Introduction

Welcome to The Rust Edition Guide! "Editions" are Rust's way of introducing changes into the language that would not otherwise be backwards compatible.

In this guide, we'll discuss:

- What editions are
- Which changes are contained in each edition
- How to migrate your code from one edition to another

What are Editions?

The release of Rust 1.0 (in May 2015) established "stability without stagnation" as a core Rust deliverable. Ever since the 1.0 release, the rule for Rust has been that once a feature has been released on stable, we are committed to supporting that feature for all future releases.

There are times, however, when it is useful to be able to make small changes to the language that are not backwards compatible. The most obvious example is introducing a new keyword, which would invalidate variables with the same name. For example, the first version of Rust did not have the `async` and `await` keywords. Suddenly changing those words to keywords in a later version would've broken code like `let async = 1;`.

Editions are the mechanism we use to solve this problem. When we want to release a feature that would otherwise be backwards incompatible, we do so as part of a new Rust *edition*. Editions are opt-in, and so existing crates do not see these changes until they explicitly migrate over to the new edition. This means that even the latest version of Rust will still *not* treat `async` as a keyword, unless edition 2018 or later is chosen. This choice is made *per crate* as part of its `Cargo.toml`. New crates created by `cargo new` are always configured to use the latest stable edition.

Editions do not split the ecosystem

The most important rule for editions is that crates in one edition can interoperate seamlessly with crates compiled in other editions. This ensures that the decision to migrate to a newer edition is a "private one" that the crate can make without affecting others.

The requirement for crate interoperability implies some limits on the kinds of changes that we can make in an edition. In general, changes that occur in an edition tend to be "skin deep". All Rust code, regardless of edition, is ultimately compiled to the same internal representation within the compiler.

Edition migration is easy and largely automated

Our goal is to make it easy for crates to upgrade to a new edition. When we release a new edition, we also provide tooling to automate the migration. It makes minor changes to your code necessary to make it compatible with the new edition. For example, when migrating to Rust 2018, it changes anything named `async` to use the equivalent raw identifier syntax: `r#async`.

The automated migrations are not necessarily perfect: there might be some corner cases where manual changes are still required. The tooling tries hard to avoid changes to semantics that could affect the correctness or performance of the code.

In addition to tooling, we also maintain this Rust Edition Guide that covers the changes that are part of an edition. This guide describes each change and gives pointers to where you can learn more about it. It also covers any corner cases or details you should be aware of. This guide serves as an overview of editions, as a migration guide for specific editions, and as a quick troubleshooting reference if you encounter problems with the automated tooling.

Creating a new project

A new project created with Cargo is configured to use the latest edition by default:

```
$ cargo new foo
     Created binary (application) `foo` project
$ cat foo/Cargo.toml
[package]
name = "foo"
version = "0.1.0"
edition = "2021"

[dependencies]
```

That `edition = "2021"` setting configures your package to be built using the Rust 2021 edition. No further configuration needed!

You can use the `--edition <YEAR>` option of `cargo new` to create the project using some specific edition. For example, creating a new project to use the Rust 2018 edition could be done like this:

```
$ cargo new --edition 2018 foo
     Created binary (application) `foo` project
$ cat foo/Cargo.toml
[package]
name = "foo"
version = "0.1.0"
edition = "2018"

[dependencies]
```

Don't worry about accidentally using an invalid year for the edition; the `cargo new` invocation will not accept an invalid edition year value:

```
$ cargo new --edition 2019 foo
error: "2019" isn't a valid value for '--edition <YEAR>'
        [possible values: 2015, 2018, 2021]

        Did you mean "2018"?

For more information try --help
```

You can change the value of the `edition` key by simply editing the `Cargo.toml` file. For example, to cause your package to be built using the Rust 2015 edition, you would set the key as in the following example:

```
[package]
name = "foo"
version = "0.1.0"
edition = "2015"

[dependencies]
```

Transitioning an existing project to a new edition

Rust includes tooling to automatically transition a project from one edition to the next. It will update your source code so that it is compatible with the next edition. Briefly, the steps to update to the next edition are:

1. Run `cargo fix --edition`
2. Edit `Cargo.toml` and set the `edition` field to the next edition, for example `edition = "2021"`
3. Run `cargo build` or `cargo test` to verify the fixes worked.

The following sections dig into the details of these steps, and some of the issues you may encounter along the way.

> It's our intention that the migration to new editions is as smooth an experience as possible. If it's difficult for you to upgrade to the latest edition, we consider that a bug. If you run into problems with this process, please file a bug report. Thank you!

Starting the migration

As an example, let's take a look at transitioning from the 2015 edition to the 2018 edition. The steps are essentially the same when transitioning to other editions like 2021.

Imagine we have a crate that has this code in `src/lib.rs`:

```
trait Foo {
    fn foo(&self, i32);
}
```

This code uses an anonymous parameter, that `i32`. This is not supported in Rust 2018, and so this would fail to compile. Let's get this code up to date!

Updating your code to be compatible with the new edition

Your code may or may not use features that are incompatible with the new edition. In order to help transition to the next edition, Cargo includes the `cargo fix` subcommand to automatically update your source code. To start, let's run it:

```
cargo fix --edition
```

This will check your code, and automatically fix any issues that it can. Let's look at `src/lib.rs` again:

```
trait Foo {
    fn foo(&self, _: i32);
}
```

It's re-written our code to introduce a parameter name for that `i32` value. In this case, since it had no name, `cargo fix` will replace it with `_`, which is conventional for unused variables.

`cargo fix` can't always fix your code automatically. If `cargo fix` can't fix something, it will print the warning that it cannot fix to the console. If you see one of these warnings, you'll have to update your code manually. See the Advanced migration strategies chapter for more on working with the migration process, and read the chapters in this guide which explain which changes are needed. If you have problems, please seek help at the user's forums.

Enabling the new edition to use new features

In order to use some new features, you must explicitly opt in to the new edition. Once you're ready to continue, change your `Cargo.toml` to add the new `edition` key/value pair. For example:

```
[package]
name = "foo"
version = "0.1.0"
edition = "2018"
```

If there's no `edition` key, Cargo will default to Rust 2015. But in this case, we've chosen `2018`, and so our code will compile with Rust 2018!

The next step is to test your project on the new edition. Run your project tests to verify that everything still works, such as running `cargo test`. If new warnings are issued, you may want

to consider running `cargo fix` again (without the `--edition` flag) to apply any suggestions given by the compiler.

Congrats! Your code is now valid in both Rust 2015 and Rust 2018!

Advanced migration strategies

How migrations work

`cargo fix --edition` works by running the equivalent of `cargo check` on your project with special lints enabled which will detect code that may not compile in the next edition. These lints include instructions on how to modify the code to make it compatible on both the current and the next edition. `cargo fix` applies these changes to the source code, and then runs `cargo check` again to verify that the fixes work. If the fixes fail, then it will back out the changes and display a warning.

Changing the code to be simultaneously compatible with both the current and next edition makes it easier to incrementally migrate the code. If the automated migration does not completely succeed, or requires manual help, you can iterate while staying on the original edition before changing `Cargo.toml` to use the next edition.

The lints that `cargo fix --edition` apply are part of a lint group. For example, when migrating from 2018 to 2021, Cargo uses the `rust-2021-compatibility` group of lints to fix the code. Check the Partial migration section below for tips on using individual lints to help with migration.

`cargo fix` may run `cargo check` multiple times. For example, after applying one set of fixes, this may trigger new warnings which require further fixes. Cargo repeats this until no new warnings are generated.

Migrating multiple configurations

`cargo fix` can only work with a single configuration at a time. If you use Cargo features or conditional compilation, then you may need to run `cargo fix` multiple times with different flags.

For example, if you have code that uses `#[cfg]` attributes to include different code for different platforms, you may need to run `cargo fix` with the `--target` option to fix for different targets. This may require moving your code between machines if you don't have cross-compiling available.

Similarly, if you have conditions on Cargo features, like `#[cfg(feature = "my-optional-thing")]`, it is recommended to use the `--all-features` flag to allow `cargo fix` to migrate all the code behind those feature gates. If you want to migrate feature code individually, you can use the `--features` flag to migrate one at a time.

Migrating a large project or workspace

You can migrate a large project incrementally to make the process easier if you run into problems.

In a Cargo workspace, each package defines its own edition, so the process naturally involves migrating one package at a time.

Within a Cargo package, you can either migrate the entire package at once, or migrate individual Cargo targets one at a time. For example, if you have multiple binaries, tests, and examples, you can use specific target selection flags with `cargo fix --edition` to migrate just that one target. By default, `cargo fix` uses `--all-targets`.

For even more advanced cases, you can specify the edition for each individual target in `Cargo.toml` like this:

```
[[bin]]
name = "my-binary"
edition = "2018"
```

This usually should not be required, but is an option if you have a lot of targets and are having difficulty migrating them all together.

Partial migration with broken code

Sometimes the fixes suggested by the compiler may fail to work. When this happens, Cargo will report a warning indicating what happened and what the error was. However, by default it will automatically back out the changes it made. It can be helpful to keep the code in the broken state and manually resolve the issue. Some of the fixes may have been correct, and the broken fix maybe be *mostly* correct, but just need minor tweaking.

In this situation, use the `--broken-code` option with `cargo fix` to tell Cargo not to back out the changes. Then, you can go manually inspect the error and investigate what is needed to fix it.

Another option to incrementally migrate a project is to apply individual fixes separately, one at a time. You can do this by adding the individual lints as warnings, and then either running `cargo fix` (without the `--edition` flag) or using your editor or IDE to apply its suggestions if it supports "Quick Fixes".

For example, the 2018 edition uses the `keyword-idents` lint to fix any conflicting keywords. You can add `#![warn(keyword_idents)]` to the top of each crate (like at the top of `src/lib.rs` or `src/main.rs`). Then, running `cargo fix` will apply just the suggestions for that lint.

You can see the list of lints enabled for each edition in the lint group page, or run the `rustc -Whelp` command.

Migrating macros

Some macros may require manual work to fix them for the next edition. For example, `cargo fix --edition` may not be able to automatically fix a macro that generates syntax that does not work in the next edition.

This may be a problem for both proc macros and `macro_rules`-style macros. `macro_rules` macros can sometimes be automatically updated if the macro is used within the same crate, but there are several situations where it cannot. Proc macros in general cannot be automatically fixed at all.

For example, if we migrate a crate containing this (contrived) macro `foo` from 2015 to 2018, `foo` would not be automatically fixed.

```
#[macro_export]
macro_rules! foo {
    () => {
        let dyn = 1;
        println!("it is {}", dyn);
    };
}
```

When this macro is defined in a 2015 crate, it can be used from a crate of any other edition due to macro hygiene (discussed below). In 2015, `dyn` is a normal identifier and can be used without restriction.

However, in 2018, `dyn` is no longer a valid identifier. When using `cargo fix --edition` to migrate to 2018, Cargo won't display any warnings or errors at all. However, `foo` won't work when called from any crate.

If you have macros, you are encouraged to make sure you have tests that fully cover the macro's syntax. You may also want to test the macros by importing and using them in crates from multiple editions, just to ensure it works correctly everywhere. If you run into issues, you'll need to read through the chapters of this guide to understand how the code can be changed to work across all editions.

Macro hygiene

Macros use a system called "edition hygiene" where the tokens within a macro are marked with which edition they come from. This allows external macros to be called from crates of varying editions without needing to worry about which edition it is called from.

Let's take a closer look at the example above that defines a `macro_rules` macro using `dyn` as an identifier. If that macro was defined in a crate using the 2015 edition, then that macro works fine, even if it were called from a 2018 crate where `dyn` is a keyword and that would normally be a syntax error. The `let dyn = 1;` tokens are marked as being from 2015, and the compiler will remember that wherever that code gets expanded. The parser looks at the edition of the tokens to know how to interpret it.

The problem arises when changing the edition to 2018 in the crate where it is defined. Now, those tokens are tagged with the 2018 edition, and those will fail to parse. However, since we never called the macro from our crate, `cargo fix --edition` never had a chance to inspect the macro and fix it.

Documentation tests

At this time, `cargo fix` is not able to update documentation tests. After updating the edition in `Cargo.toml`, you should run `cargo test` to ensure everything still passes. If your documentation tests use syntax that is not supported in the new edition, you will need to update them manually.

In rare cases, you can manually set the edition for each test. For example, you can use the `edition2018` annotation on the triple backticks to tell `rustdoc` which edition to use.

Generated code

Another area where the automated fixes cannot apply is if you have a build script which generates Rust code at compile time (see Code generation for an example). In this situation, if

you end up with code that doesn't work in the next edition, you will need to manually change the build script to generate code that is compatible.

Migrating non-Cargo projects

If your project is not using Cargo as a build system, it may still be possible to make use of the automated lints to assist migrating to the next edition. You can enable the migration lints as described above by enabling the appropriate lint group. For example, you can use the `#![warn(rust_2021_compatibility)]` attribute or the `-Wrust-2021-compatibility` or `--force-warns=rust-2021-compatibility` CLI flag.

The next step is to apply those lints to your code. There are several options here:

- Manually read the warnings and apply the suggestions recommended by the compiler.
- Use an editor or IDE that supports automatically applying suggestions. For example, Visual Studio Code with the Rust Analyzer extension has the ability to use the "Quick Fix" links to automatically apply suggestions. Many other editors and IDEs have similar functionality.
- Write a migration tool using the `rustfix` library. This is the library that Cargo uses internally to take the JSON messages from the compiler and modify the source code. Check the `examples` directory for examples of how to use the library.

Writing idiomatic code in a new edition

Editions are not only about new features and removing old ones. In any programming language, idioms change over time, and Rust is no exception. While old code will continue to compile, it might be written with different idioms today.

For example, in Rust 2015, external crates must be listed with `extern crate` like this:

```
// src/lib.rs
extern crate rand;
```

In Rust 2018, it is no longer necessary to include these items.

`cargo fix` has the `--edition-idioms` option to automatically transition some of these idioms to the new syntax.

Warning: The current *"idiom lints"* are known to have some problems. They may make incorrect suggestions which may fail to compile. The current lints are:

- Edition 2018:
  - `unused-extern-crates`
  - `explicit-outlives-requirements`
- Edition 2021 does not have any idiom lints.

The following instructions are recommended only for the intrepid who are willing to work through a few compiler/Cargo bugs! If you run into problems, you can try the `--broken-code` option described above to make as much progress as possible, and then resolve the remaining issues manually.

With that out of the way, we can instruct Cargo to fix our code snippet with:

```
cargo fix --edition-idioms
```

Afterwards, the line with `extern crate rand;` in `src/lib.rs` will be removed.

We're now more idiomatic, and we didn't have to fix our code manually!

Rust 2015

Rust 2015 has a theme of "stability". It commenced with the release of 1.0, and is the "default edition". The edition system was conceived in late 2017, but Rust 1.0 was released in May of 2015. As such, 2015 is the edition that you get when you don't specify any particular edition, for backwards compatibility reasons.

"Stability" is the theme of Rust 2015 because 1.0 marked a huge change in Rust development. Previous to Rust 1.0, Rust was changing on a daily basis. This made it very difficult to write large software in Rust, and made it difficult to learn. With the release of Rust 1.0 and Rust 2015, we committed to backwards compatibility, ensuring a solid foundation for people to build projects on top of.

Since it's the default edition, there's no way to port your code to Rust 2015; it just *is*. You'll be transitioning *away* from 2015, but never really *to* 2015. As such, there's not much else to say about it!

Rust 2018

Info	
RFC	#2052, which also proposed the Edition system
Release version	1.31.0

The edition system was created for the release of Rust 2018. The release of the Rust 2018 edition coincided with a number of other features all coordinated around the theme of *productivity*. The majority of those features were backwards compatible and are now available on all editions; however, some of those changes required the edition mechanism (most notably the module system changes).

Path and module system changes

Summary

- Paths in `use` declarations now work the same as other paths.
- Paths starting with `::` must now be followed with an external crate.
- Paths in `pub(in path)` visibility modifiers must now start with `crate`, `self`, or `super`.

Motivation

The module system is often one of the hardest things for people new to Rust. Everyone has their own things that take time to master, of course, but there's a root cause for why it's so confusing to many: while there are simple and consistent rules defining the module system, their consequences can feel inconsistent, counterintuitive and mysterious.

As such, the 2018 edition of Rust introduces a few new module system features, but they end up *simplifying* the module system, to make it more clear as to what is going on.

Here's a brief summary:

- `extern crate` is no longer needed in 99% of circumstances.
- The `crate` keyword refers to the current crate.
- Paths may start with a crate name, even within submodules.
- Paths starting with `::` must reference an external crate.
- A `foo.rs` and `foo/` subdirectory may coexist; `mod.rs` is no longer needed when placing submodules in a subdirectory.
- Paths in `use` declarations work the same as other paths.

These may seem like arbitrary new rules when put this way, but the mental model is now significantly simplified overall. Read on for more details!

More details

Let's talk about each new feature in turn.

No more `extern crate`

This one is quite straightforward: you no longer need to write `extern crate` to import a crate into your project. Before:

```
// Rust 2015

extern crate futures;

mod submodule {
    use futures::Future;
}
```

After:

```
// Rust 2018

mod submodule {
    use futures::Future;
}
```

Now, to add a new crate to your project, you can add it to your `Cargo.toml`, and then there is no step two. If you're not using Cargo, you already had to pass `--extern` flags to give `rustc` the location of external crates, so you'd just keep doing what you were doing there as well.

> One small note here: `cargo fix` will not currently automate this change. We may have it do this for you in the future.

An exception

There's one exception to this rule, and that's the "sysroot" crates. These are the crates distributed with Rust itself.

Usually these are only needed in very specialized situations. Starting in 1.41, `rustc` accepts the `--extern=CRATE_NAME` flag which automatically adds the given crate name in a way similar to `extern crate`. Build tools may use this to inject sysroot crates into the crate's prelude. Cargo does not have a general way to express this, though it uses it for `proc_macro` crates.

Some examples of needing to explicitly import sysroot crates are:

- `std`: Usually this is not necessary, because `std` is automatically imported unless the crate is marked with `#![no_std]`.
- `core`: Usually this is not necessary, because `core` is automatically imported, unless the crate is marked with `#![no_core]`. For example, some of the internal crates used by the standard library itself need this.
- `proc_macro`: This is automatically imported by Cargo if it is a proc-macro crate starting in 1.42. `extern crate proc_macro;` would be needed if you want to support older releases, or if using another build tool that does not pass the appropriate `--extern` flags to `rustc`.
- `alloc`: Items in the `alloc` crate are usually accessed via re-exports in the `std` crate. If you are working with a `no_std` crate that supports allocation, then you may need to explicitly import `alloc`.
- `test`: This is only available on the nightly channel, and is usually only used for the unstable benchmark support.

Macros

One other use for `extern crate` was to import macros; that's no longer needed. Macros may be imported with `use` like any other item. For example, the following use of `extern crate`:

```
#[macro_use]
extern crate bar;

fn main() {
    baz!();
}
```

Can be changed to something like the following:

```
use bar::baz;

fn main() {
    baz!();
}
```

Renaming crates

If you've been using `as` to rename your crate like this:

```
extern crate futures as f;

use f::Future;
```

then removing the `extern crate` line on its own won't work. You'll need to do this:

```
use futures as f;

use self::f::Future;
```

This change will need to happen in any module that uses `f`.

The `crate` keyword refers to the current crate

In `use` declarations and in other code, you can refer to the root of the current crate with the `crate::` prefix. For instance, `crate::foo::bar` will always refer to the name `bar` inside the module `foo`, from anywhere else in the same crate.

The prefix `::` previously referred to either the crate root or an external crate; it now unambiguously refers to an external crate. For instance, `::foo::bar` always refers to the name `bar` inside the external crate `foo`.

Extern crate paths

Previously, using an external crate in a module without a `use` import required a leading `::` on the path.

```
// Rust 2015

extern crate chrono;

fn foo() {
    // this works in the crate root
    let x = chrono::Utc::now();
}

mod submodule {
    fn function() {
        // but in a submodule it requires a leading :: if not imported with `use`
        let x = ::chrono::Utc::now();
    }
}
```

Now, extern crate names are in scope in the entire crate, including submodules.

```
// Rust 2018

fn foo() {
    // this works in the crate root
    let x = chrono::Utc::now();
}

mod submodule {
    fn function() {
        // crates may be referenced directly, even in submodules
        let x = chrono::Utc::now();
    }
}
```

No more `mod.rs`

In Rust 2015, if you have a submodule:

```
// This `mod` declaration looks for the `foo` module in
// `foo.rs` or `foo/mod.rs`.
mod foo;
```

It can live in `foo.rs` or `foo/mod.rs`. If it has submodules of its own, it *must* be `foo/mod.rs`. So a `bar` submodule of `foo` would live at `foo/bar.rs`.

In Rust 2018 the restriction that a module with submodules must be named `mod.rs` is lifted. `foo.rs` can just be `foo.rs`, and the submodule is still `foo/bar.rs`. This eliminates the special name, and if you have a bunch of files open in your editor, you can clearly see their names, instead of having a bunch of tabs named `mod.rs`.

Rust 2015	Rust 2018
. ├── lib.rs └── foo/ ├── mod.rs └── bar.rs	. ├── lib.rs ├── foo.rs └── foo/ └── bar.rs

`use` paths

Minimum Rust Version 1.32

Rust 2018 simplifies and unifies path handling compared to Rust 2015. In Rust 2015, paths work differently in `use` declarations than they do elsewhere. In particular, paths in `use` declarations would always start from the crate root, while paths in other code implicitly started from the current scope. Those differences didn't have any effect in the top-level module, which meant that everything would seem straightforward until working on a project large enough to have submodules.

In Rust 2018, paths in `use` declarations and in other code work the same way, both in the top-level module and in any submodule. You can use a relative path from the current scope, a path starting from an external crate name, or a path starting with `crate`, `super`, or `self`.

Code that looked like this:

```
// Rust 2015

extern crate futures;

use futures::Future;

mod foo {
    pub struct Bar;
}

use foo::Bar;

fn my_poll() -> futures::Poll { ... }

enum SomeEnum {
    V1(usize),
    V2(String),
}

fn func() {
    let five = std::sync::Arc::new(5);
    use SomeEnum::*;
    match ... {
        V1(i) => { ... }
        V2(s) => { ... }
    }
}
```

will look exactly the same in Rust 2018, except that you can delete the `extern crate` line:

```
// Rust 2018

use futures::Future;

mod foo {
    pub struct Bar;
}

use foo::Bar;

fn my_poll() -> futures::Poll { ... }

enum SomeEnum {
    V1(usize),
    V2(String),
}

fn func() {
    let five = std::sync::Arc::new(5);
    use SomeEnum::*;
    match ... {
        V1(i) => { ... }
        V2(s) => { ... }
    }
}
```

The same code will also work completely unmodified in a submodule:

```
// Rust 2018

mod submodule {
    use futures::Future;

    mod foo {
        pub struct Bar;
    }

    use foo::Bar;

    fn my_poll() -> futures::Poll { ... }

    enum SomeEnum {
        V1(usize),
        V2(String),
    }

    fn func() {
        let five = std::sync::Arc::new(5);
        use SomeEnum::*;
        match ... {
            V1(i) => { ... }
            V2(s) => { ... }
        }
    }
}
```

This makes it easy to move code around in a project, and avoids introducing additional complexity to multi-module projects.

If a path is ambiguous, such as if you have an external crate and a local module or item with the same name, you'll get an error, and you'll need to either rename one of the conflicting names or explicitly disambiguate the path. To explicitly disambiguate a path, use `::name` for an external crate name, or `self::name` for a local module or item.

Anonymous trait function parameters deprecated

Minimum Rust Version 1.31

Summary

- Trait function parameters may use any irrefutable pattern when the function has a body.

Details

In accordance with RFC #1685, parameters in trait method declarations are no longer allowed to be anonymous.

For example, in the 2015 edition, this was allowed:

```
trait Foo {
    fn foo(&self, u8);
}
```

In the 2018 edition, all parameters must be given an argument name (even if it's just `_`):

```
trait Foo {
    fn foo(&self, baz: u8);
}
```

New keywords

Minimum Rust Version 1.27

Summary

- `dyn` is a strict keyword, in 2015 it is a weak keyword.
- `async` and `await` are strict keywords.
- `try` is a reserved keyword.

Motivation

`dyn Trait` for trait objects

The `dyn Trait` feature is the new syntax for using trait objects. In short:

- `Box<Trait>` becomes `Box<dyn Trait>`
- `&Trait` and `&mut Trait` become `&dyn Trait` and `&mut dyn Trait`

And so on. In code:

```
trait Trait {}

impl Trait for i32 {}

// old
fn function1() -> Box<Trait> {
}

// new
fn function2() -> Box<dyn Trait> {
}
```

That's it!

Why?

Using just the trait name for trait objects turned out to be a bad decision. The current syntax is often ambiguous and confusing, even to veterans, and favors a feature that is not more frequently used than its alternatives, is sometimes slower, and often cannot be used at all when its alternatives can.

Furthermore, with `impl Trait` arriving, "`impl Trait` vs `dyn Trait`" is much more symmetric, and therefore a bit nicer, than "`impl Trait` vs `Trait`". `impl Trait` is explained here.

In the new edition, you should therefore prefer `dyn Trait` to just `Trait` where you need a trait object.

`async` and `await`

These keywords are reserved to implement the async-await feature of Rust, which was ultimately released to stable in 1.39.0.

`try` keyword

The `try` keyword is reserved for use in `try` blocks, which have not (as of this writing) been stabilized (tracking issue)

Method dispatch for raw pointers to inference variables

Summary

- The `tyvar_behind_raw_pointer` lint is now a hard error.

Details

See Rust issue #46906 for details.

Cargo changes

Summary

- If there is a target definition in a `Cargo.toml` manifest, it no longer automatically disables automatic discovery of other targets.
- Target paths of the form `src/{target_name}.rs` are no longer inferred for targets where the `path` field is not set.
- `cargo install` for the current directory is no longer allowed, you must specify `cargo install --path .` to install the current package.

Rust 2021

Info	
RFC	#3085
Release version	1.56.0

The Rust 2021 Edition contains several changes that bring new capabilities and more consistency to the language, and opens up room for expansion in the future. The following chapters dive into the details of each change, and they include guidance on migrating your existing code.

Additions to the prelude

Summary

- The `TryInto`, `TryFrom` and `FromIterator` traits are now part of the prelude.
- This might make calls to trait methods ambiguous which could make some code fail to compile.

Details

The prelude of the standard library is the module containing everything that is automatically imported in every module. It contains commonly used items such as `Option`, `Vec`, `drop`, and `Clone`.

The Rust compiler prioritizes any manually imported items over those from the prelude, to make sure additions to the prelude will not break any existing code. For example, if you have a crate or module called `example` containing a `pub struct Option;`, then `use example::*;` will make `Option` unambiguously refer to the one from `example`; not the one from the standard library.

However, adding a *trait* to the prelude can break existing code in a subtle way. For example, a call to `x.try_into()` which comes from a `MyTryInto` trait might fail to compile if `std`'s `TryInto` is also imported, because the call to `try_into` is now ambiguous and could come from either trait. This is the reason we haven't added `TryInto` to the prelude yet, since there is a lot of code that would break this way.

As a solution, Rust 2021 will use a new prelude. It's identical to the current one, except for three new additions:

- `std::convert::TryInto`
- `std::convert::TryFrom`
- `std::iter::FromIterator`

The tracking issue can be found here.

Migration

As a part of the 2021 edition a migration lint, `rust_2021_prelude_collisions`, has been added in order to aid in automatic migration of Rust 2018 codebases to Rust 2021.

In order to have `rustfix` migrate your code to be Rust 2021 Edition compatible, run:

```
cargo fix --edition
```

The lint detects cases where functions or methods are called that have the same name as the methods defined in one of the new prelude traits. In some cases, it may rewrite your calls in various ways to ensure that you continue to call the same function you did before.

If you'd like to migrate your code manually or better understand what `rustfix` is doing, below we've outlined the situations where a migration is needed along with a counter example of when it's not needed.

Migration needed

Conflicting trait methods

When two traits that are in scope have the same method name, it is ambiguous which trait method should be used. For example:

```
trait MyTrait<A> {
  // This name is the same as the `from_iter` method on the `FromIterator` trait
from `std`.
  fn from_iter(x: Option<A>);
}

impl<T> MyTrait<()> for Vec<T> {
  fn from_iter(_: Option<()>) {}
}

fn main() {
  // Vec<T> implements both `std::iter::FromIterator` and `MyTrait`
  // If both traits are in scope (as would be the case in Rust 2021),
  // then it becomes ambiguous which `from_iter` method to call
  <Vec<i32>>::from_iter(None);
}
```

We can fix this by using fully qualified syntax:

```
fn main() {
  // Now it is clear which trait method we're referring to
  <Vec<i32> as MyTrait<()>>::from_iter(None);
}
```

Inherent methods on `dyn Trait` objects

Some users invoke methods on a `dyn Trait` value where the method name overlaps with a new prelude trait:

```
mod submodule {
  pub trait MyTrait {
    // This has the same name as `TryInto::try_into`
    fn try_into(&self) -> Result<u32, ()>;
  }
}

// `MyTrait` isn't in scope here and can only be referred to through the path
`submodule::MyTrait`
fn bar(f: Box<dyn submodule::MyTrait>) {
  // If `std::convert::TryInto` is in scope (as would be the case in Rust 2021),
  // then it becomes ambiguous which `try_into` method to call
  f.try_into();
}
```

Unlike with static dispatch methods, calling a trait method on a trait object does not require that the trait be in scope. The code above works as long as there is no trait in scope with a conflicting method name. When the `TryInto` trait is in scope (which is the case in Rust 2021), this causes an ambiguity. Should the call be to `MyTrait::try_into` or `std::convert::TryInto::try_into`?

In these cases, we can fix this by adding an additional dereferences or otherwise clarify the type of the method receiver. This ensures that the `dyn Trait` method is chosen, versus the methods from the prelude trait. For example, turning `f.try_into()` above into `(&*f).try_into()` ensures that we're calling `try_into` on the `dyn MyTrait` which can only refer to the `MyTrait::try_into` method.

No migration needed

Inherent methods

Many types define their own inherent methods with the same name as a trait method. For instance, below the struct `MyStruct` implements `from_iter` which shares the same name with

the method from the trait `FromIterator` found in the standard library:

```
use std::iter::IntoIterator;

struct MyStruct {
  data: Vec<u32>
}

impl MyStruct {
  // This has the same name as `std::iter::FromIterator::from_iter`
  fn from_iter(iter: impl IntoIterator<Item = u32>) -> Self {
    Self {
      data: iter.into_iter().collect()
    }
  }
}

impl std::iter::FromIterator<u32> for MyStruct {
    fn from_iter<I: IntoIterator<Item = u32>>(iter: I) -> Self {
      Self {
        data: iter.into_iter().collect()
      }
    }
}
```

Inherent methods always take precedent over trait methods so there's no need for any migration.

Implementation Reference

The lint needs to take a couple of factors into account when determining whether or not introducing 2021 Edition to a codebase will cause a name resolution collision (thus breaking the code after changing edition). These factors include:

- Is the call a fully-qualified call or does it use dot-call method syntax?
    - This will affect how the name is resolved due to auto-reference and auto-dereferencing on method call syntax. Manually dereferencing/referencing will allow specifying priority in the case of dot-call method syntax, while fully-qualified call requires specification of the type and the trait name in the method path (e.g. `<Type as Trait>::method`)
- Is this an inherent method or a trait method?
    - Inherent methods that take `self` will take priority over `TryInto::try_into` as inherent methods take priority over trait methods, but inherent methods that take `&self` or `&mut self` won't take priority due to requiring a auto-reference (while `TryInto::try_into` does not, as it takes `self`)

- Is the origin of this method from `core` / `std`? (As the traits can't have a collision with themselves)
- Does the given type implement the trait it could have a collision against?
- Is the method being called via dynamic dispatch? (i.e. is the `self` type `dyn Trait`)
    - If so, trait imports don't affect resolution, and no migration lint needs to occur

Default Cargo feature resolver

Summary

- `edition = "2021"` implies `resolver = "2"` in `Cargo.toml`.

Details

Since Rust 1.51.0, Cargo has opt-in support for a new feature resolver which can be activated with `resolver = "2"` in `Cargo.toml`.

Starting in Rust 2021, this will be the default. That is, writing `edition = "2021"` in `Cargo.toml` will imply `resolver = "2"`.

The resolver is a global setting for a workspace, and the setting is ignored in dependencies. The setting is only honored for the top-level package of the workspace. If you are using a virtual workspace, you will still need to explicitly set the `resolver` field in the `[workspace]` definition if you want to opt-in to the new resolver.

The new feature resolver no longer merges all requested features for crates that are depended on in multiple ways. See the announcement of Rust 1.51 for details.

Migration

There are no automated migration tools for updating for the new resolver. For most projects, there are usually few or no changes as a result of updating.

When updating with `cargo fix --edition`, Cargo will display a report if the new resolver will build dependencies with different features. It may look something like this:

> note: Switching to Edition 2021 will enable the use of the version 2 feature resolver in Cargo. This may cause some dependencies to be built with fewer features enabled than previously. More information about the resolver changes may be found at https://doc.rust-lang.org/nightly/edition-guide/rust-2021/default-cargo-resolver.html
> When building the following dependencies, the given features will no longer be used:

```
bstr v0.2.16: default, lazy_static, regex-automata, unicode
libz-sys v1.1.3 (as host dependency): libc
```

This lets you know that certain dependencies will no longer be built with the given features.

Build failures

There may be some circumstances where your project may not build correctly after the change. If a dependency declaration in one package assumes that certain features are enabled in another, and those features are now disabled, it may fail to compile.

For example, let's say we have a dependency like this:

```
# Cargo.toml

[dependencies]
bstr = { version = "0.2.16", default-features = false }
# ...
```

And somewhere in our dependency tree, another package has this:

```
# Another package's Cargo.toml

[build-dependencies]
bstr = "0.2.16"
```

In our package, we've been using the `words_with_breaks` method from `bstr`, which requires `bstr`'s "unicode" feature to be enabled. This has historically worked because Cargo unified the features of `bstr` between the two packages. However, after updating to Rust 2021, the new resolver will build `bstr` twice, once with the default features (as a build dependency), and once with no features (as our normal dependency). Since `bstr` is now being built without the "unicode" feature, the `words_with_breaks` method doesn't exist, and the build will fail with an error that the method is missing.

The solution here is to ensure that the dependency is declared with the features you are actually using. For example:

```
[dependencies]
bstr = { version = "0.2.16", default-features = false, features = ["unicode"] }
```

In some cases, this may be a problem with a third-party dependency that you don't have direct control over. You can consider submitting a patch to that project to try to declare the correct set of features for the problematic dependency. Alternatively, you can add features to any

dependency from within your own `Cargo.toml` file. For example, if the `bstr` example given above was declared in some third-party dependency, you can just copy the correct dependency declaration into your own project. The features will be unified, as long as they match the unification rules of the new resolver. Those are:

- Features enabled on platform-specific dependencies for targets not currently being built are ignored.
- Build-dependencies and proc-macros do not share features with normal dependencies.
- Dev-dependencies do not activate features unless building a target that needs them (like tests or examples).

A real-world example is using `diesel` and `diesel_migrations`. These packages provide database support, and the database is selected using a feature, like this:

```
[dependencies]
diesel = { version = "1.4.7", features = ["postgres"] }
diesel_migrations = "1.4.0"
```

The problem is that `diesel_migrations` has an internal proc-macro which itself depends on `diesel`, and the proc-macro assumes its own copy of `diesel` has the same features enabled as the rest of the dependency graph. After updating to the new resolver, it fails to build because now there are two copies of `diesel`, and the one built for the proc-macro is missing the "postgres" feature.

A solution here is to add `diesel` as a build-dependency with the required features, for example:

```
[build-dependencies]
diesel = { version = "1.4.7", features = ["postgres"] }
```

This causes Cargo to add "postgres" as a feature for host dependencies (proc-macros and build-dependencies). Now, the `diesel_migrations` proc-macro will get the "postgres" feature enabled, and it will build correctly.

The 2.0 release of `diesel` (currently in development) does not have this problem as it has been restructured to not have this dependency requirement.

Exploring features

The `cargo tree` command has had substantial improvements to help with the migration to the new resolver. `cargo tree` can be used to explore the dependency graph, and to see which features are being enabled, and importantly *why* they are being enabled.

One option is to use the `--duplicates` flag (`-d` for short), which will tell you when a package is being built multiple times. Taking the `bstr` example from earlier, we might see:

```
> cargo tree -d
bstr v0.2.16
└── foo v0.1.0 (/MyProjects/foo)

bstr v0.2.16
[build-dependencies]
└── bar v0.1.0
    └── foo v0.1.0 (/MyProjects/foo)
```

This output tells us that `bstr` is built twice, and shows the chain of dependencies that led to its inclusion in both cases.

You can print which features each package is using with the `-f` flag, like this:

```
cargo tree -f '{p} {f}'
```

This tells Cargo to change the "format" of the output, where it will print both the package and the enabled features.

You can also use the `-e` flag to tell it which "edges" to display. For example, `cargo tree -e features` will show in-between each dependency which features are being added by each dependency. This option becomes more useful with the `-i` flag which can be used to "invert" the tree. This allows you to see how features *flow* into a given dependency. For example, let's say the dependency graph is large, and we're not quite sure who is depending on `bstr`, the following command will show that:

```
> cargo tree -e features -i bstr
bstr v0.2.16
├── bstr feature "default"
│   [build-dependencies]
│   └── bar v0.1.0
│       └── bar feature "default"
│           └── foo v0.1.0 (/MyProjects/foo)
├── bstr feature "lazy_static"
│   └── bstr feature "unicode"
│       └── bstr feature "default" (*)
├── bstr feature "regex-automata"
│   └── bstr feature "unicode" (*)
├── bstr feature "std"
│   └── bstr feature "default" (*)
└── bstr feature "unicode" (*)
```

This snippet of output shows that the project `foo` depends on `bar` with the "default" feature. Then, `bar` depends on `bstr` as a build-dependency with the "default" feature. We can further

see that `bstr`'s "default" feature enables "unicode" (among other features).

IntoIterator for arrays

Summary

- Arrays implement `IntoIterator` in *all* editions.
- Calls to `IntoIterator::into_iter` are *hidden* in Rust 2015 and Rust 2018 when using method call syntax (i.e., `array.into_iter()`). So, `array.into_iter()` still resolves to `(&array).into_iter()` as it has before.
- `array.into_iter()` changes meaning to be the call to `IntoIterator::into_iter` in Rust 2021.

Details

Until Rust 1.53, only *references* to arrays implement `IntoIterator`. This means you can iterate over `&[1, 2, 3]` and `&mut [1, 2, 3]`, but not over `[1, 2, 3]` directly.

```
for &e in &[1, 2, 3] {} // Ok :)

for e in [1, 2, 3] {} // Error :(
```

This has been a long-standing issue, but the solution is not as simple as it seems. Just adding the trait implementation would break existing code. `array.into_iter()` already compiles today because that implicitly calls `(&array).into_iter()` due to how method call syntax works. Adding the trait implementation would change the meaning.

Usually this type of breakage (adding a trait implementation) is categorized as 'minor' and acceptable. But in this case there is too much code that would be broken by it.

It has been suggested many times to "only implement `IntoIterator` for arrays in Rust 2021". However, this is simply not possible. You can't have a trait implementation exist in one edition and not in another, since editions can be mixed.

Instead, the trait implementation was added in *all* editions (starting in Rust 1.53.0) but with a small hack to avoid breakage until Rust 2021. In Rust 2015 and 2018 code, the compiler will still resolve `array.into_iter()` to `(&array).into_iter()` like before, as if the trait implementation does not exist. This *only* applies to the `.into_iter()` method call syntax. It

does not affect any other syntax such as `for e in [1, 2, 3]`, `iter.zip([1, 2, 3])` or `IntoIterator::into_iter([1, 2, 3])`. Those will start to work in *all* editions.

While it's a shame that this required a small hack to avoid breakage, this solution keeps the difference between the editions to an absolute minimum.

Migration

A lint, `array_into_iter`, gets triggered whenever there is some call to `into_iter()` that will change meaning in Rust 2021. The `array_into_iter` lint has already been a warning by default on all editions since the 1.41 release (with several enhancements made in 1.55). If your code is already warning free, then it should already be ready to go for Rust 2021!

You can automatically migrate your code to be Rust 2021 Edition compatible or ensure it is already compatible by running:

```
cargo fix --edition
```

Because the difference between editions is small, the migration to Rust 2021 is fairly straightforward.

For method calls of `into_iter` on arrays, the elements being implemented will change from references to owned values.

For example:

```
fn main() {
  let array = [1u8, 2, 3];
  for x in array.into_iter() {
    // x is a `&u8` in Rust 2015 and Rust 2018
    // x is a `u8` in Rust 2021
  }
}
```

The most straightforward way to migrate in Rust 2021, is by keeping the exact behavior from previous editions by calling `iter()` which also iterates over owned arrays by reference:

```
fn main() {
  let array = [1u8, 2, 3];
  for x in array.iter() { // <- This line changed
    // x is a `&u8` in all editions
  }
}
```

Optional migration

If you are using fully qualified method syntax (i.e., `IntoIterator::into_iter(array)`) in a previous edition, this can be upgraded to method call syntax (i.e., `array.into_iter()`).

Disjoint capture in closures

Summary

- `|| a.x + 1` now captures only `a.x` instead of `a`.
- This can cause things to be dropped at different times or affect whether closures implement traits like `Send` or `Clone`.
    - If possible changes are detected, `cargo fix` will insert statements like `let _ = &a` to force a closure to capture the entire variable.

Details

Closures automatically capture anything that you refer to from within their body. For example, `|| a + 1` automatically captures a reference to `a` from the surrounding context.

In Rust 2018 and before, closures capture entire variables, even if the closure only uses one field. For example, `|| a.x + 1` captures a reference to `a` and not just `a.x`. Capturing `a` in its entirety prevents mutation or moves from other fields of `a`, so that code like this does not compile:

```
let a = SomeStruct::new();
drop(a.x); // Move out of one field of the struct
println!("{}", a.y); // Ok: Still use another field of the struct
let c = || println!("{}", a.y); // Error: Tries to capture all of `a`
c();
```

Starting in Rust 2021, closures captures are more precise. Typically they will only capture the fields they use (in some cases, they might capture more than just what they use, see the Rust reference for full details). Therefore, the above example will compile fine in Rust 2021.

Disjoint capture was proposed as part of RFC 2229 and the RFC contains details about the motivation.

Migration

As a part of the 2021 edition a migration lint, `rust_2021_incompatible_closure_captures`, has been added in order to aid in automatic migration of Rust 2018 codebases to Rust 2021.

In order to have `rustfix` migrate your code to be Rust 2021 Edition compatible, run:

```
cargo fix --edition
```

Below is an examination of how to manually migrate code to use closure captures that are compatible with Rust 2021 should the automatic migration fail or you would like to better understand how the migration works.

Changing the variables captured by a closure can cause programs to change behavior or to stop compiling in two cases:

- changes to drop order, or when destructors run (details);
- changes to which traits a closure implements (details).

Whenever any of the scenarios below are detected, `cargo fix` will insert a "dummy let" into your closure to force it to capture the entire variable:

```
let x = (vec![22], vec![23]);
let c = move || {
    // "Dummy let" that forces `x` to be captured in its entirety
    let _ = &x;

    // Otherwise, only `x.0` would be captured here
    println!("{:?}", x.0);
};
```

This is a conservative analysis: in many cases, these dummy lets can be safely removed and your program will work fine.

Wild Card Patterns

Closures now only capture data that needs to be read, which means the following closures will not capture `x`:

```
let x = 10;
let c = || {
    let _ = x; // no-op
};

let c = || match x {
    _ => println!("Hello World!")
};
```

The `let _ = x` statement here is a no-op, since the `_` pattern completely ignores the right-hand side, and `x` is a reference to a place in memory (in this case, a variable).

This change by itself (capturing fewer values) doesn't trigger any suggestions, but it may do so in conjunction with the "drop order" change below.

Subtle: There are other similar expressions, such as the "dummy lets" `let _ = &x` that we insert, which are not no-ops. This is because the right-hand side (`&x`) is not a reference to a place in memory, but rather an expression that must first be evaluated (and whose result is then discarded).

Drop Order

When a closure takes ownership of a value from a variable `t` , that value is then dropped when the closure is dropped, and not when the variable `t` goes out of scope:

```
{
    let t = (vec![0], vec![0]);

    {
        let c = || move_value(t); // t is moved here
    } // c is dropped, which drops the tuple `t` as well
} // t goes out of scope here
```

The above code will run the same in both Rust 2018 and Rust 2021. However, in cases where the closure only takes ownership of *part* of a variable, there can be differences:

```
{
    let t = (vec![0], vec![0]);

    {
        let c = || {
            // In Rust 2018, captures all of `t`.
            // In Rust 2021, captures only `t.0`
            move_value(t.0);
        };

        // In Rust 2018, `c` (and `t`) are both dropped when we
        // exit this block.
        //
        // In Rust 2021, `c` and `t.0` are both dropped when we
        // exit this block.
    }

// In Rust 2018, the value from `t` has been moved and is
// not dropped.
//
// In Rust 2021, the value from `t.0` has been moved, but `t.1`
// remains, so it will be dropped here.
}
```

In most cases, dropping values at different times just affects when memory is freed and is not important. However, some `Drop` impls (aka, destructors) have side-effects, and changing the drop order in those cases can alter the semantics of your program. In such cases, the compiler will suggest inserting a dummy `let` to force the entire variable to be captured.

Trait implementations

Closures automatically implement the following traits based on what values they capture:

- `Clone`: if all captured values are `Clone`.
- Auto traits like `Send`, `Sync`, and `UnwindSafe`: if all captured values implement the given trait.

In Rust 2021, since different values are being captured, this can affect what traits a closure will implement. The migration lints test each closure to see whether it would have implemented a given trait before and whether it still implements it now; if they find that a trait used to be implemented but no longer is, then "dummy lets" are inserted.

For instance, a common way to allow passing around raw pointers between threads is to wrap them in a struct and then implement `Send` / `Sync` auto trait for the wrapper. The closure that is passed to `thread::spawn` uses the specific fields within the wrapper but the entire wrapper is

captured regardless. Since the wrapper is `Send` / `Sync`, the code is considered safe and therefore compiles successfully.

With disjoint captures, only the specific field mentioned in the closure gets captured, which wasn't originally `Send` / `Sync` defeating the purpose of the wrapper.

```
use std::thread;

struct Ptr(*mut i32);
unsafe impl Send for Ptr {}


let mut x = 5;
let px = Ptr(&mut x as *mut i32);

let c = thread::spawn(move || {
    unsafe {
        *(px.0) += 10;
    }
}); // Closure captured px.0 which is not Send
```

Panic macro consistency

Summary

- `panic!(..)` now always uses `format_args!(..)`, just like `println!()`.
- `panic!("{")` is no longer accepted, without escaping the `{` as `{{`.
- `panic!(x)` is no longer accepted if `x` is not a string literal.
  - Use `std::panic::panic_any(x)` to panic with a non-string payload.
  - Or use `panic!("{}", x)` to use `x`'s `Display` implementation.
- The same applies to `assert!(expr, ..)`.

Details

The `panic!()` macro is one of Rust's most well known macros. However, it has some subtle surprises that we can't just change due to backwards compatibility.

```
// Rust 2018
panic!("{}", 1); // Ok, panics with the message "1"
panic!("{}"); // Ok, panics with the message "{}"
```

The `panic!()` macro only uses string formatting when it's invoked with more than one argument. When invoked with a single argument, it doesn't even look at that argument.

```
// Rust 2018
let a = "{";
println!(a); // Error: First argument must be a format string literal
panic!(a); // Ok: The panic macro doesn't care
```

It even accepts non-strings such as `panic!(123)`, which is uncommon and rarely useful since it produces a surprisingly unhelpful message: `panicked at 'Box<Any>'`.

This will especially be a problem once implicit format arguments are stabilized. That feature will make `println!("hello {name}")` a short-hand for `println!("hello {}", name)`. However, `panic!("hello {name}")` would not work as expected, since `panic!()` doesn't process a single argument as format string.

To avoid that confusing situation, Rust 2021 features a more consistent `panic!()` macro. The new `panic!()` macro will no longer accept arbitrary expressions as the only argument. It will,

just like `println!()`, always process the first argument as format string. Since `panic!()` will no longer accept arbitrary payloads, `panic_any()` will be the only way to panic with something other than a formatted string.

```
// Rust 2021
panic!("{}", 1); // Ok, panics with the message "1"
panic!("{}"); // Error, missing argument
panic!(a); // Error, must be a string literal
```

In addition, `core::panic!()` and `std::panic!()` will be identical in Rust 2021. Currently, there are some historical differences between those two, which can be noticeable when switching `#![no_std]` on or off.

Migration

A lint, `non_fmt_panics`, gets triggered whenever there is some call to `panic` that uses some deprecated behavior that will error in Rust 2021. The `non_fmt_panics` lint has already been a warning by default on all editions since the 1.50 release (with several enhancements made in later releases). If your code is already warning free, then it should already be ready to go for Rust 2021!

You can automatically migrate your code to be Rust 2021 Edition compatible or ensure it is already compatible by running:

```
cargo fix --edition
```

Should you choose or need to manually migrate, you'll need to update all panic invocations to either use the same formatting as `println` or use `std::panic::panic_any` to panic with non-string data.

For example, in the case of `panic!(MyStruct)`, you'll need to convert to using `std::panic::panic_any` (note that this is a function not a macro): `std::panic::panic_any(MyStruct)`.

In the case of panic messages that include curly braces but the wrong number of arguments (e.g., `panic!("Some curlies: {}")`), you can panic with the string literal by either using the same syntax as `println!` (i.e., `panic!("{}", "Some curlies: {}")`) or by escaping the curly braces (i.e., `panic!("Some curlies: {{}}")`).

Reserving syntax

Summary

- `any_identifier#`, `any_identifier"..."`, and `any_identifier'...'` are now reserved syntax, and no longer tokenize.
- This is mostly relevant to macros. E.g. `quote!{ #a#b }` is no longer accepted.
- It doesn't treat keywords specially, so e.g. `match"..." {}` is no longer accepted.
- Insert whitespace between the identifier and the subsequent `#`, `"`, or `'` to avoid errors.
- Edition migrations will help you insert whitespace in such cases.

Details

To make space for new syntax in the future, we've decided to reserve syntax for prefixed identifiers and literals: `prefix#identifier`, `prefix"string"`, `prefix'c'`, and `prefix#123`, where `prefix` can be any identifier. (Except those prefixes that already have a meaning, such as `b'...'` (byte chars) and `r"..."` (raw strings).)

This provides syntax we can expand into in the future without requiring an edition boundary. We may use this for temporary syntax until the next edition, or for permanent syntax if appropriate.

Without an edition, this would be a breaking change, since macros can currently accept syntax such as `hello"world"`, which they will see as two separate tokens: `hello` and `"world"`. The (automatic) fix is simple though: just insert a space: `hello "world"`. Likewise, `prefix#ident` should become `prefix #ident`. Edition migrations will help with this fix.

Other than turning these into a tokenization error, the RFC does not attach a meaning to any prefix yet. Assigning meaning to specific prefixes is left to future proposals, which will now—thanks to reserving these prefixes—not be breaking changes.

Some new prefixes you might potentially see in the future (though we haven't committed to any of them yet):

- `k#keyword` to allow writing keywords that don't exist yet in the current edition. For example, while `async` is not a keyword in edition 2015, this prefix would've allowed us to

accept `k#async` in edition 2015 without having to wait for edition 2018 to reserve `async` as a keyword.

- `f""` as a short-hand for a format string. For example, `f"hello {name}"` as a short-hand for the equivalent `format!()` invocation.
- `s""` for `String` literals.
- `c""` or `z""` for null-terminated C strings.

Migration

As a part of the 2021 edition a migration lint, `rust_2021_prefixes_incompatible_syntax`, has been added in order to aid in automatic migration of Rust 2018 codebases to Rust 2021.

In order to have `rustfix` migrate your code to be Rust 2021 Edition compatible, run:

```
cargo fix --edition
```

Should you want or need to manually migrate your code, migration is fairly straight-forward.

Let's say you have a macro that is defined like so:

```
macro_rules! my_macro {
    ($a:tt $b:tt) => {};
}
```

In Rust 2015 and 2018 it's legal for this macro to be called like so with no space between the first token tree and the second:

```
my_macro!(z"hey");
```

This `z` prefix is no longer allowed in Rust 2021, so in order to call this macro, you must add a space after the prefix like so:

```
my_macro!(z "hey");
```

Warnings promoted to errors

Summary

- Code that triggered the `bare_trait_objects` and `ellipsis_inclusive_range_patterns` lints will error in Rust 2021.

Details

Two existing lints are becoming hard errors in Rust 2021, but these lints will remain warnings in older editions.

`bare_trait_objects`:

The use of the `dyn` keyword to identify trait objects will be mandatory in Rust 2021.

For example, the following code which does not include the `dyn` keyword in `&MyTrait` will produce an error instead of just a lint in Rust 2021:

```
pub trait MyTrait {}

pub fn my_function(_trait_object: &MyTrait) { // should be `&dyn MyTrait`
  unimplemented!()
}
```

`ellipsis_inclusive_range_patterns`:

The deprecated `...` syntax for inclusive range patterns (i.e., ranges where the end value is *included* in the range) is no longer accepted in Rust 2021. It has been superseded by `..=`, which is consistent with expressions.

For example, the following code which uses `...` in a pattern will produce an error instead of just a lint in Rust 2021:

```
pub fn less_or_eq_to_100(n: u8) -> bool {
  matches!(n, 0...100) // should be `0..=100`
}
```

Migrations

If your Rust 2015 or 2018 code does not produce any warnings for `bare_trait_objects` or `ellipsis_inclusive_range_patterns` and you've not allowed these lints through the use of `#![allow()]` or some other mechanism, then there's no need to migrate.

To automatically migrate any crate that uses `...` in patterns or does not use `dyn` with trait objects, you can run `cargo fix --edition`.

Or patterns in macro-rules

Summary

- How patterns work in `macro_rules` macros changes slightly:
  - `$_:pat` in `macro_rules` now matches usage of `|` too: e.g. `A | B`.
  - The new `$_:pat_param` behaves like `$_:pat` did before; it does not match (top level) `|`.
  - `$_:pat_param` is available in all editions.

Details

Starting in Rust 1.53.0, patterns are extended to support `|` nested anywhere in the pattern. This enables you to write `Some(1 | 2)` instead of `Some(1) | Some(2)`. Since this was simply not allowed before, this is not a breaking change.

However, this change also affects `macro_rules` macros. Such macros can accept patterns using the `:pat` fragment specifier. Currently, `:pat` does *not* match top level `|`, since before Rust 1.53, not all patterns (at all nested levels) could contain a `|`. Macros that accept patterns like `A | B`, such as `matches!()` use something like `$($_:pat)|+`.

Because this would potentially break existing macros, the meaning of `:pat` did not change in Rust 1.53.0 to include `|`. Instead, that change happens in Rust 2021. In the new edition, the `:pat` fragment specifier *will* match `A | B`.

`$_:pat` fragments in Rust 2021 cannot be followed by an explicit `|`. Since there are times that one still wishes to match pattern fragments followed by a `|`, the fragment specified `:pat_param` has been added to retain the older behavior.

It's important to remember that editions are *per crate*, so the only relevant edition is the edition of the crate where the macro is defined. The edition of the crate where the macro is used does not change how the macro works.

Migration

A lint, `rust_2021_incompatible_or_patterns`, gets triggered whenever there is a use `$_:pat` which will change meaning in Rust 2021.

You can automatically migrate your code to be Rust 2021 Edition compatible or ensure it is already compatible by running:

```
cargo fix --edition
```

If you have a macro which relies on `$_:pat` not matching the top level use of `|` in patterns, you'll need to change each occurrence of `$_:pat` to `$_:pat_param`.

For example:

```
macro_rules! my_macro {
    ($x:pat | $y:pat) => {
        // TODO: implementation
    }
}

// This macro works in Rust 2018 since `$x:pat` does not match against `|`:
my_macro!(1 | 2);

// In Rust 2021 however, the `$_:pat` fragment matches `|` and is not allowed
// to be followed by a `|`. To make sure this macro still works in Rust 2021
// change the macro to the following:
macro_rules! my_macro {
    ($x:pat_param | $y:pat) => { // <- this line is different
        // TODO: implementation
    }
}
```