KAIZOU TOOLS DEMOS ?

Unit testing with GoogleTest and CMake

05 Nov 2014 by David Corvoysier

Continuous integration requires a robust test environment to be able to detect regressions as early as possible.

A typical test environment will typically be composed of integration tests of the whole system and unit tests per components.

This post explains how to create unit tests for a `C++` component using **GoogleTest** and **CMake**.

##Project structure

I will assume here that the project structure follows the model described in a previous post:

```
+-- CMakeLists.txt
+-- main
|    +-- CMakeLists
|    +-- main.cpp
|
+-- test
|    +-- CMakeLists.txt
|    +-- testfoo
|       +-- CMakeLists.txt
|       +-- main.cpp
|       +-- testfoo.h
|       +-- testfoo.cpp
|       +-- mockbar.h
|
+-- libfoo
|    +-- CMakeLists.txt
|    +-- foo.h
|    +-- foo.cpp
```

```
|
+-- libbar
    +-- CMakeLists.txt
    +-- bar.h
    +-- bar.cpp
```

The `main` subdirectory contains the main project target, an executable providing the super-useful `libfoo` service using the awesome `libbar` backend (for example `libfoo` could be a generic face recognition library and `libbar` a GPU-based image processing library).

The `test` directory contains a single executable allowing to test the `libfoo` service using a *mock* version of `libbar`.

> From Wikipedia: In object-oriented programming, mock objects are simulated objects that mimic the behavior of real objects in controlled ways.

For those interested, the code for this sample project is on github.

##A closer look at the test directory

In my simplistic example, there is only one subdirectory under `test`, but in a typical project, it would contain several subdirectories, one for each test program.

Tests programs are based on Google's Googletest framework and its GoogleMock extension.

Since all test programs will be using these packages, the root `CMakeLists.txt` file should contain all directives required to resolve the corresponding dependencies. This is where things get a bit hairy, since Google does not recommend to install these packages in binary form, but instead to recompile them with your project.

###Resolving GoogleTest and GoogleMock dependencies

There are at least three options to integrate your project with **GoogleTest** and **GoogleMock**.

####Having both packages integrated in your build system

Obviously, this is only an option if you actually *do* have a buildsystem, but if this is the case, this would be my recommendation.

Depending on how your buildsystem is structured, your mileage may vary, but in the end you should be able to declare **GoogleTest** and **GoogleMock** as dependencies using `CMake` functions like the built-in `find_package` or the `pkg-config` based `pkg_check_modules`.

```
find_package(PkgConfig)
pkg_check_modules(GTEST REQUIRED gtest>=1.7.0)
pkg_check_modules(GMOCK REQUIRED gmock>=1.7.0)

include_directories(
    ${GTEST_INCLUDE_DIRS}
    ${GMOCK_INCLUDE_DIRS}
)
```

####Add both packages sources to your project

Adding the **GoogleTest** and **GoogleMock** sources as subdirectories of `test` would allow you to compile them as part of your project.

This is however really ugly, and I wouldn’t recommend you doing that …

####Add both packages as external CMake projects

According to various answers posted on StackOverflow, this seems to be the recommended way of resolving **GoogleTest** and **GoogleMock** dependencies on a per project basis.

It takes advantage of the `CMake` `ExternalProject` module to fetch **GoogleTest** and **GoogleMock** sources from the internet and compile them as third-party dependencies in your project.

Below is a working example, with a few comments explaining what’s going on:

```
# We need thread support
find_package(Threads REQUIRED)

# Enable ExternalProject CMake module
include(ExternalProject)

# Download and install GoogleTest
ExternalProject_Add(
    gtest
    URL https://github.com/google/googletest/archive/master.zip
    PREFIX ${CMAKE_CURRENT_BINARY_DIR}/gtest
    # Disable install step
    INSTALL_COMMAND ""
)
```

```
# Get GTest source and binary directories from CMake project
ExternalProject_Get_Property(gtest source_dir binary_dir)

# Create a libgtest target to be used as a dependency by test programs
add_library(libgtest IMPORTED STATIC GLOBAL)
add_dependencies(libgtest gtest)

# Set libgtest properties
set_target_properties(libgtest PROPERTIES
    "IMPORTED_LOCATION" "${binary_dir}/googlemock/gtest/libgtest.a"
    "IMPORTED_LINK_INTERFACE_LIBRARIES" "${CMAKE_THREAD_LIBS_INIT}"
)

# Create a libgmock target to be used as a dependency by test programs
add_library(libgmock IMPORTED STATIC GLOBAL)
add_dependencies(libgmock gtest)

# Set libgmock properties
set_target_properties(libgmock PROPERTIES
    "IMPORTED_LOCATION" "${binary_dir}/googlemock/libgmock.a"
    "IMPORTED_LINK_INTERFACE_LIBRARIES" "${CMAKE_THREAD_LIBS_INIT}"
)

# I couldn't make it work with INTERFACE_INCLUDE_DIRECTORIES
include_directories("${source_dir}/googletest/include"
                    "${source_dir}/googlemock/include")
```

> Note: It should theoretically be possible to set the **GoogleTest** and **GoogleMock** include directories as target properties using the INTERFACE_INCLUDE_DIRECTORIES variable, but it fails because these directoires don’t exist yet when they are declared. As a workaround, I had to explicitly use include_directories to specify them.

###Writing a **testfoo** test program for **libfoo**

The **testfoo** program depends on **libfoo**, **GoogleTest** and **GoogleMock**.

Here is how the **testfoo** `CMakeLists.txt` file would look like:

```
file(GLOB SRCS *.cpp)

add_executable(testfoo ${SRCS})

target_link_libraries(testfoo
    libfoo
    libgtest
    libgmock
)

install(TARGETS testfoo DESTINATION bin)
```

The libraries required for the build are listed under `target_link_libraries`. CMake will then add the appropriate include directories and link options.

The **testfoo** program will provide unit tests for the `Foo` class of the **libfoo** library defined below.

`foo.h`

```
class Bar;

class Foo
{
    Foo(const Bar& bar);
    bool baz(bool useQux);
protected:
    const Bar& m_bar;
}
```

`foo.cpp`

```
#include "bar.h"
#include "foo.h"

Foo::Foo(const bar& bar)
 : m_bar(bar) {};

bool Foo::baz(bool useQux) {
    if (useQux) {
        return m_bar.qux();
    } else {
        return m_bar.norf();
    }
}
```

The sample Test program described in the GoogleTest Documentation fits in a single file, but I prefer splitting the Unit Tests code in three types of files.

`main.cpp`

The `main.cpp` file will contain only the test program `main` function. This is where you will put the generic **Googletest** Macro invocation to launch the tests and some initializations that need to be put in the `main` (nothing in this particular case).

```
#include "gtest/gtest.h"

int main(int argc, char **argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    int ret = RUN_ALL_TESTS();
    return ret;
}
```

`testfoo.h`

This file contains the declaration of the `FooTest` class, which is the test fixture for the `Foo` class.

```
#include "gtest/gtest.h"
#include "mockbar.h"

// The fixture for testing class Foo.
class FooTest : public ::testing::Test {

protected:

    // You can do set-up work for each test here.
    FooTest();

    // You can do clean-up work that doesn't throw exceptions here.
    virtual ~FooTest();

    // If the constructor and destructor are not enough for setting up
    // and cleaning up each test, you can define the following methods:
```

```
    // Code here will be called immediately after the constructor (right
    // before each test).
    virtual void SetUp();

    // Code here will be called immediately after each test (right
    // before the destructor).
    virtual void TearDown();

    // The mock bar library shaed by all tests
    MockBar m_bar;
};
```

`mockbar.h`

Assuming the **libbar** library implements a public `Bar` interface, we use **GoogleMock** to provide a fake implementation for test purposes only:

```
#include "bar.h"

class MockBar: public Bar
{
    MOCK_METHOD0(qux, bool());
    MOCK_METHOD0(norf, bool());
}
```

This will allow us to inject controlled values into the **libfoo** library when it will invoke the `Bar` class methods.

> Please refer to the GoogleMock documentation for a detailed description of the `GoogleMock` features.

`testfoo.cpp`

This file contains the implementation of the `TestFoo` fixture class.

This is where the actual tests are written.

We will test the output of the `Foo::baz()` method, first having default values for the `Bar::qux()` and `Bar::norf()` methods returned by our mock, then overrding the value returned by `Bar::norf()` with a value specific to our test.

In all test cases, we use **GoogleTest** expectations to verify the output of the `Foo::baz` method.

```
#include "mockbar.h"
#include "testfoo.h"

using ::testing::Return;

FooTest()
{
    // Have qux return true by default
    ON_CALL(m_bar,qux()).WillByDefault(Return(true));
    // Have norf return false by default
    ON_CALL(m_bar,norf()).WillByDefault(Return(false));
}

TEST_F(FooTest, ByDefaultBazTrueIsTrue) {
    Foo foo(m_bar);
    EXPECT_EQ(foo.baz(true), true);
}

TEST_F(FooTest, ByDefaultBazFalseIsFalse) {
    Foo foo(m_bar);
    EXPECT_EQ(foo.baz(false), false);
}

TEST_F(FooTest, SometimesBazFalseIsTrue) {
    Foo foo(m_bar);
    // Have norf return true for once
    EXPECT_CALL(m_bar,norf()).WillOnce(Return(true));
    EXPECT_EQ(foo.baz(false), false);
}
```

> Please refer to the GoogleTest documentation for a much detailed presentation of how to create unit tests with Gtest.

##Building tests

As usual, it is recommended to build your program out-of-tree, ie in a directory separated from the sources.

```
mkdir build
cd build
```

First, you need to invoke the `cmake` command to generate the build files.

```
cmake ..
```

This should produce an output similar to this one:

```
-- The C compiler identification is GNU 4.8.2
-- The CXX compiler identification is GNU 4.8.2
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Looking for include file pthread.h
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: ~/gtest-cmake-example/build
```

Then, build the project targets.

```
make
```

The following output corresponds to the case where **GoogleTest** and **GoogleMock** are automatically fetched from their repositories and built as third-party dependencies.

```
Scanning dependencies of target libfoo
[  7%] Building CXX object libfoo/CMakeFiles/libfoo.dir/foo.cpp.o
Linking CXX static library liblibfoo.a
[  7%] Built target libfoo
Scanning dependencies of target libbar
[ 15%] Building CXX object libbar/CMakeFiles/libbar.dir/bar.cpp.o
```

```
Linking CXX static library liblibbar.a
[ 15%] Built target libbar
Scanning dependencies of target myApp
[ 23%] Building CXX object main/CMakeFiles/myApp.dir/main.cpp.o
Linking CXX executable myApp
[ 23%] Built target myApp
Scanning dependencies of target gtest
[ 30%] Creating directories for 'gtest'
[ 38%] Performing download step (download, verify and extract) for 'gtest'
-- downloading...
   src='https://github.com/google/googletest/archive/master.zip'
   dst='/home/david/perso/gtest-cmake-example/build/test/gtest/src/master.zip'
   timeout='none'
-- [download 0% complete]
-- [download 1% complete]
...
-- [download 99% complete]
-- [download 100% complete]
-- downloading... done
-- verifying file...
   file='/home/david/perso/gtest-cmake-example/build/test/gtest/src/master.zip'
-- verifying file... warning: did not verify file - no URL_HASH specified?
-- extracting...
   src='/home/david/perso/gtest-cmake-example/build/test/gtest/src/master.zip'
   dst='/home/david/perso/gtest-cmake-example/build/test/gtest/src/gtest'
-- extracting... [tar xfz]
-- extracting... [analysis]
-- extracting... [rename]
-- extracting... [clean up]
-- extracting... done
[ 46%] No patch step for 'gtest'
[ 53%] No update step for 'gtest'
[ 61%] Performing configure step for 'gtest'
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Found PythonInterp: /usr/bin/python (found version "2.7.6")
-- Looking for include file pthread.h
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /home/david/perso/gtest-cmake-example/build/test/gtest/src/gtest-build
[ 69%] Performing build step for 'gtest'
Scanning dependencies of target gmock
[ 14%] Building CXX object googlemock/CMakeFiles/gmock.dir/__/googletest/src/gtest-all.cc.o
[ 28%] Building CXX object googlemock/CMakeFiles/gmock.dir/src/gmock-all.cc.o
Linking CXX static library libgmock.a
[ 28%] Built target gmock
Scanning dependencies of target gmock_main
[ 42%] Building CXX object googlemock/CMakeFiles/gmock_main.dir/__/googletest/src/gtest-all.cc.o
[ 57%] Building CXX object googlemock/CMakeFiles/gmock_main.dir/src/gmock-all.cc.o
[ 71%] Building CXX object googlemock/CMakeFiles/gmock_main.dir/src/gmock_main.cc.o
Linking CXX static library libgmock_main.a
[ 71%] Built target gmock_main
Scanning dependencies of target gtest
[ 85%] Building CXX object googlemock/gtest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
Linking CXX static library libgtest.a
[ 85%] Built target gtest
Scanning dependencies of target gtest_main
[100%] Building CXX object googlemock/gtest/CMakeFiles/gtest_main.dir/src/gtest_main.cc.o
Linking CXX static library libgtest_main.a
[100%] Built target gtest_main
[ 76%] No install step for 'gtest'
[ 84%] Completed 'gtest'
[ 84%] Built target gtest
Scanning dependencies of target testfoo
[ 92%] Building CXX object test/testfoo/CMakeFiles/testfoo.dir/main.cpp.o
[100%] Building CXX object test/testfoo/CMakeFiles/testfoo.dir/testfoo.cpp.o
Linking CXX executable testfoo
[100%] Built target testfoo
```

##Running tests

Once the test programs have been built, you can run them individually …

```
test/testfoo/testfoo
```

… producing a detailed output …

```
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from FooTest
[ RUN      ] FooTest.ByDefaultBazTrueIsTrue

GMOCK WARNING:
Uninteresting mock function call - taking default action specified at:
~/gtest-cmake-example/test/testfoo/testfoo.cpp:8:
    Function call: qux()
          Returns: true
Stack trace:
[       OK ] FooTest.ByDefaultBazTrueIsTrue (0 ms)
[ RUN      ] FooTest.ByDefaultBazFalseIsFalse

GMOCK WARNING:
Uninteresting mock function call - taking default action specified at:
~/gtest-cmake-example/test/testfoo/testfoo.cpp:10:
    Function call: norf()
          Returns: false
Stack trace:
[       OK ] FooTest.ByDefaultBazFalseIsFalse (0 ms)
[ RUN      ] FooTest.SometimesBazFalseIsTrue
[       OK ] FooTest.SometimesBazFalseIsTrue (0 ms)
[----------] 3 tests from FooTest (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 3 tests.
```

> Note: You can get rid of **GoogleMock** warnings by using a **nice mock**.

… or globally through `CTest` …

```
make test
```

… producing only a test summary.

```
Running tests...
Test project ~/gtest-cmake-example/build
    Start 1: testfoo
1/1 Test #1: testfoo .........................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.00 sec
```

comments powered by Disqus

Hi

I am David Corvoysier, versatile developer and open Source enthusiast.

Follow me:

Related Posts

Explore Tensorflow features with the CIFAR10 dataset

26 Jun 2017 by David Corvoysier

The reason I started using Tensorflow was because of the limitations of my experiments so far, where I had coded my models from scratch following the guidance of the [CNN for visual recognition](http://cs231n.github.io/) course. I already knew how CNN worked, and had already a good experience of what it takes to train a good model. I had also read a lot of papers presenting multiple variations of CNN topologies, those aiming at increasing accuracy like those aiming at reducing model complexity and size. I work in the embedded world, so performance is obviously one of my primary concern, but I soon realized that the CNN state of the art for computer vision had not reached a consensus yet on the best compromise between accuracy and performance.

(more…)

Categories: Machine Learning Tags: tensorflow CIFAR10 CNN

Build and boot a minimal Linux system with qemu

23 Sep 2016 by David Corvoysier

When you want to build a Linux system for an embedded target these days, it is very unlikely that you decide to do it from scratch. Embedded Linux build systems are really smart and efficients, and will fit almost all use cases: should you need only a simple system, [buildroot](https://buildroot.org/) should be your first choice, and if you want to include more advanced features, or even create a full distribution, [Yocto](https://www.yoctoproject.org/) is the way to go. That said, even if these tools will do all the heavy-lifting for you, they are not perfect, and if you are using less common configurations, you may stumble upon issues that were not expected. In that case, it may be important to understand what happens behind the scenes. In this post, I will describe step-by-step how you can build a minimal Linux system for an embedded target and boot it using [QEMU](http://wiki.qemu.org/Main_Page).

(more…)

Categories: System Tags: build linux croostool-ng qemu busybox

Benchmarking build systems for a large C project

01 Sep 2016 by David Corvoysier

The performance of build systems has been discussed at large in the developer community, with a strong emphasis made on the limitations of the legacy Make tool when dealing with large/complex projects. I recently had to develop a build-system to create firmwares for embedded targets from more than 1000 source files. The requirements were to use build recipes that could be customized for each directory and file in the source tree, similar to what the Linux Kernel does with [kbuild](https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt). I designed a custom recursive Make solution inspired by [kbuild](https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt).

(more…)

Categories: System Tags: build make ninja kbuild cmake

Decentralized modules declarations in C using ELF sections

17 Aug 2016 by David Corvoysier

In modular programming, a standard practice is to define common interfaces allowing the same type of operation to be performed on a set of otherwise independent modules. ~~~~ modules = [a,b,...] for each m in modules: m.foo m.bar ~~~~ To implement this pattern, two mechanisms are required: - instantiation, to allow each module to define an 'instance' of the common interface, - registration, to allow each module to 'provide' this instance to other modules. Instantiation is typically supported natively in high-level languages. Registration is more difficult and usually requires specific code to be written, or relying on external frameworks. Let's see how these two mechanisms can be implemented for C programs.

(more…)

Categories: System Tags: ELF sections

Tag cloud

SVG XHTML Cache HTTP HTML HTML5 Video CSS3 Javascript Animation Benchmark Carousel Canvas video cross-compilation DBus CMake build