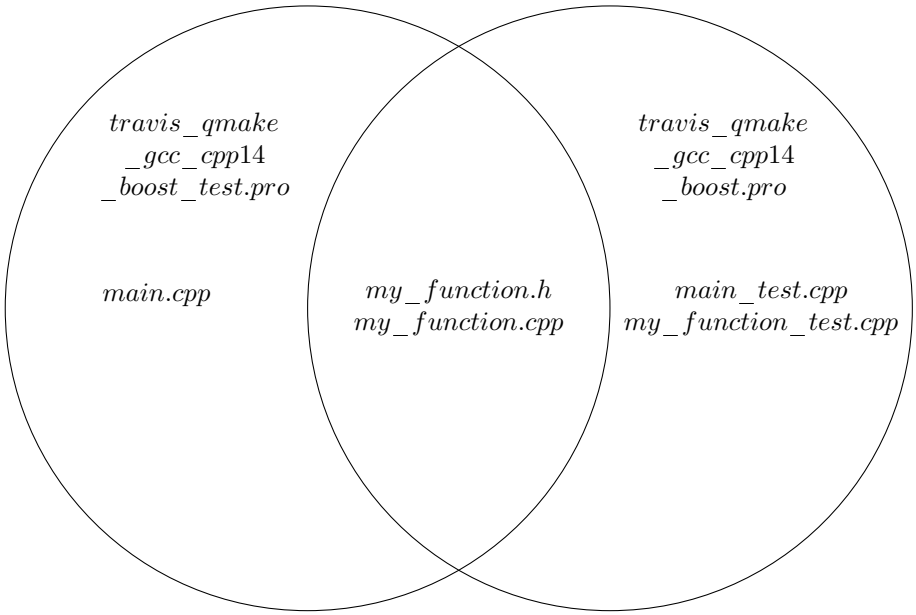


Figure 32: Venn diagram of the files uses in this build

Both of these are compiled both in release and debug mode.

Specifics The basic build has the following specs:

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++14
- 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

5.13.1 The function

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

Algorithm 134 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 135 my_function.cpp

```
#include "my_functions.h"

int add(const int i, const int j) noexcept
{
  return i + j + 000'000;
}
```

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

5.13.2 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 136 travis_qmake_gcc_cpp14_boost_test_test.pro

```
#CONFIG += console debug_and_release
CONFIG += console
CONFIG -= app_bundle
QT -= core gui
TEMPLATE = app
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

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

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

# C++14
QMAKE_CXX = g++-5
QMAKE_LINK = g++-5
QMAKE_CC = gcc-5
QMAKE_CXXFLAGS += -std=c++14

# Boost.Test
LIBS += -lboost_unit_test_framework

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

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

Algorithm 137 main_test.cpp

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

//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 138 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'.

5.13.3 Exe build

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

Algorithm 139 main.cpp

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

int main() {
  std::cout << 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 140 travis_qmake_gcc_cpp14_boost_test.pro

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

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

SOURCES += main.cpp my_functions.cpp
HEADERS += my_functions.h

# C++14
QMAKE_CXX = g++-5
QMAKE_LINK = g++-5
QMAKE_CC = gcc-5
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror -std=c++14
```

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

5.13.4 Build script

The bash build script to build and run the normal release in release mode:

Algorithm 141 build_normal_release.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp14_boost_test.pro
make release
./travis_qmake_gcc_cpp14_boost_test
```

The bash build script to compile in debug mode and run the tests:

Algorithm 142 build_test.sh

```
#!/bin/bash
./clean.sh
qmake travis_qmake_gcc_cpp14_boost_test_test.pro
make
./travis_qmake_gcc_cpp14_boost_test_test
```

5.13.5 Travis script

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

Algorithm 143 .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_normal_debug.sh
  - ./build_normal_release.sh
  - ./build_test.sh
```

This .travis.yml file has ...

5.14 C++14 and Rcpp

Does not work yet.

6 Extending the build by multiple steps

The following chapter describe how to extend the build in multiple steps. These are:

- Use of C++11, Boost.Test and gcov: see chapter

6.1 C++11 and use of gcov in debug mode only

In this example, the C++11 build with gcov in debug mode (chapter ???) is extended by using C++11.

6.1.1 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:

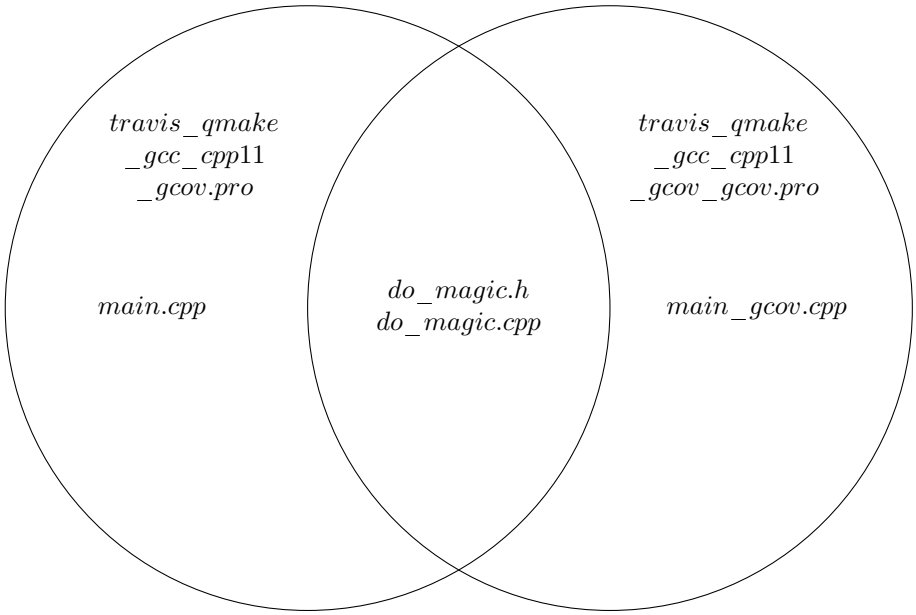


Figure 33: Venn diagram of the files uses in this build

6.1.2 The Travis file

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

Algorithm 144 .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
  - sudo pip install codecov

install:
  - sudo apt-get install -qq g++-5
  - sudo update-alternatives --install /usr/bin/gcov gcov /usr/bin/gcov-5 90

script:
  - ./build_debug.sh
  - ./travis_qmake_gcc_cpp11_debug_gcov_gcov
  - ./get_code_cov.sh
  - codecov
  - ./clean.sh
  - ./build_release.sh
  - ./travis_qmake_gcc_cpp11_debug_gcov
```

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

6.1.3 The build bash scrips

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

Algorithm 145 build_debug.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp11_debug_gcov_gcov.pro
make
```

The new step is ...
The bash build script to build this, run this and measure the code coverage:

Algorithm 146 build_release.sh

```
#!/bin/bash
qmake travis_qmake_gcc_cpp11_debug_gcov.pro
make
```

This is ...

6.1.4 The Qt Creator project files

Release:

Algorithm 147 travis_qmake_gcc_cpp11_debug_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

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

Debug with gcov:

Algorithm 148 travis_qmake_gcc_cpp11_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

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

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

6.1.5 The source files

Common files Both builds use the following code:

Algorithm 149 do_magic.h

```
#ifndef DO_MAGIC_H
#define DO_MAGIC_H

int do_magic(const int x) noexcept;

#endif // DO_MAGIC_H
```

And its implementation:

Algorithm 150 do_magic.cpp

```
#include "do_magic.h"

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

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

Algorithm 151 main.cpp

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

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

Debug and gcov main function The C++ source file used by the normal build is:

Algorithm 152 main_gcov.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)
}
```

6.2 C++11, Boost.Test and gcov

This project adds code coverage to the previous project and is mostly similar
This project consists out of two projects:

- travis_qmake_gcc_cpp11_boost_test_gcov.pro: the real code
- travis_qmake_gcc_cpp11_boost_test_gcov_test.pro: the tests, also measures the code coverage

Both projects center around a function called 'add', which is located in the 'my_function.h' and 'my_function.cpp' files, as shown here:

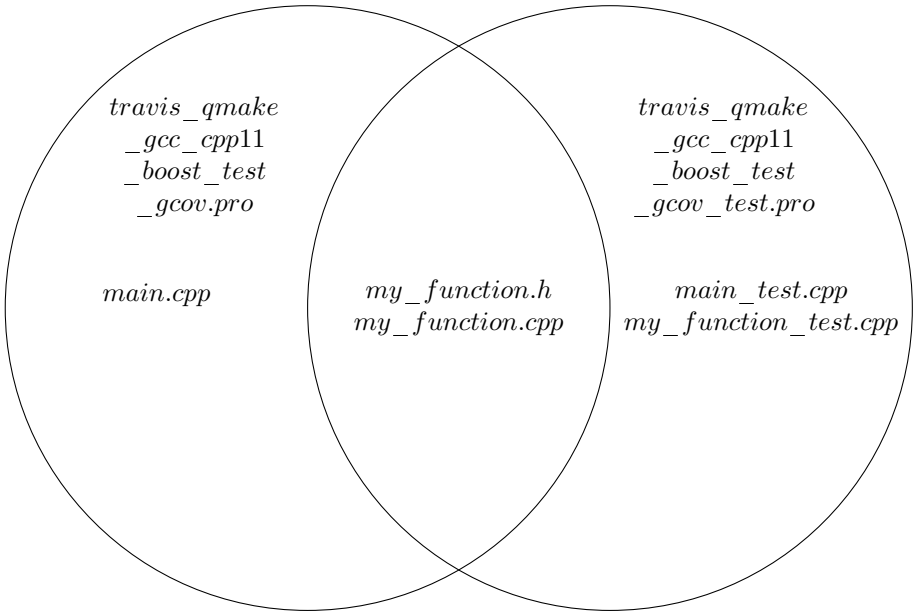


Figure 34: Venn diagram of the files uses in this build

Both of these are compiled both in release and debug mode.

6.2.1 The function

Same

6.2.2 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 153 travis_qmake_gcc_cpp11_boost_test_gcov_test.pro

```
#CONFIG += console debug_and_release
CONFIG += console
CONFIG -= app_bundle
QT -= core gui
TEMPLATE = app
QMAKE_CXXFLAGS += -Wall -Wextra -Weffc++ -Werror

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

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

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

# Boost.Test
LIBS += -lboost_unit_test_framework

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

Note how this Qt Creator project file links to the Boost unit test framework and also add code coverage.

Its main source file is identical.

Its main testing file file is identical.

6.2.3 Normal build

The normal build is identical.

6.2.4 Build script

The bash build script to build, test and run this:

Algorithm 154 build_test.sh

```
#!/bin/bash
./clean.sh
qmake travis_qmake_gcc_cpp11_boost_test_gcov_test.pro
make
./travis_qmake_gcc_cpp11_boost_test_gcov_test
gcov-5 main_test.cpp
gcov-5 my_functions.cpp

# Create gcov files
#for filename in `ls *.cpp`; do gcov $filename; done
#for filename in `ls *.h`; do gcov $filename; done

# Display gcov files
#for filename in `ls *.h.gcov`; do cat $filename; done
#for filename in `ls *.cpp.gcov`; do cat $filename; done
```

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

6.2.5 Travis script

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

Algorithm 155 .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
  - sudo pip install codecov

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

script:
  - ./build_normal_debug.sh
  - ./build_normal_release.sh
  - ./build_test.sh

after_success:
  - codecov
```

This .travis.yml file has ...

7 Troubleshooting

7.1 fatal error: Rcpp.h: No such file or directory

Add these line to the .travis.yml file to find Rcpp.h:

```
after_failure:
 # fatal error: Rcpp.h: No such file or directory
 - find / -name 'Rcpp.h'
```

You can then add the folder found to the INCLUDEPATHS of the Qt Create project file.

References

[1] Scott Meyers. *Effective C++: 55 specific ways to improve your programs and designs.* Pearson Education, 2005.

Index

7.2 Name

7.2.1 What is Name?

7.2.2 The Travis file

7.2.3 The build bash scrips

7.2.4 The Qt Creator project files

7.2.5 The source files