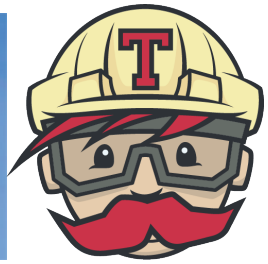
Travis C++ tutorial

Richèl Bilderbeek

April 1, 2016



Contents

1 Introduction **3**
- 1.1 License . . . 3
- 1.2 Continuous integration . . . 3
- 1.3 Tutorial style . . . 4
- 1.4 This tutorial . . . 4
- 1.5 Acknowledgements . . . 5
- 1.6 Collaboration . . . 5
- 1.7 Feedback . . . 5

2 Setting up the basic build **5**
- 2.1 Create a GitHub online . . . 5
- 2.2 Bring the git repository to your local computer . . . 6
- 2.3 Create a Qt Creator project . . . 9
- 2.4 Create the build bash scripts . . . 12

3 The basic build **12**
- 3.1 What is a C++98 'Hello world' program? . . . 13
- 3.2 The Travis file . . . 13
- 3.3 The build bash script . . . 14
- 3.4 Qt Creator project file . . . 15
- 3.5 C++ source file . . . 15

4 Extending the build by one step **16**
4.1 Use of C++11 . . . 16
4.2 Use of C++14 . . . 19
4.3 Adding Boost . . . 21
4.4 Adding Boost.Test . . . 22
4.5 Use of clang . . . 22
4.6 Adding code coverage . . . 24
4.7 Adding profiling . . . 29
4.8 Adding the Qt library . . . 29
4.9 Adding Rcpp . . . 31
4.9.1 C++ and R: the C++ function . . . 32
4.9.2 C++: main source file . . . 33
4.9.3 C++: Qt Creator project file . . . 33
4.9.4 The C++ build script . . . 34
4.9.5 R: the R function . . . 34
4.9.6 R: The R tests . . . 35
4.9.7 The R script to install packages . . . 35
4.9.8 The Travis script . . . 36
4.9.9 fatal error: Rcpp.h: No such file or directory . . . 37
4.10 Adding the SFML library . . . 38
4.11 Adding the Urho3D library . . . 40
4.12 Adding the Wt library . . . 42

5 Extending the build by two steps **46**
5.1 Qt and QTest . . . 46
5.1.1 What is QTest? . . . 46
5.1.2 Do not use Boost.Test to test graphical Qt aplications . . . 47
5.1.3 The Travis file . . . 47
5.1.4 The build bash scrips . . . 47
5.1.5 The Qt Creator project files . . . 48
5.1.6 The source files . . . 49
5.2 C++11 and Boost libraries . . . 52
5.3 C++11 and Boost.Test . . . 54
5.3.1 The function . . . 55
5.3.2 Test build . . . 56
5.3.3 Exe build . . . 58
5.3.4 Build script . . . 59
5.3.5 Travis script . . . 60
5.4 C++11 and clang . . . 60
5.5 C++11 and gcov . . . 62
5.6 C++11 and Qt . . . 67
5.7 C++11 and Rcpp . . . 70
5.7.1 C++ and R: the C++ function . . . 71
5.7.2 C++: main source file . . . 72
5.7.3 C++: Qt Creator project file . . . 72
5.7.4 C++: build script . . . 74

5.7.5 R: the R function . . . 74
5.7.6 R: The R tests . . . 75
5.7.7 R: script to install packages . . . 75
5.7.8 The Travis script . . . 76
5.8 C++11 and SFML . . . 77
5.9 C++11 and Urho3D . . . 79
5.10 C++11 and Wt . . . 84
5.11 C++14 and Boost libraries . . . 89
5.12 C++14 and Boost.Test . . . 91
5.12.1 The function . . . 92
5.12.2 Test build . . . 92
5.12.3 Exe build . . . 94
5.12.4 Build script . . . 95
5.12.5 Travis script . . . 96
5.13 C++14 and Rcpp . . . 96

6 Extending the build by multiple steps 96
6.1 C++11, Boost.Test and gcov . . . 96
6.1.1 The function . . . 97
6.1.2 Test build . . . 97
6.1.3 Normal build . . . 98
6.1.4 Build script . . . 98
6.1.5 Travis script . . . 99

References 100

1 Introduction

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

1.1 License

Figure 1: Creative Commons license 4.0

1.2 Continuous integration

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

[1] if not you

writing.

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

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

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

A continuous integration tools checks what is submitted to the project and possibly rejects it when it does not satisfy the tests and/or requirements of the project. Instead of manually proofreading and/or testing the submission and 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 2 shows the GitHub homepage, `https://github.com`.

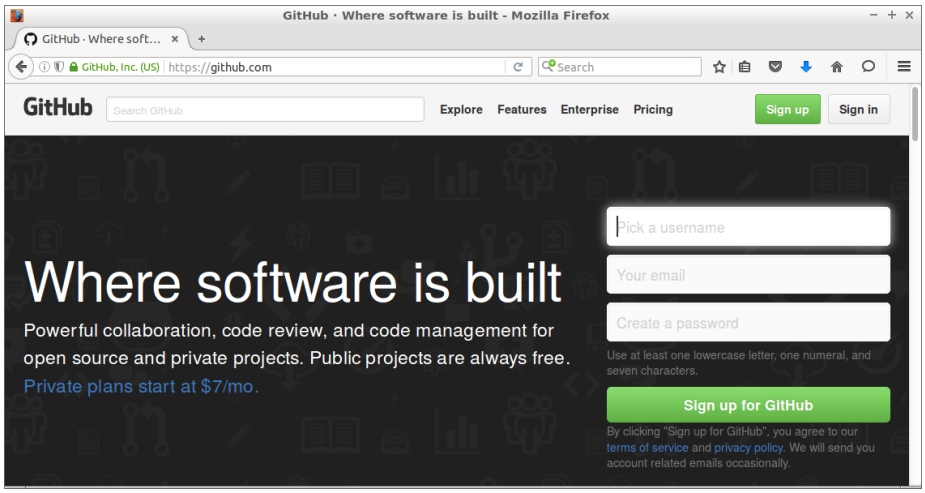


Figure 2: The GitHub homepage, `https://github.com`

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

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

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

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

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

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

2.2 Bring the git repository to your local computer

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

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

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

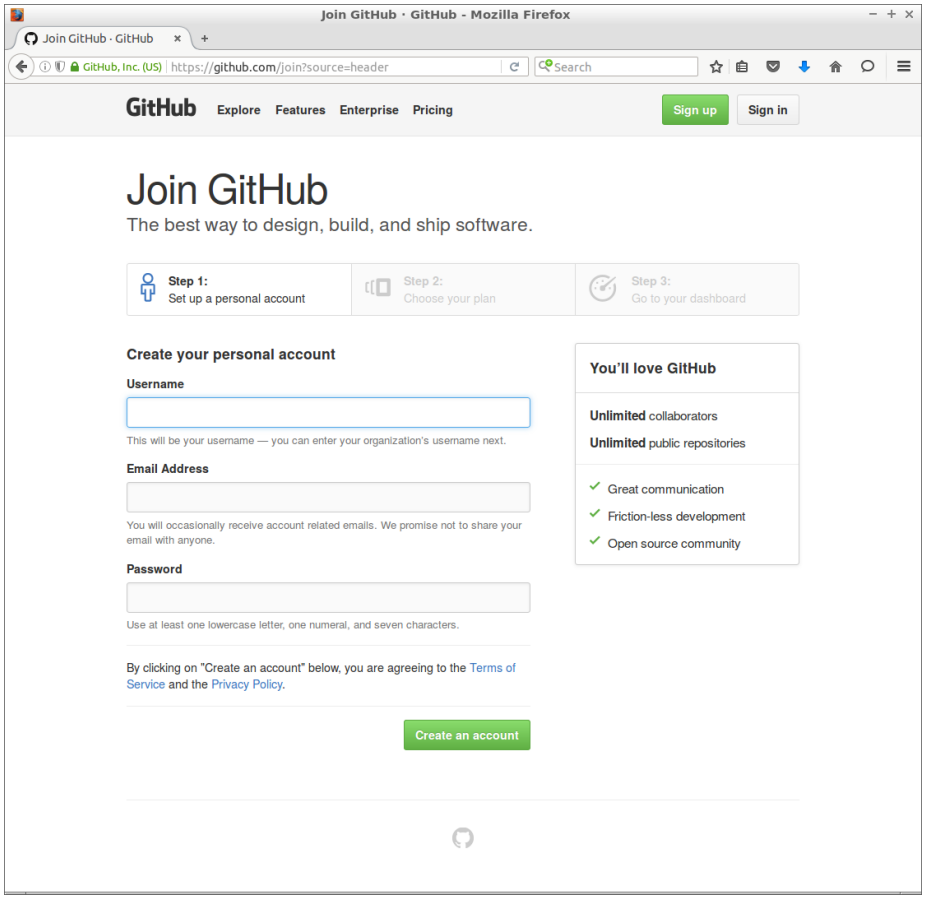


Figure 3: The join GitHub page

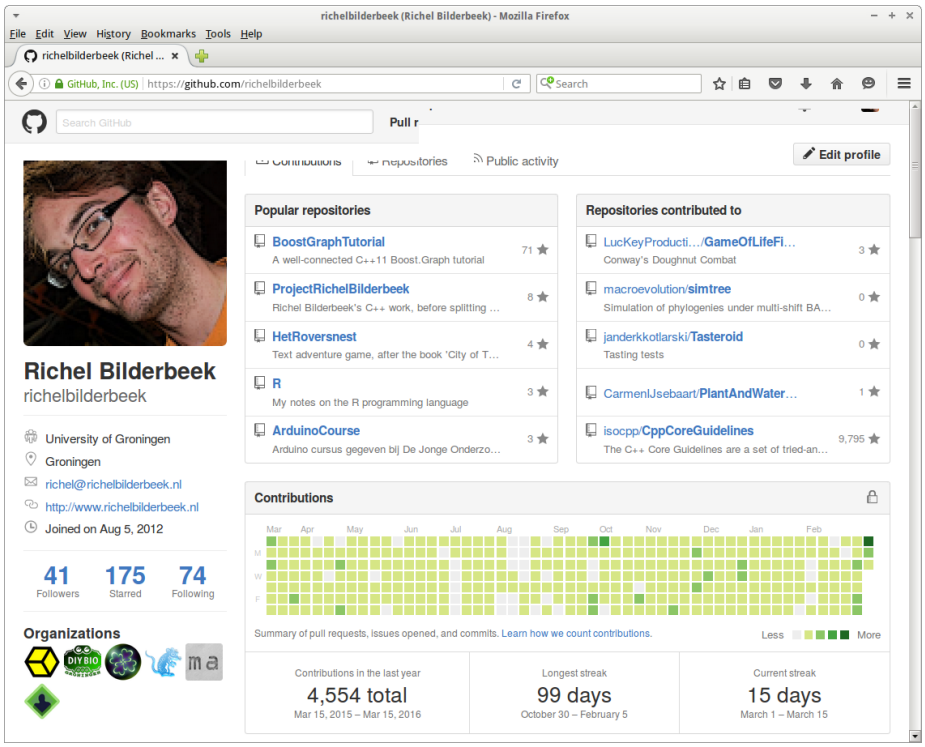


Figure 4: A GitHub profile page

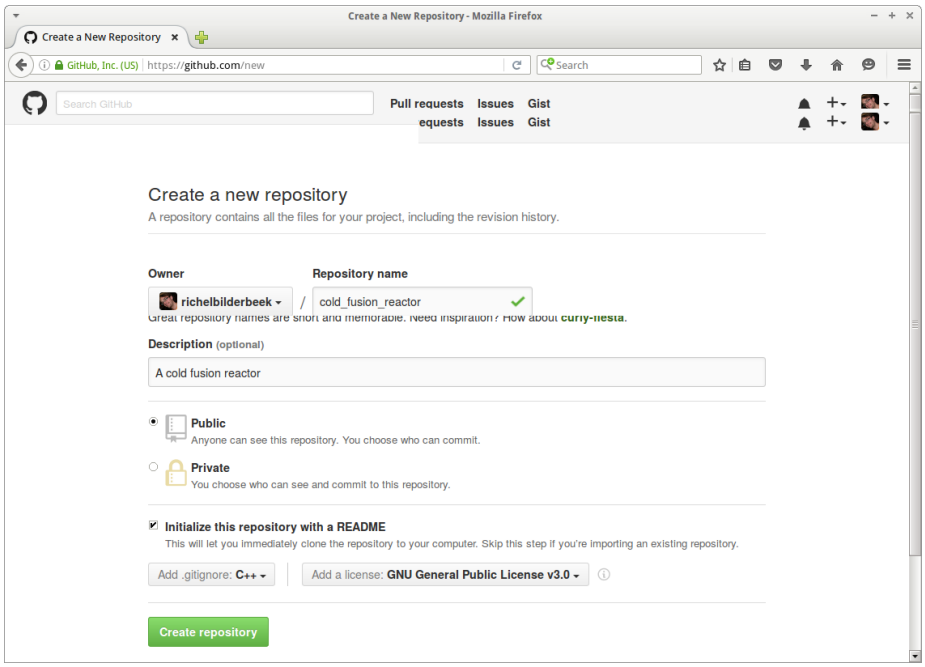


Figure 5: Create a GitHub repository

```
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

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.

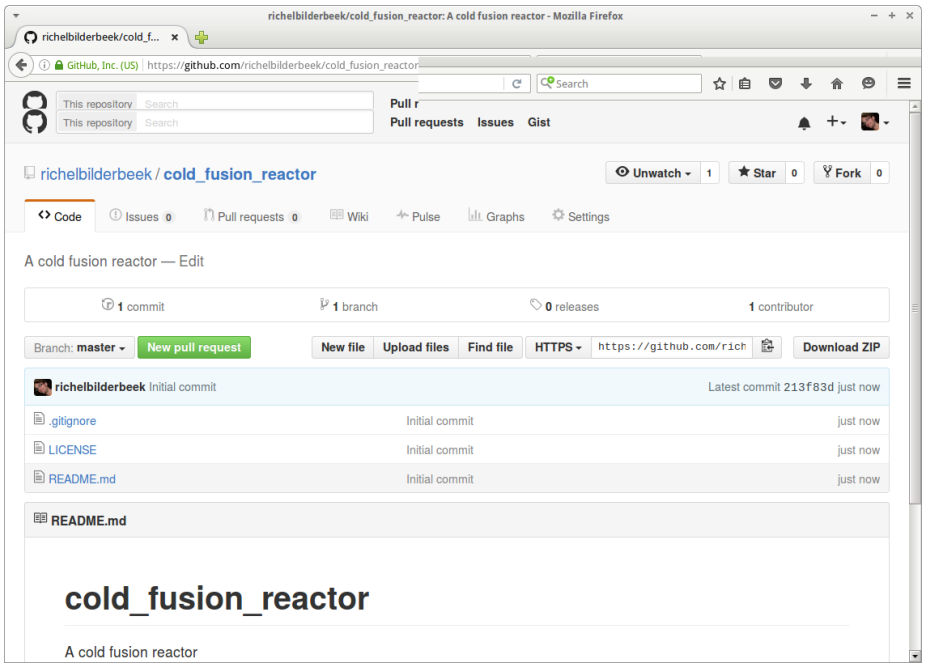


Figure 6: Created a GitHub repository

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 10: Overview of converting a C++ project to an executable

.cpp – compiler –> .o
.h –(dotted line)–> same .o
.o – linker –> executable

Figure 11: From files to 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 12: What qmake does

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

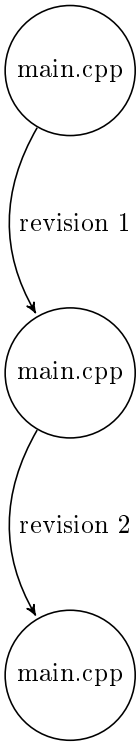


Figure 7: Multiple versions of main.cpp. git allows to always go back to each version of main

Figure 8: git logo

Figure 9: Qt creator logo

Figure 14: GCC logo

'Makefile' –[make]> 'executable'

Figure 13: What make does

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

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

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

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

2.4 Create the build bash scripts

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

3 The basic build

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

- 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
- Use of C++11: see chapter 4.1
- Use of C++14: see chapter 4.2
- Use of Boost: see chapter 4.3
- Use of Boost.Test: see chapter 4.4
- Use of clang: see chapter 4.5
- Use of gcov and Codecov: see chapter 4.6
- Use of gprof: see chapter 4.7
- Use of Qt: see chapter 4.8
- Use of QTest: see chapter
- Use of Rcpp: see chapter 4.9
- Use of SFML: see chapter 4.10
- Use of Urho3D: see chapter 4.11
- Use of Wt: see chapter 4.12

4.1 Use of C++11

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

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

Specifications The chapter has the following specs:

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

The single C++ source file used is:

Algorithm 5 main.cpp

```
#include <iostream>

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

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

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

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

Algorithm 6 travis_qmake_gcc_cpp11.pro

```
# 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

The bash build script to build and run this:

Algorithm 7 build.sh

```
#!/bin/bash
qmake
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 8 .travis.yml

```
sudo: true
language: cpp
compiler: gcc
before_install:
  - sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test
  - sudo apt-get update -qq
install: sudo apt-get install -qq g++-5
script:
  - ./build.sh
  - ./travis_qmake_gcc_cpp11
```

This .travis.yml file has some new features:

- before_install:

  The following events will take place before installation

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

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

- - sudo apt-get update -qq

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

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

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

4.2 Use of C++14

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

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

Specifications

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

The single C++ source file used is:

Algorithm 9 main.cpp

```
#include <iostream>

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

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

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

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

Algorithm 10 travis_qmake_gcc_cpp14.pro

```
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 4.1, except for that it uses one different QMAKE_CXXFLAGS item:

- QMAKE_CXXFLAGS += -std=c++14

  Compile under C++14

The bash build script to build and run this:

Algorithm 11 build.sh

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

The bash script has the same lines as the C++11 build in chapter 4.1
Setting up Travis is done by the following .travis.yml:

Algorithm 12 .travis.yml

```
sudo: true
language: cpp
compiler: gcc
before_install:
  - sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test
  - sudo apt-get update -qq
install: sudo apt-get install -qq g++-5
script:
  - ./build.sh
  - ./travis_qmake_gcc_cpp14
```

This .travis.yml file is the same as the C++11 build in chapter 4.1

Figure 15: Boost logo

4.3 Adding Boost

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

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

Specifications

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

The single C++ source file used is:

Algorithm 13 main.cpp

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

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

All the file does is to create an empty graph, from the Boost.Graph library. It will only compile when the Boost libraries are present.

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

Algorithm 14 travis_qmake_gcc_cpp98_boost.pro

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

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

The bash build script to build and run this:

Algorithm 15 build.sh

```
qmake
make
./travis_qmake_gcc_cpp98_boost
```

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

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

Algorithm 16 .travis.yml

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

This .travis.yml file has one new feature:

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

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

4.4 Adding Boost.Test

Adding only a testing framework does not work: it will not compile in C++98. Instead, this is covered in chapter 5.3.

4.5 Use of clang

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

What is clang? clang is a C++ compiler

Figure 16: clang logo

Specifications

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

The single C++ source file used is:

Algorithm 17 main.cpp

```
#include <iostream>

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

All the file does is ...

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

Algorithm 18 travis_qmake_clang_cpp98.pro

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

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

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 19 build.sh

```
#!/bin/bash
qmake
make
./travis_qmake_clang_cpp98
```

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 20 .travis.yml

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

install:
  - sudo apt-get install clang

script:
  - ./build.sh
```

This .travis.yml file has ...

4.6 Adding code coverage

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

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

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

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 17: Codecov logo

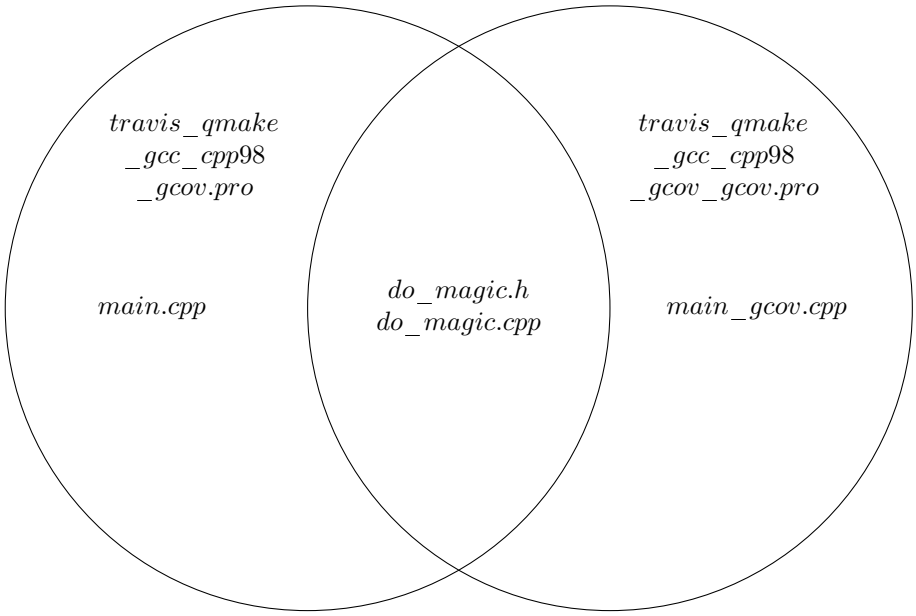


Figure 18: Venn diagram of the files uses in this build

Specifications The basic build has the following specs:

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++98
- Libraries: STL only
- Code coverage: yes
- Source: multiple files

Common files Both builds use the following code:

Algorithm 21 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 22 do_magic.cpp

```
#include "do_magic.h"

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

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

Algorithm 23 main.cpp

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

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

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

Algorithm 24 travis_qmake_gcc_cpp98_gcov.pro

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

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

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

Algorithm 25 main.cpp

```
#include "do_magic.h"

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

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

Algorithm 26 travis_qmake_gcc_cpp98_gcov.pro

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

# 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

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

Algorithm 27 build.sh

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

The new step is after having run the executable,

- gcov main_gcov.cpp

  Let gcov create a coverage report

- cat main_gcov.cpp.gcov

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

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

Algorithm 28 .travis.yml

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

This .travis.yml file has some new features:

Figure 19: Qt logo

- `sudo: true`

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

- `before_install: sudo pip install codecov`

  Travis will use pip to install codecov using super user rights

- `after_success: codecov`

  After the script has run successfully, codecov is called

4.7 Adding profiling

4.8 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.

Specifications

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

The single C++ source file used is:

Algorithm 29 main.cpp

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

int main()
{
  const std::string filename = "HelloWorld.png";
  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 30 travis_qmake_gcc_cpp98_qt.pro

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

SOURCES += main.cpp

RESOURCES += \
    travis_qmake_gcc_cpp98_qt.qrc
```

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:

Figure 20: R logo

Algorithm 31 build.sh

```
#!/bin/bash
qmake
make
./travis_qmake_gcc_cpp98_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 32 .travis.yml

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

This .travis.yml file has ...

4.9 Adding Rcpp

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

What is R? R is a programming language.

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

Specifications The chapter has the following specs:

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++98
- Libraries: STL and Rcpp

- Code coverage: none
- Source: multiple files

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

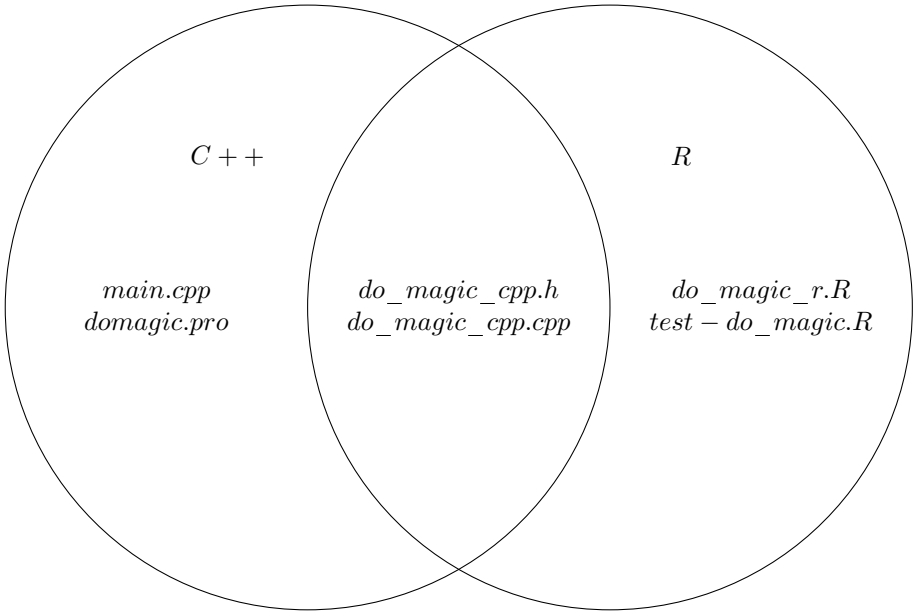


Figure 21: Venn diagram of the files uses in this build

4.9.1 C++ and R: the C++ function

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

Algorithm 33 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 34 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.

4.9.2 C++: main source file

The C++ program has a normal main function:

Algorithm 35 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.

4.9.3 C++: Qt Creator project file

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

Algorithm 36 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.

4.9.4 The C++ build script

The C++ build script is the regular canon of qmake, make and executable call.

Algorithm 37 build_cpp.sh

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

This scipt ...

4.9.5 R: the R function

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

Algorithm 38 R/do_magic_r.R

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

Next to this, it is just Roxygen2 documentation

4.9.6 R: The R tests

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

Algorithm 39 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.

4.9.7 The R script to install packages

The C++ build script is the regular canon of qmake, make and executable call.

Algorithm 40 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")
```

4.9.8 The Travis script

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

Algorithm 41 .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
 # 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.

4.9.9 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'
```

Figure 22: SFML logo

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

4.10 Adding the SFML library

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

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

Specifications

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

The single C++ source file used is:

Algorithm 42 main.cpp

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

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

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

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

Algorithm 43 travis_qmake_gcc_cpp98_sfml.pro

```
SOURCES += main.cpp

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

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

Algorithm 44 build.sh

```
#!/bin/bash
qmake
make
./travis_qmake_gcc_cpp98_sfml
```

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

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

Algorithm 45 .travis.yml

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

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

install:
  - sudo apt-get install libsfml-dev

script:
 - ./build.sh
```

This .travis.yml file has one new feature:

- install: sudo apt-get install libsfml-dev

  This makes Travis install the needed package.

Figure 23: Urho3D logo

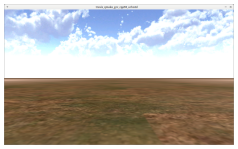


Figure 24: Screenshot of travis_qmake_gcc_cpp98_urho3d

4.11 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.

Specifications

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++98
- Libraries: STL and Urho3D
- Code coverage: none
- Source: multiple 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

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

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

Algorithm 46 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 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 47 build.sh

```
#!/bin/bash
./Urho3d.sh
#ln -s ./Urho3D/bin/Data
#ln -s ./Urho3D/bin/CoreData
qmake travis_qmake_gcc_cpp98_urho3d.pro
make
#./travis_qmake_gcc_cpp98_urho3d
```

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 48 .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 ...

4.12 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.

Specifications

- Build system: qmake

Figure 25: Wt logo

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

The single C++ source file used is:

Algorithm 49 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;
};
```

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

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

Algorithm 50 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.

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

Algorithm 51 build.sh

```
#!/bin/bash
qmake
make
# ./travis_qmake_gcc_cpp98_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:

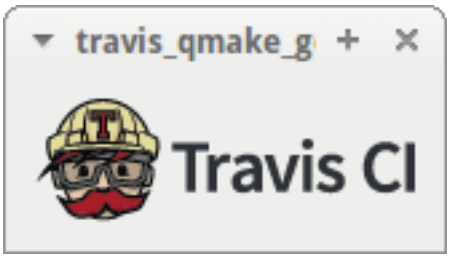


Figure 26: The application

Algorithm 52 .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 ...

5 Extending the build by two steps

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

- Use of C++11 and Boost: see chapter 5.2
- Use of C++11 and Boost.Test: see chapter 5.3
- Use of C++14 and Boost: see chapter 5.11

5.1 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 normal build is just that application.

The test 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.

5.1.1 What is QTest?

QTest is the Qt testing framework

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

The Boost.Test library (see chapter 5.3) 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 priviliges. 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.

5.1.3 The Travis file

Algorithm 53 .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.

5.1.4 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 54 build_test.sh

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

Algorithm 55 build_normal.sh

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

5.1.5 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 56 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 57 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
```

5.1.6 The source files

The dialog This is the source of dialog:

Algorithm 58 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 59 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 60 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 61 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 62 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 63 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.

5.2 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 64 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 65 travis_qmake_gcc_cpp11_boost.pro

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

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

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

The bash build script to build and run this:

Algorithm 66 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 67 .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 ...

5.3 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 5.1)

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:

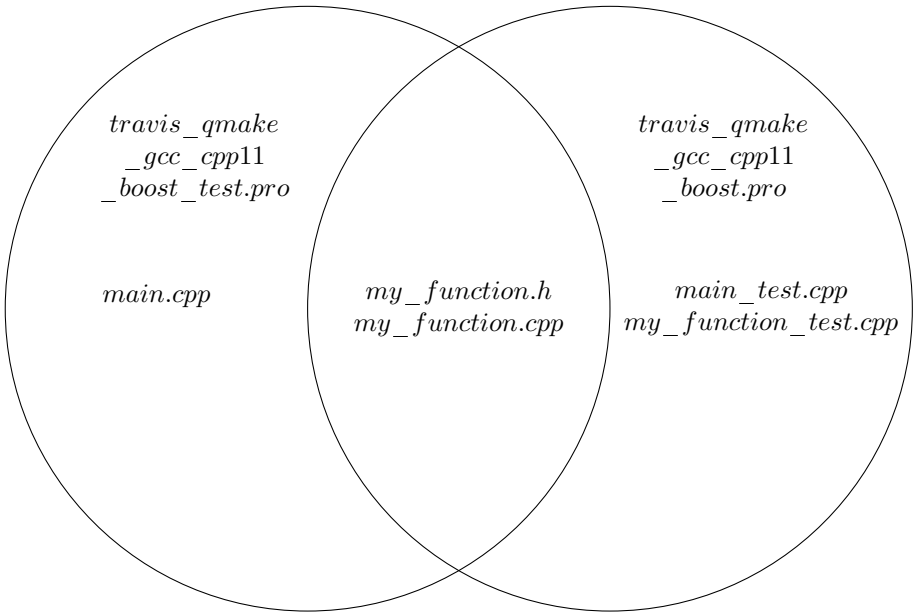


Figure 27: Venn diagram of the files uses in this build

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

5.3.1 The function

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

Algorithm 68 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 69 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

5.3.2 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 70 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 71 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 72 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'.

5.3.3 Exe build

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

Algorithm 73 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 74 travis_qmake_gcc_cpp11_boost_test.pro

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

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

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

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

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

5.3.4 Build script

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

Algorithm 75 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.

5.3.5 Travis script

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

Algorithm 76 .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 ...

5.4 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 77 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 78 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 79 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 80 .travis.yml

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

install:
  - sudo apt-get install clang

script:
  - ./build.sh
```

This .travis.yml file has ...

5.5 C++11 and gcov

In this example, the C++98 build with gcov (chapter 4.6) is extended by using C++11.

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:

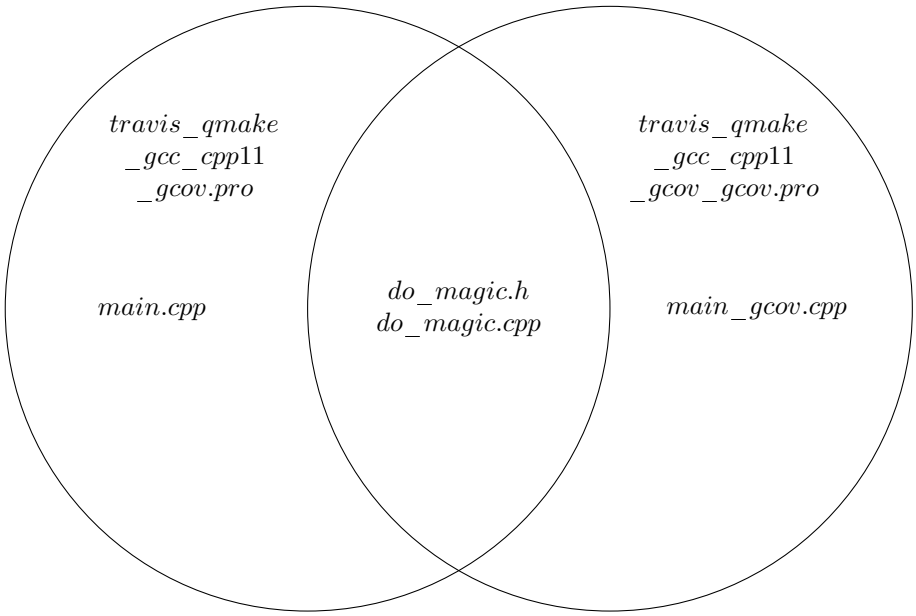


Figure 28: Venn diagram of the files uses in this build

Specifications The basic build has the following specs:

- Build system: qmake
- C++ compiler: gcc
- C++ version: C++11
- Libraries: STL only
- Code coverage: yes
- Source: multiple files

Common files Both builds use the following code:

Algorithm 81 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 82 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;
}
```

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

Algorithm 83 main.cpp

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

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

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

Algorithm 84 travis_qmake_gcc_cpp11_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
```

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

Algorithm 85 main.cpp

```
#include "do_magic.h"

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

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

Algorithm 86 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

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

Algorithm 87 build.sh

```
#!/bin/bash
echo "Normal run"
qmake travis_qmake_gcc_cpp11_gcov.pro
make
./travis_qmake_gcc_cpp11_gcov
./clean.sh
echo "Coverage run"
qmake travis_qmake_gcc_cpp11_gcov_gcov.pro
make
./travis_qmake_gcc_cpp11_gcov_gcov
gcov-5 main_gcov.cpp
gcov-5 do_magic.cpp
cat main_gcov.cpp.gcov
cat do_magic.cpp.gcov
```

The new step is after having run the executable,

- `gcov main_gcov.cpp`

  Let gcov create a coverage report

- `cat main_gcov.cpp.gcov`

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

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

Algorithm 88 .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

script: ./build.sh

after_success: codecov
```

This .travis.yml file has some new features:

- `sudo: true`

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

- `before_install: sudo pip install codecov`

  Travis will use pip to install codecov using super user rights

- `after_success: codecov`

  After the script has run successfully, codecov is called

5.6 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 89 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 90 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 91 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 92 .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 ...

5.7 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

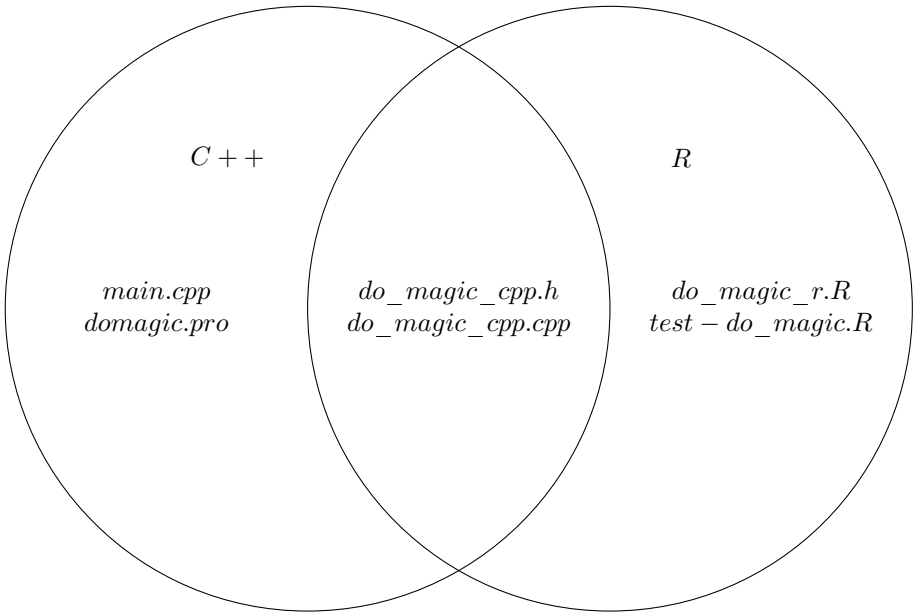


Figure 29: Venn diagram of the files uses in this build

5.7.1 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 93 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 94 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.

5.7.2 C++: main source file

The C++ program has a normal main function:

Algorithm 95 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.

5.7.3 C++: Qt Creator project file

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

Algorithm 96 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

5.7.4 C++: build script

The C++ bash build script is straightforward.

Algorithm 97 build_cpp.sh

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

This script is already described in the C++98 and Rcpp chapter (chapter 4.9, algorithm 37).

5.7.5 R: the R function

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

Algorithm 98 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.

5.7.6 R: The R tests

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

Algorithm 99 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.

5.7.7 R: script to install packages

Algorithm 100 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.

5.7.8 The Travis script

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

Algorithm 101 .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

5.8 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 102 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 103 travis_qmake_gcc_cpp11_sfml.pro

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

SOURCES += main.cpp

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

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

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 104 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 105 .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 ...

5.9 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 106 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 107 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 108 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 109 .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 ...

5.10 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 110 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 111 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 112 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 113 .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 ...

5.11 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 114 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 115 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 116 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 117 .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 ...

5.12 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:

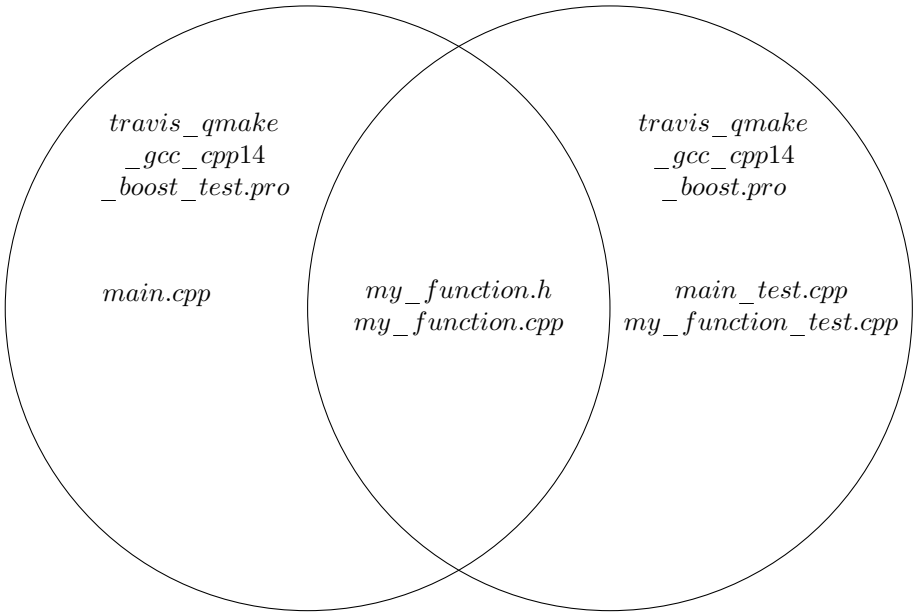


Figure 30: Venn diagram of the files uses in this build

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++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

5.12.1 The function

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

Algorithm 118 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 119 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

5.12.2 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 120 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 121 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 122 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'.

5.12.3 Exe build

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

Algorithm 123 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 124 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.

5.12.4 Build script

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

Algorithm 125 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 126 build_test.sh

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

5.12.5 Travis script

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

Algorithm 127 .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 ...

5.13 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, 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:

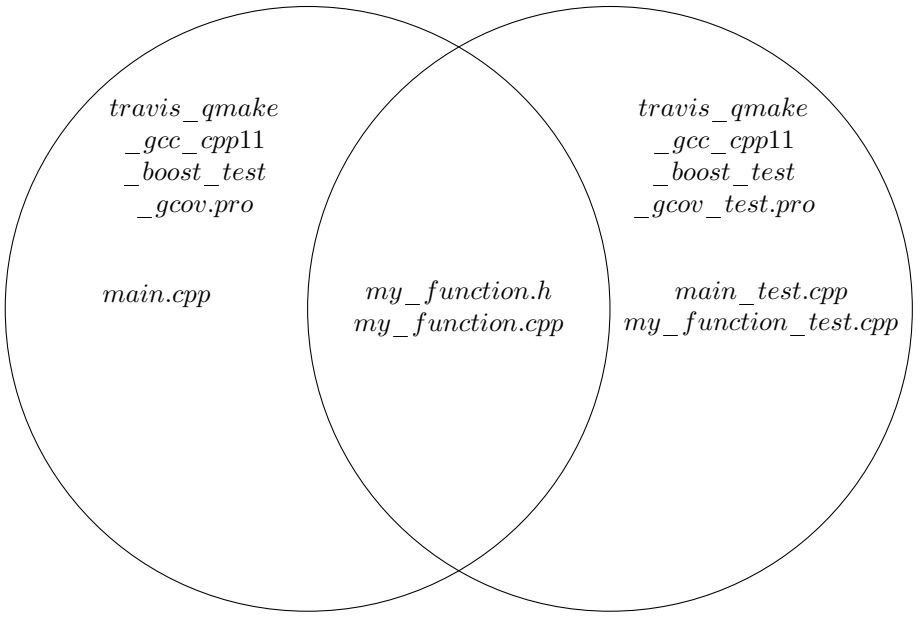


Figure 31: Venn diagram of the files uses in this build

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

6.1.1 The function

Same

6.1.2 Test build

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

Algorithm 128 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.

6.1.3 Normal build

The normal build is identical.

6.1.4 Build script

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

Algorithm 129 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.

6.1.5 Travis script

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

Algorithm 130 .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 ...

References

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

Index

#
    /bin/bash, 14
-Wall, 15
-Weffc++, 15
-Werror, 15
-Wextra, 15
.pro, 9

bash, 12
Boost, 21

C++11, 16
C++14, 19
C++98, 12
clang, 22
Codecov, 24

g++, 12
GCC, 12
gcov, 24
git, 6
GitHub, 5
GitHub, creating a repository, 6
GitHub, registration, 6

Hello world, 15

make, 10, 14
Makefile, 14

qmake, 10, 14
QMAKE_CXXFLAGS, 15
Qt, 29
Qt Creator, 9
Qt Creator project file, 9
Qt Creator, create new project, 9

R, 31
Rcpp, 31

SFML, 38
shebang, 14
SOURCES, 15
STL, 12

Urho3D, 40

Wt, 42

Yet Another Markup Language, 13
yml, 13