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 and GoogleMock frameworks.

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://googletest.googlecode.com/files/gtest-1.7.0.zip
    PREFIX ${CMAKE_CURRENT_BINARY_DIR}/gtest
    # Disable install step
    INSTALL_COMMAND ""
)

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

# Set gtest properties
ExternalProject_Get_Property(gtest source_dir binary_dir)
set_target_properties(libgtest PROPERTIES
    "IMPORTED_LOCATION" "${binary_dir}/libgtest.a"
    "IMPORTED_LINK_INTERFACE_LIBRARIES" "${CMAKE_THREAD_LIBS_INIT}"
#    "INTERFACE_INCLUDE_DIRECTORIES" "${source_dir}/include"
)
# I couldn't make it work with INTERFACE_INCLUDE_DIRECTORIES
include_directories("${source_dir}/include")
```

```
# Download and install GoogleMock
ExternalProject_Add(
    gmock
    URL https://googlemock.googlecode.com/files/gmock-1.7.0.zip
    PREFIX ${CMAKE_CURRENT_BINARY_DIR}/gmock
    # Disable install step
    INSTALL_COMMAND ""
)

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

# Set gmock properties
ExternalProject_Get_Property(gmock source_dir binary_dir)
set_target_properties(libgmock PROPERTIES
    "IMPORTED_LOCATION" "${binary_dir}/libgmock.a"
    "IMPORTED_LINK_INTERFACE_LIBRARIES" "${CMAKE_THREAD_LIBS_INIT}"
#    "INTERFACE_INCLUDE_DIRECTORIES" "${source_dir}/include"
)
# I couldn't make it work with INTERFACE_INCLUDE_DIRECTORIES
include_directories("${source_dir}/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
    ${GTEST_LIBRARIES}
    ${GMOCK_LIBRARIES}
)

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
[  4%] Building CXX object libfoo/CMakeFiles/libfoo.dir/foo.cpp.o
Linking CXX static library liblibfoo.a
[  4%] Built target libfoo
Scanning dependencies of target libbar
[  9%] Building CXX object libbar/CMakeFiles/libbar.dir/bar.cpp.o
Linking CXX static library liblibbar.a
[  9%] Built target libbar
Scanning dependencies of target myApp
[ 14%] Building CXX object main/CMakeFiles/myApp.dir/main.cpp.o
Linking CXX executable myApp
[ 14%] Built target myApp
Scanning dependencies of target gmock
[ 19%] Creating directories for 'gmock'
[ 23%] Performing download step (download, verify and extract) for 'gmock'
-- downloading...
     src='https://googlemock.googlecode.com/files/gmock-1.7.0.zip'
     dst='~/gtest-cmake-example/build/test/gmock/src/gmock-1.7.0.zip'
     timeout='none'
-- [download 0% complete]
-- [download 1% complete]
...
-- [download 99% complete]
-- [download 100% complete]
-- downloading... done
-- verifying file...
     file='~/gtest-cmake-example/build/test/gmock/src/gmock-1.7.0.zip'
-- verifying file... warning: did not verify file - no URL_HASH specified?
-- extracting...
     src='~/gtest-cmake-example/build/test/gmock/src/gmock-1.7.0.zip'
     dst='~/gtest-cmake-example/build/test/gmock/src/gmock'
-- extracting... [tar xfz]
...
-- extracting... done
[ 28%] No patch step for 'gmock'
[ 33%] No update step for 'gmock'
[ 38%] Performing configure step for 'gmock'
-- The CXX compiler identification is GNU 4.8.2
...
-- Generating done
-- Build files have been written to: ~/gtest-cmake-example/build/test/gmock/src/gmock-build
[ 42%] Performing build step for 'gmock'
...
[ 52%] Built target gmock
Scanning dependencies of target gtest
[ 57%] Creating directories for 'gtest'
[ 61%] Performing download step (download, verify and extract) for 'gtest'
-- downloading...
     src='https://googletest.googlecode.com/files/gtest-1.7.0.zip'
     dst='~/gtest-cmake-example/build/test/gtest/src/gtest-1.7.0.zip'
     timeout='none'
-- [download 0% complete]
-- [download 1% complete]
...
```

```
-- [download 98% complete]
-- [download 100% complete]
-- downloading... done
-- verifying file...
     file='~/gtest-cmake-example/build/test/gtest/src/gtest-1.7.0.zip'
-- verifying file... warning: did not verify file - no URL_HASH specified?
-- extracting...
...
-- extracting... done
[ 66%] No patch step for 'gtest'
[ 71%] No update step for 'gtest'
[ 76%] Performing configure step for 'gtest'
-- The CXX compiler identification is GNU 4.8.2
...
-- Build files have been written to: ~/gtest-cmake-example/build/test/gtest/src/gtest-build
[ 80%] Performing build step for 'gtest'
...
[ 90%] Built target gtest
Scanning dependencies of target testfoo
[ 95%] 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
```


Pekka Röyttä · 6 months ago

Hi,

Thanks for the nice post! This helped a lot to simplify my googlemock setup.

You can simplify the CMakeLists.txt a bit further, as the googlemock includes already gtest. This is true at least for the SVN version.

This can be done by adding the Gtest's include folder location from googlemock into the includes. So after you have fetched gmock's source_dir, add the following line:
include_directories(${source_dir}/gtest/include)
If it doesn't work for you, check out where the gtest/include is under googlemock folder.

You can now take away the part loading gtest, and you won't either need the libgtest target at all. Use the libgmock target in it's place.

Cheers,
Pekka

Emmanouil Matsis ➚ Pekka Röyttä · 6 months ago

Hi,

Great tutorial sir.

I think Pekka is correct. Based on this (https://code.google.com/p/goog... not it is actually recommended.

"Google Mock is implemented on top of the Google Test C++ testing framework (http://code.google.com/p/googl..., and includes the latter as part of the SVN repository and distribution package. You must use the bundled version of Google Test when using Google Mock, or you may get compiler/linker errors."

Also there are 2 libraries that you could consider linking to (similar to GTest).
1. libgmock.a
2. libgmock_main.a

Best,
Manos

myth17 · 4 months ago

Thanks for the articles on cmake. Really helped me more than any documentation online. Thanks again! :)

Joe Gagnon · 5 months ago

David, do you have an alternate way to contact you? The links provided do not work for me. I have

David, do you have an alternate way to contact you? The links provided do not work for me. I have a project with a similar directory layout and I am also using CMake to build the project and am trying to use Google Test to do unit testing.

I tried to adapt the examples you provide in this posting, but am running into a problem where the build fails in the UT code. It cannot find gtest. I'd like to show you the directory layout and what I have in my CMakeLists.txt files (where pertinent). That's too much for this comment venue.

Diego · 8 months ago

Hi David,

I recently checked this post, because I saw it in twitter.

The post is interesting, but I wanted to play a bit, how the solution could be with biicode (disclaimer, I am a founder), and the resulting solution is here in this fork: https://github.com/drodri/gtes...

You can check the diff, that all CMakeLists can be removed, and the project easily configured in the biicode.conf file.

I would love to know your feedback about this solution. In case you didnt know about biicode, it is totally free for open-source, and it will start going open-source itself this month.

Kind regards,

Diego

Hi

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

Follow me:

Related Posts

Better understanding Linux secondary dependencies

solving with examples

08 Jan 2015 by David Corvoysier

A few months ago I stumbled upon a linking problem with secondary dependencies I couldn't solved without [__overlinking__](https://wiki.mageia.org/en/Overlinking_issues_in_packaging) the corresponding libraries. I only realized today in a discussion with my friend [Yann E. Morin](http://ymorin.is-a-geek.org/) that not only did I use the wrong solution for that particular problem, but that my understanding of the gcc linking process was not as good as I had imagined. This blog post is to summarize what I have now understood. There is also a [small repository on github](https://github.com/kaizouman/linux-shlib-link-samples) with the mentioned samples.

(more…)

Categories: linux Tags: gcc ld

A typical Linux project using CMake

03 Nov 2014 by David Corvoysier

When it comes to choosing a make system on Linux, you basically only have two options: autotools or CMake. I have always found Autotools a bit counter-intuitive, but was reluctant to make the effort to switch to CMake because I was worried the learning curve would be too steep for a task you don't have to perform that much often (I mean, you usually spend more time writing code than writing build rules).

A recent project of mine required writing a lot of new Linux packages, and I decided it was a good time to give CMake a try. This article is about how I have used it to build plain old Linux packages almost effortlessly.

(more…)

Categories: linux Tags: CMake

Remote debugging a device running DBus

05 Jun 2014 by David Corvoysier

D-Bus is the application Bus used on Linux desktop, and is sometimes used on other devices running Linux due to the lack of good alternatives (Android is a good example of D-Bus being adopted by chance).

One of the key benefits of using DBus is that you can issue D-Bus requests from the command-line or from very simple python scripts, which is great for testing.

Trouble is: what if you don't have python on the device, or even worse, if you don't have access to the command-line ?

(more…)

Categories: linux Tags: DBus Remote

Introspecting D-Bus from the command-line

01 Jun 2014 by David Corvoysier

One of the cool feature of D-Bus, the Linux desktop application bus, is that it supports introspection.

Even better, you can issue D-Bus introspection requests from the command-line.

(more…)

Categories: linux Tags: DBus

Tag cloud

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

Generated by Jekyll