Introduction

Welcome to the Rust Edition Guide! "Editions" are Rust's way of communicating large changes in the way that it feels to write Rust code.

In this guide, we'll discuss:

- What editions are
- What each edition is about
- How to migrate your code from one edition to another

Note that the standard library grows with each Rust release; there are *many* additions to the standard library that are not called out in this guide. Only the major ones are, but there's tons of medium and small things that are great too. You may want to check out the standard library documentation as well.

What are Editions?

Rust ships releases on a six-week cycle. This means that users get a constant stream of new features. This is much faster than updates for other languages, but this also means that each update is smaller. After a while, all of those tiny changes add up. But, from release to release, it can be hard to look back and say *"Wow, between Rust 1.10 and Rust 1.20, Rust has changed a lot!"*

Every two or three years, we'll be producing a new *edition* of Rust. Each edition brings together the features that have landed into a clear package, with fully updated documentation and tooling. New editions ship through the usual release process.

This serves different purposes for different people:

- For active Rust users, it brings together incremental changes into an easy-to-understand package.

- For non-users, it signals that some major advancements have landed, which might make Rust worth another look.

- For those developing Rust itself, it provides a rallying point for the project as a whole.

Compatibility

When a new edition becomes available in the compiler, crates must explicitly opt in to it to take full advantage. This opt in enables editions to contain incompatible changes, like adding a new keyword that might conflict with identifiers in code, or turning warnings into errors. A Rust compiler will support all editions that existed prior to the compiler's release, and can link crates of any supported editions together. Edition changes only affect the way the compiler initially parses the code. Therefore, if you're using Rust 2015, and one of your dependencies uses Rust 2018, it all works just fine. The opposite situation works as well.

Just to be clear: most features will be available on all editions. People using any edition of Rust will continue to see improvements as new stable releases are made. In some cases however, mainly when new keywords are added, but sometimes for other reasons, there may be new features that are only available in later editions. You only need to upgrade if you want to take advantage of such features.

Trying out the 2018 edition

At the time of writing, there are two editions: 2015 and 2018. 2015 is today's Rust; Rust 2018 is currently in beta, and will land in stable in Rust 1.31, on December 6, 2018.

To give the 2018 edition a try, install the beta toolchain:

```
> rustup install beta
```

If you want the really bleeding edge, you can try nightly:

```
> rustup install nightly
```

When you see commands like `cargo fix` elsewhere in this guide, you may need to preface them with the toolchain:

```
> cargo +beta fix --edition
> cargo +nightly fix --edition
```

Creating a new project

When you create a new project with Cargo, it will automatically add configuration for the latest edition:

```
> cargo +nightly new foo
     Created binary (application) `foo` project
> cat .\foo\Cargo.toml
[package]
name = "foo"
version = "0.1.0"
authors = ["your name <you@example.com>"]
edition = "2018"

[dependencies]
```

That `edition = "2018"` setting will configure your package to use Rust 2018. No more configuration needed!

If you'd prefer to use an older edition, you can change the value in that key, for example:

```toml
[package]
name = "foo"
version = "0.1.0"
authors = ["your name <you@example.com>"]
edition = "2015"

[dependencies]
```

This will build your package in Rust 2015.

Transitioning an existing project to a new edition

New editions might change the way you write Rust – they add new syntax, language, and library features, and also remove features. For example, `try`, `async`, and `await` are keywords in Rust 2018, but not Rust 2015. If you have a project that's using Rust 2015, and you'd like to use Rust 2018 for it instead, there's a few steps that you need to take.

> 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 Rust 2018, we consider that a bug. If you run into problems with this process, please file a bug. Thank you!

Here's an example. Imagine we have a crate that has this code in `src/lib.rs`:

```rust
trait Foo {
    fn foo(&self, Box<Foo>);
}
```

This code uses an anonymous parameter, that `Box<Foo>`. 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 Rust 2018, we've included a new subcommand with Cargo. 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, _: Box<Foo>);
}
```

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

`cargo fix` is still pretty new, and so it 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 corresponding section of this guide for help, and if you have problems, please seek help at the user's forums.

Keep running `cargo fix --edition` until you have no more warnings.

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

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 commit, change your `Cargo.toml` to add the new `edition` key/value pair. For example:

```
[package]
name = "foo"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
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 is compiling with Rust 2018!

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.

Our sample code contains an outdated idiom. Here it is again:

```rust
trait Foo {
    fn foo(&self, _: Box<Foo>);
}
```

In Rust 2018, it's considered idiomatic to use the `dyn` keyword for trait objects.

Eventually, we want `cargo fix` to fix all these idioms automatically in the same manner we did for upgrading to the 2018 edition. **Currently, though, the *"idiom lints"* are not ready for widespread automatic fixing.** The compiler isn't making `cargo fix`-compatible suggestions in many cases right now, and it is making incorrect suggestions in others. Enabling the idiom lints, even with `cargo fix`, is likely to leave your crate either broken or with many warnings still remaining.

We have plans to make these idiom migrations a seamless part of the Rust 2018 experience, but we're not there yet. As a result the following instructions are recommended only for the intrepid who are willing to work through a few compiler/Cargo bugs!

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

```console
$ cargo fix --edition-idioms
```

Afterwards, `src/lib.rs` looks like this:

```rust
trait Foo {
    fn foo(&self, _: Box<dyn Foo>);
}
```

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

Note that `cargo fix` may still not be able to automatically update our code. If `cargo fix` can't fix something, it will print a warning to the console, and you'll have to fix it manually.

As mentioned before, there are known bugs around the idiom lints which means they're not all ready for prime time yet. You may get a scary-looking warning to report a bug to Cargo, which happens whenever a fix proposed by `rustc` actually caused code to stop compiling by accident. If you'd like `cargo fix` to make as much progress as possible, even if it causes code to stop compiling, you can execute:

```console
$ cargo fix --edition-idioms --broken-code
```

This will instruct `cargo fix` to apply automatic suggestions regardless of whether they work or not. Like usual, you'll see the compilation result after all fixes are applied. If you notice anything wrong or unusual, please feel free to report an issue to Cargo and we'll help prioritize and fix it.

Enjoy the new edition!

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

The edition system was created for the release of Rust 2018. The theme of Rust 2018 is *productivity*. Rust 2018 improves upon Rust 2015 through new features, simpler syntax in some cases, a smarter borrow-checker, and a host of other things. These are all in service of the productivity goal. Rust 2015 was a foundation; Rust 2018 smooths off rough edges, makes writing code simpler and easier, and removes some inconsistencies.

Module system

In this chapter of the guide, we discuss a few changes to the module system. The most notable of these are the path clarity changes.

Raw identifiers

Minimum Rust Version beta

Rust, like many programming languages, has the concept of "keywords". These identifiers mean something to the language, and so you cannot use them in places like variable names, function names, and other places. Raw identifiers let you use keywords where they would not normally be allowed.

For example, `match` is a keyword. If you try to compile this function:

```rust
fn match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}
```

You'll get this error:

```
error: expected identifier, found keyword `match`
 --> src/main.rs:4:4
  |
4 | fn match(needle: &str, haystack: &str) -> bool {
  |    ^^^^^ expected identifier, found keyword
```

You can write this with a raw identifier:

```rust
fn r#match(needle: &str, haystack: &str) -> bool {
    haystack.contains(needle)
}

fn main() {
    assert!(r#match("foo", "foobar"));
}
```

Note the `r#` prefix on both the function name as well as the call.

Motivation

This feature is useful for a few reasons, but the primary motivation was inter-edition situations. For example, `try` is not a keyword in the 2015 edition, but is in the 2018 edition. So if you have a library that is written in Rust 2015 and has a `try` function, to call it in Rust 2018, you'll need to use the raw identifier.

New keywords

The new confirmed keywords in edition 2018 are:

`async` and `await`

Here, `async` is reserved for use in `async fn` as well as in `async ||` closures and `async { .. }` blocks. Meanwhile, `await` is reserved to keep our options open with respect to `await!(expr)` syntax. See RFC 2394 for more details.

`try`

The `do catch { .. }` blocks have been renamed to `try { .. }` and to support that, the keyword `try` is reserved in edition 2018. See RFC 2388 for more details.

Path clarity

Minimum Rust Version beta Minimum Rust Version nightly for "uniform paths"

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.
- Absolute paths begin with a crate name, where the keyword `crate` refers to the current crate.
- A `foo.rs` and `foo/` subdirectory may coexist; `mod.rs` is no longer needed when placing submodules in a subdirectory.

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!

> Additionally, in nightly, there's an additional possible tweak to paths called "Uniform paths". This is backwards compatible with the new path changes. Uniform paths have a dedicated section at the end of this guide.

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
// Rust 2015

extern crate futures;

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

After:

```rust
// 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. We'd eventually like to remove the requirement for `extern crate` for them as well, but it hasn't shipped yet.

You'll need to use `extern crate` for:

- `proc_macro`

Additionally, you would need to use it for:

- `core`
- `std`

However, `extern crate std;` is already implicit, and with `#![no_std]`, `extern crate core;` is already implicit. You'll only need these in highly specialized situations.

Finally, on nightly, you'll need it for crates like:

- `alloc`
- `test`

Macros

One other use for `extern crate` was to import macros; that's no longer needed. Check the macro section for more.

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`.

Changes to paths

In Rust 2018, paths in `use` declarations *must* begin with a crate name, `crate`, `self`, or `super`.

Code that looked like this:

```
// Rust 2015

extern crate futures;

use futures::Future;

mod foo {
    pub struct Bar;
}

use foo::Bar;
```

Now looks like this:

```rust
// Rust 2018

// 'futures' is the name of a crate
use futures::Future;

mod foo {
    pub struct Bar;
}

// 'crate' means the current crate
use crate::foo::Bar;
```

In addition, all of these path forms are available outside of `use` declarations as well, which eliminates many sources of confusion. Consider this code in Rust 2015:

```rust
// Rust 2015

extern crate futures;

mod submodule {
    // this works!
    use futures::Future;

    // so why doesn't this work?
    fn my_poll() -> futures::Poll { ... }
}

fn main() {
    // this works
    let five = std::sync::Arc::new(5);
}

mod submodule {
    fn function() {
        // ... so why doesn't this work
        let five = std::sync::Arc::new(5);
    }
}
```

> In real code, you couldn't repeat `mod submodule`, and `function` would be defined in the first `mod` block.

In the `futures` example, the `my_poll` function signature is incorrect, because `submodule` contains no items named `futures`; that is, this path is considered relative. `use futures::` works even though a lone `futures::` doesn't! With `std` it can be even more confusing, as you never wrote the `extern crate std;` line at all. So why does it work in `main` but not in a submodule? Same thing: it's a relative path because it's not in a `use` declaration. `extern crate std;` is inserted at the crate root, so it's fine in `main`, but it doesn't exist in the submodule at all.

Let's look at how this change affects things:

```
// Rust 2018

// no more `extern crate futures;`

mod submodule {
    // 'futures' is the name of a crate, so this works
    use futures::Future;

    // 'futures' is the name of a crate, so this works
    fn my_poll<T, E>() -> futures::Poll {
        unimplemented!()
    }

    fn function() {
        // 'std' is the name of a crate, so this works
        let five = std::sync::Arc::new(5);
    }
}

fn main() {
    // 'std' is the name of a crate, so this works
    let five = std::sync::Arc::new(5);
}
```

Much more straightforward.

No more `mod.rs`

In Rust 2015, if you have a submodule:

```
///  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, `mod.rs` is no longer needed.

```
///  foo.rs
///  foo/bar.rs

mod foo;

/// in foo.rs
mod bar;
```

`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`.

Uniform paths

> Uniform paths are a nightly-only feature.

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

Code that looked like this:

```rust
// 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
// Rust 2018 (uniform paths variant)

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) => { ... }
    }
}
```

With Rust 2018, however, the same code will also work completely unmodified in a submodule:

```rust
// Rust 2018 (uniform paths variant)

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.

More visibility modifiers

Minimum Rust Version 1.18

You can use the `pub` keyword to make something a part of a module's public interface. But in addition, there are some new forms:

```rust
pub(crate) struct Foo;

pub(in a::b::c) struct Bar;
```

The first form makes the `Foo` struct public to your entire crate, but not externally. The second form is similar, but makes `Bar` public for one other module, `a::b::c` in this case.

Nested imports with `use`

Minimum Rust Version 1.25

A new way to write `use` statements has been added to Rust: nested import groups. If you've ever written a set of imports like this:

```rust
use std::fs::File;
use std::io::Read;
use std::path::{Path, PathBuf};
```

You can now write this:

```rust
// on one line
use std::{fs::File, io::Read, path::{Path, PathBuf}};

// with some more breathing room
use std::{
    fs::File,
    io::Read,
    path::{
        Path,
        PathBuf
    }
};
```

This can reduce some repetition, and make things a bit more clear.

Error handling and Panics

In this chapter of the guide, we discuss a few improvements to error handling in Rust. The most notable of these is the introduction of the `?` operator.

The `?` operator for easier error handling

Minimum Rust Version 1.13 for `Result<T, E>`

Minimum Rust Version 1.22 for `Option<T>`

Rust has gained a new operator, `?`, that makes error handling more pleasant by reducing the visual noise involved. It does this by solving one simple problem. To illustrate, imagine we had some code to read some data from a file:

```rust
fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("username.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

> Note: this code could be made simpler with a single call to `std::fs::read_to_string`, but we're writing it all out manually here to have an example with multiple errors.

This code has two paths that can fail, opening the file and reading the data from it. If either of these fail to work, we'd like to return an error from `read_username_from_file`. Doing so involves `match`ing on the result of the I/O operations. In simple cases like this though, where we are only propagating errors up the call stack, the matching is just boilerplate - seeing it written out, in the same pattern every time, doesn't provide the reader with a great deal of useful information.

With `?`, the above code looks like this:

```rust
fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("username.txt")?;
    let mut s = String::new();

    f.read_to_string(&mut s)?;

    Ok(s)
}
```

The `?` is shorthand for the entire match statements we wrote earlier. In other words, `?` applies to a `Result` value, and if it was an `Ok`, it unwraps it and gives the inner value. If it was an `Err`, it returns from the function you're currently in. Visually, it is much more straightforward. Instead of an entire match statement, now we are just using the single "?" character to indicate that here we are handling errors in the standard way, by passing them up the call stack.

Seasoned Rustaceans may recognize that this is the same as the `try!` macro that's been available since Rust `1.0`. And indeed, they are the same. Previously, `read_username_from_file` could have been implemented like this:

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = try!(File::open("username.txt"));
    let mut s = String::new();

    try!(f.read_to_string(&mut s));

    Ok(s)
}
```

So why extend the language when we already have a macro? There are multiple reasons. First, `try!` has proved to be extremely useful, and is used often in idiomatic Rust. It is used so often that we think it's worth having a sweet syntax. This sort of evolution is one of the great advantages of a powerful macro system: speculative extensions to the language syntax can be prototyped and iterated on without modifying the language itself, and in return, macros that turn out to be especially useful can indicate missing language features. This evolution, from `try!` to `?` is a great example.

One of the reasons `try!` needs a sweeter syntax is that it is quite unattractive when multiple invocations of `try!` are used in succession. Consider:

```
try!(try!(try!(foo()).bar()).baz())
```

as opposed to

```
foo()?.bar()?.baz()?
```

The first is quite difficult to scan visually, and each layer of error handling prefixes the expression with an additional call to `try!`. This brings undue attention to the trivial error propagation, obscuring the main code path, in this example the calls to `foo`, `bar` and `baz`. This sort of method chaining with error handling occurs in situations like the builder pattern.

Finally, the dedicated syntax will make it easier in the future to produce nicer error messages tailored specifically to `?`, whereas it is difficult to produce nice errors for macro-expanded code generally.

You can use `?` with `Result<T, E>`s, but also with `Option<T>`. In that case, `?` will return a value for `Some(T)` and return `None` for `None`. One current restriction is that you cannot use `?` for both in the same function, as the return type needs to match the type you use `?` on. In the future, this restriction will be lifted.

`?` in `main` and tests

Minimum Rust Version 1.26

Rust's error handling revolves around returning `Result<T, E>` and using `?` to propagate errors. For those who write many small programs and, hopefully, many tests, one common paper cut has been mixing entry points such as `main` and `#[test]`s with error handling.

As an example, you might have tried to write:

```
use std::fs::File;

fn main() {
    let f = File::open("bar.txt")?;
}
```

Since `?` works by propagating the `Result` with an early return to the enclosing function, the snippet above does not work, and results today in the following error:

```
error[E0277]: the `?` operator can only be used in a function that returns
`Result`
              or `Option` (or another type that implements `std::ops::Try`)
 --> src/main.rs:5:13
  |
5 |     let f = File::open("bar.txt")?;
  |             ^^^^^^^^^^^^^^^^^^^^^^ cannot use the `?` operator in a function
that returns `()`
  |
  = help: the trait `std::ops::Try` is not implemented for `()`
  = note: required by `std::ops::Try::from_error`
```

To solve this problem in Rust 2015, you might have written something like:

```
// Rust 2015


fn run() -> Result<(), Box<Error>> {
    // real logic..
    Ok(())
}

fn main() {
    if let Err(e) = run() {
        println!("Application error: {}", e);
        process::exit(1);
    }
}
```

However, in this case, the `run` function has all the interesting logic and `main` is just boilerplate. The problem is even worse for `#[test]`s, since there tend to be a lot more of them.

In Rust 2018 you can instead let your `#[test]`s and `main` functions return a `Result`:

```rust
// Rust 2018

use std::fs::File;

fn main() -> Result<(), std::io::Error> {
    let f = File::open("bar.txt")?;

    Ok(())
}
```

In this case, if say the file doesn't exist and there is an `Err(err)` somewhere, then `main` will exit with an error code (not `0`) and print out a `Debug` representation of `err`.

More details

Getting `-> Result<..>` to work in the context of `main` and `#[test]`s is not magic. It is all backed up by a `Termination` trait which all valid return types of `main` and testing functions must implement. The trait is defined as:

```rust
pub trait Termination {
    fn report(self) -> i32;
}
```

When setting up the entry point for your application, the compiler will use this trait and call `.report()` on the `Result` of the `main` function you have written.

Two simplified example implementations of this trait for `Result` and `()` are:

```rust
impl Termination for () {
    fn report(self) -> i32 {
        ExitCode::SUCCESS.report()
    }
}

impl<E: fmt::Debug> Termination for Result<(), E> {
    fn report(self) -> i32 {
        match self {
            Ok(()) => ().report(),
            Err(err) => {
                eprintln!("Error: {:?}", err);
                ExitCode::FAILURE.report()
            }
        }
    }
}
```

As you can see in the case of `()`, a success code is simply returned. In the case of `Result`, the success case delegates to the implementation for `()` but prints out an error message and a failure exit code on `Err(..)`.

To learn more about the finer details, consult either the tracking issue or the RFC.

Controlling panics with `std::panic`

Minimum Rust Version 1.9

There is a `std::panic` module, which includes methods for halting the unwinding process started by a panic:

```rust
use std::panic;

let result = panic::catch_unwind(|| {
    println!("hello!");
});
assert!(result.is_ok());

let result = panic::catch_unwind(|| {
    panic!("oh no!");
});
assert!(result.is_err());
```

In general, Rust distinguishes between two ways that an operation can fail:

- Due to an *expected problem*, like a file not being found.
- Due to an *unexpected problem*, like an index being out of bounds for an array.

Expected problems usually arise from conditions that are outside of your control; robust code should be prepared for anything its environment might throw at it. In Rust, expected problems are handled via the `Result` type, which allows a function to return information about the problem to its caller, which can then handle the error in a fine-grained way.

Unexpected problems are *bugs*: they arise due to a contract or assertion being violated. Since they are unexpected, it doesn't make sense to handle them in a fine-grained way. Instead, Rust employs a "fail fast" approach by *panicking*, which by default unwinds the stack (running destructors but no other code) of the thread which discovered the error. Other threads continue running, but will discover the panic any time they try to communicate with the panicked thread (whether through channels or shared memory). Panics thus abort execution up to some "isolation boundary", with code on the other side of the boundary still able to run, and perhaps to "recover" from the panic in some very coarse-grained way. A server, for example, does not necessarily need to go down just because of an assertion failure in one of its threads.

It's also worth noting that programs may choose to *abort* instead of unwind, and so catching panics may not work. If your code relies on `catch_unwind`, you should add this to your

Cargo.toml:

```
[profile.debug]
panic = "unwind"

[profile.release]
panic = "unwind"
```

If any of your users choose to abort, they'll get a compile-time failure.

The `catch_unwind` API offers a way to introduce new isolation boundaries *within a thread*. There are a couple of key motivating examples:

- Embedding Rust in other languages
- Abstractions that manage threads
- Test frameworks, because tests may panic and you don't want that to kill the test runner

For the first case, unwinding across a language boundary is undefined behavior, and often leads to segfaults in practice. Allowing panics to be caught means that you can safely expose Rust code via a C API, and translate unwinding into an error on the C side.

For the second case, consider a threadpool library. If a thread in the pool panics, you generally don't want to kill the thread itself, but rather catch the panic and communicate it to the client of the pool. The `catch_unwind` API is paired with `resume_unwind`, which can then be used to restart the panicking process on the client of the pool, where it belongs.

In both cases, you're introducing a new isolation boundary within a thread, and then translating the panic into some other form of error elsewhere.

Aborting on panic

Minimum Rust Version 1.10

By default, Rust programs will unwind the stack when a `panic!` happens. If you'd prefer an immediate abort instead, you can configure this in `Cargo.toml`:

```
[profile.debug]
panic = "abort"

[profile.release]
panic = "abort"
```

Why might you choose to do this? By removing support for unwinding, you'll get smaller binaries. You will lose the ability to catch panics. Which choice is right for you depends on exactly what you're doing.

Control flow

In this chapter of the guide, we discuss a few improvements to control flow. The most notable of these *will* be `async` and `await`.

`loop`s can break with a value

Minimum Rust Version 1.19

`loop`s can now break with a value:

```
// old code
let x;

loop {
    x = 7;
    break;
}

// new code
let x = loop { break 7; };
```

Rust has traditionally positioned itself as an “expression oriented language”, that is, most things are expressions that evaluate to a value, rather than statements. `loop` stuck out as strange in this way, as it was previously a statement.

For now, this only applies to `loop`, and not things like `while` or `for`. It's not clear yet, but we may add this to those in the future.

async/await for easier concurrency

Minimum Rust Version nightly

The initial release of Rust 2018 won't ship with `async`/`await` support, but we have reserved the keywords so that a future release will contain them. We'll update this page when it's closer to shipping!

Trait system

In this chapter of the guide, we discuss a few improvements to the trait system. The most notable of these is `impl Trait`.

`impl Trait` for returning complex types with ease

Minimum Rust Version 1.26

`impl Trait` is the new way to specify unnamed but concrete types that implement a specific trait. There are two places you can put it: argument position, and return position.

```rust
trait Trait {}

// argument position
fn foo(arg: impl Trait) {
}

// return position
fn foo() -> impl Trait {
}
```

Argument Position

In argument position, this feature is quite simple. These two forms are almost the same:

```rust
trait Trait {}

fn foo<T: Trait>(arg: T) {
}

fn foo(arg: impl Trait) {
}
```

That is, it's a slightly shorter syntax for a generic type parameter. It means, "`arg` is an argument that takes any type that implements the `Trait` trait."

However, there's also an important technical difference between `T: Trait` and `impl Trait` here. When you write the former, you can specify the type of `T` at the call site with turbo-fish syntax as with `foo::<usize>(1)`. In the case of `impl Trait`, if it is used anywhere in the function definition, then you can't use turbo-fish at all. Therefore, you should be mindful that changing both from and to `impl Trait` can constitute a breaking change for the users of your code.

Return Position

In return position, this feature is more interesting. It means "I am returning some type that implements the `Trait` trait, but I'm not going to tell you exactly what the type is." Before `impl Trait`, you could do this with trait objects:

```rust
trait Trait {}

impl Trait for i32 {}

fn returns_a_trait_object() -> Box<dyn Trait> {
    Box::new(5)
}
```

However, this has some overhead: the `Box<T>` means that there's a heap allocation here, and this will use dynamic dispatch. See the `dyn Trait` section for an explanation of this syntax. But we only ever return one possible thing here, the `Box<i32>`. This means that we're paying for dynamic dispatch, even though we don't use it!

With `impl Trait`, the code above could be written like this:

```rust
trait Trait {}

impl Trait for i32 {}

fn returns_a_trait_object() -> impl Trait {
    5
}
```

Here, we have no `Box<T>`, no trait object, and no dynamic dispatch. But we still can obscure the `i32` return type.

With `i32`, this isn't super useful. But there's one major place in Rust where this is much more useful: closures.

`impl Trait` and closures

> If you need to catch up on closures, check out their chapter in the book.

In Rust, closures have a unique, un-writable type. They do implement the `Fn` family of traits, however. This means that previously, the only way to return a closure from a function was to use a trait object:

```rust
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

You couldn't write the type of the closure, only use the `Fn` trait. That means that the trait object is necessary. However, with `impl Trait`:

```
fn returns_closure() -> impl Fn(i32) -> i32 {
    |x| x + 1
}
```

We can now return closures by value, just like any other type!

More details

The above is all you need to know to get going with `impl Trait`, but for some more nitty-gritty details: type parameters and `impl Trait` work slightly differently when they're in argument position versus return position. Consider this function:

```
fn foo<T: Trait>(x: T) {
```

When you call it, you set the type, `T`. "you" being the caller here. This signature says "I accept any type that implements Trait." ("any type" == universal in the jargon)

This version:

```
fn foo<T: Trait>() -> T {
```

is similar, but also different. You, the caller, provide the type you want, `T`, and then the function returns it. You can see this in Rust today with things like parse or collect:

```
let x: i32 = "5".parse()?;
let x: u64 = "5".parse()?;
```

Here, `.parse` has this signature:

```
pub fn parse<F>(&self) -> Result<F, <F as FromStr>::Err> where
    F: FromStr,
```

Same general idea, though with a result type and `FromStr` has an associated type... anyway, you can see how `F` is in the return position here. So you have the ability to choose.

With `impl Trait`, you're saying "hey, some type exists that implements this trait, but I'm not gonna tell you what it is.". So now, the caller can't choose, and the function itself gets to choose. If we tried to define parse with `Result<impl F,...` as the return type, it wouldn't work.

Using `impl Trait` in more places

As previously mentioned, as a start, you will only be able to use `impl Trait` as the argument or return type of a free or inherent function. However, `impl Trait` can't be used inside implementations of traits, nor can it be used as the type of a let binding or inside a type alias. Some of these restrictions will eventually be lifted. For more information, see the tracking issue on `impl Trait`.

`dyn Trait` for trait objects

Minimum Rust Version 1.27

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:

```rust
trait Trait {}

impl Trait for i32 {}

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

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

That's it!

More details

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.

More container types support trait objects

Minimum Rust Version 1.2

In Rust 1.0, only certain, special types could be used to create trait objects.

With Rust 1.2, that restriction was lifted, and more types became able to do this. For example, `Rc<T>`, one of Rust's reference-counted types:

```rust
use std::rc::Rc;

trait Foo {}

impl Foo for i32 {

}

fn main() {
    let obj: Rc<dyn Foo> = Rc::new(5);
}
```

This code would not work with Rust 1.0, but now works.

> If you haven't seen the `dyn` syntax before, see the section on it. For versions that do not support it, replace `Rc<dyn Foo>` with `Rc<Foo>`.

Associated constants

Minimum Rust Version 1.20

You can define traits, structs, and enums that have “associated functions”:

```rust
struct Struct;

impl Struct {
    fn foo() {
        println!("foo is an associated function of Struct");
    }
}

fn main() {
    Struct::foo();
}
```

These are called “associated functions” because they are functions that are associated with the type, that is, they're attached to the type itself, and not any particular instance.

Rust 1.20 adds the ability to define “associated constants” as well:

```rust
struct Struct;

impl Struct {
    const ID: u32 = 0;
}

fn main() {
    println!("the ID of Struct is: {}", Struct::ID);
}
```

That is, the constant `ID` is associated with `Struct`. Like functions, associated constants work with traits and enums as well.

Traits have an extra ability with associated constants that gives them some extra power. With a trait, you can use an associated constant in the same way you'd use an associated type: by declaring it, but not giving it a value. The implementor of the trait then declares its value upon implementation:

```rust
trait Trait {
    const ID: u32;
}

struct Struct;

impl Trait for Struct {
    const ID: u32 = 5;
}

fn main() {
    println!("{}", Struct::ID);
}
```

Before this feature, if you wanted to make a trait that represented floating point numbers, you'd have to write this:

```rust
trait Float {
    fn nan() -> Self;
    fn infinity() -> Self;
    // ...
}
```

This is slightly unwieldy, but more importantly, because they're functions, they cannot be used in constant expressions, even though they only return a constant. Because of this, a design for `Float` would also have to include constants as well:

```rust
mod f32 {
    const NAN: f32 = 0.0f32 / 0.0f32;
    const INFINITY: f32 = 1.0f32 / 0.0f32;

    impl Float for f32 {
        fn nan() -> Self {
            f32::NAN
        }
        fn infinity() -> Self {
            f32::INFINITY
        }
    }
}
```

Associated constants let you do this in a much cleaner way. This trait definition:

```rust
trait Float {
    const NAN: Self;
    const INFINITY: Self;
    // ...
}
```

Leads to this implementation:

```rust
mod f32 {
    impl Float for f32 {
        const NAN: f32 = 0.0f32 / 0.0f32;
        const INFINITY: f32 = 1.0f32 / 0.0f32;
    }
}
```

much cleaner, and more versatile.

No more anonymous trait parameters

Minimum Rust Version beta

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:

```rust
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);
}
```

Slice patterns

Minimum Rust Version 1.26

Have you ever tried to pattern match on the contents and structure of a slice? Rust 2018 will let you do just that.

For example, say we want to accept a list of names and respond to that with a greeting. With slice patterns, we can do that easy as pie with:

```
fn main() {
    greet(&[]);
    // output: Bummer, there's no one here :(
    greet(&["Alan"]);
    // output: Hey, there Alan! You seem to be alone.
    greet(&["Joan", "Hugh"]);
    // output: Hello, Joan and Hugh. Nice to see you are at least 2!
    greet(&["John", "Peter", "Stewart"]);
    // output: Hey everyone, we seem to be 3 here today.
}

fn greet(people: &[&str]) {
    match people {
        [] => println!("Bummer, there's no one here :("),
        [only_one] => println!("Hey, there {}! You seem to be alone.",
only_one),
        [first, second] => println!(
            "Hello, {} and {}. Nice to see you are at least 2!",
            first, second
        ),
        _ => println!("Hey everyone, we seem to be {} here today.",
people.len()),
    }
}
```

Now, you don't have to check the length first.

We can also match on arrays like so:

```
let arr = [1, 2, 3];

assert_eq!("ends with 3", match arr {
    [_, _, 3] => "ends with 3",
    [a, b, c] => "ends with something else",
});
```

More details

Exhaustive patterns

In the first example, note in particular the `_ => ...` pattern. Since we are matching on a slice, it could be of any length, so we need a *"catch all pattern"* to handle it. If we forgot the `_ => ...` or `identifier => ...` pattern, we would instead get an error saying:

```
error[E0004]: non-exhaustive patterns: `&[_, _, _]` not covered
```

If we added a case for a slice of size `3` we would instead get:

```
error[E0004]: non-exhaustive patterns: `&[_, _, _, _]` not covered
```

and so on...

Arrays and exact lengths

In the second example above, since arrays in Rust are of known lengths, we have to match on exactly three elements. If we try to match on 2 or 4 elements,we get the errors:

```
error[E0527]: pattern requires 2 elements but array has 3
```

and

```
error[E0527]: pattern requires 4 elements but array has 3
```

In the pipeline

When it comes to slice patterns, more advanced forms are planned but have not been stabilized yet. To learn more, follow the tracking issue.

Ownership and lifetimes

In this chapter of the guide, we discuss a few improvements to ownership and lifetimes. One of the most notable of these is default match binding modes.

Non-lexical lifetimes

Minimum Rust Version beta

The borrow checker has been enhanced to accept more code, via a mechanism called "non-lexical lifetimes." Consider this example:

```rust
fn main() {
    let mut x = 5;

    let y = &x;

    let z = &mut x;
}
```

In older Rust, this is a compile-time error:

```
error[E0502]: cannot borrow `x` as mutable because it is also borrowed as
immutable
 --> src/main.rs:5:18
  |
4 |     let y = &x;
  |              - immutable borrow occurs here
5 |     let z = &mut x;
  |                  ^ mutable borrow occurs here
6 | }
  | - immutable borrow ends here
```

This is because lifetimes follow "lexical scope"; that is, the borrow from `y` is considered to be held until `y` goes out of scope at the end of `main`, even though we never use `y` again. This code is fine, but the borrow checker could not handle it.

Today, this code will compile just fine.

Better errors

What if we did use `y`, like this?

```rust
fn main() {
    let mut x = 5;
    let y = &x;
    let z = &mut x;

    println!("y: {}", y);
}
```

Here's the error:

```
error[E0502]: cannot borrow `x` as mutable because it is also borrowed as
immutable
 --> src/main.rs:5:18
  |
4 |     let y = &x;
  |              - immutable borrow occurs here
5 |     let z = &mut x;
  |                  ^ mutable borrow occurs here
...
8 | }
  | - immutable borrow ends here
```

With non-lexical lifetimes, the error changes slightly:

```
error[E0502]: cannot borrow `x` as mutable because it is also borrowed as
immutable
 --> src/main.rs:5:13
  |
4 |     let y = &x;
  |             -- immutable borrow occurs here
5 |     let z = &mut x;
  |             ^^^^^^ mutable borrow occurs here
6 |
7 |     println!("y: {}", y);
  |                       - borrow later used here
```

Instead of pointing to where `y` goes out of scope, it shows you where the conflicting borrow occurs. This makes these sorts of errors *far* easier to debug.

Default match bindings

Minimum Rust Version 1.26

Have you ever had a borrowed `Option<T>` and tried to match on it? You probably wrote this:

```rust
let s: &Option<String> = &Some("hello".to_string());

match s {
    Some(s) => println!("s is: {}", s),
    _ => (),
};
```

In Rust 2015, this would fail to compile, and you would have to write the following instead:

```
// Rust 2015

let s: &Option<String> = &Some("hello".to_string());

match s {
    &Some(ref s) => println!("s is: {}", s),
    _ => (),
};
```

Rust 2018, by contrast, will infer the `&`s and `ref`s, and your original code will Just Work.

This affects not just `match`, but patterns everywhere, such as in `let` statements, closure arguments, and `for` loops.

More details

The mental model of patterns has shifted a bit with this change, to bring it into line with other aspects of the language. For example, when writing a `for` loop, you can iterate over borrowed contents of a collection by borrowing the collection itself:

```
let my_vec: Vec<i32> = vec![0, 1, 2];

for x in &my_vec { ... }
```

The idea is that an `&T` can be understood as a *borrowed view of* `T`, and so when you iterate, match, or otherwise destructure a `&T` you get a borrowed view of its internals as well.

More formally, patterns have a "binding mode," which is either by value (`x`), by reference (`ref x`), or by mutable reference (`ref mut x`). In Rust 2015, `match` always started in by-value mode, and required you to explicitly write `ref` or `ref mut` in patterns to switch to a borrowing mode. In Rust 2018, the type of the value being matched informs the binding mode, so that if you match against an `&Option<String>` with a `Some` variant, you are put into `ref` mode automatically, giving you a borrowed view of the internal data. Similarly, `&mut Option<String>` would give you a `ref mut` view.

`'_`, the anonymous lifetime

Minimum Rust Version nightly

Rust 2018 allows you to explicitly mark where a lifetime is elided, for types where this elision might otherwise be unclear. To do this, you can use the special lifetime `'_` much like you can explicitly mark that a type is inferred with the syntax `let x: _ = ..;`.

Let's say, for whatever reason, that we have a simple wrapper around `&'a str`:

```rust
struct StrWrap<'a>(&'a str);
```

In Rust 2015, you might have written:

```rust
// Rust 2015

use std::fmt;


fn make_wrapper(string: &str) -> StrWrap {
    StrWrap(string)
}

impl<'a> fmt::Debug for StrWrap<'a> {
    fn fmt(&self, fmt: &mut fmt::Formatter) -> fmt::Result {
        fmt.write_str(self.0)
    }
}
```

In Rust 2018, you can instead write:

```rust
#![feature(rust_2018_preview)]


// Rust 2018

fn make_wrapper(string: &str) -> StrWrap<'_> {
    StrWrap(string)
}

impl fmt::Debug for StrWrap<'_> {
    fn fmt(&self, fmt: &mut fmt::Formatter<'_>) -> fmt::Result {
        fmt.write_str(self.0)
    }
}
```

More details

In the Rust 2015 snippet above, we've used `-> StrWrap`. However, unless you take a look at the definition of `StrWrap`, it is not clear that the returned value is actually borrowing something. Therefore, starting with Rust 2018, it is deprecated to leave off the lifetime parameters for non-reference-types (types other than `&` and `&mut`). Instead, where you previously wrote `-> StrWrap`, you should now write `-> StrWrap<'_>`, making clear that borrowing is occurring.

What exactly does `'_` mean? It depends on the context! In output contexts, as in the return type of `make_wrapper`, it refers to a single lifetime for all "output" locations. In input contexts, a fresh lifetime is generated for each "input location". More concretely, to understand input contexts, consider the following example:

```rust
// Rust 2015

struct Foo<'a, 'b: 'a> {
    field: &'a &'b str,
}

impl<'a, 'b: 'a> Foo<'a, 'b> {
    // some methods...
}
```

We can rewrite this as:

```rust
#![feature(rust_2018_preview)]


// Rust 2018

impl Foo<'_, '_> {
    // some methods...
}
```

This is the same, because for each `'_`, a fresh lifetime is generated. Finally, the relationship `'a: 'b` which the struct requires must be upheld.

For more details, see the tracking issue on In-band lifetime bindings.

Lifetime elision in impl

Minimum Rust Version 1.31

When writing `impl` blocks, you can now elide lifetime annotations in some situations.

Consider a trait like `MyIterator`:

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

In Rust 2015, if we wanted to implement this iterator for mutable references to `Iterators`, we'd need to write this:

```
impl<'a, I: MyIterator> MyIterator for &'a mut I {
    type Item = I::Item;
    fn next(&mut self) -> Option<Self::Item> {
        (*self).next()
    }
}
```

Note all of the `'a` annotations. In Rust 2018, we can write this:

```
impl<I: MyIterator> MyIterator for &mut I {
    type Item = I::Item;
    fn next(&mut self) -> Option<Self::Item> {
        (*self).next()
    }
}
```

Similarly, lifetime annotations can appear due to a struct that contains references:

```
struct SetOnDrop<'a, T> {
    borrow: &'a mut T,
    value: Option<T>,
}
```

In Rust 2015, to implement `Drop` on this struct, we'd write:

```
impl<'a, T> Drop for SetOnDrop<'a, T> {
    fn drop(&mut self) {
        if let Some(x) = self.value.take() {
            *self.borrow = x;
        }
    }
}
```

But in Rust 2018, we can combine elision with the anonymous lifetime and write this instead.

```
impl<T> Drop for SetOnDrop<'_, T> {
    fn drop(&mut self) {
        if let Some(x) = self.value.take() {
            *self.borrow = x;
        }
    }
}
```

`T: 'a` inference in structs

Minimum Rust Version beta

An annotation in the form of `T: 'a`, where `T` is either a type or another lifetime, is called an *"outlives"* requirement. Note that *"outlives"* also implies `'a: 'a`.

One way in which edition 2018 helps you out in maintaining flow when writing programs is by removing the need to explicitly annotate these `T: 'a` outlives requirements in `struct` definitions. Instead, the requirements will be inferred from the fields present in the definitions.

Consider the following `struct` definitions in Rust 2015:

```rust
// Rust 2015

struct Ref<'a, T: 'a> {
    field: &'a T
}

// or written with a `where` clause:

struct WhereRef<'a, T> where T: 'a {
    data: &'a T
}

// with nested references:

struct RefRef<'a, 'b: 'a, T: 'b> {
    field: &'a &'b T,
}

// using an associated type:

struct ItemRef<'a, T: Iterator>
where
    T::Item: 'a
{
    field: &'a T::Item
}
```

In Rust 2018, since the requirements are inferred, you can instead write:

```rust
// Rust 2018

struct Ref<'a, T> {
    field: &'a T
}

struct WhereRef<'a, T> {
    data: &'a T
}

struct RefRef<'a, 'b, T> {
    field: &'a &'b T,
}

struct ItemRef<'a, T: Iterator> {
    field: &'a T::Item
}
```

If you prefer to be more explicit in some cases, that is still possible.

More details

For more details, see the tracking issue and the RFC.

Simpler lifetimes in `static` and `const`

Minimum Rust Version 1.17

In older Rust, you had to explicitly write the `'static` lifetime in any `static` or `const` that needed a lifetime:

```rust
const NAME: &'static str = "Ferris";
static NAME: &'static str = "Ferris";
```

But `'static` is the only possible lifetime there. So Rust now assumes the `'static` lifetime, and you don't have to write it out:

```rust
const NAME: &str = "Ferris";
static NAME: &str = "Ferris";
```

In some situations, this can remove a *lot* of boilerplate:

```rust
// old
const NAMES: &'static [&'static str; 2] = &["Ferris", "Bors"];

// new
const NAMES: &[&str; 2] = &["Ferris", "Bors"];
```

Data types

In this chapter of the guide, we discuss a few improvements to data types. One of these are field-init-shorthand.

Field init shorthand

Minimum Rust Version 1.17

In older Rust, when initializing a struct, you must always give the full set of `key: value` pairs for its fields:

```rust
struct Point {
    x: i32,
    y: i32,
}

let a = 5;
let b = 6;

let p = Point {
    x: a,
    y: b,
};
```

However, often these variables would have the same names as the fields. So you'd end up with code that looks like this:

```rust
let p = Point {
    x: x,
    y: y,
};
```

Now, if the variable is of the same name, you don't have to write out both, just write out the key:

```rust
struct Point {
    x: i32,
    y: i32,
}

let x = 5;
let y = 6;

// new
let p = Point {
    x,
    y,
};
```

`..=` for inclusive ranges

Minimum Rust Version 1.26

Since well before Rust 1.0, you've been able to create exclusive ranges with .. like this:

```rust
for i in 1..3 {
    println!("i: {}", i);
}
```

This will print `i: 1` and then `i: 2`. Today, you can now create an inclusive range, like this:

```rust
for i in 1..=3 {
    println!("i: {}", i);
}
```

This will print `i: 1` and then `i: 2` like before, but also `i: 3`; the three is included in the range. Inclusive ranges are especially useful if you want to iterate over every possible value in a range. For example, this is a surprising Rust program:

```rust
fn takes_u8(x: u8) {
    // ...
}

fn main() {
    for i in 0..256 {
        println!("i: {}", i);
        takes_u8(i);
    }
}
```

What does this program do? The answer: nothing. The warning we get when compiling has a hint:

```
warning: literal out of range for u8
 --> src/main.rs:6:17
  |
6 |     for i in 0..256 {
  |                 ^^^
  |
  = note: #[warn(overflowing_literals)] on by default
```

That's right, since `i` is a `u8`, this overflows, and is the same as writing `for i in 0..0`, so the loop executes zero times.

We can do this with inclusive ranges, however:

```rust
fn takes_u8(x: u8) {
    // ...
}

fn main() {
    for i in 0..=255 {
        println!("i: {}", i);
        takes_u8(i);
    }
}
```

This will produce those 256 lines of output you might have been expecting.

128 bit integers

Minimum Rust Version 1.26

A very simple feature: Rust now has 128 bit integers!

```rust
let x: i128 = 0;
let y: u128 = 0;
```

These are twice the size of `u64`, and so can hold more values. More specifically,

- `u128`: `0` - `340,282,366,920,938,463,463,374,607,431,768,211,455`
- `i128`: `-170,141,183,460,469,231,731,687,303,715,884,105,728` - `170,141,183,460,469,231,731,687,303,715,884,105,727`

Whew!

"Operator-equals" are now implementable

Minimum Rust Version 1.8

The various “operator equals” operators, such as `+=` and `-=`, are implementable via various traits. For example, to implement `+=` on a type of your own:

```rust
use std::ops::AddAssign;

#[derive(Debug)]
struct Count {
    value: i32,
}

impl AddAssign for Count {
    fn add_assign(&mut self, other: Count) {
        self.value += other.value;
    }
}

fn main() {
    let mut c1 = Count { value: 1 };
    let c2 = Count { value: 5 };

    c1 += c2;

    println!("{:?}", c1);
}
```

This will print `Count { value: 6 }`.

`union` for an unsafe form of `enum`

Minimum Rust Version 1.19

Rust now supports `unions`:

```rust
union MyUnion {
    f1: u32,
    f2: f32,
}
```

Unions are kind of like enums, but they are “untagged”. Enums have a “tag” that stores which variant is the correct one at runtime; unions don't have this tag.

Since we can interpret the data held in the union using the wrong variant and Rust can’t check this for us, that means reading a union’s field is unsafe:

```rust
let mut u = MyUnion { f1: 1 };

u.f1 = 5;

let value = unsafe { u.f1 };
```

Pattern matching works too:

```rust
fn f(u: MyUnion) {
    unsafe {
        match u {
            MyUnion { f1: 10 } => { println!("ten"); }
            MyUnion { f2 } => { println!("{}", f2); }
        }
    }
}
```

When are unions useful? One major use-case is interoperability with C. C APIs can (and depending on the area, often do) expose unions, and so this makes writing API wrappers for those libraries significantly easier. Additionally, unions also simplify Rust implementations of space-efficient or cache-efficient structures relying on value representation, such as machine-word-sized unions using the least-significant bits of aligned pointers to distinguish cases.

There’s still more improvements to come. For now, unions can only include `Copy` types and may not implement `Drop`. We expect to lift these restrictions in the future.

Choosing alignment with the repr attribute

Minimum Rust Version 1.25

From Wikipedia:

> The CPU in modern computer hardware performs reads and writes to memory most efficiently when the data is naturally aligned, which generally means that the data address is a multiple of the data size. Data alignment refers to aligning elements according to their natural alignment. To ensure natural alignment, it may be necessary to insert some padding between structure elements or after the last element of a structure.

The `#[repr]` attribute has a new parameter, `align`, that sets the alignment of your struct:

```rust
struct Number(i32);

assert_eq!(std::mem::align_of::<Number>(), 4);
assert_eq!(std::mem::size_of::<Number>(), 4);

#[repr(align(16))]
struct Align16(i32);

assert_eq!(std::mem::align_of::<Align16>(), 16);
assert_eq!(std::mem::size_of::<Align16>(), 16);
```

If you're working with low-level stuff, control of these kinds of things can be very important!

The alignment of a type is normally not worried about as the compiler will "do the right thing" of picking an appropriate alignment for general use cases. There are situations, however, where a nonstandard alignment may be desired when operating with foreign systems. For example these sorts of situations tend to necessitate or be much easier with a custom alignment:

- Hardware can often have obscure requirements such as "this structure is aligned to 32 bytes" when it in fact is only composed of 4-byte values. While this can typically be manually calculated and managed, it's often also useful to express this as a property of a type to get the compiler to do a little extra work instead.
- C compilers like `gcc` and `clang` offer the ability to specify a custom alignment for structures, and Rust can much more easily interoperate with these types if Rust can also mirror the request for a custom alignment (e.g. passing a structure to C correctly is much easier).
- Custom alignment can often be used for various tricks here and there and is often convenient as "let's play around with an implementation" tool. For example this can be used to statically allocate page tables in a kernel or create an at-least cache-line-sized structure easily for concurrent programming.

The purpose of this feature is to provide a lightweight annotation to alter the compiler-inferred alignment of a structure to enable these situations much more easily.

SIMD for faster computing

Minimum Rust Version 1.27

The basics of SIMD are now available! SIMD stands for “single instruction, multiple data.” Consider a function like this:

```rust
pub fn foo(a: &[u8], b: &[u8], c: &mut [u8]) {
    for ((a, b), c) in a.iter().zip(b).zip(c) {
        *c = *a + *b;
    }
}
```

Here, we’re taking two slices, and adding the numbers together, placing the result in a third slice. The simplest possible way to do this would be to do exactly what the code does, and loop through each set of elements, add them together, and store it in the result. However, compilers can often do better. LLVM will usually “autovectorize” code like this, which is a fancy term for “use SIMD.” Imagine that `a` and `b` were both 16 elements long. Each element is a `u8`, and so that means that each slice would be 128 bits of data. Using SIMD, we could put both `a` and `b` into 128 bit registers, add them together in a *single* instruction, and then copy the resulting 128 bits into `c`. That’d be much faster!

While stable Rust has always been able to take advantage of autovectorization, sometimes, the compiler just isn’t smart enough to realize that we can do something like this. Additionally, not every CPU has these features, and so LLVM may not use them so your program can be used on a wide variety of hardware. The `std::arch` module allows us to use these kinds of instructions directly, which means we don’t need to rely on a smart compiler. Additionally, it includes some features that allow us to choose a particular implementation based on various criteria. For example:

```rust
#[cfg(all(any(target_arch = "x86", target_arch = "x86_64"),
      target_feature = "avx2"))]
fn foo() {
    #[cfg(target_arch = "x86")]
    use std::arch::x86::_mm256_add_epi64;
    #[cfg(target_arch = "x86_64")]
    use std::arch::x86_64::_mm256_add_epi64;

    unsafe {
        _mm256_add_epi64(...);
    }
}
```

Here, we use cfg flags to choose the correct version based on the machine we’re targeting; on x86 we use that version, and on x86_64 we use its version. We can also choose at runtime:

```
fn foo() {
    #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
    {
        if is_x86_feature_detected!("avx2") {
            return unsafe { foo_avx2() };
        }
    }

    foo_fallback();
}
```

Here, we have two versions of the function: one which uses AVX2, a specific kind of SIMD feature that lets you do 256-bit operations. The `is_x86_feature_detected!` macro will generate code that detects if your CPU supports AVX2, and if so, calls the foo_avx2 function. If not, then we fall back to a non-AVX implementation, foo_fallback. This means that our code will run super fast on CPUs that support AVX2, but still work on ones that don't, albeit slower.

If all of this seems a bit low-level and fiddly, well, it is! `std::arch` is specifically primitives for building these kinds of things. We hope to eventually stabilize a `std::simd` module with higher-level stuff in the future. But landing the basics now lets the ecosystem experiment with higher level libraries starting today. For example, check out the faster crate. Here's a code snippet with no SIMD:

```
let lots_of_3s = (&[-123.456f32; 128][..]).iter()
    .map(|v| {
        9.0 * v.abs().sqrt().sqrt().recip().ceil().sqrt() - 4.0 - 2.0
    })
    .collect::<Vec<f32>>();
```

To use SIMD with this code via faster, you'd change it to this:

```
let lots_of_3s = (&[-123.456f32; 128][..]).simd_iter()
    .simd_map(f32s(0.0), |v| {
        f32s(9.0) * v.abs().sqrt().rsqrt().ceil().sqrt() - f32s(4.0) - f32s(2.0)
    })
    .scalar_collect();
```

It looks almost the same: `simd_iter` instead of `iter`, `simd_map` instead of `map`, `f32s(2.0)` instead of `2.0`. But you get a SIMD-ified version generated for you.

Beyond that, you may never write any of this yourself, but as always, the libraries you depend on may. For example, the regex crate contains these SIMD speedups without you needing to do anything at all!

Macros

In this chapter of the guide, we discuss a few improvements to the macro system. A notable addition here is the introduction of custom derive macros.

Custom Derive

Minimum Rust Version 1.15

In Rust, you've always been able to automatically implement some traits through the derive attribute:

```
#[derive(Debug)]
struct Pet {
    name: String,
}
```

The `Debug` trait is then implemented for `Pet`, with vastly less boilerplate. For example, without `derive`, you'd have to write this:

```
use std::fmt;

struct Pet {
    name: String,
}

impl fmt::Debug for Pet {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            Pet { name } => {
                let mut debug_trait_builder = f.debug_struct("Pet");

                let _ = debug_trait_builder.field("name", name);

                debug_trait_builder.finish()
            }
        }
    }
}
```

Whew!

However, this only worked for traits provided as part of the standard library; it was not customizable. But now, you can tell Rust what to do when someone wants to derive your trait. This is used heavily in popular crates like serde and Diesel.

For more, including learning how to build your own custom derive, see The Rust Programming Language.

Macro changes

Minimum Rust Version beta

`macro_rules!` style macros

In Rust 2018, you can import specific macros from external crates via `use` statements, rather than the old `#[macro_use]` attribute.

For example, consider a `bar` crate that implements a `baz!` macro. In `src/lib.rs`:

```rust
#[macro_export]
macro_rules! baz {
    () => ()
}
```

In your crate, you would have written

```rust
// Rust 2015

#[macro_use]
extern crate bar;

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

Now, you write:

```rust
// Rust 2018

use bar::baz;

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

This moves `macro_rules` macros to be a bit closer to other kinds of items.

Note that you'll still need `#[macro_use]` to use macros you've defined in your own crate; this feature only works for importing macros from external crates.

Procedural macros

When using procedural macros to derive traits, you will have to name the macro that provides the custom derive. This generally matches the name of the trait, but check with the documentation of the crate providing the derives to be sure.

For example, with Serde you would have written

```
// Rust 2015
extern crate serde;
#[macro_use] extern crate serde_derive;

#[derive(Serialize, Deserialize)]
struct Bar;
```

Now, you write instead:

```
// Rust 2018
use serde_derive::{Serialize, Deserialize};

#[derive(Serialize, Deserialize)]
struct Bar;
```

More details

This only works for macros defined in external crates. For macros defined locally, `#[macro_use] mod foo;` is still required, as it was in Rust 2015.

Local helper macros

Sometimes it is helpful or necessary to have helper macros inside your module. This can make supporting both versions of rust more complicated.

For example, let's make a simplified (and slightly contrived) version of the `log` crate in 2015 edition style:

```rust
use std::fmt;

/// How important/severe the log message is.
#[derive(Copy, Clone)]
pub enum LogLevel {
    Warn,
    Error
}

impl fmt::Display for LogLevel {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            LogLevel::Warn => write!(f, "warning"),
            LogLevel::Error => write!(f, "error"),
        }
    }
}

// A helper macro to log the message.
#[doc(hidden)]
#[macro_export]
macro_rules! __impl_log {
    ($level:expr, $msg:expr) => {{
        println!("{}: {}", $level, $msg)
    }}
}

/// Warn level log message
#[macro_export]
macro_rules! warn {
    ($($args:tt)*) => {
        __impl_log!($crate::LogLevel::Warn, format_args!($($args)*))
    }
}

/// Error level log message
#[macro_export]
macro_rules! error {
    ($($args:tt)*) => {
        __impl_log!($crate::LogLevel::Error, format_args!($($args)*))
    }
}
```

Our `__impl_log!` macro is private to our module, but needs to be exported as it is called by other macros, and in 2015 edition all used macros must be exported.

Now, in 2018 this example will not compile:

```rust
use log::error;

fn main() {
    error!("error message");
}
```

will give an error message about not finding the `__impl_log!` macro. This is because unlike in the 2015 edition, macros are namespaced and we must import them. We could do

```rust
use log::{__impl_log, error};
```

which would make our code compile, but `__impl_log` is meant to be an implementation detail!

Macros with `$crate::` prefix.

The cleanest way to handle this situation is to use the `$crate::` prefix for macros, the same as you would for any other path. Versions of the compiler >= 1.30 will handle this in both editions:

```rust
macro_rules! warn {
    ($($args:tt)*) => {
        $crate::__impl_log!($crate::LogLevel::Warn, format_args!($($args)*))
    }
}

// ...
```

However, this will not work for older versions of the compiler that don't understand the `$crate::` prefix for macros.

Macros using `local_inner_macros`

We also have the `local_inner_macros` modifier that we can add to our `#[macro_export]` attribute. This has the advantage of working with older rustc versions (older versions just ignore the extra modifier). The downside is that it's a bit messier:

```rust
#[macro_export(local_inner_macros)]
macro_rules! warn {
    ($($args:tt)*) => {
        __impl_log!($crate::LogLevel::Warn, format_args!($($args)*))
    }
}
```

So the code knows to look for any macros used locally. But wait - this won't compile, because we use the `format_args!` macro that isn't in our local crate (hence the convoluted example). The solution is to add a level of indirection: we create a macro that wraps `format_args`, but is local to our crate. That way everything works in both editions (sadly we have to pollute the global namespace a bit, but that's ok).

```rust
// I've used the pattern `_<my crate  name>__<macro name>` to name this macro,
hopefully avoiding
// name clashes.
#[doc(hidden)]
#[macro_export]
macro_rules! _log__format_args {
    ($($inner:tt)*) => {
        format_args! { $($inner)* }
    }
}
```

Here we're using the most general macro pattern possible, a list of token trees. We just pass whatever tokens we get to the inner macro, and rely on it to report errors.

So the full 2015/2018 working example would be:

```rust
use std::fmt;

/// How important/severe the log message is.
#[derive(Debug, Copy, Clone)]
pub enum LogLevel {
    Warn,
    Error
}

impl fmt::Display for LogLevel {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            LogLevel::Warn => write!(f, "warning"),
            LogLevel::Error => write!(f, "error"),
        }
    }
}

// A helper macro to log the message.
#[doc(hidden)]
#[macro_export]
macro_rules! __impl_log {
    ($level:expr, $msg:expr) => {{
        println!("{}: {}", $level, $msg)
    }}
}

/// Warn level log message
#[macro_export(local_inner_macros)]
macro_rules! warn {
    ($($args:tt)*) => {
        __impl_log!($crate::LogLevel::Warn, _log__format_args!($($args)*))
    }
}

/// Error level log message
#[macro_export(local_inner_macros)]
macro_rules! error {
    ($($args:tt)*) => {
        __impl_log!($crate::LogLevel::Error, _log__format_args!($($args)*))
    }
}

#[doc(hidden)]
#[macro_export]
macro_rules! _log__format_args {
    ($($inner:tt)*) => {
        format_args! { $($inner)* }
    }
}
```

Once everyone is using a rustc version >= 1.30, we can all just use the `$crate::` method (2015 crates are guaranteed to carry on compiling fine with later versions of the compiler). We need to wait for package managers and larger organisations to update their compilers before this happens, so in the mean time we can use the `local_inner_macros` method to support everybody. :)

At most one repetition

In Rust 2018, we have made a couple of changes to the macros-by-example syntax.

1. We have added a new Kleene operator `?` which means "at most one" repetition. This operator does not accept a separator token.
2. We have disallowed using `?` as a separator to remove ambiguity with `?`.

For example, consider the following Rust 2015 code:

```
# #![allow(unused_variables)]
#fn main() {
macro_rules! foo {
    ($a:ident, $b:expr) => {
        println!("{}", $a);
        println!("{}", $b);
    }
    ($a:ident) => {
        println!("{}", $a);
    }
}
#}
```

Macro `foo` can be called with 1 or 2 arguments; the second one is optional, but you need a whole other matcher to represent this possibility. This is annoying if your matchers are long. In Rust 2018, one can simply write the following:

```
# #![allow(unused_variables)]
#fn main() {
macro_rules! foo {
    ($a:ident $(, $b:expr)?) => {
        println!("{}", $a);

        $(
            println!("{}", $b);
         )?
    }
}
#}
```

The compiler

In this chapter of the guide, we discuss a few improvements to the compiler. A notable addition here is our new and improved error messages.

Improved error messages

Minimum Rust Version 1.12

We're always working on error improvements, and there are little improvements in almost every Rust version, but in Rust 1.12, a significant overhaul of the error message system was created.

For example, here's some code that produces an error:

```rust
fn main() {
    let mut x = 5;

    let y = &x;

    x += 1;
}
```

Here's the error in Rust 1.11:

```
foo.rs:6:5: 6:11 error: cannot assign to `x` because it is borrowed [E0506]
foo.rs:6     x += 1;
             ^~~~~~
foo.rs:4:14: 4:15 note: borrow of `x` occurs here
foo.rs:4     let y = &x;
                      ^
foo.rs:6:5: 6:11 help: run `rustc --explain E0506` to see a detailed explanation
```

Here's the error in Rust 1.28:

```
error[E0506]: cannot assign to `x` because it is borrowed
 --> foo.rs:6:5
  |
4 |     let y = &x;
  |              - borrow of `x` occurs here
5 |
6 |     x += 1;
  |     ^^^^^^ assignment to borrowed `x` occurs here

error: aborting due to previous error
```

This error isn't terribly different, but shows off how the format has changed. It shows off your code in context, rather than just showing the text of the lines themselves.

Incremental Compilation

Minimum Rust Version 1.24

Back in September of 2016, we blogged about Incremental Compilation. While that post goes into the details, the idea is basically this: when you're working on a project, you often compile it, then change something small, then compile again. Historically, the compiler has compiled your entire project, no matter how little you've changed the code. The idea with incremental compilation is that you only need to compile the code you've actually changed, which means that that second build is faster.

This is now turned on by default. This means that your builds should be faster! Don't forget about cargo check when trying to get the lowest possible build times.

This is still not the end story for compiler performance generally, nor incremental compilation specifically. We have a lot more work planned in the future.

One small note about this change: it makes builds faster, but makes the final binary a bit slower. For this reason, it's not turned on in release builds.

An attribute for deprecation

Minimum Rust Version 1.9

If you're writing a library, and you'd like to deprecate something, you can use the `deprecated` attribute:

```rust
#[deprecated(
    since = "0.2.1",
    note = "Please use the bar function instead"
)]
pub fn foo() {
    // ...
}
```

This will give your users a warning if they use the deprecated functionality:

```
   Compiling playground v0.0.1 (file:///playground)
warning: use of deprecated item 'foo': Please use the bar function instead
  --> src/main.rs:10:5
    |
10  |     foo();
    |     ^^^
    |
    = note: #[warn(deprecated)] on by default
```

Both `since` and `note` are optional.

`since` can be in the future; you can put whatever you'd like, and what's put in there isn't checked.

Rustup for managing Rust versions

Minimum Rust Version various (this tool has its own versioning scheme and works with all Rust versions)

The Rustup tool has become *the* recommended way to install Rust, and is advertised on our website. Its powers go further than that though, allowing you to manage various versions, components, and platforms.

For installing Rust

To install Rust through Rustup, you can go to https://www.rust-lang.org/install.html, which will let you know how to do so on your platform. This will install both `rustup` itself and the `stable` version of `rustc` and `cargo`.

To install a specific Rust version, you can use `rustup install`:

```
$ rustup install 1.30.0
```

This works for a specific nightly, as well:

```
$ rustup install nightly-2018-08-01
```

As well as any of our release channels:

```
$ rustup install stable
$ rustup install beta
$ rustup install nightly
```

For updating your installation

To update all of the various channels you may have installed:

```
$ rustup update
```

This will look at everything you've installed, and if there are new releases, will update anything that has one.

Managing versions

To set the default toolchain to something other than `stable`:

```
$ rustup toolchain default nightly
```

To use a toolchain other than the default, use `rustup run`:

```
$ rustup run nightly cargo build
```

There's also an alias for this that's a little shorter:

```
$ cargo +nightly build
```

If you'd like to have a different default per-directory, that's easy too! If you run this inside of a project:

```
$ rustup override set nightly
```

Then when you're in that directory, any invocations of `rustc` or `cargo` will use that toolchain. To share this with others, you can create a `rust-toolchain` file with the contents of a toolchain, and check it into source control. Now, when someone clones your project, they'll get the right version without needing to `override set` themselves.

Installing other targets

Rust supports cross-compiling to other targets, and Rustup can help you manage them. For example, to use MUSL:

```
$ rustup target add x86_64-unknown-linux-musl
```

And then you can

```
$ cargo build --target=x86_64-unknown-linux-musl
```

To see the full list of targets you can install:

```
$ rustup target list
```

Installing components

Components are used to install certain kinds of tools. While `cargo-install` has you covered for most tools, some tools need deep integration into the compiler. Rustup knows exactly what version of the compiler you're using, and so it's got just the information that these tools need.

Components are per-toolchain, so if you want them to be available to more than one toolchain, you'll need to install them multiple times. In the following examples, add a

`--toolchain` flag, set to the toolchain you want to install for, `nightly` for example. Without this flag, it will install the component for the default toolchain.

To see the full list of components you can install:

```
$ rustup component list
```

Next, let's talk about some popular components and when you might want to install them.

`rust-docs`, for local documentation

This first component is installed by default when you install a toolchain. It contains a copy of Rust's documentation, so that you can read it offline.

This component cannot be removed for now; if that's of interest, please comment on this issue.

`rust-src` for a copy of Rust's source code

The `rust-src` component can give you a local copy of Rust's source code. Why might you need this? Well, autocompletion tools like Racer use this information to know more about the functions you're trying to call.

```
$ rustup component add rust-src
```

The "preview" components

There are several components in a "preview" stage. These components currently have `-preview` in their name, and this indicates that they're not quite 100% ready for general consumption yet. Please try them out and give us feedback, but know that they do not follow Rust's stability guarantees, and are still actively changing, possibly in backwards-incompatible ways.

`rustfmt-preview` for automatic code formatting

Minimum Rust Version 1.24

If you'd like to have your code automatically formatted, you can install this component:

```
$ rustup component add rustfmt-preview
```

This will install two tools, `rustfmt` and `cargo-fmt`, that will reformat your code for you! For example:

```
$ cargo fmt
```

will reformat your entire Cargo project.

`rls-preview` for IDE integration

Minimum Rust Version 1.21

Many IDE features are built off of the `langserver` protocol. To gain support for Rust with these IDEs, you'll need to install the Rust language sever, aka the "RLS":

```
$ rustup component add rls-preview
```

Your IDE should take it from there.

`clippy-preview` for more lints

For even more lints to help you write Rust code, you can install `clippy`:

```
$ rustup component add clippy-preview
```

This will install `cargo-clippy` for you:

```
$ cargo clippy
```

For more, check out clippy's documentation.

`llvm-tools-preview` for using extra LLVM tools

If you'd like to use the `lld` linker, or other tools like `llvm-objdump` or `llvm-objcopy`, you can install this component:

```
$ rustup component add llvm-tools-preview
```

This is the newest component, and so doesn't have good documentation at the moment.

Cargo and crates.io

In this chapter of the guide, we discuss a few improvements to `cargo` and crates.io. A notable addition here is the new `cargo check` command.

`cargo check` for faster checking

Minimum Rust Version 1.16

`cargo check` is a new subcommand should speed up the development workflow in many cases.

What does it do? Let's take a step back and talk about how `rustc` compiles your code. Compilation has many "passes", that is, there are many distinct steps that the compiler takes on the road from your source code to producing the final binary. However, you can think of this process in two big steps: first, `rustc` does all of its safety checks, makes sure your syntax is correct, all that stuff. Second, once it's satisfied that everything is in order, it produces the actual binary code that you end up executing.

It turns out that that second step takes a lot of time. And most of the time, it's not neccesary. That is, when you're working on some Rust code, many developers will get into a workflow like this:

1. Write some code.
2. Run `cargo build` to make sure it compiles.
3. Repeat 1-2 as needed.
4. Run `cargo test` to make sure your tests pass.
5. Try the binary yourself
6. GOTO 1.

In step two, you never actually run your code. You're looking for feedback from the compiler, not to actually run the binary. `cargo check` supports exactly this use-case: it runs all of the compiler's checks, but doesn't produce the final binary. To use it:

```
$ cargo check
```

where you may normally `cargo build`. The workflow now looks like:

1. Write some code.
2. Run `cargo check` to make sure it compiles.
3. Repeat 1-2 as needed.
4. Run `cargo test` to make sure your tests pass.
5. Run `cargo build` to build a binary and try it yourself
6. GOTO 1.

So how much speedup do you actually get? Like most performance related questions, the answer is "it depends." Here are some very un-scientific benchmarks at the time of writing.

build	performance	check performance	speedup
initial compile	11s	5.6s	1.96x
second compile (no changes)	3s	1.9s	1.57x
third compile with small change	5.8s	3s	1.93x

`cargo install` for easy installation of tools

Minimum Rust Version 1.5

Cargo has grown a new `install` command. This is intended to be used for installing new subcommands for Cargo, or tools for Rust developers. This doesn't replace the need to build real, native packages for end-users on the platforms you support.

For example, this guide is created with `mdbook`. You can install it on your system with

```
$ cargo install mdbook
```

And then use it with

```
$ mdbook --help
```

As an example of extending Cargo, you can use the `cargo-update` package. To install it:

```
$ cargo install cargo-update
```

This will allow you to use this command, which checks everything you've `cargo install`'d and updates it to the latest version:

```
$ cargo install-update -a
```

`cargo new` defaults to a binary project

Minimum Rust Version 1.25

`cargo new` will now default to generating a binary, rather than a library. We try to keep Cargo's CLI quite stable, but this change is important, and is unlikely to cause breakage.

For some background, cargo new accepts two flags: `--lib`, for creating libraries, and `--bin`, for creating binaries, or executables. If you don't pass one of these flags, it used to default to `--lib`. At the time, we made this decision because each binary (often) depends on many libraries, and so we thought the library case would be more common. However, this is incorrect; each library is depended upon by many binaries. Furthermore, when getting started, what you often want is a program you can run and play around with. It's not just new Rustaceans though; even very long-time community members have said that they find this default surprising. As such, we've changed it, and it now defaults to `--bin`.

`cargo rustc` for passing arbitrary flags to rustc

Minimum Rust Version 1.1

`cargo rustc` is a new subcommand for Cargo that allows you to pass arbitrary `rustc` flags through Cargo.

For example, Cargo does not have a way to pass unstable flags built-in. But if we'd like to use `print-type-sizes` to see what layout information our types have. We can run this:

```
$ cargo rustc -- -Z print-type-sizes
```

And we'll get a bunch of output describing the size of our types.

Note

`cargo rustc` only passes these flags to invocations of your crate, and not to any `rustc` invocations used to build dependencies. If you'd like to do that, see `$RUSTFLAGS`.

Cargo workspaces for multi-package projects

Minimum Rust Version 1.12

Cargo used to have two levels of organization:

- A *package* contains one or more crates
- A crate has one or more modules

Cargo now has an additional level:

- A *workspace* contains one or more packages

This can be useful for larger projects. For example, the `futures` package is a *workspace* that contains many related packages:

- futures
- futures-util
- futures-io
- futures-channel

and more.

Workspaces allow these packages to be developed individually, but they share a single set of dependencies, and therefore have a single target directory and a single `Cargo.lock`.

For more details about workspaces, please see the Cargo documentation.

Multi-file examples

Minimum Rust Version 1.22

Cargo has an `examples` feature for showing people how to use your package. By putting individual files inside of the top-level `examples` directory, you can create multiple examples.

But what if your example is too big for a single file? Cargo supports adding sub-directories inside of `examples`, and looks for a `main.rs` inside of them to build the example. It looks like this:

```
my-package
 └──src
     └── lib.rs // code here
 └──examples
     └── simple-example.rs // a single-file example
     └── complex-example
        └── helper.rs
        └── main.rs // a more complex example that also uses `helper` as a
submodule
```

Replacing dependencies with patch

Minimum Rust Version 1.21

The `[patch]` section of your `Cargo.toml` can be used when you want to override certain parts of your dependency graph.

> Cargo has a `[replace]` feature that is similar; while we don't intend to deprecate or remove `[replace]`, you should prefer `[patch]` in all circumstances.

So what’s it look like? Let’s say we have a Cargo.toml that looks like this:

```
[dependencies]
foo = "1.2.3"
```

In addition, our `foo` package depends on a `bar` crate, and we find a bug in `bar`. To test this out, we’d download the source code for `bar`, and then update our `Cargo.toml`:

```
[dependencies]
foo = "1.2.3"

[patch.crates-io]
bar = { path = '/path/to/bar' }
```

Now, when you `cargo build`, it will use the local version of `bar`, rather than the one from crates.io that `foo` depends on. You can then try out your changes, and fix that bug!

For more details, see the documentation for `patch`.

Cargo can use a local registry replacement

Minimum Rust Version 1.12

Cargo finds its packages in a "source". The default source is crates.io. However, you can choose a different source in your `.cargo/config`:

```
[source.crates-io]
replace-with = 'my-awesome-registry'

[source.my-awesome-registry]
registry = 'https://github.com/my-awesome/registry-index'
```

This configuration means that instead of using crates.io, Cargo will query the `my-awesome-registry` source instead (configured to a different index here). This alternate source *must be the exact same* as the crates.io index. Cargo assumes that replacement sources are exact 1:1 mirrors in this respect, and the following support is designed around that assumption.

When generating a lock file for crate using a replacement registry, the original registry will be encoded into the lock file. For example in the configuration above, all lock files will still mention crates.io as the registry that packages originated from. This semantically represents how crates.io is the source of truth for all crates, and this is upheld because all replacements have a 1:1 correspondance.

Overall, this means that no matter what replacement source you're working with, you can ship your lock file to anyone else and you'll all still have verifiably reproducible builds!

This has enabled tools like `cargo-vendor` and `cargo-local-registry`, which are often useful for "offline builds." They prepare the list of all Rust dependencies ahead of time, which lets you ship them to a build machine with ease.

Crates.io disallows wildcard dependencies

Minimum Rust Version 1.6

Crates.io will not allow you to upload a package with a wildcard dependency. In other words, these:

```
[dependencies]
regex = "*"
```

A wildcard dependency means that you work with any possible version of your dependency. This is highly unlikely to be true, and would cause unnecessary breakage in the ecosystem.

Instead, depend on a version range. For example, `^` is the default, so you could use

```
[dependencies]
regex = "1.0.0"
```

instead. `>`, `<=`, and all of the other, non-`*` ranges work as well.

Documentation

In this chapter of the guide, we discuss a few improvements to documentation. A notable addition here is the second edition of "the book".

New editions of the "the book"

Minimum Rust Version 1.18 for drafts of the second edition

Minimum Rust Version 1.26 for the final version of the second edition

Minimum Rust Version 1.28 for drafts of the 2018 edition

We've distributed a copy of "The Rust Programming Language," affectionately nicknamed "the book", with every version of Rust since Rust 1.0.

However, because it was written before Rust 1.0, it started showing its age. Many parts of the book are vague, because it was written before the true details were nailed down for the 1.0 release. It didn't do a fantastic job of teaching lifetimes.

Starting with Rust 1.18, we shipped drafts of a second edition of the book. The final version was shipped with Rust 1.26. The new edition is a complete re-write from the ground up, using the last two years of knowledge we've gained from teaching people Rust. You'll find brand-new explanations for a lot of Rust's core concepts, new projects to build, and all kinds of other good stuff. Please check it out and let us know what you think!

You can also purchase a dead-tree version from No Starch Press. Now that the print version has shipped, the second edition is frozen.

The names are a bit confusing though, because the "second edition" of the book is the first printed edition of the book. As such, we decided that newer editions of the book will correspond with newer editions of Rust itself, and so starting with 1.28, we've been shipping drafts of the next version, the 2018 Edition. It's still pretty close to the second edition, but contains information about newer features since the book's content was frozen. We'll be continuing to update this edition until we decide to print a second edition in paper.

The Rust Bookshelf

Minimum Rust Version various, each book is different.

As Rust's documentation has grown, we've gained far more than just "The book" and the reference. We now have a collection of various long-form docs, nicknamed "the Rust Bookshelf." Different resources are added at various times, and we're adding new ones as more get written.

The Cargo book

Minimum Rust Version 1.21

Historically, Cargo’s docs were hosted on http://doc.crates.io, which doesn’t follow the release train model, even though Cargo itself does. This led to situations where a feature would land in Cargo nightly, the docs would be updated, and then for up to twelve weeks, users would think that it should work, but it wouldn’t yet. https://doc.rust-lang.org/cargo is the new home of Cargo’s docs, and http://doc.crates.io now redirects there.

The `rustdoc` book

Minimum Rust Version 1.21

Rustdoc, our documentation tool, now has a guide at https://doc.rust-lang.org/rustdoc.

Rust By Example

Minimum Rust Version 1.25

Rust by Example used to live at https://rustbyexample.com, but now is part of the Bookshelf! It can be found at https://doc.rust-lang.org/rust-by-example/. RBE lets you learn Rust through short code examples and exercises, as opposed to the lengthy prose of The Book.

The Rustonomicon

Minimum Rust Version 1.3

We now have a draft book, The Rustonomicon: the Dark Arts of Advanced and Unsafe Rust Programming.

From the title, I'm sure you can guess: this book discusses some advanced topics, including `unsafe`. It's a must-read for anyone who's working at the lowest levels with Rust.

`std::os` has documentation for all platforms

Minimum Rust Version 1.21

The `std::os` module contains operating system specific functionality. You'll now see more than just linux, the platform we build the documentation on.

We've long regretted that the hosted version of the documentation has been Linux-specific; this is a first step towards rectifying that. This is specific to the standard library and not for general use; we hope to improve this further in the future.

`rustdoc`

In this chapter of the guide, we discuss a few improvements to `rustdoc`. A notable addition to it was that documentation tests can now compile-fail.

Documentation tests can now compile-fail

Minimum Rust Version 1.22

You can now create `compile-fail` tests in Rustdoc, like this:

```
/// ```compile_fail
/// let x = 5;
/// x += 2; // shouldn't compile!
/// ```
# fn foo() {}
```

Please note that these kinds of tests can be more fragile than others, as additions to Rust may cause code to compile when it previously would not. Consider the first release with `?`, for example: code using `?` would fail to compile on Rust 1.21, but compile successfully on Rust 1.22, causing your test suite to start failing.

Rustdoc uses CommonMark

Minimum Rust Version 1.25 for support by default

Minimum Rust Version 1.23 for support via a flag

Rustdoc lets you write documentation comments in Markdown. At Rust 1.0, we were using the `hoedown` markdown implementation, written in C. Markdown is more of a family of implementations of an idea, and so `hoedown` had its own dialect, like many parsers. The CommonMark project has attempted to define a more strict version of Markdown, and so now, Rustdoc uses it by default.

As of Rust 1.23, we still defaulted to `hoedown`, but you could enable Commonmark via a flag, `--enable-commonmark`. Today, we only support CommonMark.

Platform and target support

In this chapter of the guide, we discuss a few improvements to platform and target support. A notable addition to it was that the `libcore` library now works on stable Rust.

libcore for low-level Rust

Minimum Rust Version 1.6

Rust's standard library is two-tiered: there's a small core library, `libcore`, and the full standard library, `libstd`, that builds on top of it. `libcore` is completely platform agnostic, and requires only a handful of external symbols to be defined. Rust's `libstd` builds on top of `libcore`, adding support for things like memory allocation and I/O. Applications using Rust in the embedded space, as well as those writing operating systems, often eschew `libstd`, using only `libcore`.

As an additional note, while building *libraries* with `libcore` is supported today, building full applications is not yet stable.

To use `libcore`, add this flag to your crate root:

```
#![no_std]
```

This will remove the standard library, and bring the `core` crate into your namespace for use:

```
#![no_std]

use core::cell::Cell;
```

You can find `libcore`'s documentation here.

WebAssembly support

Minimum Rust Version 1.14 for `emscripten`

Minimum Rust Version nightly for `wasm32-unknown-unknown`

Rust has gained support for WebAssembly, meaning that you can run Rust code in your browser, client-side.

In Rust 1.14, we gained support through emscripten. With it installed, you can write Rust code and have it produce asm.js (the precusor to wasm) and/or WebAssembly.

Here's an example of using this support:

```
$ rustup target add wasm32-unknown-emscripten
$ echo 'fn main() { println!("Hello, Emscripten!"); }' > hello.rs
$ rustc --target=wasm32-unknown-emscripten hello.rs
$ node hello.js
```

However, in the meantime, Rust has also grown its own support, independent from Emscripten. This is known as "the unknown target", because instead of `wasm32-unknown-emscripten`, it's `wasm32-unknown-unknown`. This will be the preferred target to use once it's ready, but for now, it's really only well-supported in nightly.

Global allocators

Minimum Rust Version 1.28

Allocators are the way that programs in Rust obtain memory from the system at runtime. Previously, Rust did not allow changing the way memory is obtained, which prevented some use cases. On some platforms, this meant using jemalloc, on others, the system allocator, but there was no way for users to control this key component. With 1.28.0, the `#[global_allocator]` attribute is now stable, which allows Rust programs to set their allocator to the system allocator, as well as define new allocators by implementing the `GlobalAlloc` trait.

The default allocator for Rust programs on some platforms is jemalloc. The standard library now provides a handle to the system allocator, which can be used to switch to the system allocator when desired, by declaring a static and marking it with the `#[global_allocator]` attribute.

```rust
use std::alloc::System;

#[global_allocator]
static GLOBAL: System = System;

fn main() {
    let mut v = Vec::new();
    // This will allocate memory using the system allocator.
    v.push(1);
}
```

However, sometimes you want to define a custom allocator for a given application domain. This is also relatively easy to do by implementing the `GlobalAlloc` trait. You can read more about how to do this in the documentation.

MSVC toolchain support

Minimum Rust Version 1.2

At the release of Rust 1.0, we only supported the GNU toolchain on Windows. With the release of Rust 1.2, we introduced initial support for the MSVC toolchain. After that, as support matured, we eventually made it the default choice for Windows users.

The difference between the two matters for interacting with C. If you're using a library built with one toolchain or another, you need to match that with the appropriate Rust toolchain. If you're not sure, go with MSVC; it's the default for good reason.

To use this feature, simply use Rust on Windows, and the installer will default to it. If you'd prefer to switch to the GNU toolchain, you can install it with Rustup:

```
$ rustup toolchain install stable-x86_64-pc-windows-gnu
```

MUSL support for fully static binaries

Minimum Rust Version 1.1

By default, Rust will statically link all Rust code. However, if you use the standard library, it will dynamically link to the system's `libc` implementation.

If you'd like a 100% static binary, the `MUSL libc` can be used on Linux.

Installing MUSL support

To add support for MUSL, you need to choose the correct target. The forge has a full list of targets supported, with a number of ones using `musl`.

If you're not sure what you want, it's probably `x86_64-unknown-linux-musl`, for 64-bit Linux. We'll be using this target in this guide, but the instructions remain the same for other targets, just change the name wherever we mention the target.

To get support for this target, you use `rustup`:

```
$ rustup target add x86_64-unknown-linux-musl
```

This will install support for the default toolchain; to install for other toolchains, add the `--toolchain` flag. For example:

```
$ rustup target add x86_64-unknown-linux-musl --toolchain=nightly
```

Building with MUSL

To use this new target, pass the `--target` flag to Cargo:

```
$ cargo build --target x86_64-unknown-linux-musl
```

The binary produced will now be built with MUSL!

cdylib crates for C interoperability

Minimum Rust Version 1.10 for `rustc`

Minimum Rust Version 1.11 for `cargo`

If you're producing a library that you intend to be used from C (or another language through a C FFI), there's no need for Rust to include Rust-specific stuff in the final object code. For libraries like that, you'll want to use the `cdylib` crate type in your `Cargo.toml`:

```
[lib]
crate-type = ["cdylib"]
```

This will produce a smaller binary, with no Rust-specific information inside of it.