Rust for "modern" C++ devs

it is 2022 so let's say C++17 - C++20 counts as modern

Graydon Hoare, March 2022

Overview

- Talk will have 3 parts
  - Orientation and introduction
  - Hard stuff: memory safety
  - Easy stuff: everything else

- I'm sorry my presentation style is just huge walls of text that I read
- Also very sorry there are 107 slides

Part 1

Orientation & Introduction

Orientation

Orientation

Who this talk is for

- Assuming you know C++ fairly well
  - At least well enough to be scared of it
- Will use some analogies to "modern" C++, newer features
  - C++11: `std::move`, `unique_ptr`, `shared_ptr`, `promise`/`future`, `mutex`/`scoped_lock`, `auto`, range-for loops, lambdas
  - C++17: `optional`, `variant`, `string_view`, `execution::par`
  - C++20: concepts, modules, ranges, coroutines
- You don't *have* to know *all* these things, it'll just help a bit

Orientation

Who is giving this talk: me, hello!

- Designed, implemented early Rust 16 years ago
  - Can't describe how old I feel saying that
- Worked on it for 7ish years, left project 8ish years ago
  - Amazingly, still 2 years before 1.0 shipped
- Used it for hobby work since, followed development
- Not 100% fond of everything, won't sugarcoat problems
  - Still think it's a good step up from C++, recommend it

Introduction

Introduction

A quick taste of Rust

Introduction

A quick taste of Rust

- Let's just look at a bit of random Rust code so you can see it.
- I will only put these on the screen for a few seconds.
- This is like an aesthetic-familiarity thing.
- So you are not shocked later.
- It tastes a little like C++.
- But also .. different.

Introduction

A quick taste of Rust

```
struct Point {
    x: i32,
    y: i32,
}
fn main() {
    let p = Point {x: 1, y: 2};
}
```

Introduction

A quick taste of Rust

```
let mut x: i32 = 5;
let mut done: bool = false;
while !done {
    x += x - 3;
    if x % 5 == 0 {
        done = true;
    }
}
```

Introduction

A quick taste of Rust

```
struct Circle {
   radius: f64,
}
trait HasArea {
    fn area(&self) -> f64;
}
impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

Introduction

A quick taste of Rust

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


let mut map = HashMap::new();
map.insert("a", 1);
map.insert("b", 2);
map.insert("c", 3);


for key in map.keys() {
    println!("{}", key);
}
```

Introduction

A quick taste of Rust

```
let v: Vec<i32> =
    vec![1, 2, 3]
        .into_iter()
        .map(|x| x + 1)
        .rev()
        .collect();

assert_eq!(v, [4, 3, 2]);
```

Introduction

What is good about Rust

Introduction

What is good about Rust

- The main selling feature: **Safety**
  - *Without* performance overheads
- Rust mostly prevents several **big**, **bad** classes of bugs
  - "Spatial" memory errors: out-of-bounds / wild / null pointers
  - "Temporal" memory errors: use-before-init, use-after-free
  - Multiple threads racing on same data
  - Pretty much all UB

Introduction

What is good about Rust

- Many minor fixes that prevent ubiquitous C++ bugs
  - Can't write dangling-else, switch-case-fallthrough, or goto
  - No silent int-float, signed-unsigned coercions
  - Booleans, enums, integers, pointers not mixable
  - Can't typo == as =
  - Strings guaranteed UTF-8, no random access
  - Culture of partial functions that return precise errors

Introduction

What is good about Rust

- It's also quite expressive
  - Sum types and pattern matching
  - Rich generic traits system
  - Extensive type inference
  - Macros and attributes for boilerplate

Introduction

What is good about Rust

- Slightly more building blocks than C++, but simpler and more orthogonal
  - Not everything jammed into `class` and `template`
  - Fewer things to remember to write "the way that doesn't explode"
  - Less implicit code, more C-like: data separate from code
    - Operator overloading did, unfortunately, make it in
    - But without conversion or copy-constructors, it's not *so* bad
- It is not a *small* language, but I think it's smaller than C++

Introduction

What is good about Rust

- It also has great, very standardized tooling
  - Configuration, building, testing
  - Packages, dependencies
  - Profiling, fuzzing
  - Documentation, code-formatting
  - A very good editor / IDE backend

Introduction

What is good about Rust

- The compiler has fantastic diagnostics
  - With standard, fine-grained control over checks
  - The IDE often gives you a push-button fix
  - You can opt-in to zillions of extras
  - Everything is *very* documented

Introduction

What is good about Rust

- You can use it where you use C++
  - Ahead-of-time static compilation
    - Everything specialized, inlined, LLVM-optimized, quite fast
  - Standard linkage to C/C++ programs
    - No significant "runtime" to embed
  - Lots and lots of C/C++ programs including bits of Rust now

Introduction

What is good about Rust

- There is an enormous package ecosystem
  - Turns out standard package tooling helps a lot
- Seriously, 77,136 crates as of today
  - *Likely* a package for anything you need
- Due to safety, orthogonality and standardization, packages often combine ok
  - Not always, but it's less worrisome than C++

Introduction

What is bad about Rust

Introduction

What is bad about Rust

- The language is somewhat inflexible
  - You have to "work with the grain of it"
  - It will fight you, stubbornly, if you do not
    - This is not an enjoyable fight
  - Especially if it involves shared memory
    - Or cycles in memory
    - Really, it *greatly* prefers trees

Introduction

What is bad about Rust

- It has high cognitive load
  - Lots to keep in mind when working
  - Lots of time spent with code not compiling
  - Lots to learn just to get basics working
  - Certain amount of "type tetris" busy-work, over-design
  - And also a new thing: "borrow checking"
    - (We will get to it later)

Introduction

What is bad about Rust

- The compiler is quite resource intensive
  - Buy a fast machine, really
  - People are working on improving this
  - Use a good IDE for fast, live errors
  - Remote dev works well: vscode on laptop, rust-analyzer on big machine
    - Workstation, remote cloud machine, etc.

Introduction

What is bad about Rust

- Some parts are clearly not done
  - Much of the async story is messy
  - Much of the error handling story too
  - A few bits of the stdlib are vestigial or bad ideas that stuck
  - Third party libraries are filling in as best they can
  - Keep in mind, C++20 still can't open a socket portably
    - Things take time

Introduction

What is bad about Rust

- The enormous package ecosystem is a mixed blessing
  - Easy to wind up with enormous mystery-meat *dependency trees*
    - Add 1 dependency, suddenly you're building 200 new packages
  - Many packages are immature or abandonware
  - Even among working options, so much choice means paralysis
    - Days spent browsing for the right library
    - Possibly faster to write your own

Part 2

Hard stuff: memory safety

Memory safety

Want to make sure we cover this while you are hopefully still awake

I'm sorry my slide style is so boring

Try to make it through this part

This is the somewhat high-tech bit

Memory safety
The main thing

- Main theme in design: *better-behaved pointers*
  - Steep 60% part of Rust learning curve is internalizing *pointer rules*
    - May seem arbitrary, tyrannical, cruel
    - Really just formalize "safe version of C++ idioms"
    - But no wiggle room: won't compile if violated
    - Once reflexive, fairly smooth sailing
  - Other 40% is much less steep, just "work"

Memory safety
The main thing

- C++ pointer-ish types have many *ok-ish* ingredients
  - Owners: `unique_ptr<T>` and `shared_ptr<T>`
  - Non-owning, transient refs: `&` and `const&`
  - Owner/transient split has advantages
    - Avoids most book-keeping, no tracing GC
    - Faster, more deterministic behaviour

Memory safety
The main thing

- But C++ pointer-ish types let you do terrible things!
  - `const&` can point to a mutable value, may change
    - I guess it's cool that it stops *you* from mutating?
  - `&` and `const&` can point to already-freed memory
    - Or something half-initialized, or half-destructed
  - Owner-pointers can be `nullptr`
  - Threads can race on all of them

Memory safety
The main thing

- Rust prohibits all of this nonsense
  - Eg. `Box<T>` is like `unique_ptr<T>` except
    - Has no null value, always points to a `T`
    - Is immutable if there's any live immutable `&`-reference
    - Is *inaccessible* if there's any live mutable `&mut`-reference
    - Can only *move* between variables, and not while `&`-referenced
    - Thus two threads cannot race on it (more on this later)

Memory safety
The main thing

- Rust has analogies for everything pointery you're used to in C++

C++	Rust
`unique_ptr<T>`	`Box<T>`
`shared_ptr<T>`	`Rc<T>` and `Arc<T>`
`weak_ptr<T>`	`Weak<T>`
`& T`	`&mut T`
`const& T`	`& T`
`* T`	`*mut T`
`const* T`	`*const T`

Memory safety
The main thing

- Those last two are for `unsafe`, which we're not going to discuss past here

C++	Rust
`* T`	`*mut T`
`const* T`	`*const T`

- Unsafe lets you break key rules, is used very carefully inside libraries
- Do not use it for the first 18 months -- you should rarely if ever need it

Memory safety
The main thing

- Two main (weird) ingredients to the rules
  - **Move semantics**
  - **Borrow checking**
- These two ingredients reinforce each other *remarkably well*
  - Commonly called Rust's "ownership system"

Move semantics

Show up in type systems as types called "substructural" or "affine" or "linear" or "resource" or "unique"

Let us set all this jargon aside and enjoy a photograph of a train

The finest way to move

Move semantics

Move is just copy + forget

- Move is just copy + forget
- Lets you make a stronger assumption: **uniqueness**
  - If exactly 1 copy before move, then exactly 1 after
  - No need to worry about accidental duplicates of things
  - Reference counts, multiple pointers to a value, multiple writers...
  - Unique "identity" from creation, through multiple moves, to destruction

Move semantics

Move is just copy + forget

- In C++ you have to do a bunch of work to *equip* a type with move semantics
  - Delete lval-ref copy constructor and assignment operator
  - Add rval-ref of same
  - Use `std::move` to turn one into the other

Move semantics

Move is just copy + forget

- Even then C++ keeps it dangerous
  - Rval-ref operators need to be written "just right" to move values
  - Leave behind a "moved from" husk-of-a-value, can still be used
  - Eg. moving from `unique_ptr` leaves behind a "moved from" `nullptr`

Move semantics

Move is just copy + forget

- In Rust move semantics are the normal case
  - Any non-primitive is moved when assigned or passed as argument
  - What you get (and all you get) if you define a new `struct`
  - Many types never implement anything besides this
  - Common to assume all you can do is move a type

Move semantics

Move is just copy + forget

- Can opt-in some POD types to be `Copy`
  - Allows trivial, implicit memcpy-duplications
  - But only if opted-in and guaranteed safe:
    - Nothing with mutable references or pointers that might own data
- Can also opt-in some types to have deep-`Clone`
  - Allows nontrivial, explicit `.clone()` for allocating-duplication
  - An attribute will "derive" (code-generate) this, no boilerplate needed

Move semantics

Move is just copy + forget

- Critically, in Rust when you move a value it's *gone*!
  - Statically
  - You can't use the place moved-from anymore
  - Local variable is de-initialized, can't be referenced anymore
  - Struct field can only be moved-out of by deconstructing the struct
    - Or swapped-in-place with some other value of same type

Move semantics

Move is just copy + forget

- This makes for some interesting (often helpfully safe) patterns
  - Moving a value into a mutex or channel, for multithreading
  - Iterators that drain their container, moving-out all values
  - Methods that take `self` by-move, taking it away from caller
  - Closure types that can only be called at most once
  - Encoding FSMs with phantom types and arg-consuming functions
  - The universal `drop<T>(x: T) { }` function that takes and drops anything

Move semantics

Move is just copy + forget

- And, as a segue to our next section:
  - Immutable references can be copied
  - Mutable references can only be moved
- This will make more sense as we explore borrowing

Borrow checking

This is maybe the strangest part

It's "borrow checking" time!

Let us picture an orderly library

Borrow any book you want to read

(So long as it exists)

But only write in your own book

Borrow checking

Preface about `mut` keyword

- Everything in Rust is default-immutable
- Mutability is opt-in with `mut` keyword, unlike C++ *immutability* opt-in `const`
- This is true about references too:
  - C++ immutable `const&` is written as just plain `&` in Rust
  - C++ mutable `&` is written as `&mut` in Rust
- Except, of course, that the rules are also somewhat different in Rust

Borrow checking

Preface about term "borrowing", at least among true fans

	act or object	`&`	`&mut`
verb	forming a reference	"*borrowing*"	"*mutably borrowing*"
noun	reference	"*a borrow*"	"*a mutable borrow*"
adjective	referent	"*borrowed*"	"*mutably borrowed*"

Borrow checking

The rules

- Borrow checking enforces two main sets of rules
  - "**Referent outlives reference**" / "lifetimes"
    - To avoid dangling/wild pointers
  - "**Shared xor mutable**" / "static reader-writer locking"
    - To avoid invalidating one pointer by writing through another
- This is all static, compile-time checking, not runtime mechanisms

Borrow checking

Referent outlives reference

- First set of rules: **referent outlives reference**
  - Prevents borrows pointing at garbage memory
- You can only borrow something (form a `&` or `&mut` reference) if:
  - The borrowed thing exists before the borrow starts
  - The borrowed thing exists after the borrow is done

Borrow checking

Referent outlives reference

- This is tracked through extra (static, compile-time) variables called "lifetimes"
  - Often inferred / invisible, but sometimes written explicitly
  - Written as a name with a leading single quote, after the ampersand
    - Like `&'a` or `&'some_lifetime mut`
    - It can help to give them informative names
      - Unfortunately lots of times they get named `'a`, `'b`, `'c`

Borrow checking

Referent outlives reference

- This is weird and does not read like C++. It's new.
- Here's an example:

```
struct Foo<'a> {
  x: &'a i32,
  y: &'a mut i64
}
```

- Nobody likes how this reads and it never gets easier on the eyes

Borrow checking

Referent outlives reference

- Sometimes you have to write `'a`-outlives-`'b` relationships between them

```
struct Foo<'event, 'session: 'event> {
  e: &'event i32,
  s: &'session mut i64
}
```

- This means `'session` is some lifetime that outlives `'event`

Borrow checking

Referent outlives reference

- Yes, that `:` means lifetimes are related through *subtyping*
- References with longer lifetimes are subtypes of those with shorter ones
  - This is confusing to everyone: it *seems* backwards
  - It's correct because lifetimes are "at least" statements / lower bounds
    - Something that lives "at least 100 years" also lives "at least 2 years"
    - Not vice-versa
    - Stare at those statements until you're sure -- it confuses everyone

Borrow checking

Shared xor mutable

- Second set of rules: **shared xor mutable**
  - Means that *any* memory-write *will not* invalidate some other pointer
  - Most important "at a distance" (other functions, other threads)
  - Can even save you "up close" (same function and thread)
    - Eg. borrow an enum (variant type) in case X, then set it to case Y, oops

Borrow checking

Shared xor mutable

- Repeat until it is a reflex in your thoughts
  - Sharing, or mutating, but not both
    - You can have multiple `&` refs that share read-access to some memory
    - You can have **one** `&mut` ref that has write-access to some memory
  - You (statically) can't have both at once
    - Just a static, compile-time-enforced version of a read-write lock

Borrow checking

Shared xor mutable

- In fact any `&mut` ref is an **exclusive** access path to the referent
- Eg. this will not compile, since `x` can't be read while mutably borrowed:

```
fn main() {
    let mut x = 10;
    let r = &mut x;
    let y = x + 1;
    println!("*r = {}", *r);
}
```

Borrow checking

Shared xor mutable

- This version will compile, since the borrow is "shared" (immutable):

```
fn main() {
    let mut x = 10;
    let r = &x;    // <- changed to immutable borrow
    let y = x + 1;
    println!("*r = {}", *r);
}
```

Borrow checking

Shared xor mutable

- This version will again not compile, since the borrow freezes `x`:

```
fn main() {
    let mut x = 10;
    let r = &x;
    x = x + 1;     // <- changed to mutate x
    println!("*r = {}", *r);
}
```

Borrow checking
Summary

- Borrow checking has *lots* of implications
  - All iterator invalidation cases in C++ are statically prohibited
    - Looping over a collection: collection frozen
    - Accessing a bucket in a hashtable: table frozen
    - This actually makes many collection-access patterns tricky!

Borrow checking
Summary

- Borrow checking has *lots* of implications
  - Immutability is *deep*
    - In C++ terms, `const` (methods, pointers) are *transitive through pointers*
  - Interestingly, so is mutability
    - Mutability *isn't designated* on individual `struct` fields
    - Either a whole `struct` is held in an exclusive mutable owner
      - Or someone has borrowed an exclusive `&mut` pointing into that owner

Borrow checking

Summary

- Borrow checking is an *over-approximate* safety check
  - Some things that seem ok-ish and useful are unfortunately off limits
    - Eg. "parent" links in trees using `&` are impossible
    - Don't even ask about doubly-linked lists
- It is being made more precise as time passes
  - Newer Rust editions typically enable finer-grained borrow tracking
  - Things you thought "ought to be ok", but weren't, might start being ok

Borrow checking

Summary

- Just getting program past borrow checker can be hard
  - Surprising how many aliases were hiding in C++
  - Eg. how every non-`const` method call can change members via `this`
    - If you felt OOP was a little error-prone, now you know why
- Helps to only borrow specific fields, not `self` as a whole
- Helps to restrict to arg-passing and returns, not store `&` refs in structs

Borrow checking
Summary

- Don't be too much of a purist:
  - Borrow checker only cares about `&` and `&mut`
  - You don't have to use them too extensively
  - Move owned values around instead of borrowing
  - `Box`, `Rc` and `.clone()` occasionally, it's fine, I permit you
  - Much of my own code rarely mentions lifetimes
  - Get your program working, add borrows *gradually*

Part 3

Easy stuff: everything else

Everything else

A tour of some highlights

- Stretch legs, get a snack
- Remainder is more casual
- Grocery list of features
- Knowing what to read up on
  - And some quirks, surprises
- Resources for learning more

Modeling data

Rust nouns

Modeling data
Primitives

- The easiest slide:
  - Signed integers: `i8`, `i16`, `i32`, `i64`, `i128`, `isize`
  - Unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128`, `usize`
  - Floating point: `f32`, `f64`
  - Boolean: `bool`
  - Unicode codepoint: `char`

Modeling data

Less obvious primitives

- The "unit" type, with one inhabitant: `()`
  - You'll see this all the time, it's like `void` in C++
- The "never" type, with zero inhabitants: `!`
  - This is much less common, more like `[[noreturn]]` in C++11
- Tuple types: `(T, U, V)`
  - Similar to `std::tuple`, for when a named struct type is overkill
- Arrays and slices (covered next), plus pointers and references also called "primitive"

Modeling data

Arrays, vectors and slices

- Homogeneous bulk data has 3 main forms
  - **Array**: fixed size, owns data stored *in the value*
    - eg. an array of 12 `i8` elements: `[i8; 12]`
  - **Vector**: dynamic size, owns data stored *in the heap*
    - eg. a vector of `i8`, unknown length: `Vec<i8>`
  - **Slice**: pointer-and-length pair, *borrowed* from somewhere else
    - eg. a slice of some `i8` values in some vector or array: `&[i8]`

Modeling data

Unfortunately, as you know, Unicode

- Strings are treated *similarly* to arrays, but with special cases
  - `char`: 21bit "Unicode scalar value", stored in 32bit word
    - Can theoretically make `[char;N]` or `Vec<char>` or `&[char]`
    - But people very rarely use this; instead they use
  - `String`: dynamic size, a bit like `Vec<u8>`, but *guaranteed UTF-8*
  - `&str`: pointer-and-length pair, UTF-8, *borrowed* from some `String` or static `str`
    - Similar to C++17 `std::string_view`

Modeling data

A surprise about strings

- You cannot randomly access `&str` or `String` and get a `char`
  - If you think about it, this is just being honest about `char` vs. UTF-8
  - You can *iterate* over the `char`s (produced one by one)
  - You can extract optional sub-slices at UTF-8 codepoint boundaries
  - You can borrow a `&[u8]` slice and randomly access the `u8`s
  - You can *convert* to a `Vec<char>` and randomly access the `char`s

Modeling data
Structs and enums

- Rust code uses two main type-constructors for *named* aggregates:
  - **Struct**:
    - Fairly similar to C++ `struct`, won't say much more
  - **Enum**:
    - Much richer than C++ `enum` or even `enum class`
    - Tagged disjoint union, more like C++17 `std::variant`
    - Pattern-based `match` with exhaustiveness checking, more like `switch`

Modeling data
Structs and enums

- Two generic `enum`s used very ubiquitously:
  - `Option<T>` which can be either `Some(T)` or `None`
    - Similar to C++17 `std::optional`, used anytime you need a sentinel value
    - Recall: the pointer-like types in Rust *have no* `nullptr` inhabitant!
  - `Result<T,E>` which can be either `Ok(T)` or `Err(E)`
    - Used for conveying value-or-an-error, have special syntax (see later)

Modeling data
Limitations

- There is no inheritance or subtyping of structs or enums
  - Manually nest / delegate to inner shared types
- There are no copy constructors, assignments or conversions through user code
  - Really there are no constructors at all, just `Foo::new()` by convention
- Some patterns are basically forbidden, or at least Very Surprisingly Hard
  - Cycles and sharing, mutable globals, anything violating ownership rules
  - For both: try using existing "special library types" with safe *interfaces*
  - Rules *can* be broken through `unsafe`; that's what the they do in *implementation*

Making things happen

Rust verbs

Making things happen

Functions, statements, expressions

- Functions always have type signatures; inside types optional, like C++ `auto`
- Function decls look like `fn foo(x: u8) -> u8 { ... }`
- Body contains a bunch of *statements*: local decls or expressions
- `let id: type = ...` is the only local decl worth mentioning here
- Everything else is an expression, which can *nest*
- Eg. `let x = if foo() { 10 } else { 11 };`
- The most surprising part is "where to put the semicolons", yawn

Making things happen

A few expression peculiarities

- `loop { ... }` is an infinite loop, for various reasons
- Heads of control structures don't have parentheses
  - `while x > 10 { ... }`
  - `if x + 5 < 11 { ... }`
- Last expression in a block or `fn` is its value, don't need to write `return`
- And then also: semicolon discards previous expr, provides implicit `()`
  - Thus: `fn add(x: u8, y: u8) { x + y }`
  - Wasn't kidding about semicolons tripping you up

Making things happen

A few special expressions

- Struct initialization: `Foo { x: 10, y: 11 }`
- Error-propagation sugar on `Result<T,E>`-typed calls: `foo()?.x.bar()?`
- Closures a bit like C++11 lambdas: `|x, y| x + y`
- Macro-invocations like `println!(...)`, always marked with `!` unlike C/C++
- Pattern-match, like C/C++ `switch` but much more powerful, used everywhere:

```
match ... {
    pattern => ...,
    pattern => ...
}
```

Making things happen
Iterators and ranges

- As with C++ there's an **iterator** abstraction and a **range** abstraction
  - In Rust, `Range` implements the `Iterator` trait; we'll meet traits shortly
  - Range expressions have syntax sugar like `1..10` or `x..=y`
- `for ... in ... { ... }` is sugar for an iterator while-match loop
  - Iterators are *very powerful* but still get optimized away completely
  - Read the assembly, it'll just be pointer-bumping loops like C++
  - *Lots* of generic iterator combinators, like C++20 `std::range` library

Making things happen

Method calls, traits and impls

- As with C++ a lot of the action happens via type-directed dispatch
- A.k.a. method calls: `foo.bar(x,y,z)`
- What method does this call? If only the answer was simple!
- In Rust, this will be resolved to an `impl` of a `trait`
- `trait` is like C++20 `concept`, or a partial abstract `class`
- `impl` is a bit unlike anything in C++

Making things happen

Traits

- A `trait` defines a named set of methods over unknown type `Self`
- Then usable as a *constraint* on a type parameter
  - Like a C++20 `concept` applied to a `typename`
- Typechecked as pure abstractions, *independent of implementations*
- As are functions generic over traits
  - Errors caught *before* instantiation, unlike C++ templates
  - No SFINAE! Generic function can *only* call trait-defined methods

Making things happen

Traits

- Example:

```
trait Beeper {
    fn beep(&self);
}


fn beep_one<B:Beeper>(b: &B) {
    b.beep();
}
```

Making things happen

Trait additional wrinkles

- Trait may also have some associated types and constants
- Possibly also *generic* over various type *parameters* with trait bounds
- May also contain default method bodies, in terms of the constraining traits
- May extend other traits in a DAG, a bit like abstract classes
- Can also be used as an existential "object" type, with *dynamic dispatch*

Making things happen

Impls

- `impl` declares that (and how) some type implements some trait
- Example

```
trait Beeper {
    fn beep(&self);
}

impl Beeper for i32 {
    fn beep(&self) {
        println!("beep {} times", self);
    }
}
```

Making things happen

Impls

- Unlike C++, `impl` is not directly part of a *type* declaration
- Can happen in crate declaring type *or* crate declaring trait
  - Not by a third party though -- must be one or the other, for coherence
- `impl` blocks can also be generic over types, so-called *blanket* impls
  - Behaves a bit like a `template class` that's retroactively automatically inherited by anything meeting its template requirements
  - Can also impl traits for primitives, not just user types
  - Very powerful for applying broad new behaviours to existing types

Making things happen

Trait limitations

- There is no way to *specialize* a trait or have overlapping impls
  - There might be someday but don't hold your breath
- There is no ad-hoc function overloading except on `Self`, via traits
- There is no SFINAE (I think this is good on balance, but..)
  - You need traits for *any* call through generic types in generic code
  - By default you can't even `x.clone()` some `x:T`, much less do arithmetic

Making things happen
Multithreading

- Remember how I said "concurrency" earlier?
- There's a *lot* of multithreading but it's all surprisingly safe
- Special traits to mark sharable data like mutexes, atomics
- Mutexes *own data they protect*, so must be locked to borrow-through
  - Access through RAII guard objects, like C++ `std::scoped_lock`
- Magic parallel-iterator traits in "Rayon" crate, as easy as `execution::par`
  - Fairly standard for programs to shard-and-parallelize any heavy process

Making things happen
Async/await

- Multithreading also has a weird cousin "async/await"
- For "much higher concurrency than thread count"
  - Eg. server with 100,000 concurrent TCP connections, uses much less memory
- Careful (but awkward) separation between language and supporting crates
  - Need to use a "runtime" and special traits from crate such as Tokio
  - Language part is `async { ... }` and `.await` expressions
  - Sugar for making and interacting with resumable finite state machines
  - Like C++20 coroutines and `std::promise` / `std::future`

Making things happen
Errors

- Errors have special support
- If anything unrecoverable happens, just `panic!()` and the program will (mostly-unrecoverably) unwind, abort, halt. A few exceptions to this, but assume fatal.
- *Plausibly recoverable* errors are modelled by returning `enum Result<T,E>`
- There is a special postfix `?` operator like `foo()?` or `(a + b)?`
  - Error-converting early-return if `Err(...)`, or ok value if `Ok(...)`
  - Can see and control where errors are checked and propagated
  - Intentionally small, so can chain: `foo()?.bar()?.baz()`

Organizing code

Rust filing techniques

Organizing code
Crates and modules

- Rust code is organized via two main structures:
  - **Crates**:
    - unit of compilation, linking, versioning, distribution, workflow
  - **Modules**:
    - hierarchical container of item definitions / names within a crate

Organizing code
Crates

- Rust compiler parses, loads deps for, compiles and links 1 crate at a time
  - One `.rs` file is the crate root, all files it mentions (transitively) get included
- Think of a crate as "a single library or executable" ("all its inputs" and/or "the output")
  - A bit like a C++20 *module*, more like a C# *assembly* or Java JSR 376 *module*
- Crates have **versions** and other **metadata** serialized on disk
  - Versioning machinery allows coexistence of 2 or more versions of same crate
    - Also prevents crate-name collisions: 2 crates named `foo` can coexist
    - Symbols all have crate metadata and version info mangled into them

Organizing code
Crates and cargo

- For historical reasons, there is a program called `cargo`
  - Deals with everything at-or-above crate level of abstraction
  - Versioning, dependencies, metadata
  - Package management, upload and download (see crates.io)
  - Build, test, profile, documentation automation
  - Runs the compiler `rustc`, which was originally supposed to do all this itself
- Controlled via `Cargo.toml` file
- You may actually never run the compiler by hand, `cargo` is very thorough

Organizing code
Modules

- Modules are a static tree of containers for declarations
  - Similar to C++ `namespace`, maybe more similar to Java `package`
  - Map to a specific scope or file in a source directory tree, have visibility controls
- Crate root `.rs` file is an *anonymous* module (keyword `crate`) the root of any module path
  - Can declare sub-modules, which may take 3 forms
    - Inline modules: `mod foo { ... }`
    - Source module in curr dir: `mod foo; // at root, attaches foo.rs`
    - Source module in sub dir: `mod bar; // in foo.rs, attaches foo/bar.rs`

Organizing code

Three module-system surprises

- Visibility control defaults to private, must mark public things `pub`
- Modules default to empty, must explicitly `use` anything you want
  - There is a special set of standard decls called "the prelude" auto-`use`d
  - You can turn that off too if you want to be extra strict
- Often re-export imported items with cutesy `pub use` item
- Module system appears to trip lots of people up
  - More hierarchy and explicit structure than many languages
  - More things denied-by-default

Resources

Things to click on, install, read, watch, buy

Resources

Tools and setup

- Community makes its own very good installer rustup.rs
- Can also `apt install cargo` or `brew install rustup-init`
- You almost certainly want to use an IDE based on `rust-analyzer`
  - vscode works very well, vim and emacs apparently also good
  - rust-analyzer.github.io
  - There is an older, deprecated backend called RLS; skip it
  - JetBrains and some other vendors have their own also

Resources

Major Rust websites to bookmark

- If you can only bookmark one thing, bookmark this
  - **Rust Language Cheat Sheet: cheats.rs**
- Covers the language exhaustively
- Covers project layout, coding guides, tool references
- Has diagrams, idioms, conversion tables, calculators
- Links to every major additional doc and index site
- Very dense, but an absolute work of art

Resources

Major Rust websites to bookmark

- Main website: www.rust-lang.org
- Language docs:
  - "The Book / TRPL": doc.rust-lang.org/book
  - Reference: doc.rust-lang.org/stable/reference
  - Little Book of Rust Books: lborb.github.io/book/
- Stdlib docs doc.rust-lang.org/stable/std
- Packages: crates.io or alternative site lib.rs
- Package docs docs.rs

Resources

If you prefer to learn from examples

- "Rust by Example": doc.rust-lang.org/stable/rust-by-example
- "Rust by Practice": practice.rs
- "Rust Cookbook": rust-lang-nursery.github.io/rust-cookbook/
- "½ Hour to Learn Rust": fasterthanli.me/articles/a-half-hour-to-learn-rust

Resources

Major crates to know & use

- anyhow: error handling
- rand: randomness
- regex: regexes
- im: immutable / functional collections
- chrono: dates and times
- itertools: iterator combinators
- log, tracing: logging and tracing
- num: generic & wide arithmetic
- hyper, reqwest, warp: HTTP
- nom: binary parser combinators
- criterion: profiling
- bumpalo: arena allocation
- rayon, crossbeam: parallelism
- rustls, ring, dalek: cryptography
- tokio: async/await & network IO
- serde: serialize/deserialize
- proptest, cargo-fuzz: random testing
- bindgen, cxx, autocxx: C/C++ interop

Resources

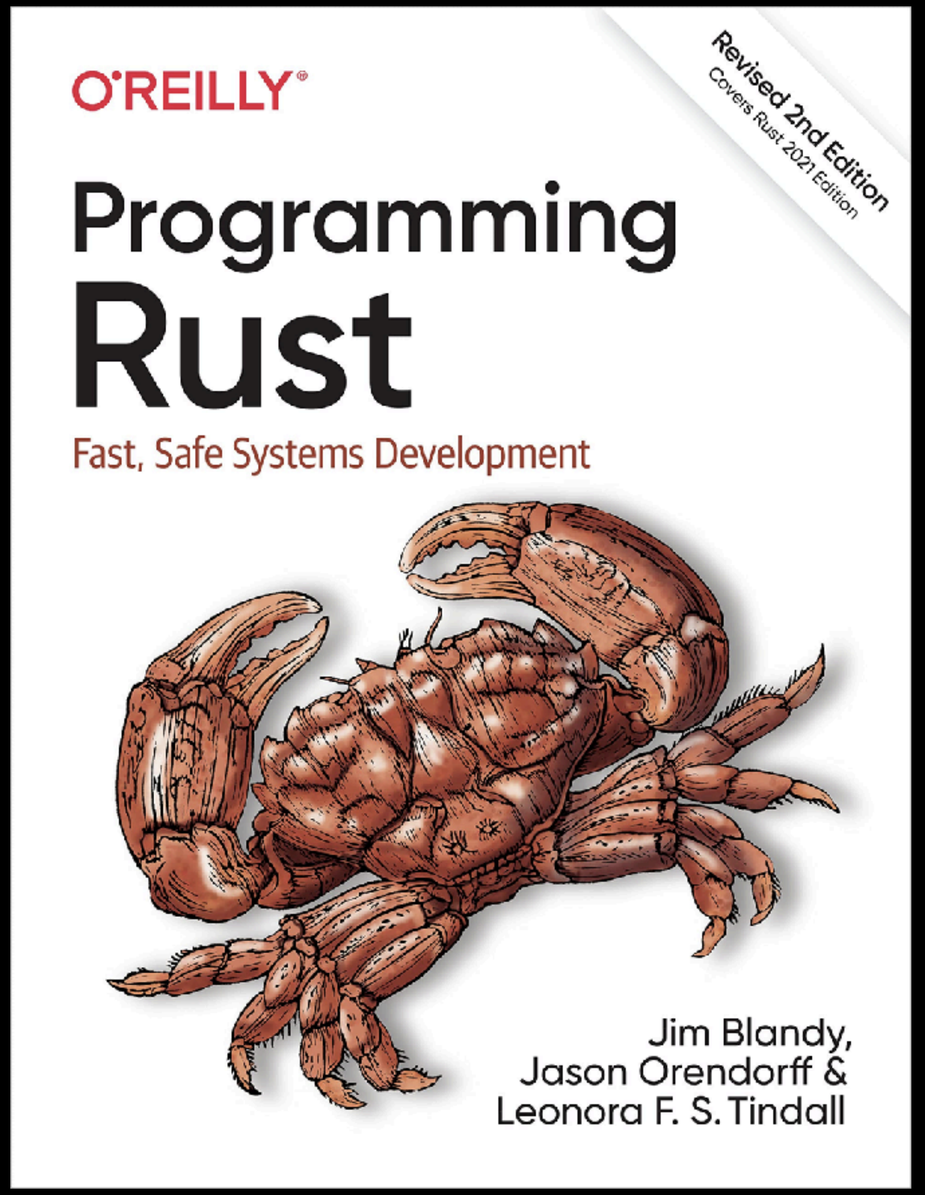
If you like watching several more hours of talking and slides

- Rust for C++ developers - What you need to know to get rolling with crates
  - www.youtube.com/watch?v=k7nAtrwPhR8
- A Firehose of Rust, for busy people who know some C++
  - www.youtube.com/watch?v=IPmRDS0OSxM
- A C++ Programmer's View on Rust
  - www.youtube.com/watch?v=DGbsHENouy4

Resources

If you like giant books

- Highly recommend this one if you happen to want 738 pages of exquisite detail about why everything in Rust is the way it is and how to use it well.
- Written by my friends and former colleagues, obvious conflict of interest but I do actually think it's very good also.
- There are other good books!



Fini