[How to Write Tests 2](#_Toc494370187)

[The Anatomy of a Test Function 2](#_Toc494370188)

[Checking Results with the assert! Macro 6](#_Toc494370189)

[Testing Equality with the assert\_eq! and assert\_ne! Macros 10](#_Toc494370190)

[Custom Failure Messages 12](#_Toc494370191)

[Checking for Panics with should\_panic 14](#_Toc494370192)

[Controlling How Tests are Run 19](#_Toc494370193)

[Running Tests in Parallel or Consecutively 19](#_Toc494370194)

[Showing Function Output 20](#_Toc494370195)

[Running a Subset of Tests by Name 22](#_Toc494370196)

[Running Single Tests 23](#_Toc494370197)

[Filtering to Run Multiple Tests 24](#_Toc494370198)

[Ignore Some Tests Unless Specifically Requested 24](#_Toc494370199)

[Test Organization 26](#_Toc494370200)

[Unit Tests 26](#_Toc494370201)

[The Tests Module and #[cfg(test)] 26](#_Toc494370202)

[Testing Private Functions 27](#_Toc494370203)

[Integration Tests 28](#_Toc494370204)

[The *tests* Directory 28](#_Toc494370205)

[Submodules in Integration Tests 30](#_Toc494370206)

[Integration Tests for Binary Crates 32](#_Toc494370207)

[Summary 33](#_Toc494370208)

Chapter 11

Writing Automated Tests

In his 1972 essay “The Humble Programmer,” Edsger W. Dijkstra said that “Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.” That doesn’t mean we shouldn’t try to test as much as we can! Correctness in our programs is the extent to which our code does what we intend it to do. Rust is a programming language designed with a high degree of concern about the correctness of programs, but correctness is complex and not easy to prove. Rust’s type system shoulders a huge part of this burden, but the type system cannot catch every kind of incorrectness. As such, Rust includes support for writing automated software tests within the language.

As an example, say we write a function called add\_two that adds two to whatever number is passed to it. This function’s signature accepts an integer as a parameter and returns an integer as a result. When we implement and compile that function, Rust does all the type checking and borrow checking that you’ve learned so far to ensure that, for instance, we aren’t passing a String value or an invalid reference to this function. But Rust can’t check that this function will do precisely what we intend, which is return the parameter plus two rather than, say, the parameter plus 10 or the parameter minus 50! That’s where tests come in.

We can write tests that assert, for example, that when we pass 3 to the add\_two function, the returned value is 5. We can run these tests whenever we make changes to our code to make sure any existing correct behavior has not changed.

Testing is a complex skill: although we can’t cover every detail about how to write good tests in one chapter, we’ll discuss the mechanics of Rust’s testing facilities. We’ll talk about the annotations and macros available to you when writing your tests, the default behavior and options provided for running your tests, and how to organize tests into unit tests and integration tests.

How to Write Tests

Tests are Rust functions that verify that the non-test code is functioning in the expected manner. The bodies of test functions typically perform these three actions:

Set up any needed data or state

Run the code we want to test

Assert the results are what we expect

Let’s look at the features Rust provides specifically for writing tests that take these actions, which include the test attribute, a few macros, and the should\_panic attribute.

The Anatomy of a Test Function

At its simplest, a test in Rust is a function that’s annotated with the test attribute. Attributes are metadata about pieces of Rust code; one example is the derive attribute we used with structs in Chapter 5. To change a function into a test function, we add #[test] on the line before fn. When we run our tests with the cargo test command, Rust builds a test runner binary that runs the functions annotated with the test attribute and reports on whether each test function passes or fails.

Prod: xref OK

In Chapter 7, we saw that when we make a new library project with Cargo, a test module with a test function in it is automatically generated for us. This module helps us start writing our tests so we don’t have to look up the exact structure and syntax of test functions every time we start a new project. We can add as many additional test functions and as many test modules as we want!

Prod: xref OK

We’ll explore some aspects of how tests work by experimenting with the template test generated for us without actually testing any code. Then we’ll write some real-world tests that call some code that we’ve written and assert that its behavior is correct.

Let’s create a new library project called adder:

$ cargo new adder

Created library `adder` project

$ cd adder

The contents of the src/lib.rs file in your adder library should look like Listing 11-1:

prod: remove “filename:” from margin filenames, global

src/lib.rs

#[cfg(test)]

mod tests {

#[test]

fn it\_works() {

assert\_eq!(2 + 2, 4);

}

}

Listing 11-1: The test module and function generated automatically by cargo new

For now, let’s ignore the top two lines and focus on the function to see how it works. Note the #[test] annotation before the fn line: this attribute indicates this is a test function, so the test runner knows to treat this function as a test. We could also have non-test functions in the tests module to help set up common scenarios or perform common operations, so we need to indicate which functions are tests by using the #[test] attribute.

The function body uses the assert\_eq! macro to assert that 2 + 2 equals 4. This assertion serves as an example of the format for a typical test. Let’s run it to see that this test passes.

The cargo test command runs all tests in our project, as shown in Listing 11-2:

$ cargo test

Compiling adder v0.1.0 (file:///projects/adder)

Finished dev [unoptimized + debuginfo] target(s) in 0.22 secs

Running target/debug/deps/adder-ce99bcc2479f4607

 running 1 test

 test tests::it\_works ... ok

 test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

 Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Listing 11-2: The output from running the automatically generated test

Cargo compiled and ran the test. After the Compiling, Finished, and Running lines is the line running 1 test . The next line  shows the name of the generated test function, called it\_works, and the result of running that test, ok. The overall summary of running the tests appears next. The text test result: ok. means that all the tests passed, and the portion that reads 1 passed; 0 failed totals the number of tests that passed or failed.

Because we don’t have any tests we’ve marked as ignored, the summary shows 0 ignored. We also haven’t filtered the tests being run, so the end of the summary shows 0 filtered out. We’ll talk about ignoring and filtering out tests in the next section, “Controlling How Tests Are Run.”

The 0 measured statistic is for benchmark tests that measure performance. Benchmark tests are, as of this writing, only available in nightly Rust. See Chapter 1 on page XX for more information about nightly Rust.

prod: add xref

The next part of the test output , which starts with Doc-tests adder, is for the results of any documentation tests. We don’t have any documentation tests yet, but Rust can compile any code examples that appear in our API documentation. This feature helps us keep our docs and our code in sync! We’ll discuss how to write documentation tests in the “Documentation Comments” section of Chapter 14 on page XX. For now, we’ll ignore the Doc-tests output.

prod: add xref

Let’s change the name of our test to see how that changes the test output. Change the it\_works function to a different name, such as exploration, like so:

src/lib.rs

#[cfg(test)]

mod tests {

#[test]

fn exploration() {

assert\_eq!(2 + 2, 4);

}

}

Then run cargo test again. The output now shows exploration instead of it\_works:

running 1 test

test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Let’s add another test, but this time we’ll make a test that fails! Tests fail when something in the test function panics. Each test is run in a new thread, and when the main thread sees that a test thread has died, the test is marked as failed. We talked about the simplest way to cause a panic in Chapter 9, which is to call the panic! macro. Enter the new test, another, so your src/lib.rs file looks like Listing 11-3:

prod: xref ok

src/lib.rs

#[cfg(test)]

mod tests {

#[test]

fn exploration() {

assert\_eq!(2 + 2, 4);

}

#[test]

fn another() {

panic!("Make this test fail");

}

}

Listing 11-3: Adding a second test that will fail because we call the panic! macro

Run the tests again using cargo test. The output should look like Listing 11-4, which shows that our exploration test passed and another failed:

running 2 tests

test tests::exploration ... ok

 test tests::another ... FAILED

 failures:

---- tests::another stdout ----

thread 'tests::another' panicked at 'Make this test fail', src/lib.rs:10:8

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

 failures:

tests::another

 test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

error: test failed

Listing 11-4: Test results when one test passes and one test fails

Instead of ok, the line test tests::another shows FAILED . Two new sections appear between the individual results and the summary: the first section  displays the detailed reason for each test failure. In this case, another failed because it panicked at 'Make this test fail', which happened on line 10 in the src/lib.rs file. The next section  lists just the names of all the failing tests, which is useful when there are lots of tests and lots of detailed failing test output. We can use the name of a failing test to run just that test to more easily debug it; we’ll talk more about ways to run tests in the “Controlling How Tests Are Run” section.

The summary line displays at the end : overall, our test result is FAILED. We had one test pass and one test fail.

Now that you’ve seen what the test results look like in different scenarios, let’s look at some macros other than panic! that are useful in tests.

Checking Results with the assert! Macro

The assert! macro, provided by the standard library, is useful when you want to ensure that some condition in a test evaluates to true. We give the assert! macro an argument that evaluates to a boolean. If the value is true, assert! does nothing and the test passes. If the value is false, the assert! macro calls the panic! macro, which causes the test to fail. Using the assert! macro helps us check that our code is functioning in the way we intend.

In Chapter 5, Listing 5-9 on page XX, we used a Rectangle struct and a can\_hold method, which are repeated here in Listing 11-5. Let’s put this code in the src/lib.rs file and write some tests for it using the assert! macro.

Prod: fill page xref

src/lib.rs

#[derive(Debug)]

pub struct Rectangle {

length: u32,

width: u32,

}

impl Rectangle {

pub fn can\_hold(&self, other: &Rectangle) -> bool {

self.length > other.length && self.width > other.width

}

}

Listing 11-5: Using the Rectangle struct and its can\_hold method from Chapter 5

The can\_hold method returns a boolean, which means it’s a perfect use case for the assert! macro. In Listing 11-6, we write a test that exercises the can\_hold method by creating a Rectangle instance that has a length of 8 and a width of 7, and asserting that it can hold another Rectangle instance that has a length of 5 and a width of 1:

src/lib.rs

#[cfg(test)]

mod tests {

 use super::\*;

#[test]

 fn larger\_can\_hold\_smaller() {

 let larger = Rectangle { length: 8, width: 7 };

let smaller = Rectangle { length: 5, width: 1 };

 assert!(larger.can\_hold(&smaller));

}

}

Listing 11-6: A test for can\_hold that checks that a larger rectangle can indeed hold a smaller rectangle

Note that we’ve added a new line inside the tests module: the use super::\*; line . The tests module is a regular module that follows the usual visibility rules we covered in Chapter 7 in the “Privacy Rules” section on page XX. Because the tests module is an inner module, we need to bring the code under test in the outer module into the scope of the inner module. We use a glob here so anything we define in the outer module is available to this tests module.

prod: fill/link xref

We’ve named our test larger\_can\_hold\_smaller , and we’ve created the two Rectangle instances that we need . Then we called the assert! macro and passed it the result of calling larger.can\_hold(&smaller) . This expression is supposed to return true, so our test should pass. Let’s find out!

running 1 test

test tests::larger\_can\_hold\_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

It does pass! Let’s add another test, this time asserting that a smaller rectangle cannot hold a larger rectangle:

src/lib.rs

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn larger\_can\_hold\_smaller() {

// ...snip...

}

#[test]

fn smaller\_cannot\_hold\_larger() {

let larger = Rectangle { length: 8, width: 7 };

let smaller = Rectangle { length: 5, width: 1 };

assert!(!smaller.can\_hold(&larger));

}

}

Because the correct result of the can\_hold function in this case is false, we need to negate that result before we pass it to the assert! macro. As a result, our test will pass if can\_hold returns false:

running 2 tests

test tests::smaller\_cannot\_hold\_larger ... ok

test tests::larger\_can\_hold\_smaller ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Two tests that pass! Now let’s see what happens to our test results when we introduce a bug in our code. Let’s change the implementation of the can\_hold method by replacing the greater-than sign with a less-than sign when it compares the lengths:

// ...snip...

impl Rectangle {

pub fn can\_hold(&self, other: &Rectangle) -> bool {

self.length < other.length && self.width > other.width

}

}

Running the tests now produces the following:

running 2 tests

test tests::smaller\_cannot\_hold\_larger ... ok

test tests::larger\_can\_hold\_smaller ... FAILED

failures:

---- tests::larger\_can\_hold\_smaller stdout ----

thread 'tests::larger\_can\_hold\_smaller' panicked at 'assertion failed:

larger.can\_hold(&smaller)', src/lib.rs:22:8

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

failures:

tests::larger\_can\_hold\_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

Our tests caught the bug! Because larger.length is 8 and smaller.length is 5, the comparison of the lengths in can\_hold now returns false: 8 is not less than 5.

Testing Equality with the assert\_eq! and assert\_ne! Macros

A common way to test functionality is to compare the result of the code under test to the value we expect the code to return to make sure they’re equal. We could do this using the assert! macro and passing it an expression using the == operator. However, this is such a common test that the standard library provides a pair of macros—assert\_eq! and assert\_ne!—to perform this test more conveniently. These macros compare two arguments for equality or inequality, respectively. They’ll also print the two values if the assertion fails, which makes it easier to see why the test failed; conversely, the assert! macro only indicates that it got a false value for the == expression, not the values that lead to the false value.

In Listing 11-7, we write a function named add\_two that adds 2 to its parameter and returns the result. Then we test this function using the assert\_eq! macro.

src/lib.rs

pub fn add\_two(a: i32) -> i32 {

a + 2

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn it\_adds\_two() {

assert\_eq!(4, add\_two(2));

}

}

Listing 11-7: Testing the function add\_two using the assert\_eq! macro

Let’s check that it passes!

running 1 test

test tests::it\_adds\_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

The first argument we gave to the assert\_eq! macro, 4, is equal to the result of calling add\_two(2). The line for this test is test tests::it\_adds\_two ... ok, and the ok text indicates that our test passed!

Let’s introduce a bug into our code to see what it looks like when a test that uses assert\_eq! fails. Change the implementation of the add\_two function to instead add 3:

pub fn add\_two(a: i32) -> i32 {

a + 3

}

Run the tests again:

running 1 test

test tests::it\_adds\_two ... FAILED

failures:

---- tests::it\_adds\_two stdout ----

 thread 'tests::it\_adds\_two' panicked at 'assertion failed: `(left == right)`

v left: `4`,

w right: `5`', src/lib.rs:11:8

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

failures:

tests::it\_adds\_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

Our test caught the bug! The it\_adds\_two test failed, displaying the message assertion failed: `(left == right)` and showing that left was 4  and right was 5. This message is useful and helps us start debugging: it means the left argument to assert\_eq! was 4, but the right argument, where we had add\_two(2), was 5.

Note that in some languages and test frameworks, the parameters to the functions that assert two values are equal are called expected and actual, and the order in which we specify the arguments matters. However, in Rust, they’re called left and right, and the order in which we specify the value we expect and the value that the code under test produces doesn’t matter. We could write the assertion in this test as assert\_eq!(add\_two(2), 4), which would result in a failure message that displays assertion failed: `(left == right)` and that left was 5 and right was 4.

The assert\_ne! macro will pass if the two values we give it are not equal and fail if they’re equal. This macro is most useful for cases when we’re not sure what a value will be, but we know what the value definitely won’t be if our code is functioning as we intend. For example, if we’re testing a function that is guaranteed to change its input in some way, but the way in which the input is changed depends on the day of the week that we run our tests, the best thing to assert might be that the output of the function is not equal to the input.

Under the surface, the assert\_eq! and assert\_ne! macros use the operators == and !=, respectively. When the assertions fail, these macros print their arguments using debug formatting, which means the values being compared must implement the PartialEq and Debug traits. All the primitive types and most of the standard library types implement these traits. For structs and enums that you define, you’ll need to implement PartialEq to assert that values of those types are equal or not equal. You’ll need to implement Debug to print out the values when the assertion fails. Because both traits are derivable traits, as mentioned in Listing 5-12 in Chapter 5, this is usually as straightforward as adding the #[derive(PartialEq, Debug)] annotation to your struct or enum definition. See Appendix C on page XX for more details about these and other derivable traits.

prod: link xrefs

Adding Custom Failure Messages

We can also add a custom message to be printed with the failure message as optional arguments to the assert!, assert\_eq!, and assert\_ne! macros. Any arguments specified after the one required argument to assert! or the two required arguments to assert\_eq! and assert\_ne! are passed along to the format! macro (discussed in Chapter 8 in the “Concatenation with the + Operator or the format! Macro” section on page XX), so you can pass a format string that contains {} placeholders and values to go in those placeholders. Custom messages are useful to document what an assertion means; when a test fails, we’ll have a better idea of what the problem is with the code.

prod: Check/link xref

For example, let’s say we have a function that greets people by name, and we want to test that the name we pass into the function appears in the output:

src/lib.rs

pub fn greeting(name: &str) -> String {

format!("Hello {}!", name)

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn greeting\_contains\_name() {

let result = greeting("Carol");

assert!(result.contains("Carol"));

}

}

The requirements for this program haven’t been agreed upon yet, and we’re pretty sure the Hello text at the beginning of the greeting will change. We decided we don’t want to have to update the test for the name when that happens, so instead of checking for exact equality to the value returned from the greeting function, we’ll just assert that the output contains the text of the input parameter.

Let’s introduce a bug into this code by changing greeting to not include name to see what this test failure looks like:

pub fn greeting(name: &str) -> String {

String::from("Hello!")

}

Running this test produces the following:

running 1 test

test tests::greeting\_contains\_name ... FAILED

failures:

---- tests::greeting\_contains\_name stdout ----

thread 'tests::greeting\_contains\_name' panicked at 'assertion failed:

result.contains("Carol")', src/lib.rs:12:8

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

failures:

tests::greeting\_contains\_name

This result just indicates that the assertion failed and which line the assertion is on. A more useful failure message in this case would print the value we got from the greeting function. Let’s change the test function, giving it a custom failure message made from a format string with a placeholder filled in with the actual value we got from the greeting function:

#[test]

fn greeting\_contains\_name() {

let result = greeting("Carol");

assert!(

result.contains("Carol"),

"Greeting did not contain name, value was `{}`", result

);

}

Now when we run the test, we’ll get a more informative error message:

---- tests::greeting\_contains\_name stdout ----

thread 'tests::greeting\_contains\_name' panicked at 'Greeting did not contain

name, value was `Hello!`', src/lib.rs:12:8

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

We can see the value we actually got in the test output, which would help us debug what happened instead of what we were expecting to happen.

Checking for Panics with should\_panic

In addition to checking that our code returns the correct values we expect, it’s also important to check that our code handles error conditions as we expect. For example, consider the Guess type that we created in Chapter 9, Listing 9-9 on page XX. Other code that uses Guess depends on the guarantee that Guess instances will only contain values between 1 and 100. We can write a test that ensures that attempting to create a Guess instance with a value outside that range panics.

prod: fill/link xref

We do this by adding another attribute, should\_panic, to our test function. This attribute makes a test pass if the code inside the function panics; the test will fail if the code inside the function doesn’t panic.

Listing 11-8 shows a test that checks that the error conditions of Guess::new happen when we expect:

src/lib.rs

pub struct Guess {

value: u32,

}

impl Guess {

pub fn new(value: u32) -> Guess {

if value < 1 || value > 100 {

panic!("Guess value must be between 1 and 100, got {}.", value);

}

Guess {

value

}

}

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

#[should\_panic]

fn greater\_than\_100() {

Guess::new(200);

}

}

Listing 11-8: Testing that a condition will cause a panic!

We place the #[should\_panic] attribute after the #[test] attribute and before the test function it applies to. Let’s look at the result when this test passes:

running 1 test

test tests::greater\_than\_100 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Looks good! Now let’s introduce a bug in our code by removing the condition that the new function will panic if the value is greater than 100:

// ...snip...

impl Guess {

pub fn new(value: u32) -> Guess {

if value < 1 {

panic!("Guess value must be between 1 and 100, got {}.", value);

}

Guess {

value

}

}

}

When we run the test in Listing 11-8, it will fail:

running 1 test

test tests::greater\_than\_100 ... FAILED

failures:

failures:

tests::greater\_than\_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

We don’t get a very helpful message in this case, but when we look at the test function, we see that it’s annotated with #[should\_panic]. The failure we got means that the code in the test function did not cause a panic.

Tests that use should\_panic can be imprecise because they only indicate that the code has caused some panic. A should\_panic test would pass even if the test panics for a different reason than the one we were expecting to happen. To make should\_panic tests more precise, we can add an optional expected parameter to the should\_panic attribute. The test harness will make sure that the failure message contains the provided text. For example, consider the modified code for Guess in Listing 11-9 where the new function panics with different messages depending on whether the value was too small or too large:

src/lib.rs

// ...snip...

impl Guess {

pub fn new(value: u32) -> Guess {

if value < 1 {

panic!("Guess value must be greater than or equal to 1, got {}.",

value);

} else if value > 100 {

panic!("Guess value must be less than or equal to 100, got {}.",

value);

}

Guess {

value

}

}

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

#[should\_panic(expected = "Guess value must be less than or equal to 100")]

fn greater\_than\_100() {

Guess::new(200);

}

}

Listing 11-9: Testing that a condition will cause a panic! with a particular panic message

This test will pass because the value we put in the should\_panic attribute’s expected parameter is a substring of the message that the Guess::new function panics with. We could have specified the entire panic message that we expect, which in this case would be Guess value must be less than or equal to 100, got 200. What you choose to specify in the expected parameter for should\_panic depends on how much of the panic message is unique or dynamic and how precise you want your test to be. In this case, a substring of the panic message is enough to ensure that the code in the test function executes the else if value > 100 case.

To see what happens when a should\_panic test with an expected message fails, let’s again introduce a bug into our code by swapping the bodies of the if value < 1 and the else if value > 100 blocks:

if value < 1 {

panic!("Guess value must be less than or equal to 100, got {}.", value);

} else if value > 100 {

panic!("Guess value must be greater than or equal to 1, got {}.", value);

}

This time when we run the should\_panic test, it will fail:

running 1 test

test tests::greater\_than\_100 ... FAILED

failures:

---- tests::greater\_than\_100 stdout ----

thread 'tests::greater\_than\_100' panicked at 'Guess value must be greater than or equal to 1, got 200.', src/lib.rs:11:12

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

note: Panic did not include expected string 'Guess value must be less than or

equal to 100'

failures:

tests::greater\_than\_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

The failure message indicates that this test did indeed panic as we expected, but the panic message did not include the expected string 'Guess value must be less than or equal to 100'. The panic message that we did get in this case was Guess value must be greater than or equal to 1, got 200. Now we can start figuring out where our bug is!

Now that you know several ways to write tests, let’s look at what is happening when we run our tests and explore the different options we can use with cargo test.

Controlling How Tests Are Run

Just as cargo run compiles your code and then runs the resulting binary, cargo test compiles your code in test mode and runs the resulting test binary. You can specify command line options to change the default behavior of cargo test. For example, the default behavior of the binary produced by cargo test is to run all the tests in parallel and capture output generated during test runs, preventing the output from being displayed and making it easier to read the output related to the test results.

Some command line options go to cargo test and some go to the resulting test binary. To separate these two types of arguments, you list the arguments that go to cargo test followed by the separator -- and then the arguments that go to the test binary. Running cargo test --help displays the options you can use with cargo test, and running cargo test -- --help displays the options you can use after the separator --.

Running Tests in Parallel or Consecutively

When you run multiple tests, by default they run in parallel using threads. This means the tests will finish running faster so you can get feedback quicker on whether or not your code is working. Because the tests are running at the same time, make sure your tests don’t depend on each other or on any shared state, including a shared environment, such as the current working directory or environment variables.

For example, say each of your tests runs some code that creates a file on disk named test-output.txt and writes some data to that file. Then each test reads the data in that file and asserts that the file contains a particular value, which is different in each test. Because the tests run at the same time, one test might overwrite the file between when another test writes and reads the file. The second test will then fail, not because the code is incorrect, but because the tests have interfered with each other while running in parallel. One solution is to make sure each test writes to a different file; another solution is to run the tests one at a time.

If you don’t want to run the tests in parallel or if you want more fine-grained control over the number of threads used, you can send the --test-threads flag and the number of threads you want to use to the test binary. Take a look at the following example:

$ cargo test -- --test-threads=1

We set the number of test threads to 1, telling the program not to use any parallelism. Running the tests using one thread will take longer than running them in parallel, but the tests won’t interfere with each other if they share state.

Showing Function Output

By default, if a test passes, Rust’s test library captures anything printed to standard output. For example, if we call println! in a test and the test passes, we won’t see the println! output in the terminal: we’ll only see the line that indicates the test passed. If a test fails, we’ll see whatever was printed to standard output with the rest of the failure message.

As an example, Listing 11-10 has a silly function that prints the value of its parameter and returns 10, as well as a test that passes and a test that fails.

src/lib.rs

fn prints\_and\_returns\_10(a: i32) -> i32 {

println!("I got the value {}", a);

10

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn this\_test\_will\_pass() {

let value = prints\_and\_returns\_10(4);

assert\_eq!(10, value);

}

#[test]

fn this\_test\_will\_fail() {

let value = prints\_and\_returns\_10(8);

assert\_eq!(5, value);

}

}

Listing 11-10: Tests for a function that calls println!

When we run these tests with cargo test, we’ll see the following output:

running 2 tests

test tests::this\_test\_will\_pass ... ok

test tests::this\_test\_will\_fail ... FAILED

failures:

---- tests::this\_test\_will\_fail stdout ----

 I got the value 8

thread 'tests::this\_test\_will\_fail' panicked at 'assertion failed: `(left == right)`

left: `5`,

right: `10`', src/lib.rs:19:8

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

failures:

tests::this\_test\_will\_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

Note that nowhere in this output do we see I got the value 4, which is what is printed when the test that passes runs. That output has been captured. The output from the test that failed, I got the value 8 , appears in the section of the test summary output, which also shows the cause of the test failure.

If we want to see printed values for passing tests as well, we can disable the output capture behavior by using the --nocapture flag:

$ cargo test -- --nocapture

When we run the tests in Listing 11-10 again with the --nocapture flag, we see the following output:

running 2 tests

I got the value 4

I got the value 8

test tests::this\_test\_will\_pass ... ok

thread 'tests::this\_test\_will\_fail' panicked at 'assertion failed: `(left == right)`

left: `5`,

right: `10`', src/lib.rs:19:8

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

test tests::this\_test\_will\_fail ... FAILED

failures:

failures:

tests::this\_test\_will\_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

Note that the output for the tests and the test results are interleaved; the reason is that the tests are running in parallel, as we talked about in the previous section. Try using the --test-threads=1 option and the --nocapture flag, and see what the output looks like then!

Running a Subset of Tests by Name

Sometimes, running a full test suite can take a long time. If you’re working on code in a particular area, you might want to run only the tests pertaining to that code. You can choose which tests to run by passing cargo test the name or names of the test(s) you want to run as an argument.

To demonstrate how to run a subset of tests, we’ll create three tests for our add\_two function, as shown in Listing 11-11, and choose which ones to run:

src/lib.rs

pub fn add\_two(a: i32) -> i32 {

a + 2

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn add\_two\_and\_two() {

assert\_eq!(4, add\_two(2));

}

#[test]

fn add\_three\_and\_two() {

assert\_eq!(5, add\_two(3));

}

#[test]

fn one\_hundred() {

assert\_eq!(102, add\_two(100));

}

}

Listing 11-11: Three tests with three different names

If we run the tests without passing any arguments, as we saw earlier, all the tests will run in parallel:

running 3 tests

test tests::add\_two\_and\_two ... ok

test tests::add\_three\_and\_two ... ok

test tests::one\_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Running Single Tests

We can pass the name of any test function to cargo test to run only that test:

$ cargo test one\_hundred

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running target/debug/deps/adder-06a75b4a1f2515e9

running 1 test

test tests::one\_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out

Only the test with the name one\_hundred ran; the other two tests didn't match that name. The test output lets us know we had more tests than what this command ran by displaying 2 filtered out at the end of the summary line.

We can’t specify the names of multiple tests in this way; only the first value given to cargo test will be used. But there is a way to run multiple tests.

Filtering to Run Multiple Tests

We can specify part of a test name, and any test whose name matches that value will be run. For example, because two of our tests’ names contain add, we can run those two by running cargo test add:

$ cargo test add

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running target/debug/deps/adder-06a75b4a1f2515e9

running 2 tests

test tests::add\_two\_and\_two ... ok

test tests::add\_three\_and\_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out

This command ran all tests with add in the name and filtered out the test named one\_hundred. Also note that the module in which tests appear becomes part of the test’s name, so we can run all the tests in a module by filtering on the module’s name.

Ignoring Some Tests Unless Specifically Requested

Sometimes a few specific tests can be very time-consuming to execute, so you might want to exclude them during most runs of cargo test. Rather than listing as arguments all tests you do want to run, you can instead annotate the time-consuming tests using the ignore attribute to exclude them, as shown here:

src/lib.rs

#[test]

fn it\_works() {

assert\_eq!(2 + 2, 4);

}

#[test]

#[ignore]

fn expensive\_test() {

// code that takes an hour to run

}

After #[test] we add the #[ignore] line to the test we want to exclude. Now when we run our tests, it\_works runs, but expensive\_test doesn’t:

$ cargo test

Compiling adder v0.1.0 (file:///projects/adder)

Finished dev [unoptimized + debuginfo] target(s) in 0.24 secs

Running target/debug/deps/adder-ce99bcc2479f4607

running 2 tests

test expensive\_test ... ignored

test it\_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out

The expensive\_test function is listed as ignored. If we want to run only the ignored tests, we can use cargo test -- --ignored:

$ cargo test -- --ignored

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test

test expensive\_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out

By controlling which tests run, you can make sure your cargo test results will be fast. When you’re at a point where it makes sense to check the results of the ignored tests and you have time to wait for the results, you can run cargo test -- --ignored instead.

Test Organization

As mentioned at the start of the chapter, testing is a complex discipline, and different people use different terminology and organization. The Rust community thinks about tests in terms of two main categories: unit tests and integration tests. Unit tests are small and more focused, testing one module in isolation at a time, and can test private interfaces. Integration tests are entirely external to your library and use your code in the same way any other external code would, using only the public interface and potentially exercising multiple modules per test.

Writing both kinds of tests is important to ensure that the pieces of your library are doing what you expect them to separately and together.

Unit Tests

The purpose of unit tests is to test each unit of code in isolation from the rest of the code to quickly pinpoint where code is and isn’t working as expected. We put unit tests in the src directory in each file with the code that they’re testing. The convention is that we create a module named tests in each file to contain the test functions, and we annotate the module with cfg(test).

The Tests Module and #[cfg(test)]

The #[cfg(test)] annotation on the tests module tells Rust to compile and run the test code only when we run cargo test, but not when we run cargo build. This saves compile time when we only want to build the library and saves space in the resulting compiled artifact because the tests are not included. You’ll see that because integration tests go in a different directory, they don’t need the #[cfg(test)] annotation. However, because unit tests go in the same files as the code, we use #[cfg(test)] to specify that they shouldn’t be included in the compiled result.

Recall that when we generated the new adder project in the first section of this chapter, Cargo generated this code for us:

src/lib.rs

#[cfg(test)]

mod tests {

#[test]

fn it\_works() {

assert\_eq!(2 + 2, 4);

}

}

This code is the automatically generated test module. The attribute cfg stands for configuration and tells Rust that the following item should only be included given a certain configuration option. In this case, the configuration option is test, which is provided by Rust for compiling and running tests. By using the cfg attribute, Cargo compiles our test code only if we actively run the tests with cargo test. This includes any helper functions that might be within this module, in addition to the functions annotated with #[test].

Testing Private Functions

There’s debate within the testing community about whether or not private functions should be tested directly, and other languages make it difficult or impossible to test private functions. Regardless of which testing ideology you adhere to, Rust’s privacy rules do allow you to test private functions. Consider the code in Listing 11-12 with the private function internal\_adder:

src/lib.rs

pub fn add\_two(a: i32) -> i32 {

internal\_adder(a, 2)

}

fn internal\_adder(a: i32, b: i32) -> i32 {

a + b

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn internal() {

assert\_eq!(4, internal\_adder(2, 2));

}

}

Listing 11-12: Testing a private function

Note that the internal\_adder function is not marked as pub, but because tests are just Rust code and the tests module is just another module, we can import and call internal\_adder in a test just fine. If you don’t think private functions should be tested, there’s nothing in Rust that will compel you to do so.

Integration Tests

In Rust, integration tests are entirely external to your library. They use your library in the same way any other code would, which means they can only call functions that are part of your library’s public API. Their purpose is to test that many parts of your library work together correctly. Units of code that work correctly on their own could have problems when integrated, so test coverage of the integrated code is important as well. To create integration tests, you first need a tests directory.

The tests Directory

We create a tests directory at the top level of our project directory, next to src. Cargo knows to look for integration test files in this directory. We can then make as many test files as we want to in this directory, and Cargo will compile each of the files as an individual crate.

Let’s create an integration test. With the code in Listing 11-12 still in the src/lib.rs file, make a tests directory, create a new file named tests/integration\_test.rs, and enter the code in Listing 11-13:

tests/integration\_test.rs

extern crate adder;

#[test]

fn it\_adds\_two() {

assert\_eq!(4, adder::add\_two(2));

}

Listing 11-13: An integration test of a function in the adder crate

We’ve added extern crate adder at the top of the code, which we didn’t need in the unit tests. The reason is that each test in the tests directory is a separate crate, so we need to import our library into each of them.

We don’t need to annotate any code in tests/integration\_test.rs with #[cfg(test)]. Cargo treats the tests directory specially and compiles files in this directory only when we run cargo test. Run cargo test now:

$ cargo test

Compiling adder v0.1.0 (file:///projects/adder)

Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs

Running target/debug/deps/adder-abcabcabc

 running 1 test

test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

 Running target/debug/deps/integration\_test-ce99bcc2479f4607

running 1 test

 test it\_adds\_two ... ok

 test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

The three sections of output include the unit tests, the integration test, and the doc tests. The first section for the unit tests  is the same as we’ve been seeing: one line for each unit test (one named internal that we added in Listing 11-12) and then a summary line for the unit tests.

The integration tests section starts with the line Running target/debug/deps/integration-test-ce99bcc2479f4607  (the hash at the end of your output will be different). Next, there is a line for each test function in that integration test  and a summary line for the results of the integration test  just before the Doc-tests adder section starts.

Recall that adding more unit test functions in any src file adds more test result lines to the unit tests section. Adding more test functions to the integration test file we created adds more lines to that file’s section. Each integration test file has its own section, so if we add more files in the tests directory, there will be more integration test sections.

We can still run a particular integration test function by specifying the test function’s name as an argument to cargo test. To run all the tests in a particular integration test file, use the --test argument of cargo test followed by the name of the file:

$ cargo test --test integration\_test

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running target/debug/integration\_test-952a27e0126bb565

running 1 test

test it\_adds\_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

This command runs only the tests in the tests/integration\_test.rs file.

Submodules in Integration Tests

As you add more integration tests, you might want to make more than one file in the tests directory to help organize them; for example, you can group the test functions by the functionality they’re testing. As mentioned earlier, each file in the tests directory is compiled as its own separate crate.

Treating each integration test file as its own crate is useful to create separate scopes that are more like the way end users will be using your crate. However, this means files in the tests directory don’t share the same behavior as files in src do, which you learned in Chapter 7 regarding how to separate code into modules and files.

prod: xref ok

The different behavior of files in the tests directory is most noticeable when you have a set of helper functions that would be useful in multiple integration test files and you try to follow the steps in the “Moving Modules to Other Files” section of Chapter 7 to extract them into a common module. For example, if we create tests/common.rs and place a function named setup in it, we can add some code to setup that we want to call from multiple test functions in multiple test files:

prod: check xref

tests/common.rs

pub fn setup() {

// setup code specific to your library's tests would go here

}

When we run the tests again, we’ll see a new section in the test output for the common.rs file, even though this file doesn’t contain any test functions, nor did we call the setup function from anywhere:

running 1 test

test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Running target/debug/deps/common-b8b07b6f1be2db70

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Running target/debug/deps/integration\_test-d993c68b431d39df

running 1 test

test it\_adds\_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Having common appear in the test results with running 0 tests displayed for it is not what we wanted. We just wanted to share some code with the other integration test files.

To avoid having common appear in the test output, instead of creating tests/common.rs, we’ll create tests/common/mod.rs. In the “Rules of Module Filesystems” section of Chapter 7, we used the naming convention module\_name/mod.rs for files of modules that have submodules, and we don’t have submodules for common here, but naming the file this way tells Rust not to treat the common module as an integration test file. When we move the setup function code into tests/common/mod.rs and delete the tests/common.rs file, the section in the test output will no longer appear. Files in subdirectories of the tests directory don’t get compiled as separate crates or have sections in the test output.

prod: check xref

After we’ve created tests/common/mod.rs, we can use it from any of the integration test files as a module. Here’s an example of calling the setup function from the it\_adds\_two test in tests/integration\_test.rs:

tests/integration\_test.rs

extern crate adder;

mod common;

#[test]

fn it\_adds\_two() {

common::setup();

assert\_eq!(4, adder::add\_two(2));

}

Note that the mod common; declaration is the same as the module declarations we demonstrated in Listing 7-4 on page XX. Then in the test function, we can call the common::setup() function.

prod: check xref

Integration Tests for Binary Crates

If our project is a binary crate that only contains a src/main.rs file and doesn’t have a src/lib.rs file, we can’t create integration tests in the tests directory and use extern crate to import functions defined in the src/main.rs file. Only library crates expose functions that other crates can call and use; binary crates are meant to be run on their own.

This is one of the reasons Rust projects that provide a binary have a straightforward src/main.rs file that calls logic that lives in the src/lib.rs file. Using that structure, integration tests can test the library crate by using extern crate to exercise the important functionality. If the important functionality works, the small amount of code in the src/main.rs file will work as well, and that small amount of code doesn’t need to be tested.

Summary

Rust’s testing features provide a way to specify how code should function to ensure it continues to work as we expect even as we make changes. Unit tests exercise different parts of a library separately and can test private implementation details. Integration tests check that many parts of the library work together correctly, and they use the library’s public API to test the code in the same way external code will use it. Even though Rust’s type system and ownership rules help prevent some kinds of bugs, tests are still important to help reduce logic bugs having to do with how your code is expected to behave.

Let’s combine the knowledge you learned in this chapter and in previous chapters and work on a project in the next chapter!