Testing

Learn to Code with Rust / Section Review

Testing

- Tests validate that the source code works as expected.
- An **assertion** is a verification that a statement is valid.
- Tests identify regressions in the code. A **regression** is a bug introduced into working software.

Tests in Rust

- A test is a plain Rust function annotated with the **#[test]** attribute.
- An attribute is metadata that changes how the compiler parses the code that follows.
- Our test involve some use of the source code along with a hardcoded expected value.
- Run **cargo test** to execute the full test suite. Cargo will run the tests in parallel.

Assertions

- The **assert_eq** macro validates that two values are equal. The values must implement the **PartialEq** trait.
- The **assert_ne** macro validates that two values are not equal.
- The **assert** macro validates that its argument is true.
- The final optional argument to any assertion is a custom error message in case of test failure.
- The **should_panic** attribute validates that the test code *causes* a panic.

Test Setup

- In Rust, unit tests are written alongside the implementation code.
- Nest the tests within a **tests** module to distinguish it from the source code. The module name is technically arbitrary.
- Add the **#[cfg(test)]** attribute to tell the compiler to exclude the tests when building the main executable.
- Use **super::*** to bring in all names from the outer scope into the module's scope.

Test Output

- **cargo test** will summarize the passing and failing tests.
- For a failing equality test, the test output will show the two values side by side.
- The two arguments are called left and right.

Testing Crates I

- Add testing crates to the **[dev-dependencies]** section of the **Cargo.toml** file.
- The **pretty_assertions** crate provides assertions that makes it easier to parse the differences between unequal values.
- Use the **use** keyword inside the **tests** module to bring the crate's **assert_eq!** macro into scope, overriding Rust's default macro.

Testing Crates II

- The **rstest** crate assists with creating fixtures, preconfigured instances of the type.
- Annotate a fixture function with **#[fixture]** and a test function with **#[rstest]**.
- Provide the fixture function name as a parameter to either a test or another fixture function.

Using Result Enum In Tests

- The **Result** type models an evaluation that could be either successful or erroneous which makes it a great fit for tests.
- Declare the return type of a test function as a **Result**. Provide types for the two generics.
- Return an **Ok** to indicate a passing test. If there is nothing of value to return, a unit is a common data type to package within the variant.
- Return an **Err** to indicate a failing test. The error message will be printed in the output.

Integration Tests

- A **unit test** targets a small, independent chunk of value (a function, a method, etc).
- An **integration test** tests a larger feature or the interaction of multiple units within the system.
- Write tests in plain **.rs** files within a top-level **tests/** directory.
- In the test files, there is no need for a **tests** module.
- Add the **pub** keyword for constructs in the source code to enable them to be pulled into tests.

Documentation Tests

- A **documentation test** is written using documentation comments (**///**).
- Use **#** to create a section header.
- Use ~~~ to mark the beginning and end of a code sample. The code has access to assertions.
- Run **cargo test** to run the documentation tests.
- Run **cargo doc** to generate browser documentation from the comments.

Dependency Injection

- **Dependency injection** is a design pattern where a type receives its dependencies from the "outside" world instead of creating them itself.
- Hardcoding a specific dependency in a constructor function couples the type to that dependency.
- Use **traits** to both enforce consistency and constraints on injected types.
- Dependency injection simplifies testing because tests can pass in dummies/replacements for the real-world dependencies.

Test Options

- Provide a search term after **cargo test** to filter tests by description.
- Add **-- --show-output** to show the standard output (println!) from passing tests. By default, Cargo will print content from failing tests.
- Execute **cargo test --lib** to run only library tests. This will ignore integration tests.
- Execute **cargo test --test '*'** to run only integration tests.

Test-Driven Development

- Test-driven development (TDD) is a testing paradigm that encourages writing tests first.
- TDD follows a red-green-refactor cycle. Write a failing test, make it pass, then improve the code.
- Advantages of TDD include good test coverage, better test design, better implementation, the simplification of complexity, and more.