Testing Compile-time Constructs Within a Runtime Unit Testing Framework

IGOR BOGOSSLAVSKYI
We use C++ (14) for safety-critical applications that we deliver to our customers.

Errors have a high cost, so rigorous testing is a must.
We use increasingly more compile-time polymorphism and checks.

Question: how to test if something is compilable in a rigorous way?
Imagine designing a user-facing API

```c
float get_half_of(float smth) {
    return smth / 2.0F;
}
```
Maybe a bit more generic

template<
    class T,
    class = std::enable_if_t<std::is_floating_point_v<T>>>
T get_half_of(T smth) {
    return smth / T{2};
}
Write a rigorous testing suite

#include <gtest/gtest.h>

TEST(TestMyApi, Halving) {
    EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
    EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
}
#include <gtest/gtest.h>

TEST(TestMyApi, Halving) {
    EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
    EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
    // This should NOT compile:
    // get_half_of(23);
}
Now throw it into the rest of the code base

```cpp
} ————————————————————————————————————
template<
  class T,
  class = std::enable_if_t<std::is_floating_point_v<T>>>
T get_half_of(T smth) {
  return smth / T{2};
}
} ————————————————————————————————————

TEST(TestMyApi, Halving) {
  EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
  EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
  // This should NOT compile:
  // get_half_of(23);
} ————————————————————————————————————
```
And deliver it to the customers

```cpp
#include <iostream>

template<
    class T,
    class = std::enable_if_t<std::is_floating_point_v<T>>
>
T get_half_of(T smth) {
    return smth / T{2};
}

TEST(TestMyApi, Halving) {
    EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
    EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
    // This should NOT compile:
    // get_half_of(23);
}
```
Correct requests get correct responses

```
float a = get_half_of(23.0F);
```

```
template<
    class T,
    class = std::enable_if_t<std::is_floating_point_v<T>>
>
T get_half_of(T smth) {
    return smth / T{2};
}
```

```
TEST(TestMyApi, Halving) {
    EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
    EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
    // This should NOT compile:
    // get_half_of(23);
}
```
Wrong requests cause a compilation error

float a = get_half_of(23);

Compilation error

std::enable_if_t<std::is_floating_point_v<T>>
Code has a tendency to evolve with time

template<
    class T,
    class = std::enable_if_t<std::is_floating_point_v<T>>>
T get_half_of(T smth) {
    return smth / T{2};
}
template<
class T,
class = std::enable_if_t<std::is_floating_point_v<T>>>
T get_half_of(T smth) {
    return smth / T{2};
}
Code has a tendency to evolve with time

template<
    class T,
    class = std::enable_if_t<std::is_floating_point_v<T>> T
T get_half_of(T smth) {
    return smth / T{2};
}
Code has a tendency to evolve with time

```cpp
template<class T>
T get_half_of(T smth) {
    return smth / T{2};
}
```

Much better! We anyway use it only with float numbers! I’ll go sip a coffee!
The code passes all checks and lands to the customer instance

```cpp
template<class T>
T get_half_of(T smth) {
    return smth / T{2};
}
```

commit & push

CI passes

```text
TEST(TestMyApi, Halving) {
    EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
    EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
    // This should NOT compile:
    // get_half_of(23);
}
```
After some time customers use the API

```cpp
template<class T>
T get_half_of(T smth) {
    return smth / T(2);
}
```

```cpp
TEST(TestMyApi, Halving) {
    EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
    EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
    // This should NOT compile:
    // get_half_of(23);
}
```

```cpp
float a = get_half_of(23);
```
And receive a wrong response

```cpp
template< typename T>
T get_half_of(T smth) {
  return smth / T{2};
}

TEST(TestMyApi, Halving) {
  EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
  EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
  // This would NOT compile:
  // get_half_of(23);
}

float a = get_half_of(23);
return smth / T{2};
```
And they spend two weeks debugging the failure

```cpp
template<class T>
T get_half_of(T smth) {
  return smth / T(2);
}

TEST(TestMyApi, Halving) {
  EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
  EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
  // This should NOT compile:
  // get_half_of(23);
}

float a = get_half_of(23);
```

11.0F
What went wrong? What can we do about it?

- We were relying on the code not being able to compile
- We missed the change at which the unwanted code started compiling
- Why did our test suite not help us?

```cpp
#include <gtest/gtest.h>

TEST(TestMyLib, Halving) {
  EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
  EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
  // This should NOT compile:
  // get_half_of(23);
}
```
What went wrong? What can we do about it?

- We were relying on the code not being able to compile
- We missed the change at which the unwanted code started compiling
- Why did our test suite not help us?

```cpp
#include <gtest/gtest.h>

TEST(TestMyLib, Halving) {
  EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
  EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
  // This should NOT compile:
  // get_half_of(23);
}
```

Would be really cool to enforce and test this!
We can do better!

```cpp
#include <gtest/gtest.h>

TEST(TestMyLib, Halving) {
    EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
    EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
    // This should NOT compile:
    // get_half_of(23);
}
```
STATIC_TEST and SHOULD_NOT_COMPILE to the rescue!

#include <gtest/gtest.h>
#include <static_test/static_test.h>

TEST(TestMyLib, Halving) {
    EXPECT_FLOAT_EQ(21.0F, get_half_of(42.0F));
    EXPECT_DOUBLE_EQ(21.0, get_half_of(42.0));
}

STATIC_TEST(Halving) {
    SHOULD_NOT_COMPILE(get_half_of(23));
}
### STATIC_TEST output when our bug is present

[----------] 1 test from StaticTest__Halving
[ RUN       ] StaticTest__SomeTest.Halving
[ COMPILE STATIC TEST ] Halving
ERROR: my_api/test_halving.cpp:35: must fail to compile.
my_api/test_halving.cpp:0: Failure
Some of the static tests failed. See above for error.
[ FAILED    ] Halving
[ FAILED    ] StaticTest__Halving.Halving (1403 ms)
[----------] 1 test from StaticTest__SomeTest (1403 ms)
STATIC_TEST output when our bug is present

[----------] 1 test from StaticTest__Halving
[ RUN      ] StaticTest__SomeTest.Halving
[ COMPILE STATIC TEST ] Halving
ERROR: my_api/test_halving.cpp:35: must fail to compile.
my_api/test_halving.cpp:0: Failure
Some of the static tests failed. See above for error.
[       FAILED ] Halving
[ FAILED     ] StaticTest__Halving.Halving (1403 ms)
[----------] 1 test from StaticTest__SomeTest (1403 ms)
Once we fix the bug this is what we expect to see

[----------] 1 test from StaticTest__Halving
[ RUN      ] StaticTest__Halving.Halving
[ COMPILE STATIC TEST ] Halving
[           OK    ] Halving
[           OK    ] StaticTest__Halving.Halving (966 ms)
[----------] 1 test from StaticTest__Halving (966 ms total)
Standard testing pipeline

test_halving.cpp

Compiler

Test binary

GoogleTest

flags
Proposed testing pipeline

test_halving.cpp

test_compile_1.cpp

test_compile_2.cpp

Compiler

flags

Test binary

GoogleTest

StaticTest Fixture

flags
Things our library does

test_halving.cpp

Test binary

Compiler

flags

GoogleTest

StaticTest Fixture

test_compile_1.cpp

test_compile_2.cpp
Things our library does

1. Generate “compilation” tests

Test binary

Compiler

flags

GoogleTest

StaticTest Fixture

test_halving.cpp

test_compile_1.cpp

test_compile_2.cpp
Things our library does

2. Feed the generated tests to the compiler with the correct flags
Things our library does

3. Interpret the compilation outcome

Compiler

flags

Test binary

GoogleTest

StaticTest Fixture

test_halving.cpp

test_compile_1.cpp

test_compile_2.cpp
Integrate with the existing flow

- Build upon GoogleTest library by using a custom fixture
- Introduce the following macros:
  - `STATIC_TEST(foo) {}`
  - `SHOULD_NOT_COMPILE(code);`
  - `SHOULD_NOT_COMPILE_WITH_MESSAGE(code, message);`
- These macros hide calls to custom scripts that do all the work
- These scripts generate custom test files, feed them to the compiler and interpret the compilation results
#define STATIC_TEST(NAME)  
  class StaticTest__##NAME : public StaticTest {
public:
  StaticTest__##NAME() noexcept : StaticTest{#NAME, __FILE__} {}  
};  
/* Create a test that uses this fixture and wraps the user code.*/
TEST_F(StaticTest__##NAME, NAME)

#define SHOULD_NOT_COMPILE(IGNORED_CODE)

#define SHOULD_NOT_COMPILE_WITH_MESSAGE(IGNORED_CODE, IGNORED_MESSAGE)
#define STATIC_TEST(NAME)

class StaticTest__##NAME : public StaticTest {
public:
    StaticTest__##NAME() noexcept : StaticTest(#NAME, __FILE__) {} 
};

/* Create a test that uses this fixture and wraps the user code.*/
TEST_F(StaticTest__##NAME, NAME)

#define SHOULD_NOT_COMPILE(IGNORED_CODE)
#define SHOULD_NOT_COMPILE_WITH_MESSAGE(IGNORED_CODE, IGNORED_MESSAGE)

Our custom GoogleTest fixture that hides the code to generate, compile, run the compilability tests and interpret the compilation results.
#define STATIC_TEST(NAME)
    class StaticTest__##NAME : public StaticTest {
    public:
        StaticTest__##NAME() noexcept : StaticTest{#NAME, __FILE__} {} 
    };
    /* Create a test that uses this fixture and wraps the user code.*/
    TEST_F(StaticTest__##NAME, NAME)

#define SHOULD_NOT_COMPILE(IGNORED_CODE)
#define SHOULD_NOT_COMPILE_WITH_MESSAGE(IGNORED_CODE, IGNORED_MESSAGE)
#define STATIC_TEST(NAME)                                                 
    class StaticTest__##NAME : public StaticTest {                          
    public:                                                                 
        StaticTest__##NAME() noexcept : StaticTest{#NAME, __FILE__} {}        
    };                                                                      

/* Create a test that uses this fixture and wraps the user code.*/      
TEST_F(StaticTest__##NAME, NAME)

#define SHOULD_NOT_COMPILE(IGNORED_CODE)                                      

#define SHOULD_NOT_COMPILE_WITH_MESSAGE(IGNORED_CODE, IGNORED_MESSAGE)
Our custom StaticTest fixture

class StaticTest : public ::testing::Test {
    public:
        StaticTest(const string &name, const string &file) {
            const auto cmd = "command " + name + " " + file;
            const auto exit_status = std::system(cmd.c_str());
            if (exit_status != 0) {
                GTEST_MESSAGE_AT_(file_.c_str(), 0,
                    "Some of the static tests failed. See above.",
                    ::testing::TestPartResult::kNonFatalFailure);
            }
        }
    }
};
Our custom StaticTest fixture

```cpp
class StaticTest : public ::testing::Test {
public:
    StaticTest(const string &name, const string &file) {
        const auto cmd = "command " + name + " " + file;
        const auto exit_status = std::system(cmd.c_str());
        if (exit_status != 0) {
            GTEST_MESSAGE_AT_(file_.c_str(), 0,
                "Some of the static tests failed. See above.
                ::testing::TestPartResult::kNonFatalFailure);
        }
    }
};
```

Construct a command string to call the external script
class StaticTest : public ::testing::Test {
public:
    StaticTest(const string &name, const string &file) {
        const auto cmd = "command " + name + " " + file;
        const auto exit_status = std::system(cmd.c_str());
        if (exit_status != 0) {
            GTEST_MESSAGE_AT_(file_.c_str(), 0,
                "Some of the static tests failed. See above.",
                ::testing::TestPartResult::kNonFatalFailure);
        }
    }
};
class StaticTest : public ::testing::Test {
public:
    StaticTest(const string &name, const string &file) {
        const auto cmd = "command " + name + " " + file;
        const auto exit_status = std::system(cmd.c_str());
        if (exit_status != 0) {
            GTEST_MESSAGE_AT_(file_.c_str(), 0,
                "Some of the static tests failed. See above.",
                ::testing::TestPartResult::kNonFatalFailure);
        }
    }
};

If the command failed - insert a GoogleTest error that will be shown to the user
Let's talk about that external script

- We use an external script that gets called from the fixture
- This script does the following:
  a. parses the original test file as text
  b. generates new test files from STATIC_TESTS in the original test file
  c. gets proper flags for the original test file
  d. feeds the generated test files to the compiler
  e. interprets the results and returns an error code to the fixture
- The script can be written in any language as long as it can be called from within our StaticTest fixture
Let’s look at a concrete example

```cpp
#include <gtest/gtest.h>
#include "foo/foo.h"

STATIC_TEST(Halving) {
    SHOULD_NOT_COMPILE(get_half_of(23));
    SHOULD_NOT_COMPILE_WITH_MESSAGE(
        get_half_of("23"),
        "no matching function for call");  
}
### STATIC_TEST

```cpp
#include <gtest/gtest.h>
#include "foo/foo.h"

STATIC_TEST(Halving) {
  SHOULD_NOT_COMPILE(get_half_of(23));
  SHOULD_NOT_COMPILE_WITH_MESSAGE(get_half_of("23"),
                                   "no matching function for call");
}
```
Each STATIC_TEST generates a main function

```cpp
#include <gtest/gtest.h>
#include "foo/foo.h"

STATIC_TEST(Halving) {
    SHOULD_NOT_COMPILE(get_half_of(23));
    SHOULD_NOT_COMPILE_WITH_MESSAGE(
        get_half_of("23"),
        "no matching function for call");
}

#include <gtest/gtest.h>
#include "foo/foo.h"

int main() {

}
```
Each SHOULD_NOT_COMPILE is copied to a new file

```c
#include <gtest/gtest.h>
#include "foo/foo.h"

STATIC_TEST(Halving) {
    SHOULD_NOT_COMPILE(get_half_of(23));
    SHOULD_NOT_COMPILE_WITH_MESSAGE(
        get_half_of("23"),
        "no matching function for call");
}
```

```c
#include <gtest/gtest.h>
#include "foo/foo.h"

int main() {
    get_half_of(23);
    return 0;
}
```
Feed the generated files to the compiler

Our library compiles the generated files and parses the compilation result.

```cpp
#include <gtest/gtest.h>
#include "foo/foo.h"
int main() {
    get_half_of(23);
    return 0;
}
```

```cpp
#include <gtest/gtest.h>
#include "foo/foo.h"
int main() {
    get_half_of("23");
    return 0;
}
```
How to get proper compilation flags?

- Compilation highly depends on the compilation flags provided by the user
- We derive the compilation flags for the generated files from the original test file in which the STATIC_TESTs are written
- We will compile all the generated files with the same set of flags
- There are 2 broad ways of how to get flags:
  - By using strings stored in the original test binary compiled with `-frecord-gcc-switches`
  - By parsing a compilation database that can be generated on the fly or beforehand
Interpreting the compilation results

- The script passes files to the compiler and receives the compilation output
- For SHOULD_NOT_COMPILE:
  - If compilation is successful - **fail** the STATIC_TEST and print the line of code in the original test file to notify the user
  - If compilation fails - print the STATIC_TEST **success** message
- For SHOULD_NOT_COMPILE_WITH_MESSAGE:
  - If compilation is successful - **fail** the STATIC_TEST and print the line of code in the original test file to notify the user
  - If compilation fails check that the output matches a specified pattern:
    - If pattern matches - the STATIC_TEST is **successful**
    - If not - the static test has **failed**
Summary

- We can test the compilability of the code
- We must use the compiler in the loop
- We integrate with GoogleTest library to run our tooling
- We generate custom tests that compile generated code
  - If this compilation succeeds the static test fails
  - If this compilation fails the static test succeeds
- Most of the functionality lives in an external script called by a fixture
- This allows catching bugs that were impossible to catch before
- The final product is more safe to use
- A proof of concept implementation is incoming
Questions?