What makes a good unit test?

Step 0

Write Tests!

Properties

- Correctness
- Readability
- Completeness
- Demonstrability
- Resilience

Correctness

Tests must verify the requirements of the system are met.

You shouldn’t:

- write tests that depend on known bugs

Correctness

```
int square(int x) {
// TODO(student): Implement
return 0;
}

TEST(SquareTest, MathTests) {
  REQUIRE(0 == square(2));
  REQUIRE(0 == square(3));
  REQUIRE(0 == square(7));
}
```

Correctness

```
int square(int x) {
// TODO(student): Implement
return 0;
}

TEST(SquareTest, MathTests) {
  REQUIRE(4 == square(2));
  REQUIRE(9 == square(3));
  REQUIRE(49 == square(7));
}
```

Correctness

Tests must verify the requirements of the system are met.

You shouldn’t write:

- tests that depend on known bugs
- tests that don’t actually execute real scenarios

Correctness

```
class MockWorld : public World {
  // For simplicity, we assume the world is flat
  bool IsFlat() override { return true; }
};
TEST_CASE("WorldTests", "[Flat]") {
  MockWorld world;
  REQUIRE(world.Populate());
  REQUIRE(world.IsFlat());
}
```

Readability

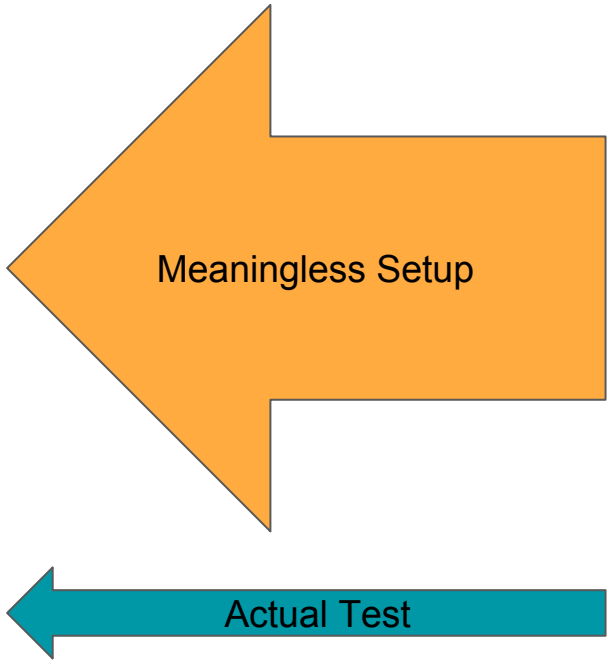
Write tests that are easy to understand, and clearly correct.
Tests should be obvious to the future reader (including yourself!).

Avoid tests having:

- too much boilerplate and distraction

Readability

```
TEST_CASE("CallIsUnimplemented", "[BigSystemTest]") {
  TestStorageSystem storage;
  auto test_data = GetTestFileMap();
  storage.MapFilesystem(test_data);
  BigSystem system;
  REQUIRE(system.Initialize(5));
  ThreadPool pool(10);
  pool.StartThreads();
  storage.SetThreads(pool);
  system.SetStorage(storage);
  REQUIRE(system.IsRunning());
  CHECK(IsUnimplemented (system.Status() ));
}
```



Readability

Write tests that are easy to understand, and clearly correct.
Tests should be obvious to the future reader (including yourself!).

Avoid tests having:

- too much boilerplate and distraction
- not enough context in the test

Readbility

Keep enough context for the reader

```
TEST_CASE("ReadMagicBytes", "[BigSystemTest]") {
  BigSystem system = InitializeTestSystemAndTestData ();
  REQUIRE(42 == system.PrivateKey());
}
```

Readbility

Write tests that are easy to understand, and clearly correct.
Tests should be obvious to the future reader (including yourself!).

Avoid tests having:

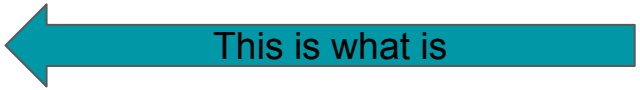
- too much boilerplate and distraction
- not enough context in the test
- superfluous use of advanced test framework features

Readability

Don’t use advanced test framework features when it isn’t necessary.

```
class BigSystemTest : public ::testing::Test {
 public:
  BigSystemTest() : filename_("/foo/bar/baz") { }
  void SetUp() {
    REQUIRE(file::WriteData(filename_, "Hello World!\n"));
  }
 protected:
  BigSystem system_;
  string filename_;
};

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



Readability

Write tests that are easy to understand, and clearly correct.
Tests should be obvious to the future reader (including yourself!).

Avoid tests having:

- too much boilerplate and distraction
- not enough context in the test
- superfluous use of advanced test framework features

A test should be like a novel: setup, action, conclusion, and it should all make sense.

Completeness

Test edge cases to demonstrate, that your code is correct.

Don’t just write:

```
TEST_CASE("FactorialTest", "[BasicTests]") {
  REQUIRE(1, Factorial(1));
  REQUIRE(120, Factorial(5));
}

int Factorial(int n) {
  if (n == 1) return 1;
  if (n == 5) return 120;
  return -1; // TODO(student): figure this out.
}
```

Completeness

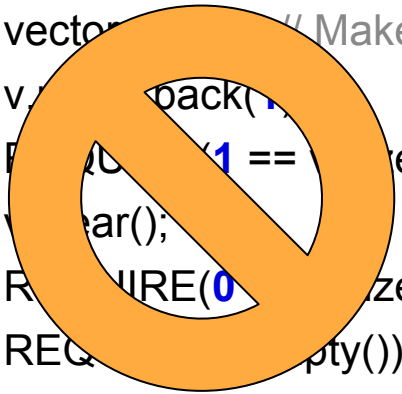
You should write tests for common inputs, corner cases and outlandish cases.

```
TEST_CASE(“FactorialTest”, “[BasicTests]”) {
  REQUIRE(1 == Factorial(1));
  REQUIRE(120 == Factorial(5));
  REQUIRE(1 == Factorial(0));
  REQUIRE(479001600 == Factorial(12));
  REQUIRE(std::numeric_limits::max<int>() == Factorial(13));
  REQUIRE(1 == Factorial(0));
  REQUIRE(std::numeric_limits::max<int>() == Factorial(-10));
}
```

Completeness

Don’t write tests for APIs that aren’t yours (see resilience).

```
TEST_CASE(“FilterTest”, “[WithVector]”) {
vecto                // Make sure that vector is working.
v.                back(1);
R                 (1 == v.size());
v                 ear();
R                 IRE(0 == v.size());
REQ                       pty());
// Now test our filter.
v = Filter({1, 2, 3, 4, 5}, [](int x) { return x % 2 == 0; });
REQUIRE_THAT(v, Equals({2, 4}));
}
```



Only test your API while using that other API!

Demonstability

Don’t use private APIs in Tests that users couldn’t. Use your tests to show how your API should be used.

Don’t write tests with:

- reliance on private methods + friends / test-only methods
- bad usage in unit tests, suggesting a bad API

Demonstrability

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

TEST_CASE("FooTest", "[Setup]") {
  REQUIRE(ShortcutSetupForTesting());
}
```

REQUIRE(Setup());

Resiliance

You should aim for tests that depend only on published API guarantees. Don’t write tests that fail in all sorts of surprising ways.

- Flaky tests
- Brittle tests
- Tests that depend on execution ordering
- Mocks with deep dependence upon underlying APIs
- Non-hermetic tests

Resiliance

Don’t write flaky tests: Tests that can be re-run with the same build in the same state and flip from passing to failing (or timing out).

```
TEST_CASE("UpdaterTest", "[RunsFast]") {
  Updater updater;
  updater.UpdateAsync();
  SleepFor(Seconds(.5)); // Half a second should be *plenty*.
  REQUIRE(updater.Updated());
}
```

Resiliance

Don’t write brittle tests: Tests that can fail for changes unrelated to the code under test.

```
TEST_CASE("ContentsAreCorrect", "[Tags]") {
  TagSet tags = {5, 8, 10};
  // TODO(student): Figure out why these are ordered funny.
  REQUIRE_THAT(tags, Equals(8, 5, 10));
}
```

UnorderedEquals

Resiliance

Don’t write brittle tests: Tests that can fail for changes unrelated to the code under test.

```
TEST_CASE("MyTest", "[LogWasCalled]") {
  StartLogCapture();
  REQUIRE(Frobber::Start());
  REQUIRE_THAT(Logs(), Contains("file.cc:421: Opened file frobber.config" ));
}
```

Resiliance

Resiliance

Resiliance - Ordering

Don’t write tests that fail if they aren’t run all together or in a particular order.

```
static int i = 0;

TEST_CASE("First", "[Foo]") {
  REQUIRE(0 == i);
  ++i;
}

TEST_CASE("Second", "[Foo]") {
  REQUIRE(1 == i);
  ++i;
}
```

Resiliance - Nonhermeticity

Don’t write tests that fail if anyone else in the company runs the same test at the same time.

```
TEST_CASE("Foo", "[StorageTest]") {
  StorageServer* server = GetStorageServerHandle ();
  auto my_val = rand();
  server->Store("testkey", my_val);
  REQUIRE(my_val == server->Load("testkey"));
}
```

Resiliance - Deep Dependence

Don’t write mock tests that fail if anyone refactors those classes.

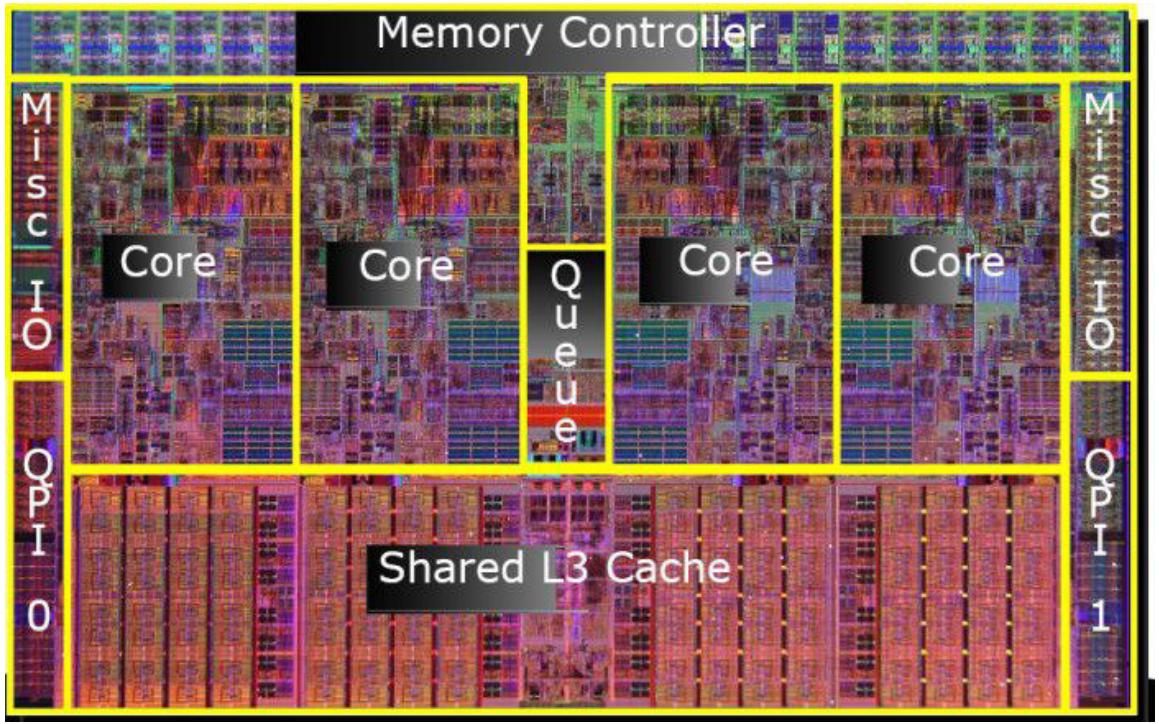
```
class File {
 public:
  ...
  virtual bool Stat(Stat* stat);
  virtual bool StatWithOptions(Stat* stat, StatOptions options) {
    return Stat(stat); // Ignore options by default
  }
};

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

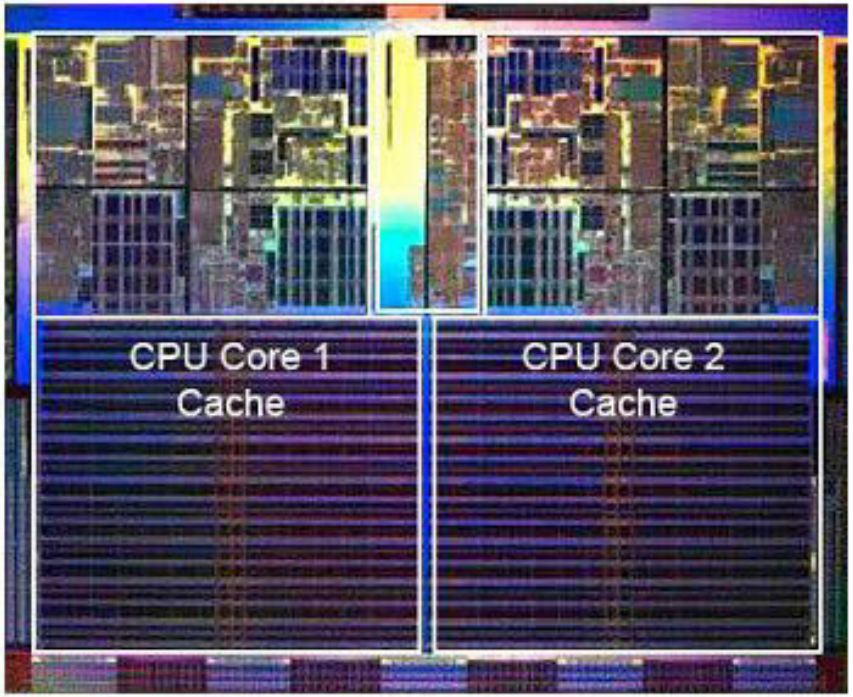
Recap: What’s the goal?

0. Write Tests!
1. Write tests that test what you wanted to test.
2. Write readable tests: correct by inspection.
3. Write complete tests: test all the edge cases.
4. Write demonstrative tests: show how to use the API.
5. Write resilient tests: hermetic, only breaks when there is an unacceptable behavior change.

Caches


Memory Controller
Misc IO
Core
Core
Queue
Core
Core
Misc IO
QPI 0
Shared L3 Cache
QPI 1

Caches



Cache Structure

- modern CPUs are much faster than main memory
  throughput: 8000 GB/s compute
  ~140 GB/s fetch
  latency: 1 ns compute
  ~100 ns cache miss
- orders of magnitude difference
- this resulted in multi-megabyte sized caches

Cache attached to Core