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 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 everything. As such, Rust includes support for writing automated software tests.

Say we write a function `add_two` that adds 2 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 2 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:

1. Set up any needed data or state.
2. Run the code you want to test.
3. Assert the results are what you 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, add `#[test]` on the line before `fn`. When you run your tests with the `cargo test` command, Rust builds a test runner binary that runs the annotated functions and reports on whether each test function passes or fails.

Whenever we make a new library project with Cargo, a test module with a test function in it is automatically generated for us. This module gives you a template for writing your tests so you don't have to look up the exact structure and syntax every time you start a new project. You can add as many additional test functions and as many test modules as you want!

We'll explore some aspects of how tests work by experimenting with the template test before we actually test 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` that will add two numbers:

```
$ cargo new adder --lib
     Created library `adder` project
$ cd adder
```

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

Filename: src/lib.rs

```
pub fn add(left: usize, right: usize) -> usize {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
}
```

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

For now, let's focus solely on the `it_works()` function. Note the `#[test]` annotation: this attribute indicates this is a test function, so the test runner knows to treat this function as a test. We might also have non-test functions in the `tests` module to help set up common scenarios or perform common operations, so we always need to indicate which functions are tests.

The example function body uses the `assert_eq!` macro to assert that `result`, which contains the result of adding 2 and 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 `test` profile [unoptimized + debuginfo] target(s) in 0.57s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

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

Cargo compiled and ran the test. We see the line `running 1 test`. The next line shows the name of the generated test function, called `it_works`, and that the result of running that test is `ok`. The overall summary `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.

It's possible to mark a test as ignored so it doesn't run in a particular instance; we'll cover that in the "Ignoring Some Tests Unless Specifically Requested" section later in this chapter. Because we haven't done that here, the summary shows `0 ignored`. We can also pass an argument to the `cargo test` command to run only tests whose name matches a string; this is called *filtering* and we'll cover that in the "Running a Subset of Tests by Name" section. We also haven't filtered the tests being run, so the end of the summary shows `0 filtered out`.

The `0 measured` statistic is for benchmark tests that measure performance. Benchmark tests are, as of this writing, only available in nightly Rust. See the documentation about benchmark tests to learn more.

The next part of the test output starting at `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 keep your docs and your code in sync! We'll discuss how to write documentation tests in the "Documentation Comments as Tests" section of Chapter 14. For now, we'll ignore the `Doc-tests` output.

Let's start to customize the test to our own needs. First change the name of the `it_works` function to a different name, such as `exploration`, like so:

Filename: src/lib.rs

```
pub fn add(left: usize, right: usize) -> usize {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn exploration() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
}
```

Then run `cargo test` again. The output now shows `exploration` instead of `it_works`:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.59s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Now we'll 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. In Chapter 9, we talked about how the simplest way to panic is to call the `panic!` macro. Enter the new test as a function named `another`, so your *src/lib.rs* file looks like Listing 11-3.

Filename: src/lib.rs

```
pub fn add(left: usize, right: usize) -> usize {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn exploration() {
        let result = add(2, 2);
        assert_eq!(result, 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.

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.72s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::another ... FAILED
test tests::exploration ... ok

failures:

---- tests::another stdout ----
thread 'tests::another' panicked at src/lib.rs:17:9:
Make this test fail
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

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 displays the detailed reason for each test failure. In this case, we get the details that `another` failed because it `panicked at 'Make this test fail'` 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`, nothing happens and the test passes. If the value is `false`, the `assert!` macro calls `panic!` to cause 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 5-15, 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, then write some tests for it using the `assert!` macro.

Filename: src/lib.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

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 width of 8 and a height of 7 and asserting that it can hold another `Rectangle` instance that has a width of 5 and a height of 1.

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(larger.can_hold(&smaller));
    }
}
```

Listing 11-6: A test for `can_hold` that checks whether a larger rectangle can indeed hold a smaller rectangle

Note that we've added a new line inside the `tests` module: `use super::*;`. The `tests` module is a regular module that follows the usual visibility rules we covered in Chapter 7 in the "Paths for Referring to an Item in the Module Tree" section. 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.

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!

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

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

Filename: 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 {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 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`:

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... ok
test tests::smaller_cannot_hold_larger ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Two tests that pass! Now let's see what happens to our test results when we introduce a bug in our code. We'll change the implementation of the `can_hold` method by replacing the greater-than sign with a less-than sign when it compares the widths:

```
// --snip--
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width < other.width && self.height > other.height
    }
}
```

Running the tests now produces the following:

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... FAILED
test tests::smaller_cannot_hold_larger ... ok

failures:

---- tests::larger_can_hold_smaller stdout ----
thread 'tests::larger_can_hold_smaller' panicked at src/lib.rs:28:9:
assertion failed: larger.can_hold(&smaller)
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

Our tests caught the bug! Because `larger.width` is 8 and `smaller.width` is 5, the comparison of the widths 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 verify functionality is to test for equality between the result of the code under test and the value you expect the code to return. You 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, without printing the values that led to the `false` value.

In Listing 11-7, we write a function named `add_two` that adds `2` to its parameter, then we test this function using the `assert_eq!` macro.

Filename: 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!

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

We pass `4` as the argument to `assert_eq!`, which 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 `assert_eq!` looks like when it 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:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
thread 'tests::it_adds_two' panicked at src/lib.rs:11:9:
assertion `left == right` failed
  left: 4
 right: 5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

Our test caught the bug! The `it_adds_two` test failed, and the message tells us that the assertion that fails was `assertion `left == right` failed` and what the `left` and `right` values are. This message helps us start debugging: the `left` argument was `4` but the `right` argument, where we had `add_two(2)`, was `5`. You can imagine that this would be especially helpful when we have a lot of tests going on.

Note that in some languages and test frameworks, the parameters to equality assertion functions 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 the code produces doesn't matter. We could write the assertion in this test as `assert_eq!(add_two(2), 4)`, which would result in the same failure message that displays `assertion failed: `(left == right)``.

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 *shouldn't* be. 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 primitive types and most of the standard library types implement these traits. For structs and enums that you define yourself, you'll need to implement `PartialEq` to assert equality of those types. You'll also need to implement `Debug` to print 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, "Derivable Traits," for more details about these and other derivable traits.

Adding Custom Failure Messages

You 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 required arguments are passed along to the `format!` macro (discussed in Chapter 8 in the "Concatenation with the `+` Operator or the `format!` Macro" section), so you can pass a format string that contains `{}` placeholders and values to go in those placeholders. Custom messages are useful for documenting what an assertion means; when a test fails, you'll have a better idea of what the problem is with the code.

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:

Filename: 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 when the requirements change, 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.

Now let's introduce a bug into this code by changing `greeting` to exclude `name` to see what the default test failure looks like:

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```

Running this test produces the following:

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests src/lib.rs (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at src/lib.rs:12:9:
assertion failed: result.contains("Carol")
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

This result just indicates that the assertion failed and which line the assertion is on. A more useful failure message would print the value from the `greeting` function. Let's add a custom failure message composed of 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:

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.93s
     Running unittests src/lib.rs (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'tests::greeting_contains_name' panicked at src/lib.rs:12:9:
Greeting did not contain name, value was `Hello!`
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

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 return values, it's 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 9-13. Other code that uses `Guess` depends on the guarantee that `Guess` instances will contain only 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.

We do this by adding the attribute `should_panic` to our test function. The test passes if the code inside the function panics; the test fails 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 them to.

Filename: src/lib.rs

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> 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:

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/guessing_game-
57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests guessing_game

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

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: i32) -> 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:

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests src/lib.rs (target/debug/deps/guessing_game-
57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
note: test did not panic as expected

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

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. A `should_panic` test would pass even if the test panics for a different reason from the one we were expecting. 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 is too small or too large.

Filename: src/lib.rs

```
// --snip--

impl Guess {
    pub fn new(value: i32) -> 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 = "less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Listing 11-9: Testing for a `panic!` with a panic message containing a specified substring

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 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:

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests src/lib.rs (target/debug/deps/guessing_game-
57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
thread 'tests::greater_than_100' panicked at src/lib.rs:12:13:
Guess value must be greater than or equal to 1, got 200.
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
note: panic did not contain expected string
      panic message: `"Guess value must be greater than or equal to 1, got 200."`,
 expected substring: `"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;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

The failure message indicates that this test did indeed panic as we expected, but the panic message did not include the expected string `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!

Using `Result<T, E>` in Tests

Our tests so far all panic when they fail. We can also write tests that use `Result<T, E>`! Here's the test from Listing 11-1, rewritten to use `Result<T, E>` and return an `Err` instead of

panicking:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
}
```

The `it_works` function now has the `Result<(), String>` return type. In the body of the function, rather than calling the `assert_eq!` macro, we return `Ok(())` when the test passes and an `Err` with a `String` inside when the test fails.

Writing tests so they return a `Result<T, E>` enables you to use the question mark operator in the body of tests, which can be a convenient way to write tests that should fail if any operation within them returns an `Err` variant.

You can't use the `#[should_panic]` annotation on tests that use `Result<T, E>`. To assert that an operation returns an `Err` variant, *don't* use the question mark operator on the `Result<T, E>` value. Instead, use `assert!(value.is_err())`.

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. 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. You can, however, specify command line options to change this default behavior.

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 ones 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, meaning they finish running faster and you get feedback quicker. Because the tests are running at the same time, you must 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 in the time between another test writing and reading 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 see only 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.

Filename: 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:

```
$ cargo test
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests src/lib.rs (target/debug/deps/silly_function-
160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'tests::this_test_will_fail' panicked at src/lib.rs:19:9:
assertion `left == right` failed
  left: 5
 right: 10
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

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 tell Rust to also show the output of successful tests with `--show-output`.

```
$ cargo test -- --show-output
```

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

```
$ cargo test -- --show-output
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests src/lib.rs (target/debug/deps/silly_function-
160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

successes:

---- tests::this_test_will_pass stdout ----
I got the value 4


successes:
    tests::this_test_will_pass

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'tests::this_test_will_fail' panicked at src/lib.rs:19:9:
assertion `left == right` failed
  left: 5
 right: 10
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass `--lib`
```

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 first create three tests for our `add_two` function, as shown in Listing 11-11, and choose which ones to run.

Filename: 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:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 3 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Running Single Tests

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

```
$ cargo test one_hundred
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.69s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out;
finished in 0.00s
```

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 that didn't run by displaying `2 filtered out` at the end.

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
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
finished in 0.00s
```

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 a test appears 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:

Filename: 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 `test` profile [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

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;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

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
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

By controlling which tests run, you can make sure your `cargo test` results will be fast. When you'll 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. If you want to run all tests whether they're ignored or not, you can run `cargo test -- --include-ignored`.

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. You'll put unit tests in the *src* directory in each file with the code that they're testing. The convention is to create a module named `tests` in each file to contain the test functions and to 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 you run `cargo test`, not when you run `cargo build`. This saves compile time when you 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, you'll 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:

Filename: src/lib.rs

```
pub fn add(left: usize, right: usize) -> usize {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let result = add(2, 2);
        assert_eq!(result, 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`.

Filename: 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`. Tests are just Rust code, and the `tests` module is just another module. As we discussed in the "Paths for Referring to an Item in the Module Tree" section, items in child modules can use the items in their ancestor modules. In this test, we bring all of the `tests` module's parent's items into scope with `use super::*`, and then the test can call `internal_adder`. 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 whether 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, 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, and create a new file named *tests/integration_test.rs*. Your directory structure

should look like this:

```
adder
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    └── integration_test.rs
```

Enter the code in Listing 11-13 into the *tests/integration_test.rs* file:

Filename: tests/integration_test.rs

```
use adder::add_two;

#[test]
fn it_adds_two() {
    assert_eq!(4, add_two(2));
}
```

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

Each file in the `tests` directory is a separate crate, so we need to bring our library into each test crate's scope. For that reason we add `use adder::add_two` at the top of the code, which we didn't need in the unit tests.

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 `test` profile [unoptimized + debuginfo] target(s) in 1.31s
     Running unittests src/lib.rs (target/debug/deps/adder-1082c4b063a8fbe6)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running tests/integration_test.rs (target/debug/deps/integration_test-
1082c4b063a8fbe6)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

The three sections of output include the unit tests, the integration test, and the doc tests. Note that if any test in a section fails, the following sections will not be run. For example, if a unit test fails, there won't be any output for integration and doc tests because those tests will only be run if all unit tests are passing.

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 tests/integration_test.rs`. 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.

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
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.64s
     Running tests/integration_test.rs (target/debug/deps/integration_test-
82e7799c1bc62298)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

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 files 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, which is useful for creating separate scopes to more closely imitate 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, as you learned in Chapter 7 regarding how to separate code into modules and files.

The different behavior of *tests* directory files is most noticeable when you have a set of helper functions to use in multiple integration test files and you try to follow the steps in the "Separating Modules into Different 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:

Filename: 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:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.89s
     Running unittests src/lib.rs (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running tests/common.rs (target/debug/deps/common-92948b65e88960b4)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running tests/integration_test.rs (target/debug/deps/integration_test-
92948b65e88960b4)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

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*. The project directory now looks like this:

```
├── Cargo.lock
├── Cargo.toml
├── src
│   └── lib.rs
└── tests
    ├── common
    │   └── mod.rs
    └── integration_test.rs
```

This is the older naming convention that Rust also understands that we mentioned in the "Alternate File Paths" section of Chapter 7. 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.

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*:

Filename: tests/integration_test.rs

```
use 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 declaration we demonstrated in Listing 7-21. Then in the test function, we can call the `common::setup()` function.

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 bring functions defined in the *src/main.rs* file into scope with a `use` statement. Only library crates expose functions that other crates can 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 with `use` to make the important functionality available. 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 you expect, even as you 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 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 to work on a project!