Rust Fundamentals

Basics of Rust
Part 7

AZZAM S.A

RustCourse
Nov. 10th, 2023

Azzam S.A

OSS devotee, speaker, and teacher.

Open sourceror. Namely Rust, Python, and Emacs.

azzamsa azzamsyawqi azzamsa.com

Follow along!

- Rust Playground
- Exercises

▶ Show hints

Error Handling

Error Handling

- Rust groups errors into two major categories: *recoverable* and *unrecoverable errors*.
- Rust doesn't have exceptions. Instead, it has the type `` `Result\<T, E\>` `` and `` `panic!` ``.

Unrecoverable Errors with `` `panic!` ``

- *Unwinding the stack* vs *aborting*.
- No *buffer overread* in Rust. It stop and refuse to continue.
- Use `` `RUST_BACKTRACE ` ``environment variable to get a backtrace.
- Debug symbols are enabled by default, unless built with the `` `--release` `` flag.

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
99', src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Recoverable Errors with Result

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- In the case where `` `File::open` `` succeeds ➔an instance of `` `Ok` `` that contains a file handle.
- In the case where it fails ➔an instance of `` `Err` `` that contains more information about the kind of error that occurred.

```rust
fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => {
            panic!("Problem opening the file: {:?}", error);
        }
    };
}
```

Alternatives

- Using `if`-`else`
- `unwrap` and `expect`

```rust
fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

Using `expect` instead of `unwrap` and providing good error messages can convey your intent and make tracking down the source of a panic easier.

```rust
fn main() {
    let greeting_file = File::open("hello.txt")
        .expect("hello.txt should be included in this project");
}
```

In production enviroment, use `expect` to get give more context about why the operation is expected to always succeed.

Propagating Errors

- Returns the error to the caller.

```rust
fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

A Shortcut for Propagating Errors

- The `` `?` `` Operator
- Unlike `` `match` ``, `` `?` `` goes though the `` `from` `` funcion in `` `From` `` trait.

```rust
fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

Where we can use the `` `?` `` operator

- As with the `` `Result` ``, we can use `` `?` `` with `` `Option<T>` `` as long the function returns `` `Options` ``.

```rust
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

- You can't mix and match `` `?` `` in `` `Result` `` and `` `Option` ``.
- The `` `?` `` operator won't automatically convert a `` `Result` `` to an `` `Option` `` or vice versa; use `` `ok` `` method on Result or the `` `ok_or` `` method on Option to do the conversion explicitly.

Executable return values

```rust
fn main() -> Result<(), Box<dyn Error>> {
    let greeting_file = File::open("hello.txt")?;
    Ok(())
}
```

- If a main function returns a `` `Result<(), E>` ``, the executable will exit with `` `0` `` or nonzero value.
- Rust follows C convention in this case.

To `panic!` vs Not to `panic!`

- `panic!`
    - Examples, prototype code, and tests
    - You Have More Information Than the Compiler

- Using robust error-handling code can make the example and the target concept less clear.

- `unwrap` and `expect` act as clear markers in prototype, before you're ready to decide how to handle errors.

```rust
let home: IpAddr = "127.0.0.1"
    .parse()
    .expect("Hardcoded IP address should be valid");
```

Guidelines

- `` `panic!` ``
  - The program end up in bad state.
  - Something unexpected, not something that will likely happen occasionally.
  - Harmful or insecure such out-of-bounds memory access.
- Not `` `panic!` ``
  - Occasional error.
  - Bad HTTP request, malformed input for parser.

Relies as much as possible to the Rust type system, such as missing arguments, and negative values.

Creating Custom Types for Validation

Avoid performance penalty by not doing too much checking.

```rust
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 }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

Use public method to access `value` to prevent setting a value directly.

```rust
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 }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

Testing

- Strong type is not enough

```
fn add_ten(x: i32) -> i32 {
    x + 10
}
```

The test function anatomy:

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

- Test module can contain non-test function.
- It’s possible to mark a test as ignored so it doesn’t run in a particular instance
- Cargo able run specific test. Test filtering.
- Doc-tests helps keep your docs and your code in sync.

```rust
/// Shortens a string to the given length.
///
/// ```
/// use playground::shorten_string;
/// assert_eq!(shorten_string("Hello World", 5), "Hello");
/// assert_eq!(shorten_string("Hello World", 20), "Hello World");
/// ```
pub fn shorten_string(s: &str, length: usize) -> &str {
    &s[..std::cmp::min(length, s.len())]
}
```

```
$ cargo test

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
```

Checking Test Results

- Using `` `assert!` `` macro for boolean condition.
- `` `assert_eq!` ``, and `` `assert_ne!` `` for testing equality.
- `` `assert_eq!(<left>, <right>)` ``. expected <=> actual.
- These macros print using debug formatting, means the values being compared must implement `` `PartialEq` `` and `` `Debug` `` traits.
- Can have custom failure messages.

```rust
#[test]
fn test_empty() {
    assert_eq!(first_word(""), "");
}
```

```rust
assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was `{result}`"
    );
```

Checking Test Results

- Use `` `should_panic` `` to check panics.

```
#[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
```

- Make it more precise with `` `expected` ``.

```
#[test]
    #[should_panic(expected = "less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
```

Using `Result<T, E>` in Tests

- Gives the ability to use `?` inside tests.

```rust
#[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"))
        }
    }
}
```

More...

- ``cargo test -- --test-threads=1``
- ``cargo test -- --show-output``
- ``cargo test <test name>``
- Ignoring some tests unless specifically requested.

```rust
#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

Integration tests

Create a `` `.rs` `` file under `` `tests/` ``:

```rust
use my_library::init;

#[test]
fn test_init() {
    assert!(init().is_ok());
}
```

Credits 🌟

- Mo’s (mo8it) Comprehensive Rust 🦀
- rustlings 🦀