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

Jacob Pratt
2023-09-13
Albuquerque

RUST CONF 2023

Disclaimer

How is the standard library special? Why change?

How is the standard library special? Why change?

- use of nightly language features

How is the standard library special? Why change?

- use of nightly language features…on stable

How is the standard library special? Why change?

- use of nightly language features…on stable
- can bypass coherence

How is the standard library special? Why change?

- use of nightly language features…on stable
- can bypass coherence
- has a prelude

How is the standard library special? Why change?

- use of nightly language features…on stable
- can bypass coherence
- has a prelude
- not included with Cargo

How is the standard library special? Why change?

- use of nightly language features…on stable
- can bypass coherence
- has a prelude
- not included with Cargo
  - no feature flags

How is the standard library special? Why change?

- use of nightly language features…on stable
- can bypass coherence
- has a prelude
- not included with Cargo
  - no feature flags
  - no options for optimization

How is the standard library special? Why change?

- use of nightly language features…on stable
- can bypass coherence
- has a prelude
- not included with Cargo
  - no feature flags
  - no options for optimization
  - no versioning

How is the standard library special? Why change?

- use of nightly language features…on stable
- can bypass coherence
- has a prelude
- not included with Cargo
    - no feature flags
    - no options for optimization
    - no versioning
    - incapable of making breaking changes

How is the standard library special? Why change?

- use of nightly language features…on stable
- can bypass coherence
- has a prelude
- not included with Cargo
  - no feature flags
  - no options for optimization
  - no versioning
  - incapable of making breaking changes
- **We already are making the standard library less special.**

Minimal subset

Minimal subset

What inherently must be special-cased?

Minimal subset

What inherently must be special-cased?

- Language items

Minimal subset

What inherently must be special-cased?

- Language items
    - There are currently **130** language items.

Minimal subset

What inherently must be special-cased?

- Language items
  - There are currently **130** language items.
  - Many are necessary; some can be removed.

Minimal subset

What inherently must be special-cased?

- Language items
  - There are currently **130** language items.
  - Many are necessary; some can be removed.
- Compiler intrinsics

Minimal subset

What inherently must be special-cased?

- Language items
  - There are currently **130** language items.
  - Many are necessary; some can be removed.
- Compiler intrinsics
  - There are currently **232** compiler intrinsics.

Minimal subset

What inherently must be special-cased?

- Language items
  - There are currently **130** language items.
  - Many are necessary; some can be removed.
- Compiler intrinsics
  - There are currently **232** compiler intrinsics.
  - We can eliminate lots of these as well.

Prior History

Prior History

- "`std` aware Cargo" working group

Prior History

- "`std` aware Cargo" working group
  - `-Zbuild-std` is available on nightly.

Prior History

- "`std` aware Cargo" working group
  - `-Zbuild-std` is available on nightly.
- `#[diagnostic::on_unimplemented]`

Prior History

- "`std` aware Cargo" working group
  - `-Zbuild-std` is available on nightly.
- `#[diagnostic::on_unimplemented]`
  - Previously `#[rustc_on_unimplemented]`

Prior History

- "`std` aware Cargo" working group
  - `-Zbuild-std` is available on nightly.
- `#[diagnostic::on_unimplemented]`
  - Previously `#[rustc_on_unimplemented]`
  - RFC-accepted, partially implemented

Prior History

- "`std` aware Cargo" working group
  - `-Zbuild-std` is available on nightly.
- `#[diagnostic::on_unimplemented]`
  - Previously `#[rustc_on_unimplemented]`
  - RFC-accepted, partially implemented
- `#[deprecated]` was originally only for the standard library.

Prior History

- "`std` aware Cargo" working group
  - `-Zbuild-std` is available on nightly.
- `#[diagnostic::on_unimplemented]`
  - Previously `#[rustc_on_unimplemented]`
  - RFC-accepted, partially implemented
- `#[deprecated]` was originally only for the standard library.
  - Original implementation renamed to `#[rustc_deprecated]` before Rust 1.0

Suggestions for deprecated items

Indicate what a deprecated item is replaced with.

Suggestions for deprecated items

Indicate what a deprecated item is replaced with.

```
#[deprecated(suggestion = "beta")]
fn alpha() {}
```

Suggestions for deprecated items

Indicate what a deprecated item is replaced with.

```
#[deprecated(suggestion = "beta")]
fn alpha() {}
```

```
help: replace the use of the deprecated function
   |
L  |     beta();
   |     ~~~~
```

Suggestions for deprecated items

Indicate what a deprecated item is replaced with.

```
#[deprecated(suggestion = "beta")]
fn alpha() {}
```

```
help: replace the use of the deprecated function
   |
L  |     beta();
   |     ~~~~
```

Usable with `cargo fix` and `rust-analyzer`.

The standard library relies on unspecified behavior.

The standard library relies on unspecified behavior.

```
// Only std can make this guarantee.
```

The standard library relies on unspecified behavior.

```
// Only std can make this guarantee.
```

- The standard library relies on the size and layout of fat pointers.

The standard library relies on unspecified behavior.

```
// Only std can make this guarantee.
```

- The standard library relies on the size and layout of fat pointers.
- Both are *unspecified*.

The standard library relies on unspecified behavior.

```
// Only std can make this guarantee.
```

- The standard library relies on the size and layout of fat pointers.
- Both are *unspecified*.
- This code can only exist because the compiler is coupled with the standard library.

Negative implementations

Negative implementations

Promise to never implement a trait

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
  - Currently requires hackery

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
  - Currently requires hackery

```
MutexGuard<'_, ()>  : !Send
```

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
  - Currently requires hackery

```
MutexGuard<'_, ()>  : !Send
Cell<()>            :         !Sync
```

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
  - Currently requires hackery

```
MutexGuard<'_, ()>  : !Send
Cell<()>            :         !Sync
*mut ()             : !Send + !Sync
```

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
  - Currently requires hackery

```
PhantomData<MutexGuard<'_, ()>> : !Send
PhantomData<Cell<()>>           :          !Sync
PhantomData<*mut ()>            : !Send + !Sync
```

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
- Used for trait resolution

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
- Used for trait resolution

```
impl<E> From<E> for Box<dyn Error> where E: Error {}
```

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
- Used for trait resolution

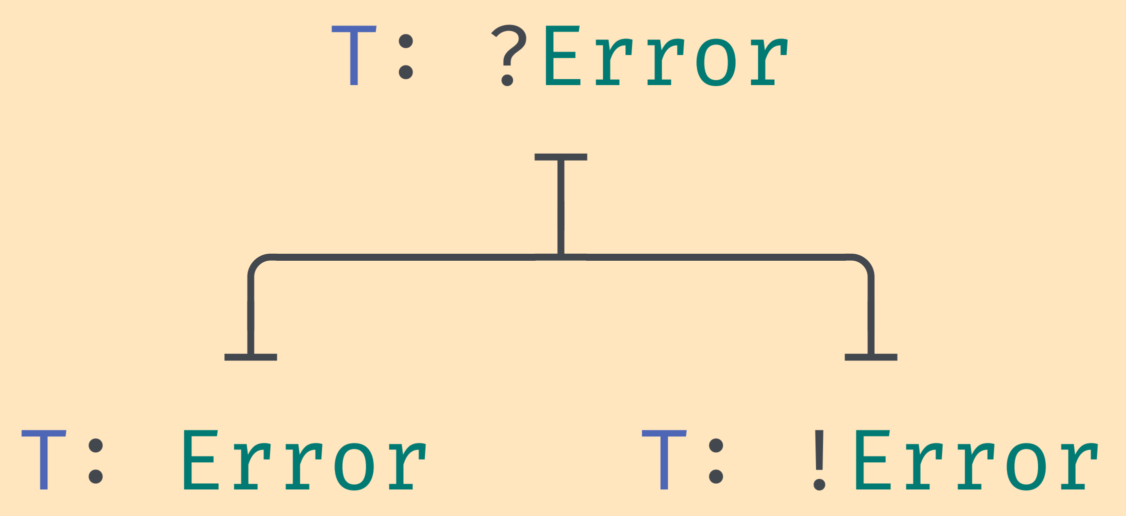
```
impl<E> From<E> for Box<dyn Error> where E: Error {}
impl From<&str> for Box<dyn Error> {}
```

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
- Used for trait resolution

```
impl<E> From<E> for Box<dyn Error> where E: Error {}
impl From<&str> for Box<dyn Error> {}
```

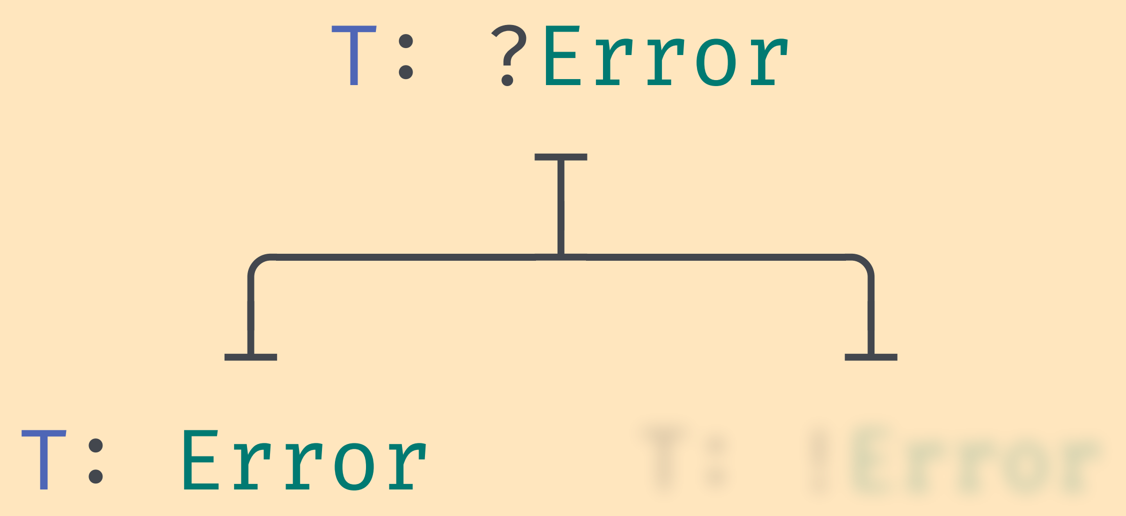


Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
- Used for trait resolution

```
impl<E> From<E> for Box<dyn Error> where E: Error {}
impl From<&str> for Box<dyn Error> {}
```



Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
- Used for trait resolution

```rust
impl<E> From<E> for Box<dyn Error> where E: Error {}
impl From<&str> for Box<dyn Error> {}
impl !Error for &str {}
```

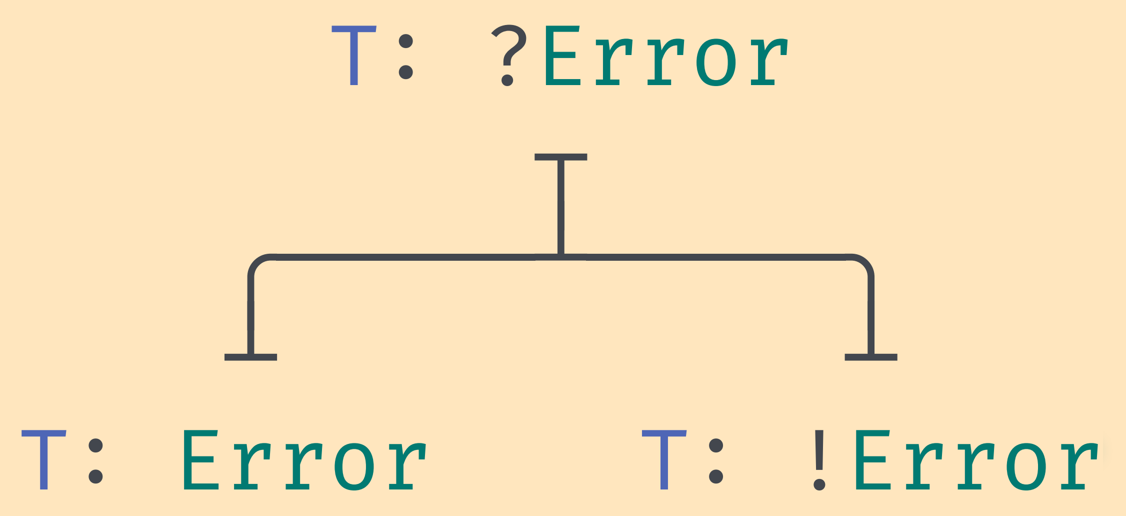
$$\begin{tikzcd} & \texttt{T: ?Error} \arrow[dl, no head] \arrow[dr, no head] & \\ \texttt{T: Error} & & \texttt{T: !Error} \end{tikzcd}$$

Negative implementations

Promise to never implement a trait

- Allow opting out of auto traits
- Used for trait resolution

```
impl<E> From<E> for Box<dyn Error> where E: Error {}
impl From<&str> for Box<dyn Error> {}
impl !Error for &str {}
```



Specialization

Specialization

- Allows otherwise overlapping `impls`

Specialization

- Allows otherwise overlapping `impls`

```rust
impl<T> Default for [T; 0] {}
impl<T> Default for [T; 1]
    where T: Default {}
```

Specialization

- Allows otherwise overlapping impls

```
impl<T> Default for [T; 0] {}
default impl<T, const N: usize> Default for [T; N]
    where T: Default {}
```

Specialization

- Allows otherwise overlapping `impls`

```
impl<T> Default for [T; 0] {}
default impl<T, const N: usize> Default for [T; N]
    where T: Default {}
```

- Current implementation is unsound.

Specialization

- Allows otherwise overlapping `impls`

```
impl<T> Default for [T; 0] {}
default impl<T, const N: usize> Default for [T; N]
    where T: Default {}
```

- Current implementation is unsound.
- Largely stagnated.

Specialization

- Allows otherwise overlapping `impls`

```
impl<T> Default for [T; 0] {}
default impl<T, const N: usize> Default for [T; N]
    where T: Default {}
```

- Current implementation is unsound.
- Largely stagnated.
- Used for optimizations; not present in public API

Specialization

- Allows otherwise overlapping `impls`

```
impl<T> Default for [T; 0] {}
default impl<T, const N: usize> Default for [T; N]
    where T: Default {}
```

- Current implementation is unsound.
- Largely stagnated.
- Used for optimizations; not present in public API
- "Relatively small extension to the trait system"

Crate preludes

Crate preludes

- `core::prelude` and `std::prelude` are implicit by default.

Crate preludes

- `core::prelude` and `std::prelude` are implicit by default.
  - `alloc::prelude` used to exist on nightly.

Crate preludes

- `core::prelude` and `std::prelude` are implicit by default.
  - `alloc::prelude` used to exist on nightly.
- Why not let every crate declare a prelude?

Crate preludes

- `core::prelude` and `std::prelude` are implicit by default.
  - `alloc::prelude` used to exist on nightly.
- Why not let every crate declare a prelude?

```rust
#[prelude(as _)]
pub trait Itertools: Iterator { /* ... */ }
```

Crate preludes

- `core::prelude` and `std::prelude` are implicit by default.
  - `alloc::prelude` used to exist on nightly.
- Why not let every crate declare a prelude?

  ```
  #[prelude(as _)]
  pub trait Itertools: Iterator { /* ... */ }
  ```

- Opt-out as necessary.

Crate preludes

- `core::prelude` and `std::prelude` are implicit by default.
  - `alloc::prelude` used to exist on nightly.
- Why not let every crate declare a prelude?

```
#[prelude(as _)]
pub trait Itertools: Iterator { /* ... */ }
```

- Opt-out as necessary.

```
#[no_prelude(itertools)]
mod foo { /* ... */ }
```

Crate preludes

- `core::prelude` and `std::prelude` are implicit by default.
  - `alloc::prelude` used to exist on nightly.
- Why not let every crate declare a prelude?

```
#[prelude(as _)]
pub trait Itertools: Iterator { /* ... */ }
```

- Opt-out as necessary.

```
#[no_prelude(itertools)]
mod foo { /* ... */ }
```

- The end user chooses which crate's preludes to use.

Crate preludes

- `core::prelude` and `std::prelude` are implicit by default.
  - `alloc::prelude` used to exist on nightly.
- Why not let every crate declare a prelude?

```
#[prelude(as _)]
pub trait Itertools: Iterator { /* ... */ }
```

- Opt-out as necessary.

```
#[no_prelude(itertools)]
mod foo { /* ... */ }
```

- The end user chooses which crate's preludes to use.

```
[dependencies]
itertools = { prelude = true }
```

Stability attributes

Stability attributes

- Used to indicate if an item is stable or not

Stability attributes

- Used to indicate if an item is stable or not

```rust
#[stable(feature = "once_cell", since = "1.70.0")]
pub struct OnceCell<T> { /* ... */ }
```

Stability attributes

- Used to indicate if an item is stable or not

```rust
#[stable(feature = "once_cell", since = "1.70.0")]
pub struct OnceCell<T> { /* ... */ }

#[unstable(feature = "lazy_cell", issue = "109736")]
pub struct LazyLock<T, F = fn() -> T> { /* ... */ }
```

Stability attributes

- Used to indicate if an item is stable or not

```rust
#[stable(feature = "once_cell", since = "1.70.0")]
pub struct OnceCell<T> { /* ... */ }

#[unstable(feature = "lazy_cell", issue = "109736")]
pub struct LazyLock<T, F = fn() -> T> { /* ... */ }
```

- All `pub` items require a stability attribute.

Stability attributes

- Used to indicate if an item is stable or not

```rust
#[stable(feature = "once_cell", since = "1.70.0")]
pub struct OnceCell<T> { /* ... */ }

#[unstable(feature = "lazy_cell", issue = "109736")]
pub struct LazyLock<T, F = fn() -> T> { /* ... */ }
```

- All `pub` items require a stability attribute.
- Unstable items are opt-in and require a feature gate.

Stability attributes

- Used to indicate if an item is stable or not

```
#[stable(feature = "once_cell", since = "1.70.0")]
pub struct OnceCell<T> { /* ... */ }

#[unstable(feature = "lazy_cell", issue = "109736")]
pub struct LazyLock<T, F = fn() -> T> { /* ... */ }
```

- All `pub` items require a stability attribute.
- Unstable items are opt-in and require a feature gate.

```
let _ = LazyLock::new(|| {});
```

Stability attributes

- Used to indicate if an item is stable or not

```
#[stable(feature = "once_cell", since = "1.70.0")]
pub struct OnceCell<T> { /* ... */ }

#[unstable(feature = "lazy_cell", issue = "109736")]
pub struct LazyLock<T, F = fn() -> T> { /* ... */ }
```

- All `pub` items require a stability attribute.
- Unstable items are opt-in and require a feature gate.

```
#![feature(std::lazy_cell)]
let _ = LazyLock::new(|| {});
```

Stability attributes

- Used to indicate if an item is stable or not

```
#[stable(feature = "once_cell", since = "1.70.0")]
pub struct OnceCell<T> { /* ... */ }

#[unstable(feature = "lazy_cell", issue = "109736")]
pub struct LazyLock<T, F = fn() -> T> { /* ... */ }
```

- All `pub` items require a stability attribute.
- Unstable items are opt-in and require a feature gate.

```
#![feature(std::lazy_cell)]
let _ = LazyLock::new(|| {});
```

- "Library authors can continue to use stability attributes"

Progress

Progress

- Integration with Cargo

Progress

- Integration with Cargo
- Reducing number of lang items & intrinsics

Progress

- Integration with Cargo
- Reducing number of lang items & intrinsics
- Suggestions on deprecated items

Progress

- Integration with Cargo
- Reducing number of lang items & intrinsics
- Suggestions on deprecated items
- Unspecified behavior

Progress

- Integration with Cargo
- Reducing number of lang items & intrinsics
- Suggestions on deprecated items
- Unspecified behavior
- Negative implementations

Progress

- Integration with Cargo
- Reducing number of lang items & intrinsics
- Suggestions on deprecated items
- Unspecified behavior
- Negative implementations
- Specialization

Progress

- Integration with Cargo
- Reducing number of lang items & intrinsics
- Suggestions on deprecated items
- Unspecified behavior
- Negative implementations
- Specialization
- Crate preludes

Progress

- Integration with Cargo
- Reducing number of lang items & intrinsics
- Suggestions on deprecated items
- Unspecified behavior
- Negative implementations
- Specialization
- Crate preludes
- Stability attributes

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

Jacob Pratt

@jhpratt

@jhpratt@mastodon.social

jacob@jhpratt.dev

jhpratt.dev

sponsor.jhpratt.dev

2023-09-13
Albuquerque