Continuous Integration INRIA

C++ Exercises

Vincent Rouvreau - https://sed.saclay.inria.fr [*]

February 28, 2017

Contents

C++ Exercises

1 Preamble

In this exercise, we will focus on the configuration of `Jenkins` for:

1. A simple aspect of C++ unit testing
2. An aspect of dynamic testing : source coverage
3. A demonstration of static analysis with `cppcheck`

To perform this C++ exercise, we rely on these development tools:

- A C++ compiler, for example `GNU C++` or `clang`.
- A build process manager: `CMake`
- A testing framework: `cppunit`
- A static analysis tool: `cppcheck`
- Coverage analysis tools: `gcov`, `lcov` and `gcvor`.
- A testing framework: `cppunit` ...

On Linux, BSD, MACOS X systems, suitable versions may simply be installed with the package manager (`yum`, `apt`, ...).
For example:

[*]This practical tutorial has been originally written by Maurice Brémond, Gaëtan Harter and David Parsons from SED INRIA Rhône-Alpes. I would like to thank them for their help and support.

- On Fedora 21 and earlier systems, one may use the command:
  `sudo yum install cmake gcc-c++ cppunit-devel cppcheck gcov lcov python-pip`
- On Fedora 22 and later systems, one may use the command:
  `sudo dnf install cmake gcc-c++ cppunit-devel cppcheck gcov lcov python-pip`
- On recent Debian or Ubuntu systems:
  `sudo apt-get install cmake gcc-c++ libcppunit-dev cppcheck gcov lcov python-pip`

For `gcovr`, use:
`sudo pip install gcovr`

2 Unit testing

2.1 Local setup

Clone the git repository from INRIA forge and create your own branch

First, you will create your personal `git` repository on the INRIA forge as a branch of the main repository for the `TPCISedSaclay` project:

- Go to https://gforge.inria.fr/projects/tpcisedsaclay
- Click on the **CODE SOURCE** or **SCM** tab
- Click on **Request a personal repository**
- Back to the **SCM** tab, look for the command to access your personal repository

WARNING : do not use the anonymous access (containing anon-scm)

Then on a terminal, clone the content of your personal `git` repository. The correct command should look like this:

```
git clone \
  git+ssh://<yourforgelogin>@scm.gforge.inria.fr/gitroot/
  tpcisedsaclay/users/<yourforgelogin>.git
cd <yourforgelogin>/cxx
```

Project file tree

A minimal `CMake` project is present under the file tree:

```
<yourforgelogin>/cxx
|_ CMakeLists.txt
|_ cmake
|  |_ TP.cmake
|_ Sphere.hpp
|_ Sphere.cpp
|_ bench.cpp
|_ tests
   |_ CMakeLists.txt
   |_ TestTP.hpp
   |_ TestTP.cpp
   |_ TestMain.cpp
```

This `CMake` project provides the framework needed to build and test a versionized, dynamic shared library named TP [1]. The `CMake` configuration is specified in two `CMakeLists.txt` files:

- One under the main directory: the main `CMakeLists.txt`
- One under the `tests` directory

The API provided by this library is at this level composed by a single `Sphere` class. The `Sphere` class offers an object constructor with the `radius` as parameter and a `volume` method.
A benchmark program named `bench.cpp` is also built and linked with the library. Although the computation is very simple, this benchmark program may be used to compare the influence of some compiler optimization flags.

Check that you can build the project and run the test

On a Unix system :

```
mkdir -p build      # Create a build directory
cd build
cmake ..            # Create the build environment with CMake
make                # Build the project
make test           # Run the test
```

[1] **TP** stands for **Travaux Pratiques** in french

Run the benchmark

`CMake` provides pre-defined build configurations that may be chosen with the `CMAKE BUILD TYPE variable`.
Among them we are going to consider:

- `Debug` build type which sets the debug flags (`-g` with `GNU C++`)
- `Release` build type which sets some optimization flags (`-O3 -DNDEBUG` with `GNU C++`)

Once a directory has been given as argument to CMake it remains in a cache, the CMakeCache.txt file in your build directory. This cache file is a text file and may be edited by hand.
CMake variables may be set on the command line with -D arguments:

```
cmake .. -D<VARIABLE_NAME>=<VALUE>
```

The value remains in the cache file, so when a variable has been modified once with a call to `CMake`, it is not necessary to define its value anymore on the command line.
To configure our build to produce a `Debug` type build:

```
cmake .. -DCMAKE_BUILD_TYPE=Debug
make
./bench
```

For an optimized build, we change its configuration to:

```
cmake .. -DCMAKE_BUILD_TYPE=Release
make
./bench
```

Over this very simple computation, the benchmark time difference between `Debug` and `Release` is not dramatic.
With `GNU C++` compiler, we can go beyond `-O3` optimization flag with the `fast-math` option, which implies `-funsafe-math-optimizations` and may break some of the requirements of IEEE and ANSI standards.
To pass a specific argument to the `GNU C++` compiler, we use the variable `CMAKE CXX FLAGS`. The arguments passed on the command line to this variable are added to the other compiler arguments. In doubt with the generated specification, one can use the argument `VERBOSE=1` with make tool, in order to see all the flags passed to the compiler.

```
cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_FLAGS=-ffast-math
make VERBOSE=1
./bench
```

You should see a difference on time.
Now, let's keep this configuration in our `build` directory.

2.2 Jenkins setup

Log into the project's Jenkins instance

1. Connect to the INRIA Continuous Integration Web portal : https://ci.inria.fr/.
2. Log in and click **Dashboard** in the top menu
3. As you have been added to project **TPCISedSaclay**, click on the **Jenkins** button. You may be prompted to log into `Jenkins`, use the same login/passwd as for the `ci.inria.fr` portal.

Running your first test with Jenkins:

- From our Jenkins dashboard page, click **New Item** in the menu on the left
- Provide a name (`<yourforgelogin>` for instance) for this new item (avoid spaces since it is likely to lead to errors) and select **Freestyle project**.

`Git` configuration :

- In the new item's configuration page (which you will be redirected to after clicking **OK**), choose **Git** as your **Source Code Manager**
- Copy the anonymous URL to your personal repository into the **Repository URL** field:

```
https://scm.gforge.inria.fr/anonscm/git/tpcisedsaclay/users/<yourforgelogin>.git
```

An important step for the continuous integration setup is the build trigger. A simple option is to choose to build periodically : this is suitable for some nightly or weekly tests that may be time consuming and are not meant to be launched after each commit, but this should be avoided for short periods.
In our case, we want the results to be displayed as soon as possible, so we choose to launch the build after a `post-commit` hook [2]:

[2] It is explained in ci.inria.fr FAQ documentation

- Click on **Poll SCM**
- In the **Schedule** field, cut and paste:

```
# Leave empty. We don't poll periodically, but need
# polling enabled to let HTTP trigger work
```

Click on **Save** button.

create the post-commit hook on the INRIA forge server

For this, you can copy the following file:

```
$ ssh <yourforgelogin>@scm.gforge.inria.fr
$ cp /gitroot/tpcisedsaclay/users/vrouvrea.git/hooks/post-receive \
    /gitroot/tpcisedsaclay/users/<yourforgelogin>.git/hooks/
```

And then modify the post-commit hook `post-receive` with your personal repository:

```
#!/bin/sh
wget -q -O - --auth-no-challenge --no-check-certificate \
  http://ci.inria.fr/tpcisedsaclay/git/notifyCommit?url=
  https://scm.gforge.inria.fr/anonscm/git/tpcisedsaclay/users/<yourforgelogin>.git
```

2.3 Exercise 1

On the Jenkins dashboard, click on little triangle close to your newly created item to populate the actions window.
Click on **Configure** button.
Add a **new build step** using **Execute shell**, where you inform `Jenkins` how to build and test the project:

- You have to choose the `Debug` configuration for this build, this will be needed for coverage as we will see later.
  In the shell working directory (the result of the command `/bin/pwd` in this shell) `Jenkins` has cloned your source directory.

- Then save the project and verify the configuration with a click on **Build Now** in the menu on the left.
- In the **Build History** on the left, click on the last build (hopefully #1), then select `Console Output`.
- You can also check the **Test Result**.

2.4 Exercise 2

First part

Edit the implementation of the `Sphere` class in the `<yourforgelogin>/c++/src/Sphere.cpp` file and have a look at the code of the method `Sphere::volume()`.

```
double Sphere::volume() const
{
  return 4 * M_PI * pow (this->_radius, 3.) / 3.;
}
```

This is the method which is tested in the unique test of the library, and the test is implemented in the file `<yourforgelogin>/c++/src/tests/testTP.cpp`. Let's imagine we want to improve the efficiency of the `Sphere::volume()` method by pre-computing the value `4 * Math.PI / 3`
TODO :

- Extract this value into a class data member.
- Run the benchmark. This may show only a very, very minimal improvement !
- Run the test.

The test may still be successful : this may depend on your hardware and compiler.
Let's assume it is successful : do not forget to commit your modifications.

```
git commit -a -m "precomputation of 4pi/3"
git push
```

The test should now fail on `Jenkins`.
Here is the output you can see on `Jenkins` console:

```
+ make test
Running tests...
Test project /builds/workspace/mb-cxx/build
Start 1: TestVolume
1/1 Test #1: TestVolume .......................***Failed

0.00 sec

0% tests passed, 1 tests failed out of 1
Total Test time (real) =

0.00 sec

The following tests FAILED:
1 - TestVolume (Failed)
```

Second part

The `-funsafe-math-optimizations` implied by `-ffast-math` allows for re-ordering of floating points operations and this may lead to different results.
The direct check of equality of floating point numbers with the == operator should be avoided : so we can add a warning (only if understood by the compiler) in order not to reproduce this kind of error.
Since version 4.6, `GNU C++` provides this warning `-Wfloat-equal`. With portability in mind, before adding the flag to the compiler command, we need to check that the compiler accepts it.
With CMake, there is no built-in function for this operation, but that can be achieved with a simple macro:

```
include(TestCXXAcceptsFlag)
macro(add_cxx_compiler_flag _flag)
  string(REPLACE "-" "_" _flag_var ${_flag})
  check_cxx_accepts_flag("${_flag}" CXX_COMPILER_${_flag_var}_OK)
  if (CXX_COMPILER_${_flag_var}_OK)
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${_flag}")
  endif()
endmacro()
```

This macro is provided in the TP.cmake module file under cmake directory. To load this TP module, you need to modify the CMakelists.txt file:

- Set the `CMAKE_MODULE_PATH` to this directory:

```
set(CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/cmake)
```

- Load the module TP. Beware : this should be done before using the macro:

```
include(TP)
```

On `Jenkins` side, compiler warnings can be checked in the **Post build Action** step of your build project.
TODO :

- In the `CMake` configuration file, `CMakeLists.txt`, include the TP module.
- With the provided `CMake` TP macro, add the compiler warning to the `CMake` configuration,
- Verify that the warning is printed during compilation.
- In `Jenkins` :
  - Add the scan for compiler warnings in the **Post build Action** of your project.
  - In the **Execute shell** section, comment the line `make test`. This is necessary because as the test is failing, the parsing will not be done by `Jenkins`.
- Commit the code in order to check the parsing done by `Jenkins`
- If it goes well, then uncomment the `make test` line.
- Modify the appropriate test in `TestTP.cpp` so small rounding differences do not cause errors.
  `cppunit` provides a macro for this:

```
CPPUNIT_ASSERT_DOUBLES_EQUAL(expected, actual, delta)
```

- Once it is OK and the warning has gone, commit.
- Rebuild on `Jenkins` to check.

3 Code coverage

Here we get a taste of some dynamics analysis. After the tests are executed, the code coverage tools (mainly `gcov`) show which lines of code have been involved in the execution.

3.1 Gcov, lcov, gcovr

Coverage with `gcov` needs the `GNU C++` compiler and the `-fprofile-arcs` and `-ftest-coverage` compilation options.
`gcov` generates raw output. We are going to use the `lcov` utility for the post-processing and the generation of html pages and the `gcovr` utility in order to present parseable results to `Jenkins`.
A `CMake` function `add_test_with_coverage` is provided in the TP module to simplify the whole setup.
This function, when used in place of the standard `CMake` function `add_test` function, provides coverage support through a new `make coverage` target.
Remark :

- You can see all `make` targets using : `make help`.

3.2 Exercise 3

TODO :

- Install coverage for the test routine `TestVolume`. In the `CMake` configurations files, you need to set the coverage flags for the build of the library and the test file with the `add_cxx_compiler_flag`.
- Run a test with `make coverage`
- Open the file `index.html` under `TestTP__testVolume` directory with a Web browser to see the result
- On the `Jenkins` side :
  - Modify the build process in **Execute shell** to add coverage target
  - In **Post-Build Action**, add **Publish Cobertura Report** with the Cobertura xml report pattern set to : `**/coverage.xml`
- Commit
- Generate and visualize the corresponding coverage report.

3.3 Exercise 3bis (optional)

This is optional, if you prefer, go directly to the static analysis section in page 9.
Let's check if it works with more than one C++ class. Use another development branch to add a new class `Alphabet` to our project:

```
git merge origin/alpha-cxx
```

You should retrieve the following file tree :

```
<yourforgelogin>/cxx
|_ CMakeLists.txt
|_ cmake
|  |_ TP.cmake
|_ Alphabet.hpp
|_ Alphabet.cpp
|_ Sphere.hpp
|_ Sphere.cpp
|_ bench.cpp
|_ tests
   |_ CMakeLists.txt
   |_ TestTP.hpp
   |_ TestTP.cpp
   |_ TestMain.cpp
```

TODO :

- Resolve git merge conflicts (keep the flags you added, use the new library generation dependency)
- Generate and visualize the corresponding coverage report
- Improve the test coverage and commit.

4 Static analysis

With our library, one can check that the following code is valid but it may not be what the programmer intended:

```
Sphere s1(.1);
Sphere s2(.2);
std::pair<Sphere, Sphere> p(s1,2);
```

As the implicit conversion is allowed on the `Sphere` constructor, the pair of `Sphere` objects `p` is composed of the `Sphere s1` and the `Sphere(2)` which is not the same as a pair of `s1` and `s2`. In the case of a typo in the code source, this may certainly lead to bugs.
A static code analysis may help in the discovery of those potentials bugs.

4.1 cppcheck

`cppcheck` is a command-line tool dedicated to static C/C++ code analysis. It tries to detect bugs that your C/C++ compiler does not see. It is versatile, and can check non-standard code including various compiler extensions, inline assembly code, etc. Its internal preprocessor can handle includes, macros, and several preprocessor commands.
Applying this tool on our project, we get an analysis report using the command:

```
cppcheck <your source directory> -f -q --enable=style
```

Beware : The results we obtain depend on the version of `cppcheck` !
On the virtual machine configured for ci.inria.fr/tpcisedsaclay, cppcheck version 1.61 is installed. It is not the latest version.

4.2 Exercise 4

TODO :

- Test the command given below on your project locally
- Install this check on `Jenkins` :
  - Modify the **Build** section **Execute shell field** to call `cppcheck`
  - Add a **Post-build Action** for publishing `cppcheck` results

  Beware : change the `cppcheck` command argument so it generates an XML output (use `--xml` option) and redirect the standard error to a file named `cppcheck-result.xml`.
  `Jenkins` can read this file if configured properly and if the appropriate **post-build** action is set.
- Fix some warnings and commit.