

## 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`



```
#[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`:

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`, but because tests are just Rust code and the `tests` module is just another module, you 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 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 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:

Filename: `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.

Similarly to how adding more unit test functions adds more result lines to the unit tests section, adding more test functions to the integration test file adds more result lines to this integration test 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, as you learned in Chapter 7 regarding how to separate code into modules and files.

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:

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:

```
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. 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.

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*

```
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. 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 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 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!