Lecture 4: Cargo and Iterators

Alexander Stanovoy

March 22, 2022

alex.stanovoy@gmail.com

In this lecture

- Cargo and crates
- Modules
- Iterators

Cargo and crates

Cargo

Cargo is the Rust language package manager. It's one of the greatest things about Rust!

What Cargo does do?

- Downloads and manages your dependencies.
- Compiles your packages.
- Makes distributable packages.
- And more!

Crate

A crate is a compilation unit in Rust. It's like a package in other languages. An example of crate created by `cargo new --bin example` command:

```
example
├── Cargo.lock
├── Cargo.toml
└── src
└── main.rs
```

Packages can be uploaded to crates.io, the Rust community's crate registry. It makes them available to everyone, and users will have an opportunity to use your crate as a dependency at the manifest.

Crate

A package is described using manifest file called `Cargo.toml`. Here's an example:

```toml
[package]
name = "example"
version = "0.1.0"
edition = "2021"

[dependencies]
clap = "3.1.0"
```

Crate: `Cargo.toml`

`Cargo.toml` consists of multiple entries.[1] Here's an example:

- `[package]` has the meta information about package like name, version, authors, edition, compiler version, build scripts...
- `[dependencies]` describes dependencies of our package, their versions, needed features.
- `[features]` provides a mechanism to express conditional compilation and optional dependencies.
- `[profile.TYPE]` describes how to compile in different profiles: `dev`, `release`, `test` and `bench`.
- And more!

[1] The Manifest Format

Crate

There are multiple types of crates.[2]

- `bin` - a runnable executable. It's default crate type
- `lib` - a "compiler recommended" Rust library.
- `dylib` - a dynamic Rust library.
- `staticlib` - a static system library.
- `cdylib` - a C dynamic library.
- `rlib` - a "Rust library" file.
- `proc-macro` - a procedural macros crate.

`bin` or `lib` types should be sufficient for all compilation needs.

[2] Linkage, The Rust Reference

Crate: versions

In Cargo, versions of packages **must be changed** accordingly to Semantic Versioning (semver).[3]

Given a version number `MAJOR.MINOR.PATCH`, increment the:

- `MAJOR` version when you make incompatible API changes.
- `MINOR` version when you add functionality in a backwards compatible manner.
- `PATCH` version when you make backwards compatible bug fixes.

[3] Semantic Versioning 2.0.0

Crate: versions

For instance, this version changes are legal:[4]

- `1.3.7` -> `1.3.8` (bug fix).
- `1.5.5` -> `1.6.0` (added functionality).
- `1.7.2` -> `2.0.0` (major update, incompatible changes).

If your `MAJOR` version number is 0, Cargo will treat `MINOR` as `MAJOR` and `PATCH` as `MINOR`.

[4] SemVer Compatibility

Crate: versions

In `Cargo.toml`:

- `^1.2.3` - semver compatible (`< 2.0.0`)
- `~1.2.3` - only the last number is updated (`< 1.3.0`)
- `1.2.*`
- `>= 1.2`

If your `MAJOR` version number is 0, Cargo will treat `MINOR` as `MAJOR` and `PATCH` as `MINOR`.

`Cargo.toml` vs `Cargo.lock`

- `Cargo.toml` is about describing your dependencies in a broad sense, and is written by you.
- `Cargo.lock` contains exact information about your dependencies. It is maintained by Cargo and should not be manually edited.

`Cargo.toml` vs `Cargo.lock`

- `Cargo.toml` is about describing your dependencies in a broad sense, and is written by you.
- `Cargo.lock` contains exact information about your dependencies. It is maintained by Cargo and should not be manually edited.

The purpose of a `Cargo.lock` lockfile is to describe the state of the world at the time of a successful build.

`Cargo.toml` vs `Cargo.lock`

- `Cargo.toml` is about describing your dependencies in a broad sense, and is written by you.
- `Cargo.lock` contains exact information about your dependencies. It is maintained by Cargo and should not be manually edited.

The purpose of a `Cargo.lock` lockfile is to describe the state of the world at the time of a successful build.

This property is the most desirable for packages that are at the very end of the dependency chain (binaries).

`Cargo.toml` vs `Cargo.lock`

- `Cargo.toml` is about describing your dependencies in a broad sense, and is written by you.
- `Cargo.lock` contains exact information about your dependencies. It is maintained by Cargo and should not be manually edited.

The purpose of a `Cargo.lock` lockfile is to describe the state of the world at the time of a successful build.

This property is the most desirable for packages that are at the very end of the dependency chain (binaries).

But the library is not only used by the library developers, but also any downstream consumers of the library. Libraries specify semver requirements for their dependencies but cannot see the full picture. Only end products like binaries have a full picture to decide what versions of dependencies should be used.

`cargo-edit`

- `cargo add foo` - add the latest version of library `foo` to `Cargo.toml`.
- `cargo rm foo` - remove library `foo` from `Cargo.toml`.
- `cargo upgrade` - upgrade all dependencies versions in `Cargo.toml` to latest.

To install, run `cargo install cargo-edit`

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- Before version 1.0, as a new product, Rust changed frequently and hadn't given any guarantees about how your code will be compiled in future releases.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- Before version 1.0, as a new product, Rust changed frequently and hadn't given any guarantees about how your code will be compiled in future releases.
- After the release of 1.0, Rust started following one of the variations of "stability without stagnation" model, first introduced in web browsers.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- Before version 1.0, as a new product, Rust changed frequently and hadn't given any guarantees about how your code will be compiled in future releases.
- After the release of 1.0, Rust started following one of the variations of "stability without stagnation" model, first introduced in web browsers.
- New work lands directly in the master branch.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- Before version 1.0, as a new product, Rust changed frequently and hadn't given any guarantees about how your code will be compiled in future releases.
- After the release of 1.0, Rust started following one of the variations of "stability without stagnation" model, first introduced in web browsers.
- New work lands directly in the master branch.
- Each day, the last successful build from the master becomes the new nightly release.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- Before version 1.0, as a new product, Rust changed frequently and hadn't given any guarantees about how your code will be compiled in future releases.
- After the release of 1.0, Rust started following one of the variations of "stability without stagnation" model, first introduced in web browsers.
- New work lands directly in the master branch.
- Each day, the last successful build from the master becomes the new nightly release.
- Every six weeks, a beta branch is created from the current state of the master, and the previous beta is promoted to be the new stable release.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- In short, there are three release channels - nightly, beta, and stable - with regular, frequent promotions from one channel to the next.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- In short, there are three release channels - nightly, beta, and stable - with regular, frequent promotions from one channel to the next.
- New features and new APIs will be flagged as unstable via feature gates and stability attributes respectively. Unstable features and standard library APIs will only be available on the nightly branch, and only if you explicitly “opt-in” to the instability.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- In short, there are three release channels - nightly, beta, and stable - with regular, frequent promotions from one channel to the next.
- New features and new APIs will be flagged as unstable via feature gates and stability attributes respectively. Unstable features and standard library APIs will only be available on the nightly branch, and only if you explicitly “opt-in” to the instability.
- But sometimes, make small changes to the language that are not backward compatible. The most obvious example is introducing a new keyword, which would invalidate variables with the same name.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- In short, there are three release channels - nightly, beta, and stable - with regular, frequent promotions from one channel to the next.
- New features and new APIs will be flagged as unstable via feature gates and stability attributes respectively. Unstable features and standard library APIs will only be available on the nightly branch, and only if you explicitly “opt-in” to the instability.
- But sometimes, make small changes to the language that are not backward compatible. The most obvious example is introducing a new keyword, which would invalidate variables with the same name.
- For instance, before 2018 there were no `async` and `await` keywords.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- In short, there are three release channels - nightly, beta, and stable - with regular, frequent promotions from one channel to the next.
- New features and new APIs will be flagged as unstable via feature gates and stability attributes respectively. Unstable features and standard library APIs will only be available on the nightly branch, and only if you explicitly “opt-in” to the instability.
- But sometimes, make small changes to the language that are not backward compatible. The most obvious example is introducing a new keyword, which would invalidate variables with the same name.
- For instance, before 2018 there were no `async` and `await` keywords.
- When the release is about to break code, it becomes a part of the new edition. The choice of edition is made in `Cargo.toml` for a crate.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- This ensures that the decision to migrate to a newer edition is a “private one” that the crate can make without affecting your downstream crates.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- This ensures that the decision to migrate to a newer edition is a “private one” that the crate can make without affecting your downstream crates.
- To automate migration, there’s `cargo fix --edition` command.

Rust versions

Rust version changes accordingly to semver, but how this exactly works?

- This ensures that the decision to migrate to a newer edition is a “private one” that the crate can make without affecting your downstream crates.
- To automate migration, there’s `cargo fix --edition` command.
- For example, when migrating to Rust 2018, it changes anything named async to use the equivalent raw identifier syntax: `r#async`.

Features

Cargo “features” provide a mechanism to express conditional compilation and optional dependencies.

```
[features]
bmp = []
png = []
ico = ["bmp", "png"]
default = ["ico"]
```

```
#[cfg(feature = "ico")]
pub mod ico;
```

Features

```toml
[dependencies]
reqwest = {
    version = ">= 0.11.9",
    features = ["blocking", "multipart"]
}
```

Features

The crate is compiled with all features needed for all dependencies.

Important: Features are **additive**!

Question: What can be wrong if your feature appeared to be non-additive?

Features

The crate is compiled with all features needed for all dependencies.

Important: Features are **additive**!

Question: What can be wrong if your feature appeared to be non-additive?

Rust is frequently used for embedded programming. Some crates provide the `no_std` feature to enable usage of the crate in.

Features

The crate is compiled with all features needed for all dependencies.

Important: Features are **additive**!

Question: What can be wrong if your feature appeared to be non-additive?

Rust is frequently used for embedded programming. Some crates provide the `no_std` feature to enable usage of the crate in.

As you may notice, `no_std` is **non-additive**, since `std` can do the same and even more.

Features

The crate is compiled with all features needed for all dependencies.

Important: Features are **additive**!

Question: What can be wrong if your feature appeared to be non-additive?

Rust is frequently used for embedded programming. Some crates provide the `no_std` feature to enable usage of the crate in.

As you may notice, `no_std` is **non-additive**, since `std` can do the same and even more.

I f there will be two crates depending on this crate with `no_std` enabled in one crate and disabled in another, Cargo will compile with this crate with `no_std`, leading to errors!

Features

It’s a common practice to implement some additional features as a crate feature.

Features

It's a common practice to implement some additional features as a crate feature.

Such common that Rust API guidelines book suggests to implement `Serialize` and `Deserialize` traits from crate `serde`.

Features

It's a common practice to implement some additional features as a crate feature.

Such common that Rust API guidelines book suggests to implement `Serialize` and `Deserialize` traits from crate `serde`.

Features give users a chance to conditionally opt-in to additional features rather than compiling them all the time.

As you can see, we don't need to even know something about `rustc`! (expect version, of course)

This is one of the cool things about Cargo.

A detailed discussion of it is not part of this lecture.

Modules

Modules

In C language, there are no namespaces. Combination of this and how `#include` preprocessor directive works results in name pollution. There are no good ways to solve this problem.[5]

[5] At least no direct solutions.

Modules

In C language, there are no namespaces. Combination of this and how `#include` preprocessor directive works results in name pollution. There are no good ways to solve this problem.[5]

In C++, name pollution is solved using namespaces. Namespaces are a pretty simple and practical solution.

[5] At least no direct solutions.

Modules

In C language, there are no namespaces. Combination of this and how `#include` preprocessor directive works results in name pollution. There are no good ways to solve this problem.[5]

In C++, name pollution is solved using namespaces. Namespaces are a pretty simple and practical solution.

What we'll do in Rust to prevent name pollution and use multiple files in the project?

[5] At least no direct solutions.

Modules

```rust
mod one {
    mod nested {
        mod nested2 {
            struct Foo { /* ... */ }
        }
        enum Count { /* ... */ }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    struct Bar { /* ... */ }
    fn use_me() { /* ... */ }
}
```

`mod` keyword defines a module. Modules are used to control the visibility of declarations inside it and to prevent namespace pollution.

Modules

We can use the `use_me` using full path:

```
two::use_me();
```

Modules

We can use the `use_me` using full path:

```
two::use_me();
```

But unlike in C++, this code won't compile:

```
error[E0603]: function `use_me` is private
  --> src/main.rs:16:12
   |
16 | two::use_me();
   |      ^^^^^^ private function
```

Modules

```
mod one {
    mod nested {
        mod nested2 {
            struct Foo { /* ... */ }
        }
        enum Count { /* ... */ }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

We'll use `pub` keyword. It means "make it private for all parent modules".

Modules

Next, we'll create `Foo` structure:

```
let _ = one::nested::nested2::Foo {};
```

Next, we'll create `Foo` structure:

```
let _ = one::nested::nested2::Foo {};
```

We'll find out our module is private!

```
error[E0603]: module `nested` is private
  --> src/main.rs:16:18
   |
16 | let _ = one::nested::nested2::Foo {};
   |              ^^^^^^ private module
```

Modules

So, unlike in C++ namespaces, Rust modules are private by default!

But why compiler asked to make public the declaration of `nested`? And in the previous case - `two`?

Modules

So, unlike in C++ namespaces, Rust modules are private by default!

But why compiler asked to make public the declaration of `nested`? And in the previous case - `two`?

Because `one` and `two` are the part of our current module called **the root module**. We don't need to have rights to use the declarations in our current module - everything is public for us.

Modules

So, unlike in C++ namespaces, Rust modules are private by default!

But why compiler asked to make public the declaration of `nested`? And in the previous case - `two`?

Because `one` and `two` are the part of our current module called **the root module**. We don't need to have rights to use the declarations in our current module - everything is public for us.

Let's make `nested` public. Then we'll get compiler errors since `nested2` and `Foo` are private and make them public them too.

Modules

```rust
mod one {
    pub mod nested {
        pub mod nested2 {
            pub struct Foo { /* ... */ }
        }
        enum Count { /* ... */ }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

Modules

```
mod one {
    pub mod nested {
        mod nested2 { // No 'pub'
            pub struct Foo { /* ... */ }
        }
        enum Count { /* ... */ }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

Note that the following code won't make `Foo` available since `nested2` is private for us.

```
mod one {
    pub mod nested {
        pub mod nested2 {
            pub struct Foo { bar: two::Bar }
        }
        enum Count { /* ... */ }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

Next, we'll try to add a field `bar` of type `two::Bar` to `one::nested::nested2::Foo`.

Modules

```
error[E0433]: failed to resolve: use of undeclared crate
              or module `two`
  --> src/main.rs:4:35
   |
 4 | pub struct Foo { bar: two::Bar }
   |                       ^^^ use of undeclared crate or module `two`
```

Why did this happen?

```
error[E0433]: failed to resolve: use of undeclared crate
              or module `two`
  --> src/main.rs:4:35
   |
 4 | pub struct Foo { bar: two::Bar }
   |                       ^^^ use of undeclared crate or module `two`
```

Why did this happen?

By default, paths are relative. To make them absolute, we should use `crate` keyword (remember the path / in Unix systems?). In this case, our path will always start from the root module, not from the current module.

Modules

```rust
mod one {
    pub mod nested {
        pub mod nested2 {
            pub struct Foo { bar: crate::two::Bar }
        }
        enum Count { /* ... */ }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    // Note the 'pub'
    pub struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

Since `Bar` is private in module `two`, we should use `pub` here too.

Note that now we cannot construct `Foo` because it's `bar` field is private!

```
let _ = one::nested::nested2::Foo { bar: two::Bar {} };
```

```
error[E0451]: field `bar` of struct `Foo` is private
  --> src/main.rs:17:41
   |
17 |let _ = one::nested::nested2::Foo { bar: two::Bar {} };
   |                                    ^^^^^^^^^^^^^^^^ private field
```

In Rust, fields of structs are private by default and available only for the current module. It's the cause why you cannot access fields of, for instance, `Vec`. To fix this, you should make the field public.

But, of course, do not do this without reason: it's much better to implement type constructors and getters.

Modules

```rust
mod one {
    pub mod nested {
        pub mod nested2 {
            pub struct Foo { pub bar: crate::two::Bar }
        }
    enum Count { /* ... */ }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    pub struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

Modules

```rust
mod one {
    pub mod nested {
        pub mod nested2 {
            pub struct Foo { bar: crate::two::Bar }
        }
        pub enum Count { Example(nested2::Foo) }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    pub struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

Let’s add an enumeration variant to count and make it public as usual.

Modules

Using the enumeration:

```rust
let bar = two::Bar {};
let foo = one::nested::nested2::Foo { bar };
let example = one::nested::Count::Example(foo);
```

Unlike in `struct`, all enumuration variants are available if the `enum` is available.

Modules

```
mod one {
    pub mod nested {
        pub mod nested2 {
            pub struct Foo { bar: crate::two::Bar }
            impl crate::one::MyTrait for Foo {}
        }
        pub enum Count { Example(nested2::Foo) }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    pub struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

We want to implement `MyTrait` for `Foo`. It's done pretty easily. In Rust, we don't need an object to be `pub` when it's defined in one of the ancestor modules.

```rust
mod one {
    pub mod nested {
        pub mod nested2 {
            pub struct Foo { bar: super::super::super::two::Bar }
            impl super::super::MyTrait for Foo {}
        }
        pub enum Count { Example(nested2::Foo) }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    pub struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

We can also use `super` keyword (remember the `..` on Unix?). Just for example, the declaration of field `bar` is also changed.

Modules

```rust
mod one {
    pub mod nested {
        pub mod nested2 {
            pub struct Foo { bar: crate::two::Bar }
        }
        impl crate::one::MyTrait for nested2::Foo {}
        impl nested2::Foo {}
        pub(self) enum Count { Example(nested2::Foo) }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    pub struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

We can write `impl` where we want! (But please, don't do this)

```
mod one {
    pub mod nested {
        pub mod nested2 {
            pub struct Foo { bar: crate::two::Bar }
            impl crate::one::MyTrait for Foo {}
        }
        impl crate::one::MyTrait for nested2::Foo {}
        pub(self) enum Count { Example(nested2::Foo) }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    pub struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

Ok, I want multiple implementations of the trait in different modules.

Coherence

```
error[E0119]: conflicting implementations of trait `one::MyTrait`
              for type `one::nested::nested2::Foo`
 --> src/main.rs:8:9
  |
5 | impl crate::one::MyTrait for Foo {}
  | -------------------------------- first implementation here
 ...
8 | impl crate::one::MyTrait for nested2::Foo {}
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ conflicting
  |                                           implementation for
  |                                           `/* */::Foo`
```

Coherence

- Rust must guarrantie that there’s only one implementation of trait for every object.

Coherence

- Rust must guarrantie that there’s only one implementation of trait for every object.
- More generally, there’s *coherence* property: for any given type and method, there is only ever one correct choice for which implementation of the method to use for that type.

Coherence

- Rust must guarrantie that there's only one implementation of trait for every object.
- More generally, there's *coherence* property: for any given type and method, there is only ever one correct choice for which implementation of the method to use for that type.
- Consider what would happen if we could write own implementation of the `Display` trait for the `bool` type from the standard library. Now, for any code that tries to print a `bool` value and includes my crate, the compiler won't know whether to pick the implementation I wrote or the one from the standard library. Neither choice is correct or better than the other, and the compiler obviously cannot choose randomly.

Coherence

- Rust must guarrantie that there’s only one implementation of trait for every object.

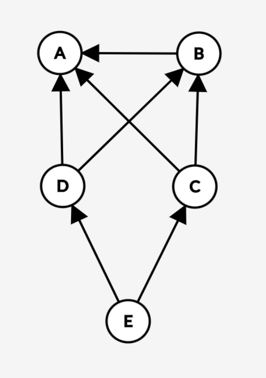
Coherence

- Rust must guarrantie that there's only one implementation of trait for every object.
- More generally, there's *coherence* property: for any given type and method, there is only ever one correct choice for which implementation of the method to use for that type.

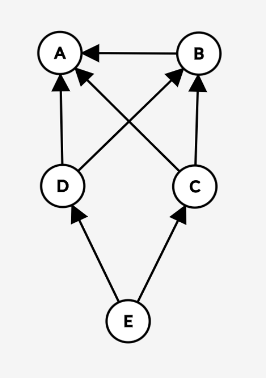
Coherence

- Rust must guarrantie that there's only one implementation of trait for every object.
- More generally, there's *coherence* property: for any given type and method, there is only ever one correct choice for which implementation of the method to use for that type.
- Consider what would happen if we could write own implementation of the `Display` trait for the `bool` type from the standard library. Now, for any code that tries to print a `bool` value and includes my crate, the compiler won't know whether to pick the implementation we wrote or the one from the standard library. Neither choice is correct or better than the other, and the compiler obviously cannot choose randomly.

Coherence

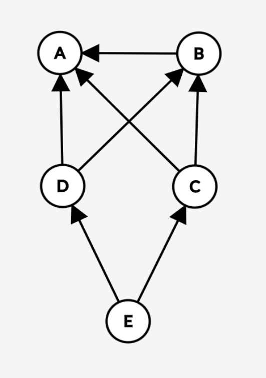


- Or the following: we've got crates A and B, in A there's trait, and in B there's a type. B depends on A. We want to use both of them as dependencies in the crate C. Also, there's crate D that depends on both A and B.



- Or the following: we've got crates A and B, in A there's trait, and in B there's a type. B depends on A. We want to use both of them as dependencies in the crate C. Also, there's crate D that depends on both A and B.
- Let's implement a trait from A for structure from B in C and D.

Coherence



- Or the following: we've got crates A and B, in A there's trait, and in B there's a type. B depends on A. We want to use both of them as dependencies in the crate C. Also, there's crate D that depends on both A and B.
- Let's implement a trait from A for structure from B in C and D.
- In E, we have two implementations of trait!

Orphan rule

To make sure the compiler will see only one implementation, there's *orphan rule*.

Simply stated, the orphan rule says that you can implement a trait for a type only if the trait or the type is local to your crate (not module!).

In our example, we'll be able to implement a trait for type only in crate `B`.

In this rule, there are some exceptions.

Orphan rule: Blanket Implementations

Remember: this is a *blanket implementation*.

```
impl<T> MyTrait for T where T: Something { /* ... */ }
```

Only the crate that defines a trait is allowed to write a blanket implementation!

Adding a blanket implementation to an existing trait is considered a breaking change. If it were not, a downstream crate that contained `impl MyTrait for Foo` could suddenly stop compiling just because you update the crate that defines `MyTrait` with an error about a conflicting implementation.

Orphan rule: Fundamental Types

Some types are so essential that it's necessary to allow anyone to implement traits on them.

These types currently include `&`, `&mut`, and `Box`. For the purposes of the orphan rule, fundamental types are erased before the orphan rule is checked.

```rust
impl IntoIterator for &MyType { /* ... */ }
```

With just the orphan rule, this implementation would not be permitted since it implements a foreign trait for a foreign type - `IntoIterator` and `&` both come from the standard library.

Note: In standard library this types are marked by `#[fundamental]` attribute.

Orphan rule: Covered Implementations

There are some limited cases where we want to allow implementing a foreign trait for a foreign type, which the orphan rule does not normally allow:

```rust
impl From<MyType> for Vec<i32> { /* ... */ }
```

Here, the `From` trait is foreign, as is the `Vec` type.

Given `impl<P1..Pn> ForeignTrait<T1..Tm> for T0` is allowed only if at least one `Ti` is a local type and no `T` before the first such `Ti` is one of the generic types `P1..Pn`:

Generic type parameters (`P1..Pn`) are allowed to appear in `T0..Ti` as long as they are covered by some intermediate type. A `T` is covered if it appears as a type parameter to some other type (like `Vec<T>`), but not if it stands on its own (just `T`) or just appears behind a fundamental type like `&T`.

Orphan rule: Covered Implementations

A clarification example:

```rust
// 'X, Y, .., Z' - some generics
// 'A, B, .., C' - some local types
impl<X, Y, .., Z> ForeignTrait<u32, A, B, Vec<X>, C> for Vec<i32> {
    /* ... */
}
```

Orphan rule: Covered Implementations

Note that:

```rust
impl<T> ForeignTrait<LocalType, T> for ForeignType {}
```

Is valid, but:

```rust
impl<T> ForeignTrait<T, LocalType> for ForeignType {}
```

Is not! Without “generic comes after local” rule, we could the code above, and another crate could write:

```rust
impl<T> ForeignTrait<TheirType, T> for ForeignType {}
```

And a conflict would arise only when the two crates were brought together. The orphan rule requires that your local type come before the type parameter.

```rust
// Note the 'pub'
pub mod one {
    pub mod nested {
        pub mod nested2 {
            pub(crate) struct Foo { bar: crate::two::Bar }
            impl crate::one::MyTrait for Foo {}
        }
        pub(self) enum Count { Example(nested2::Foo) }
    }
    trait MyTrait { /* ... */ }
}
```

Imagine we've put this in `lib.rs` file and published our crate. Since we added a `pub` keyword before `one`, and `one` is in our root module, everything inside became accessible for foreign crates. We don't want users to access `Foo`. One of the ways it to add `pub(crate)` visibility to `Foo`. Works just like `pub`, but only in our crate.

Modules

```
pub mod one {
    pub mod nested {
        pub mod nested2 {
            pub(super) struct Foo { bar: crate::two::Bar }
            impl crate::one::MyTrait for Foo {}
        }
        pub(self) enum Count { Example(nested2::Foo) }
    }
    trait MyTrait { /* ... */ }
}
```

If we don't use `Foo` in any other modules than `nested2` and `nested`, there's `pub(super)` to help. It makes object available only for current module and parent module.

```rust
pub mod one {
    pub mod nested {
        pub mod nested2 {
            pub(in crate::one::nested) struct Foo { /* ... */ }
            impl crate::one::MyTrait for Foo {}
        }
        pub(self) enum Count { Example(nested2::Foo) }
    }
    trait MyTrait { /* ... */ }
}
```

We can use `pub(in PATH)` to make the object visible for all ancestors from specified. For instance, in this example, we won't see `Foo` in module `one`, but all other ancestor modules will.

The availability of the type and definition is not the same in Rust. When you make the function public, you must have your input and output types to be at least with the same availability.

The same applies to enumerations and structures.

```rust
mod A {
    pub mod B {
        enum Private { X }
        pub fn MyFunc(x: Private) {}
        pub enum MyEnum { Variant(Private) }
        pub struct S { pub x: Private }
    }
}
```

use keyword

```
error[E0446]: private type `Private` in public interface
 --> src/main.rs:4:9
  |
3 | enum Private { X }
  | ------------ `Private` declared as private
4 | pub fn MyFunc(x: Private) {}
  | ^^^^^^^^^^^^^^^^^^^^^^^^^ can't leak private type
6 | pub struct S { pub x: Private }
  | ^^^^^^^^^^^^^^ can't leak private type
```

use keyword

```
warning: private type `Private` in public interface (error E0446)
 --> src/main.rs:5:35
  |
5 | pub enum MyEnum { Variant(Private) }
  |                           ^^^^^^^
  |
 = note: `#[warn(private_in_public)]` on by default
 = warning: this was previously accepted by the compiler
 but is being phased out; it will become a hard error in
 a future release!
 = note: for more information, see issue #34537
 <https://github.com/rust-lang/rust/issues/34537>
```

use keyword

In C++, this code will compile just fine:

```cpp
class Class {
private:
    struct Example {};

public:
    static Example get() {
        return Example {};
    }
};

int main() {
    auto example = Class::get();
}
```

Modules and files

We've already seen `super` and `crate` keywords, and they was pretty close to Unix paths. It's not a coincidence! This code...

```
mod one {
    pub mod nested {
        pub mod nested2 {
            pub struct Foo { bar: crate::two::Bar }
        }
        impl crate::one::MyTrait for nested2::Foo {}
        pub(self) enum Count { Example(nested2::Foo) }
    }
    trait MyTrait { /* ... */ }
}
mod two {
    pub struct Bar { /* ... */ }
    pub fn use_me() { /* ... */ }
}
```

Modules and files

...Translates to this in filesystem!

```
.
├── Cargo.toml
└── src
    ├── lib.rs
    ├── one
    │   ├── mod.rs
    │   └── nested
    │       ├── mod.rs
    │       └── nested2.rs
    └── two.rs
```

Modules and files

Well, how this works?

- Every file is a module. The path to this file including it's name is module name. Exceptions - `main.rs`, `lib.rs` and `mod.rs` files. Their names "empty" for Rust, and everything inside them is in the root module.

Well, how this works?

- Every file is a module. The path to this file including it's name is module name. Exceptions - `main.rs`, `lib.rs` and `mod.rs` files. Their names "empty" for Rust, and everything inside them is in the root module.
- For instance, `two.rs` file has path `crate::two`, and `nested2.rs` - `crate::one::nested::nested2`.

Modules and files

Well, how this works?

- When your module contains not only code but also other modules, you should create a directory. Inside it, you'll have `mod.rs` file in which declarations will have the path of the directory.

Modules and files

Well, how this works?

- When your module contains not only code but also other modules, you should create a directory. Inside it, you'll have `mod.rs` file in which declarations will have the path of the directory.
- For instance, code inside `mod.rs` in `nested` have module path of `crate::one::nested`.

Modules and files

Well, how this works?

- Modules aren't available to the whole program by default! To include module in file, use `pub mod MODULE;` syntax: because there's no `{}`, Rust finds out it's not a declaration but usage of the module. File `src/one/mod.rs` contains:

```rust
pub mod nested;
trait MyTrait { /* ... */ }
```

use keyword

There's one convenient thing - `use` keyword.

- If you want to use the name, you may want to write `use`. It's not required, but you've already seen it in homework that, for instance, writing `use std::rc::Rc` and then `Rc` is much better than writing `std::rc::Rc` everywhere.

```rust
use std::rc::Rc;
let r = Rc::new(/* ... */);
```

use keyword

There's one convenient thing - `use` keyword.

- If you want to use the name, you may want to write `use`. It's not required, but you've already seen it in homework that, for instance, writing `use std::rc::Rc` and then `Rc` is much better than writing `std::rc::Rc` everywhere.

```rust
use std::rc::Rc;
let r = Rc::new(/* ... */);
```

- We can give an alias to the `use` declaration.

```rust
use one::nested::nested2::Foo as Test;
let _ = Test {};
```

use keyword

There's one convenient thing - `use` keyword.

- The `impl` blocks for structures are available when the structure is available, and trait implementations are available when both structure and the trait are available. Moreover, to use the trait, you should first import it. When importing a trait, consider *private-importing* it: it enables trait methods but won't add a name in the scope.

```
use std::io::{Write as _, BufWriter};
```

use keyword

There's one convenient thing - `use` keyword.

- We can import all definitions in specified module, but not nested modules, using the keyword `self`:

```
use std::collections::{self};
let map = HashMap::new();
// Won't compile: hash_map is nested
// let mut hasher = hash_map::DefaultHasher::new();
```

use keyword

There's one convenient thing - `use` keyword.

- Or import everything inside the module by using `*`:

```
mod A {
    pub mod B {
        pub enum C { X, Y }
    }
}

use A::*;
let x = B::C::X;

// Importing specific variant
// use A::B::C::X
// let x = X;
```

use keyword

There's one convenient thing - `use` keyword.

- If you want to import enum variants, you can do it by using `*`. It's how it's done in the standard library prelude with `Option` variants `Some` and `None`:

```rust
mod A {
    pub mod B {
        pub enum C { X, Y }
    }
}

use A::B::C*;
let x = X;
```

`std::prelude`

`std` is a crate too. It's version depends on your compiler version.

When you're writing a Rust program, you don't have to write `use std::vec::Vec` or `use std::boxed::Box`. Since these imports are frequently used, they're added to your code by default by a prelude.

Prelude is a module. You don't see that, but your program by default contains `use std::prelude::*`. Inside, there's some necessary imports marked by `pub`:

```rust
pub use std::vec::Vec;
```

std::prelude

More specific example:

```rust
// Importing mycrate as private module
mod mycrate;
// Making 'Foo' public for other crate
pub use mycrate::nested::Foo;

// ...

// In some downstream crate
use mycrate::Foo;
```

If you'll ever have your not very small crate, it's a good idea to add your own prelude.

Moreover, `std` tries to be as small as possible, providing only a necessary minimum of traits and functions.

`std`

Moreover, `std` tries to be as small as possible, providing only a necessary minimum of traits and functions.

For instance, things like `HashMap` appear in almost any program.

Moreover, `std` tries to be as small as possible, providing only a necessary minimum of traits and functions.

For instance, things like `HashMap` appear in almost any program.

Things like asynchronous runtimes can be designed in many ways, but Rust wants them to be consistent with each other, and provide things like `Future` trait, keywords `async` and `await`, macros `join!` and `select!`. We'll discuss them in later lectures.

Iterators and closures

`Iterator` trait

In Rust, the iterator is not some specific structure - it's a trait.

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Returns `None` when iteration is finished. Individual iterator implementations may choose to resume iteration, and so calling `next()` again may or may not eventually start returning `Some(Item)` again at some point.

Iterator trait

Don't hurry, it's not all methods :D

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;

    fn size_hint(&self) -> (usize, Option<usize>) { ... }
    fn last(self) -> Option<Self::Item> { ... }
    fn enumerate(self) -> Enumerate<Self> { ... }
    fn peekable(self) -> Peekable<Self> { ... }
    fn map<B, F>(self, f: F) -> Map<Self, F> { ... }
    fn lt<I>(self, other: I) -> bool { ... }
    // A total of 71 methods!
}
```

`Iterator` trait

Don't hurry, it's not all methods :D

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;

    fn size_hint(&self) -> (usize, Option<usize>) { ... }
    fn last(self) -> Option<Self::Item> { ... }
    fn enumerate(self) -> Enumerate<Self> { ... }
    fn peekable(self) -> Peekable<Self> { ... }
    fn map<B, F>(self, f: F) -> Map<Self, F> { ... }
    fn lt<I>(self, other: I) -> bool { ... }
    // A total of 71 methods!
}
```

Before we continue, the most important thing you should know about iterators is that they are lazy and strictly give ownership of elements to you.

`Iterator` trait: `&self` methods

- `size_hint` - returns lower and upper iteration bound. Used for optimizations, should not be trusted (e.q checking bounds in unsafe code).

`Iterator` trait: `self` methods

- `last` - returns the last element (if there's some).

`Iterator` trait: `self` methods

- `last` - returns the last element (if there’s some).
- `enumerate` - returns iterator which `Item` is `(i, val)`, i.e enumerates the elements starting from zero.

`Iterator` trait: `self` methods

- `last` - returns the last element (if there's some).
- `enumerate` - returns iterator which `Item` is `(i, val)`, i.e enumerates the elements starting from zero.
- `peekable` - returns iterator which is peekable, i.e have `peek()` method.

`Iterator` trait: `self` methods

- `last` - returns the last element (if there's some).
- `enumerate` - returns iterator which `Item` is `(i, val)`, i.e enumerates the elements starting from zero.
- `peekable` - returns iterator which is peekable, i.e have `peek()` method.
- `count` - count number of elements in iterator

`Iterator` trait: `self` methods

- `last` - returns the last element (if there's some).
- `enumerate` - returns iterator which `Item` is `(i, val)`, i.e enumerates the elements starting from zero.
- `peekable` - returns iterator which is peekable, i.e have `peek()` method.
- `count` - count number of elements in iterator
- `lt` - compares two iterators by "less". Note it's **not lazy** operation.

`Iterator` trait: `self` methods

- `last` - returns the last element (if there's some).
- `enumerate` - returns iterator which `Item` is `(i, val)`, i.e enumerates the elements starting from zero.
- `peekable` - returns iterator which is peekable, i.e have `peek()` method.
- `count` - count number of elements in iterator
- `lt` - compares two iterators by "less". Note it's **not lazy** operation.
- `filter` - changes iterator of iterators to just plain iterator.
- And more!

`Iterator` trait: `&mut self` methods

- `map` - returns iterator which maps all elements to the new ones. Note this is **lazy** operation.

`Iterator` trait: `&mut self` methods

- `map` - returns iterator which maps all elements to the new ones. Note this is **lazy** operation.
- `by_ref` - returns `&mut Self` link, allowing to call `next` without having an ownership.

`Iterator` trait: `&mut self` methods

- `map` - returns iterator which maps all elements to the new ones. Note this is **lazy** operation.
- `by_ref` - returns `&mut Self` link, allowing to call `next` without having an ownership.
- `nth` - returns n-th element of the iterator.

`Iterator` trait: `&mut self` methods

- `map` - returns iterator which maps all elements to the new ones. Note this is **lazy** operation.
- `by_ref` - returns `&mut Self` link, allowing to call `next` without having an ownership.
- `nth` - returns n-th element of the iterator.
- `all` - tests whether all elements of the iterator match a predicate.
- And more!

`IntoIterator` trait

To abstract over objects that **can be turned into iterator**, there's `IntoIterator` trait.

```
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;
    fn into_iter(self) -> Self::IntoIter;
}
```

Note that we are consuming `self` and returning an iterator that returns owned types.

IntoIterator trait

For instance, this will create an iterator from vector, **consuming** the iterator:

```
let vec = vec![1, 2, 3];
let into_iter = vec.into_iter();
// This won't compile: the vector is moved!
// let first = vec[0];
```

IntoIterator trait

IntoIterator is used in for loop. So, really for loop:

```
for v in vec {
    // Code
}
```

Desugars to:

```
let mut __into_iter = vec.into_iter();
    while let Some(v) = __into_iter.next() {
    // Code
}
```

`IntoIterator` trait

So, if you want to use your collection in `for` loop, the “overload” is `intoIterator` trait.

Since any `Iterator` can be converted to `IntoInterator` by just returning itself, iterators can be used in `for` loop.

`IntoIterator` trait

Question: How this works?

```rust
for v in &vec {
    // Code
}
for v in &mut vec {
    // Code
}
```

`IntoIterator` trait

Question: How this works?

```
for v in &vec {
    // Code
}
for v in &mut vec {
    // Code
}
```

`IntoInterator` is implemented for both `&Vec` and `&mut Vec`, returning the same as `.iter()` and `.iter_mut()` accordingly.

Range

`a..b` syntax is syntax sugar for `Range`, `RangeFrom`, `RangeTo` and `RangeFull` declarations.

The first two are iterators since we know how to iterate. The last two are used for matching.

```
for i in 0..10 {
    // Code
}
for i in 10.. {
    // Code
}
```

Pythagorean Triplets

Iterators in Rust are powerful and enable you to write code in a functional style!

```
let triplets = (1u32..)
    .flat_map(|z| (1..=z).map(move |y| (y, z)))
    .flat_map(|(y, z)| (1..=y).map(move |x| (x, y, z)))
    .filter(|(x, y, z)| x*x + y*y == z*z);

let first_ten: Vec<(u32, u32, u32)> =
    triplets.take(10).collect();

// [(3, 4, 5), (6, 8, 10) ... (20, 21, 29)]
println!("{}", first_ten)
```

`FromIterator` trait

We can convert not only to iterator but from the iterator too!

```
pub trait FromIterator<A> {
    fn from_iter<T>(iter: T) -> Self
    where
        T: IntoIterator<Item = A>;
}
```

This will create a new collection A from iterator T. For instance:

```
let vec = vec![(0, 1), (1, 2)];
let map: HashMap<i32, i32> = HashMap::from_iter(vec);
```

`FromIterator` trait

It's exactly how `.collect()` function works: it expects some collection as generic which can be constructed from iterator with such `Item`.

```
fn collect<B: FromIterator<Self::Item>>(self) -> B
where
    Self: Sized,
{ /* ... */ }
```

`FromIterator` trait: `Result`

Also, this trait is implemented for quite unexpected types: `Result`, `Option` and `()`.

```
impl<A, E, V> FromIterator<Result<A, E>> for Result<V, E>
where
    V: FromIterator<A>,
{ /* ... */ }
```

Example:

```
let integers: Vec<&str> = vec!["0", "17", "2", "42"];
let res: Result<Vec<u32>, ParseIntError> = integers
    .into_iter()
    .map(|x| x.parse())
    // ^- impl Iterator<Item = Result<u32, ParseIntError>>;
    .collect();
assert_eq!(res, Ok(vec![0, 17, 2, 42]));
```

FromIterator trait: Option

```
impl<A, V> FromIterator<Option<A>> for Option<V>
where
    V: FromIterator<A>
{ /* ... */ }
```

Example:

```
let v: Vec<u32> = vec![1, 2, 11, 12];
let res: Option<Vec<u32:> = v.iter()
    .map(|x| x.checked_sub(1)).collect();
assert_eq!(res, Some(vec![0, 1, 10, 11]));

let v: Vec<u32> = vec![1, 2, 0, 12];
let res: Option<Vec<u32:> = v.iter()
    .map(|x| x.checked_sub(1)).collect();
assert_eq!(res, None);
```

FromIterator trait: ()

```
impl FromIterator<()> for () { /* ... */ }
```

Example:

```
let data = vec![1, 2, 3, 4, 5];
let res: io::Result<()> = data
    .into_iter()
    // Alternatively: try_for_each
    .map(|x| writeln!(stdout(), "{}", x))
    .collect();
assert!(res.is_ok());
```

`ExactSizeIterator` trait

In Rust, we have `ExactSizeIterator` trait, which means our iterator exactly knows its length.

```
pub trait ExactSizeIterator: Iterator {
    fn len(&self) -> usize { ... }
    fn is_empty(&self) -> bool { ... }
}
```

Note that this trait is safe and as such does not and cannot guarantee that the returned length is correct: unsafe code must not rely on the correctness of its implementation.

`DoubleEndedIterator` trait

There's a type of iterator that can be iterated from both sides!

```
pub trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;

    fn advance_back_by(&mut self, n: usize) -> Result<(), usize>;
    fn nth_back(&mut self, n: usize) -> Option<Self::Item>;
    fn try_rfold<B, F, R>(&mut self, init: B, f: F) -> R;
    fn rfold<B, F>(self, init: B, f: F) -> B;
    fn rfind<P>(&mut self, predicate: P) -> Option<Self::Item>;
}
```

`DoubleEndedIterator` trait

Example:

```
let data = vec![1, 2, 3];
let mut iter = data.iter();
assert_eq!(iter.next(), Some(&1));
assert_eq!(iter.next(), Some(&2));
assert_eq!(iter.next_back(), Some(&3));
// Asserion failed: left = None, right = Some(3)
// assert_eq!(iter.next_back(), Some(&3));
```

Iterators and performance

- As we seen in the lecture and will see in homework, it's difficult fo write iterator type.
- `size_of` complex iterator is `size_of` it's parts.
- All the state is just flags in the stack.
- Since compiler knows what functions are calling, it inlines calls and in result code works fast!
- Moreover, the iterator code is vectorized. For instance, `flatten` is specialized to be vectorized efficiently.

Iterators in C++

<table>
<tr><th colspan="4">category</th><th>properties</th><th>valid expressions</th></tr>
<tr><td colspan="4" rowspan="2">all categories</td><td>copy-constructible, copy-assignable and destructible</td><td><code>X b(a);</code>
<code>b = a;</code></td></tr>
<tr><td>Can be incremented</td><td><code>++a</code>
<code>a++</code></td></tr>
<tr><td rowspan="10">Random Access</td><td rowspan="6">Bidirectional</td><td rowspan="5">Forward</td><td rowspan="2">Input</td><td>Supports equality/inequality comparisons</td><td><code>a == b</code>
<code>a != b</code></td></tr>
<tr><td>Can be dereferenced as an rvalue</td><td><code>*a</code>
<code>a->m</code></td></tr>
<tr><td>Output</td><td>Can be dereferenced as an lvalue
(only for mutable iterator types)</td><td><code>*a = t</code>
<code>*a++ = t</code></td></tr>
<tr><td rowspan="2"></td><td>default-constructible</td><td><code>X a;</code>
<code>X()</code></td></tr>
<tr><td>Multi-pass: neither dereferencing nor incrementing affects dereferenceability</td><td><code>{ b=a; *a++; *b; }</code></td></tr>
<tr><td colspan="2"></td><td>Can be decremented</td><td><code>--a</code>
<code>a--</code>
<code>*a--</code></td></tr>
<tr><td colspan="3" rowspan="4"></td><td>Supports arithmetic operators + and -</td><td><code>a + n</code>
<code>n + a</code>
<code>a - n</code>
<code>a - b</code></td></tr>
<tr><td>Supports inequality comparisons (<, >, <= and >=) between iterators</td><td><code>a < b</code>
<code>a > b</code>
<code>a <= b</code>
<code>a >= b</code></td></tr>
<tr><td>Supports compound assignment operations += and -=</td><td><code>a += n</code>
<code>a -= n</code></td></tr>
<tr><td>Supports offset dereference operator (<code>[]</code>)</td><td><code>a[n]</code></td></tr>
</table>

Iterators in C++

Why we don't want this in Rust?

- C++ iterators are universal, but their implementation totally violates the iterator concept, since you're actually iterating only in the case of "Input iterator".

Iterators in C++

Why we don’t want this in Rust?

- C++ iterators are universal, but their implementation totally violates the iterator concept, since you’re actually iterating only in the case of “Input iterator”.
- Since C++ is unsafe and don’t have ownership like in Rust, they are always returning a reference, and moving object from the collection can be dangerous! But at the same time, almost all types of iterators allow that.

Why we don’t want this in Rust?

- C++ iterators are universal, but their implementation totally violates the iterator concept, since you’re actually iterating only in the case of “Input iterator”.
- Since C++ is unsafe and don’t have ownership like in Rust, they are always returning a reference, and moving object from the collection can be dangerous! But at the same time, almost all types of iterators allow that.
- It doesn’t make sense to create traits for all types of “iterators” like in C++ since iterator should *hide* details rather than *expose* them, and moreover - it can give us misguided hopes.

Iterators in C++

For instance, a list can also give us a random access iterator, but very inefficient. So, this code will compile but work extremely slowly:

```cpp
template <class RandomIt>
void func(RandomIt begin, RandomIt end) {
    // ...
    std::sort(begin, end);
    // ...
}
```

In Rust, you must know what collection you are using, and it's less error-prone.