NOTE *A common gotcha getting Google Test set up in Visual Studio is that the C/C++ > Code Generation > Runtime Library option for Google Test must match your project's option. By default, Google Test compiles the runtime statically (that is, with the `/MT` or `/MTd` options). This choice is different from the default, which is to compile the runtime dynamically (for example, with the `/MD` or `/MDd` options in Visual Studio).*

Defining an Entry Point

Google Test will supply a `main()` function for you when you link `gtest_main` into your unit-test project. Think of this as Google Test's analogy for Catch's `#define CATCH_CONFIG_MAIN`; it will locate all the unit tests you've defined and roll them together into a nice test harness.

Defining Test Cases

To define test cases, all you need to do is provide unit tests using the `TEST` macro, which is quite similar to Catch's `TEST_CASE`. Listing 10-37 illustrates the basic setup of a Google Test unit test.

```
#include "gtest/gtest.h" ❶

TEST❷(AutoBrake❸, UnitTestName❹) {
  // Unit test here ❺
}
```

```
Running main() from gtest_main.cc ❻
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from AutoBrake
[ RUN      ] AutoBrake.UnitTestName
[       OK ] AutoBrake.UnitTestName (0 ms)
[----------] 1 test from AutoBrake (0 ms total)

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

Listing 10-37: An example Google Test unit test

First, you include the `gtest/gtest.h` header ❶. This pulls in all the definitions you need to define your unit tests. Each unit test starts with the `TEST` macro ❷. You define each unit test with two labels: a *test case name*, which is `AutoBrake` ❸ and a *test name*, which is `UnitTestName` ❹. These are roughly analogous to the `TEST_CASE` and `SECTION` names (respectively) in Catch. A test case contains one or many tests. Usually, you place tests together that share some a common theme. The framework will group the tests together, which can be useful for some of the more advanced uses. Different test cases can have tests with the same name.

You would put the code for your unit test within the braces ❺. When you run the resulting unit-test binary, you can see that Google Test provides an entry point for you ❻. Because you provided no assertions (or code that could throw an exception), your unit tests pass with flying colors ❼.

Making Assertions

Assertions in Google Test are less magical than in Catch's `REQUIRE`. Although they're also macros, the Google Test assertions require a lot more work on the programmer's part. Where `REQUIRE` will parse the Boolean expression and determine whether you're testing for equality, a greater-than relationship, and so on, Google Test's assertions don't. You must pass in each component of the assertion separately.

There are many other options for formulating assertions in Google Test. Table 10-1 summarizes them.

Table 10-1: Google Test Assertions

Assertion	Verifies that . . .
`ASSERT_TRUE(`*`condition`*`)`	*`condition`* is true.
`ASSERT_FALSE(`*`condition`*`)`	*`condition`* is false.
`ASSERT_EQ(`*`val1`*`, `*`val2`*`)`	*`val1`* `==` *`val2`* is true.
`ASSERT_FLOAT_EQ(`*`val1`*`, `*`val2`*`)`	*`val1`* `-` *`val2`* is a rounding error (`float`).
`ASSERT_DOUBLE_EQ(`*`val1`*`, `*`val2`*`)`	*`val1`* `-` *`val2`* is a rounding error (`double`).
`ASSERT_NE(`*`val1`*`, `*`val2`*`)`	*`val1`* `!=` *`val2`* is true.
`ASSERT_LT(`*`val1`*`, `*`val2`*`)`	*`val1`* `<` *`val2`* is true.
`ASSERT_LE(`*`val1`*`, `*`val2`*`)`	*`val1`* `<=` *`val2`* is true.
`ASSERT_GT(`*`val1`*`, `*`val2`*`)`	*`val1`* `>` *`val2`* is true.
`ASSERT_GE(`*`val1`*`, `*`val2`*`)`	*`val1`* `>=` *`val2`* is true.
`ASSERT_STREQ(`*`str1`*`, `*`str2`*`)`	The two C-style strings *`str1`* and *`str2`* have the same content.
`ASSERT_STRNE(`*`str1`*`, `*`str2`*`)`	The two C-style strings *`str1`* and *`str2`* have different content.
`ASSERT_STRCASEEQ(`*`str1`*`, `*`str2`*`)`	The two C-style strings *`str1`* and *`str2`* have the same content, ignoring case.
`ASSERT_STRCASENE(`*`str1`*`, `*`str2`*`)`	The two C-style strings *`str1`* and *`str2`* have different content, ignoring case.
`ASSERT_THROW(`*`statement`*`, `*`ex_type`*`)`	The evaluating *`statement`* causes an exception of type *`ex_type`* to be thrown.
`ASSERT_ANY_THROW(`*`statement`*`)`	The evaluating *`statement`* causes an exception of any type to be thrown.
`ASSERT_NO_THROW(`*`statement`*`)`	The evaluating *`statement`* causes no exception to be thrown.
`ASSERT_HRESULT_SUCCEEDED(`*`statement`*`)`	The `HRESULT` returned by *`statement`* corresponds with a success (Win32 API only).
`ASSERT_HRESULT_FAILED(`*`statement`*`)`	The `HRESULT` returned by *`statement`* corresponds with a failure (Win32 API only).

Let's combine a unit-test definition with an assertion to see Google Test in action.

Refactoring the initial_car_speed_is_zero Test to Google Test

With the intentionally broken `AutoBrake` in Listing 10-32, you can run the following unit test to see what the test harness's failure messages look like. (Recall that you commented out the member initializer for `speed_mps`.) Listing 10-38 uses `ASSERT_FLOAT_EQ` to assert that the car's initial speed is zero.

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

struct IServiceBus {
  --snip--
};

struct MockServiceBus : IServiceBus {
  --snip--
};

struct AutoBrake {
  AutoBrake(IServiceBus& bus)
    : collision_threshold_s{ 5 }/*,
    speed_mps{} */ {
  --snip--
};

TEST❶(AutoBrakeTest❷, InitialCarSpeedIsZero❸) {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  ASSERT_FLOAT_EQ❹(0❺, auto_brake.get_speed_mps()❻);
}
```

```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from AutoBrakeTest
[ RUN      ] AutoBrakeTest.InitialCarSpeedIsZero
C:\Users\josh\AutoBrake\gtest.cpp(80): error: Expected equality of these
values:
  0 ❺
  auto_brake.get_speed_mps()❻
    Which is: -inf
[  FAILED  ] AutoBrakeTest❷.InitialCarSpeedIsZero❸ (5 ms)
[----------] 1 test from AutoBrakeTest (5 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (7 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] AutoBrakeTest.InitialCarSpeedIsZero

 1 FAILED TEST
```

Listing 10-38: Intentionally commenting out the `collision_threshold_s` member initializer to cause test failures (using Google Test)

You declare a unit test ❶ with the test case name `AutoBrakeTest` ❷ and test name `InitialCarSpeedIsZero` ❸. Within the test, you set up the `auto_brake` and assert ❹ that the car's initial speed is zero ❺. Notice that the constant value is the first parameter and the quantity you're testing is the second parameter ❻.

Like the Catch output in Listing 10-33, the Google Test output in Listing 10-38 is very clear. It tells you that a test failed, identifies the failed assertion, and gives a good indication of how you might fix the issue.

Test Fixtures

Unlike Catch's `TEST_CASE` and `SECTION` approach, Google Test's approach is to formulate *test fixture classes* when a common setup is involved. These fixtures are classes that inherit from the `::testing::Test` class that the framework provides.

Any members you plan to use inside tests you should mark as `public` or `protected`. If you want some setup or teardown computation, you can put it inside the (default) constructor or destructor (respectively).

NOTE *You can also place such setup and teardown logic in overridden `SetUp()` and `TearDown()` functions, although it's rare that you would need to. One case is if the teardown computation might throw an exception. Because you generally shouldn't allow an uncaught exception to throw from a destructor, you would have to put such code in a `TearDown()` function. (Recall from "Throwing in Destructors" on page 106 that throwing an uncaught exception in a destructor when another exception is already in flight calls `std::terminate`.)*

If a test fixture is like a Catch `TEST_CASE`, then `TEST_F` is like a Catch `SECTION`. Like `TEST`, `TEST_F` takes two parameters. The first *must* be the exact name of the test fixture class. The second is the name of the unit test. Listing 10-39 illustrates the basic usage of Google Test's test fixtures.

```
#include "gtest/gtest.h"

struct MyTestFixture❶ : ::testing::Test❷ { };

TEST_F(MyTestFixture❸, MyTestA❹) {
  // Test A here
}

TEST_F(MyTestFixture, MyTestB❺) {
  // Test B here
}
```

```
Running main() from gtest_main.cc
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from MyTestFixture
[ RUN      ] MyTestFixture.MyTestA
[       OK ] MyTestFixture.MyTestA (0 ms)
[ RUN      ] MyTestFixture.MyTestB
```

```
[       OK ] MyTestFixture.MyTestB (0 ms)
[----------] 2 tests from MyTestFixture (1 ms total)

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

Listing 10-39: The basic setup of Google Test's test fixtures

You declare a class `MyTestFixture` ❶ that inherits from the `::testing::Test` class that Google Test provides ❷. You use the class's name as the first parameter to the `TEST_F` macro ❸. The unit test then has access to any public or protected methods inside `MyTestFixture`, and you can use the constructor and destructor of `MyTestFixture` to perform any common test setup/teardown. The second argument is the name of the unit test ❹❺.

Next, let's look at how to use Google Test Fixtures to reimplement the `AutoBrake` unit tests.

Refactoring AutoBrake Unit Tests with Google Test

Listing 10-40 reimplements all the `AutoBrake` unit tests into Google Test's test-fixture framework.

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

struct IServiceBus {
  --snip--
};

struct MockServiceBus : IServiceBus {
  --snip--
};

struct AutoBrake {
  --snip--
};

struct AutoBrakeTest : ::testing::Test { ❶
  MockServiceBus bus{};
  AutoBrake auto_brake { bus };
};

TEST_F❷(AutoBrakeTest❸, InitialCarSpeedIsZero❹) {
  ASSERT_DOUBLE_EQ(0, auto_brake.get_speed_mps()); ❺
}

TEST_F(AutoBrakeTest, InitialSensitivityIsFive) {
  ASSERT_DOUBLE_EQ(5, auto_brake.get_collision_threshold_s());
}

TEST_F(AutoBrakeTest, SensitivityGreaterThanOne) {
  ASSERT_ANY_THROW(auto_brake.set_collision_threshold_s(0.5L)); ❻
}
```

```
TEST_F(AutoBrakeTest, SpeedIsSaved) {
  bus.speed_update_callback(SpeedUpdate{ 100L });
  ASSERT_EQ(100, auto_brake.get_speed_mps());
  bus.speed_update_callback(SpeedUpdate{ 50L });
  ASSERT_EQ(50, auto_brake.get_speed_mps());
  bus.speed_update_callback(SpeedUpdate{ 0L });
  ASSERT_DOUBLE_EQ(0, auto_brake.get_speed_mps());
}

TEST_F(AutoBrakeTest, NoAlertWhenNotImminent) {
  auto_brake.set_collision_threshold_s(2L);
  bus.speed_update_callback(SpeedUpdate{ 100L });
  bus.car_detected_callback(CarDetected{ 1000L, 50L });
  ASSERT_EQ(0, bus.commands_published);
}

TEST_F(AutoBrakeTest, AlertWhenImminent) {
  auto_brake.set_collision_threshold_s(10L);
  bus.speed_update_callback(SpeedUpdate{ 100L });
  bus.car_detected_callback(CarDetected{ 100L, 0L });
  ASSERT_EQ(1, bus.commands_published);
  ASSERT_DOUBLE_EQ(1L, bus.last_command.time_to_collision_s);
}
--------------------------------------------------------------------------------
Running main() from gtest_main.cc
[==========] Running 6 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 6 tests from AutoBrakeTest
[ RUN      ] AutoBrakeTest.InitialCarSpeedIsZero
[       OK ] AutoBrakeTest.InitialCarSpeedIsZero (0 ms)
[ RUN      ] AutoBrakeTest.InitialSensitivityIsFive
[       OK ] AutoBrakeTest.InitialSensitivityIsFive (0 ms)
[ RUN      ] AutoBrakeTest.SensitivityGreaterThanOne
[       OK ] AutoBrakeTest.SensitivityGreaterThanOne (1 ms)
[ RUN      ] AutoBrakeTest.SpeedIsSaved
[       OK ] AutoBrakeTest.SpeedIsSaved (0 ms)
[ RUN      ] AutoBrakeTest.NoAlertWhenNotImminent
[       OK ] AutoBrakeTest.NoAlertWhenNotImminent (1 ms)
[ RUN      ] AutoBrakeTest.AlertWhenImminent
[       OK ] AutoBrakeTest.AlertWhenImminent (0 ms)
[----------] 6 tests from AutoBrakeTest (3 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 1 test case ran. (4 ms total)
[  PASSED  ] 6 tests.
```

Listing 10-40: Using Google Test to implement the `AutoBrake` unit tests

First, you implement the test fixture `AutoBrakeTest` ❶. This class encapsulates the common setup code across all the unit tests: to construct a `MockServiceBus` and use it to construct an `AutoBrake`. Each of the unit tests is represented by a `TEST_F` macro ❷. These macros take two parameters: the test fixture, such as `AutoBrakeTest` ❸, and the name of the test, such as

`InitialCarSpeedIsZero` ❹. Within the body of the unit tests, you have the correct invocations for each of the assertions, such as `ASSERT_DOUBLE_EQ` ❺ and `ASSERT_ANY_THROW` ❻.

Comparing Google Test and Catch

As you've seen, several major differences exist between Google Test and Catch. The most striking initial impression should be your investment in installing Google Test and making it work correctly in your solution. Catch is on the opposite end of this spectrum: as a header-only library, it's trivial to make it work in your project.

Another major difference is the assertions. To a newcomer, `REQUIRE` is a lot simpler to use than the Google Test assertion style. To a seasoned user of another xUnit framework, Google Test might seem more natural. The failure messages are also a bit different. It's really up to you to determine which of these styles is more sensible.

Finally, there's performance. Theoretically, Google Test will compile more quickly than Catch because all of Catch must be compiled for each translation unit in your unit-test suite. This is the trade-off for header-only libraries; the setup investment you make when setting up Google Test pays you back later with faster compilation. This might or might not be perceptible depending on the size of your unit-test suite.

Boost Test

Boost Test is a unit-testing framework that ships as part of the *Boost C++ libraries* (or simply *Boost*). Boost is an excellent collection of open source C++ libraries. It has a history of incubating many ideas that are eventually incorporated into the C++ standard, although not all Boost libraries aim for eventual inclusion. You'll see mention of a number of Boost libraries throughout the remainder of this book, and Boost Test is the first. For help installing boost into your environment, see Boost's home page *https://www.boost.org* or have a look at this book's companion code.

NOTE *At press time, the latest version of the Boost libraries is 1.70.0.*

You can use Boost Test in three modes: as a header-only library (like Catch), as a static library (like Google Test), or as a shared library, which will link the Boost Test module at runtime. The dynamic library usage can save quite a bit of disk space in the event you have multiple unit-test binaries. Rather than baking the unit-test framework into each of the unit-test binaries, you can build a single shared library (like a *.so* or *.dll*) and load it at runtime.

As you've discovered while exploring Catch and Google Test, trade-offs are involved with each of these approaches. A major advantage of Boost Test is that it allows you to choose the best mode as you see fit. It's not terribly difficult to switch modes should a project evolve, so one possible approach is to begin using Boost Test as a header-only library and transition into another mode as requirements change.

Setting Up Boost Test

To set up Boost Test in the header-only mode (what Boost documentation calls the "single-header variant"), you simply include the `<boost/test/included/unit_test.hpp>` header. For this header to compile, you need to define `BOOST_TEST_MODULE` with a user-defined name. For example:

```
#define BOOST_TEST_MODULE test_module_name
#include <boost/test/included/unit_test.hpp>
```

Unfortunately, you cannot take this approach if you have more than one translation unit. For such situations, Boost Test contains prebuilt static libraries that you can use. By linking these in, you avoid having to compile the same code for every translation unit. When taking this approach, you include the `boost/test/unit_test.hpp` header for each translation unit in the unit-test suite:

```
#include <boost/test/unit_test.hpp>
```

In exactly *one* translation unit, you also include the `BOOST_TEST_MODULE` definition:

```
#define BOOST_TEST_MODULE AutoBrake
#include <boost/test/unit_test.hpp>
```

You must also configure the linker to include the appropriate Boost Test static library that comes with the Boost Test installation. The compiler and architecture corresponding to the selected static library must match the rest of your unit-test project.

Setting Up Shared Library Mode

To set up Boost Test in shared library mode, you must add the following lines to each translation unit of the unit-test suite:

```
#define BOOST_TEST_DYN_LINK
#include <boost/test/unit_test.hpp>
```

In exactly *one* translation unit, you must also define `BOOST_TEST_MODULE`:

```
#define BOOST_TEST_MODULE AutoBrake
#define BOOST_TEST_DYN_LINK
#include <boost/test/unit_test.hpp>
```

As with the static library usage, you must instruct the linker to include Boost Test. At runtime, the unit-test shared library must be available as well.

Defining Test Cases

You can define a unit test in Boost Test with the `BOOST_AUTO_TEST_CASE` macro, which takes a single parameter corresponding to the name of the test. Listing 10-41 shows the basic usage.

```
#define BOOST_TEST_MODULE TestModuleName ❶
#include <boost/test/unit_test.hpp> ❷

BOOST_AUTO_TEST_CASE❸(TestA❹) {
  // Unit Test A here ❺
}
--------------------------------------------------------------------------------
Running 1 test case...

*** No errors detected
```

Listing 10-41: Using Google Test to implement the `AutoBrake` unit tests

The test module's name is `TestModuleName` ❶, which you define as the `BOOST_TEST_MODULE`. You include the `boost/test/unit_test.hpp` header ❷, which provides you with access to all the components you need from Boost Test. The `BOOST_AUTO_TEST_CASE` declaration ❸ denotes a unit test called `TestA` ❹. The body of the unit test goes between the braces ❺.

Making Assertions

Assertions in Boost are very similar to the assertions in Catch. The `BOOST_TEST` macro is like the `REQUIRE` macro in Catch. You simply provide an expression that evaluates to true if the assertion passes and false if it fails:

```
BOOST_TEST(assertion-expression)
```

To require an expression to throw an exception upon evaluation, use the `BOOST_REQUIRE_THROW` macro, which is similar to Catch's `REQUIRE_THROWS` macro, except you must also provide the type of the exception you want thrown. Its usage is as follows:

```
BOOST_REQUIRE_THROW(expression, desired-exception-type);
```

If the *`expression`* doesn't throw an exception of type *`desired-exception-type`*, the assertion will fail.

Let's examine what the `AutoBrake` unit-test suite looks like using Boost Test.

Refactoring the initial_car_speed_is_zero Test to Boost Test

You'll use the intentionally broken `AutoBrake` in Listing 10-32 with the missing member initializer for `speed_mps`. Listing 10-42 causes Boost Test to deal with a failed unit test.

```
#define BOOST_TEST_MODULE AutoBrakeTest ❶
#include <boost/test/unit_test.hpp>
#include <functional>

struct IServiceBus {
  --snip--
};

struct MockServiceBus : IServiceBus {
  --snip--
};

struct AutoBrake {
  AutoBrake(IServiceBus& bus)
    : collision_threshold_s{ 5 }/*,
      speed_mps{} */❷ {
  --snip--
};

BOOST_AUTO_TEST_CASE(InitialCarSpeedIsZero❸) {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  BOOST_TEST(0 == auto_brake.get_speed_mps()); ❹
}
```

```
Running 1 test case...
C:/Users/josh/projects/cpp-book/manuscript/part_2/10-testing/samples/boost/
minimal.cpp(80): error: in "InitialCarSpeedIsZero": check 0 == auto_brake.
get_speed_mps() has failed [0 != -9.2559631349317831e+61] ❺
*** 1 failure is detected in the test module "AutoBrakeTest"
```

Listing 10-42: Intentionally commenting out the speed_mps member initializer to cause test failures (using Boost Test)

The test module name is `AutoBrakeTest` ❶. After commenting out the `speed_mps` member initializer ❷, you have the `InitialCarSpeedIsZero` test ❸. The `BOOST_TEST` assertion tests whether `speed_mps` is zero ❹. As with Catch and Google Test, you have an informative error message that tells you what went wrong ❺.

Test Fixtures

Like Google Test, Boost Test deals with common setup code using the notion of test fixtures. Using them is as simple as declaring an RAII object where the setup logic for the test is contained in that class's constructor and the teardown logic is contained in the destructor. Unlike Google Test, you don't have to derive from a parent class in your test fixture. The test fixtures work with any user-defined structure.

To use the test fixture in a unit test, you employ the `BOOST_FIXTURE_TEST_CASE` macro, which takes two parameters. The first parameter is the name of the unit test, and the second parameter is the test fixture class. Within

the body of the macro, you implement a unit test as if it were a method of the test fixture class, as demonstrated in Listing 10-43.

```
#define BOOST_TEST_MODULE TestModuleName
#include <boost/test/unit_test.hpp>

struct MyTestFixture { }; ❶

BOOST_FIXTURE_TEST_CASE❷(MyTestA❸, MyTestFixture) {
  // Test A here
}

BOOST_FIXTURE_TEST_CASE(MyTestB❹, MyTestFixture) {
  // Test B here
}
```

```
Running 2 test cases...

*** No errors detected
```

Listing 10-43: Illustrating Boost test fixture usage

Here, you define a class called `MyTestFixture` ❶ and use it as the second parameter for each instance of `BOOST_FIXTURE_TEST_CASE` ❷. You declare two unit tests: `MyTestA` ❸ and `MyTestB` ❹. Any setup you perform within `MyTestFixture` affects each `BOOST_FIXTURE_TEST_CASE`.

Next, you'll use Boost Test fixtures to reimplement the `AutoBrake` test suite.

Refactoring AutoBrake Unit Tests with Boost Test

Listing 10-44 implements the `AutoBrake` unit-test suite using Boost Test's test fixture.

```
#define BOOST_TEST_MODULE AutoBrakeTest
#include <boost/test/unit_test.hpp>
#include <functional>

struct IServiceBus {
  --snip--
};

struct MockServiceBus : IServiceBus {
  --snip--
};

struct AutoBrakeTest { ❶
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
};

BOOST_FIXTURE_TEST_CASE❷(InitialCarSpeedIsZero, AutoBrakeTest) {
  BOOST_TEST(0 == auto_brake.get_speed_mps());
}
```

```
BOOST_FIXTURE_TEST_CASE(InitialSensitivityIsFive, AutoBrakeTest) {
  BOOST_TEST(5 == auto_brake.get_collision_threshold_s());
}

BOOST_FIXTURE_TEST_CASE(SensitivityGreaterThanOne, AutoBrakeTest) {
  BOOST_REQUIRE_THROW(auto_brake.set_collision_threshold_s(0.5L),
                      std::exception);
}

BOOST_FIXTURE_TEST_CASE(SpeedIsSaved, AutoBrakeTest) {
  bus.speed_update_callback(SpeedUpdate{ 100L });
  BOOST_TEST(100 == auto_brake.get_speed_mps());
  bus.speed_update_callback(SpeedUpdate{ 50L });
  BOOST_TEST(50 == auto_brake.get_speed_mps());
  bus.speed_update_callback(SpeedUpdate{ 0L });
  BOOST_TEST(0 == auto_brake.get_speed_mps());
}

BOOST_FIXTURE_TEST_CASE(NoAlertWhenNotImminent, AutoBrakeTest) {
  auto_brake.set_collision_threshold_s(2L);
  bus.speed_update_callback(SpeedUpdate{ 100L });
  bus.car_detected_callback(CarDetected{ 1000L, 50L });
  BOOST_TEST(0 == bus.commands_published);
}

BOOST_FIXTURE_TEST_CASE(AlertWhenImminent, AutoBrakeTest) {
  auto_brake.set_collision_threshold_s(10L);
  bus.speed_update_callback(SpeedUpdate{ 100L });
  bus.car_detected_callback(CarDetected{ 100L, 0L });
  BOOST_TEST(1 == bus.commands_published);
  BOOST_TEST(1L == bus.last_command.time_to_collision_s);
}
--------------------------------------------------------------------------------
Running 6 test cases...

*** No errors detected
```

Listing 10-44: Using Boost Test to implement your unit tests

You define the test fixture class `AutoBrakeTest` to perform the setup of the `AutoBrake` and `MockServiceBus` ❶. It's identical to the Google Test test fixture except you didn't need to inherit from any framework-issued parent classes. You represent each unit test with a `BOOST_FIXTURE_TEST_CASE` macro ❷. The rest of the tests use the `BOOST_TEST` and `BOOST_REQUIRE_THROW` assertion macros; otherwise, the tests look very similar to Catch tests. Instead of `TEST_CASE` and `SECTION` elements, you have a test fixture class and `BOOST_FIXTURE_TEST_CASE`.

Summary: Testing Frameworks

Although three different unit-testing frameworks were presented in this section, dozens of high-quality options are available. None of them is universally superior. Most frameworks support the same basic set of features,

whereas some of the more advanced features will have heterogeneous support. Mainly, you should select a unit-testing framework based on the style that makes you comfortable and productive.

Mocking Frameworks

The unit-testing frameworks you just explored will work in a wide range of settings. It would be totally feasible to build integration tests, acceptance tests, unit tests, and even performance tests using Google Test, for example. The testing frameworks support a broad range of programming styles, and their creators have only modest opinions about how you must design your software to make them testable.

Mocking frameworks are a bit more opinionated than unit-testing frameworks. Depending on the mocking framework, you must follow certain design guidelines for how classes depend on each other. The `AutoBrake` class used a modern design pattern called *dependency injection*. The `AutoBrake` class depends on an `IServiceBus`, which you injected using the constructor of `AutoBrake`. You also made `IServiceBus` an interface. Other methods for achieving polymorphic behavior exist (like templates), and each involves trade-offs.

All the mocking frameworks discussed in this section work extremely well with dependency injection. To varying degrees, the mocking frameworks remove the need to define your own mocks. Recall that you implemented a `MockServiceBus` to allow you to unit test `AutoBrake`, as displayed in Listing 10-45.

```
struct MockServiceBus : IServiceBus {
  void publish(const BrakeCommand& cmd) override {
    commands_published++;
    last_command = cmd;
  };
  void subscribe(SpeedUpdateCallback callback) override {
    speed_update_callback = callback;
  };
  void subscribe(CarDetectedCallback callback) override {
    car_detected_callback = callback;
  };
  BrakeCommand last_command{};
  int commands_published{};
  SpeedUpdateCallback speed_update_callback{};
  CarDetectedCallback car_detected_callback{};
};
```

Listing 10-45: Your hand-rolled `MockServiceBus`

Each time you want to add a unit test involving some new kind of interaction with `IServiceBus`, you'll likely need to update your `MockServiceBus` class. This is tedious and error prone. Additionally, it's not clear that you can share this mock class with other teams: you've implemented a lot of your own logic in it that won't be very useful to, say, the tire-pressure-sensor team. Also, each test might have different requirements. Mocking frameworks enables you to define mock classes, often using macro or template

voodoo. Within each unit test, you can customize the mock specifically for that test. This would be extremely difficult to do with a single mock definition.

This decoupling of the mock's declaration from the mock's test-specific definition is extremely powerful for two reasons. First, you can define different kinds of behavior for each unit test. This allows you to, for example, simulate exceptional conditions for some unit tests but not for others. Second, it makes the unit tests far more specific. By placing the custom mock's behavior within a unit test rather than in a separate source file, it's much clearer to the developer what the test is trying to achieve.

The net effect of using a mocking framework is that it makes mocking much less problematic. When mocking is easy, it makes good unit testing (and TDD) possible. Without mocking, unit testing can be very difficult; tests can be slow, unreliable, and brittle due to slow or error-prone dependencies. It's generally preferable, for example, to use a mock database connection instead of a full-blown production instance while you're trying to use TDD to implement new features into a class.

This section provides a tour of two mocking frameworks, Google Mock and HippoMocks, and includes a brief mention of two others, FakeIt and Trompeloeil. For technical reasons having to do with a lack of compile time code generation, creating a mocking framework is much harder in C++ than in most other languages, especially those with type reflection, a language feature that allows code to programmatically reason about type information. Consequently, there are a lot of high-quality mocking frameworks, each with their own trade-offs resulting from the fundamental difficulties associated with mocking C++.

Google Mock

One of the most popular mocking frameworks is the Google C++ Mocking Framework (or Google Mock), which is included as part of Google Test. It's one of the oldest and most feature-rich mocking frameworks. If you've already installed Google Test, incorporating Google Mock is easy. First, make sure you include the `gmock` static library in your linker, as you did for `gtest` and `gtest_main`. Next, add `#include "gmock/gmock.h"`.

If you're using Google Test as your unit-testing framework, that's all the setup you'll need to do. Google Mock will work seamlessly with its sister library. If you're using another unit-testing framework, you'll need to provide the initialization code in the entry point of the binary, as shown in Listing 10-46.

```
#include "gmock/gmock.h"

int main(int argc, char** argv) {
  ::testing::GTEST_FLAG(throw_on_failure) = true; ❶
  ::testing::InitGoogleMock(&argc, argv); ❷
  // Unit test as usual, Google Mock is initialized
}
```

Listing 10-46: Adding Google Mock to a third-party unit-testing framework

The GTEST_FLAG throw_on_failure ❶ causes Google Mock to throw an exception when some mock-related assertion fails. The call to InitGoogleMock ❷ consumes the command line arguments to make any necessary customization (refer to the Google Mock documentation for more details).

Mocking an Interface

For each interface you need to mock, there is some unfortunate ceremony. You need to take each virtual function of the interface and transmute it into a macro. For non-const methods, you use MOCK_METHOD*, and for const methods, you use MOCK_CONST_METHOD*, replacing * with the number of parameters that the function takes. The first parameter of MOCK_METHOD is the name of the virtual function. The second parameter is the function prototype. For example, to make a mock IServiceBus, you would build the definition shown in Listing 10-47.

```
struct MockServiceBus : IServiceBus { ❶
  MOCK_METHOD1❷(publish❸, void(const BrakeCommand& cmd)❹);
  MOCK_METHOD1(subscribe, void(SpeedUpdateCallback callback));
  MOCK_METHOD1(subscribe, void(CarDetectedCallback callback));
};
```

Listing 10-47: A Google Mock MockServiceBus

The beginning of the definition of MockServiceBus is identical to the definition of any other IServiceBus implementation ❶. You then employ MOCK_METHOD three times ❷. The first parameter ❸ is the name of the virtual function, and the second parameter ❹ is the prototype of the function.

It's a bit tedious to have to generate these definitions on your own. There's no additional information in the MockServiceBus definition that isn't already available in the IServiceBus. For better or worse, this is one of the costs of using Google Mock. You can take the sting out of generating this boilerplate by using the gmock_gen.py tool included in the scripts/generator folder of the Google Mock distribution. You'll need Python 2 installed, and it's not guaranteed to work in all situations. See the Google Mock documentation for more information.

Now that you've defined a MockServiceBus, you can use it in your unit tests. Unlike the mock you defined on your own, you can configure a Google Mock specifically for each unit test. You have an incredible amount of flexibility in this configuration. The key to successful mock configuration is the use of appropriate expectations.

Expectations

An *expectation* is like an assertion for a mock object; it expresses the circumstances in which the mock expects to be called and what it should do in response. The "circumstances" are specified using objects called *matchers*. The "what it should do in response" part is called an *action*. The sections that follow will introduce each of these concepts.

Expectations are declared with the `EXPECT_CALL` macro. The first parameter to this macro is the mock object, and the second is the expected method call. This method call can optionally contain matchers for each parameter. These matchers help Google Mock decide whether a particular method invocation qualifies as an expected call. The format is as follows:

```
EXPECT_CALL(mock_object, method(matchers))
```

There are several ways to formulate assertions about expectations, and which you choose depends on how strict your requirements are for how the unit being tested interacts with the mock. Do you care whether your code calls mocked functions that you didn't expect? It really depends on the application. That's why there are three options: naggy, nice, and strict.

A *naggy mock* is the default. If a naggy mock's function is called and no `EXPECT_CALL` matches the call, Google Mock will print a warning about an "uninteresting call," but the test won't fail just because of the uninteresting call. You can just add an `EXPECT_CALL` into the test as a quick fix to suppress the uninteresting call warning, because the call then ceases to be unexpected.

In some situations, there might be too many uninteresting calls. In such cases, you should use a *nice mock*. The nice mock won't produce a warning about uninteresting calls.

If you're very concerned about any interaction with the mock that you haven't accounted for, you might use a *strict mock*. Strict mocks will fail the test if any call is made to the mock for which you don't have a corresponding `EXPECT_CALL`.

Each of these types of mocks is a class template. The way to instantiate these classes is straightforward, as outlined in Listing 10-48.

```
MockServiceBus naggy_mock❶;
::testing::NiceMock<MockServiceBus> nice_mock❷;
::testing::StrictMock<MockServiceBus> strict_mock❸;
```

Listing 10-48: Three different styles of Google Mock

Naggy mocks ❶ are the default. Every `::testing::NiceMock` ❷ and `::testing::StrictMock` ❸ takes a single template parameter, the class of the underlying mock. All three of these options are perfectly valid first parameters to an `EXPECT_CALL`.

As a general rule, you should use nice mocks. Using naggy and strict mocks can lead to very brittle tests. When you're using a strict mock, consider whether it's really necessary to be so restrictive about the way the unit under test collaborates with the mock.

The second parameter to `EXPECT_CALL` is the name of the method you expect to be called followed by the parameters you expect the method to be called with. Sometimes, this is easy. Other times, there are more complicated conditions you want to express for what invocations match and don't match. In such situations, you use matchers.

Matchers

When a mock's method takes arguments, you have broad discretion over whether an invocation matches the expectation. In simple cases, you can use literal values. If the mock method is invoked with exactly the specified literal value, the invocation matches the expectation; otherwise, it doesn't. On the other extreme, you can use Google Mock's `::testing::_` object, which tells Google Mock that *any* value matches.

Suppose, for example, that you want to invoke `publish`, and you don't care what the argument is. The `EXPECT_CALL` in Listing 10-49 would be appropriate.

```
--snip--
using ::testing::_; ❶

TEST(AutoBrakeTest, PublishIsCalled) {
  MockServiceBus bus;
  EXPECT_CALL(bus, publish(_❷));
  --snip--
}
```

Listing 10-49: Using the `::testing::_` matcher in an expectation

To make the unit test nice and tidy, you employ a `using` for `::testing::_` ❶. You use `_` to tell Google Mock that *any* invocation of `publish` with a single argument will match ❷.

A slightly more selective matcher is the class template `::testing::A`, which will match only if a method is invoked with a particular type of parameter. This type is expressed as the template parameter to `A`, so `A<MyType>` will match only a parameter of type `MyType`. In Listing 10-50, the modification to Listing 10-49 illustrates a more restrictive expectation that requires a `BrakeCommand` as the parameter to `publish`.

```
--snip--
using ::testing::A; ❶

TEST(AutoBrakeTest, PublishIsCalled) {
  MockServiceBus bus;
  EXPECT_CALL(bus, publish(A<BrakeCommand>❷));
  --snip--
}
```

Listing 10-50: Using the `::testing::A` matcher in an expectation

Again, you employ `using` ❶ and use `A<BrakeCommand>` to specify that only a `BrakeCommand` will match this expectation.

Another matcher, `::testing::Field`, allows you to inspect fields on arguments passed to the mock. The `Field` matcher takes two parameters: a pointer to the field you want to expect and then another matcher to express whether the pointed-to field meets the criteria. Suppose you want to be even more specific about the call to `publish` ❷: you want to specify that the `time_to_collision_s` is equal to 1 second. You can accomplish this task with the refactor of Listing 10-49 shown in Listing 10-51.

```
--snip--
using ::testing::Field; ❶
using ::testing::DoubleEq; ❷

TEST(AutoBrakeTest, PublishIsCalled) {
  MockServiceBus bus;
  EXPECT_CALL(bus, publish(Field(&BrakeCommand::time_to_collision_s❸,
                                 DoubleEq(1L)❹)));
  --snip--
}
```

Listing 10-51: Using the `Field` matcher in an expectation

You employ `using` for `Field` ❶ and `DoubleEq` ❷ to clean up the expectation code a bit. The `Field` matcher takes a pointer to the field you're interested in `time_to_collision_s` ❸ and the matcher that decides whether the field meets the criteria `DoubleEq` ❹.

Many other matchers are available, and they're summarized in Table 10-2. But refer to the Google Mock documentation for all the details about their usages.

Table 10-2: Google Mock Matchers

Matcher	Matches when argument is . . .
`_`	Any value of the correct type
`A<type>)()`	Of the given *type*
`An<type>)()`	Of the given *type*
`Ge(value)`	Greater than or equal to *value*
`Gt(value)`	Greater than *value*
`Le(value)`	Less than or equal to *value*
`Lt(value)`	Less than *value*
`Ne(value)`	Not equal to *value*
`IsNull()`	Null
`NotNull()`	Not null
`Ref(variable)`	A reference to *variable*
`DoubleEq(variable)`	A `double` value approximately equal to *variable*
`FloatEq(variable)`	A `float` value approximately equal to *variable*
`EndsWith(str)`	A string ending with *str*
`HasSubstr(str)`	A string containing the substring *str*
`StartsWith(str)`	A string starting with *str*
`StrCaseEq(str)`	A string equal to *str* (ignoring case)
`StrCaseNe(str)`	A string not equal to *str* (ignoring case)
`StrEq(str)`	A string equal to *str*
`StrNeq(string)`	A string not equal to *str*

NOTE *One beneficial feature of matchers is that you can use them as an alternate kind of assertion for your unit tests. The alternate macro is one of* `EXPECT_THAT(value, matcher)` *or* `ASSERT_THAT(value, matcher)`*. For example, you could replace the assertion*

```
ASSERT_GT(power_level, 9000);
```

with the more syntactically pleasing

```
ASSERT_THAT(power_level, Gt(9000));
```

You can use `EXPECT_CALL` with `StrictMock` to enforce how the unit under test interacts with the mock. But you might also want to specify how many times the mock should respond to calls. This is called the expectation's *cardinality*.

Cardinality

Perhaps the most common method for specifying cardinality is `Times`, which specifies the number of times that a mock should expect to be called. The `Times` method takes a single parameter, which can be an integer literal or one of the functions listed in Table 10-3.

Table 10-3: A Listing of the Cardinality Specifiers in Google Mock

Cardinality	Specifies that a method will be called . . .
`AnyNumber()`	Any number of times
`AtLeast(n)`	At least *n* times
`AtMost(n)`	At most *n* times
`Between(m, n)`	Between *m* and *n* times
`Exactly(n)`	Exactly *n* times

Listing 10-52 elaborates Listing 10-51 to indicate that `publish` must be called only once.

```
--snip--
using ::testing::Field;
using ::testing::DoubleEq;

TEST(AutoBrakeTest, PublishIsCalled) {
  MockServiceBus bus;
  EXPECT_CALL(bus, publish(Field(&BrakeCommand::time_to_collision_s,
                                 DoubleEq(1L)))).Times(1)❶;
  --snip--
}
```

Listing 10-52: Using the `Times` *cardinality specifier in an expectation*

The `Times` call ❶ ensures that `publish` gets called exactly once (regardless of whether you use a nice, strict, or naggy mock).

NOTE *Equivalently, you could have specified* `Times(Exactly(1))`.

Now that you have some tools to specify the criteria and cardinality for an expected invocation, you can customize how the mock should respond to expectations. For this, you employ actions.

Actions

Like cardinalities, all actions are chained off `EXPECT_CALL` statements. These statements can help clarify how many times a mock expects to be called, what values to return each time it's called, and any side effects (like throwing an exception) it should perform. The `WillOnce` and `WillRepeatedly` actions specify what a mock should do in response to a query. These actions can get quite complicated, but for brevity's sake, this section covers two usages. First, you can use the `Return` construct to return values to the caller:

```
EXPECT_CALL(jenny_mock, get_your_number()) ❶
  .WillOnce(Return(8675309)) ❷
  .WillRepeatedly(Return(911))❸;
```

You set up an `EXPECT_CALL` the usual way and then tag on some actions that specify what value the `jenny_mock` will return each time `get_your_number` is called ❶. These are read sequentially from left to right, so the first action, `WillOnce` ❷, specifies that the first time `get_your_number` is called, the value `8675309` is returned by `jenny_mock`. The next action, `WillRepeatedly` ❸, specifies that for all subsequent calls, the value `911` will be returned.

Because `IServiceBus` doesn't return any values, you'll need the action to be a little more involved. For highly customizable behavior, you can use the `Invoke` construct, which enables you to pass an `Invocable` that will get called with the exact arguments passed into the mock's method. Let's say you want to save off a reference to the callback function that the `AutoBrake` registers via `subscribe`. You can do this easily with an `Invoke`, as illustrated in Listing 10-53.

```
CarDetectedCallback callback; ❶
EXPECT_CALL(bus, subscribe(A<CarDetectedCallback>()))
    .Times(1)
    .WillOnce(Invoke([&callback❷](const auto& callback_in❸) {
      callback = callback_in; ❹
    }));
```

Listing 10-53: Using `Invoke` to save off a reference to the `subscribe` callback registered by an `AutoBrake`

The first (and only) time that `subscribe` is called with a `CarDetectedCallback`, the `WillOnce(Invoke(...))` action will call the lambda that's been passed in as a parameter. This lambda captures the `CarDetectedCallback` declared ❶ by reference ❷. By definition, the lambda has the same function prototype as the `subscribe` function, so you can use auto-type deduction ❸ to determine the correct type for `callback_in` (it's `CarDetectedCallback`). Finally, you assign `callback_in` to `callback` ❹. Now, you can pass events off to whoever `subscribes`

simply by invoking your `callback` ❶. The `Invoke` construct is the Swiss Army Knife of actions, because you get to execute arbitrary code with full information about the invocation parameters. *Invocation parameters* are the parameters that the mocked method received at runtime.

Putting It All Together

Reconsidering our `AutoBrake` testing suite, you can reimplement the Google Test unit-test binary to use Google Mock rather than the hand-rolled mock, as demonstrated in Listing 10-54.

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

using ::testing::_;
using ::testing::A;
using ::testing::Field;
using ::testing::DoubleEq;
using ::testing::NiceMock;
using ::testing::StrictMock;
using ::testing::Invoke;

struct NiceAutoBrakeTest : ::testing::Test { ❶
  NiceMock<MockServiceBus> bus;
  AutoBrake auto_brake{ bus };
};

struct StrictAutoBrakeTest : ::testing::Test { ❷
  StrictAutoBrakeTest() {
    EXPECT_CALL(bus, subscribe(A<CarDetectedCallback>())) ❸
      .Times(1)
      .WillOnce(Invoke([this](const auto& x) {
        car_detected_callback = x;
      }));
    EXPECT_CALL(bus, subscribe(A<SpeedUpdateCallback>())) ❹
      .Times(1)
      .WillOnce(Invoke([this](const auto& x) {
        speed_update_callback = x;
      }));;
  }
  CarDetectedCallback car_detected_callback;
  SpeedUpdateCallback speed_update_callback;
  StrictMock<MockServiceBus> bus;
};

TEST_F(NiceAutoBrakeTest, InitialCarSpeedIsZero) {
  ASSERT_DOUBLE_EQ(0, auto_brake.get_speed_mps());
}

TEST_F(NiceAutoBrakeTest, InitialSensitivityIsFive) {
  ASSERT_DOUBLE_EQ(5, auto_brake.get_collision_threshold_s());
}
```

```
TEST_F(NiceAutoBrakeTest, SensitivityGreaterThanOne) {
  ASSERT_ANY_THROW(auto_brake.set_collision_threshold_s(0.5L));
}

TEST_F(StrictAutoBrakeTest, NoAlertWhenNotImminent) {
  AutoBrake auto_brake{ bus };

  auto_brake.set_collision_threshold_s(2L);
  speed_update_callback(SpeedUpdate{ 100L });
  car_detected_callback(CarDetected{ 1000L, 50L });
}

TEST_F(StrictAutoBrakeTest, AlertWhenImminent) {
  EXPECT_CALL(bus, publish(
                       Field(&BrakeCommand::time_to_collision_s, DoubleEq{ 1L
}))
                   ).Times(1);
  AutoBrake auto_brake{ bus };

  auto_brake.set_collision_threshold_s(10L);
  speed_update_callback(SpeedUpdate{ 100L });
  car_detected_callback(CarDetected{ 100L, 0L });
}
```

Listing 10-54: Reimplementing your unit tests using a Google Mock rather than a roll-your-own mock

Here, you actually have two different test fixtures: `NiceAutoBrakeTest` ❶ and `StrictAutoBrakeTest` ❷. The `NiceAutoBrakeTest` test instantiates a `NiceMock`. This is useful for `InitialCarSpeedIsZero`, `InitialSensitivityIsFive`, and `SensitivityGreaterThanOne`, because you don't want to test any meaningful interactions with the mock; it's not the focus of these tests. But you do want to focus on `AlertWhenImminent` and `NoAlertWhenNotImminent`. Each time an event is published or a type is subscribed to, it could have potentially major ramifications on your system. The paranoia of a `StrictMock` here is warranted.

In the `StrictAutoBrakeTest` definition, you can see the `WillOnce/Invoke` approach to saving off the callbacks for each subscription ❸❹. These are used in `AlertWhenImminent` and `NoAlertWhenNotImminent` to simulate events coming off the service bus. It gives the unit tests a nice, clean, succinct feel, even though there's a lot of mocking logic going on behind the scenes. Remember, you don't even require a working service bus to do all this testing!

HippoMocks

Google Mock is one of the original C++ mocking frameworks, and it's still a mainstream choice today. HippoMocks is an alternative mocking framework created by Peter Bindels. As a header-only library, HippoMocks is trivial

to install. Simply pull down the latest version from GitHub (*https://github.com/dascandy/hippomocks/*). You must include the `"hippomocks.h"` header in your tests. HippoMocks will work with any testing framework.

NOTE *At press time, the latest version of HippoMocks is v5.0.*

To create a mock using HippoMocks, you start by instantiating a `MockRespository` object. By default, all the mocks derived from this `MockRepository` will require *strict ordering* of expectations. Strictly ordered expectations cause a test to fail if each of the expectations is not invoked in the exact order you've specified. Usually, this is not what you want. To modify this default behavior, set the `autoExpect` field on `MockRepository` to `false`:

```
MockRepository mocks;
mocks.autoExpect = false;
```

Now you can use `MockRepository` to generate a mock of `IServiceBus`. This is done through the (member) function template `Mock`. This function will return a pointer to your newly minted mock:

```
auto* bus = mocks.Mock<IServiceBus>();
```

A major selling point of `HippoMocks` is illustrated here: notice that you didn't need to generate any macro-laden boilerplate for the mock `IServiceBus` like you did for Google Mock. The framework can handle vanilla interfaces without any further effort on your part.

Setting up expectations is very straightforward as well. For this, use the `ExpectCall` macro on `MockRespository`. The `ExpectCall` macro takes two parameters: a pointer to your mock and a pointer to the method you're expecting:

```
mocks.ExpectCall(bus, IServiceBus::subscribe_to_speed)
```

This example adds an expectation that `bus.subscribe_to_speed` will be invoked. You have several matchers you can add to this expectation, as summarized in Table 10-4.

Table 10-4: HippoMocks Matchers

Matcher	Specifies that an expectation matches when . . .
`With(`*`args`*`)`	The invocation parameters match *args*
`Match(`*`predicate`*`)`	*predicate* invoked with the invocation parameters returns `true`
`After(`*`expectation`*`)`	*expectation* has already been satisfied (This is useful for referring to a previously registered call.)

You can define actions to perform in response to `ExpectCall`, as summarized in Table 10-5.

Table 10-5: HippoMocks Actions

Action	Does the following upon invocation:
`Return(value)`	Returns *value* to the caller
`Throw(exception)`	Throws *exception*
`Do(callable)`	Executes *callable* with the invocation parameters

By default, HippoMocks requires an expectation to be met exactly once (like Google Mock's `.Times(1)` cardinality).

For example, you can express the expectation that `publish` is called with a `BrakeCommand` having a `time_to_collision_s` of `1.0` in the following way:

```
mocks.ExpectCall❶(bus, IServiceBus::publish)
  .Match❷([](const BrakeCommand& cmd) {
    return cmd.time_to_collision_s == Approx(1); ❸
  });
```

You use `ExpectCall` to specify that `bus` should be called with the `publish` method ❶. You refine this expectation with the `Match` matcher ❷, which takes a predicate accepting the same arguments as the `publish` method—a single `const BrakeCommand` reference. You return `true` if the `time_to_collision_s` field of the `BrakeCommand` is 1.0; otherwise, you return `false` ❸, which is fully compatible.

NOTE *As of v5.0, HippoMocks doesn't have built-in support for approximate matchers. Instead, Catch's* `Approx` ❸ *was used.*

HippoMocks supports function overloads for free functions. It also supports overloads for methods, but the syntax is not very pleasing to the eye. If you are using HippoMocks, it is best to avoid method overloads in your interface, so it would be better to refactor `IServiceBus` along the following lines:

```
struct IServiceBus {
  virtual ~IServiceBus() = default;
  virtual void publish(const BrakeCommand&) = 0;
  virtual void subscribe_to_speed(SpeedUpdateCallback) = 0;
  virtual void subscribe_to_car_detected(CarDetectedCallback) = 0;
};
```

NOTE *One design philosophy states that it's undesirable to have an overloaded method in an interface, so if you subscribe to that philosophy, the lack of support in HippoMocks is a moot point.*

Now `subscribe` is no longer overloaded, and it's possible to use HippoMocks. Listing 10-55 refactors the test suite to use HippoMocks with Catch.

```
#include "hippomocks.h"
--snip--
TEST_CASE("AutoBrake") {
  MockRepository mocks; ❶
  mocks.autoExpect = false;
  CarDetectedCallback car_detected_callback;
  SpeedUpdateCallback speed_update_callback;
  auto* bus = mocks.Mock<IServiceBus>();
  mocks.ExpectCall(bus, IServiceBus::subscribe_to_speed) ❷
    .Do([&](const auto& x) {
      speed_update_callback = x;
    });
  mocks.ExpectCall(bus, IServiceBus::subscribe_to_car_detected) ❸
    .Do([&](const auto& x) {
    car_detected_callback = x;
  });
  AutoBrake auto_brake{ *bus };

  SECTION("initializes speed to zero") {
    REQUIRE(auto_brake.get_speed_mps() == Approx(0));
  }

  SECTION("initializes sensitivity to five") {
    REQUIRE(auto_brake.get_collision_threshold_s() == Approx(5));
  }

  SECTION("throws when sensitivity less than one") {
    REQUIRE_THROWS(auto_brake.set_collision_threshold_s(0.5L));
  }

  SECTION("saves speed after update") {
    speed_update_callback(SpeedUpdate{ 100L }); ❹
    REQUIRE(100L == auto_brake.get_speed_mps());
    speed_update_callback(SpeedUpdate{ 50L });
    REQUIRE(50L == auto_brake.get_speed_mps());
    speed_update_callback(SpeedUpdate{ 0L });
    REQUIRE(0L == auto_brake.get_speed_mps());
  }

  SECTION("no alert when not imminent") {
    auto_brake.set_collision_threshold_s(2L);
    speed_update_callback(SpeedUpdate{ 100L }); ❺
    car_detected_callback(CarDetected{ 1000L, 50L });
  }

  SECTION("alert when imminent") {
    mocks.ExpectCall(bus, IServiceBus::publish) ❻
      .Match([](const auto& cmd) {
        return cmd.time_to_collision_s == Approx(1);
      });

    auto_brake.set_collision_threshold_s(10L);
    speed_update_callback(SpeedUpdate{ 100L });
```

```
    car_detected_callback(CarDetected{ 100L, 0L });
  }
}
```

Listing 10-55: Reimplementing Listing 10-54 to use HippoMocks and Catch rather than Google Mock and Google Test.

NOTE *This section couples HippoMocks with Catch for demonstration purposes, but HippoMocks works with all the unit-testing frameworks discussed in this chapter.*

You create the `MockRepository` ❶ and relax the strict ordering requirements by setting `autoExpect` to `false`. After declaring the two callbacks, you create an `IServiceBusMock` (without having to define a mock class!), and then set expectations ❷❸ that will hook up your callback functions with `AutoBrake`. Finally, you create `auto_brake` using a reference to the mock bus.

The `initializes speed to zero`, `initializes sensitivity to five`, and `throws when sensitivity less than one` tests require no further interaction with the mock. In fact, as a strict mock, `bus` won't let any further interactions happen without complaining. Because HippoMocks doesn't allow nice mocks like Google Mock, this is actually a fundamental difference between Listing 10-54 and Listing 10-55.

In the `saves speed after update` test ❹, you issue a series of `speed_update` callbacks and assert that the speeds are saved off correctly as before. Because `bus` is a strict mock, you're also implicitly asserting that no further interaction happens with the service bus here.

In the `no alert when not imminent` test, no changes are needed to `speed_update_callback` ❺. Because the mock is strict (and you don't expect a `BrakeCommand` to get published), no further expectations are needed.

NOTE *HippoMocks offers the `NeverCall` method on its mocks, which will improve the clarity of your tests and errors if it's called.*

However, in the `alert when imminent` test, you expect that your program will invoke `publish` on a `BrakeCommand`, so you set up this expectation ❻. You use the `Match` matcher to provide a predicate that checks for `time_to_collision_s` to equal approximately `1`. The rest of the test is as before: you send `AutoBrake` a `SpeedUpdate` event and a subsequent `CarDetected` event that should cause a collision to be detected.

HippoMocks is a more streamlined mocking framework than Google Mock is. It requires far less ceremony, but it's a little less flexible.

NOTE *One area where HippoMocks is* more *flexible than Google Mock is in mocking free functions. HippoMocks can mock free functions and static class functions directly, whereas Google Mock requires you to rewrite the code to use an interface.*

A Note on Other Mocking Options: FakeIt and Trompeloeil

A number of other excellent mocking frameworks are available. But for the sake of keeping an already long chapter from getting much longer, let's briefly look at two more frameworks: FakeIt (by Eran Pe'er, available at *https://github.com/eranpeer/FakeIt/*) and Trompeloeil (by Björn Fahller, available at *https://github.com/rollbear/trompeloeil/*).

FakeIt is similar to HippoMocks in its succinct usage patterns, and it's a header-only library. It differs in that it follows the record-by-default pattern in building expectations. Rather than specifying expectations up front, FakeIt verifies that a mock's methods were invoked correctly at the *end* of the test. Actions, of course, are still specified at the beginning.

Although this is a totally valid approach, I prefer the Google Mock/HippoMocks approach of specifying expectations—and their associated actions—all up front in one concise location.

Trompeloeil (from the French *trompe-l'œil* for "deceive the eye") can be considered a modern replacement for Google Mock. Like Google Mock, it requires some macro-laden boilerplate for each of the interfaces you want to mock. In exchange for this extra effort, you gain many powerful features, including actions, such as setting test variables, returning values based on invocation parameters, and forbidding particular invocations. Like Google Mock and HippoMocks, Trompeloeil requires you to specify your expectations and actions up front (see the documentation for more details).

Summary

This chapter used an extended example of building the automatic braking system for an autonomous vehicle to explore the basics of TDD. You rolled your own testing and mocking framework, then learned about the many benefits of using available testing and mocking frameworks. You toured Catch, Google Test, and Boost Test as possible testing frameworks. For mocking frameworks, you dove into Google Mock and HippoMocks (with a brief mention of FakeIt and Trompeloeil). Each of these frameworks has strengths and weaknesses. Which you choose should be driven principally by which frameworks make you most efficient and productive.

NOTE *For the remainder of the book, examples will be couched in terms of unit tests. Accordingly, I had to choose a framework for the examples. I've chosen Catch for a few reasons. First, Catch's syntax is the most succinct, and it lends itself well to book form. In header-only mode, Catch compiles much quicker than Boost Test. This might be considered an endorsement of the framework (and it is), but it's not my intention to discourage the use of Google Test, Boost Test, or any other testing framework. You should make such decisions after careful consideration (and hopefully some experimentation.)*

EXERCISES

10-1. Your car company has completed work on a service that detects speed limits based on signage it observes on the side of the road. The speed-limit-detection team will publish objects of the following type to the event bus periodically:

```
struct SpeedLimitDetected {
  unsigned short speed_mps;
}
```

The service bus has been extended to incorporate this new type:

```
#include <functional>
--snip--
using SpeedUpdateCallback = std::function<void(const SpeedUpdate&)>;
using CarDetectedCallback = std::function<void(const CarDetected&)>;
using SpeedLimitCallback = std::function<void(const SpeedLimitDetected&)>;

struct IServiceBus {
  virtual ~IServiceBus() = default;
  virtual void publish(const BrakeCommand&) = 0;
  virtual void subscribe(SpeedUpdateCallback) = 0;
  virtual void subscribe(CarDetectedCallback) = 0;
  virtual void subscribe(SpeedLimitCallback) = 0;
};
```

Update the service with the new interface and make sure the tests still pass.

10-2. Add a private field for the last known speed limit. Implement a getter method for this field.

10-3. The product owner wants you to initialize the last known speed limit to 39 meters per second. Implement a unit test that checks a newly constructed AutoBrake that has a last known speed limit of 39.

10-4. Make unit tests pass.

10-5. Implement a unit test where you publish three different `SpeedLimitDetected` objects using the same callback technique you used for `SpeedUpdate` and `CarDetected`. After invoking each of the callbacks, check the last known speed limit on the `AutoBrake` object to ensure it matches.

10-6. Make all unit tests pass.

10-7. Implement a unit test where the last known speed limit is 35 meters per second, and you're traveling at 34 meters per second. Ensure that no `BrakeCommand` is published by `AutoBrake`.

10-8. Make all unit tests pass.

10-9. Implement a unit test where the last known speed limit is 35 meters per second and then publish a `SpeedUpdate` at 40 meters per second. Ensure that exactly one `BrakeCommand` is issued. The `time_to_collision_s` field should equal 0.

10-10. Make all unit tests pass.

10-11. Implement a new unit test where the last known speed limit is 35 meters per second and then publish a `SpeedUpdate` at 30 meters per second. Then issue a `SpeedLimitDetected` with a `speed_mps` of 25 meters per second. Ensure that exactly one `BrakeCommand` is issued. The `time_to_collision_s` field should equal 0.

10-12. Make all unit tests pass.

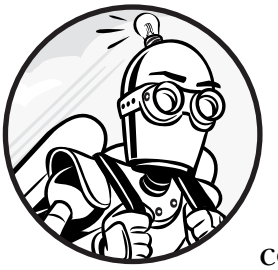
FURTHER READING

- *Specification by Example* by Gojko Adzic (Manning, 2011)
- *BDD in Action* by John Ferguson Smart (Manning, 2014)
- *Optimized C++: Proven Techniques for Heightened Performance* by Kurt Guntheroth (O'Reilly, 2016)
- *Agile Software Development and Agile Principles, Patterns, and Practices in C#* by Robert C. Martin (Prentice Hall, 2006)
- *Test-Driven Development: By Example* by Kent Beck (Pearson, 2002)
- *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce (Addison-Wesley, 2009)
- "Editor war." *https://en.wikipedia.org/wiki/Editor_war*
- "Tabs versus Spaces: An Eternal Holy War" by Jamie Zawinski. *https://www.jwz.org/doc/tabs-vs-spaces.html*
- "Is TDD dead?" by Martin Fowler. *https://martinfowler.com/articles/is-tdd-dead/*

11

SMART POINTERS

If you want to do a few small things right, do them yourself. If you want to do great things and make a big impact, learn to delegate.
—John C. Maxwell



In this chapter, you'll explore stdlib and Boost libraries. These libraries contain a collection of smart pointers, which manage dynamic objects with the RAII paradigm you learned in Chapter 4. They also facilitate the most powerful resource management model in any programming language. Because some smart pointers use *allocators* to customize dynamic memory allocation, the chapter also outlines how to provide a user-defined allocator.

Smart Pointers

Dynamic objects have the most flexible lifetimes. With great flexibility comes great responsibility, so you must make sure each dynamic object gets destructed *exactly* once. This might not look daunting with small programs, but looks can be deceiving. Just consider how exceptions factor

into dynamic memory management. Each time an error or an exception could occur, you need to keep track of which allocations you've made successfully and be sure to release them in the correct order.

Fortunately, you can use RAII to handle such tedium. By acquiring dynamic storage in the constructor of the RAII object and releasing dynamic storage in the destructor, it's relatively difficult to leak (or double free) dynamic memory. This enables you to manage dynamic object lifetimes using move and copy semantics.

You could write these RAII objects yourself, but you can also use some excellent prewritten implementations called *smart pointers*. Smart pointers are class templates that behave like pointers and implement RAII for dynamic objects.

This section delves into five available options included in stdlib and Boost: scoped, unique, shared, weak, and intrusive pointers. Their ownership models differentiate these five smart pointer categories.

Smart Pointer Ownership

Every smart pointer has an *ownership* model that specifies its relationship with a dynamically allocated object. When a smart pointer owns an object, the smart pointer's lifetime is guaranteed to be at least as long as the object's. Put another way, when you use a smart pointer, you can rest assured that the pointed-to object is alive and that the pointed-to object won't leak. The smart pointer manages the object it owns, so you can't forget to destroy it thanks to RAII.

When considering which smart pointer to use, your ownership requirements drive your choice.

Scoped Pointers

A *scoped pointer* expresses *non-transferable*, *exclusive ownership* over a single dynamic object. Non-transferable means that the scoped pointers cannot be moved from one scope to another. Exclusive ownership means that they can't be copied, so no other smart pointers can have ownership of a scoped pointer's dynamic object. (Recall from "Memory Management" on page 90 that an object's scope is where it's visible to the program.)

The `boost::scoped_ptr` is defined in the `<boost/smart_ptr/scoped_ptr.hpp>` header.

NOTE *There is no stdlib scoped pointer.*

Constructing

The `boost::scoped_ptr` takes a single template parameter corresponding to the pointed-to type, as in `boost::scoped_ptr<int>` for a "scoped pointer to `int`" type.

All smart pointers, including scoped pointers, have two modes: *empty* and *full*. An empty smart pointer owns no object and is roughly analogous to a `nullptr`. When a smart pointer is default constructed, it begins life empty.

The scoped pointer provides a constructor taking a raw pointer. (The pointed-to type must match the template parameter.) This creates a full-scoped pointer. The usual idiom is to create a dynamic object with `new` and pass the result to the constructor, like this:

```
boost::scoped_ptr<PointedToType> my_ptr{ new PointedToType };
```

This line dynamically allocates a `PointedToType` and passes its pointer to the scoped pointer constructor.

Bring in the Oath Breakers

To explore scoped pointers, let's create a Catch unit-test suite and a `DeadMenOfDunharrow` class that keeps track of how many objects are alive, as shown in Listing 11-1.

```
#define CATCH_CONFIG_MAIN ❶
#include "catch.hpp" ❷
#include <boost/smart_ptr/scoped_ptr.hpp> ❸

struct DeadMenOfDunharrow { ❹
  DeadMenOfDunharrow(const char* m="") ❺
    : message{ m } {
    oaths_to_fulfill++; ❻
  }
  ~DeadMenOfDunharrow() {
    oaths_to_fulfill--; ❼
  }
  const char* message;
  static int oaths_to_fulfill;
};
int DeadMenOfDunharrow::oaths_to_fulfill{};
using ScopedOathbreakers = boost::scoped_ptr<DeadMenOfDunharrow>; ❽
```

Listing 11-1: Setting up a Catch unit-test suite with a `DeadMenOfDunharrow` class to investigate scoped pointers

First, you declare `CATCH_CONFIG_MAIN` so Catch will provide an entry point ❶ and include the Catch header ❷ and then the Boost scoped pointer's header ❸. Next, you declare the `DeadMenOfDunharrow` class ❹, which takes an optional null-terminated string that you save into the `message` field ❺. The `static int` field called `oaths_to_fulfill` tracks how many `DeadMenOfDunharrow` objects have been constructed. Accordingly, you increment in the constructor ❻, and you decrement in the destructor ❼. Finally, you declare the `ScopedOathbreakers` type alias for convenience ❽.

CATCH LISTINGS

You'll use Catch unit tests in most listings from now on. For conciseness, the listings omit the following Catch ceremony:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

All listings containing `TEST_CASE` require this preamble.

Also, every test case in each listing passes unless a comment indicates otherwise. Again, for conciseness, the listings omit the `All tests pass` output from the listings.

Finally, tests that employ user-defined types, functions, and variables from a previous listing will omit them for brevity.

Implicit bool Conversion Based on Ownership

Sometimes you need to determine whether a scoped pointer owns an object or whether it's empty. Conveniently, `scoped_ptr` casts implicitly to `bool` depending on its ownership status: `true` if it owns an object; `false` otherwise. Listing 11-2 illustrates how this implicit casting behavior works.

```
TEST_CASE("ScopedPtr evaluates to") {
  SECTION("true when full") {
    ScopedOathbreakers aragorn{ new DeadMenOfDunharrow{} }; ❶
    REQUIRE(aragorn); ❷
  }
  SECTION("false when empty") {
    ScopedOathbreakers aragorn; ❸
    REQUIRE_FALSE(aragorn); ❹
  }
}
```

Listing 11-2: The `boost::scoped_ptr` casts implicitly to `bool`.

When you use the constructor taking a pointer ❶, the `scoped_ptr` converts to `true` ❷. When you use the default constructor ❸, the `scoped_ptr` converts to `false` ❹.

RAII Wrapper

When a `scoped_ptr` owns a dynamic object, it ensures proper dynamic object management. In the `scoped_ptr` destructor, it checks whether it owns an object. If it does, the `scoped_ptr` destructor deletes the dynamic object.

Listing 11-3 illustrates this behavior by investigating the static `oaths_to_fulfill` variable between scoped pointer initializations.

```
TEST_CASE("ScopedPtr is an RAII wrapper.") {
  REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 0); ❶
  ScopedOathbreakers aragorn{ new DeadMenOfDunharrow{} }; ❷
  REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❸
  {
    ScopedOathbreakers legolas{ new DeadMenOfDunharrow{} }; ❹
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 2); ❺
  } ❻
  REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❼
}
```

Listing 11-3: The `boost::scoped_ptr` is an RAII wrapper.

At the beginning of the test, `oaths_to_fulfill` is 0 because you haven't constructed any `DeadMenOfDunharrow` yet ❶. You construct the scoped pointer `aragorn` and pass in a pointer to the dynamic `DeadMenOfDunharrow` object ❷. This increments the `oaths_to_fulfill` to 1 ❸. Within a nested scope, you declare another scoped pointer `legolas` ❹. Because `aragorn` is still alive, `oaths_to_fulfill` is now 2 ❺. Once the inner scope closes, `legolas` falls out of scope and destructs, taking a `DeadMenOfDunharrow` with it ❻. This decrements `DeadMenOfDunharrow` to 1 ❼.

Pointer Semantics

For convenience, `scoped_ptr` implements the dereference `operator*` and the member dereference `operator->`, which simply delegate the calls to the owned dynamic object. You can even extract a raw pointer from a `scoped_ptr` with the `get` method, as demonstrated in Listing 11-4.

```
TEST_CASE("ScopedPtr supports pointer semantics, like") {
  auto message = "The way is shut";
  ScopedOathbreakers aragorn{ new DeadMenOfDunharrow{ message } }; ❶
  SECTION("operator*") {
    REQUIRE((*aragorn).message == message); ❷
  }
  SECTION("operator->") {
    REQUIRE(aragorn->message == message); ❸
  }
  SECTION("get(), which returns a raw pointer") {
    REQUIRE(aragorn.get() != nullptr); ❹
  }
}
```

Listing 11-4: The `boost::scoped_ptr` supports pointer semantics.

You construct the scoped pointer `aragorn` with a `message` of `The way is shut` ❶, which you use in three separate scenarios to test pointer semantics. First, you can use `operator*` to dereference the underlying, pointed-to dynamic object. In the example, you dereference `aragorn` and extract the `message` to verify that it matches ❷. You can also use `operator->` to perform member dereference ❸. Finally, if you want a raw pointer to the dynamic object, you can use the `get` method to extract it ❹.

Comparison with nullptr

The scoped_ptr class template implements the comparison operators operator== and operator!=, which are only defined when comparing a scoped_ptr with a nullptr. Functionally, this is essentially identical to implicit bool conversion, as Listing 11-5 illustrates.

```
TEST_CASE("ScopedPtr supports comparison with nullptr") {
  SECTION("operator==") {
    ScopedOathbreakers legolas{};
    REQUIRE(legolas == nullptr); ❶
  }
  SECTION("operator!=") {
    ScopedOathbreakers aragorn{ new DeadMenOfDunharrow{} };
    REQUIRE(aragorn != nullptr); ❷
  }
}
```

Listing 11-5: The `boost::scoped_ptr` supports comparison with `nullptr`.

An empty scoped pointer equals (==) nullptr ❶, whereas a full scoped pointer doesn't equal (!=) nullptr ❷.

Swapping

Sometimes you want to switch the dynamic object owned by a scoped_ptr with the dynamic object owned by another scoped_ptr. This is called an *object swap*, and scoped_ptr contains a swap method that implements this behavior, as shown in Listing 11-6.

```
TEST_CASE("ScopedPtr supports swap") {
  auto message1 = "The way is shut.";
  auto message2 = "Until the time comes.";
  ScopedOathbreakers aragorn {
    new DeadMenOfDunharrow{ message1 } ❶
  };
  ScopedOathbreakers legolas {
    new DeadMenOfDunharrow{ message2 } ❷
  };
  aragorn.swap(legolas); ❸
  REQUIRE(legolas->message == message1); ❹
  REQUIRE(aragorn->message == message2); ❺
}
```

Listing 11-6: The `boost::scoped_ptr` supports `swap`.

You construct two scoped_ptr objects, aragorn ❶ and legolas ❷, each with a different message. After you perform a swap between aragorn and legolas ❸, they exchange dynamic objects. When you pull out their messages after the swap, you find that they've switched ❹❺.

Resetting and Replacing a scoped_ptr

Rarely do you want to destruct an object owned by `scoped_ptr` before the `scoped_ptr` dies. For example, you might want to replace its owned object with a new dynamic object. You can handle both of these tasks with the overloaded `reset` method of `scoped_ptr`.

If you provide no argument, `reset` simply destroys the owned object.

If you instead provide a new dynamic object as a parameter, `reset` will first destroy the currently owned object and then gain ownership of the parameter. Listing 11-7 illustrates such behavior with one test for each scenario.

```
TEST_CASE("ScopedPtr reset") {
  ScopedOathbreakers aragorn{ new DeadMenOfDunharrow{} }; ❶
  SECTION("destructs owned object.") {
    aragorn.reset(); ❷
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 0); ❸
  }
  SECTION("can replace an owned object.") {
    auto message = "It was made by those who are Dead.";
    auto new_dead_men = new DeadMenOfDunharrow{ message }; ❹
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 2); ❺
    aragorn.reset(new_dead_men); ❻
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❼
    REQUIRE(aragorn->message == new_dead_men->message); ❽
    REQUIRE(aragorn.get() == new_dead_men); ❾
  }
}
```

Listing 11-7: The `boost::scoped_ptr` supports `reset`.

The first step in both tests is to construct the scoped pointer `aragorn` owning a `DeadMenOfDunharrow` ❶. In the first test, you call `reset` without an argument ❷. This causes the scoped pointer to destruct its owned object, and `oaths_to_fulfill` decrements to 0 ❸.

In the second test, you create the new, dynamically allocated `new_dead_men` with a custom `message` ❹. This increases the `oaths_to_fill` to 2, because `aragorn` is also still alive ❺. Next, you invoke `reset` with `new_dead_men` as the argument ❻, which does two things:

- It causes the original `DeadMenOfDunharrow` owned by `aragorn` to get destructed, which decrements `oaths_to_fulfill` to 1 ❼.
- It emplaces `new_dead_men` as the dynamically allocated object owned by `aragorn`. When you dereference the `message` field, notice that it matches the `message` held by `new_dead_men` ❽. (Equivalently, `aragorn.get()` yields `new_dead_men` ❾.)