:fire: Rust Programming Tipz :fire:

A collection of software engineering techniques for effectively expressing intent with Rust.

Cleanup

This section is about improving clarity.

Combating Rightward Pressure

After sparring with the compiler, it's not unusual to stand back and see several nested combinator chains or match statements. Much of the art of writing clean Rust has to do with the judicious application of de-nesting techniques.

Basics of De-nesting

- Use `?` to flatten error handling, but be careful not to convert errors into top-level enums unless it makes sense to handle them at the same point in your code. Keep separate concerns in separate types.
- Split combinator chains apart when they grow beyond one line. Assign

useful names to the intermediate steps. In many cases, a multi-line combinator chain can be more clearly rewritten as a for-loop.
- pattern match on the full complex type instead of using nested match statements
- If your match statement only has a single pattern that you care about, followed by a wildcard, replace the match statement with an `if let My(Match(Pattern(thing))) = matched_thing { /*...*/ }` possibly with an `else` branch if you cared about the wildcard earlier.
- Run cargo clippy! It can provide many legitimately helpful suggestions for cleaning up your code

Tuple Matching

If you find yourself writing code that looks like:

```
let a = Some(5);
let b = Some(false);

let c = match a {
    Some(a) => {
        match b {
            Some(b) => whatever,
            None => other_thing,
        }
    }
    None => {
        match b {
            Some(b) => another_thing,
            None => a_fourth_thing,
        }
    }
};
```

it can be de-nested by doing a tuple match:

```
let a = Some(5);
let b = Some(false);

let c = match (a, b) {
    (Some(a), Some(b)) => whatever,
    (Some(a), None) => other_thing,
    (None, Some(b)) => another_thing,
    (None, None) => a_fourth_thing,
};
```

As a special case, matching on tuples of booleans can be used to encode decision tables. For example, here's roughly how `cargo new` handles `--bin` and `--lib` arguments:

```
let kind = match (args.is_present("bin"), args.is_present("lib")) {
    (true, true) => failure::bail!("can't specify both lib and binary outputs"),
    (false, true) => NewProjectKind::Lib,
    // default to bin
    (_, false) => NewProjectKind::Bin,
};
```

Iteration Issues

Pulling the First Error out of an Iterator over `Results`

The `collect` method is extremely powerful, and if you have an iterator of `Result` types, you can use it to either return a collection of the `Ok` items, or the very first `Err` item.

From the std docs on collect:

```
let results = [Ok(1), Err("nope"), Ok(3), Err("bad")];

let result: Result<Vec<_>, &str> = results.iter().cloned().collect();

// gives us the first error
assert_eq!(Err("nope"), result);

let results = [Ok(1), Ok(3)];

let result: Result<Vec<_>, &str> = results.iter().cloned().collect();

// gives us the list of answers
assert_eq!(Ok(vec![1, 3]), result);
```

Seen in Sunjay's tweet

This functionality is unlocked by the `Result` type implementing `FromIterator<Result<A, E>> for Result<V, E>` where `V` implements `FromIterator<A>`. This may look a bit hairy, but it means that `collect` (which relies on the `FromIterator` trait for its functionality) can output a `Result` where the success type is some collection that can be built from `A` - the success type of the original results. The error type `E` is the same in both, meaning that the returned `Result` will just return the first encountered error.

Reverse Iterator Ranges

In Rust, we can write `for item in 0..50` to go from 0 to 49 but what if we wanted to iterate from 49 to 0? Many of us have written `for item in 50..0` and been surprised that nothing happened. Instead, we can write:

```
// iterate from 49 to 0
for item in (0..50).rev() {}
```

```rust
// iterate from 50 to 0
for item in (0..=50).rev() {}

// iterate from 50 to 1
for item in (1..=50).rev() {}
```

Under the hood, when we write a range with this syntax, we are constructing a `RangeInclusive` instead of the normal `Range`. You can also construct ranges for everything with `..`, or have a range be half-open like `..50` or `..=50` or `0...`

Seen in Andrea Pessino's tweet

Empty and Singular Iterators

The standard library also includes helpers for empty and singular iterators, using the functions `std::iter::empty` and `std::iter::once`, which can be a small cleanup of common code like `vec![].into_iter()` or `vec![my_item].into_iter()`.

Tuple Structs and Enum Tuple Variants as Functions

You may have received an error message at some point when you wrote an enum variant, but not the members inside it, and it complained about how you supplied a function instead of an enum:

```rust
enum E {
    A(u64),
}

// ERROR: expected enum `E`, found `fn(u64) -> E {E::A}`
let a: E = E::A;
```

Well, it turns out that enum tuple variants as well as tuple structs can be used as functions from their members to an instance of that enum or struct. This can be used to encapsulate items in a collection inside that object:

```rust
// create a vector of E::A's using the variant as a constructor function
let v_of_es: Vec<E> = (0..50).map(E::A).collect();

// v_of_es is now vec![A(0), A(1), A(2), A(3), A(4), ..]

// create a vector of Options using Some as a constructor function
let v_of_options: Vec<Option<u64>> = (0..50).map(Some).collect();

struct B(u64);

// create a vector of B's using the struct as a constructor function
let v_of_bs: Vec<B> = (0..50).map(B).collect();
```

Blocks for Clarity

Blocks allow us to detangle complex expressions, and can be used anywhere that a one-liner expression would be valid.

Closure Capture

Specifying variables for use in a closure can be frustrating, and it's common to see code that jumps through hoops to avoid shadowing variables. This is quite common when cloning an `Arc` before spawning a new thread that will own it. But a closure definition is an expression. Anywhere a closure is accepted, we could use a block that evaluates to a closure. In the example below, we use blocks to avoid shadowing the config that we want to pass to several threads, without creating gross names like `config1`, `config2` etc… Seen in salsa and described in more detail in Rust pattern: Precise closure capture clauses.

Before, painfully avoiding shadowing config:

```
fn spawn_threads(config: Arc<Config>) {
    let config1 = Arc::clone(&config);
    thread::spawn(move || do_x(config1));

    let config2 = Arc::clone(&config);
    thread::spawn(move || do_y(config2));
}
```

After, no need to invent config_n names:

```
fn spawn_threads(config: Arc<Config>) {
    thread::spawn({
        let config = Arc::clone(&config);
        move || do_x(config)
    });

    thread::spawn({
        let config = Arc::clone(&config);
        move || do_y(config)
    });
}
```

Ergonomics

One of the most important aspects of feeling productive with the Rust programming language is to find harmony with the compiler. We've all introduced a single error and been whipped in the face by dozens of error messages. Even after years of professional Rust usage, it can feel like a cause for celebration when there are no errors after introducing more than a few new lines of code.

Remember that the strictness of the compiler is what gives us so much freedom. Rust is useful for building back-ends, front-ends, embedded systems, databases, and so much more because the compiler knows how long our variables are valid for without using a garbage collector at runtime. Any lifetime-related bug that fails to compile in Rust might have been an exploitable memory corruption issue in C or C++. The compiler pain frees us from exploitation and gives us the ability to work on a wider range of projects.

Unification and Reading the Error Messages That Matter

Rust requires that arguments and return types are made explicit in function definitions. The compiler will use these explicit types at the boundaries of a function to drive type inference. It will take the input argument types and work from the top of the function toward the bottom. It will take the return type and work its way up. Hopefully they can meet in the middle. The process under the hood is actually a little more complicated than this but this simplified model is adequate to reason about this particular subject. The point is, there has to be an unbroken chain of type evidence that connects the input arguments to the return type through the body. When there is a gap in the chain, all ambiguous types will turn into errors. This is partially why rust will emit many pages of errors sometimes when there's actually only a single thing that needs to be fixed.

A big part of avoiding compiler fatigue is to just filter out the errors that don't matter. Start with the first one, and work your way down. See the next section for a command that will do this automatically when your code changes.

Write-Compile-Fix Loop Latency

Programming Rust is a long game. It's common to see beginners spending lots of energy switching back and forth between their editor and a terminal to run rustc, and then scrolling around to find the next error that they want to fix. This is high-friction, and will tire you out faster than if it were automated.

There is a cargo plugin called `cargo watch` that will look for changes in source files descendent from the current working directory, and then run `cargo check` which skips the LLVM codegen and only looks for compilation errors in your Rust code. It can be installed by typing `cargo install cargo-watch`.

You can use the `cargo watch` plugin to call a specific command when your code changes as well. I like to filter out the lines after the beginning of the error messages, after clearing the terminal:

```
cargo watch -s 'clear; cargo check --tests --color=always 2>&1 | head -40'
```

This way I just save my code and it shows the next error.

Caching with sccache

`sccache` is a tool written by Mozilla that supports `ccache`-style build caching for Rust. This is particularly useful if you frequently clean and build projects with lots of dependencies, as normally they would all need to be recompiled, but with `sccache` they will be stored and used to back a cache that is accessible while building any project on your system. It takes care to do the right thing in the presence of different feature flags, versions etc… For projects with lots of dependencies it can make a huge difference over time.

The installation is simple:

```
cargo install sccache
export RUSTC_WRAPPER=sccache
```

It can also be configured to use a remote cache like s3, memcached, redis, etc… which is quite useful for building speedy CI clusters.

Editor support for jumping to compiler errors

To go even farther than the last section, most editors have support for jumping to the next Rust error. In vim, you can use the `vim.rust` plugin in combination with Syntastic to automatically run rustc when you save a file, and to jump to errors using a keybind. Emacs users can use `flycheck-rust` for similar functionality.

Lockdown

This section is about preventing undesirable usage.

Never

To make a type that can never be created, simply create an empty enum. Use this where you want to represent something that should never actually exist, but a placeholder is required. This is being brought into the standard library piece by piece, and it's already possible to have a function return `!` if it exits the process or ends in an infinite loop (important for embedded work where main should never return). This can also be used when working with a Result that will never actually be an `Err` but you need to adhere to that interface.

```
enum Never {}

let never = Never:: // oh yeah, can't actually create one...
```

Deactivating Mutability

Here's a pattern for disabling mutability for "finalized" objects, even in mutable owned copies of a thing, preventing misuse. Done by wrapping it in a newtype

with a private inner value that implements Deref but not DerefMut:

```
mod config {
    #[derive(Clone, Debug, PartialOrd, Ord, Eq, PartialEq)]
    pub struct Immutable<T>(T);

    impl<T> Copy for Immutable<T> where T: Copy {}

    impl<T> std::ops::Deref for Immutable<T> {
        type Target = T;

        fn deref(&self) -> &T {
            &self.0
        }
    }

    #[derive(Default)]
    pub struct Config {
        pub a: usize,
        pub b: String,
    }

    impl Config {
        pub fn build(self) -> Immutable<Config> {
            Immutable(self)
        }
    }
}

use config::Config;

fn main() {
    let mut under_construction = Config {
        a: 5,
        b: "yo".into(),
    };

    under_construction.a = 6;

    let finalized = under_construction.build();

    // at this point, you can make tons of copies,
    // and even if somebody has an owned local version,
    // they won't be able to accidentally change some
    // configuration that
    println!("finalized.a: {}", finalized.a);
```

```
    let mut finalized = finalized;

    // the below WON'T work bwahahaha
    // finalized.a = 666;
    // finalized.0.a = 666;
}
```

Avoiding Limitations

`Box<FnOnce>`

Obsolete since Rust 1.35 (release notes), `Box<dyn FnOnce>` just works now.

Currently, it's not possible to call `Box<FnOnce(T) -> R>` on stable Rust. The common workaround is to use `Box<FnMut(T) -> R>`, store internal state inside of an `Option` and `take` the state out (with a potential run-time panic) in the call. However, a solution that statically guarantees that fn can be called at most once is possible. Seen in Cargo.

```
trait FnBox<A, R> {
    fn call_box(self: Box<Self>, a: A) -> R;
}

impl<A, R, F: FnOnce(A) -> R> FnBox<A, R> for F {
    fn call_box(self: Box<F>, a: A) -> R {
        (*self)(a)
    }
}

fn demo(f: Box<dyn FnBox<(), String>>) -> String {
    f.call_box(())
}

#[test]
fn test_demo() {
    let hello = "hello".to_string();
    let f: Box<dyn FnBox<(), String>> = Box::new(move |()| hello);
    assert_eq!(&demo(f), "hello");
}
```

Note that `self: Box<Self>` is stable and object-safe.

Shared Reference Swap Trick

`std::cell::Cell` is most commonly used with `Copy` types, because `Cell::<T>::get` method requires `T: Copy`. However, a `Cell` can be use-

ful with non-copy types as well, thanks to these two methods:

```
fn replace(&self, val: T) -> T;

fn take(&self) -> T
where
    T: Default
;
```

In particular, using a `Cell<T>` one can implement an analogue of `std::mem::swap` (swap trick) or `std::mem::replace` (Jones's trick) which doesn't need a `&mut` reference.

The following example uses `Cell::take` to implement `fmt::Display` for the iterator. `fmt::Display` has only `&self`, but we need to consume the iterator to print it. `Cell` allows us to do exactly that:

```
use std::{cell::Cell, fmt};

fn display_iter<I>(xs: I) -> impl fmt::Display
where
    I: Iterator,
    I::Item: fmt::Display,
{
    struct IterFmt<I>(Cell<Option<I>>);

    impl<I> fmt::Display for IterFmt<I>
    where
        I: Iterator,
        I::Item: fmt::Display,
    {
        fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
            // Advanced Jones's trick: `mem::replace` with `&` reference!
            let xs: Option<I> = self.0.take();
            let xs: I = xs.unwrap();

            let mut first = true;
            for item in xs {
                if !first {
                    f.write_str(", ")?
                }
                first = false;
                fmt::Display::fmt(&item, f)?
            }

            Ok(())
        }
    }
```

```
    IterFmt(Cell::new(Some(xs)))
}

fn main() {
    let xs = vec![1, 2, 3].into_iter();
    assert_eq!(display_iter(xs).to_string(), "1, 2, 3");
}
```

First seen in rustc.

Using Sets As Maps

Often, you want to store structs in an associative container keyed by a field of the struct. For example

```
use std::collections::HashMap;

struct Person {
    name: String,
    age: u32,
}

fn persons_by_name(persons: Vec<Person>) -> HashMap<String, Person> {
    persons.into_iter()
        .map(|p| (p.name.clone(), p))
        .collect()
}
```

The simplest way to do this is to clone the keys, like in the example above. The drawback of this approach is that the container contains two copies of each key, which might be a problem if the keys are large or are not `Clone`.

It seems like it should be possible to borrow the key from the struct, but doing this in a straight forward way fails:

```
fn persons_by_name(persons: Vec<Person>) -> HashMap<'??? str, Person>
```

There isn't any lifetime in sight you can use instead of `???`.

However, it is possible to use a `Set` instead of a `Map` to get a similar effect:

```
use std::{
    borrow::Borrow,
    collections::HashSet,
    hash::{Hash, Hasher},
};

struct Person {
    name: String,
```

```
    age: u32,
}

impl Borrow<str> for Person {
    fn borrow(&self) -> &str {
        &self.name
    }
}

impl PartialEq for Person {
    fn eq(&self, other: &Person) -> bool {
        self.name == other.name
    }
}

impl Eq for Person {}

impl Hash for Person {
    fn hash<H: Hasher>(&self, hasher: &mut H) {
        self.name.hash(hasher)
    }
}

fn persons_by_name(persons: Vec<Person>) -> HashSet<Person> {
    persons.into_iter().collect()
}

fn get_person_by_name<'p>(persons: &'p HashSet<Person>, name: &str) -> Option<&'p Person> {
    persons.get(name)
}
```

The crux of the trick here is that Rust sets and maps use the `Borrow` trait. It allows lookup operations by types different than those stored in the container. In this example, by implementing `Borrow<str> for Person`, we get the ability to get a `&Person` out of `&HashSet<Person>` by `&str`.

Note that, because we implement `Borrow`, we **must** override `Eq` and `Hash` to be consistent with it. That is, we compare persons by looking only at the name, and ignore the age. This is likely not what you want for the rest of your application, so you might need to additionally create an `struct PersonByName(Person)` wrapper type and implement `Borrow` for that.

Sources: the original pattern overheard from @elizarov in the context of Kotlin, Rust's stdlib docs, recent conversation on r/rust.