Advanced testing with GoogleTest & GoogleMock

C++ User Group Aachen

Disclaimer

- This talk focuses on GoogleTest & GoogleMock
- Use the testing/mocking framework you like most
- Most concepts are transferrable

Previously …

- GoogleTest assertion macros
- Text fixtures (setup & teardown)
- Mocks
  - Nice, strict and naggy
- Expectations
- Visual Studio Code integration

The Beyonce Rule

(Software Engineering at Google)

Today’s plan

- Writing DRY tests
- Integrating custom classes & assertions
- Testing exceptions
- Repeating & filtering tests
- Death tests
- Advanced mocking
- Best practices

Writing DRY tests

```
// testees
std::string timeToString(time_t t, const char* zone);
time_t      stringToTime(const std::string& s);

static void convertToFromString(const char* timeZone)
{
    time_t input = time(nullptr);
    std::string timeStr = timeToString(input, timeZone);
    time_t parsedTime = stringToTime(timeStr);
    ASSERT_EQ(input, parsedTime);
}

TEST(TimeTests, convertToFromString)
{
    convertToFromString("UTC");
    convertToFromString("Europe/Berlin");
    convertToFromString("Asia/Pacific");
    convertToFromString("AWCST"); // UTC+8:45
}
```

Which test failed?

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TimeTests
[ RUN      ] TimeTests.convertToFromString
dry.cpp:12: Failure
Expected equality of these values:
  input
    Which is: 1748546913
  parsedTime
    Which is: 1748546790

[  FAILED  ] TimeTests.convertToFromString (0 ms)
```

Writing DRY tests: custom messages

```
static void convertToFromString(const char* timeZone)
{
    time_t input = time(nullptr);
    std::string timeStr = timeToString(input, timeZone);
    time_t parsedTime = stringToTime(timeStr);
    ASSERT_EQ(input, parsedTime) << "Timezone: " << timeZone;
}

TEST(TimeTests, convertToFromString)
{
    convertToFromString("UTC");
    convertToFromString("Europe/Berlin");
    convertToFromString("Asia/Pacific");
    convertToFromString("AWCST"); // UTC+8:45
}
```

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TimeTests
[ RUN      ] TimeTests.convertToFromString
dry.cpp:12: Failure
Expected equality of these values:
  input
    Which is: 1748546913
  parsedTime
    Which is: 1748546790
Timezone: AWCST

[  FAILED  ] TimeTests.convertToFromString (0 ms)
```

Writing DRY tests: Scopes

```
static void convertToFromString(const char* timeZone)
{
    SCOPED_TRACE(timeZone);

    time_t input = time(nullptr);
    std::string timeStr = timeToString(input, timeZone);
    EXPECT_GE(timeStr.length(), 8);

    time_t parsedTime = stringToTime(timeStr);
    ASSERT_EQ(input, parsedTime);
}

TEST(TimeTests, convertToFromString)
{
    convertToFromString("UTC");
    convertToFromString("Europe/Berlin");
    convertToFromString("Asia/Pacific");
    convertToFromString("AWCST"); // UTC +8:45
}
```

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TimeTests
[ RUN      ] TimeTests.convertToFromString
dry.cpp:16: Failure
Expected equality of these values:
  input
    Which is: 1748547897
  parsedTime
    Which is: 1748547774
Google Test trace:
dry.cpp:9: AWCST

[  FAILED  ] TimeTests.convertToFromString (0 ms)
```

Can we do better?

Writing DRY tests: Parametrized tests

```
class TimeTests : public testing::TestWithParam<const char*>
{
};

TEST_P(TimeTests, convertToFromString)
{
    const char* timeZone = GetParam();
    time_t input = time(nullptr);
    std::string timeStr = timeToString(input, timeZone);
    EXPECT_GE(timeStr.length(), 8);

    time_t parsedTime = stringToTime(timeStr);
    ASSERT_EQ(input, parsedTime);
}

INSTANTIATE_TEST_SUITE_P(TimeTestsInst, TimeTests,
                         testing::Values("UTC", "Europe/Berlin",
                                         "Asia/Pacific", "AWCST"));
```

Test fixture with parameter

Instantiation with parameters

Writing DRY tests: Parametrized tests

```
[==========] Running 4 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 4 tests from TimeTestsInst/TimeTests
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/0
[       OK ] TimeTestsInst/TimeTests.convertToFromString/0 (0 ms)
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/1
[       OK ] TimeTestsInst/TimeTests.convertToFromString/1 (0 ms)
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/2
[       OK ] TimeTestsInst/TimeTests.convertToFromString/2 (0 ms)
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/3
/Users/daniel/git/presentations/GoogleTest_GoogleMock/dry_parameterized.cpp:20: Failure
Expected equality of these values:
  input
    Which is: 1748681000
  parsedTime
    Which is: 1748680877

[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/3, where GetParam() = "AWCST" (0 ms)
[----------] 4 tests from TimeTestsInst/TimeTests (0 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 3 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/3, where GetParam() = "AWCST"
```

Writing DRY tests: Parametrized tests (with nice names)

```
// ...

static std::string tznameToString(const testing::TestParamInfo<const char*>& param) {
    std::string name = param.param;
    // "test names must be non-empty, unique, and may only contain ASCII alphanumeric
    //  characters. In particular, they should not contain underscores"
    std::erase_if(name, [] (char c) { return !std::isalnum(c); });
    return name;
}

INSTANTIATE_TEST_SUITE_P(TimeTestsInst, TimeTests,
                         testing::Values("UTC", "Europe/Berlin",
                                         "Asia/Pacific", "AWCST"),
                         tznameToString);
```

Writing DRY tests: Parametrized tests

```
[==========] Running 4 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 4 tests from TimeTestsInst/TimeTests
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/UTC
[       OK ] TimeTestsInst/TimeTests.convertToFromString/UTC (0 ms)
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/EuropeBerlin
[       OK ] TimeTestsInst/TimeTests.convertToFromString/EuropeBerlin (0 ms)
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/AsiaPacific
[       OK ] TimeTestsInst/TimeTests.convertToFromString/AsiaPacific (0 ms)
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/AWCST
/Users/daniel/git/presentations/GoogleTest_GoogleMock/dry_parameterized.cpp:20: Failure
Expected equality of these values:
  input
    Which is: 1748681739
  parsedTime
    Which is: 1748681616

[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/AWCST, where GetParam() = "AWCST" (0 ms)
[----------] 4 tests from TimeTestsInst/TimeTests (0 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 3 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/AWCST, where GetParam() = "AWCST"
```

Writing DRY tests: templates

- You can write type-parameterized tests, too!
- Look for `TYPED_TEST*` in the documentation

Custom classes & assertions

```
template<class Value>
class Result {
public:
    Result(Value v) : val(std::move(v)) {}
    Result(std::error_code e) : val(e) {}

    bool has_value() const noexcept { return val.index() == 0; }
    bool has_error() const noexcept { return val.index() == 1; }

    const Value& value() const {
        if (!has_value()) throw std::system_error(error());
        return std::get<Value>(val);
    }
    std::error_code error() const {
        if (!has_error()) throw std::logic_error("no error");
        return std::get<std::error_code>(val);
    }

    bool operator==(const Value& v) const noexcept {
        return has_value() && value() == v;
    }
private:
    std::variant<Value, std::error_code> val;
};
```

Custom classes & assertions

```
TEST_P(TimeTests, convertToFromString)
{
    const char* timeZone = GetParam();
    time_t input = time(nullptr);
    std::string timeStr = timeToString(input, timeZone);
    EXPECT_GE(timeStr.length(), 8);

    Result<time_t> parsedTime = stringToTime(timeStr);
    ASSERT_EQ(input, parsedTime);
}
```

```
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/AWCST
/Users/daniel/git/presentations/GoogleTest_GoogleMock/custom.cpp:46: Failure
Expected equality of these values:
  input
    Which is: 1748684526
  parsedTime
    Which is: 24-byte object <16-00 00-00 01-00 00-00 30-01 E1-F9 01-00 00-00 01-00 00-00 00-60 00-00>

[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/AWCST, where GetParam() = "AWCST" (0 ms)
```

Custom classes & assertions

- Printing your custom classes
  - Simplest solution: implement `operator<<`
- If you want different output for your tests (e.g. with more debug info), implement one of these (as friend or in same namespace):
  - `template<typename Sink> void AbslStringify(Sink&, const MyClass&)`
  - `void PrintTo(const MyClass&, std::ostream* os)`

Custom classes & assertions

```
template<class Value>
static void PrintTo(const Result<Value>& val, std::ostream* os) {
    using testing::internal::PrintTo;
    if (val.has_value()) {
        PrintTo(val.value(), os);
    } else {
        PrintTo(val.error().message(), os);
    }
}
```

```
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/AWCST
/Users/daniel/git/presentations/GoogleTest_GoogleMock/custom.cpp:56: Failure
Expected equality of these values:
  input
    Which is: 1748687255
  parsedTime
    Which is: "Invalid argument"

[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/AWCST, where GetParam() = "AWCST" (0 ms)
```

Custom classes & assertions

- Limitations of this approach:
  - Intend is unclear (what does „A != B“ mean?)
  - Ambiguity in output (e.g. for `Result<std::string>`)
- You want to have checks **matching your domain**
  - Write custom predicate or macro

Predicates & assertions

- `EXPECT_PRED1(pred,val1)`
  `EXPECT_PRED2(pred,val1,val2)`
  `EXPECT_PRED3(pred,val1,val2,val3)`
  `EXPECT_PRED4(pred,val1,val2,val3,val4)`
  `EXPECT_PRED5(pred,val1,val2,val3,val4,val5)`
- `ASSERT_PRED1(pred,val1)`
  `ASSERT_PRED2(pred,val1,val2)`
  `ASSERT_PRED3(pred,val1,val2,val3)`
  `ASSERT_PRED4(pred,val1,val2,val3,val4)`
  `ASSERT_PRED5(pred,val1,val2,val3,val4,val5)`

Predicates & assertions

```
// Returns true if m and n have no common divisors except 1.
bool MutuallyPrime(int m, int n) { ... }
...
const int a = 3;
const int b = 4;
const int c = 10;
...
EXPECT_PRED2(MutuallyPrime, a, b);  // Succeeds
EXPECT_PRED2(MutuallyPrime, b, c);  // Fails
```

Predicates & assertions

```
template <typename T>
static bool ResultValueEquals(const Result<T>& res, T value)
{
    if (res.has_error())
    {
        return false;
    }

    return res.value() == value;
}

TEST_P(TimeTests, convertToFromString)
{
    const char* timeZone = GetParam();
    time_t input = time(nullptr);
    std::string timeStr = timeToString(input, timeZone);
    EXPECT_GE(timeStr.length(), 8);

    Result<time_t> parsedTime = stringToTime(timeStr);
    ASSERT_PRED2(ResultValueEquals<time_t>, parsedTime, input);
}
```

Predicates & assertions

```
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/AsiaPacific
/Users/daniel/git/presentations/GoogleTest_GoogleMock/custom.cpp:76: Failure
ResultValueEquals<time_t>(parsedTime, input) evaluates to false, where
parsedTime evaluates to 1748694591
input evaluates to 1748694590

[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/AsiaPacific, where GetParam() = "Asia/Pacific" (0 ms)
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/AWCST
/Users/daniel/git/presentations/GoogleTest_GoogleMock/custom.cpp:76: Failure
ResultValueEquals<time_t>(parsedTime, input) evaluates to false, where
parsedTime evaluates to "Invalid argument"
input evaluates to 1748694590

[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/AWCST, where GetParam() = "AWCST" (0 ms)
```

IMO: This got worse …

Custom predicate function

```
template<typename T>
::testing::AssertionResult ResultValueEquals(const Result<T>& res, T value)
{
    if (res.has_error()) {
        return ::testing::AssertionFailure() << "result contains an error: " << res.error();
    }

    const auto& val = res.value();
    if (val == value) {
        return ::testing::AssertionSuccess();
    } else {
        return ::testing::AssertionFailure() << val << " not equal to " << value;
    }
}

TEST_P(TimeTests, convertToFromString)
{
    const char* timeZone = GetParam();
    time_t input = time(nullptr);
    std::string timeStr = timeToString(input, timeZone);
    EXPECT_GE(timeStr.length(), 8);

    Result<time_t> parsedTime = stringToTime(timeStr);
    ASSERT_TRUE(ResultValueEquals(parsedTime, input));
}
```

Custom predicate function

```
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/AsiaPacific
/Users/daniel/git/presentations/GoogleTest_GoogleMock/custom.cpp:70: Failure
Value of: ResultValueEquals(parsedTime, input)
  Actual: false (1748687972 not equal to 1748687971)
Expected: true

[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/AsiaPacific, where GetParam() = "Asia/Pacific" (0 ms)
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/AWCST
/Users/daniel/git/presentations/GoogleTest_GoogleMock/custom.cpp:70: Failure
Value of: ResultValueEquals(parsedTime, input)
  Actual: false (result contains an error: generic:22)
Expected: true

[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/AWCST, where GetParam() = "AWCST" (0 ms)
```

Somewhat better

Custom assertion

```
#define ASSERT_RESULT_EQUALS(res, val)                                          \
    if (res.has_error())                                                        \
    {                                                                           \
        auto msg = fmt::format("expected '{}' to contain a value, but it contains an error: {}",   \
                               #res, res.error().message());                    \
        GTEST_FATAL_FAILURE_(msg.c_str());                                      \
        return;                                                                 \
    }                                                                           \
    ASSERT_EQ(res.value(), val)


TEST_P(TimeTests, convertToFromString)
{
    const char* timeZone = GetParam();
    time_t input = time(nullptr);
    std::string timeStr = timeToString(input, timeZone);
    EXPECT_GE(timeStr.length(), 8);

    Result<time_t> parsedTime = stringToTime(timeStr);
    ASSERT_RESULT_EQUALS(parsedTime, input);
}
```

Custom assertion

```
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/AsiaPacific
/Users/daniel/git/presentations/GoogleTest_GoogleMock/custom.cpp:94: Failure
Expected equality of these values:
  parsedTime.value()
    Which is: 1748689377
  input
    Which is: 1748689376

[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/AsiaPacific, where GetParam() = "Asia/Pacific" (0 ms)
[ RUN      ] TimeTestsInst/TimeTests.convertToFromString/AWCST
/Users/daniel/git/presentations/GoogleTest_GoogleMock/custom.cpp:94: Failure
expected 'parsedTime' to contain a value, but it contains an error: Invalid argument

[  FAILED  ] TimeTestsInst/TimeTests.convertToFromString/AWCST, where GetParam() = "AWCST" (0 ms)
```

Nicer

Testing exceptions

```
TEST(ResultTest, value_throws_on_error)
{
    Result<int> res(make_error_code(std::errc::value_too_large));
    try {
        res.value();
        GTEST_FAIL() << "Should have thrown";
    } catch (const std::system_error&) {
        // ok
    } catch (...) {
        GTEST_FAIL() << "Bad exception type";
    }
}
```

```
TEST(ResultTest, value_throws_on_error)
{
    Result<int> res(make_error_code(std::errc::value_too_large));
    EXPECT_THROW(res.value(), std::system_error);
}
```

Repeating & filtering tests

- Have a look at the test program's `--help`:
- `--gtest_filter=POSITIVE_PATTERNS[-NEGATIVE_PATTERNS]`
  - Select a subset of tests to run (e.g. the component you're working on)
- `--gtest_repeat=[COUNT]`
  - Repeat tests (great with TSAN & `stress` !)
- `--gtest_shuffle`
  - Feeling lucky?
- `--gtest_output=(json|xml)[:DIRECTORY_PATH/|:FILE_PATH]`
  - Integrate test results with other systems

```
This program contains tests written using Google Test. You can use the
following command line flags to control its behavior:

Test Selection:
  --gtest_list_tests
      List the names of all tests instead of running them. The name of
      TEST(Foo, Bar) is "Foo.Bar".
  --gtest_filter=POSITIVE_PATTERNS[-NEGATIVE_PATTERNS]
      Run only the tests whose name matches one of the positive patterns but
      none of the negative patterns. '?' matches any single character; '*'
      matches any substring; ':' separates two patterns.
  --gtest_also_run_disabled_tests
      Run all disabled tests too.

Test Execution:
  --gtest_repeat=[COUNT]
      Run the tests repeatedly; use a negative count to repeat forever.
  --gtest_shuffle
      Randomize tests' orders on every iteration.
  --gtest_random_seed=[NUMBER]
      Random number seed to use for shuffling test orders (between 1 and
      99999, or 0 to use a seed based on the current time).
  --gtest_recreate_environments_when_repeating
      Sets up and tears down the global test environment on each repeat
      of the test.

Test Output:
  --gtest_color=(yes|no|auto)
      Enable/disable colored output. The default is auto.
  --gtest_brief=1
      Only print test failures.
  --gtest_print_time=0
      Don't print the elapsed time of each test.
  --gtest_output=(json|xml)[:DIRECTORY_PATH/|:FILE_PATH]
      Generate a JSON or XML report in the given directory or with the given
      file name. FILE_PATH defaults to test_detail.xml.
  --gtest_stream_result_to=HOST:PORT
      Stream test results to the given server.

Assertion Behavior:
  --gtest_death_test_style=(fast|threadsafe)
      Set the default death test style.
  --gtest_break_on_failure
      Turn assertion failures into debugger break-points.
  --gtest_throw_on_failure
      Turn assertion failures into C++ exceptions for use by an external
      test framework.
  --gtest_catch_exceptions=0
      Do not report exceptions as test failures. Instead, allow them
      to crash the program or throw a pop-up (on Windows).

Except for --gtest_list_tests, you can alternatively set the corresponding
environment variable of a flag (all letters in upper-case). For example, to
disable colored text output, you can either specify --gtest_color=no or set
the GTEST_COLOR environment variable to no.

For more information, please read the Google Test documentation at
https://github.com/google/googletest/. If you find a bug in Google Test
(not one in your own code or tests), please report it to
<googletestframework@googlegroups.com>.
```

Death tests

- Great for my crash handler library :)
- Check that the program terminates with a specific message

Death tests

```
static int getValueSafe(const Result<int>& res) noexcept
{
    return res.value();
}

TEST(ResultTest, death_test_ok)
{
    Result<int> res(make_error_code(std::errc::value_too_large));
    ASSERT_DEATH(getValueSafe(res), "Value too large to be stored in data type");
}
TEST(ResultTest, death_test_fail)
{
    Result<int> res(make_error_code(std::errc::io_error));
    ASSERT_DEATH(getValueSafe(res), "Value too large to be stored in data type");
}
TEST(ResultTest, death_test_fail2)
{
    Result<int> res(7);
    ASSERT_DEATH(getValueSafe(res), "Some message");
}
```

Death tests

```
[ RUN      ] ResultTest.death_test_ok
[       OK ] ResultTest.death_test_ok (2 ms)
[ RUN      ] ResultTest.death_test_fail
/Users/daniel/git/presentations/GoogleTest_GoogleMock/custom.cpp:117: Failure
Death test: getValueSafe(res)
    Result: died but not with expected error.
  Expected: contains regular expression "Value too large to be stored in data type"
Actual msg:
[  DEATH   ] libc++abi: terminating due to uncaught exception of type std::__1::system_error: Input/output error
[  DEATH   ]

[  FAILED  ] ResultTest.death_test_fail (1 ms)
[ RUN      ] ResultTest.death_test_fail2
/Users/daniel/git/presentations/GoogleTest_GoogleMock/custom.cpp:122: Failure
Death test: getValueSafe(res)
    Result: failed to die.
 Error msg:
[  DEATH   ]

[  FAILED  ] ResultTest.death_test_fail2 (1 ms)
```

Advanced mocking

- Quick refresher:

```
TEST(ServerTest, WHEN_can_list_persons_in_db_THEN_returns_serialized_persons)
{
    std::vector<Person> samplePersons;
    samplePersons.emplace_back();

    auto dbConn = std::make_shared<StrictMock<MockDbConnection>>();
    EXPECT_CALL(*dbConn, listPersons()).WillOnce(Return(samplePersons));

    NiceMock<MockClientConnection> conn;
    EXPECT_CALL(conn, sendResponse(_, "<serialized persons>"));

    Server server(dbConn);
    server.listPersons(conn);
}
```

Advanced mocking

- (My) most common problems:
  - How can I mock a C function?
  - How can I mock overloads?
  - How can I match partial object properties?
  - Mocks & multiprocessing (i.e. `fork()`)

Advanced mocking: C functions

```cpp
bool writeFile(const char* path, std::string_view content)
{
    FILE* f = fopen(path, "rb");
    // ...
    if (fclose(f) != 0)
    {
        // TODO: error handling ... when will this happen?!
    }
    return true;
}
```

- https://drewdevault.com/2016/07/19/Using-Wl-wrap-for-mocking-in-C.html

Advanced mocking: C functions

- Compile with `-Wl,--wrap=fclose`
  - Replaces calls to fclose with `__wrap_fclose`
  - Provides the original symbol as `__real_fclose`
- Feel free to delegate calls to a mock class
- Use `MockFunction` if you already have an `std::function`

Advanced mocking: C functions

```
#include <cstdio>

extern "C" int __wrap_fclose(FILE*);
extern "C" int __real_fclose(FILE*);

// "raw C" approach
int fclose_error = 0;

int __wrap_fclose(FILE* f)
{
    // really close the file
    EXPECT_EQ(__real_fclose(f), 0) << strerror(errno);
    // return a (potentially) "fake" error
    return fclose_error;
}
```

Advanced mocking: C functions

```
// "gmock approach"
class MockFclose
{
public:
    MOCK_METHOD(int, close, (FILE*));
};
MockFclose* fclose_mock = nullptr; // set from your test

int __wrap_fclose(FILE* f)
{
    int rc = __real_fclose(f);
    EXPECT_EQ(rc, 0) << strerror(errno);

    // delegate to the mock (if any)
    if (fclose_mock)
        return fclose_mock->close(f);
    else
        return rc;
}
```

Advanced mocking: overloads

- Straight-forward: add all overloaded methods in the mock class
- If expectations are ambiguous etc., forward to unique methods
  - Same approach for methods with default parameters

Advanced mocking: overloads

```
struct Stringify
{
    virtual std::string to_string(int value);
    virtual std::string to_string(long value);
    virtual std::string to_string(std::string_view value);
};
struct MockStringify1 : public Stringify
{
    MOCK_METHOD(std::string, to_string, (int value), (override));
    MOCK_METHOD(std::string, to_string, (long value), (override));
    MOCK_METHOD(std::string, to_string, (std::string_view value), (override));
};
struct MockStringify2 : public Stringify
{
    MOCK_METHOD(std::string, to_string_int, (int value));
    std::string to_string(int value) override { return to_string_int(value); }
    MOCK_METHOD(std::string, to_string_long, (long value));
    std::string to_string(long value) override { return to_string_long(value); }
    MOCK_METHOD(std::string, to_string_sv, (std::string_view value));
    std::string to_string(std::string_view value) override { return to_string_sv(value); }
};
```

Advanced mocking: matching object properties

- `EXPECT_CALL(mock, function(arg))`
  - Compares args fully (==)
  - May be „too much“ for the current test case (can make tests brittle)
- How to check only „some properties“ of arg?

Advanced mocking: matching object properties

```
struct Json { /* ... */ };


struct MockClientConnection : IClientConnection
{
    MOCK_METHOD(void, sendResponse, (int code, const Json& data), (override));
};


TEST(Server, response_version)
{
    StrictMock<MockClientConnection> conn;
    // ...
    EXPECT_CALL(conn, sendResponse(200, /* some JSON with version=1 */));
}
```

Advanced mocking: matching object properties („simple“ solution)

```
struct Json { /* ... */ };

std::optional<int> getMemberAsInt(const Json&, const char*);

struct MockClientConnection : IClientConnection
{
    MOCK_METHOD(void, sendResponse, (int code, const Json& data), (override));
};


TEST(Server, response_version)
{
    StrictMock<MockClientConnection> conn;
    // ...
    EXPECT_CALL(conn, sendResponse(200, _)).WillOnce([] (int, const Json& data) {
        EXPECT_EQ(getMemberAsInt(data, "version"), 1);
    });
}
```

(1) match every JSON param

(2) verify properties in body

Advanced mocking: matching object properties („official“ solution)

```
struct Json { /* ... */ };

std::optional<int> getMemberAsInt(const Json&, const char*);

struct MockClientConnection : IClientConnection
{
    MOCK_METHOD(void, sendResponse, (int code, const Json& data), (override));
};

TEST(Server, response_version)
{
    StrictMock<MockClientConnection> conn;
    // ...
    auto getVersion = [](const Json& j) { return getMemberAsInt(j, "version"); };
    EXPECT_CALL(conn, sendResponse(200, testing::ResultOf(getVersion, testing::Eq(1))));
}
```

Advanced mocking: matching object properties („official“ solution)

```
struct Json { /* ... */ };

std::optional<int> getMemberAsInt(const Json&, const char*);

struct MockClientConnection : IClientConnection
{
    MOCK_METHOD(void, sendString, (int code, const std::string& data));
};

TEST(Server, response_version)
{
    StrictMock<MockClientConnection> conn;
    // ...
    EXPECT_CALL(conn, sendString(200,
                                 testing::Property(&std::string::size, testing::Gt(42))));
}
```

Advanced mocking: forks

Advanced mocking: `fork()`s

- Mocks check their expectations in the destructor
  - If they are set before forking and fulfilled in the child, the parent's mock will complain…
- Set expectations in the correct process
  - Use factory to create instances on-demand when you need them

Best practices

- Keep your tests *readable*, but also *fast*
  - Balance test case length vs. setup/runtime overhead
  - Focus on on *content*, not housekeeping
    - Move helper stuff and mocks to dedicated files/libs
- Provide as much *information* as possible on failure
  - Use specific macros (avoid `ASSERT_TRUE/FALSE`)
  - Define custom predicates/macros if necessary/useful
- Keep tests *independent* and *repeatable*
- Naming tests is hard, but valuable (given-when-then)