C++ testing

Zoltán Porkoláb, PhD.
gsd@elte.hu
http://gsd.web.elte.hu

Outline

- General rules
- Gtest
- Gmock
- Coverage
- Dependency injection techniques
- Static analysis

General rules of testing

- Correctness
- Readability
- Completeness
- Demonstratives
- Resilience

Correctness

- Test depend upon known bugs – BAD!

```
int square( int x)
{
  // TODO: Implement
  return 0;
}
TEST (SquareTest, MathTests)
{
  EXPECT_EQ( 0, square(2));
  EXPECT_EQ( 0, square(3));
  EXPECT_EQ( 0, square(5));
}
```

Correctness

- Code review!
- If the test fails we should decide who to blame!
  - The author of the code?
  - The writer of the test?

```
int square( int x)
{
  // TODO: Implement
  return 0;
}
TEST (SquareTest, MathTests)
{
  EXPECT_EQ( 4, square(2));
  EXPECT_EQ( 9, square(3));
  EXPECT_EQ(25, square(5));
}
```

Correctness

- Test that do not execute real scenarios – BAD!
- Testing the Mock, not the real world

```
class MockWorld : public World
{
  // assume world is flat
  bool isFlat() override { return true; }
}
TEST (Flat, WorldTests)
{
  MockWorld world;
  EXPECT_TRUE( world.Populate() );
  EXPECT_TRUE( world.isFlat() );
}
```

Readablity

- Test should be obvious to the future reader
  - Including yourself
- Typical mistakes
  - Too much boilerplate – too much distraction

```
TEST (BigSystemTest, CallIsUnimplemented)
{
  TestStorageSystem storage;
  auto testData = getTestDileMap();
  storage.MapFilesystem(test_data);
  BigSystem system;
  ASSERT_OK( system.initialize(5));
  ThreadPool pool(10);
  pool.startThreads();
  storage.setThreads(pool);
  System.setStorage(storage);

  ASSERT_TRUE( system.isRunning() );
  EXPECT_TRUE( isUnimplemented(system.status()) );  //actual test
}
```

Readablity

- Typical mistakes
  - Not enough context – hiding important details

```
TEST (BigSystemTest, ReadMagicBytes)
{
  BigSystem system = initTestSystemAndTestData();
  EXPECT_EQ( 42, system.PrivateKey()) );  //actual test
}
```

- At least comment it!

Readablity

- Typical mistakes
  - Do not use advanced test features when not necessary – KISS

```
class BigSystemTest : public ::testing::Test
{
public:
  BigSystemTest() : filename_(“/tmp/test”) { }
  void SetUp()
  {
    ASSERT_OK( file::writeData(filename_, “Hello world\n!” ) );
  }
protected:
  BigSystem   system_;
  std::string filename_;
};

TEST_F( BigSystemTest, BasicTest)
{
  EXPECT_TRUE( system_.initialize() );
}
```

Readablity

- Typical mistakes
  - Do not use advanced test features when not necessary – KISS

```
class BigSystemTest : public ::testing::Test
{
public:
  BigSystemTest() : filename_(“/tmp/test”) { }
  void SetUp()
  {
    ASSERT_OK( file::writeData(filename_, “Hello world\n!” ) );
  }
protected:
  BigSystem   system_;
  std::string filename_;
};
```

```
TEST( BigSystemTest, BasicTest)
{
  BigSystem system;
  EXPECT_TRUE( system_.initialize() );
}
```

Readablity

- A test is like a novel
  - Setup
  - Action
  - Conclusion

Completeness

- Typical mistakes
  - Test for the easy cases – BAD!

```
TEST (FactorialTest, BasicTest)
{
  EXPECT_EQ(  1, factorial(1) );
  EXPECT_EQ(120, factorial(5) );
};
```

Completeness

- Typical mistakes
  - Test for the easy cases – BAD!

```
TEST (FactorialTest, BasicTest)
{
  EXPECT_EQ(  1, factorial(1) );
  EXPECT_EQ(120, factorial(5) );
};

int factorial( int n)
{
  if ( 1 == n )  return 1;
  if ( 5 == n )  return 120;
}
```

Completeness

- Test for all the edge cases

```
TEST (FactorialTest, basicTests)
{
  EXPECT_EQ( 1,   Factorial(1) );
  EXPECT_EQ( 120, Factorial(5) );

  EXPECT_EQ( 1,   Factorial(0) );
  EXPECT_EQ( 479001600, Factorial(12) );

  // overflow
  EXPECT_EQ( std::numeric_limits::max<int>(), Factorial(13) );

  // check: no internal state
  EXPECT_EQ( 1,   Factorial(0) );
  EXPECT_EQ( 120, Factorial(5) );
}
```

Completeness

- Test driven design:
  - Write test first, not driven by implementation
  - Write test only for the next feature to implement

Completeness

- Test only what we are responsible
  - Test what we implemented

```
TEST (FilterTest, WithVector)
{
  vector<int> v;   // make sure vector is working
  v.push_back(1);
  EXPECT_EQ( 1, v.size() );
  v.clear();
  EXPECT_EQ( 0, v.size() );
  EXPECT_TRUE( v.empty() );

  // Now test our stuff
  v = Filter( { 1,2,3,4,5 }, [](int x) { 0 == return x % 2; } );
  EXPECT_THAT( v, ElementsAre(2,4) );
}
```

Completeness

- Typical mistakes
  - Test what we are not responsible for

```
TEST (FilterTest, WithVector)
{
  vector<int> v;   // make sure vector is working
  v.push_back(1);
  EXPECT_EQ( 1, v.size() );
  v.clear();
  EXPECT_EQ( 0, v.size() );
  EXPECT_TRUE( v.empty() );

  // Now test our stuff
  v = Filter( { 1,2,3,4,5 }, [](int x) { 0 == return x % 2; } );
  EXPECT_THAT( v, ElementsAre(2,4) );
}
```

Demonstrability

- Clients will learn the system via tests
- Tests should serve as a demonstration of how the API works
- Typical mistakes
  - Using private API is bad.
  - Using friends + test only methods are bad. – later we refine this
  - Bad usage in unit tests suggesting a bad API

Demonstrability

- No user can call ShortcutSetupForTesting
- But sometimes we have to check the state after action

```
class Foo
{
  friend FooTest;
public:
  bool Setup();
private:
  bool ShortcutSetupForTesting();
};

TEST (FooTest, Setup)
{
  EXPECT_TRUE( ShortcutSetupForTesting() );
}
```

Resilience

- Write tests that depend only on published API guarantees!
- Typical mistakes
  - Flaky tests (re-run gets different results)
  - Brittle tests (depends on too many assumptions, implementation details)
  - Tests depending on execution order
  - Non-hermetic tests
  - Mocks depending upon underlying APIs

Resilience

- Flaky test
  - Multiple runs get different results

```
TEST ( UpdaterTest, RunsFast)
{
  Updater updater;
  updater.updateAsync();
  sleepFor(Seconds(.5));  // should be enough
  EXPECT_TRUE( updater.updated() );
}
```

```
// e.g. RotatingLogFile
```

Resilience

- Brittle test
  - Tests that can fail for changes unrelated to the tested code
  - Reason might be change in our code – but not this part

```
TEST ( Tags, ContentsAreCorrect)
{
  TagSet tags = {5,8,10};  // unordered set

  EXPECT_THAT( tags, ElementsAre(5,8,10) );
}
```

Resilience

- Brittle test
  - Tests that can fail for changes unrelated to the tested code
  - Reason might be change in our code – but not this part

```
TEST ( Tags, ContentsAreCorrect)
{
  TagSet tags = {5,8,10};  // unordered set

  EXPECT_THAT( tags, UnorderedElementsAre(5,8,10) );
}
```

Resilience

- Brittle test
  - Tests that can fail for changes unrelated to the tested code
  - Reason might be change in our code – but not this part

```
TEST ( MyTest, LogWasCalled)
{
  StartLogCapture();
  EXPECT_TRUE( Frobber::start() );
  EXPECT_TRUE( Logs(),
           Contains("file.cc:421: OpenedFile frobber.config") );
}




// Use regular expressions
// Boundaries for the file location
```

Resilience

- Execution order
  - Tests fail if they aren't run all together or in a particular order.
  - Tests fail if they aren't run in a particular order.

```
static int i = 0;

TEST ( Foo, First)
{
  ASSERT_EQ( 0, i);
  ++i;
}

TEST ( Foo, Second)
{
  ASSERT_EQ( 1, i);
  ++i;
}
```

Resilience

- Execution order
  - Many test framework runs test cases parallel
  - Global state is bad idea – hidden dependency.
  - Files, threads...

Resilience

- Hermetic
  - Test fails if anyone else runs the same test at the same time.

```
TEST (Foo, StorageTest)
{
  StorageServer *server = GetStorageServerHandle();
  auto val = rand();

  server->Store("testkey", val);
  EXPECT_EQ( val, server->Load("testkey") );
}


// std::this_thread::get_id()
// putenv()
```

Resilience

- Deep dependency
  - Depends on the underlaying implementation not on the tested code
  - Will fail when the implementation changes

```
class File
{
public:
  ...
  virtual bool Stat( Stat *stat);
  virtual bool StatWithOptions( Stat *stat, Options, options)
  {
    return Stat(stat);  // ignore options
  }
};

TEST (MyTest, FSUsage)
{
  EXPECT_CALL( file, Stat(_)).Times(1);
  Frobber::Stat();
}
```

Design goals

- Correctness: write test testing what you wanted to test
- Readability : write readable tests, use code review
- Completeness: test all edge cases, but test only what you are responsible for
- Demonstrability: show how to use the API
- Resilience:
  - Stable, hermetic, correct, non-order-dependent, only breaks when unacceptable behavior change happens

Design goals – are not rules

- No test is perfect
- No reason to hunt perfection
- Questions:
  - Who writes the test: implementer or somebody else?
  - What to do when we have a large test with complex state?
  - How to test asynchronous events?
    - Does it have the correct result?
    - Is it within time limits?

Google test

- Unit testing library
- Based on xUnit
  - SUnit – Kent Back 1998, Smalltalk
  - Highly object-oriented structure
  - Ported to many languages: Java: JUnit, R: Runit, ...
- Components
  - Test runner
  - Test cases
  - Test fixtures (set of preconditions)
  - Test suit (sharing the same fixture)
  - Test results formatter
  - Assertations

Basic test

```
#include <gtest/gtest.h>
TEST (BasicTest, OneEqOne)
{
  EXPECT_EQ( 1, 1);
}
int main( int argc, char *argv[])
{
  ::testing::InitGoogleTest( &argc, argv);
  return RUN_ALL_TESTS();
}
$ g++ -I../../googletest/googletest/include/ basic1.cpp
                ../../lib/libgmock.a -pthread -o basic1
$ ./basic1
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from BasicTest
[ RUN      ] BasicTest.OneEqOne
[       OK ] BasicTest.OneEqOne (0 ms)
[----------] 1 test from BasicTest (0 ms total)

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

Basic test

- RUN_ALL_TESTS
- Macro magic: std::vector to collect test cases
- Automatically detects and runs test cases defined by TEST macro
- Must be called only once

Factorial

```
/* minimath.h */
#ifndef MINIMATH_H
#define MINIMATH_H

class MiniMath
{
public:
  int factorial(int n);
};

#endif

/* minimath.cpp */
#include "minimath.h"

int MiniMath::factorial(int n)
{
  int res = 1;
  for(int i=2; i<=n; ++i)
    res *= i;
  return res;
}
```

Factorial

```
/* test.c */
#include <gtest/gtest.h>
#include "minimath.h"

TEST(FactorialTest, withPositiveNumbers)
{
  MiniMath mm;
  EXPECT_EQ(120, mm.factorial(5));
  EXPECT_EQ(  6, mm.factorial(3));
}

TEST(FactorialTest, withZero)
{
  MiniMath mm;
  EXPECT_EQ(1, mm.factorial(0));
}

int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Factorial

```
$  g++  -I../../googletest/googletest/include/ minimath.cpp
             test.cpp ../../lib/libgmock.a -pthread -o test
$  ./test
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from FactorialTest
[ RUN      ] FactorialTest.withPositiveNumbers
[       OK ] FactorialTest.withPositiveNumbers (0 ms)
[ RUN      ] FactorialTest.withZero
[       OK ] FactorialTest.withZero (0 ms)
[----------] 2 tests from FactorialTest (0 ms total)

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

Factorial – wrong

```
/* minimath.h */
#ifndef MINIMATH_H
#define MINIMATH_H

class MiniMath
{
public:
  int factorial(int n);
};

#endif

/* minimath.cpp */
#include "minimath.h"

int MiniMath::factorial(int n)
{
  int res = 1;
  for(int i=2; i<=n; ++i)
    res *= i;
  return res==120 ? 1 : res;  // BUG HERE!
}
```

Factorial

```
$  g++  -I../../googletest/googletest/include/ minimath.cpp
             test.cpp ../../lib/libgmock.a -pthread -o test
$ ./test
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from FactorialTest
[ RUN      ] FactorialTest.withPositiveNumbers
test1.cpp:7: Failure
Value of: mm.factorial(5)
  Actual: 1
Expected: 120
[  FAILED  ] FactorialTest.withPositiveNumbers (0 ms)
[ RUN      ] FactorialTest.withZero
[       OK ] FactorialTest.withZero (0 ms)
[----------] 2 tests from FactorialTest (0 ms total)

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

 1 FAILED TEST
```

EXPECT_EQ vs ASSERT_EQ

```
/* test.c */
#include <gtest/gtest.h>
#include "minimath.h"

TEST(FactorialTest, withPositiveNumbers)
{
  MiniMath mm;
  ASSERT_EQ(120, mm.factorial(5));  // was: EXPECT_EQ
  printf(“***still running***”);
  ASSERT_EQ(  6, mm.factorial(3));  // was: EXPECT_EQ
}

TEST(FactorialTest, withZero)
{
  MiniMath mm;
  ASSERT_EQ(1, mm.factorial(0));    // was: EXPECT_EQ
}

int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Factorial

```
$  g++  -I../../googletest/googletest/include/ minimath.cpp
             test.cpp ../../lib/libgmock.a -pthread -o test
$ ./test
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from FactorialTest
[ RUN      ] FactorialTest.withPositiveNumbers
test1.cpp:7: Failure
Value of: mm.factorial(5)
  Actual: 1
Expected: 120
[  FAILED  ] FactorialTest.withPositiveNumbers (0 ms)
[ RUN      ] FactorialTest.withZero
[       OK ] FactorialTest.withZero (0 ms)
[----------] 2 tests from FactorialTest (0 ms total)

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

 1 FAILED TEST
```

EXPECT and ASSERT API

- ASSERT_TRUE ASSERT_FALSE
- ASSERT_EQ ASSERT_NE ASSERT_LT ASSERT_GT ASSERT_GE ...
- ASSERT_STREQ ASSERT_STRNE

  ASSERT_STRCASEEQ ASSERT_STRCASEBE
- ASSERT_TRUE ASSERT_FALSE

- Same for EXPECT_*

- SUCCEED() // not used, reserved
- FAIL()
- ADD_FAILURE()

Command line arguments

- Generate XML output
- Convertable to formatted HTML

```
$ test1 --gtest_output="xml:test1-report.xml"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="2" failures="0" disabled="0" errors="0"
timestamp="2016-01-09T19:21:32" time="0" name="AllTests">
  <testsuite name="FactorialTest" tests="2" failures="0"
disabled="0" errors="0" time="0">
    <testcase name="withPositiveNumbers" status="run" time="0"
classname="FactorialTest" />
    <testcase name="withZero" status="run" time="0"
classname="FactorialTest" />
  </testsuite>
</testsuites>
```

Command line arguments

- Repeating all tests

```
$ ./test --gtest_repeat=3
Repeating all tests (iteration 1) . .

[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from FactorialTest
[ RUN      ] FactorialTest.withPositiveNumbers
test1.cpp:7: Failure
Value of: mm.factorial(5)
  Actual: 1
Expected: 120
[  FAILED  ] FactorialTest.withPositiveNumbers (0 ms)
[ RUN      ] FactorialTest.withZero
[       OK ] FactorialTest.withZero (0 ms)
[----------] 2 tests from FactorialTest (0 ms total)

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

 1 FAILED TEST
```

Command line arguments

- Repeating all tests

```
$ ./test --gtest_repeat=3
Repeating all tests (iteration 2) . .

[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from FactorialTest
[ RUN      ] FactorialTest.withPositiveNumbers
test1.cpp:7: Failure
Value of: mm.factorial(5)
  Actual: 1
Expected: 120
[  FAILED  ] FactorialTest.withPositiveNumbers (0 ms)
[ RUN      ] FactorialTest.withZero
[       OK ] FactorialTest.withZero (0 ms)
[----------] 2 tests from FactorialTest (0 ms total)

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

 1 FAILED TEST
```

Command line arguments

- Filtering

```
$ ./test1 --gtest_filter=FactorialTest.withZero
Note: Google Test filter = FactorialTest.withZero

[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 1 tests from FactorialTest
[ RUN      ] FactorialTest.withZero
[       OK ] FactorialTest.withZero (0 ms)
[----------] 1 tests from FactorialTest (0 ms total)

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

Command line arguments

- Filtering

```
$ ./test1 --gtest_filter=FactorialTest.withZero
Note: Google Test filter = FactorialTest.withZero

$ ./test1 --gtest_filter=FactorialTest.*Zero
Note: Google Test filter = FactorialTest.*Zero

$ ./test1 –gtest_filter=FactorialTest.*-FactorialTest.withPositiveNu
mbers
Note: Google Test filter = FactorialTest.*FactorialTest.withPositive
Numbers
```

Floating point

```
/* minimath.h */
#ifndef MINIMATH_H
#define MINIMATH_H
class MiniMath
{
public:
  int factorial(int n);
  double div(double x, double y);
};
#endif

/* minimath.cpp */
#include "minimath.h"
int MiniMath::factorial(int n)
{
  int res = 1;
  for(int i=2; i<=n; ++i)
    res *= i;
  return res;
}
double MiniMath::div( double x, double y)
{
  return x/y;
}
```

Floating point

```
/* test.c */

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

TEST(DivisionTest, SimpleTest)
{
  MiniMath mm;
  EXPECT_EQ(1.66, mm.div(5,3));
}

int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Floating point

```
$ ./test1
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from DivisionTest
[ RUN      ] DivisionTest.SimpleTest
test1.cpp:7: Failure
Value of: mm.div(5,3)
  Actual: 1.66667
Expected: 1.66
[  FAILED  ] DivisionTest.SimpleTest (0 ms)
[----------] 1 test from DivisionTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] DivisionTest.SimpleTest

 1 FAILED TEST
```

Floating point

```
/* test.c */

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

TEST(DivisionTest, SimpleTest)
{
  MiniMath mm;
  EXPECT_EQ(1.66667, mm.div(5,3));
}

int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Floating point

```
$ ./test1
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from DivisionTest
[ RUN      ] DivisionTest.SimpleTest
test1.cpp:7: Failure
Value of: mm.div(5,3)
  Actual: 1.66667
Expected: 1.66667
[  FAILED  ] DivisionTest.SimpleTest (0 ms)
[----------] 1 test from DivisionTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] DivisionTest.SimpleTest

 1 FAILED TEST
```

Floating point

- EXPECT_FLOAT_EQ
- EXPECT_DOUBLE_EQ
- EXPECT_NEAR

- ASSERT_FLOAT_EQ
- ASSERT_DOUBLE_EQ
- ASSERT_NEAR

Floating point

```
/* floating point tests */
#include <gtest/gtest.h>
#include "minimath.h"

TEST(DivisionTest, FloatTest)
{
  MiniMath mm;
  EXPECT_FLOAT_EQ(1.66667, mm.div(5,3));
}
TEST(DivisionTest, DoubleTest)
{
  MiniMath mm;
  EXPECT_DOUBLE_EQ(1.66667, mm.div(5,3));
}
TEST(DivisionTest, NearTest)
{
  MiniMath mm;
  EXPECT_NEAR(1.66667, mm.div(5,3), 0.0001);
}
int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Floating point

```
$ ./test2
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from DivisionTest
[ RUN      ] DivisionTest.FloatTest
test2.cpp:7: Failure
Value of: mm.div(5,3)
  Actual: 1.6666666
Expected: 1.66667
[  FAILED  ] DivisionTest.FloatTest (1 ms)
[ RUN      ] DivisionTest.DoubleTest
test2.cpp:13: Failure
Value of: mm.div(5,3)
  Actual: 1.6666666666666667
Expected: 1.66667
Which is: 1.6666700000000001
[  FAILED  ] DivisionTest.DoubleTest (0 ms)
[ RUN      ] DivisionTest.NearTest
[       OK ] DivisionTest.NearTest (0 ms)
[----------] 3 tests from DivisionTest (1 ms total)
[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 2 tests, listed below:
[  FAILED  ] DivisionTest.FloatTest
[  FAILED  ] DivisionTest.DoubleTest
 2 FAILED TESTS
```

Floating point

```
/* floating point tests */
#include <gtest/gtest.h>
#include "minimath.h"

/* test near */
#include <gtest/gtest.h>
#include "minimath.h"

TEST(DivisionTest, Float6digit)
{
  EXPECT_NEAR(1.6666661, 1.6666669, 1e-7);
}

TEST(DivisionTest, Float7digit)
{
  EXPECT_NEAR(1.66666661, 1.66666669, 1e-7);
}

int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Floating point

```
$ ./test4
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from DivisionTest
[ RUN      ] DivisionTest.Float6digit
test4.cpp:6: Failure
The difference between 1.6666661 and 1.6666669 is
8.0000000002300453e-07,
which exceeds 1e-7, where
1.6666661 evaluates to 1.6666661,
1.6666669 evaluates to 1.6666669000000001, and
1e-7 evaluates to 9.9999999999999995e-08.
[  FAILED  ] DivisionTest.Float6digit (0 ms)
[ RUN      ] DivisionTest.Float7digit
[       OK ] DivisionTest.Float7digit (0 ms)
[----------] 2 tests from DivisionTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] DivisionTest.Float6digit

 1 FAILED TES
```

Floating point as string

```
/* floating point as string tests */
#include <gtest/gtest.h>
#include <iomanip>
#include <sstream>

TEST(StringTest, Expect_Eq)
{
  double d = 5./3.;
  std::ostringstream s;
  s << std::setprecision(6) << d;
  EXPECT_EQ("1.66667", s.str());  // "1.66667" converted
}
TEST(StringTest, Expect_StringEq)
{
  double d = 5./3.;
  std::ostringstream s;
  s << std::setprecision(6) << d;
  EXPECT_STREQ("1.66667", s.str().c_str());
}
int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Floating point

```
$ ./test1
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from StringTest
[ RUN      ] StringTest.Expect_Eq
[       OK ] StringTest.Expect_Eq (0 ms)
[ RUN      ] StringTest.Expect_StringEq
[       OK ] StringTest.Expect_StringEq (0 ms)
[----------] 2 tests from StringTest (0 ms total)

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

Death tests

```
/* minimath.cpp */
#include <iostream>
#include <cstdlib>
#include "minimath.h"

int MiniMath::factorial(int n)
{
  if ( n < 0 )
  {
    std::cerr << "Negative input" << std::endl;
    std::exit(1);
  }
  int res = 1;
  for(int i=2; i<=n; ++i)
    res *= i;
  return res;
}

double MiniMath::div( double x, double y)
{
  return x/y;
}
```

Death tests

- ASSERT_DEATH(statement, expected_message)
- ASSERT_EXIT(statement, predicate, expected_message)

since gtest version 1.4.0:

- ASSERT_DEATH_IF_SUPPORTED
- EXPECT_DEATH_IF_SUPPORTED

Death tests

```
/* death tests */
TEST(FactorialTest, withNegative1)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial( 0),::testing::ExitedWithCode(1),"");
}
TEST(FactorialTest, withNegative2)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial(-1),::testing::ExitedWithCode(-1),"");
}
TEST(FactorialTest, withNegative3)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial(-1),::testing::ExitedWithCode(1),
                                                     "Bad input");
}
TEST(FactorialTest, withNegative4)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial(-1),::testing::ExitedWithCode(1),
                                                "Negative input");
}
```

Death tests

```
TEST(FactorialTest, withNegative1)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial( 0),::testing::ExitedWithCode(1),"");
}
```

```
$ ./test1
[==========] Running 4 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 4 tests from FactorialTest
[ RUN      ] FactorialTest.withNegative1
test1.cpp:15: Failure
Death test: mm.factorial( 0)
    Result: failed to die.
 Error msg:
[  DEATH   ]
[  FAILED  ] FactorialTest.withNegative1 (0 ms)
```

Death tests

```
TEST(FactorialTest, withNegative2)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial(-1),::testing::ExitedWithCode(-1),"");
}

[ RUN      ] FactorialTest.withNegative2
test1.cpp:21: Failure
Death test: mm.factorial(-1)
    Result: died but not with expected exit code:
            Exited with exit status 1
Actual msg:
[  DEATH   ] Negative input
[  DEATH   ]
[  FAILED  ] FactorialTest.withNegative2 (0 ms)
```

Death tests

```
TEST(FactorialTest, withNegative3)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial(-1),::testing::ExitedWithCode(1),
                                                "Negative input");
}
```

```
[ RUN      ] FactorialTest.withNegative3
test1.cpp:27: Failure
Death test: mm.factorial(-1)
    Result: died but not with expected error.
  Expected: Bad input
Actual msg:
[  DEATH   ] Negative input
[  DEATH   ]
[  FAILED  ] FactorialTest.withNegative3 (1 ms)
```

Death tests

```
EST(FactorialTest, withNegative4)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial(-1),::testing::ExitedWithCode(1),
                                                "Negative input");
}

[ RUN      ] FactorialTest.withNegative4
[       OK ] FactorialTest.withNegative4 (1 ms)
[ RUN      ] FactorialTest.withZero
[       OK ] FactorialTest.withZero (0 ms)
[----------] 4 tests from FactorialTest (3 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (3 ms total)
[  PASSED  ] 1 tests.
[  FAILED  ] 3 tests, listed below:
[  FAILED  ] FactorialTest.withNegative1
[  FAILED  ] FactorialTest.withNegative2
[  FAILED  ] FactorialTest.withNegative3

 3 FAILED TESTS
```

Be care with error messages

```
TEST(FactorialTest, withNegative3)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial(-1),::testing::ExitedWithCode(1),"");
}
...
[ RUN      ] FactorialTest.withNegative3
[       OK ] FactorialTest.withNegative3 (1 ms)
[ RUN      ] FactorialTest.withNegative4
[       OK ] FactorialTest.withNegative4 (1 ms)
[ RUN      ] FactorialTest.withZero
[       OK ] FactorialTest.withZero (0 ms)
[----------] 4 tests from FactorialTest (3 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (3 ms total)
[  PASSED  ] 2 tests.
[  FAILED  ] 2 tests, listed below:
[  FAILED  ] FactorialTest.withNegative1
[  FAILED  ] FactorialTest.withNegative2

 2 FAILED TESTS
```

Death predicates

- `::testing::ExitedWithCode(exit_code)`
- `::testing::KilledBySignal(signal_number)`

  `(not available on Windows)`

Death predicates

```
/* minimath.cpp */
#include <iostream>
#include <cstdlib>
#include "minimath.h"

int MiniMath::factorial(int n)
{
  if ( n < 0 )
  {
    std::cerr << "Negative input" << std::endl;
    kill( getpid(), SIGUSR1);
  }
  int res = 1;
  for(int i=2; i<=n; ++i)
    res *= i;
  return res;
}


double MiniMath::div( double x, double y)
{
  return x/y;
}
```

Death predicates

```
TEST(FactorialTest, withNegative1)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial( 0),::testing::ExitedWithCode(1),"");
}
TEST(FactorialTest, withNegative2)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial(-1),
               ::testing::KilledBySignal(SIGUSR1),
               "Negative input");
}
```

```
[ RUN      ] FactorialTest.withNegative1
test4.cpp:17: Failure
Death test: mm.factorial( 0)
    Result: failed to die.
 Error msg:
[  DEATH   ]
[  FAILED  ] FactorialTest.withNegative1 (1 ms)
[ RUN      ] FactorialTest.withNegative2
[       OK ] FactorialTest.withNegative2 (0 ms)
```

Exceptions

- ASSERT_THROW
- ASSERT_ANY_THROW
- ASSERT_NO_THROW

Exceptions

```
TEST(FactorialTest, notThrow)
{
  MiniMath mm;
  ASSERT_NO_THROW( mm.factorial(0) );
}

TEST(FactorialTest, throwError)
{
  MiniMath mm;
  ASSERT_THROW( mm.factorial(-1), MiniMath::Error );
}

TEST(FactorialTest, throwSomething)
{
  MiniMath mm;
  ASSERT_ANY_THROW( mm.factorial(-1) );
}
```

Exceptions

```
/*
 *  ASSERT_NO_THROW takes a statement (not an expression)
 *  as an argument
 */


TEST(FactorialTest, notThrow)
{
  ASSERT_NO_THROW(
  {
    MiniMath mm;
    mm.factorial(0);
  }
  );
}
```

Death tests – how they work?

- Starting a new process and execute death test there

- ::testing::GTEST_FLAG(death_test_style)

  is set by --gtest_death_test_style parameter

- e.g.

  --gtest_death_test_style=(fast|threadsafe)

User defined predicates

- Sometimes we have to check complex expressions
- We can use EXPECT_TRUE(expr)
- Problem: this will not show details about the failure

```
ASSERT_PRED1( pred, arg)
ASSERT_PRED2( pred, arg1, arg2)
EXPECT_PRED1( pred, arg)
EXPECT_PRED2( pred, arg1, arg2)
```

- Up to 5 parameters

User predicates

```
#ifndef MINIMATH_H  /* minimath.h with gcd and mutPrime */
#define MINIMATH_H

class MiniMath
{
public:
  int gcd(int a, int b);               // greatest common divider
  static bool mutPrime(int a, int b);  // is mutual prime
};
#endif

#include "minimath.h"
int MiniMath::gcd(int a, int b)
{
  while( a != b )
    if( a > b )   a -= b;
    else          b -= a;
  return a;
}
bool MiniMath::mutPrime(int a, int b)
{
  MiniMath mm;
  return 1 == mm.gcd( a, b);
}
```

User predicates

```
#include <iostream>
#include <gtest/gtest.h>
#include "minimath.h"

TEST(MiniMath, gcd)
{
  MiniMath mm;
  EXPECT_EQ(1, mm.gcd(9,16) );
  EXPECT_EQ(4, mm.gcd(12,8) );
  EXPECT_EQ(5, mm.gcd(15,10) );
}
TEST(MiniMath, mutPrime)
{
  EXPECT_TRUE( MiniMath::mutPrime(9,16) );
  EXPECT_FALSE( MiniMath::mutPrime(12,8) );
  EXPECT_TRUE( MiniMath::mutPrime(3*5,2*5) ); // should fail
}
TEST(MiniMath, mutPrimePred)
{
  EXPECT_PRED2( MiniMath::mutPrime, 9,16 );
  EXPECT_PRED2( MiniMath::mutPrime,12, 8 );   // should fail
  EXPECT_PRED2( MiniMath::mutPrime,3*5,2*5 );
}
```

User predicates

```
$ ./test1
[----------] 3 tests from MiniMath
[ RUN      ] MiniMath.gcd
[       OK ] MiniMath.gcd (0 ms)
[ RUN      ] MiniMath.mutPrime
test1.cpp:18: Failure
Value of: MiniMath::mutPrime(3*5,2*5)
  Actual: false
Expected: true
[  FAILED  ] MiniMath.mutPrime (1 ms)
[ RUN      ] MiniMath.mutPrimePred
test1.cpp:24: Failure
MiniMath::mutPrime(12, 8) evaluates to false, where
12 evaluates to 12
8 evaluates to 8
test1.cpp:25: Failure
MiniMath::mutPrime(3*5, 2*5) evaluates to false, where
3*5 evaluates to 15
2*5 evaluates to 10
[  FAILED  ] MiniMath.mutPrimePred (0 ms)
[----------] 3 tests from MiniMath (1 ms total)
```

User predicates

```
EXPECT_PRED2( ! MiniMath::mutPrime,12, 8);
```

```
test1.cpp:24:28: warning: the address of static bool
MiniMath::mutPrime(int, int) will always evaluate as trueâ [-
Waddress]
   EXPECT_PRED2(! MiniMath::mutPrime,12, 8);
                            ^
```

Assertion objects

- An AssertionResult object represents the result of an assertion

  Whether it is a success or a failure + an associate message)
- AssertionResult can be created using these factory functions
- The operator<< is used to stream messages to the AssertionResult object.

```
namespace testing
{
  // Returns an AssertionResult object to indicate that succeeded.
  AssertionResult AssertionSuccess();

  // Returns an AssertionResult object to indicate that failed.
  AssertionResult AssertionFailure();
}
```

User predicates

```
::testing::AssertionResult isMutPrime( int a, int b)
{
  MiniMath mm;
  if ( MiniMath::mutPrime(a,b) )
    return ::testing::AssertionSuccess();
  else
    return ::testing::AssertionFailure() << "gcd(" << a
                                         << "," << b << ") ="
                                         << mm.gcd(a,b);
}
TEST(MiniMath, gcd)
{
  MiniMath mm;
  EXPECT_EQ(1, mm.gcd(9,16) );
  EXPECT_EQ(4, mm.gcd(12,8) );
  EXPECT_EQ(5, mm.gcd(15,10) );
}
TEST(MiniMath, mutPrime)
{
  EXPECT_TRUE(isMutPrime(9,16) );
  EXPECT_FALSE(isMutPrime(12,8) );   // should fail
  EXPECT_TRUE(isMutPrime(3*5,2*5) ); // should fail
}
```

User predicates

```
$ ./test2
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from MiniMath
[ RUN      ] MiniMath.gcd
[       OK ] MiniMath.gcd (0 ms)
[ RUN      ] MiniMath.mutPrime
test2.cpp:29: Failure
Value of: isMutPrime(3*5,2*5)
  Actual: false (gcd(15,10) = 5)
Expected: true
[  FAILED  ] MiniMath.mutPrime (0 ms)
[----------] 2 tests from MiniMath (0 ms total)

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

 1 FAILED TEST
```

Assertion objects

- Also, assertation objects can be used as boolean expressions

  EXPECT_TRUE(! isMutPrime(12,8));

- EXPECT_PRED_FORMAT1 ... macros allow you further formatting

Type assertions

::testing::StaticAssertTypeEq<T1, T2>()

falls back to std::static_assert(expr, msg);

Assertions can be put in any subroutine, but assertions that generate a fatal failure (FAIL_ and ASSERT_) can only be used in void-returning functions.

Fixtures

- We usually execute some initialization before executing unit tests. Test fixtures are for helping this initialization task. They are especially useful when multiple test cases share common resources.
- A fixture class should be inherited from ::testing::Test class.
- Its data members are accessible from the tests
- Instead of TEST macro we should use TEST_F with the fixture class name as the mandatory first parameter of the macro
- Fixtures have SetUp and TearDown virtual methods
- SetUp runs before each test cases
- TearDown runs after each test cases
- These should be defined as public or protected methods.

Fixtures

```
/* test with fixtures */
class MiniMathTest : public ::testing::Test
{
protected:
  MiniMath mm;

  void SetUp() { std::cout << "Before test" << std::endl; }
  void TearDown() { std::cout << "After test" << std::endl; }
};
TEST_F(MiniMathTest, withPositiveNumbers)
{
  EXPECT_EQ(120, mm.factorial(5));
  EXPECT_EQ(6, mm.factorial(3));
}
TEST_F(MiniMathTest, withZero)
{
  EXPECT_EQ(1, mm.factorial(0));
}
int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Fixtures

```
$ ./test1
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from MiniMathTest
[ RUN      ] MiniMathTest.withPositiveNumbers
Before test
After test
[       OK ] MiniMathTest.withPositiveNumbers (0 ms)
[ RUN      ] MiniMathTest.withZero
Before test
After test
[       OK ] MiniMathTest.withZero (0 ms)
[----------] 2 tests from MiniMathTest (0 ms total)

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

Fixtures

```
/* death tests and fixtures */
class MiniMathTest : public ::testing::Test
{
protected:
  MiniMath mm;

  void SetUp() { std::cout << "Before test" << std::endl; }
  void TearDown() { std::cout << "After test" << std::endl; }
};
TEST_F(MiniMathTest, withPositiveNumbers)
{
  EXPECT_EQ(120, mm.factorial(5));
  EXPECT_EQ(6, mm.factorial(3));
}
TEST_F(MiniMathTest, withZero)
{
  EXPECT_EQ(1, mm.factorial(0));
}
TEST_F(MiniMathTest, withNegative)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial(-1),::testing::ExitedWithCode(1),"");
}
```

Fixtures

```
$ ./test2
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from MiniMathTest
[ RUN      ] MiniMathTest.withPositiveNumbers
Before test
After test
[       OK ] MiniMathTest.withPositiveNumbers (0 ms)
[ RUN      ] MiniMathTest.withZero
Before test
After test
[       OK ] MiniMathTest.withZero (0 ms)
[ RUN      ] MiniMathTest.withNegative
Before test
After test
[       OK ] MiniMathTest.withNegative (1 ms)
[----------] 3 tests from MiniMathTest (1 ms total)

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

Fixtures

- We can add constructor and destructor to Fixture class.
- Allocate resources can be done either in constructor or in SetUp(), deallocation in either TearDown() or destructor.
- But as usual: destructor must not throw exception!
- Hint: put ASSERT_ macros to TearDown() instead of destructor, since Google Test may throw exceptions from ASSERT_ macros in the future.
- The same test fixture is not used across multiple tests. For every unit test, the framework creates a new test fixture object.

Fixtures

```
class MiniMathTest : public ::testing::Test
{
public:
  MiniMathTest(){ std::cout<<"Fixture constructor"<<std::endl; }
  ~MiniMathTest() override { std::cout<<"Fix destr"<<std::endl;}
protected:
  MiniMath mm;
  void SetUp() override { std::cout<<"Before test"<<std::endl; }
  void TearDown() override { std::cout<<"After test"<<std::endl; }
};
TEST_F(MiniMathTest, withPositiveNumbers)
{
  EXPECT_EQ(120, mm.factorial(5));
  EXPECT_EQ(6, mm.factorial(3));
}
TEST_F(MiniMathTest, withZero)
{
  EXPECT_EQ(1, mm.factorial(0));
}
TEST_F(MiniMathTest, withNegative)
{
  MiniMath mm;
  ASSERT_EXIT( mm.factorial(-1),::testing::ExitedWithCode(1),"");
}
```

Fixtures

- Theoretically one can use static members, but this makes tests depending on execution order of cases.

```
class MiniMathTest : public ::testing::Test
{
public:
  MiniMathTest()  { std::cout<<"Fixture constructor"<<std::endl; }
  ~MiniMathTest() override { std::cout<<"Fixt destr"<<std::endl; }
  static int cnt;
protected:
  MiniMath mm;

  void SetUp() override { ++cnt; }
  void TearDown() override { std::cout<<"cnt = "<<cnt<<std::endl; }

};
int MiniMathTest::cnt = 0;
```

Fixtures

```
$ ./test4
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from MiniMathTest
[ RUN      ] MiniMathTest.withPositiveNumbers
Fixture constructor
cnt = 1
Fixture destructor
[       OK ] MiniMathTest.withPositiveNumbers (0 ms)
[ RUN      ] MiniMathTest.withZero
Fixture constructor
cnt = 2
Fixture destructor
[       OK ] MiniMathTest.withZero (0 ms)
[ RUN      ] MiniMathTest.withNegative
Fixture constructor
cnt = 3
Fixture destructor
[       OK ] MiniMathTest.withNegative (0 ms)
[----------] 3 tests from MiniMathTest (1 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (2 ms total)
[  PASSED  ] 3 tests.
```

Async calls

- Googletest does not include timeout feature yet.
- There is an open request since 2015, some of the comments are from 2015 summer. timeout: https://github.com/google/googletest/issues/348
- Anton Lipov created a nice solution using C++11:

Async calls

```
/* func.h */
#ifndef LONGFUNC_H
#define LONGFUNC_H

int long_function(int i);

#endif /* LONGFUNC_H */


/* func.cpp */
#ifndef LONGFUNC_H
#define LONGFUNC_H

#include <thread>
#include <chrono>

int long_function(int i)
{
  if ( i < 0 )
  {
    for(;;); /* forever */
  }
  std::this_thread::sleep_for(std::chrono::milliseconds(i));
  return i;
}
```

Async calls

```
/* timeout.h */
#ifndef TIMEOUT_H
#define TIMEOUT_H

#include <future>
#define TEST_TIMEOUT_BEGIN std::promise<bool> promisedFinished; \
             auto futureResult = promisedFinished.get_future(); \
             std::thread([](std::promise<bool>& finished) {

#define TEST_TIMEOUT_FAIL_END(X)  finished.set_value(true);    \
                     } , std::ref(promisedFinished)).detach(); \
EXPECT_TRUE(futureResult.wait_for(std::chrono::milliseconds(X))!= \
                              std::future_status::timeout);

#define TEST_TIMEOUT_SUCCESS_END(X)  finished.set_value(true); \
                      }, std::ref(promisedFinished)).detach(); \
EXPECT_FALSE(futureResult.wait_for(std::chrono::milliseconds(X))!= \
                              std::future_status::timeout);
#endif /* TIMEOUT_H */
```

Async calls

```
/* test1.cpp */
#include <iostream>
#include <gtest/gtest.h>
#include "timeout.h"
#include "func.h"

TEST(Timeout, NoTimeoutOk)
{
  TEST_TIMEOUT_BEGIN
    EXPECT_EQ(10, long_function(10));
  TEST_TIMEOUT_FAIL_END(1000)
}

TEST(Timeout, Timeout)
{
  TEST_TIMEOUT_BEGIN
    EXPECT_EQ(42, long_function(5000));
  TEST_TIMEOUT_FAIL_END(1000)
}
```

Async calls

```
TEST(Timeout, NoTimeoutBadReturn)
{
  TEST_TIMEOUT_BEGIN
    EXPECT_EQ(40, long_function(100));
  TEST_TIMEOUT_FAIL_END(1000)
}

TEST(Timeout, Eternity)
{
  TEST_TIMEOUT_BEGIN
    EXPECT_EQ(40, long_function(-1));
  TEST_TIMEOUT_FAIL_END(1000)
}

int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Async calls

```
$ ./test1
[==========] Running 4 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 4 tests from Timeout
[ RUN      ] Timeout.NoTimeoutOk
[       OK ] Timeout.NoTimeoutOk (10 ms)
[ RUN      ] Timeout.Timeout
test1.cpp:22: Failure
Value of: futureResult.wait_for(std::chrono::milliseconds(1000)) !=
std::future_status::timeout
  Actual: false
Expected: true
[  FAILED  ] Timeout.Timeout (1000 ms)
```

Async calls

```
[ RUN      ] Timeout.Eternity
test1.cpp:38: Failure
Value of: futureResult.wait_for(std::chrono::milliseconds(1000)) !=
std::future_status::timeout
  Actual: false
Expected: true
[  FAILED  ] Timeout.Eternity (1000 ms)
[----------] 4 tests from Timeout (2111 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (2111 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 3 tests, listed below:
[  FAILED  ] Timeout.Timeout
[  FAILED  ] Timeout.NoTimeoutBadReturn
[  FAILED  ] Timeout.Eternity

 3 FAILED TESTS
```

Practice1

Implement **deque** with tests

Mocking with gmock

- C++ is an object-oriented language. C++ objects live in a "society", they communicate with other objects with the same or different type.
- Communication
  - Sending messages
  - Receiving responses

- State-based testing (gtest) is good for testing how the object responds to messages not that good for testing when sending messages

Mocking with gmock

- We replace the communication partners with fake/mock objects to test the BEHAVIOR of the object.
- Problems with dependencies:
  - communication may be non-deterministic (e.g. time related)
  - can be flaky
  - difficult/expensive to create or reproduce (e.g. database)
  - slow
  - hard to simulate failures
  - not exists yet
- Mock object allows you to check the interaction between itself and the user (the code we test)

Mock class

- has the same interface then the real
- can control the behavior at run time
- verify interactions

Mock class

- How to create mock objects?

- By hand
- Automatically (jMock, EasyMock using reflection)
  - in C++, we have no reflection
  - therefore gMock is nor a transcript of jMock/EasyMock

- Different design choices:
  - Macros
  - DSL for expectations and actions

Mock class

- Using a mock class

```
class Foo
{
  virtual void DoThis() = 0;
  virtual bool DoThat( int n, double x) = 0;
};
```

Mock class

- Using a mock class

```
class Foo
{
  virtual void DoThis() = 0;
  virtual bool DoThat( int n, double x) = 0;
};

class MockFoo : pucblic Foo
{
  MOCK_METHOD0( DoThis, void() );
  MOCK_METHOD2( DoThat, bool(int n, double x) );
};


MockFoo mock_foo;
```

Mock template

```
template <typename Elem>
class StackInterface {
 public:
  ...
  virtual ~StackInterface();
  virtual int GetSize() const = 0;
  virtual void Push(const Elem& x) = 0;
};
```

Mock template

```
template <typename Elem>
class StackInterface {
 public:
  ...
  virtual ~StackInterface();
  virtual int GetSize() const = 0;
  virtual void Push(const Elem& x) = 0;
};

template <typename Elem>
class MockStack : public StackInterface<Elem> {
 public:
  ...
  MOCK_CONST_METHOD0_T(GetSize, int());
  MOCK_METHOD1_T(Push, void(const Elem& x));
};

MockStack<int> mock_foo;
```

Mock class

- The mock interface should answer:
  - Which methods were called?
  - What arguments?
  - How many times?
  - Which order?
  - What responses?
- DSL is used to describe these properties

Sample

- A turtle class is used by a graphical system

```
/* turtle.h */

/* Turtle abstract base class */
class Turtle
{
public:
  virtual ~Turtle() {}
  virtual void PenUp() = 0;
  virtual void PenDown() = 0;
  virtual void Forward(int distance) = 0;
  virtual void Turn(int degrees) = 0;
  virtual void GoTo(int x, int y) = 0;
  virtual int GetX() const = 0;
  virtual int GetY() const = 0;
};
```

Sample

- A Painter class calls Turtle public member functions

```
/* painter.h */
#ifndef PAINTER_H
#define PAINTER_H
#include "turtle.h"
class Painter
{
public:
  Painter( Turtle *trt);
  bool DrawCircle(int x, int y, int r);
private:
  Turtle *turtle;
};
#endif /* PAINTER_H */

/* painter.cpp */
#include "painter.h"
Painter::Painter( Turtle *trt) : turtle(trt) { }
bool Painter::DrawCircle( int x, int y, int r)
{
  return true;
}
```

Sample

- MockTurtle implements the Turtle interface
- One can use the gmock_gen.py script to generate the mock class.

```
/* mock_turtle.h */
/* The Mock Turtle */
#include <gmock/gmock.h>

class MockTurtle : public Turtle
{
public:
  MOCK_METHOD0( PenUp, void() );
  MOCK_METHOD0( PenDown, void() );
  MOCK_METHOD1( Forward, void (int distance) );
  MOCK_METHOD1( Turn, void (int degrees) );
  MOCK_METHOD2( GoTo, void (int x, int y) );
  MOCK_CONST_METHOD( GetX, int () );
  MOCK_CONST_METHOD( GetY, int () );
};
```

Sample

- In the test MockTurtle is used instead of Turtle
- Expectations are set BEFORE the actual test calls

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

#include "painter.h"
#include "mock_turtle.h"

using ::testing::AtLeast;
TEST(PainterTest, PenDownBeforeDraw)
{
  MockTurtle turtle;
  EXPECT_CALL(turtle, PenDown())   // expectations are set
      .Times(AtLeast(1));          // at least 1 call of PenDown()
  Painter painter(&turtle);
  EXPECT_TRUE(painter.DrawCircle(0, 0, 10)); // usual gtest assert
}
int main(int argc, char** argv)
{
  ::testing::InitGoogleMock(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Sample

```
$ g++ -std=c++11 -pedantic -I../../googletest/googletest/include/
-I../../googletest/googlemock/include/ painter.cpp  test1.cpp
../../lib/libgmock.a -pthread -o test1

$ ./test1
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from PainterTest
[ RUN      ] PainterTest.PenDownBeforeDraw
test1.cpp:13: Failure
Actual function call count doesn't match EXPECT_CALL(turtle,
PenDown())...
         Expected: to be called at least once
           Actual: never called - unsatisfied and active
[  FAILED  ] PainterTest.PenDownBeforeDraw (0 ms)
[----------] 1 test from PainterTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] PainterTest.PenDownBeforeDraw

 1 FAILED TEST
```

Sample

```
/* Fix it in painter.cpp */
/* painter.cpp */
#include "painter.h"

Painter::Painter( Turtle *trt) : turtle(trt) { }

bool Painter::DrawCircle( int x, int y, int r)
{
  turtle->PenDown();
  return true;
}

$ ./test1
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from PainterTest
[ RUN      ] PainterTest.PenDownBeforeDraw
[       OK ] PainterTest.PenDownBeforeDraw (0 ms)
[----------] 1 test from PainterTest (0 ms total)

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

Expect_call

- The generic form of EXPECT_CALL is:

```
EXPECT_CALL(mock_object, method(matchers))
      .With(multi_argument_matcher)
      .Times(cardinality)
      .InSequence(sequences)
      .After(expectations)
      .WillOnce(action)
      .WillRepeatedly(action)
      .RetiresOnSaturation();
```

Matchers

- The generic form of EXPECT_CALL is:

```
EXPECT_CALL(mock_object, method(matchers))
      .With(multi_argument_matcher)
      .Times(cardinality)
      .InSequence(sequences)
      .After(expectations)
      .WillOnce(action)
      .WillRepeatedly(action)
      .RetiresOnSaturation();
```

Matchers

- Are used inside EXPECT_CALL() or directly:

```
 EXPECT_THAT( value, matcher)
 ASSERT_THAT( value, matcher)      // fatal
```

- Wildcard

```
_  (underscore)        // any value of the correct type
A<type>() An<type>()   // any value of type
```

Matchers

- Comparison (these matchers make copy of value)
- If type is not copyable use **byRef(value)** **`Eq(ByRef(non_copyable_value))`**

```
Eq(value)
 value                 arg == value
Ge(value)              arg >= value
Gt(value)
Le(value)
Lt(value)
Ne(value)
IsNull()               null ptr (raw or smart)
NotNull()              not null ptr (raw or smart)
Ref(variable)          arg is reference
TypedEq<type>(value)   arg has type "type" and equal to value
DoubleEq(dvalue)       NaNs are unequal.
FloatEq(fvalue)
NanSensitiveDoubleEq(dvalue)  NaNs are equal.
NanSensitiveFloatEq(fvalue)
DoubleNear(dvalue, maxerr)
```

Matchers

- String matchers

```
ContainsRegex(string)
EndsWith(suffix)
HasSubstr(string)
MatchesRegex(string)   matches from first to last pos.
StartsWith(prefix)
StrCaseEq(string)      ignoring case
StrCaseNeq(string)     ignoring case
StrEq(string)
StrNe(string)
```

Matchers

- Wildcard

```
EXPECT_THAT( value, matcher)
ASSERT_THAT( value, matcher)      // fatal
```

Matchers

- STL containers can be checked with Eq, since they support ==

```
Contains(e)        Argument containes element e,
                   e can be a further matcher

Each(e)            Every element matches e

ElementsAre(e0, e1, ..., en)  (max 0..10 arguments)

AlementsAreArray(...)        values coming from C array,
                             init list, STL container
IsEmpty()
SizeIs(m)
UnorderedElementsAre(e0, e1, ..., en)
WhenSorted(m)                checks whether sorted with <
WhenSortedBy(comparator,m)   checks whether sorted with comp
```

Matchers

- Member matchers

```
Field( &class:field, m)     argobj.field or
                                   argptr->field matches m
Key(e)                      arg.first matches e
                                   e.g. Contains(Key(Le(5)))
Pair(m1,m2)                 std::pair, first matches m1,
                                       second matches m2
```

- Functor

```
ResultOf(f,m)          f function/functor, f(args) matches m
```

- Pointer

```
Pointee(m)            arg is (raw or smart) pointer
                                 pointing something matches m

WhenDynamicCastTo<T>(m)    dynamic_cast<T>(arg) matches m
```

Matchers

- Composit matchers

`AllOf(m1, m2, ..., mN)`	mathes all of m1 ... mN
`AnyOf(m1, m2, ..., mN)`	at least one
`Not(m)`	does not match m

- User defined matchers

```
- MATCHER macros must be used outside a function or class
- must not be side effect
- PrintToString(x) converts x value to string
```

```
MATCHER(IsEven, "") { return (arg % 2) == 0; }
```

Cardinality

- The generic form of EXPECT_CALL is:

```
EXPECT_CALL(mock_object, method(matchers))
      .With(multi_argument_matcher)
      .Times(cardinality)
      .InSequence(sequences)
      .After(expectations)
      .WillOnce(action)
      .WillRepeatedly(action)
      .RetiresOnSaturation();
```

Cardinality

- If Times() is omitted, the default is:
  - Times(1) when neither WillOnce nor WillRepeatedly specified
  - Times(n) when n WillOnes and no WillRepeatedly specified (n>=1)
  - Times(AtLeast(n)) when n WillOnes and a WillRepeatedly specified (n>=0)
- Times(0) means the method must not be called
- Cardinality can be:

```
AnyNumber()
AtLeast(n)
AtMost(n)
Between(m, n)
Exactly(n)
n
0
```

Actions

- The generic form of EXPECT_CALL is:

```
EXPECT_CALL(mock_object, method(matchers))
      .With(multi_argument_matcher)
      .Times(cardinality)
      .InSequence(sequences)
      .After(expectations)
      .WillOnce(action)
      .WillRepeatedly(action)
      .RetiresOnSaturation();
```

Actions

- String matchers

```
Return()                    void
Return(value)
ReturnArg<N>()              N-th arg
ReturnNew<T>(a1, ..., ak)   new T(a1,...,ak)
ReturnNull()
ReturnPointee(ptr)
ReturnRef(variable)
ReturnRefOfCopy(value)      copy lives as long as action
Assign(&variable, value)
DeleteArg<N>()
SaveArg<N>(pointer)         *pointer = N-th arg

Throw(exception)
Invoke(f)                    call f with args passed to mock
function
Invoke(object_pointer, &class::method)
InvokeWithoutArgs(f)
InvokeWithoutArgs(object_pointer, &class::method)
```

Expectation order

- The generic form of EXPECT_CALL is:

```
EXPECT_CALL(mock_object, method(matchers))
      .With(multi_argument_matcher)
      .Times(cardinality)
      .InSequence(sequences)
      .After(expectations)
      .WillOnce(action)
      .WillRepeatedly(action)
      .RetiresOnSaturation();
```

Expectation order

```
using ::testing::Expectation;

Expectation init_x = EXPECT_CALL(foo, InitX());
Expectation init_y = EXPECT_CALL(foo, InitY());

EXPECT_CALL(foo, Bar())  // Bar() called after InitX and InitY
    .After(init_x, init_y);


using ::testing::ExpectationSet;

ExpectationSet all_inits;
for (int i = 0; i < element_count; i++)
{
  all_inits += EXPECT_CALL(foo, InitElement(i));
}

EXPECT_CALL(foo, Bar())
    .After(all_inits);
```

Sequence

- The generic form of EXPECT_CALL is:

```
EXPECT_CALL(mock_object, method(matchers))
      .With(multi_argument_matcher)
      .Times(cardinality)
      .InSequence(sequences)
      .After(expectations)
      .WillOnce(action)
      .WillRepeatedly(action)
      .RetiresOnSaturation();
```

Sequence

```
/* First Reset() than any of GetSize() or Describe() */

using ::testing::Sequence;
Sequence s1, s2;

EXPECT_CALL(foo, Reset())
    .InSequence(s1, s2)
    .WillOnce(Return(true));
EXPECT_CALL(foo, GetSize())
    .InSequence(s1)
    .WillOnce(Return(1));
EXPECT_CALL(foo, Describe(A<const char*>()))
    .InSequence(s2)
    .WillOnce(Return("dummy"));

/* All expected calls in the same s sequence must occur
   as they were defined */
```

Sequence

```
/* strict order */

using ::testing::Sequence;
Sequence s1, s2;

EXPECT_CALL(foo, Reset())
    .InSequence(s1, s2)
    .WillOnce(Return(true));
EXPECT_CALL(foo, GetSize())
    .InSequence(s1)
    .WillOnce(Return(1));
EXPECT_CALL(foo, Describe(A<const char*>()))
    .InSequence(s2)
    .WillOnce(Return("dummy"));
```

Sequence

```
/* strict order */

using ::testing::InSequence;
{
  InSequence dummy;

  EXPECT_CALL(...)...;
  EXPECT_CALL(...)...;
  ...
  EXPECT_CALL(...)...;
}

/* all calls in the scope of dummy should be in sequence */
```

Sample

```
#ifndef PAINTER_H
#define PAINTER_H

#include "turtle.h"

class Painter
{
public:
  Painter( Turtle *trt);
  bool DrawCircle(int x, int y, int r);
  bool DrawZigzag(int n);
private:
  Turtle *turtle;
};

#endif /* PAINTER_H */
```

Sample

```
#include "painter.h"

Painter::Painter( Turtle *trt) : turtle(trt) { }

bool Painter::DrawCircle( int x, int y, int r)
{
  turtle->PenDown();
  return true;
}
bool Painter::DrawZigzag(int n)
{
  turtle->PenDown();
  for (int i = 0; i < n; ++i)
  {
    turtle->Turn(10);
    turtle->Forward(5);
  }
  return true;
}
```

Sample

```
#include <gmock/gmock.h>
#include <gtest/gtest.h>
#include "painter.h"
#include "mock_turtle.h"

using ::testing::AtLeast;
using ::testing::Ge;
TEST(PainterTest, PenDownBeforeDraw)
{
  MockTurtle turtle;
  EXPECT_CALL(turtle, PenDown()).Times(AtLeast(1));
  Painter painter(&turtle);
  EXPECT_TRUE(painter.DrawCircle(0, 0, 10));
}
TEST(PainterTest, XwithZigzag)
{
  MockTurtle turtle;
  EXPECT_CALL(turtle, Forward(Ge(2))).Times(AtLeast(3));
  Painter painter(&turtle);
  EXPECT_TRUE(painter.DrawZigzag(4));
}
int main(int argc, char** argv)
{
  ::testing::InitGoogleMock(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Sample

```
$ ./test1
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from PainterTest
[ RUN      ] PainterTest.PenDownBeforeDraw
[       OK ] PainterTest.PenDownBeforeDraw (0 ms)
[ RUN      ] PainterTest.XwithZigzag
GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: PenDown()
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
http://code.google.com/p/googlemock/wiki/CookBook#Knowing_When_to_Expect for
details.
GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: Turn(10)
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
http://code.google.com/p/googlemock/wiki/CookBook#Knowing_When_to_Expect for
details.
[       OK ] PainterTest.XwithZigzag (0 ms)
[----------] 2 tests from PainterTest (0 ms total)

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

Sample

```
#include <gmock/gmock.h>
#include <gtest/gtest.h>
#include "painter.h"
#include "mock_turtle.h"

using ::testing::AtLeast;
using ::testing::Ge;
using ::testing::InSequence;
using ::testing::_;

TEST(PainterTest, PenDownBeforeDraw)
{
  NiceMock<MockTurtle> turtle;
  EXPECT_CALL(turtle, PenDown()).Times(AtLeast(1));
  Painter painter(&turtle);
  EXPECT_TRUE(painter.DrawLine(10));
}
TEST(PainterTest, XwithZigzag)
{
  NiceMock<MockTurtle> turtle;
  EXPECT_CALL(turtle, Forward(Ge(2))).Times(AtLeast(3));
   Painter painter(&turtle);
  EXPECT_TRUE(painter.DrawZigzag(4));
}
```

Sample

```
TEST(PainterTest, DrawLineSequence)
{
  MockTurtle turtle;
  {
    InSequence dummy;

    EXPECT_CALL(turtle, PenDown());
    EXPECT_CALL(turtle, Forward(_));
    EXPECT_CALL(turtle, PenUp());
  }

  Painter painter(&turtle);
  painter.DrawLine(4);
}

int main(int argc, char** argv)
{
  ::testing::InitGoogleMock(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Sample

```
$ ./test2
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from PainterTest
[ RUN      ] PainterTest.PenDownBeforeDraw
[       OK ] PainterTest.PenDownBeforeDraw (0 ms)
[ RUN      ] PainterTest.XwithZigzag
[       OK ] PainterTest.XwithZigzag (1 ms)
[ RUN      ] PainterTest.DrawLineSequence
[       OK ] PainterTest.DrawLineSequence (0 ms)
[----------] 3 tests from PainterTest (1 ms total)

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

State maintenance

- The mock class can be defined with state.
- We can use manual implementation
- Or we can use predefined gmock features

Sample

```
class Configurator
{
public:
  virtual ~Configurator() {}

  virtual void setParamX(int n) = 0;
  virtual int  getParamX() = 0;
};
class Client
{
public:
  Client(Configurator &cfg);
  virtual ~Client() {}

  void setParamX(int n);
  void incParamXBy(int n);
  int getParamX();
private:
  Configurator & _cfg;
};
void Client::incParamXBy(int n)
{
  _cfg.setParamX(_cfg.getParamX() + n);
}
```

State maintenance

- Suppose that the initial value of paramX is A. We want to increase paramX by B each time we call incParamXBy.

- Our expectation is that if incParamXBy is called for the first time, it will result in calling cfg.setParamX(A+B).
- second call of incParamXBy(B) will result in calling cfg.setPAramX(A + 2*B)
- third call: cfg.setPAramX(A + 3*B), and so on...

- Since the Client behavior relies on Configurator, to test Client the Configurator should remember the previous paramX value: should store a state.

State maintenance

```
/* mock_configurator.h */
#ifndef MOCK_CONFIGURATOR
#define MOCK_CONFIGURATOR

#include <gmock/gmock.h>

#include "configurator.h"

class MockConfigurator : public Configurator
{
public:
  int  paramX;
  int *paramX_ptr;

  MockConfigurator()
  {
    paramX = 0;
    paramX_ptr = &paramX;
  }

  MOCK_METHOD1(setParamX, void(int n));
  MOCK_METHOD0(getParamX, int());
};

#endif /* MOCK_CONFIGURATOR */
```

State maintenance

```
/* client.h */
#ifndef CLIENT_H
#define CLIENT_H
class Client
{
public:
  Client(Configurator &cfg) : _cfg(cfg) {};
  virtual ~Client() {}

  void setParamX(int n);
  void incParamXBy(int n);
  int getParamX();
private:
  Configurator & _cfg;
};
#endif /* CLIENT_H */

/* client.cpp */
#include "configurator.h"
#include "client.h"

void Client::incParamXBy(int n)
{
  _cfg.setParamX(_cfg.getParamX() + n);
}
```

State maintenance

```
/* test1.cpp */
#include <gmock/gmock.h>
#include <gtest/gtest.h>

#include "mock_configurator.h"
#include "client.h"

using namespace testing;

TEST(PainterTest, PenDownBeforeDraw)
{
  MockConfigurator cfg;
  Client client(cfg);

  int inc_value = 10;

  // getParamX will be called a number of times.
  // when called, we will return the value pointed to by paramX_ptr.
  // Returning with ReturnPointee is necessary, since we need to
  // have the actual (updated) value each time the method is called.
  EXPECT_CALL(cfg, getParamX())
      .Times(AnyNumber())
      .WillRepeatedly(ReturnPointee(cfg.paramX_ptr));
```

State maintenance

```
/* test1.cpp */
  // SaveArg stores the 0th parameter of the call in the value
  // pointed to by paramX_ptr (paramX).
  // expectation 3
  EXPECT_CALL(cfg, setParamX(cfg.paramX + 3*inc_value))
      .Times(1)
      .WillOnce(DoAll(SaveArg<0>(cfg.paramX_ptr), Return()));
  // expectation 2
  EXPECT_CALL(cfg, setParamX(cfg.paramX + 2*inc_value))
      .Times(1)
      .WillOnce(DoAll(SaveArg<0>(cfg.paramX_ptr), Return()));
  // expectation 1
  EXPECT_CALL(cfg, setParamX(cfg.paramX + inc_value))
      .Times(1)
      .WillOnce(DoAll(SaveArg<0>(cfg.paramX_ptr), Return()));

  client.incParamXBy(inc_value); //this will match expectation 1
  client.incParamXBy(inc_value); //this will match expectation 2
  client.incParamXBy(inc_value); //this will match expectation 3
}
int main(int argc, char** argv)
{
  ::testing::InitGoogleMock(&argc, argv);
  return RUN_ALL_TESTS();
}
```

State maintenance

```
$ ./test1
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from PainterTest
[ RUN      ] PainterTest.PenDownBeforeDraw
[       OK ] PainterTest.PenDownBeforeDraw (0 ms)
[----------] 1 test from PainterTest (0 ms total)

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

State maintenance

- Other possibility
  - We could use pre-calculated values (no state is required)
  - We could use *Invoice* action

Practice2

Test **deque** memory handling mocking the **allocator**

Test coverage with Gcov

- Test coverage
  - To measure test coverage
  - For debugging
- Features
  - What lines of code are actually executed
  - How often each line of code executes
  - Multithreaded
  - Slow

Gcov

```
/* lib.h */
#ifndef LIB_H
#define LIB_H

int libfn1();
int libfn2(int b);

#endif /* LIB_H */
```

```
/* test.cpp */
#include "lib.h"

int main ()
{
     libfn1();
     libfn2(5);
}
```

```
/* lib.cpp */
#include "lib.h"
int libfn1()
{
    int a =5;
    a++;
    return (a);
}

int libfn2( int b)
{
    if (b>10)
    {
        libfn1();
        return(b);
    }
    else
        return(0);
}
```

Using gcov

```
# generates .gcno files (flow graph) per source files
# instruments the object code
$ g++ --coverage test.cpp lib.cpp  -o test1


$ ls
lib.cpp  lib.gcno  lib.h  test1  test.cpp  test.gcno


# Running the program generates .gcda files,
# containing the coverage info.
$ ./test1
$ ls
lib.cpp  lib.gcda  lib.gcno  lib.h  test1  test.cpp
test.gcda  test.gcno
```

Textual output

```
$ gcov -abcfu lib.c
Function '_Z6libfn2i'
Lines executed:60.00% of 5
No branches
No calls

Function '_Z6libfn1v'
Lines executed:100.00% of 4
No branches
No calls

File 'lib.cpp'
Lines executed:77.78% of 9
Branches executed:100.00% of 2
Taken at least once:50.00% of 2
Calls executed:0.00% of 1
Creating 'lib.cpp.gcov'
```

Textual output

```
$ gcov -abcfu lib.c
Function '_Z6libfn2i'
Lines executed:60.00% of 5
No branches
No calls

Function '_Z6libfn1v'
Lines executed:100.00% of 4
No branches
No calls

File 'lib.cpp'
Lines executed:77.78% of 9
Branches executed:100.00% of 2
Taken at least once:50.00% of 2
Calls executed:0.00% of 1
Creating 'lib.cpp.gcov'

$ ls  # .gcov files generated
lib.cpp       lib.gcda  lib.h  test.cpp   test.gcno
lib.cpp.gcov  lib.gcno  test1  test.gcda
```

Textual output

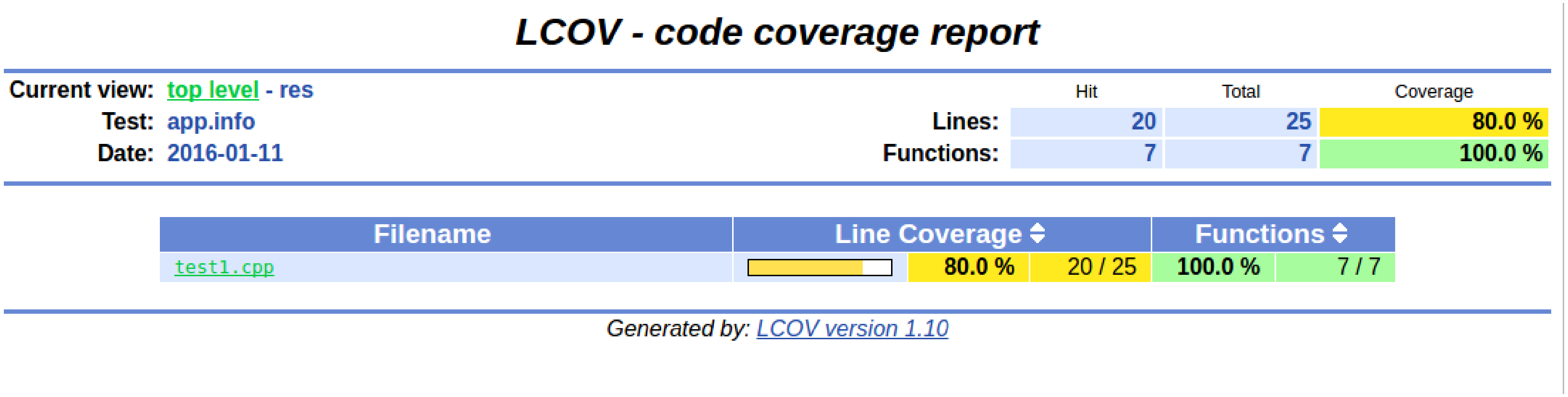
```
$ cat lib.cpp.gcov
        -:    0:Source:lib.cpp
        -:    0:Graph:lib.gcno
        -:    0:Data:lib.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include "lib.h"
        -:    2:
function _Z6libfn1v called 1 returned 100% blocks executed 100%
        1:    3:int libfn1()
        1:    3-block  0
        -:    4:{
        1:    5:    int a =5;
        1:    6:    a++;
        1:    7:    return (a);
        1:    7-block  0
unconditional  0 taken 1
        -:    8:}
        -:    9:
function _Z6libfn2i called 1 returned 100% blocks executed 60%
        1:   10:int libfn2( int b)
        1:   10-block  0
        -:   11:{
        1:   12:    if (b>10)
        1:   12-block  0
branch  0 taken 0 (fallthrough)
branch  1 taken 1
        -:   13:    {
    #####:   14:        libfn1();
    $$$$$:   14-block  0
call    0 never executed
    #####:   15:        return(b);
unconditional  0 never executed
        -:   16:    }
        -:   17:    else
        1:   18:        return(0);
        1:   18-block  0
unconditional  0 taken 1
        -:   19:}
        -:   20:
        -:   21:
```

HTML output

```
$ lcov --directory . --capture --output-file app.info
Capturing coverage data from .
Found gcov version: 4.9.3
Scanning . for .gcda files ...
Found 2 data files in .
Processing test.gcda
geninfo: WARNING: cannot find an entry for lib.cpp.gcov in .gcno
file, skipping file!
Processing lib.gcda
Finished .info-file creation

$ genhtml app.info
Reading data file app.info
Found 2 entries.
Found common filename prefix
"/home/gsd/work/zolix/tanf/NNG/gmock/mytests/9"
Writing .css and .png files.
Generating output.
Processing file h/lib.cpp
Processing file h/test.cpp
Writing directory view page.
Overall coverage rate:
  lines......: 84.6% (11 of 13 lines)
  functions..: 100.0% (3 of 3 functions)
```

HTML output



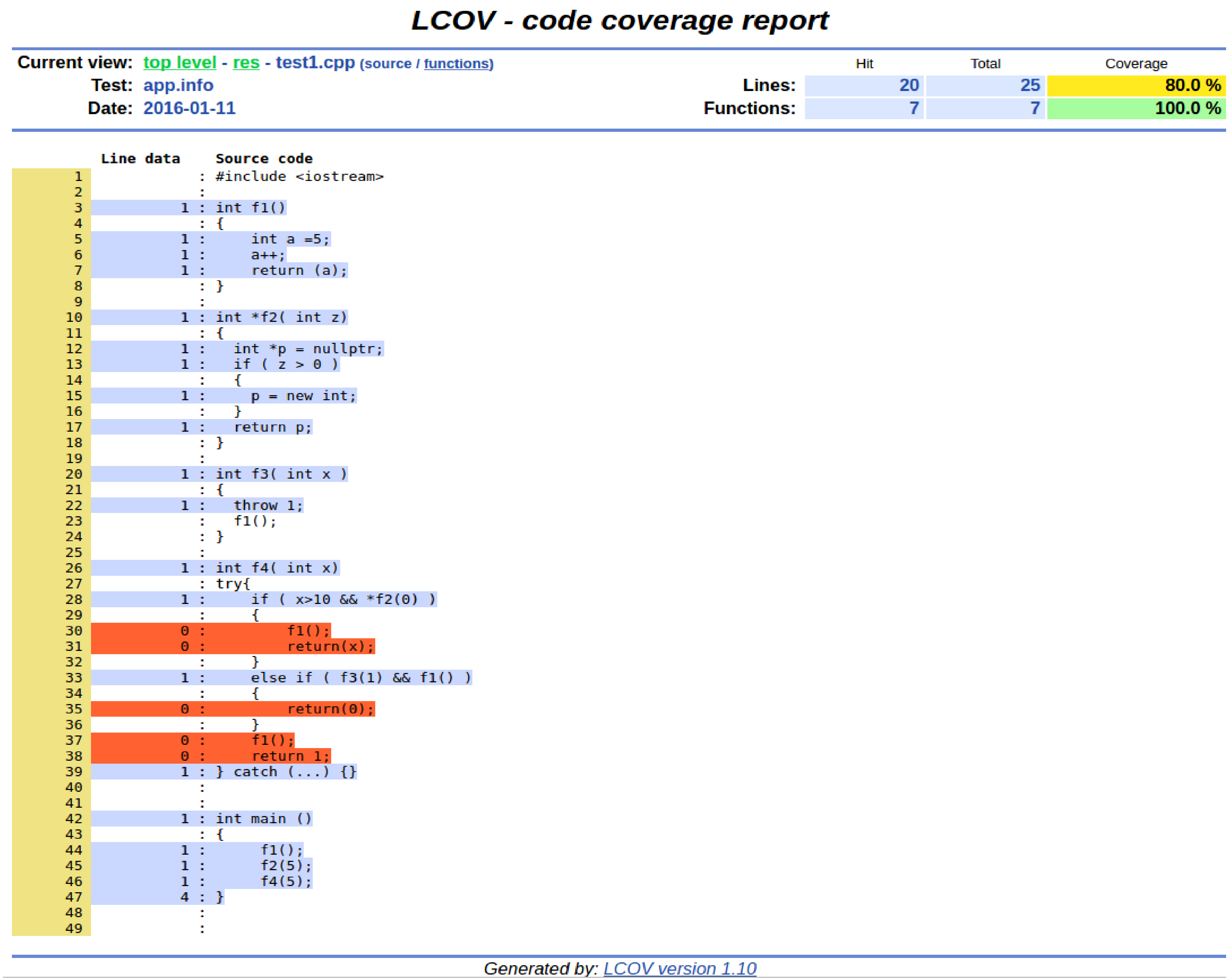
LCOV - code coverage report

Current view:	top level - res			Hit	Total	Coverage
Test:	app.info		**Lines:**	20	25	80.0 %
Date:	2016-01-11		**Functions:**	7	7	100.0 %

Filename	Line Coverage ⬍			Functions ⬍	
test1.cpp		80.0 %	20 / 25	100.0 %	7 / 7

Generated by: LCOV version 1.10

HTML output



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - res - test1.cpp (source / functions)				
Test:	app.info	**Lines:**	20	25	80.0 %
Date:	2016-01-11	**Functions:**	7	7	100.0 %

```
          Line data    Source code
  1                  : #include <iostream>
  2                  :
  3                1 : int f1()
  4                  : {
  5                1 :     int a =5;
  6                1 :     a++;
  7                1 :     return (a);
  8                  : }
  9                  :
 10                1 : int *f2( int z)
 11                  : {
 12                1 :   int *p = nullptr;
 13                1 :   if ( z > 0 )
 14                  :   {
 15                1 :     p = new int;
 16                  :   }
 17                1 :   return p;
 18                  : }
 19                  :
 20                1 : int f3( int x )
 21                  : {
 22                1 :   throw 1;
 23                  :   f1();
 24                  : }
 25                  :
 26                1 : int f4( int x)
 27                  : try{
 28                1 :     if ( x>10 && *f2(0) )
 29                  :     {
 30                0 :         f1();
 31                0 :         return(x);
 32                  :     }
 33                1 :     else if ( f3(1) && f1() )
 34                  :     {
 35                0 :         return(0);
 36                  :     }
 37                0 :     f1();
 38                0 :     return 1;
 39                1 : } catch (...) {}
 40                  :
 41                  :
 42                1 : int main ()
 43                  : {
 44                1 :      f1();
 45                1 :      f2(5);
 46                1 :      f4(5);
 47                4 : }
 48                  :
 49                  :
```

Generated by: LCOV version 1.10

Problems with Gcov

- Gcov is thread-safe, e.g. works correctly with multi-threaded applications. But uses lock and storing counters for visited lines in close-proximity, therefore execution of multithreaded applications is _very_ slow.To measure test coverage
- Statement level coverage can not extract expressions. When there is a shortcut operator, we do not know whether the right hand side evaluated.
- For an expression level coverage, see: MooCov by Gabor Kozar:

  https://github.com/shdnx/MooCoverage

Thank you!

gsd@elte.hu
http://gsd.web.elte.hu