Introduction into GoogleTest & GoogleMock

C++ User Group Aachen

Why unit tests?

- Make sure your code works
  - While writing it
  - While changing it
- Manual tests are expensive
- Feel confident in your code :)
  - Greatest combo: unit tests + sanitizers
  - Aim for high coverage

The Beyonce Rule

(Software Engineering at Google)

Disclaimer

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

Sample program

```
class Stringify
{
public:
    std::string to_string(int value);
};

std::string Stringify::to_string(int value)
{
    std::string s;
    while (value > 0)
    {
        int8_t remainder = value % 10;
        value /= 10;
        s += ('0' + remainder);
    }
    return s;
}
```

GoogleTest

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

TEST(Stringify, to_string1)
{
    Stringify str;
    ASSERT_TRUE(str.to_string(5) == "5");
}
```

GoogleTest

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

TEST(Stringify, to_string1)
{
    Stringify str;
    ASSERT_TRUE(str.to_string(5) == "5");
}
```

GoogleTest

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

TEST(Stringify, to_string1)
{
    Stringify str;
    ASSERT_TRUE(str.to_string(5) == "5");
}
```

Test suite name

Test name

GoogleTest

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

TEST(Stringify, to_string1)
{
    Stringify str;
    ASSERT_TRUE(str.to_string(5) == "5");
}
```

Test suite name

Test name

void function

GoogleTest

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

TEST(Stringify, to_string1)
{
    Stringify str;
    ASSERT_TRUE(str.to_string(5) == "5");
}
```

Test suite name

Test name

void function

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from Stringify
[ RUN      ] Stringify.to_string1
[       OK ] Stringify.to_string1 (0 ms)
[----------] 1 test from Stringify (0 ms total)

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

```
TEST(Stringify, to_string2)
{
    Stringify str;
    ASSERT_TRUE(str.to_string(123) == "123");
}
```

```
TEST(Stringify, to_string2)
{
    Stringify str;
    ASSERT_TRUE(str.to_string(123) == "123");
}
```

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from Stringify
[ RUN      ] Stringify.to_string1
[       OK ] Stringify.to_string1 (0 ms)
[ RUN      ] Stringify.to_string2
/Users/daniel/git/presentations/GoogleTest_GoogleMock/tests.cpp:14: Failure
Value of: str.to_string(123) == "123"
  Actual: false
Expected: true

[  FAILED  ] Stringify.to_string2 (0 ms)
[----------] 2 tests from Stringify (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] Stringify.to_string2

 1 FAILED TEST
```

Assertion macros

- `ASSERT_TRUE, EXPECT_TRUE` `condition`
- `ASSERT_FALSE, EXPECT_FALSE` `!condition`
- `ASSERT_EQ, EXPECT_EQ` `a == b`
- `ASSERT_NE, EXPECT_NE` `a != b`
- … `_LE/LT/GE/GT` `a <= b, …`

Assertion macros

Fatal

- `ASSERT_TRUE, EXPECT_TRUE` `condition`
- `ASSERT_FALSE, EXPECT_FALSE` `!condition`
- `ASSERT_EQ, EXPECT_EQ` `a == b`
- `ASSERT_NE, EXPECT_NE` `a != b`
- … `_LE/LT/GE/GT` `a <= b, …`

Assertion macros

Fatal

Non-fatal

- `ASSERT_TRUE, EXPECT_TRUE` `condition`
- `ASSERT_FALSE, EXPECT_FALSE` `!condition`
- `ASSERT_EQ, EXPECT_EQ` `a == b`
- `ASSERT_NE, EXPECT_NE` `a != b`
- … `_LE/LT/GE/GT` `a <= b, …`

Assertion macros for C strings

- `EXPECT_STREQ` `s1 == s2`
- `EXPECT_STRNE` `s1 != s2`
- `EXPECT_STRCASEEQ` `lower(s1) == lower(s2)`
- `EXPECT_STRCASENE` `lower(s1) != lower(s2)`

Assertion macros for C strings

- `EXPECT_STREQ` `s1 == s2`
- `EXPECT_STRNE` `s1 != s2`
- `EXPECT_STRCASEEQ` `lower(s1) == lower(s2)`
- `EXPECT_STRCASENE` `lower(s1) != lower(s2)`

… + many more for floating points and general predicates

```
TEST(Stringify, to_string2)
{
    Stringify str;
    ASSERT_EQ(str.to_string(123), "123");
}
```

```
TEST(Stringify, to_string2)
{
    Stringify str;
    ASSERT_EQ(str.to_string(123), "123");
}
```

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from Stringify
[ RUN      ] Stringify.to_string1
[       OK ] Stringify.to_string1 (0 ms)
[ RUN      ] Stringify.to_string2
/Users/daniel/git/presentations/GoogleTest_GoogleMock/tests.cpp:14: Failure
Expected equality of these values:
  str.to_string(123)
    Which is: "321"
  "123"

[  FAILED  ] Stringify.to_string2 (0 ms)
[----------] 2 tests from Stringify (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] Stringify.to_string2

 1 FAILED TEST
```

```cpp
class Stringify
{
public:
    std::string to_string(int value);
};

std::string Stringify::to_string(int value)
{
    std::string s;
    while (value > 0)
    {
        int8_t remainder = value % 10;
        value /= 10;
        s += ('0' + remainder);
    }
    return s;
}
```

```cpp
class Stringify
{
public:
    std::string to_string(int value);
};

std::string Stringify::to_string(int value)
{
    std::string s;
    while (value > 0)
    {
        int8_t remainder = value % 10;
        value /= 10;
        s.insert(0, 1, '0' + remainder);
    }
    return s;
}
```

```cpp
class Stringify
{
public:
    std::string to_string(int value);
};

std::string Stringify::to_string(int value)
{
    std::string s;
    while (value > 0)
    {
        int8_t remainder = value % 10;
        value /= 10;
        s.insert(0, 1, '0' + remainder);
    }
    return s;
}
```

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from Stringify
[ RUN      ] Stringify.to_string1
[       OK ] Stringify.to_string1 (0 ms)
[ RUN      ] Stringify.to_string2
[       OK ] Stringify.to_string2 (0 ms)
[----------] 2 tests from Stringify (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 2 tests.
```

```
TEST(Stringify, to_string3)
{
    Stringify str;
    ASSERT_EQ(str.to_string(4711), "4711");
    ASSERT_EQ(str.to_string(0), "0");
    ASSERT_EQ(str.to_string(-12), "-12");
    // ...
}
```

```cpp
TEST(Stringify, to_string3)
{
    Stringify str;
    ASSERT_EQ(str.to_string(4711), "4711");
    ASSERT_EQ(str.to_string(0), "0");
    ASSERT_EQ(str.to_string(-12), "-12");
    // ...
}
```

```
[ RUN      ] Stringify.to_string3
/Users/daniel/git/presentations/GoogleTest_GoogleMock/tests.cpp:21: Failure
Expected equality of these values:
  str.to_string(0)
    Which is: ""
  "0"

[  FAILED  ] Stringify.to_string3 (0 ms)
```

```
TEST(Stringify, to_string3)
{
    Stringify str;
    ASSERT_EQ(str.to_string(4711), "4711");
    ASSERT_EQ(str.to_string(0), "0");
    ASSERT_EQ(str.to_string(-12), "-12");
    // ...
}
```

Never executed

```
[ RUN      ] Stringify.to_string3
/Users/daniel/git/presentations/GoogleTest_GoogleMock/tests.cpp:21: Failure
Expected equality of these values:
  str.to_string(0)
    Which is: ""
  "0"

[  FAILED  ] Stringify.to_string3 (0 ms)
```

```
TEST(Stringify, to_string3)
{
    Stringify str;
    EXPECT_EQ(str.to_string(4711), "4711");
    EXPECT_EQ(str.to_string(0), "0");
    EXPECT_EQ(str.to_string(-12), "-12");
    // ...
}
```

```cpp
TEST(Stringify, to_string3)
{
    Stringify str;
    EXPECT_EQ(str.to_string(4711), "4711");
    EXPECT_EQ(str.to_string(0), "0");
    EXPECT_EQ(str.to_string(-12), "-12");
    // ...
}
```

```
[ RUN      ] Stringify.to_string3
/Users/daniel/git/presentations/GoogleTest_GoogleMock/tests.cpp:21: Failure
Expected equality of these values:
  str.to_string(0)
    Which is: ""
  "0"

/Users/daniel/git/presentations/GoogleTest_GoogleMock/tests.cpp:22: Failure
Expected equality of these values:
  str.to_string(-12)
    Which is: ""
  "-12"

[  FAILED  ] Stringify.to_string3 (0 ms)
```

```cpp
std::string Stringify::to_string(int value)
{
    std::string s;
    while (value > 0)
    {
        int8_t remainder = value % 10;
        value /= 10;
        s.insert(0, 1, '0' + remainder);
    }

    return s;
}
```

```cpp
std::string Stringify::to_string(int value)
{
    std::string s;
    do
    {
        int8_t remainder = value % 10;
        value /= 10;
        s.insert(0, 1, '0' + remainder);
    } while (value > 0);

    return s;
}
```

```
std::string Stringify::to_string(int value)
{
    bool neg = value < 0;
    if (neg) {
        value = abs(value);
    }

    std::string s;
    do
    {
        int8_t remainder = value % 10;
        value /= 10;
        s.insert(0, 1, '0' + remainder);
    } while (value > 0);

    if (neg) {
        s.insert(0, 1, '-');
    }
    return s;
}
```

```cpp
std::string Stringify::to_string(int value)
{
    bool neg = value < 0;
    if (neg) {
        value = abs(value);
    }

    std::string s;
    do
    {
        int8_t remainder = value % 10;
        value /= 10;
        s.insert(0, 1, '0' + remainder);
    } while (value > 0);

    if (neg) {
        s.insert(0, 1, '-');
    }
    return s;
}
```

```
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from Stringify
[ RUN      ] Stringify.to_string1
[       OK ] Stringify.to_string1 (0 ms)
[ RUN      ] Stringify.to_string2
[       OK ] Stringify.to_string2 (0 ms)
[ RUN      ] Stringify.to_string3
[       OK ] Stringify.to_string3 (0 ms)
[----------] 3 tests from Stringify (0 ms total)

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

```
TEST(Stringify, to_string3)
{
    Stringify str;
    EXPECT_EQ(str.to_string(4711), "4711");
    EXPECT_EQ(str.to_string(0), "0");
    EXPECT_EQ(str.to_string(-12), "-12");
    // ...
}
```

```
TEST(Stringify, to_string3)
{
    Stringify str;
    EXPECT_EQ(str.to_string(4711), "4711");
    EXPECT_EQ(str.to_string(0), "0");
    EXPECT_EQ(str.to_string(-12), "-12");
    // ...
}
```

Repeated setup

Test fixtures: setup & teardown

Test fixtures: setup & teardown

- Fixture is instantiated for every test (not shared)

Test fixtures: setup & teardown

- Fixture is instantiated for every test (not shared)
- Preparations can be implemented in constructor or `SetUp()`

Test fixtures: setup & teardown

- Fixture is instantiated for every test (not shared)
- Preparations can be implemented in constructor or `SetUp()`
- Cleanup can be implemented in destructor or `TearDown()`

Test fixtures: setup & teardown

- Fixture is instantiated for every test (not shared)
- Preparations can be implemented in constructor or `SetUp()`
- Cleanup can be implemented in destructor or `TearDown()`
- Use constructor/destructor for constant objects

Test fixtures: setup & teardown

- Fixture is instantiated for every test (not shared)
- Preparations can be implemented in constructor or `SetUp()`
- Cleanup can be implemented in destructor or `TearDown()`
- Use constructor/destructor for constant objects
- Use `SetUp()/TearDown()` with `ASSERT_xx` and exceptions

Test fixtures: setup & teardown

- Fixture is instantiated for every test (not shared)
- Preparations can be implemented in constructor or `SetUp()`
- Cleanup can be implemented in destructor or `TearDown()`
- Use constructor/destructor for constant objects
- Use `SetUp()/TearDown()` with `ASSERT_xx` and exceptions
- Beware to call the parent's functions when deriving & overwriting

```
class StringifyTest : public testing::Test
{
protected:
    // StringifyTest() = default;
    // ~StringifyTest() override = default;
    // void SetUp() override {}
    // void TearDown() override {}
    Stringify str;
};
```

```
class StringifyTest : public testing::Test
{
protected:
    // StringifyTest() = default;
    // ~StringifyTest() override = default;
    // void SetUp() override {}
    // void TearDown() override {}
    Stringify str;
};



TEST_F(StringifyTest, to_string4)
{
    EXPECT_EQ(str.to_string(4711), "4711");
    EXPECT_EQ(str.to_string(0), "0");
    EXPECT_EQ(str.to_string(-12), "-12");
    // ...
}
```

But my code is more complicated …

```
class Person { /* ... */ };

struct DbConnection
{
    class Params {};
    std::vector<Person> listPersons() const;
};

struct ClientConnection
{
    void sendResponse(int code, const std::string& data);
};

struct Server
{
    explicit Server(const DbConnection::Params&);
    void listPersons(ClientConnection& conn)
    {
        try {
            std::vector<Person> persons = m_dbConn.listPersons();
            conn.sendResponse(200, to_json(persons));
        } catch (const std::exception& exc) {
            conn.sendResponse(500, exc.what());
        }
    }
    DbConnection m_dbConn;
};
```

Should I set up a database for my unit tests?

Should I set up a database
for my unit tests?

No - mock external dependencies!

Nomenclature

- **Fake**: simplified/limited implementation of an interface
  - e.g. in-memory database or filesystem
- **Stub**: provides predefined answers and/or records calls
- **Mock**: verifies pre-programmed expectations
  - That’s what we need here.

Add interfaces

```
struct IDbConnection
{
    virtual ~IDbConnection() = default;
    virtual std::vector<Person> listPersons() const = 0;
};

struct DbConnection final : IDbConnection
{
    std::vector<Person> listPersons() const final;
};

struct IClientConnection
{
    virtual ~IClientConnection() = default;
    virtual void sendResponse(int code, const std::string& data) = 0;
};

struct ClientConnection final : IClientConnection
{
    void sendResponse(int code, const std::string& data) final;
};
```

```
struct Server
{
    explicit Server(std::shared_ptr<IDbConnection> db);

    void listPersons(IClientConnection& conn)
    {
        try
        {
            std::vector<Person> persons = m_dbConn->listPersons();
            conn.sendResponse(200, to_json(persons));
        }
        catch (const std::exception& exc)
        {
            conn.sendResponse(500, exc.what());
        }
    }

    std::shared_ptr<IDbConnection> m_dbConn;
};
```

```
struct Server
{
    explicit Server(std::shared_ptr<IDbConnection> db);

    void listPersons(IClientConnection& conn)
    {
        try
        {
            std::vector<Person> persons = m_dbConn->listPersons();
            conn.sendResponse(200, to_json(persons));
        }
        catch (const std::exception& exc)
        {
            conn.sendResponse(500, exc.what());
        }
    }

    std::shared_ptr<IDbConnection> m_dbConn;
};
```

```
struct Server
{
    explicit Server(std::shared_ptr<IDbConnection> db);

    void listPersons(IClientConnection& conn)
    {
        try
        {
            std::vector<Person> persons = m_dbConn->listPersons();
            conn.sendResponse(200, to_json(persons));
        }
        catch (const std::exception& exc)
        {
            conn.sendResponse(500, exc.what());
        }
    }

    std::shared_ptr<IDbConnection> m_dbConn;
};
```

```
struct Server
{
    explicit Server(std::shared_ptr<IDbConnection> db);

    void listPersons(IClientConnection& conn)
    {
        try
        {
            std::vector<Person> persons = m_dbConn->listPersons();
            conn.sendResponse(200, to_json(persons));
        }
        catch (const std::exception& exc)
        {
            conn.sendResponse(500, exc.what());
        }
    }

    std::shared_ptr<IDbConnection> m_dbConn;
};
```

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

struct MockDbConnection : IDbConnection
{
    MOCK_METHOD(std::vector<Person>, listPersons, (), (const override));
};

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

TEST(ServerTest, first_mock)
{
    auto dbConn = std::make_shared<MockDbConnection>();
    MockClientConnection conn;

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

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from ServerTest
[ RUN      ] ServerTest.first_mock

GMOCK WARNING:
Uninteresting mock function call - returning default value.
    Function call: listPersons()
          Returns: {}
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress
it by blindly adding an EXPECT_CALL() if you don't mean to enforce the call.  See https://
github.com/google/googletest/blob/main/docs/gmock_cook_book.md#knowing-when-to-expect-useoncall for
details.

GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: sendResponse(200, @0x16cfab090 "{}")
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress
it by blindly adding an EXPECT_CALL() if you don't mean to enforce the call.  See https://
github.com/google/googletest/blob/main/docs/gmock_cook_book.md#knowing-when-to-expect-useoncall for
details.
[       OK ] ServerTest.first_mock (0 ms)
[----------] 1 test from ServerTest (0 ms total)
```

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from ServerTest
[ RUN      ] ServerTest.first_mock

GMOCK WARNING:
Uninteresting mock function call - returning default value.
    Function call: listPersons()
          Returns: {}
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress
it by blindly adding an EXPECT_CALL() if you don't mean to enforce the call.  See https://
github.com/google/googletest/blob/main/docs/gmock_cook_book.md#knowing-when-to-expect-useoncall for
details.

GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: sendResponse(200, @0x16cfab090 "{}")
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress
it by blindly adding an EXPECT_CALL() if you don't mean to enforce the call.  See https://
github.com/google/googletest/blob/main/docs/gmock_cook_book.md#knowing-when-to-expect-useoncall for
details.
[       OK ] ServerTest.first_mock (0 ms)
[----------] 1 test from ServerTest (0 ms total)
```

- Mocks have default behavior

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from ServerTest
[ RUN      ] ServerTest.first_mock

GMOCK WARNING:
Uninteresting mock function call - returning default value.
    Function call: listPersons()
          Returns: {}
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress
it by blindly adding an EXPECT_CALL() if you don't mean to enforce the call.  See https://
github.com/google/googletest/blob/main/docs/gmock_cook_book.md#knowing-when-to-expect-useoncall for
details.

GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: sendResponse(200, @0x16cfab090 "{}")
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress
it by blindly adding an EXPECT_CALL() if you don't mean to enforce the call.  See https://
github.com/google/googletest/blob/main/docs/gmock_cook_book.md#knowing-when-to-expect-useoncall for
details.
[       OK ] ServerTest.first_mock (0 ms)
[----------] 1 test from ServerTest (0 ms total)
```

- Mocks have default behavior
- Test succeeds without any expectations

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from ServerTest
[ RUN      ] ServerTest.first_mock

GMOCK WARNING:
Uninteresting mock function call - returning default value.
    Function call: listPersons()
          Returns: {}
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress
it by blindly adding an EXPECT_CALL() if you don't mean to enforce the call.  See https://
github.com/google/googletest/blob/main/docs/gmock_cook_book.md#knowing-when-to-expect-useoncall for
details.

GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: sendResponse(200, @0x16cfab090 "{}")
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress
it by blindly adding an EXPECT_CALL() if you don't mean to enforce the call.  See https://
github.com/google/googletest/blob/main/docs/gmock_cook_book.md#knowing-when-to-expect-useoncall for
details.
[       OK ] ServerTest.first_mock (0 ms)
[----------] 1 test from ServerTest (0 ms total)
```

- Mocks have default behavior
- Test succeeds without any expectations
- I don’t like warnings …

The Nice, the Strict, and the Naggy

- `MockFoo`: warns about "uninteresting calls"
- `NiceMock<MockFoo>`: doesn’t care
- `StrictMock<MockFoo>`: fails the test in case of unexpected calls

```
using testing::StrictMock;
using testing::NiceMock;

TEST(ServerTest, mock_types)
{
    auto dbConn = std::make_shared<StrictMock<MockDbConnection>>();
    NiceMock<MockClientConnection> conn;

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

```cpp
using testing::StrictMock;
using testing::NiceMock;

TEST(ServerTest, mock_types)
{
    auto dbConn = std::make_shared<StrictMock<MockDbConnection>>();
    NiceMock<MockClientConnection> conn;

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

```
[ RUN      ] ServerTest.mock_types
unknown file: Failure
Uninteresting mock function call - returning default value.
    Function call: listPersons()
          Returns: {}

[  FAILED  ] ServerTest.mock_types (0 ms)
```

```
using testing::Return;
using testing::_;

TEST(ServerTest, WHEN_can_list_persons_in_db_THEN_returns_200)
{
    std::vector<Person> samplePersons;
    samplePersons.emplace_back();

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

    StrictMock<MockClientConnection> conn;
    EXPECT_CALL(conn, sendResponse(200, _));

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

```cpp
using testing::Return;
using testing::_;

TEST(ServerTest, WHEN_can_list_persons_in_db_THEN_returns_200)
{
    std::vector<Person> samplePersons;
    samplePersons.emplace_back();

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

    StrictMock<MockClientConnection> conn;
    EXPECT_CALL(conn, sendResponse(200, _));

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

```cpp
using testing::Return;
using testing::_;

TEST(ServerTest, WHEN_can_list_persons_in_db_THEN_returns_200)
{
    std::vector<Person> samplePersons;
    samplePersons.emplace_back();

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

    StrictMock<MockClientConnection> conn;
    EXPECT_CALL(conn, sendResponse(200, _));

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

```
using testing::Return;
using testing::_;

TEST(ServerTest, WHEN_can_list_persons_in_db_THEN_returns_200)
{
    std::vector<Person> samplePersons;
    samplePersons.emplace_back();

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

    StrictMock<MockClientConnection> conn;
    EXPECT_CALL(conn, sendResponse(200, _));

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

```
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));

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

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

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

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

    StrictMock<MockClientConnection> conn;
    EXPECT_CALL(conn, sendResponse(123, _));

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

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

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

    StrictMock<MockClientConnection> conn;
    EXPECT_CALL(conn, sendResponse(123, _));

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

```
Unexpected mock function call - returning directly.
    Function call: sendResponse(200, @0x16d016f50 "<serialized persons>")
Google Mock tried the following 1 expectation, but it didn't match:

/Users/daniel/git/presentations/GoogleTest_GoogleMock/tests_with_mocks.cpp:79: EXPECT_CALL(conn, sendResponse(123, _))...
  Expected arg #0: is equal to 123
           Actual: 200
         Expected: to be called once
           Actual: never called - unsatisfied and active

/Users/daniel/git/presentations/GoogleTest_GoogleMock/tests_with_mocks.cpp:79: Failure
Actual function call count doesn't match EXPECT_CALL(conn, sendResponse(123, _))...
         Expected: to be called once
           Actual: never called - unsatisfied and active

[  FAILED  ] ServerTest.test_failed_expectation (0 ms)
```

Summary

- Please use unit tests in every project!
  - … using whatever library/framework you like most
- `ASSERT` or `EXPECT` as needed
- Use test fixtures for common setup and teardown
- Mock your dependencies - use interfaces
- Start strict
- Use *C++ Test Mate* in VS code

http://google.github.io/googletest/

Next time

- Scopes, custom messages and printers
- Generic predicate assertions, asserting exceptions
- Parametrized tests (values & types)
- Repeating & filtering tests
- Death tests
- Advanced matchers
- Coverage (gcov, OpenCppCoverage)
- Lots of (my) best practices …