Rust API Guidelines

This is a set of recommendations on how to design and present APIs for the Rust programming language. They are authored largely by the Rust library team, based on experiences building the Rust standard library and other crates in the Rust ecosystem.

These are only guidelines, some more firm than others. In some cases they are vague and still in development. Rust crate authors should consider them as a set of important considerations in the development of idiomatic and interoperable Rust libraries, to use as they see fit. These guidelines should not in any way be considered a mandate that crate authors must follow, though they may find that crates that conform well to these guidelines integrate better with the existing crate ecosystem than those that do not.

This book is organized in two parts: the concise checklist of all individual guidelines, suitable for quick scanning during crate reviews; and topical chapters containing explanations of the guidelines in detail.

If you are interested in contributing to the API guidelines, check out contributing.md and join our Gitter channel.

Rust API Guidelines Checklist

- **Naming** *(crate aligns with Rust naming conventions)*
    - [ ] Casing conforms to RFC 430 (C-CASE)
    - [ ] Ad-hoc conversions follow `as_`, `to_`, `into_` conventions (C-CONV)
    - [ ] Getter names follow Rust convention (C-GETTER)
    - [ ] Methods on collections that produce iterators follow `iter`, `iter_mut`, `into_iter` (C-ITER)
    - [ ] Iterator type names match the methods that produce them (C-ITER-TY)
    - [ ] Feature names are free of placeholder words (C-FEATURE)
    - [ ] Names use a consistent word order (C-WORD-ORDER)
- **Interoperability** *(crate interacts nicely with other library functionality)*
    - [ ] Types eagerly implement common traits (C-COMMON-TRAITS)
        - `Copy`, `Clone`, `Eq`, `PartialEq`, `Ord`, `PartialOrd`, `Hash`, `Debug`, `Display`, `Default`
    - [ ] Conversions use the standard traits `From`, `AsRef`, `AsMut` (C-CONV-TRAITS)
    - [ ] Collections implement `FromIterator` and `Extend` (C-COLLECT)
    - [ ] Data structures implement Serde's `Serialize`, `Deserialize` (C-SERDE)
    - [ ] Types are `Send` and `Sync` where possible (C-SEND-SYNC)
    - [ ] Error types are meaningful and well-behaved (C-GOOD-ERR)
    - [ ] Binary number types provide `Hex`, `Octal`, `Binary` formatting (C-NUM-FMT)
    - [ ] Generic reader/writer functions take `R: Read` and `W: Write` by value (C-RW-VALUE)
- **Macros** *(crate presents well-behaved macros)*
    - [ ] Input syntax is evocative of the output (C-EVOCATIVE)
    - [ ] Macros compose well with attributes (C-MACRO-ATTR)
    - [ ] Item macros work anywhere that items are allowed (C-ANYWHERE)
    - [ ] Item macros support visibility specifiers (C-MACRO-VIS)
    - [ ] Type fragments are flexible (C-MACRO-TY)
- **Documentation** *(crate is abundantly documented)*
    - [ ] Crate level docs are thorough and include examples (C-CRATE-DOC)
    - [ ] All items have a rustdoc example (C-EXAMPLE)
    - [ ] Examples use `?`, not `try!`, not `unwrap` (C-QUESTION-MARK)
    - [ ] Function docs include error, panic, and safety considerations (C-FAILURE)
    - [ ] Prose contains hyperlinks to relevant things (C-LINK)
    - [ ] Cargo.toml includes all common metadata (C-METADATA)

        - authors, description, license, homepage, documentation, repository, keywords, categories
    - ☐ Release notes document all significant changes (C-RELNOTES)
    - ☐ Rustdoc does not show unhelpful implementation details (C-HIDDEN)
- **Predictability** *(crate enables legible code that acts how it looks)*
    - ☐ Smart pointers do not add inherent methods (C-SMART-PTR)
    - ☐ Conversions live on the most specific type involved (C-CONV-SPECIFIC)
    - ☐ Functions with a clear receiver are methods (C-METHOD)
    - ☐ Functions do not take out-parameters (C-NO-OUT)
    - ☐ Operator overloads are unsurprising (C-OVERLOAD)
    - ☐ Only smart pointers implement `Deref` and `DerefMut` (C-DEREF)
    - ☐ Constructors are static, inherent methods (C-CTOR)
- **Flexibility** *(crate supports diverse real-world use cases)*
    - ☐ Functions expose intermediate results to avoid duplicate work (C-INTERMEDIATE)
    - ☐ Caller decides where to copy and place data (C-CALLER-CONTROL)
    - ☐ Functions minimize assumptions about parameters by using generics (C-GENERIC)
    - ☐ Traits are object-safe if they may be useful as a trait object (C-OBJECT)
- **Type safety** *(crate leverages the type system effectively)*
    - ☐ Newtypes provide static distinctions (C-NEWTYPE)
    - ☐ Arguments convey meaning through types, not `bool` or `Option` (C-CUSTOM-TYPE)
    - ☐ Types for a set of flags are `bitflags`, not enums (C-BITFLAG)
    - ☐ Builders enable construction of complex values (C-BUILDER)
- **Dependability** *(crate is unlikely to do the wrong thing)*
    - ☐ Functions validate their arguments (C-VALIDATE)
    - ☐ Destructors never fail (C-DTOR-FAIL)
    - ☐ Destructors that may block have alternatives (C-DTOR-BLOCK)
- **Debuggability** *(crate is conducive to easy debugging)*
    - ☐ All public types implement `Debug` (C-DEBUG)
    - ☐ `Debug` representation is never empty (C-DEBUG-NONEMPTY)
- **Future proofing** *(crate is free to improve without breaking users' code)*
    - ☐ Sealed traits protect against downstream implementations (C-SEALED)
    - ☐ Structs have private fields (C-STRUCT-PRIVATE)
    - ☐ Newtypes encapsulate implementation details (C-NEWTYPE-HIDE)
    - ☐ Data structures do not duplicate derived trait bounds (C-STRUCT-BOUNDS)
- **Necessities** *(to whom they matter, they really matter)*
    - ☐ Public dependencies of a stable crate are stable (C-STABLE)
    - ☐ Crate and its dependencies have a permissive license (C-PERMISSIVE)

Naming

Casing conforms to RFC 430 (C-CASE)

Basic Rust naming conventions are described in RFC 430.

In general, Rust tends to use `UpperCamelCase` for "type-level" constructs (types and traits) and `snake_case` for "value-level" constructs. More precisely:

Item	Convention
Crates	unclear
Modules	`snake_case`
Types	`UpperCamelCase`
Traits	`UpperCamelCase`

Item	Convention
Enum variants	`UpperCamelCase`
Functions	`snake_case`
Methods	`snake_case`
General constructors	`new` or `with_more_details`
Conversion constructors	`from_some_other_type`
Macros	`snake_case!`
Local variables	`snake_case`
Statics	`SCREAMING_SNAKE_CASE`
Constants	`SCREAMING_SNAKE_CASE`
Type parameters	concise `UpperCamelCase`, usually single uppercase letter: `T`
Lifetimes	short `lowercase`, usually a single letter: `'a`, `'de`, `'src`
Features	unclear but see C-FEATURE

In `UpperCamelCase`, acronyms and contractions of compound words count as one word: use `Uuid` rather than `UUID`, `Usize` rather than `USize` or `Stdin` rather than `StdIn`. In `snake_case`, acronyms and contractions are lower-cased: `is_xid_start`.

In `snake_case` or `SCREAMING_SNAKE_CASE`, a "word" should never consist of a single letter unless it is the last "word". So, we have `btree_map` rather than `b_tree_map`, but `PI_2` rather than `PI2`.

Crate names should not use `-rs` or `-rust` as a suffix or prefix. Every crate is Rust! It serves no purpose to remind users of this constantly.

Examples from the standard library

The whole standard library. This guideline should be easy!

Ad-hoc conversions follow `as_`, `to_`, `into_` conventions (C-CONV)

Conversions should be provided as methods, with names prefixed as follows:

Prefix	Cost	Ownership
`as_`	Free	borrowed -> borrowed
`to_`	Expensive	borrowed -> borrowed borrowed -> owned (non-Copy types) owned -> owned (Copy types)
`into_`	Variable	owned -> owned (non-Copy types)

For example:

- `str::as_bytes()` gives a view of a `str` as a slice of UTF-8 bytes, which is free. The input is a borrowed `&str` and the output is a borrowed `&[u8]`.
- `Path::to_str` performs an expensive UTF-8 check on the bytes of an operating system path. The input and output are both borrowed. It would not be correct to call this `as_str` because this method has nontrivial cost at runtime.
- `str::to_lowercase()` produces the Unicode-correct lowercase equivalent of a `str`, which involves iterating through characters of the string and may require memory allocation. The input is a borrowed `&str` and the output is an owned `String`.

- `f64::to_radians()` converts a floating point quantity from degrees to radians. The input is `f64`. Passing a reference `&f64` is not warranted because `f64` is cheap to copy. Calling the function `into_radians` would be misleading because the input is not consumed.
- `String::into_bytes()` extracts the underlying `Vec<u8>` of a `String`, which is free. It takes ownership of a `String` and returns an owned `Vec<u8>`.
- `BufReader::into_inner()` takes ownership of a buffered reader and extracts out the underlying reader, which is free. Data in the buffer is discarded.
- `BufWriter::into_inner()` takes ownership of a buffered writer and extracts out the underlying writer, which requires a potentially expensive flush of any buffered data.

Conversions prefixed `as_` and `into_` typically *decrease abstraction*, either exposing a view into the underlying representation (`as`) or deconstructing data into its underlying representation (`into`). Conversions prefixed `to_`, on the other hand, typically stay at the same level of abstraction but do some work to change from one representation to another.

When a type wraps a single value to associate it with higher-level semantics, access to the wrapped value should be provided by an `into_inner()` method. This applies to wrappers that provide buffering like `BufReader`, encoding or decoding like `GzDecoder`, atomic access like `AtomicBool`, or any similar semantics.

If the `mut` qualifier in the name of a conversion method constitutes part of the return type, it should appear as it would appear in the type. For example `Vec::as_mut_slice` returns a mut slice; it does what it says. This name is preferred over `as_slice_mut`.

```
// Return type is a mut slice.
fn as_mut_slice(&mut self) -> &mut [T];
```

More examples from the standard library

- `Result::as_ref`
- `RefCell::as_ptr`
- `slice::to_vec`
- `Option::into_iter`

Getter names follow Rust convention (C-GETTER)

With a few exceptions, the `get_` prefix is not used for getters in Rust code.

```
pub struct S {
    first: First,
    second: Second,
}

impl S {
    // Not get_first.
    pub fn first(&self) -> &First {
        &self.first
    }

    // Not get_first_mut, get_mut_first, or mut_first.
    pub fn first_mut(&mut self) -> &mut First {
        &mut self.first
    }
}
```

The `get` naming is used only when there is a single and obvious thing that could reasonably be gotten by a getter. For example `Cell::get` accesses the content of a `Cell`.

For getters that do runtime validation such as bounds checking, consider adding unsafe `_unchecked` variants. Typically those will have the following signatures.

```
fn get(&self, index: K) -> Option<&V>;
fn get_mut(&mut self, index: K) -> Option<&mut V>;
unsafe fn get_unchecked(&self, index: K) -> &V;
unsafe fn get_unchecked_mut(&mut self, index: K) -> &mut V;
```

The difference between getters and conversions (C-CONV) can be subtle and is not always clear-cut. For example `TempDir::path` can be understood as a getter for the filesystem path of the temporary directory, while `TempDir::into_path` is a conversion that transfers responsibility for deleting the temporary directory to the caller. Since `path` is a getter, it would not be correct to call it `get_path` or `as_path`.

Examples from the standard library

- `std::io::Cursor::get_mut`
- `std::pin::Pin::get_mut`
- `std::sync::PoisonError::get_mut`
- `std::sync::atomic::AtomicBool::get_mut`
- `std::collections::hash_map::OccupiedEntry::get_mut`
- `<[T]>::get_unchecked`

Methods on collections that produce iterators follow `iter`, `iter_mut`, `into_iter` (C-ITER)

Per RFC 199.

For a container with elements of type `U`, iterator methods should be named:

```
fn iter(&self) -> Iter             // Iter implements Iterator<Item = &U>
fn iter_mut(&mut self) -> IterMut  // IterMut implements Iterator<Item = &mut U>
fn into_iter(self) -> IntoIter     // IntoIter implements Iterator<Item = U>
```

This guideline applies to data structures that are conceptually homogeneous collections. As a counterexample, the `str` type is slice of bytes that are guaranteed to be valid UTF-8. This is conceptually more nuanced than a homogeneous collection so rather than providing the `iter`/`iter_mut`/`into_iter` group of iterator methods, it provides `str::bytes` to iterate as bytes and `str::chars` to iterate as chars.

This guideline applies to methods only, not functions. For example `percent_encode` from the `url` crate returns an iterator over percent-encoded string fragments. There would be no clarity to be had by using an `iter`/`iter_mut`/`into_iter` convention.

Examples from the standard library

- `Vec::iter`
- `Vec::iter_mut`
- `Vec::into_iter`

- `BTreeMap::iter`
- `BTreeMap::iter_mut`

Iterator type names match the methods that produce them (C-ITER-TY)

A method called `into_iter()` should return a type called `IntoIter` and similarly for all other methods that return iterators.

This guideline applies chiefly to methods, but often makes sense for functions as well. For example the `percent_encode` function from the `url` crate returns an iterator type called `PercentEncode`.

These type names make the most sense when prefixed with their owning module, for example `vec::IntoIter`.

Examples from the standard library

- `Vec::iter` returns `Iter`
- `Vec::iter_mut` returns `IterMut`
- `Vec::into_iter` returns `IntoIter`
- `BTreeMap::keys` returns `Keys`
- `BTreeMap::values` returns `Values`

Feature names are free of placeholder words (C-FEATURE)

Do not include words in the name of a Cargo feature that convey zero meaning, as in `use-abc` or `with-abc`. Name the feature `abc` directly.

This arises most commonly for crates that have an optional dependency on the Rust standard library. The canonical way to do this correctly is:

```toml
# In Cargo.toml

[features]
default = ["std"]
std = []
```

```rust
// In lib.rs
#![no_std]

#[cfg(feature = "std")]
extern crate std;
```

Do not call the feature `use-std` or `with-std` or any creative name that is not `std`. This naming convention aligns with the naming of implicit features inferred by Cargo for optional dependencies. Consider crate `x` with optional dependencies on Serde and on the Rust standard library:

```toml
[package]
name = "x"
version = "0.1.0"

[features]
std = ["serde/std"]

[dependencies]
serde = { version = "1.0", optional = true }
```

When we depend on `x`, we can enable the optional Serde dependency with `features = ["serde"]`. Similarly we can enable the optional standard library dependency with `features = ["std"]`. The implicit feature inferred by Cargo for the optional dependency is called `serde`, not `use-serde` or `with-serde`, so we like for explicit features to behave the same way.

As a related note, Cargo requires that features are additive so a feature named negatively like `no-abc` is practically never correct.

Names use a consistent word order (C-WORD-ORDER)

Here are some error types from the standard library:

- `JoinPathsError`
- `ParseBoolError`
- `ParseCharError`
- `ParseFloatError`
- `ParseIntError`
- `RecvTimeoutError`
- `StripPrefixError`

All of these use verb-object-error word order. If we were adding an error to represent an address failing to parse, for consistency we would want to name it in verb-object-error order like `ParseAddrError` rather than `AddrParseError`.

The particular choice of word order is not important, but pay attention to consistency within the crate and consistency with similar functionality in the standard library.

Interoperability

Types eagerly implement common traits (C-COMMON-TRAITS)

Rust's trait system does not allow *orphans*: roughly, every `impl` must live either in the crate that defines the trait or the implementing type. Consequently, crates that define new types should eagerly implement all applicable, common traits.

To see why, consider the following situation:

- Crate `std` defines trait `Display`.
- Crate `url` defines type `Url`, without implementing `Display`.
- Crate `webapp` imports from both `std` and `url`,

There is no way for `webapp` to add `Display` to `Url`, since it defines neither. (Note: the newtype pattern can provide an efficient, but inconvenient workaround.)

The most important common traits to implement from `std` are:

- `Copy`
- `Clone`
- `Eq`
- `PartialEq`
- `Ord`
- `PartialOrd`
- `Hash`
- `Debug`
- `Display`
- `Default`

Note that it is common and expected for types to implement both `Default` and an empty `new` constructor. `new` is the constructor convention in Rust, and users expect it to exist, so if it is reasonable for the basic constructor to take no arguments, then it should, even if it is functionally identical to `default`.

Conversions use the standard traits `From`, `AsRef`, `AsMut` (C-CONV-TRAITS)

The following conversion traits should be implemented where it makes sense:

- `From`
- `TryFrom`
- `AsRef`
- `AsMut`

The following conversion traits should never be implemented:

- `Into`
- `TryInto`

These traits have a blanket impl based on `From` and `TryFrom`. Implement those instead.

Examples from the standard library

- `From<u16>` is implemented for `u32` because a smaller integer can always be converted to a bigger integer.
- `From<u32>` is *not* implemented for `u16` because the conversion may not be possible if the integer is too big.
- `TryFrom<u32>` is implemented for `u16` and returns an error if the integer is too big to fit in `u16`.
- `From<Ipv6Addr>` is implemented for `IpAddr`, which is a type that can represent both v4 and v6 IP addresses.

Collections implement `FromIterator` and `Extend` (C-COLLECT)

`FromIterator` and `Extend` enable collections to be used conveniently with the following iterator methods:

- `Iterator::collect`
- `Iterator::partition`

- `Iterator::unzip`

`FromIterator` is for creating a new collection containing items from an iterator, and `Extend` is for adding items from an iterator onto an existing collection.

Examples from the standard library

- `Vec<T>` implements both `FromIterator<T>` and `Extend<T>`.

Data structures implement Serde's `Serialize`, `Deserialize` (C-SERDE)

Types that play the role of a data structure should implement `Serialize` and `Deserialize`.

There is a continuum of types between things that are clearly a data structure and things that are clearly not, with gray area in between. `LinkedHashMap` and `IpAddr` are data structures. It would be completely reasonable for somebody to want to read in a `LinkedHashMap` or `IpAddr` from a JSON file, or send one over IPC to another process. `LittleEndian` is not a data structure. It is a marker used by the `byteorder` crate to optimize at compile time for bytes in a particular order, and in fact an instance of `LittleEndian` can never exist at runtime. So these are clear-cut examples; the #rust or #serde IRC channels can help assess more ambiguous cases if necessary.

If a crate does not already depend on Serde for other reasons, it may wish to gate Serde impls behind a Cargo cfg. This way downstream libraries only need to pay the cost of compiling Serde if they need those impls to exist.

For consistency with other Serde-based libraries, the name of the Cargo cfg should be simply `"serde"`. Do not use a different name for the cfg like `"serde_impls"` or `"serde_serialization"`.

The canonical implementation looks like this when not using derive:

```toml
[dependencies]
serde = { version = "1.0", optional = true }
```

```rust

pub struct T { /* ... */ }

#[cfg(feature = "serde")]
impl Serialize for T { /* ... */ }

#[cfg(feature = "serde")]
impl<'de> Deserialize<'de> for T { /* ... */ }
```

And when using derive:

```toml
[dependencies]
serde = { version = "1.0", optional = true, features = ["derive"] }
```

```rust

#[cfg_attr(feature = "serde", derive(Serialize, Deserialize))]
pub struct T { /* ... */ }
```

Types are `Send` and `Sync` where possible (C-SEND-SYNC)

`Send` and `Sync` are automatically implemented when the compiler determines it is appropriate.

In types that manipulate raw pointers, be vigilant that the `Send` and `Sync` status of your type accurately reflects its thread safety characteristics. Tests like the following can help catch unintentional regressions in whether the type implements `Send` or `Sync`.

```
#[test]
fn test_send() {
    fn assert_send<T: Send>() {}
    assert_send::<MyStrangeType>();
}

#[test]
fn test_sync() {
    fn assert_sync<T: Sync>() {}
    assert_sync::<MyStrangeType>();
}
```

Error types are meaningful and well-behaved (C-GOOD-ERR)

An error type is any type `E` used in a `Result<T, E>` returned by any public function of your crate. Error types should always implement the `std::error::Error` trait which is the mechanism by which error handling libraries like `error-chain` abstract over different types of errors, and which allows the error to be used as the `source()` of another error.

Additionally, error types should implement the `Send` and `Sync` traits. An error that is not `Send` cannot be returned by a thread run with `thread::spawn`. An error that is not `Sync` cannot be passed across threads using an `Arc`. These are common requirements for basic error handling in a multithreaded application.

`Send` and `Sync` are also important for being able to package a custom error into an IO error using `std::io::Error::new`, which requires a trait bound of `Error + Send + Sync`.

One place to be vigilant about this guideline is in functions that return Error trait objects, for example `reqwest::Error::get_ref`. Typically `Error + Send + Sync + 'static` will be the most useful for callers. The addition of `'static` allows the trait object to be used with `Error::downcast_ref`.

Never use `()` as an error type, even where there is no useful additional information for the error to carry.

- `()` does not implement `Error` so it cannot be used with error handling libraries like `error-chain`.
- `()` does not implement `Display` so a user would need to write an error message of their own if they want to fail because of the error.
- `()` has an unhelpful `Debug` representation for users that decide to `unwrap()` the error.
- It would not be semantically meaningful for a downstream library to implement `From<()>` for their error type, so `()` as an error type cannot be used with the `?` operator.

Instead, define a meaningful error type specific to your crate or to the individual function. Provide appropriate `Error` and `Display` impls. If there is no useful information for the error to carry, it can be implemented as a unit struct.

```
use std::error::Error;
use std::fmt::Display;

// Instead of this...
fn do_the_thing() -> Result<Wow, ()>

// Prefer this...
fn do_the_thing() -> Result<Wow, DoError>

#[derive(Debug)]
struct DoError;

impl Display for DoError { /* ... */ }
impl Error for DoError { /* ... */ }
```

The error message given by the `Display` representation of an error type should be lowercase without trailing punctuation, and typically concise.

`Error::description()` should not be implemented. It has been deprecated and users should always use `Display` instead of `description()` to print the error.

Examples from the standard library

- `ParseBoolError` is returned when failing to parse a bool from a string.

Examples of error messages

- "unexpected end of file"
- "provided string was not `true` or `false`"
- "invalid IP address syntax"
- "second time provided was later than self"
- "invalid UTF-8 sequence of {} bytes from index {}"
- "environment variable was not valid unicode: {:?}"

Binary number types provide `Hex`, `Octal`, `Binary` formatting (C-NUM-FMT)

- `std::fmt::UpperHex`
- `std::fmt::LowerHex`
- `std::fmt::Octal`
- `std::fmt::Binary`

These traits control the representation of a type under the `{:X}`, `{:x}`, `{:o}`, and `{:b}` format specifiers.

Implement these traits for any number type on which you would consider doing bitwise manipulations like `|` or `&`. This is especially appropriate for bitflag types. Numeric quantity types like `struct Nanoseconds(u64)` probably do not need these.

Generic reader/writer functions take `R: Read` and `W: Write` by value (C-RW-VALUE)

The standard library contains these two impls:

```
impl<'a, R: Read + ?Sized> Read for &'a mut R { /* ... */ }

impl<'a, W: Write + ?Sized> Write for &'a mut W { /* ... */ }
```

That means any function that accepts `R: Read` or `W: Write` generic parameters by value can be called with a mut reference if necessary.

In the documentation of such functions, briefly remind users that a mut reference can be passed. New Rust users often struggle with this. They may have opened a file and want to read multiple pieces of data out of it, but the function to read one piece consumes the reader by value, so they are stuck. The solution would be to leverage one of the above impls and pass `&mut f` instead of `f` as the reader parameter.

Examples

- `flate2::read::GzDecoder::new`
- `flate2::write::GzEncoder::new`
- `serde_json::from_reader`
- `serde_json::to_writer`

Macros

Input syntax is evocative of the output (C-EVOCATIVE)

Rust macros let you dream up practically whatever input syntax you want. Aim to keep input syntax familiar and cohesive with the rest of your users' code by mirroring existing Rust syntax where possible. Pay attention to the choice and placement of keywords and punctuation.

A good guide is to use syntax, especially keywords and punctuation, that is similar to what will be produced in the output of the macro.

For example if your macro declares a struct with a particular name given in the input, preface the name with the keyword `struct` to signal to readers that a struct is being declared with the given name.

```
// Prefer this...
bitflags! {
    struct S: u32 { /* ... */ }
}

// ...over no keyword...
bitflags! {
    S: u32 { /* ... */ }
}

// ...or some ad-hoc word.
bitflags! {
    flags S: u32 { /* ... */ }
}
```

Another example is semicolons vs commas. Constants in Rust are followed by semicolons so if your macro declares a chain of constants, they should likely be followed by semicolons even if the syntax is otherwise slightly different from Rust's.

```rust
// Ordinary constants use semicolons.
const A: u32 = 0b000001;
const B: u32 = 0b000010;

// So prefer this...
bitflags! {
    struct S: u32 {
        const C = 0b000100;
        const D = 0b001000;
    }
}

// ...over this.
bitflags! {
    struct S: u32 {
        const E = 0b010000,
        const F = 0b100000,
    }
}
```

Macros are so diverse that these specific examples won't be relevant, but think about how to apply the same principles to your situation.

Item macros compose well with attributes (C-MACRO-ATTR)

Macros that produce more than one output item should support adding attributes to any one of those items. One common use case would be putting individual items behind a cfg.

```rust
bitflags! {
    struct Flags: u8 {
        #[cfg(windows)]
        const ControlCenter = 0b001;
        #[cfg(unix)]
        const Terminal = 0b010;
    }
}
```

Macros that produce a struct or enum as output should support attributes so that the output can be used with derive.

```rust
bitflags! {
    #[derive(Default, Serialize)]
    struct Flags: u8 {
        const ControlCenter = 0b001;
        const Terminal = 0b010;
    }
}
```

Item macros work anywhere that items are allowed (C-ANYWHERE)

Rust allows items to be placed at the module level or within a tighter scope like a function. Item macros should work equally well as ordinary items in all of these places. The test suite should include invocations of the macro in at least the module scope and function scope.

```rust
#[cfg(test)]
mod tests {
    test_your_macro_in_a!(module);

    #[test]
    fn anywhere() {
        test_your_macro_in_a!(function);
    }
}
```

As a simple example of how things can go wrong, this macro works great in a module scope but fails in a function scope.

```rust
macro_rules! broken {
    ($m:ident :: $t:ident) => {
        pub struct $t;
        pub mod $m {
            pub use super::$t;
        }
    }
}

broken!(m::T); // okay, expands to T and m::T

fn g() {
    broken!(m::U); // fails to compile, super::U refers to the containing module not g
}
```

Item macros support visibility specifiers (C-MACRO-VIS)

Follow Rust syntax for visibility of items produced by a macro. Private by default, public if `pub` is specified.

```rust
bitflags! {
    struct PrivateFlags: u8 {
        const A = 0b0001;
        const B = 0b0010;
    }
}

bitflags! {
    pub struct PublicFlags: u8 {
        const C = 0b0100;
        const D = 0b1000;
    }
}
```

Type fragments are flexible (C-MACRO-TY)

If your macro accepts a type fragment like `$t:ty` in the input, it should be usable with all of the following:

- Primitives: `u8`, `&str`
- Relative paths: `m::Data`
- Absolute paths: `::base::Data`

- Upward relative paths: `super::Data`
- Generics: `Vec<String>`

As a simple example of how things can go wrong, this macro works great with primitives and absolute paths but fails with relative paths.

```rust
macro_rules! broken {
    ($m:ident => $t:ty) => {
        pub mod $m {
            pub struct Wrapper($t);
        }
    }
}

broken!(a => u8); // okay

broken!(b => ::std::marker::PhantomData<()>); // okay

struct S;
broken!(c => S); // fails to compile
```

Documentation

Crate level docs are thorough and include examples (C-CRATE-DOC)

See RFC 1687.

All items have a rustdoc example (C-EXAMPLE)

Every public module, trait, struct, enum, function, method, macro, and type definition should have an example that exercises the functionality.

This guideline should be applied within reason.

A link to an applicable example on another item may be sufficient. For example if exactly one function uses a particular type, it may be appropriate to write a single example on either the function or the type and link to it from the other.

The purpose of an example is not always to show *how to use* the item. Readers can be expected to understand how to invoke functions, match on enums, and other fundamental tasks. Rather, an example is often intended to show *why someone would want to use* the item.

```rust
// This would be a poor example of using clone(). It mechanically shows *how* to
// call clone(), but does nothing to show *why* somebody would want this.
fn main() {
    let hello = "hello";

    hello.clone();
}
```

Examples use `?`, not `try!`, not `unwrap` (C-QUESTION-MARK)

Like it or not, example code is often copied verbatim by users. Unwrapping an error should be a conscious decision that the user needs to make.

A common way of structuring fallible example code is the following. The lines beginning with `#` are compiled by `cargo test` when building the example but will not appear in user-visible rustdoc.

```
/// ```rust
/// # use std::error::Error;
/// #
/// # fn main() -> Result<(), Box<dyn Error>> {
/// your;
/// example?;
/// code;
/// #
/// #     Ok(())
/// # }
/// ```
```

Function docs include error, panic, and safety considerations (C-FAILURE)

Error conditions should be documented in an "Errors" section. This applies to trait methods as well -- trait methods for which the implementation is allowed or expected to return an error should be documented with an "Errors" section.

For example in the standard library, Some implementations of the `std::io::Read::read` trait method may return an error.

```
/// Pull some bytes from this source into the specified buffer, returning
/// how many bytes were read.
///
/// ... lots more info ...
///
/// # Errors
///
/// If this function encounters any form of I/O or other error, an error
/// variant will be returned. If an error is returned then it must be
/// guaranteed that no bytes were read.
```

Panic conditions should be documented in a "Panics" section. This applies to trait methods as well -- traits methods for which the implementation is allowed or expected to panic should be documented with a "Panics" section.

In the standard library the `Vec::insert` method may panic.

```
/// Inserts an element at position `index` within the vector, shifting all
/// elements after it to the right.
///
/// # Panics
///
/// Panics if `index` is out of bounds.
```

It is not necessary to document all conceivable panic cases, especially if the panic occurs in logic provided by the caller. For example documenting the `Display` panic in the following code seems excessive. But when in doubt, err on the side of documenting more panic cases.

```rust
/// # Panics
///
/// This function panics if `T`'s implementation of `Display` panics.
pub fn print<T: Display>(t: T) {
    println!("{}", t.to_string());
}
```

Unsafe functions should be documented with a "Safety" section that explains all invariants that the caller is responsible for upholding to use the function correctly.

The unsafe `std::ptr::read` requires the following of the caller.

```
/// Reads the value from `src` without moving it. This leaves the
/// memory in `src` unchanged.
///
/// # Safety
///
/// Beyond accepting a raw pointer, this is unsafe because it semantically
/// moves the value out of `src` without preventing further usage of `src`.
/// If `T` is not `Copy`, then care must be taken to ensure that the value at
/// `src` is not used before the data is overwritten again (e.g. with `write`,
/// `zero_memory`, or `copy_memory`). Note that `*src = foo` counts as a use
/// because it will attempt to drop the value previously at `*src`.
///
/// The pointer must be aligned; use `read_unaligned` if that is not the case.
```

Prose contains hyperlinks to relevant things (C-LINK)

Regular links can be added inline with the usual markdown syntax of `[text](url)`. Links to other types can be added by marking them with ``[`text`]``, then adding the link target in a new line at the end of the docstring with ``[`text`]: <target>``, where `<target>` is described below.

Link targets to methods within the same type usually look like this:

```
[`serialize_struct`]: #method.serialize_struct
```

Link targets to other types usually look like this:

```
[`Deserialize`]: trait.Deserialize.html
```

Link targets may also point to a parent or child module:

```
[`Value`]: ../enum.Value.html
[`DeserializeOwned`]: de/trait.DeserializeOwned.html
```

This guideline is officially recommended by RFC 1574 under the heading "Link all the things".

Cargo.toml includes all common metadata (C-METADATA)

The `[package]` section of `Cargo.toml` should include the following values:

- `authors`
- `description`
- `license`
- `repository`
- `keywords`

- `categories`

In addition, there are two optional metadata fields:

- `documentation`
- `homepage`

By default, *crates.io* links to documentation for the crate on *docs.rs*. The `documentation` metadata only needs to be set if the documentation is hosted somewhere other than *docs.rs*, for example because the crate links against a shared library that is not available in the build environment of *docs.rs*.

The `homepage` metadata should only be set if there is a unique website for the crate other than the source repository or API documentation. Do not make `homepage` redundant with either the `documentation` or `repository` values. For example, serde sets `homepage` to *https://serde.rs*, a dedicated website.

Release notes document all significant changes (C-RELNOTES)

Users of the crate can read the release notes to find a summary of what changed in each published release of the crate. A link to the release notes, or the notes themselves, should be included in the crate-level documentation and/or the repository linked in Cargo.toml.

Breaking changes (as defined in RFC 1105) should be clearly identified in the release notes.

If using Git to track the source of a crate, every release published to *crates.io* should have a corresponding tag identifying the commit that was published. A similar process should be used for non-Git VCS tools as well.

```
# Tag the current commit
GIT_COMMITTER_DATE=$(git log -n1 --pretty=%aD) git tag -a -m "Release 0.3.0" 0.3.0
git push --tags
```

Annotated tags are preferred because some Git commands ignore unannotated tags if any annotated tags exist.

Examples

- Serde 1.0.0 release notes
- Serde 0.9.8 release notes
- Serde 0.9.0 release notes
- Diesel change log

Rustdoc does not show unhelpful implementation details (C-HIDDEN)

Rustdoc is supposed to include everything users need to use the crate fully and nothing more. It is fine to explain relevant implementation details in prose but they should not be real entries in the documentation.

Especially be selective about which impls are visible in rustdoc -- all the ones that users would need for using the crate fully, but no others. In the following code the rustdoc of `PublicError` by default would show the `From<PrivateError>` impl. We choose to hide it with `#[doc(hidden)]` because users can never have a `PrivateError` in their code so this impl would never be relevant to them.

```
// This error type is returned to users.
pub struct PublicError { /* ... */ }

// This error type is returned by some private helper functions.
struct PrivateError { /* ... */ }

// Enable use of `?` operator.
#[doc(hidden)]
impl From<PrivateError> for PublicError {
    fn from(err: PrivateError) -> PublicError {
        /* ... */
    }
}
```

`pub(crate)` is another great tool for removing implementation details from the public API. It allows items to be used from outside of their own module but not outside of the same crate.

Predictability

Smart pointers do not add inherent methods (C-SMART-PTR)

For example, this is why the `Box::into_raw` function is defined the way it is.

```
impl<T> Box<T> where T: ?Sized {
    fn into_raw(b: Box<T>) -> *mut T { /* ... */ }
}

let boxed_str: Box<str> = /* ... */;
let ptr = Box::into_raw(boxed_str);
```

If this were defined as an inherent method instead, it would be confusing at the call site whether the method being called is a method on `Box<T>` or a method on `T`.

```
impl<T> Box<T> where T: ?Sized {
    // Do not do this.
    fn into_raw(self) -> *mut T { /* ... */ }
}

let boxed_str: Box<str> = /* ... */;

// This is a method on str accessed through the smart pointer Deref impl.
boxed_str.chars()

// This is a method on Box<str>...?
boxed_str.into_raw()
```

Conversions live on the most specific type involved (C-CONV-SPECIFIC)

When in doubt, prefer `to_` / `as_` / `into_` to `from_`, because they are more ergonomic to use (and can be chained with other methods).

For many conversions between two types, one of the types is clearly more "specific": it provides some additional invariant or interpretation that is not present in the other type. For example, `str` is more specific than `&[u8]`, since it is a UTF-8 encoded sequence of bytes.

Conversions should live with the more specific of the involved types. Thus, `str` provides both the `as_bytes` method and the `from_utf8` constructor for converting to and from `&[u8]` values. Besides being intuitive, this convention avoids polluting concrete types like `&[u8]` with endless conversion methods.

Functions with a clear receiver are methods (C-METHOD)

Prefer

```
impl Foo {
    pub fn frob(&self, w: widget) { /* ... */ }
}
```

over

```
pub fn frob(foo: &Foo, w: widget) { /* ... */ }
```

for any operation that is clearly associated with a particular type.

Methods have numerous advantages over functions:

- They do not need to be imported or qualified to be used: all you need is a value of the appropriate type.
- Their invocation performs autoborrowing (including mutable borrows).
- They make it easy to answer the question "what can I do with a value of type `T`" (especially when using rustdoc).
- They provide `self` notation, which is more concise and often more clearly conveys ownership distinctions.

Functions do not take out-parameters (C-NO-OUT)

Prefer

```
fn foo() -> (Bar, Bar)
```

over

```
fn foo(output: &mut Bar) -> Bar
```

for returning multiple `Bar` values.

Compound return types like tuples and structs are efficiently compiled and do not require heap allocation. If a function needs to return multiple values, it should do so via one of these types.

The primary exception: sometimes a function is meant to modify data that the caller already owns, for example to re-use a buffer:

```
fn read(&mut self, buf: &mut [u8]) -> io::Result<usize>
```

Operator overloads are unsurprising (C-OVERLOAD)

Operators with built in syntax (`*`, `|`, and so on) can be provided for a type by implementing the traits in `std::ops`. These operators come with strong expectations: implement `Mul` only for an operation that bears some resemblance to multiplication (and shares the expected properties, e.g. associativity), and so on for the other traits.

Only smart pointers implement `Deref` and `DerefMut` (C-DEREF)

The `Deref` traits are used implicitly by the compiler in many circumstances, and interact with method resolution. The relevant rules are designed specifically to accommodate smart pointers, and so the traits should be used only for that purpose.

Examples from the standard library

- `Box<T>`
- `String` is a smart pointer to `str`
- `Rc<T>`
- `Arc<T>`
- `Cow<'a, T>`

Constructors are static, inherent methods (C-CTOR)

In Rust, "constructors" are just a convention. There are a variety of conventions around constructor naming, and the distinctions are often subtle.

A constructor in its most basic form is a `new` method with no arguments.

```
impl<T> Example<T> {
    pub fn new() -> Example<T> { /* ... */ }
}
```

Constructors are static (no `self`) inherent methods for the type that they construct. Combined with the practice of fully importing type names, this convention leads to informative but concise construction:

```
use example::Example;

// Construct a new Example.
let ex = Example::new();
```

The name `new` should generally be used for the primary method of instantiating a type. Sometimes it takes no arguments, as in the examples above. Sometimes it does take arguments, like `Box::new` which is passed the value to place in the `Box`.

Some types' constructors, most notably I/O resource types, use distinct naming conventions for their constructors, as in `File::open`, `Mmap::open`, `TcpStream::connect`, and `UdpSocket::bind`. In these cases names are chosen as appropriate for the domain.

Often there are multiple ways to construct a type. It's common in these cases for secondary constructors to be suffixed `_with_foo`, as in `Mmap::open_with_offset`. If your type has a multiplicity of construction options though, consider the builder pattern (C-BUILDER) instead.

Some constructors are "conversion constructors", methods that create a new type from an existing value of a different type. These typically have names beginning with `from_` as in `std::io::Error::from_raw_os_error`. Note also though the `From` trait (C-CONV-TRAITS), which is quite similar. There are three distinctions between a `from_`-prefixed conversion constructor and a `From<T>` impl.

- A `from_` constructor can be unsafe; a `From` impl cannot. One example of this is `Box::from_raw`.
- A `from_` constructor can accept additional arguments to disambiguate the meaning of the source data, as in `u64::from_str_radix`.
- A `From` impl is only appropriate when the source data type is sufficient to determine the encoding of the output data type. When the input is just a bag of bits like in `u64::from_be` or `String::from_utf8`, the conversion constructor name is able to identify their meaning.

Note that it is common and expected for types to implement both `Default` and a `new` constructor. For types that have both, they should have the same behavior. Either one may be implemented in terms of the other.

Examples from the standard library

- `std::io::Error::new` is the commonly used constructor for an IO error.
- `std::io::Error::from_raw_os_error` is a conversion constructor based on an error code received from the operating system.
- `Box::new` creates a new container type, taking a single argument.
- `File::open` opens a file resource.
- `Mmap::open_with_offset` opens a memory-mapped file, with additional options.

Flexibility

Functions expose intermediate results to avoid duplicate work (C-INTERMEDIATE)

Many functions that answer a question also compute interesting related data. If this data is potentially of interest to the client, consider exposing it in the API.

Examples from the standard library

- `Vec::binary_search` does not return a `bool` of whether the value was found, nor an `Option<usize>` of the index at which the value was maybe found. Instead it returns information about the index if found, and also the index at which the value would need to be inserted if not found.

- `String::from_utf8` may fail if the input bytes are not UTF-8. In the error case it returns an intermediate result that exposes the byte offset up to which the input was valid UTF-8, as well as handing back ownership of the input bytes.

- `HashMap::insert` returns an `Option<T>` that returns the preexisting value for a given key, if any. For cases where the user wants to recover this value having it returned by the insert operation avoids the user having to do a second hash table lookup.

Caller decides where to copy and place data (C-CALLER-CONTROL)

If a function requires ownership of an argument, it should take ownership of the argument rather than borrowing and cloning the argument.

```
// Prefer this:
fn foo(b: Bar) {
    /* use b as owned, directly */
}

// Over this:
fn foo(b: &Bar) {
    let b = b.clone();
    /* use b as owned after cloning */
}
```

If a function *does not* require ownership of an argument, it should take a shared or exclusive borrow of the argument rather than taking ownership and dropping the argument.

```
// Prefer this:
fn foo(b: &Bar) {
    /* use b as borrowed */
}

// Over this:
fn foo(b: Bar) {
    /* use b as borrowed, it is implicitly dropped before function returns */
}
```

The `Copy` trait should only be used as a bound when absolutely needed, not as a way of signaling that copies should be cheap to make.

Functions minimize assumptions about parameters by using generics (C-GENERIC)

The fewer assumptions a function makes about its inputs, the more widely usable it becomes.

Prefer

```
fn foo<I: IntoIterator<Item = i64>>(iter: I) { /* ... */ }
```

over any of

```
fn foo(c: &[i64]) { /* ... */ }
fn foo(c: &Vec<i64>) { /* ... */ }
fn foo(c: &SomeOtherCollection<i64>) { /* ... */ }
```

if the function only needs to iterate over the data.

More generally, consider using generics to pinpoint the assumptions a function needs to make about its arguments.

Advantages of generics

- *Reusability*. Generic functions can be applied to an open-ended collection of types, while giving a clear contract for the functionality those types must provide.
- *Static dispatch and optimization*. Each use of a generic function is specialized ("monomorphized") to the particular types implementing the trait bounds, which means that (1) invocations of trait methods are static, direct calls to the implementation and (2) the compiler can inline and otherwise optimize these calls.
- *Inline layout*. If a `struct` and `enum` type is generic over some type parameter `T`, values of type `T` will be laid out inline in the `struct`/`enum`, without any indirection.
- *Inference*. Since the type parameters to generic functions can usually be inferred, generic functions can help cut down on verbosity in code where explicit conversions or other method calls would usually be necessary.
- *Precise types*. Because generics give a *name* to the specific type implementing a trait, it is possible to be precise about places where that exact type is required or produced. For example, a function

  ```
  fn binary<T: Trait>(x: T, y: T) -> T
  ```

  is guaranteed to consume and produce elements of exactly the same type `T`; it cannot be invoked with parameters of different types that both implement `Trait`.

Disadvantages of generics

- *Code size*. Specializing generic functions means that the function body is duplicated. The increase in code size must be weighed against the performance benefits of static dispatch.
- *Homogeneous types*. This is the other side of the "precise types" coin: if `T` is a type parameter, it stands for a *single* actual type. So for example a `Vec<T>` contains elements of a single concrete type (and, indeed, the vector representation is specialized to lay these out in line). Sometimes heterogeneous collections are useful; see trait objects.
- *Signature verbosity*. Heavy use of generics can make it more difficult to read and understand a function's signature.

Examples from the standard library

- `std::fs::File::open` takes an argument of generic type `AsRef<Path>`. This allows files to be opened conveniently from a string literal `"f.txt"`, a `Path`, an `OsString`, and a few other types.

Traits are object-safe if they may be useful as a trait object (C-OBJECT)

Trait objects have some significant limitations: methods invoked through a trait object cannot use generics, and cannot use `Self` except in receiver position.

When designing a trait, decide early on whether the trait will be used as an object or as a bound on generics.

If a trait is meant to be used as an object, its methods should take and return trait objects rather than use generics.

A `where` clause of `Self: Sized` may be used to exclude specific methods from the trait's object. The following trait is not object-safe due to the generic method.

```rust
trait MyTrait {
    fn object_safe(&self, i: i32);

    fn not_object_safe<T>(&self, t: T);
}
```

Adding a requirement of `Self: Sized` to the generic method excludes it from the trait object and makes the trait object-safe.

```rust
trait MyTrait {
    fn object_safe(&self, i: i32);

    fn not_object_safe<T>(&self, t: T) where Self: Sized;
}
```

Advantages of trait objects

- *Heterogeneity*. When you need it, you really need it.
- *Code size*. Unlike generics, trait objects do not generate specialized (monomorphized) versions of code, which can greatly reduce code size.

Disadvantages of trait objects

- *No generic methods*. Trait objects cannot currently provide generic methods.
- *Dynamic dispatch and fat pointers*. Trait objects inherently involve indirection and vtable dispatch, which can carry a performance penalty.
- *No Self*. Except for the method receiver argument, methods on trait objects cannot use the `Self` type.

Examples from the standard library

- The `io::Read` and `io::Write` traits are often used as objects.
- The `Iterator` trait has several generic methods marked with `where Self: Sized` to retain the ability to use `Iterator` as an object.

Type safety

Newtypes provide static distinctions (C-NEWTYPE)

Newtypes can statically distinguish between different interpretations of an underlying type.

For example, a `f64` value might be used to represent a quantity in miles or in kilometers. Using newtypes, we can keep track of the intended interpretation:

```
struct Miles(pub f64);
struct Kilometers(pub f64);

impl Miles {
    fn to_kilometers(self) -> Kilometers { /* ... */ }
}
impl Kilometers {
    fn to_miles(self) -> Miles { /* ... */ }
}
```

Once we have separated these two types, we can statically ensure that we do not confuse them. For example, the function

```
fn are_we_there_yet(distance_travelled: Miles) -> bool { /* ... */ }
```

cannot accidentally be called with a `Kilometers` value. The compiler will remind us to perform the conversion, thus averting certain catastrophic bugs.

Arguments convey meaning through types, not `bool` or `Option` (C-CUSTOM-TYPE)

Prefer

```
let w = Widget::new(Small, Round)
```

over

```
let w = Widget::new(true, false)
```

Core types like `bool`, `u8` and `Option` have many possible interpretations.

Use a deliberate type (whether enum, struct, or tuple) to convey interpretation and invariants. In the above example, it is not immediately clear what `true` and `false` are conveying without looking up the argument names, but `Small` and `Round` are more suggestive.

Using custom types makes it easier to expand the options later on, for example by adding an `ExtraLarge` variant.

See the newtype pattern (C-NEWTYPE) for a no-cost way to wrap existing types with a distinguished name.

Types for a set of flags are `bitflags`, not enums (C-BITFLAG)

Rust supports `enum` types with explicitly specified discriminants:

```
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff,
}
```

Custom discriminants are useful when an `enum` type needs to be serialized to an integer value compatibly with some other system/language. They support "typesafe" APIs: by taking a `Color`, rather than an integer, a function is guaranteed to get well-formed inputs, even if it later views those inputs as integers.

An `enum` allows an API to request exactly one choice from among many. Sometimes an API's input is instead the presence or absence of a set of flags. In C code, this is often done by having each flag correspond to a particular bit, allowing a single integer to represent, say, 32 or 64 flags. Rust's `bitflags` crate provides a typesafe representation of this pattern.

```
use bitflags::bitflags;

bitflags! {
    struct Flags: u32 {
        const FLAG_A = 0b00000001;
        const FLAG_B = 0b00000010;
        const FLAG_C = 0b00000100;
    }
}

fn f(settings: Flags) {
    if settings.contains(Flags::FLAG_A) {
        println!("doing thing A");
    }
    if settings.contains(Flags::FLAG_B) {
        println!("doing thing B");
    }
    if settings.contains(Flags::FLAG_C) {
        println!("doing thing C");
    }
}

fn main() {
    f(Flags::FLAG_A | Flags::FLAG_C);
}
```

Builders enable construction of complex values (C-BUILDER)

Some data structures are complicated to construct, due to their construction needing:

- a large number of inputs
- compound data (e.g. slices)
- optional configuration data
- choice between several flavors

which can easily lead to a large number of distinct constructors with many arguments each.

If `T` is such a data structure, consider introducing a `T` *builder*:

1. Introduce a separate data type `TBuilder` for incrementally configuring a `T` value. When possible, choose a better name: e.g. `Command` is the builder for a child process, `Url` can be created from a `ParseOptions`.
2. The builder constructor should take as parameters only the data *required* to make a `T`.
3. The builder should offer a suite of convenient methods for configuration, including setting up compound inputs (like slices) incrementally. These methods should return `self` to allow

   chaining.
4. The builder should provide one or more "*terminal*" methods for actually building a `T`.

The builder pattern is especially appropriate when building a `T` involves side effects, such as spawning a task or launching a process.

In Rust, there are two variants of the builder pattern, differing in the treatment of ownership, as described below.

Non-consuming builders (preferred)

In some cases, constructing the final `T` does not require the builder itself to be consumed. The following variant on `std::process::Command` is one example:

```rust
// NOTE: the actual Command API does not use owned Strings;
// this is a simplified version.

pub struct Command {
    program: String,
    args: Vec<String>,
    cwd: Option<String>,
    // etc
}

impl Command {
    pub fn new(program: String) -> Command {
        Command {
            program: program,
            args: Vec::new(),
            cwd: None,
        }
    }

    /// Add an argument to pass to the program.
    pub fn arg(&mut self, arg: String) -> &mut Command {
        self.args.push(arg);
        self
    }

    /// Add multiple arguments to pass to the program.
    pub fn args(&mut self, args: &[String]) -> &mut Command {
        self.args.extend_from_slice(args);
        self
    }

    /// Set the working directory for the child process.
    pub fn current_dir(&mut self, dir: String) -> &mut Command {
        self.cwd = Some(dir);
        self
    }

    /// Executes the command as a child process, which is returned.
    pub fn spawn(&self) -> io::Result<Child> {
        /* ... */
    }
}
```

Note that the `spawn` method, which actually uses the builder configuration to spawn a process, takes the builder by shared reference. This is possible because spawning the process does not require ownership of the configuration data.

Because the terminal `spawn` method only needs a reference, the configuration methods take and return a mutable borrow of `self`.

The benefit

By using borrows throughout, `Command` can be used conveniently for both one-liner and more complex constructions:

```rust
// One-liners
Command::new("/bin/cat").arg("file.txt").spawn();

// Complex configuration
let mut cmd = Command::new("/bin/ls");
if size_sorted {
    cmd.arg("-S");
}
cmd.arg(".");
cmd.spawn();
```

Consuming builders

Sometimes builders must transfer ownership when constructing the final type `T`, meaning that the terminal methods must take `self` rather than `&self`.

```rust
impl TaskBuilder {
    /// Name the task-to-be.
    pub fn named(mut self, name: String) -> TaskBuilder {
        self.name = Some(name);
        self
    }

    /// Redirect task-local stdout.
    pub fn stdout(mut self, stdout: Box<io::Write + Send>) -> TaskBuilder {
        self.stdout = Some(stdout);
        self
    }

    /// Creates and executes a new child task.
    pub fn spawn<F>(self, f: F) where F: FnOnce() + Send {
        /* ... */
    }
}
```

Here, the `stdout` configuration involves passing ownership of an `io::Write`, which must be transferred to the task upon construction (in `spawn`).

When the terminal methods of the builder require ownership, there is a basic tradeoff:

- If the other builder methods take/return a mutable borrow, the complex configuration case will work well, but one-liner configuration becomes impossible.

- If the other builder methods take/return an owned `self`, one-liners continue to work well but complex configuration is less convenient.

Under the rubric of making easy things easy and hard things possible, all builder methods for a consuming builder should take and return an owned `self`. Then client code works as follows:

```rust
// One-liners
TaskBuilder::new("my_task").spawn(|| { /* ... */ });

// Complex configuration
let mut task = TaskBuilder::new();
task = task.named("my_task_2"); // must re-assign to retain ownership
if reroute {
    task = task.stdout(mywriter);
}
task.spawn(|| { /* ... */ });
```

One-liners work as before, because ownership is threaded through each of the builder methods until being consumed by `spawn`. Complex configuration, however, is more verbose: it requires re-assigning the builder at each step.

Dependability

Functions validate their arguments (C-VALIDATE)

Rust APIs do *not* generally follow the robustness principle: "be conservative in what you send; be liberal in what you accept".

Instead, Rust code should *enforce* the validity of input whenever practical.

Enforcement can be achieved through the following mechanisms (listed in order of preference).

Static enforcement

Choose an argument type that rules out bad inputs.

For example, prefer

```rust
fn foo(a: Ascii) { /* ... */ }
```

over

```rust
fn foo(a: u8) { /* ... */ }
```

where `Ascii` is a *wrapper* around `u8` that guarantees the highest bit is zero; see newtype patterns (C-NEWTYPE) for more details on creating typesafe wrappers.

Static enforcement usually comes at little run-time cost: it pushes the costs to the boundaries (e.g. when a `u8` is first converted into an `Ascii`). It also catches bugs early, during compilation, rather than through run-time failures.

On the other hand, some properties are difficult or impossible to express using types.

Dynamic enforcement

Validate the input as it is processed (or ahead of time, if necessary). Dynamic checking is often easier to implement than static checking, but has several downsides:

1. Runtime overhead (unless checking can be done as part of processing the input).

2. Delayed detection of bugs.
3. Introduces failure cases, either via `panic!` or `Result`/`Option` types, which must then be dealt with by client code.

Dynamic enforcement with `debug_assert!`

Same as dynamic enforcement, but with the possibility of easily turning off expensive checks for production builds.

Dynamic enforcement with opt-out

Same as dynamic enforcement, but adds sibling functions that opt out of the checking.

The convention is to mark these opt-out functions with a suffix like `_unchecked` or by placing them in a `raw` submodule.

The unchecked functions can be used judiciously in cases where (1) performance dictates avoiding checks and (2) the client is otherwise confident that the inputs are valid.

Destructors never fail (C-DTOR-FAIL)

Destructors are executed while panicking, and in that context a failing destructor causes the program to abort.

Instead of failing in a destructor, provide a separate method for checking for clean teardown, e.g. a `close` method, that returns a `Result` to signal problems. If that `close` method is not called, the `Drop` implementation should do the teardown and ignore or log/trace any errors it produces.

Destructors that may block have alternatives (C-DTOR-BLOCK)

Similarly, destructors should not invoke blocking operations, which can make debugging much more difficult. Again, consider providing a separate method for preparing for an infallible, nonblocking teardown.

Debuggability

All public types implement `Debug` (C-DEBUG)

If there are exceptions, they are rare.

`Debug` representation is never empty (C-DEBUG-NONEMPTY)

Even for conceptually empty values, the `Debug` representation should never be empty.

```
let empty_str = "";
assert_eq!(format!("{:?}", empty_str), "\"\"");

let empty_vec = Vec::<bool>::new();
assert_eq!(format!("{:?}", empty_vec), "[]");
```

Future proofing

Sealed traits protect against downstream implementations (C-SEALED)

Some traits are only meant to be implemented within the crate that defines them. In such cases, we can retain the ability to make changes to the trait in a non-breaking way by using the sealed trait pattern.

```
/// This trait is sealed and cannot be implemented for types outside this crate.
pub trait TheTrait: private::Sealed {
    // Zero or more methods that the user is allowed to call.
    fn ...();

    // Zero or more private methods, not allowed for user to call.
    #[doc(hidden)]
    fn ...();
}

// Implement for some types.
impl TheTrait for usize {
    /* ... */
}

mod private {
    pub trait Sealed {}

    // Implement for those same types, but no others.
    impl Sealed for usize {}
}
```

The empty private `Sealed` supertrait cannot be named by downstream crates, so we are guaranteed that implementations of `Sealed` (and therefore `TheTrait`) only exist in the current crate. We are free to add methods to `TheTrait` in a non-breaking release even though that would ordinarily be a breaking change for traits that are not sealed. Also we are free to change the signature of methods that are not publicly documented.

Note that removing a public method or changing the signature of a public method in a sealed trait are still breaking changes.

To avoid frustrated users trying to implement the trait, it should be documented in rustdoc that the trait is sealed and not meant to be implemented outside of the current crate.

Examples

- `serde_json::value::Index`
- `byteorder::ByteOrder`

Structs have private fields (C-STRUCT-PRIVATE)

Making a field public is a strong commitment: it pins down a representation choice, *and* prevents the type from providing any validation or maintaining any invariants on the contents of the field, since clients can mutate it arbitrarily.

Public fields are most appropriate for `struct` types in the C spirit: compound, passive data structures. Otherwise, consider providing getter/setter methods and hiding fields instead.

Newtypes encapsulate implementation details (C-NEWTYPE-HIDE)

A newtype can be used to hide representation details while making precise promises to the client.

For example, consider a function `my_transform` that returns a compound iterator type.

```rust
use std::iter::{Enumerate, Skip};

pub fn my_transform<I: Iterator>(input: I) -> Enumerate<Skip<I>> {
    input.skip(3).enumerate()
}
```

We wish to hide this type from the client, so that the client's view of the return type is roughly `Iterator<Item = (usize, T)>`. We can do so using the newtype pattern:

```rust
use std::iter::{Enumerate, Skip};

pub struct MyTransformResult<I>(Enumerate<Skip<I>>);

impl<I: Iterator> Iterator for MyTransformResult<I> {
    type Item = (usize, I::Item);

    fn next(&mut self) -> Option<Self::Item> {
        self.0.next()
    }
}

pub fn my_transform<I: Iterator>(input: I) -> MyTransformResult<I> {
    MyTransformResult(input.skip(3).enumerate())
}
```

Aside from simplifying the signature, this use of newtypes allows us to promise less to the client. The client does not know *how* the result iterator is constructed or represented, which means the representation can change in the future without breaking client code.

Rust 1.26 also introduces the `impl Trait` feature, which is more concise than the newtype pattern but with some additional trade offs, namely with `impl Trait` you are limited in what you can express. For example, returning an iterator that impls `Debug` or `Clone` or some combination of the other iterator extension traits can be problematic. In summary `impl Trait` as a return type is probably great for internal APIs and may even be appropriate for public APIs, but probably not in all cases. See the "`impl Trait` for returning complex types with ease" section of the Edition Guide for more details.

```rust
pub fn my_transform<I: Iterator>(input: I) -> impl Iterator<Item = (usize, I::Item)> {
    input.skip(3).enumerate()
}
```

Data structures do not duplicate derived trait bounds (C-STRUCT-BOUNDS)

Generic data structures should not use trait bounds that can be derived or do not otherwise add semantic value. Each trait in the `derive` attribute will be expanded into a separate `impl` block that only applies to generic arguments that implement that trait.

```
// Prefer this:
#[derive(Clone, Debug, PartialEq)]
struct Good<T> { /* ... */ }

// Over this:
#[derive(Clone, Debug, PartialEq)]
struct Bad<T: Clone + Debug + PartialEq> { /* ... */ }
```

Duplicating derived traits as bounds on `Bad` is unnecessary and a backwards-compatibiliity hazard. To illustrate this point, consider deriving `PartialOrd` on the structures in the previous example:

```
// Non-breaking change:
#[derive(Clone, Debug, PartialEq, PartialOrd)]
struct Good<T> { /* ... */ }

// Breaking change:
#[derive(Clone, Debug, PartialEq, PartialOrd)]
struct Bad<T: Clone + Debug + PartialEq + PartialOrd> { /* ... */ }
```

Generally speaking, adding a trait bound to a data structure is a breaking change because every consumer of that structure will need to start satisfying the additional bound. Deriving more traits from the standard library using the `derive` attribute is not a breaking change.

The following traits should never be used in bounds on data structures:

- `Clone`
- `PartialEq`
- `PartialOrd`
- `Debug`
- `Display`
- `Default`
- `Error`
- `Serialize`
- `Deserialize`
- `DeserializeOwned`

There is a grey area around other non-derivable trait bounds that are not strictly required by the structure definition, like `Read` or `Write`. They may communicate the intended behavior of the type better in its definition but also limits future extensibility. Including semantically useful trait bounds on data structures is still less problematic than including derivable traits as bounds.

Exceptions

There are three exceptions where trait bounds on structures are required:

1. The data structure refers to an associated type on the trait.
2. The bound is `?Sized`.
3. The data structure has a `Drop` impl that requires trait bounds. Rust currently requires all trait bounds on the `Drop` impl are also present on the data structure.

Examples from the standard library

- `std::borrow::Cow` refers to an associated type on the `Borrow` trait.
- `std::boxed::Box` opts out of the implicit `Sized` bound.
- `std::io::BufWriter` requires a trait bound in its `Drop` impl.

Necessities

Public dependencies of a stable crate are stable (C-STABLE)

A crate cannot be stable (>=1.0.0) without all of its public dependencies being stable.

Public dependencies are crates from which types are used in the public API of the current crate.

```
pub fn do_my_thing(arg: other_crate::TheirThing) { /* ... */ }
```

A crate containing this function cannot be stable unless `other_crate` is also stable.

Be careful because public dependencies can sneak in at unexpected places.

```
pub struct Error {
    private: ErrorImpl,
}

enum ErrorImpl {
    Io(io::Error),
    // Should be okay even if other_crate isn't
    // stable, because ErrorImpl is private.
    Dep(other_crate::Error),
}

// Oh no! This puts other_crate into the public API
// of the current crate.
impl From<other_crate::Error> for Error {
    fn from(err: other_crate::Error) -> Self {
        Error { private: ErrorImpl::Dep(err) }
    }
}
```

Crate and its dependencies have a permissive license (C-PERMISSIVE)

The software produced by the Rust project is dual-licensed, under either the MIT or Apache 2.0 licenses. Crates that simply need the maximum compatibility with the Rust ecosystem are recommended to do the same, in the manner described herein. Other options are described below.

These API guidelines do not provide a detailed explanation of Rust's license, but there is a small amount said in the Rust FAQ. These guidelines are concerned with matters of interoperability with Rust, and are not comprehensive over licensing options.

To apply the Rust license to your project, define the `license` field in your `Cargo.toml` as:

```toml
[package]
name = "..."
version = "..."
authors = ["..."]
license = "MIT OR Apache-2.0"
```

Then add the files `LICENSE-APACHE` and `LICENSE-MIT` in the repository root, containing the text of the licenses (which you can obtain, for instance, from choosealicense.com, for Apache-2.0 and MIT).

And toward the end of your README.md:

```
## License

Licensed under either of

 * Apache License, Version 2.0
   ([LICENSE-APACHE](LICENSE-APACHE) or <http://www.apache.org/licenses/LICENSE-2.0>)
 * MIT license
   ([LICENSE-MIT](LICENSE-MIT) or <http://opensource.org/licenses/MIT>)

at your option.

## Contribution

Unless you explicitly state otherwise, any contribution intentionally submitted
for inclusion in the work by you, as defined in the Apache-2.0 license, shall be
dual licensed as above, without any additional terms or conditions.
```

Besides the dual MIT/Apache-2.0 license, another common licensing approach used by Rust crate authors is to apply a single permissive license such as MIT or BSD. This license scheme is also entirely compatible with Rust's, because it imposes the minimal restrictions of Rust's MIT license.

Crates that desire perfect license compatibility with Rust are not recommended to choose only the Apache license. The Apache license, though it is a permissive license, imposes restrictions beyond the MIT and BSD licenses that can discourage or prevent their use in some scenarios, so Apache-only software cannot be used in some situations where most of the Rust runtime stack can.

The license of a crate's dependencies can affect the restrictions on distribution of the crate itself, so a permissively-licensed crate should generally only depend on permissively-licensed crates.

External links

- RFC 199 - Ownership naming conventions
- RFC 344 - Naming conventions
- RFC 430 - Naming conventions
- RFC 505 - Doc conventions
- RFC 1574 - Doc conventions
- RFC 1687 - Crate-level documentation
- Elegant Library APIs in Rust
- Rust Design Patterns