Next generation unit testing using static reflection

Manu Sanchez @Manu343726

$whoami

- Reflection obsessed nerd
- Self-taught in C++ in general and metaprogramming in particular
- Spanish ISO C++ national body
- Since 2015 working with access control systems and video-IP

Unit testing?

Unit testing

- Black box tests
- No dependencies
- Test one functionality at a time

Real world unit testing

- Having no dependencies is hard
- Software architectures are complex and often poorly designed
- Only the cool kids start projects from scratch

Real world unit testing: Solutions?

Mocks, Fakes, etc: Dependency injection

Real world unit testing: Solutions?

- Classic approach: The class instances initialize themselves

```
struct FileDownloader {
    FileDownloader(const Url& serverUrl) :
        _repo{serverUrl}
    {}

    File download(const Path& directory, const String& filename) {
        const Url url = _repo.lookup(directory, filename);
        return _repo.download_file(url);
    }

private:
    Repository _repo; // Full fledged repo API
};
```

Real world unit testing: Solutions?

- Tests require connection to a server :(

```
void FileDownloaderTest_download() {
    FileDownloader downloader{"https://dummy_testing_server.companylan.net"};

    // Actual file download from real server here:
    auto file = downloader.download("Assets/Images", "Dummy.jpg");

    ASSERT(file.valid());
    REQUIRE(file.filename() == "Dummy.jpg");
}
```

Real world unit testing: Solutions?

- Dependency injection: Classes know nothing about deps initialization

```
struct FileDownloader {
    FileDownloader(IRepository& repo) :
        _repo(repo)
    {}

    File download(const Path& directory, const String& filename);

private:
    IRepository& _repo; // Reference to repo API, either real or fake
};
```

Real world unit testing: Solutions?

- Tests no longer require connection to a server :)

```
struct DummyRepo : public IRepo {
    Url lookup(const Path& directory, const String& filename) overrid final {
        return "";
    }

    File download_file(const Url& url) override final {
        return File{"Dummy.jpg"};
    }
};
```

Real world unit testing: Solutions?

- Tests no longer require connection to a server :)

```
void FileDownloaderTest_download() {
    DummyRepo repo;
    FileDownloader downloader{repo};

    // No file download:
    auto file = downloader.download("Assets/Images", "Dummy.jpg");

    ASSERT(file.valid());
    REQUIRE(file.filename() == "Dummy.jpg");
}
```

Dependency injection: Issues

- RAII reference-based dependency injection does not scale for large architectures.
- DI frameworks (Boost.DI, kangaroo, etc) **are glorified singletons**.
- Global factories make nearly impossible **to track the lifetime of the different modules of the application**.

Dependency injection: Issues

Change class design and architecture in general to make testing easy

Inversion of control: Where it shines

```
struct CommStack {
  Expected<Response> send(const Command& command);

private:
    ByteArray serialize(const Command& command);
    Expected<Response> deserialize(const ByteArray& response);

    int sendBytes(const ByteArray& bytes);
    ByteArray tryReceiveBytes(int length);

    TcpSocket _socket;
};
```

Inversion of control: Where it shines

```
// Generic App-level protocol implementation:

struct CommStack {
   CommStack(IMarshaller& marshaller, IConnection& connection) :
       _marshaller(marshaller),
       _connection(connection)
   {}

   Expected<Response> send(const Command& command);

private:
   IMarshaller& _marshaller;
   IConnection& _connection;
};
```

Inversion of control: Where it shines

Infinite combinations, app protocol doesn't change:

```
struct LetsReinventTheWheelFrameBasedLegacyProtocol : IMarshaller { ... };
struct Protobuf2Protocol : IMarshaller { ... };
struct Protobuf3Protocol : IMarshaller { ... };
struct JsonProtocol : IMarshaller { ... };

struct RS232 : IConnection { ... };
struct Tcp : IConnection { ... };
struct WebSocket : IConnection { ... };

CommStack v0       {LegacyProtocol{},    RS232{}};
CommStack v1       {Protobuf2Protocol{}, RS232{}};
CommStack v1Tcp    {Protobuf2Protocol{}, Tcp{}};
CommStack vCoolKids{JsonProtocol{},      WebSocket{}};
```

What if?

- Design independent from testing, but allow mocking
- Just write tests: Reduce setup to the minumum

Python `unittest` package

Python `unittest` package

- Standard Python package
- `TestCase`s, mocking, etc
- Automagic test discovery

Python `unittest` package

```
class ExampleClass:

    def identity(self, value):
        return value

    def methodThatCallsIdentity(self):
        return self.identity(42)
```

Python `unittest` package

```
import unittest, unittest.mock
import mynamespace

class ExampleTestCase(unittest.TestCase):

    @unittest.mock.patch('mynamespace.ExampleClass.identity', return_value=42)
    def test_identity(self, identity):
        object = mynamespace.ExampleClass()

        self.assertEqual(object.methodThatCallsIdentity(), 42)
        identity.assert_called_once_with(43)
```

`@unittest.mock.patch()`?

- Function decorator
- Substitutes the given class, method, or attribute by a mock during the test

```
@unittest.mock.patch("mymodule.MyClass.method")
def calls_mock(mock):
    obj = mymodule.MyClass()
    obj.method() # Calls mock

def calls_method():
    obj = mymodule.MyClass()
    obj.method() # Calls real method
```

Python `unittest` package

Automatic test discovery

```
$ python3 -m unittest -v
test_identity (test_example.ExampleTestCase) ... FAIL

======================================================================
FAIL: test_identity (test_example.ExampleTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python3.7/unittest/mock.py", line 1209, in patched
    return func(*args, **keywargs)
  File "/home/manu343726/Documents/unittest/examples/python_equivalent/test_example.py", line 11, in test_identity
    identity.assert_called_once_with(43)
  File "/usr/lib/python3.7/unittest/mock.py", line 845, in assert_called_once_with
    return self.assert_called_with(*args, **kwargs)
  File "/usr/lib/python3.7/unittest/mock.py", line 834, in assert_called_with
    raise AssertionError(_error_message()) from causeAssertionError: Expected call: identity(43)
Actual call: identity(42)

----------------------------------------------------------------------
Ran 1 test in 0.001s
```

Forget C++ and switch to Python

You will learn what happiness is

Wait, C++ can also be simple and fun!

Wait, C++ can also be simple and fun!

```
#include <unittest/unittest.hpp>
#include <libexample/example.hpp>
#include <libexample/example.hpp.tinyrefl>

namespace test_example {

struct ExampleTestCase : public unittest::TestCase {

    [[unittest::patch("mynamespace::ExampleClass::identity(int) const")]]
    void test_identity(unittest::MethodSpy<int(int)>& identity) {
        mynamespace::ExampleClass object;

        self.assertEqual(object.methodThatCallsIdentity(), 42);
        identity.assert_called_once_with(43);
    }
};

}
```

Wait, C++ can also be simple and fun!

```
$ make && ./bin/test_example
[100%] Built target test_example
test_identity (test_example::ExampleTestCase) ... FAIL

======================================================
FAIL: test_identity (test_example::ExampleTestCase)
------------------------------------------------------
Stack trace (most recent call last):
#0    Source "unittest/examples/test_example.hpp", line 16, in test_identity
         15:         self.assertEqual(object.methodThatCallsIdentity(), 42);
      >  16:         identity.assert_called_once_with(43);
         17:     }

AssertionError: Expected call: mynamespace::ExampleClass::identity(43)
Actual call: mynamespace::ExampleClass::identity(42)

------------------------------------------------------
Ran 1 tests in  0.001s
```

Wait, C++ can also be simple and fun!

- We can also have non intrusive mocking
- We can also have automatic test discovery
- We can mimic Python syntax, have our own `patch()`

Non intrusive method mocking in C++

Non intrusive method mocking in C++

Different approaches:

- Runtime code monkey patching
- Runtime library jump table patching (elfspy)
- Runtime virtual method table patching (hippomocks)

I'm using elfspy, but any other monkey patching library could do the job.

elfspy

- Implements dynamic library hooking through its `spy::Hook<CRTP, ReturnType, ClassType, Args...>` class template.
- `spy::Hook` is initialized with a pointer to the method we want to hook.
- Provides call introspection methods inherent to mocks such as the list of intercepted calls, the arguments of the different calls, etc.

elfspy

```
// class.hpp

struct Class {
    void method(int i) const;
};
```

elfspy

```
// hook.cpp

struct MethodHook : spy::Hook<Hook, void, Class*, int> {
    MethodHook(const char* name, void(Class::* pointer)(int)) :
        spy::Hook{name, spy::Method<void, Class*, int>{pointer}.resolve()}
    {}
};
```

elfspy

```
int main() {
    Class obj; {
        MethodHook hook{"method", &Class::method};

        // Call is registered in hook, calls Class::method(1)
        obj.method(1);

        // Use the hook as a mock object to check calls:

        // Number of calls is 1:
        assert(spy::call(hook).count() == 1);
        // First argument of first call is 1:
        assert(spy::arg<0>(hook).value(0) == 1);
    }

    // Call is not registered, the hook was uninstalled
    obj.method(2);
}
```

Automatic test discovery

- Let's fix a search criteria first:

> A test is any method of a test case class which name is prefixed by `test_`.
> A test case class is any class inheriting from `TestCase`.

Recipe to find a test function

1. Find any test case class, that is, any class that inherits from `TestCase`.
2. For each test case class found, find any member function named `"test_XXXXXX"`.
3. For each test method found, instantiate its test case class, invoke the method, and report any execution failures.

How? Static reflection to the rescue

tinyrefl

- `constexpr` object oriented reflection API
- External parsing and metadata generation tool
- Built-in cmake integration

tinyrefl

```
find_package(tinyrefl_tool REQUIRED)

add_library(mylib mylib.cpp)

# Parses mylib.hpp before mylib compilation
# Generates mylib.hpp.tinyrefl file with reflection data
tinyrefl_tool(TARGET mylib HEADERS mylib.hpp)
```

tinyrefl: The generated code

- Independent from the API
- Users can write their own API and metadata structures
- Both generated code and APIs check for matching codegen version for compatibility
- Metadata represented as preprocessor macros

```
#ifndef TINYREFL_GENERATED_FILE_10403593754518508410_INCLUDED
#define TINYREFL_GENERATED_FILE_10403593754518508410_INCLUDED

// VERSION CHECKS HERE

TINYREFL_REGISTER_FILE(TINYREFL_FILE((TINYREFL_STRING(test_example.hpp)); ...))
TINYREFL_REGISTER_NAMESPACE(TINYREFL_NAMESPACE((TINYREFL_STRING(test_example)); ...))
TINYREFL_REGISTER_CLASS(TINYREFL_CLASS((TINYREFL_STRING(ExampleTestCase)), ...))
TINYREFL_REGISTER_MEMBER_FUNCTION(TINYREFL_MEMBER_FUNCTION((TINYREFL_STRING(test_identity)), ...))

#endif // TINYREFL_GENERATED_FILE_10403593754518508410_INCLUDED
```

tinyrefl: The generated code

- Metadata includes full file information, namespaces, classes, free functions, enums, public members, constructors, static members, etc.
- Metadata also includes **attributes applied to each entity**.
- Multiple metadata files can be included in one translation unit, **namespace contents are automatically merged together**.
- There's also a list of all the reflected entities, propagated automatically as more metadata files are included.

tinyrefl: The API

- A reflection API is implemented by giving meaning to each metadata `TINYREFL_XXXX` macros.
- Just include your API `#define`s before the generated code files.
- The library comes with a full working `constexpr` API.

```
#define TINYREFL_CLASS(name, fullname, ...) Class{name, fullname, ...}
#define TINYREFL_REGISTER_CLASS(class) registerClass(class);

#include <mylib.hpp>
#include <mylib.hpp.tinyrefl>
```

tinyrefl: The bult-in API

- `constexpr` object oriented API
- Each C++ entity (class, namespace, member function, etc) is represented as an object of type `class_`, `namespace_`, `member_function`, etc.
- Metadata objects have methods to get information of the reflected entity
- `tinyrefl::metadata<>` returns the reflection metadata class of the given entity

```
constexpr auto class_metadata = tinyrefl::metadata<Class>();

static_assert(class_metadata.full_name() == "mylib::Class");
```

tinyrefl: Walking through the metadata

To simplify introspection, the API provides a visitor API with different filters:

```
tinyrefl::visit("mylib.hpp"_id, tinyrefl::class_visitor(
    [](const auto& class_) constexpr {
        cout << "Found class \"" << class_.name() << "\" with "
             << class_.children().size() << " members\n";
    }
));
```

tinyrefl: Walking through the metadata

Visitors are `constexpr` too:

```
static_assert([] constexpr {
  bool found = false;

  tinyrefl::visit<Class>(tinyrefl::static_member_function_visitor(
    [&found](const auto& function) constexpr {
       if(function.name() == "factory") {
          found = true;
       }
    }
  ));

  return found;
}(),
"I hate factory functions");
```

Back to our recipe...

1. Find any test case class:

```
void runTestCases()
{
    // Visit all reflected classes:
    TINYREFL_VISIT_ENTITIES(tinyrefl::class_visitor(
      [](const auto& class_) {

        // Get the class type:
        using Class = typename decltype(class_)::class_type;

        if constexpr(std::is_base_of_v<unittest::TestCase, Class>) {
         runTestCase(class_);
        }
      }
    ))
}
```

2. For each test case class found, find any test case method

```
template<typename ClassMetadata>
void runTestCase(const ClassMetadata& class_) {
    using TestCase = typename ClassMetadata::class_type;

    tinyrefl::visit(class_, tinyrefl::member_function_visitor(
      [](const auto& method) {
          // Trick: method is not constexpr in this context
          constexpr decltype(method) constexpr_method;

          if constexpr(isTestMethod(constexpr_method)) {
             // Instance test case and run the test:
             TestCase testCase;
             method(testCase);
          }
      }
    ))
}
```

3. `isTestCaseMethod()` ?

```
template<typename MethodMetadata>
constexpr bool isTestMethod(const MethodMetadata& method) {
  return beginsWith(method.name(), "_test");
}
```

Where is my main?

`tinyrefl` parser works per file, so we need something that includes every test header for us in one translation unit and calls `runTestCases()`.

Use ugly CMake:

```
add_unittest(TARGET mylib TESTS
   identity_test.hpp)
```

Where is my main?

`add_unittest()` calls `tinyrefl_tool()` and adds an executable with a generated `main.cpp` file like this:

```
#include <tinyrefl/api.hpp>
#include <test1.hpp>
#include <test1.hpp.tinyrefl>
#include <test2.hpp>
#include <test2.hpp.tinyrefl>
...
#include <testN.hpp>
#include <testN.hpp.tinyrefl>

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

Okay, cool, but what happenned to `patch()`?

Okay, cool, but what happenned to `patch()`?

Just read the `[[attributes]]` of the test method!

```
if constexpr(constexpr_method.has_attribute("tinyrefl::patch")) {
  constexpr auto patch = constexpr_method.attribute("unittest::patch");
  static_assert(patch.arguments().size() == 1);
  constexpr auto method_id = tinyrefl::hash_constant<
      patch.arguments()[0].hash()
  >;

  static_assert(tinyrefl::has_metadata(method_id), "Patch target not found";
  ...
} else {
  // Run test method without patch, as usual
  TestCase testCase;
  method(testCase);
}
```

Okay, cool, but what happenned to `patch()`?

```
constexpr auto patch_method = tinyrefl::metadata(method_id);

// Create hook with the right signature:
auto hook = makeHook(patch_method.full_name(), patch_method.pointer());

// Run test:
TestCase testCase;
method(testCase, hook);
```

Future

Future: Using the Reflection TS

Future: Using the Reflection TS

- No need for an external parser
- Reflection information available from any point in our source
- **Problem**: No global list of reflected entities
- **Problem**: `[[attributes]]` are not reflected
- **Problem**: No project global metadata, CMake generated `main.cpp` cannot be avoided.

Future: Using the Reflection TS

- Use a specific namespace for tests?

> Nope, namespaces are not registered as `Record`s in the TS,
> so we have no access to traverse their members.

- Use test return type for patches?

```
struct MyTestCase : TestCase {
  Patch<"mylib::Class::method"_id>
  test_something() {
    ...
  }
};
```

Future: Using the Reflection TS

There's no way to implement `unittest` with the current Reflectio TS.

Yeah, I want to cry too.

> Note the target use case of the reflection TS is **reflection of individual entities**, not global library introspection

Future: More ideas

- Different test discovery criteria. Why test cases should be classes?:

```
[[unittest::TestCase]]
namespace MyTestCase {
  void test_something() {
    ...
  }
}
```

- Patching of private member functions. `tinyrefl` does not currently reflect private members.
- Multiple patches in the same test

Thank you