Testing Organization
We'll think 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 functions. Integration tests 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)]` attribute.
The Tests Module and `#[cfg(test)]`
The `#[cfg(test)]` annotation on the tests module tells Cairo to compile and run the test code only when you run `scarb test`, not when you run `scarb build`. This saves compile time when you only want to build the project 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 created the new `adder` project in the first section of this chapter, we wrote this first test:
```cairo
{{#include ../listings/ch10-testing-cairo-programs/listing_10_01/src/lib.cairo:it_works}}
```
The attribute `cfg` stands for _configuration_ and tells Cairo 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 Cairo for compiling and running tests. By using the `cfg` attribute, Cairo compiles our test code only if we actively run the tests with `scarb 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, Cairo's privacy rules do allow you to test private functions. Consider the code below with the private function `internal_adder`.
<span class="caption">Filename: src/lib.cairo</span>
```cairo, noplayground
{{#include ../listings/ch10-testing-cairo-programs/no_listing_11_test_private_function/src/lib.cairo}}
```
{{#label test_internal}}
<span class="caption">Listing {{#ref test_internal}}: Testing a private function</span>

Note that the `internal_adder` function is not marked as `pub`. Tests are just Cairo code, and the tests module is just another module. As we discussed in the ["Paths for Referring to an Item in the Module Tree"](ch07-03-paths-for-referring-to-an-item-in-the-module-tree.md) section, items in child modules can use the items in their ancestor modules. In this test, we bring the `tests` module's parent `internal_adder` into scope with `use super::internal_adder;` and then the test can call `internal_adder`. If you don't think private functions should be tested, there's nothing in Cairo that will compel you to do so.
Integration Tests
Integration tests use your library in the same way any other code would. 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_. Scarb knows to look for integration test files in this directory. We can then make as many test files as we want, and Scarb will compile each of the files as an individual crate.
Let's create an integration test. With the code in Listing {{#ref test_internal}} still in the _src/lib.cairo_ file, make a _tests_ directory, and create a new file named _tests/integration_test.cairo_. Your directory structure should look like this:
```shell
adder
% % %  Scarb.lock
% % %  Scarb.toml
% % %  src
%    % % %  lib.cairo
% % %  tests
   % % %  integration_tests.cairo
```
Enter the code in Listing {{#ref test_integration}} into the _tests/integration_test.cairo_ file:
<span class="caption">Filename: tests/integration_tests.cairo</span>
```cairo, noplayground
{{#include ../listings/ch10-testing-cairo-programs/no_listing_09_integration_test/tests/integration_tests.cairo}}
```
{{#label test_integration}}
<span class="caption">Listing {{#ref test_integration}}: An integration test of a function in the `adder` crate</span>
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.cairo_ with `#[cfg(test)]`. Scarb treats the tests directory specially and compiles files in this directory only when

we run `scarb test`. Run `scarb test` now:
```shell
{{#include ../listings/ch10-testing-cairo-programs/no_listing_09_integration_test/output.txt}}
```
The two sections of output include the unit tests and the integration 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 tests because those tests will only be run if all unit tests are passing.

The first displayed section is for the integration tests.

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.

The second displayed section is the same as we've been seeing: one line for each unit test (one named add that we added just above) and then a summary line for the unit tests.

We can still run a particular integration test function by specifying the test function's name as an argument of the option -f to `scarb test` like for instance `scarb test -f integration_tests::internal`. To run all the tests in a particular integration test file, we use the same option of `scarb test` but using only the name of the file.

Then, to run all of our integration tests, we can just add a filter to only run tests whose path contains _integration_tests_.
```shell
{{#include ../listings/ch10-testing-cairo-programs/no_listing_09_integration_test/output_integration.txt}}
```
We see that in the second section for the unit tests, 1 has been filtered out because it is not in the _integration_tests_ 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](ch07-05-separating-modules-into-different-files.md) section of Chapter 7 to extract them into a common module. For example, if we create _tests/common.cairo_ 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:

<span class="caption">Filename: tests/common.cairo</span>
```cairo, noplayground
{{#include ../listings/ch10-testing-cairo-programs/no_listing_12_submodules/tests/common.cairo}}
```

~~~
<span class="caption">Filename: tests/integration_tests.cairo</span>
```cairo, noplayground
{{#include ../listings/ch10-testing-cairo-programs/no_listing_12_submodules/tests/integration_tests.cairo}}
~~~
<span class="caption">Filename: src/lib.cairo</span>
```cairo, noplayground
{{#include ../listings/ch10-testing-cairo-programs/no_listing_12_submodules/src/lib.cairo}}
```
When we run the tests with `scarb test`, we'll see a new section in the test output for the _common.cairo_ file, even though this file doesn't contain any test functions nor did we call the setup function from anywhere:
```shell
{{#include ../listings/ch10-testing-cairo-programs/no_listing_12_submodules/output.txt}}
```
To avoid systematically getting a section for each file of the _tests_ folder, we also have the option of making the `tests/` directory behave like a regular crate, by adding a `tests/lib.cairo` file. In that case, the `tests` directory will no longer compile as one crate per file, but as one crate for the whole directory.
Let's create this _tests/lib.cairo_ file :
<span class="caption">Filename: tests/lib.cairo</span>
```cairo, noplayground
{{#include ../listings/ch10-testing-cairo-programs/no_listing_13_single_integration_crate/tests/lib.cairo}}
```
The project directory will now look like this :
```shell
adder
% % %  Scarb.lock
% % %  Scarb.toml
% % %  src
%    % % %  lib.cairo
% % %  tests
    % % %  common.cairo
    % % %  integration_tests.cairo
    % % %  lib.cairo
```
When we run the `scarb test` command again, here is the output :
```shell
{{#include ../listings/ch10-testing-cairo-programs/no_listing_13_single_integration_crate/output.txt}}
```
This way, only the test functions will be tested and the `setup` function can be imported without being tested.

Summary
Cairo'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 Cairo'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.
{{#quiz ../quizzes/ch10-02-testing-organization.toml}}