Miscellaneous Cool Stuff

“The problem with going faster than light is that you live in darkness”

Jack Duvall

Carnegie Mellon University

Course Summary

- Structs, Enums, Pattern matching, Traits
- Function Types, Ownership, Borrowing
- Polymorphism
- Crates and Modules
- Standard Library
- Error handling, Testing
- Macros
- Unsafe
- Parallelism (threads), Concurrency (async/await)

Outline

Outline

Why Would You Want To Do This?

(Recall: sync code can be made async with `tokio::spawn_blocking`)

- Having your top-level function be `async` isn't the best, sometimes you want to architecture your own event loop for GUI things

Before, we just used the `#[tokio::main]` macro. What does that expand do/can we do it ourselves?

Doing What #[tokio::main] Does

```
fn main() {
    tokio::runtime::Builder::new_multi_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async {
            println!("Hello world");
        })
}
```

Manual Expansion Gives More Power

- Change parameters of the runtime.
- Spawn multiple futures onto runtime at once, without `join!`
- Run futures ”in background”, while running other sync code

See The Tokio Docs for lots of code examples

Outline

You Can't Have `async fn` In Traits (right now)

```rust
trait Webserver {
    async fn handle(&self, r: Request) -> Response;
}
```

Too bad Rust doesn't like this... Why?

- Short Answer: `async fn` only guarantees a trait, not a type
- Long Answer: mostly stolen from Niko Matsakis' Blog

`async fn` Is Syntatic Sugar For This

```rust
trait Webserver {
    fn handle(&self, r: Request) ->
        impl Future<Output = Response> + '_;
}
```

...roughly speaking, that is

It Gets Funkier

```rust
trait Webserver {
    type HandleFuture<'a>: Future<Output = Response> + 'a;
    fn handle(&'a self, r: Request) -> Self::HandleFuture<'a>;
}
```

This is a “Generic Associated Type”, not supported in Rust yet, no concrete plans

More Unresolved Questions

Even if GATs were solved, what if you wanted to constrain futures returned by an implementation?

```rust
fn launch_on_multiple_threads<W>(webserver: W)
where for<'a> W::HandleFuture<'a>: Send
{
    // `Send` lets us share futures returned by
    // `webserver.handle(r)` between threads
}
```

- We needed to know the name of the associated type. Is it auto-generated? Or do people need to desugar manually?
- If you use a lot of futures, there's a lot more `Send` bounds you need; is there a better way to combine them all?

Even More Considerations

If you use regular generics, many copies of code are made. Could be better to force the use of trait objects:

```
trait Webserver {
    fn handle(&self, r: Request) ->
        dyn Future<Output = Response> + '_;
}
```

New problem: now the return type isn't `Sized` (don't know the size at compile time), so we can't generate code! Need a wrapper, but how to choose between `Box`, `Arc`, others?

A Good Enough Solution: `async-trait` Crate

Applying `#[async_trait]` to the original trait with an `async fn` results in the following desugaring:

```
trait Webserver {
    fn handle(&self, r: Request) ->
        Pin<Box<dyn Future<Output=Response> + Send + '_>>;
}
```

mm delicous type + trait soup

Outline

What's In The Standard Library?

- Option, Result, integer types
- Iterator, Default, From
- String, Vec, Box, Arc, Allocation
- Platform Intrinsics (`std::arch::x86_64::__cpuid`)
- Filesystem abstractions
- Futures
- HashMap
- Networking
- Threads

Rust Can Be A “Portable Assembler” Too!

It is possible to write code without the standard library! Really, it’s split into two parts:

- `std`: All the “fancy” stuff
- `core`: Only the stuff you actually need to write Rust

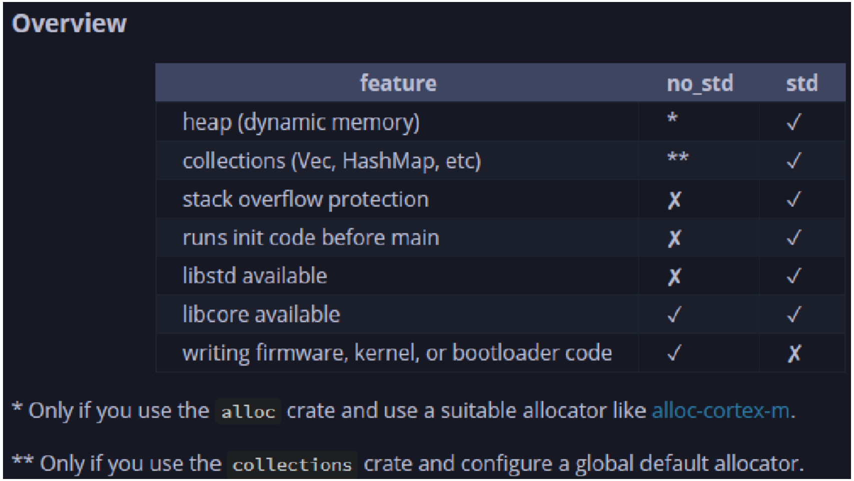
What's In The Core Library?

- Option, Result, integer types
- Iterator, Default, From
- Platform Intrinsics
- Futures
- Types + traits for everything else that makes sense
  - Hash trait, no HashMap
  - Debug trait, no default formatter

Why Would You Ever Do This?

Embedded Systems!
Write An Operating System!
Write A Library People Would Use For The Above Projects!

This Table Puts It Nicely



Overview

feature	no_std	std
heap (dynamic memory)	*	✓
collections (Vec, HashMap, etc)	**	✓
stack overflow protection	✗	✓
runs init code before main	✗	✓
libstd available	✗	✓
libcore available	✓	✓
writing firmware, kernel, or bootloader code	✓	✗

* Only if you use the `alloc` crate and use a suitable allocator like alloc-cortex-m.

** Only if you use the `collections` crate and configure a global default allocator.

Image credit: https://docs.rust-embedded.org/book/intro/no-std.html

Outline

The Past, A Tale Of Woe

Before const generics: the array type `[T; N]` is the only type that can have a number in it!
Problem: Array methods weren't implemented for arrays larger than 32!
Why? Rust compiler didn't let us implement traits that were "generic over values", and traits are the only way to get associated functions!

Today, Living In The Future

We can have user-defined types and traits that are generic over integer values!

```rust
struct ArrayPair<T, const N: usize> {
    left: [T; N],
    right: [T; N],
}
impl<T: Debug, const N: usize> Debug for ArrayPair<T, N> {
    // ...
}
```

The Future's Future, Even Better

https://blog.rust-lang.org/inside-rust/2021/09/06/Splitting-const-generics.html

Outline

We Have Compile-Time Evaluation Too!

https://doc.rust-lang.org/reference/const_eval.html

const context: Array length, const/static/enum discriminant, const generic argument

Only certain compile-time-evaluable operations permitted inside these; turns out there's quite a lot! Besides the obvious, also have:

- Block expressions (`unsafe`), `if`, `match`
- Array indexing
- Borrows, dereferencing

and more!

`const fn` are functions made up of only these, enforced by the compiler, also callable from const contexts

A Bit Uncontrollable

- Not always guaranteed that const context or `const fn` will be evaluated at compile time, only that it can
- Plenty of strange rules
- `const fn` is restrictive in terms of ABI compatability: need to make sure function will stay `const` in the future

Fun With Rust's Type System!

It’s Turing Complete!

https://sdleffler.github.io/RustTypeSystemTuringComplete/

It's A Little Unsound!

https://counterexamples.org/eventually-nothing.html#eventually-nothing
https://counterexamples.org/nearly-universal.html#nearly-universal-quantification
https://counterexamples.org/mutable-matching.html#mutable-matching