CoE 164

Computing Platforms

04c: Rust Unit Testing

TESTS

Testing software code is an important skill to make sure programs work as expected by the programmer.

Rust provides has a basic built-in capability for writing **unit tests**, which are tests on a specific function or module.

TESTS: ANATOMY

Rust follows the following three steps in order when running a unit test:

- Set-up any needed data or state
- Run the code to test
- Assert the expected results

TESTS: BASICS

A **test** is written as a function annotated with the `test` attribute.

Several test functions can be grouped into a module which should be annotated with the `cfg(test)` attribute.

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

Example

TESTS: BASICS

A single test file can be run by first compiling the file using `rustc` with the `--test` flag. An executable file will then be generated, which will be run in a special "test mode" that shows the number of tests passed and failed, and a list and details of each of the failed tests.

```
PS D:\Users\[redacted]\Documents\UPDClasses\CoE164\2324s2\sample_codes> rustc .\test_basic.rs --test
PS D:\Users\[redacted]\Documents\UPDClasses\CoE164\2324s2\sample_codes> .\test_basic.exe

running 2 tests
test sanity_test::it_works ... ok
test sanity_test::it_panics - should panic ... ok

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

ASSERT: MACRO

The `assert!` macro panics if the expression in its first parameter is false. This is used to assert truth or correct behavior.

```
// Everything below will not panic
assert!(true);
assert!(2 + 2 == 4);
assert!(9 > 5);

// Everything below will panic
assert!(2 > 7);
```

Example

ASSERT MACRO: EQUALITY

The specific versions `assert_eq!` and `assert_neq!` assert whether the value of the two parameters entered are equal or not, respectively. These provide more details about test failure, such as the final values of the two parameters.

```
let x = 3 + 5;

// Does not fail
assert_eq!(x, 8);
assert_ne!(x, 0);

// Fails
assert_eq!(x, 7);
```

Example

ASSERT MACRO: MESSAGES

The `assert!` macros can print custom messages when they fail. The remaining arguments of the macros are passed to the `format!` macro.

```
// Will fail with a message
// Math failed or wrong assertion: 3 + 5 = 8
let x = 3 + 5;
assert_eq!(x, 7, "Math failed or wrong assertion: 3 + 5 = {}", x);
```

Example

ASSERT: PANIC

The `should_panic` attribute should be added to a test function if the code inside it *should* panic.

Note that *any* kind of panic will lead to a test pass.

```
#[cfg(test)]
mod tests {
    #[test]
    #[should_panic]
    fn it_panics() {
        panic!("Hi!");
    }
}
```

Example

ASSERT: PANIC

The `should_panic` attribute accepts an `expected` parameter to filter in messages that contain that substring. Otherwise, the test will fail.

Note that the substring is case-sensitive.

```
#[cfg(test)]
mod tests {
    #[test]
    #[should_panic(expected="Math failed")]
    fn it_fails_panic_expect() {
        let x = 3 + 5;

        assert_eq!(x, 7, "Math failed or wrong assertion: 3 + 5 = {}", x);
    }
}
```

Example

ASSERT: RESULT ENUM

Alternatively, instead of using `assert!` macros, test functions can return a `Result` enum to flag success or failure. The `Ok` variant should either be an empty tuple or one that returns an `ExitCode` struct.

Note that the `should_panic` attribute cannot be used on such test functions.

Example

```
#[test]
fn it_fails_result() -> Result <(),
String> {
    let d = String::from("hello!");

    if let Ok(_) = d.parse::<u64>() {
        Ok(())
    }
    else {
        Err(String::from("Cannot parse
string to u64"))
    }
}
```

TESTS: IGNORE

All tests are run by default. If a test may take some time to run and should not be run by default unless explicitly stated, the `ignore` attribute can be added to the test function.

```
#[test]
#[ignore]
#[should_panic]
fn it_panics_ignore() {
    panic!("Hi!");
}
```

Example

TESTS: IGNORE

Ignored tests can be run by adding the `--ignored` flag to the test executable compiled using `rustc`, or running `cargo test -- --ignored` inside a package.

```
PS D:\Users\[redacted]\Documents\UPDClasses\CoE164\2324s2\sample_codes> .\test_basic.exe --ignored

running 1 test
test assert_basic::it_panics_ignore - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 8 filtered out; finished in 0.00s
```

TESTS: PACKAGES

Tests in a package can be run by running `cargo test`. Cargo will then compile the package and run all available tests.

```
PS D:\Users\[redacted]\Documents\UPDClasses\CoE164\2324s2\sample_codes\complex_nums> cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.01s
     Running unittests src\main.rs (target\debug\deps\complex_nums-d1e8a03ffb4a660e.exe)

running 2 tests
test tests::it_sets_ma ... ok
test tests::it_sets_ri ... ok

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

TESTS: UNIT TESTS

By convention, **unit tests** are written in the same file as the code they are testing - just inside a module with the `cfg(test)` attribute.

Testing private functions is possible by `use`ing the `super` keyword inside the test module.

Example

```
fn main() {
    // Main code here...
}

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

    #[test]
    fn it_sets_ri() {
        let ri = ComplexRI::new(1.0, 2.0);

        assert_eq!(ri.real(), 1.0);
        assert_eq!(ri.imag(), 2.0);
    }

    // Other tests here...
}
```

TESTS: INTEGRATION TESTS

By convention, integration tests are placed in a separate folder in the same level as `src`. This means that these tests are their own separate crate.

Integration tests test for functionality of source code when combined together. They are usually created for library crates.

Example

```
complex_nums
    | src
    |   | main.rs
    | .gitignore
    | Cargo.toml
    | tests
    |   | i_test.rs
```

TESTS: INTEGRATED TESTS

Note that tests inside the integrated test folder are *not* enclosed in modules annotated with the `cfg(test)` attribute. However, the test functions are still annotated with the `test` attribute.

```
#[test]
fn it_sets_ri() {
    let ri = ComplexRI::new(1.0, 2.0);

    assert_eq!(ri.real(), 1.0);
    assert_eq!(ri.imag(), 2.0);
}
```

Example

RESOURCES

- The Rust Book

CoE 164

Computing Platforms

04c: Rust Unit Testing