The standard library is special. Let’s change that.

■ Intro

Hello everyone! Today I will be talking about the standard library in Rust. More specifically — about how the standard library has a special place within the Rust ecosystem and how and why we should change that.

■ Disclaimer

Before I begin, I would like to provide a brief disclaimer. No member of any team has seen this talk; everything here is my opinion alone. This is partly observational and partly aspirational. Needless to say, the latter is not guaranteed to happen.

■ How is the standard library special?

So, what exactly do I mean when I say the standard library is “special”? And more importantly, why should I care? As it turns out, the standard library is special in a number of ways.

First, ■ the standard library uses nightly language features ■ …on stable. While a large number of these features are intended to be stabilized, some of them are intended to remain on nightly forever. These should be removed wherever possible.

The standard library is also ■ capable of bypassing coherence. This is not strictly necessary, either. Currently, `core`, `alloc`, and `std` are three separate crates. If they were merged into a single crate, the need for bypassing coherence would be wholly eliminated.

Another manner in which the standard library is unique is that it ■ has a prelude. Other crates have preludes as well, but they are not *true* preludes, as they must be manually imported in every module.

The most obvious situation where the standard library is special is that it is ■ not included with Cargo. While seemingly inconsequential, it does have real-world effects.

The standard library does not have any publicly-exposed feature flags that could be used for a capability-based library. Among other possibilities, this would permit for file system or network access to be disabled. `std` is also built for a single optimization goal — speed — while use cases like embedded systems may reasonably want to optimize for size instead. Another limitation is that `std` is not independently versioned and is incapable of making breaking changes. There is a possible future where breaking changes could be made without `std` 2.0, but I will not be going into this today.

Finally, **we already are making the standard library less special**. I think it makes sense for the effort to be made in an organized manner rather than *ad hoc*, as it has been for years.

Minimal subset

Okay, so we want to make the standard library less special. Surely there is a limit to this, right? In a word, yes. But that raises a question: *what inherently must be special-cased*?

Language items

First up are language items. A language item is a function, type, trait, et cetera that the compiler needs to know about for a variety of reasons. There are **130** of these. As an example, the `Add` trait is a lang item because the compiler needs to know about it in order to support the "plus" operator. Many of these are necessary, such as those used for syntax integration and the runtime for Rust (namely panicking and allocation). However, the number of lang items has been increasing over time, in part because some are unnecessary.

`Result` and its variants are all lang items. Why? Optimization. They were not lang items until August 2020, when they were added to give a slight performance gain. So why not instead implement an optimization for all `enum`s that are like `Result`? It avoids the need for a language item and is generally applicable to the entire ecosystem. That would give a performance benefit for all crates, not just the standard library.

Compiler intrinsics

Another item that must be special-cased is compiler intrinsics. As its name suggests, an intrinsic is a function that is implemented in the compiler itself. There are **232** of these, and every backend must implement them. Some exist to interact directly with hardware; 101 exist just for atomics. Other intrinsics exist to access information that only the compiler has. *How large is a type*? That is impossible to answer without asking the compiler.

🟩 Like language items, a number of compiler intrinsics can be removed. Taking the square root of a floating point value is an intrinsic, but this can be rewritten to use inline assembly. On 64-bit x86 architectures, I have actually verified that this is possible without any side effects. Other examples that I easily found are `assume`, which can be trivially implemented in terms of `unreachable`, and `unlikely` in terms of `likely`.

Language items and compiler intrinsics form the backbone of what must inherently be special-cased. For that reason, the number should be shrunk to the bare minimum and moved to its own crate, which will remain special. This would permit the rest of the standard library to continue its journey towards being "just another crate".

Prior history

In the beginning, I mentioned that this talk was partly observational. This is because making `std` less special is not a new idea. There has been a fair amount of work made on this front already, going back multiple years.

🟩 There is a working group, called the "`std` aware Cargo" working group, whose goal is to permit users to build `std` locally. 🟩 An experimental implementation of this exists on nightly. Bugs do exist, such as not working with code coverage, but it is largely usable. The ergonomics can and will be improved in the future, as it can sometimes be finicky to make what you want to happen, happen.

In more concrete terms, there is also 🟩 `#[diagnostic::on_unimplemented]`. This has existed inside the standard library for quite some time as 🟩 `#[rustc_on_unimplemented]`. It improves error messages when a trait is not implemented when it is expected to be, likely due to a trait bound. Moving this away from a `rustc` attribute in favor of a `diagnostic` namespace 🟩 has been accepted by an RFC, which is partially implemented at the moment.

Perhaps the most surprising item in this list is that 🟩 the `deprecated` attribute was originally only for the standard library. It has a lengthy history. Beginning prior to the Rust 1.0 release, 🟩 the original implementation was renamed to `rustc_deprecated` and made available only to the standard library. A new `deprecated` attribute was then introduced, which shared no code whatsoever with `rustc_deprecated`. The `deprecated` attribute was stabilized about a year after Rust 1.0 was released, but `rustc_deprecated` remained. This was the case until March 2022 when I fully merged the front end for the two attributes, at which point the functionality was nearly identical.

I say *nearly* identical because there is still one difference.

Suggestions on deprecated items

That is suggestions for deprecated items. This allows authors to indicate what a deprecated item is replaced with. This currently has to be done via the `note` field, which requires the end user to read and manually apply the change.

In this example, `alpha` has been deprecated in favor of `beta`. This is indicated with a new `suggestion` field. With the suggestion, the compiler will produce better diagnostics; it will clearly show that calls to `alpha` must be replaced with calls to `beta`. This suggestion is machine-applicable, by the way, so it is usable with both `cargo fix` and `rust-anaylzer`!

This feature has been implemented since January 2019 when it was still part of the `rustc_deprecated` attribute. When that was merged with `deprecated`, the feature was made available to everyone on nightly, which is its current status.

Unspecified behavior

There is one interesting manner in which the standard library is unique. That is the fact that the standard library relies on unspecified behavior. There is a comment in the standard library that says “only std can make this guarantee”. A handful of similar comments can be found throughout `std`. When I first ran across a comment like this, I found it peculiar. Why would the standard library do this?

During my research for this talk, I found two primary situations where the standard library relies on behavior that is not guaranteed to other libraries. One involved niche value optimization. That situation is not unspecified behavior, but rather it is not guaranteed to compile at all.

The other case is where the standard library relies on the size and layout of fat pointers, which is what this comment is regarding. Only one problem — both are unspecified. The compiler guarantees *nothing* about this code. If the layout of a pointer changes, the behavior will quietly change. Far worse is if the *size* of a pointer changes. That would result in *undefined* behavior, as some code would be performing an out of bounds read.

While it has always been the case that a fat pointer is twice the size of the `usize` type, as far as I can tell this has not been formally guaranteed. The same is true for the layout — it has never changed but is not guaranteed.

🟩 This code can only exist because the compiler is coupled with the standard library. As far as the standard library is concerned, its code works with the *current* behavior and that is all that matters. With that said, to make the standard library less special, there are two options. One is to make fat pointers into a language item, which would ensure that it is implemented by the compiler and thus remain in sync. The other option, of course, is to simply guarantee the size and layout of fat pointers. That would be preferable in my opinion, as it would also be a small but important step towards a stable ABI.

Negative implementations

On to something that is still on nightly, yet unlike most nightly features is exposed within the public API of the standard library. That feature is negative implementations. 🟩 Negative impls are functionally a promise to never implement a trait.

Likely the most common use case for wanting this is to 🟩 opt out of auto traits — namely `Send` and `Sync`. This is *technically* possible already, but doing so is quite unergonomic, as it 🟩 requires some hackery. How is this possible? Well…the standard library contains types that have a negative implementation of these traits. Specifically, 🟩 `MutexGuard` implements not `Send`, 🟩 `Cell` implements not `Sync`, and 🟩 mutable pointers implement both not `Send` and not `Sync`. Which is great, except you likely want to avoid storing these types in memory. So instead, you can 🟩 wrap them in a `PhantomData`, which does not exist at runtime. But it sure would be simpler if anyone could opt out of auto traits without this hack.

🟩 Negative impls have other uses, though! 🟩 They are also useful for trait resolution, as they allow implementations that would seemingly overlap. As an example, the standard library 🟩 has an impl that permits any type implementing `Error` to be converted into a `Box<dyn Error>`. But what if 🟩 we also wanted to allow converting a string to a `Box<dyn Error>`? For this, 🟩 we need a diagram. At the top are types that may implement `Error` (either currently or in the future), on the left are types that already do, and on the right sit types that never will. All types start at the top of the diagram, but authors *may* choose to move down to either the left or the right. The blanket impl for all error types considers all types that may implement `Error`, regardless of whether they currently do. For this reason, the second line is prohibited — strings may implement `Error` in the future. In order to satisfy the overlap check in the compiler, we must explicitly promise that strings will never implement `Error`. In doing so, we have moved strings to the right side of the diagram, eliminating them from the blanket implementation.

It is worth noting at this point that, as I said previously, all authors *may* choose to move down in the diagram. However, moving back up is a breaking change, as it is removing a promise that has been made to both the compiler and other users. This is true for both positive and negative implementations.

Specialization

One long-awaited feature of Rust — specialization. Specialization is a feature that, to an extent, allows otherwise overlapping implementations. The limitation here is that one impl must be a subset of the other. However, this still allows for very useful behavior.

Currently, `Default` is only implemented for arrays of lengths up to 32. This is because an array with length zero does not require a `Default` bound, while all other lengths do. With specialization, we could have a default impl for all lengths, specializing length zero to avoid the bound. While this seems simple enough, this example does not currently compile on nightly.

While specialization is a very powerful feature, it is also very difficult to get right. The current implementation is known to be unsound. There is `min_specialization` that attempts to avoid unsoundness, but it still falls short.

Likely due to the difficulty, work on specialization is largely stagnated. Specialization likely needs people with a fair amount of knowledge of type theory to come up with a sound subset, which would require significant effort before it could be stabilized.

Despite the difficulty, specialization is currently used for optimization. All uses are carefully checked, and none are present in the public API.

The RFC that proposed specialization was posted in July 2015, just two months after Rust 1.0 was released. In the RFC, specialization was described as a “relatively small extension to the trait system”. I think we can all agree that this was a *tad* optimistic. Specialization is probably the hardest item mentioned in this talk.

Crate preludes

While specialization may be extremely difficult, here is one that should not be. A prelude is something provided by a crate that is automatically imported in every module. This is what lets you use `Vec` without having to import it manually. The contents of `core::prelude` and `std::prelude` are implicit by default. `alloc::prelude` used to exist, but it was removed because its contents were not automatically in scope.

So I have a question. Why not let every crate declare a prelude? While some crates do have modules that they *call* prelude, their contents are not automatically in scope. What I would like to do is relatively simple from a design perspective.

First, we annotate any items that will make up part of the prelude. Any number of items can be annotated, and they do not need to be in a single, shared module. In this case, we are also renaming the item. This is the same as an ordinary `use` statement, and is used here to avoid potential naming conflicts.

What if we want to exclude the prelude in a certain location? Not an issue, modules can opt out as necessary. This would be accomplished by placing an annotation on the module and indicating the crates whose preludes we want to exclude.

Probably the most important question, what prevents crates from declaring enormous preludes, ruining the experience for everyone? This has a simple solution: leave it up to the end user! The end user chooses which preludes to use. In `Cargo.toml`, a user must opt-*in* to using a prelude from a given crate. This allows for maximum flexibility, as nothing is ever in scope without it being explicitly requested.

While custom preludes were first proposed in February 2015, this is a significant deviation from that proposal. I believe that crate preludes would provide significant benefit when used in moderation, with crates such as `itertools` and `rayon` being excellent use cases.

Stability attributes

One final item for today — stability attributes. This is likely the most visible manner in which the standard library can do things that other libraries cannot. Stability attributes are used to indicate if an item is stable or not.

As an example, `OnceCell` is stable, and the stability attribute indicates both the feature name that it was previously available under and the version in which that feature was stabilized.

What if an item is *not* stable, though? Unstable items are very useful, as they allow crates to experiment with APIs without providing a semver guarantee. ■ `LazyLock` is currently unstable. The attribute shows the feature name that is used to enable the use of `LazyLock`. More importantly, it shows the issue number! This allows users to know exactly where to look if they want to see the current status or, even better, provide feedback.

What is noteworthy is that ■ every public item requires a stability attribute if they are used anywhere in a crate. This is to ensure that everything is either stable or unstable. It is not possible to be neither, after all.

With that said, it is not possible to use unstable items freely. ■ Unstable items are opt-in and require a feature gate. ■ If you try to use an unstable item incorrectly, you *will* get a compiler error. The only way to avoid this error is to ■ add the feature gate at the crate root.

There are edge cases to consider, such as when an unstable item is used in a stable context, such as a trait bound. This is allowed, but it should absolutely have a lint to ensure that it is deliberate, as it would be confusing for downstream users if not handled carefully.

The `stable` and `unstable` attributes handle whether an item is stable in general, but there are also `const_stable` and `const_unstable` attributes to handle whether functions are guaranteed to be `const fn` going forward.

Stability attributes have been around for a very long time in the Rust world. In October 2014, it was said in an official blog post that ■ “library authors can continue to use stability attributes”. It has been almost nine years since that was said. Yet contrary to the post, stability attributes are *explicitly* only for the standard library at the moment. Trying to use them in other crates results in a warning from the compiler. I think it is time that we finally provide this functionality to everyone, as we know that it is extraordinarily useful.

■ Progress

That is quite a bit. What has actually been done on these fronts? Surprisingly, a fair amount.

■ Integration with Cargo is being worked on by a working group that exists specifically for this purpose. It is available on nightly and it is in a usable state.

■ Reducing the number of lang items and intrinsics is a goal with no work done yet. I intend to look into some of the simpler cases, including those mentioned earlier. I believe some can be trivially eliminated.

■ Suggestions on deprecated items has no formal proposal, but it has been implemented for a while. It is available on nightly under the `deprecated_suggestion` feature flag. There are a couple of points that will likely need to be resolved before stabilization, but it should not require too much effort.

■ For unspecified behavior in the standard library, this is thankfully very small in scope. There will need to be a discussion on the desired solution, but any implementation would occur soon afterwards.

■ Negative impls are implemented on nightly, but the feature has known bugs and edge cases that must be resolved before it could be stabilized.

■ Specialization is unfortunately stagnated. As far as I can tell, no one is actively working on it. However, the desire for specialization unquestionably exists. Someone with knowledge of type theory is likely needed to make progress. Given this, there is not a clear timeframe for the issues to be resolved, let alone for the feature to be stabilized.

■ As to crate preludes, I am currently writing an RFC for this. As anyone familiar with the process knows, this takes a while! There will be significant feedback and revisions before it is even accepted.

After I finish writing the RFC for crate preludes, I will ■ begin working on one for stability attributes. The attributes are widely used within the standard library, so we possess the capability and knowledge to make this happen. There are known limitations with stability attributes that should be resolved before being made widely available, but it is a solvable problem.

Overall, many of the items mentioned today have already been worked on, though the level of work varies. Some need formal proposals, while others need subject-matter experts. *All* need additional work. I am personally doing what I can to bring useful features from the standard library to everyone. I hope that you share my enthusiasm for this goal and hope that you will assist in achieving it however possible. Making the standard library less special will take a significant amount of time and effort, but it is an overarching and long-standing goal of the Rust project, and one that the Rust community at large will benefit from.

Outro

With that, there is a slew of information on the screen. You can find me on many platforms as jhpratt, including GitHub and Mastodon. If you are interested in sponsoring my work, please do so! I assure you it will be worthwhile. I am Jacob Pratt. Thank you!