ⓘ About

A collection of pragmatic design guidelines helping application and library developers to produce idiomatic Rust that scales.

Meta Design Principles

We build on existing high-quality guidelines, most notably the Rust API Guidelines, addressing topics often encountered by Rust developers. For a guideline to make it into this book, we expect it to meet the following criteria:

- ■ It positively affects { safety, COGs, maintenance }; i.e., it must where applicable
    - ■ promote **safety best-practices** and prevent sources of risk
    - ■ lead to **high throughput**, **low latency**, and **low memory usage**
    - ■ make code **readable and understandable**
- ■ A majority of experienced (3+ years) **Rust developers would agree with the guideline**.
- ■ The guideline is **reasonably comprehensible** to Rust novices (4+ weeks).
- ■ It is **pragmatic**, as unrealistic guidelines won't be followed.

Applicability

Guidelines declared *must* are supposed to always hold, where *should* guidelines indicate more flexibility.

That said, teams are free to adopt these guidelines as they see fit, and you occasionally might have good reasons to do things differently.

We recommend you try to apply all items to your project. If an item does not make sense, get in touch. Either the item has issues and we should update it, or it does not apply (and we might update it anyway to point these edge cases out).

💡 The Golden Rule

Each item here exists for a reason; and it is the spirit that counts, not the letter.

Before attempting to work around a guideline, you should understand why it exists and what it tries to safeguard. Likewise, you should not blindly follow a guideline if it becomes apparent that doing so would violate its underlying motivation!

Guideline Maturity

We expect our guidelines to evolve over time, taking into account lessons learned, and following changes to the language. Each guideline therefore comes with a version number, analogous to Rust's semver usage in spirit.

Submitting New Guidelines

Do you have a practical guideline that leads to better safety, COGS or maintainability? We'd love to hear from you! Here is the process you should follow:

- Check if your guideline follows the meta design principles above.
- Check if your suggestion is not already covered by the API Guidelines or Clippy.
- File a PR or issue.

This book was last generated on: 2025-11-05

Checklist

- **Universal**
  - ■ Follow the Upstream Guidelines (M-UPSTREAM-GUIDELINES)
  - ■ Use Static Verification (M-STATIC-VERIFICATION)
  - ■ Lint Overrides Should Use `#[expect]` (M-LINT-OVERRIDE-EXPECT)
  - ■ Public Types are Debug (M-PUBLIC-DEBUG)
  - ■ Public Types Meant to be Read are Display (M-PUBLIC-DISPLAY)
  - ■ If in Doubt, Split the Crate (M-SMALLER-CRATES)
  - ■ Names are Free of Weasel Words (M-CONCISE-NAMES)
  - ■ Prefer Regular over Associated Functions (M-REGULAR-FN)
  - ■ Panic Means 'Stop the Program' (M-PANIC-IS-STOP)
  - ■ Detected Programming Bugs are Panics, Not Errors (M-PANIC-ON-BUG)
  - ■ All Magic Values and Behaviors are Documented (M-DOCUMENTED-MAGIC)
  - ■ Use Structured Logging with Message Templates (M-LOG-STRUCTURED)
- **Library / Interoperability**
  - ■ Types are Send (M-TYPES-SEND)
  - ■ Native Escape Hatches (M-ESCAPE-HATCHES)
  - ■ Don't Leak External Types (M-DONT-LEAK-TYPES)
- **Library / UX**
  - ■ Abstractions Don't Visibly Nest (M-SIMPLE-ABSTRACTIONS)
  - ■ Avoid Smart Pointers and Wrappers in APIs (M-AVOID-WRAPPERS)
  - ■ Prefer Types over Generics, Generics over Dyn Traits (M-DI-HIERARCHY)
  - ■ Error are Canonical Structs (M-ERRORS-CANONICAL-STRUCTS)
  - ■ Complex Type Construction has Builders (M-INIT-BUILDER)
  - ■ Complex Type Initialization Hierarchies are Cascaded (M-INIT-CASCADED)
  - ■ Services are Clone (M-SERVICES-CLONE)
  - ■ Accept `impl AsRef<>` Where Feasible (M-IMPL-ASREF)
  - ■ Accept `impl RangeBounds<>` Where Feasible (M-IMPL-RANGEBOUNDS)
  - ■ Accept `impl 'IO'` Where Feasible ('Sans IO') (M-IMPL-IO)
  - ■ Essential Functionality Should be Inherent (M-ESSENTIAL-FN-INHERENT)
- **Library / Resilience**
  - ■ I/O and System Calls Are Mockable (M-MOCKABLE-SYSCALLS)
  - ■ Test Utilities are Feature Gated (M-TEST-UTIL)
  - ■ Use the Proper Type Family (M-STRONG-TYPES)
  - ■ Don't Glob Re-Export Items (M-NO-GLOB-REEXPORTS)
  - ■ Avoid Statics (M-AVOID-STATICS)
- **Library / Building**
  - ■ Libraries Work Out of the Box (M-OOBE)
  - ■ Native `-sys` Crates Compile Without Dependencies (M-SYS-CRATES)
  - ■ Features are Additive (M-FEATURES-ADDITIVE)
- **Applications**
  - ■ Use Mimalloc for Apps (M-MIMALLOC-APP)
  - ■ Applications may use Anyhow or Derivatives (M-APP-ERROR)
- **FFI**
  - ■ Isolate DLL State Between FFI Libraries (M-ISOLATE-DLL-STATE)
- **Safety**
  - ■ Unsafe Needs Reason, Should be Avoided (M-UNSAFE)
  - ■ Unsafe Implies Undefined Behavior (M-UNSAFE-IMPLIES-UB)

- ◦ ⬛ All Code Must be Sound (M-UNSOUND)
- **Performance**
    - ⬛ Optimize for Throughput, Avoid Empty Cycles (M-THROUGHPUT)
    - ⬛ Identify, Profile, Optimize the Hot Path Early (M-HOTPATH)
    - ⬛ Long-Running Tasks Should Have Yield Points. (M-YIELD-POINTS)
- **Documentation**
    - ⬛ First Sentence is One Line; Approx. 15 Words (M-FIRST-DOC-SENTENCE)
    - ⬛ Has Module Documentation (M-MODULE-DOCS)
    - ⬛ Documentation Has Canonical Sections (M-CANONICAL-DOCS)
    - ⬛ Mark `pub use` Items with `#[doc(inline)]` (M-DOC-INLINE)
- **AI**
    - ⬛ Design with AI use in Mind (M-DESIGN-FOR-AI)

Universal Guidelines

Follow the Upstream Guidelines (M-UPSTREAM-GUIDELINES)

Why this entry exists:	To avoid repeating mistakes the community has already learned from, and to have a codebase that does not surprise users and contributors.
Version:	1.0

The guidelines in this book complement existing Rust guidelines, in particular:

- Rust API Guidelines
- Rust Style Guide
- Rust Design Patterns
- Rust Reference - Undefined Behavior

We recommend you read through these as well, and apply them in addition to this book's items. Pay special attention to the ones below, as they are frequently forgotten:

- ■ C-CONV - Ad-hoc conversions follow `as_`, `to_`, `into_` conventions
- ■ C-GETTER - Getter names follow Rust convention
- ■ C-COMMON-TRAITS - Types eagerly implement common traits
  - `Copy`, `Clone`, `Eq`, `PartialEq`, `Ord`, `PartialOrd`, `Hash`, `Default`, `Debug`
  - `Display` where type wants to be displayed
- ■ C-CTOR - Constructors are static, inherent methods
  - In particular, have `Foo::new()`, even if you have `Foo::default()`
- ■ C-FEATURE - Feature names are free of placeholder words

Use Static Verification (M-STATIC-VERIFICATION)

Why this entry exists:	To ensure consistency and avoid common issues.
Version:	1.0

Projects should use the following static verification tools to help maintain the quality of the code. These tools can be configured to run on a developer's machine during normal work, and should be used as part of check-in gates.

- compiler lints offer many lints to avoid bugs and improve code quality.
- clippy lints contain hundreds of lints to avoid bugs and improve code quality.
- rustfmt ensures consistent source formatting.
- cargo-audit verifies crate dependencies for security vulnerabilities.
- cargo-hack validates that all combinations of crate features work correctly.
- cargo-udeps detects unused dependencies in Cargo.toml files.
- miri validates the correctness of unsafe code.

Compiler Lints

The Rust compiler generally produces exceptionally good diagnostics. In addition to the default set of diagnostics, projects should explicitly enable the following set of compiler lints:

```toml
[lints.rust]
ambiguous_negative_literals = "warn"
missing_debug_implementations = "warn"
redundant_imports = "warn"
redundant_lifetimes = "warn"
trivial_numeric_casts = "warn"
unsafe_op_in_unsafe_fn = "warn"
unused_lifetimes = "warn"
```

Clippy Lints

For clippy, projects should enable all major lint categories, and additionally enable some lints from the `restriction` lint group. Undesired lints (e.g., numeric casts) can be opted back out of on a case-by-case basis:

```
[lints.clippy]
cargo = { level = "warn", priority = -1 }
complexity = { level = "warn", priority = -1 }
correctness = { level = "warn", priority = -1 }
pedantic = { level = "warn", priority = -1 }
perf = { level = "warn", priority = -1 }
style = { level = "warn", priority = -1 }
suspicious = { level = "warn", priority = -1 }
# nursery = { level = "warn", priority = -1 }  # optional, might cause more false
positives

# These lints are from the `restriction` lint group and prevent specific
# constructs being used in source code in order to drive up consistency,
# quality, and brevity
allow_attributes_without_reason = "warn"
as_pointer_underscore = "warn"
assertions_on_result_states = "warn"
clone_on_ref_ptr = "warn"
deref_by_slicing = "warn"
disallowed_script_idents = "warn"
empty_drop = "warn"
empty_enum_variants_with_brackets = "warn"
empty_structs_with_brackets = "warn"
fn_to_numeric_cast_any = "warn"
if_then_some_else_none = "warn"
map_err_ignore = "warn"
redundant_type_annotations = "warn"
renamed_function_params = "warn"
semicolon_outside_block = "warn"
string_to_string = "warn"
undocumented_unsafe_blocks = "warn"
unnecessary_safety_comment = "warn"
unnecessary_safety_doc = "warn"
unneeded_field_pattern = "warn"
unused_result_ok = "warn"

# May cause issues with structured logging otherwise.
literal_string_with_formatting_args = "allow"

# Define custom opt outs here
# ...
```

Lint Overrides Should Use `#[expect]` (M-LINT-OVERRIDE-EXPECT)

Why this entry exists: To prevent the accumulation of outdated lints.
Version: 1.0

When overriding project-global lints inside a submodule or item, you should do so via `#[expect]`, not `#[allow]`.

Expected lints emit a warning if the marked warning was not encountered, thus preventing the accumulation of stale lints. That said, `#[allow]` lints are still useful when applied to generated code, and can appear in macros.

Overrides should be accompanied by a `reason`:

```rust
#[expect(clippy::unused_async, reason = "API fixed, will use I/O later")]
pub async fn ping_server() {
  // Stubbed out for now
}
```

Public Types are Debug (M-PUBLIC-DEBUG)

Why this entry exists: To simplify debugging and prevent leaking sensitive data.

Version: 1.0

All public types exposed by a crate should implement `Debug`. Most types can do so via `#[derive(Debug)]`:

```rust
#[derive(Debug)]
struct Endpoint(String);
```

Types designed to hold sensitive data should also implement `Debug`, but do so via a custom implementation. This implementation must employ unit tests to ensure sensitive data isn't actually leaked, and will not be in the future.

```rust
use std::fmt::{Debug, Formatter};

struct UserSecret(String);

impl Debug for UserSecret {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        write!(f, "UserSecret(...)")
    }
}

#[test]
fn test() {
    let key = "552d3454-d0d5-445d-ab9f-ef2ae3a8896a";
    let secret = UserSecret(key.to_string());
    let rendered = format!("{:?}", secret);

    assert!(rendered.contains("UserSecret"));
    assert!(!rendered.contains(key));
}
```

Public Types Meant to be Read are Display (M-PUBLIC-DISPLAY)

Why this entry exists: To improve usability.

Version: 1.0

If your type is expected to be read by upstream consumers, be it developers or end users, it should implement `Display`. This in particular includes:

- Error types, which are mandated by `std::error::Error` to implement `Display`
- Wrappers around string-like data

Implementations of `Display` should follow Rust customs; this includes rendering newlines and escape sequences. The handling of sensitive data outlined in M-PUBLIC-DEBUG applies analogously.

If in Doubt, Split the Crate (M-SMALLER-CRATES)

Why this entry exists:	To improve compile times and modularity.
Version:	1.0

You should err on the side of having too many crates rather than too few, as this leads to dramatic compile time improvements—especially during the development of these crates—and prevents cyclic component dependencies.

Essentially, if a submodule can be used independently, its contents should be moved into a separate crate.

Performing this crate split may cause you to lose access to some `pub(crate)` fields or methods. In many situations, this is a desirable side-effect and should prompt you to design more flexible abstractions that would give your users similar affordances.

In some cases, it is desirable to re-join individual crates back into a single *umbrella crate*, such as when dealing with proc macros, or runtimes. Functionality split for technical reasons (e.g., a `foo_proc` proc macro crate) should always be re-exported. Otherwise, re-exports should be used sparingly.

💡 Features vs. Crates

As a rule of thumb, crates are for items that can reasonably be used on their own. Features should unlock extra functionality that can't live on its own. In the case of umbrella crates, see below, features may also be used to enable constituents (but then that functionality was extracted into crates already).

For example, if you defined a `web` crate with the following modules, users only needing client calls would also have to pay for the compilation of server code:

```
web::server
web::client
web::protocols
```

Instead, you should introduce individual crates that give users the ability to pick and choose:

```
web_server
web_client
web_protocols
```

Names are Free of Weasel Words (M-CONCISE-NAMES)

Why this entry exists: To improve readability.
Version: 1.0

Symbol names, especially types and traits names, should be free of weasel words that do not meaningfully add information. Common offenders include `Service`, `Manager`, and `Factory`. For example:

While your library may very well contain or communicate with a booking service—or even hold an `HttpClient` instance named `booking_service`—one should rarely encounter a `BookingService` *type* in code.

An item handling many bookings can just be called `Bookings`. If it does anything more specific, then that quality should be appended instead. It submits these items elsewhere? Calling it `BookingDispatcher` would be more helpful.

The same is true for `Manager`s. Every code manages *something*, so that moniker is rarely useful. With rare exceptions, life cycle issues should likewise not be made the subject of some manager. Items are created in whatever way they are needed, their disposal is governed by `Drop`, and only `Drop`.

Regarding factories, at least the term should be avoided. While the concept `FooFactory` has its use, its canonical Rust name is `Builder` (compare M-INIT-BUILDER). A builder that can produce items repeatedly is still a builder.

In addition, accepting factories (builders) as parameters is an unidiomatic import of OO concepts into Rust. If repeatable instantiation is required, functions should ask for an `impl Fn() -> Foo` over a `FooBuilder` or similar. In contrast, standalone builders have their use, but primarily to reduce parametric permutation complexity around optional values (again, M-INIT-BUILDER).

Prefer Regular over Associated Functions (M-REGULAR-FN)

Why this entry exists: To improve readability.
Version: 1.0

Associated functions should primarily be used for instance creation, not general purpose computation.

In contrast to some OO languages, regular functions are first-class citizens in Rust and need no module or *class* to host them. Functionality that does not clearly belong to a receiver should therefore not reside in a type's `impl` block:

```rust
struct Database {}

impl Database {
    // Ok, associated function creates an instance
    fn new() -> Self {}

    // Ok, regular method with `&self` as receiver
    fn query(&self) {}

    // Not ok, this function is not directly related to `Database`,
    // it should therefore not live under `Database` as an associated
    // function.
    fn check_parameters(p: &str) {}
}

// As a regular function this is fine
fn check_parameters(p: &str) {}
```

Regular functions are more idiomatic, and reduce unnecessary noise on the caller side. Associated trait functions are perfectly idiomatic though:

```rust
pub trait Default {
    fn default() -> Self;
}

struct Foo;

impl Default for Foo {
    fn default() -> Self { Self }
}
```

Panic Means 'Stop the Program' (M-PANIC-IS-STOP)

Why this entry exists: To ensure soundness and predictability.
Version: 1.0

Panics are not exceptions. Instead, they suggest immediate program termination.

Although your code must be panic-safe (i.e., a survived panic may not lead to inconsistent state), invoking a panic means *this program should stop now*. It is not valid to:

- use panics to communicate (errors) upstream,
- use panics to handle self-inflicted error conditions,
- assume panics will be caught, even by your own code.

For example, if the application calling you is compiled with a `Cargo.toml` containing

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

then any invocation of panic will cause an otherwise functioning program to needlessly abort. Valid reasons to panic are:

- when encountering a programming error, e.g., `x.expect("must never happen")`,
- anything invoked from const contexts, e.g., `const { foo.unwrap() }`,
- when user requested, e.g., providing an `unwrap()` method yourself,
- when encountering a poison, e.g., by calling `unwrap()` on a lock result (a poisoned lock signals another thread has panicked already).

Any of those are directly or indirectly linked to programming errors.

Detected Programming Bugs are Panics, Not Errors (M-PANIC-ON-BUG)

Why this entry exists: To avoid impossible error handling code and ensure runtime consistency.
Version: 1.0

As an extension of M-PANIC-IS-STOP above, when an unrecoverable programming error has been detected, libraries and applications must panic, i.e., request program termination.

In these cases, no `Error` type should be introduced or returned, as any such error could not be acted upon at runtime.

Contract violations, i.e., the breaking of invariants either within a library or by a caller, are programming errors and must therefore panic.

However, what constitutes a violation is situational. APIs are not expected to go out of their way to detect them, as such checks can be impossible or expensive. Encountering `must_be_even == 3` during an already existing check clearly warrants a panic, while a function `parse(&str)` clearly must return a `Result`. If in doubt, we recommend you take inspiration from the standard library.

```rust
// Generally, a function with bad parameters must either
// - Ignore a parameter and/or return the wrong result
// - Signal an issue via Result or similar
// - Panic
// If in this `divide_by` we see that y == 0, panicking is
// the correct approach.
fn divide_by(x: u32, y: u32) -> u32 { ... }

// However, it can also be permissible to omit such checks
// and return an unspecified (but not an undefined) result.
fn divide_by_fast(x: u32, y: u32) -> u32 { ... }

// Here, passing an invalid URI is not a contract violation.
// Since parsing is inherently fallible, a Result must be returned.
fn parse_uri(s: &str) -> Result<Uri, ParseError> { };
```

💡 Make it 'Correct by Construction'

While panicking on a detected programming error is the 'least bad option', your panic might still ruin someone's day. For any user input or calling sequence that would otherwise panic, you should also explore if you can use the type system to avoid panicking code paths altogether.

Magic Values are Documented (M-DOCUMENTED-MAGIC)

Why this entry exists: To ensure maintainability and prevent misunderstandings when refactoring.
Version: 1.0

Hardcoded *magic* values in production code must be accompanied by a comment. The comment should outline:

- why this value was chosen,
- non-obvious side effects if that value is changed,
- external systems that interact with this constant.

You should prefer named constants over inline values.

```
// Bad: it's relatively obvious that this waits for a day, but not why
wait_timeout(60 * 60 * 24).await // Wait at most a day

// Better
wait_timeout(60 * 60 * 24).await // Large enough value to ensure the server
                                 // can finish. Setting this too low might
                                 // make us abort a valid request. Based on
                                 // `api.foo.com` timeout policies.

// Best

/// How long we wait for the server.
///
/// Large enough value to ensure the server
/// can finish. Setting this too low might
/// make us abort a valid request. Based on
/// `api.foo.com` timeout policies.
const UPSTREAM_SERVER_TIMEOUT: Duration = Duration::from_secs(60 * 60 * 24);
```

Use Structured Logging with Message Templates (M-LOG-STRUCTURED)

Why this entry exists: To minimize the cost of logging and to improve filtering capabilities.
Version: 0.1

Logging should use structured events with named properties and message templates following the message templates specification.

> **Note:** Examples use the `tracing` crate's `event!` macro, but these principles apply to any logging API that supports structured logging (e.g., `log`, `slog`, custom telemetry systems).

Avoid String Formatting

String formatting allocates memory at runtime. Message templates defer formatting until viewing time. We recommend that message template includes all named properties for easier inspection at viewing time.

```rust
// Bad: String formatting causes allocations
tracing::info!("file opened: {}", path);
tracing::info!(format!("file opened: {}", path));

// Good: Message templates with named properties
event!(
    name: "file.open.success",
    Level::INFO,
    file.path = path.display(),
    "file opened: {{file.path}}",
);
```

Note: Use the `{{property}}` syntax in message templates which preserves the literal text while escaping Rust's format syntax. String formatting is deferred until logs are viewed.

Name Your Events

Use hierarchical dot-notation: `<component>.<operation>.<state>`

```rust
// Bad: Unnamed events
event!(
    Level::INFO,
    file.path = file_path,
    "file {{file.path}} processed succesfully",
);

// Good: Named events
event!(
    name: "file.processing.success", // event identifier
    Level::INFO,
    file.path = file_path,
    "file {{file.path}} processed succesfully",
);
```

Named events enable grouping and filtering across log entries.

Follow OpenTelemetry Semantic Conventions

Use OTel semantic conventions for common attributes if needed. This enables standardization and interoperability.

```rust
event!(
    name: "file.write.success",
    Level::INFO,
    file.path = path.display(),         // Standard OTel name
    file.size = bytes_written,          // Standard OTel name
    file.directory = dir_path,          // Standard OTel name
    file.extension = extension,         // Standard OTel name
    file.operation = "write",           // Custom name
    "{{file.operation}} {{file.size}} bytes to {{file.path}} in {{file.directory}}
extension={{file.extension}}",
);
```

Common conventions:

- HTTP: `http.request.method`, `http.response.status_code`, `url.scheme`, `url.path`, `server.address`
- File: `file.path`, `file.directory`, `file.name`, `file.extension`, `file.size`
- Database: `db.system.name`, `db.namespace`, `db.operation.name`, `db.query.text`
- Errors: `error.type`, `error.message`, `exception.type`, `exception.stacktrace`

Redact Sensitive Data

Do not log plain sensitive data as this might lead to privacy and security incidents.

```
// Bad: Logs potentially sensitive data
event!(
    name: "file.operation.started",
    Level::INFO,
    user.email = user.email,  // Sensitive data
    file.name = "license.txt",
    "reading file {{file.name}} for user {{user.email}}",
);

// Good: Redact sensitive parts
event!(
    name: "file.operation.started",
    Level::INFO,
    user.email.redacted = redact_email(user.email),
    file.name = "license.txt",
    "reading file {{file.name}} for user {{user.email.redacted}}",
);
```

Sensitive data includes email addresses, file paths revealing user identity, filenames containing secrets or tokens, file contents with PII, temporary file paths with session IDs and more. Consider using the `data_privacy` crate for consistent redaction.

Further Reading

- Message Templates Specification
- OpenTelemetry Semantic Conventions
- OWASP Logging Cheat Sheet

Library Guidelines

Guidelines for libraries. If your crate contains a `lib.rs` you should consider these:

- Interoperability
- UX
- Resilience
- Building

Libraries / Interoperability Guidelines

Types are Send (M-TYPES-SEND)

Why this entry exists:	To enable the use of types in Tokio and behind runtime abstractions
Version:	1.0

Public types should be `Send` for compatibility reasons:

- All futures produced (explicitly or implicitly) must be `Send`
- Most other types should be `Send`, but there might be exceptions

Futures

When declaring a future explicitly you should ensure it is, and remains, `Send`.

```
struct Foo {}

impl Future for Foo {
    // Explicit implementation of `Future` for your type
}

// You should assert your type is `Send`
const fn assert_send<T: Send>() {}
const _: () = assert_send::<Foo>();
```

When returning futures implicitly through `async` method calls, you should make sure these are `Send` too. You do not have to test every single method, but you should at least validate your main entry points.

```
async fn foo() { }

// TODO: We want this as a macro as well
fn assert_send<T: Send>(_: T) {}
_ = assert_send(foo());
```

Regular Types

Most regular types should be `Send`, as they otherwise infect futures turning them `!Send` if held across `.await` points.

```
async fn foo() {
    let rc = Rc::new(123);      // <-- Holding this across an .await point prevents
    read_file("foo.txt").await; //     the future from being `Send`.
    dbg!(rc);
}
```

That said, if the default use of your type is *instantaneous*, and there is no reason for it to be otherwise held across `.await` boundaries, it may be `!Send`.

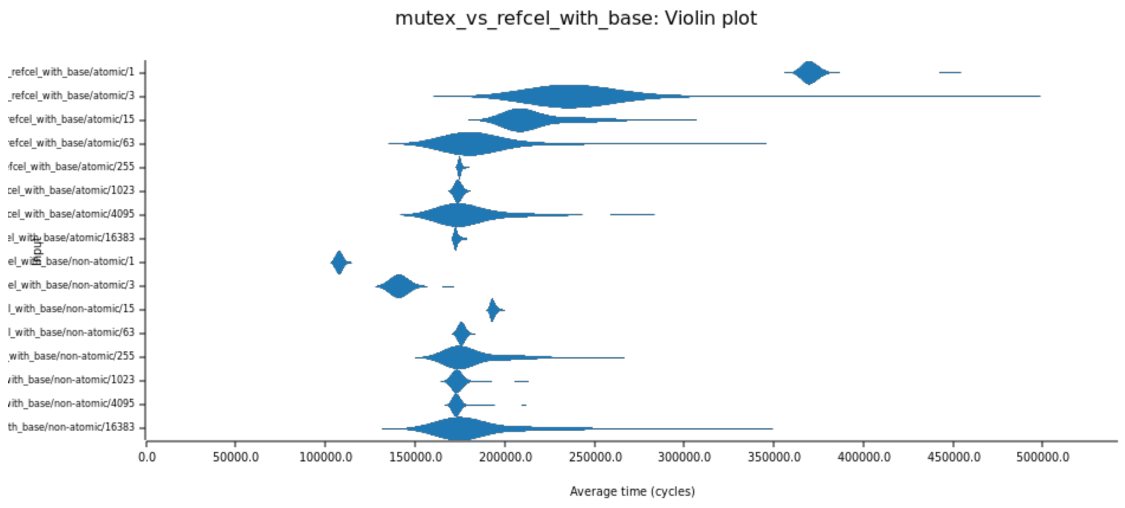
```
async fn foo() {
    // Here a hypothetical instance Telemetry is summoned
    // and used ad-hoc. It may be ok for Telemetry to be !Send.
    telemetry().ping(0);
    read_file("foo.txt").await;
    telemetry().ping(1);
}
```

💡 The Cost of Send

Ideally, there would be abstractions that are `Send` in work-stealing runtimes, and `!Send` in thread-per-core models based on non-atomic types like `Rc` and `RefCell` instead.

Practically these abstractions don't exist, preventing Tokio compatibility in the non-atomic case. That in turn means you would have to "reinvent the world" to get anything done in a thread-per-core universe.

The good news is, in most cases atomics and uncontended locks only have a measurable impact if accessed more frequently than every 64 words or so.



Working with a large `Vec<AtomicUsize>` in a hot loop is a bad idea, but doing the occasional uncontended atomic operation from otherwise thread-per-core async code has no performance impact, but gives you widespread ecosystem compatibility.

Native Escape Hatches (M-ESCAPE-HATCHES)

Why this entry exists:	To allow users to work around unsupported use cases until alternatives are available.
Version:	0.1

Types wrapping native handles should provide `unsafe` escape hatches. In interop scenarios your users might have gotten a native handle from somewhere else, or they might have to pass your wrapped handle over FFI. To enable these use cases you should provide `unsafe` conversion methods.

```
pub struct Handle(HNATIVE);

impl Handle {
    pub fn new() -> Self {
        // Safely creates handle via API calls
    }

    // Constructs a new Handle from a native handle the user got elsewhere.
    // This method  should then also document all safety requirements that
    // must be fulfilled.
    pub unsafe fn from_native(native: HNATIVE) -> Self {
        Self(native)
    }

    // Various extra methods to permanently or temporarily obtain
    // a native handle.
    pub fn into_native(self) -> HNATIVE { self.0 }
    pub fn to_native(&self) -> HNATIVE { self.0 }
}
```

Don't Leak External Types (M-DONT-LEAK-TYPES)

Why this entry exists:	To prevent accidental breakage and long-term maintenance cost.
Version:	0.1

Where possible, you should prefer `std` [1] types in public APIs over types coming from external crates. Exceptions should be carefully considered.

Any type in any public API will become part of that API's contract. Since `std` and constituents are the only crates shipped by default, and since they come with a permanent stability guarantee, their types are the only ones that come without an interoperability risk.

A crate that exposes another crate's type is said to *leak* that type.

For maximal long term stability your crate should, theoretically, not leak any types. Practically, some leakage is unavoidable, sometimes even beneficial. We recommend you follow this heuristic:

- ⬛ if you can avoid it, do not leak third-party types
- ⬛ if you are part of an umbrella crate,[2] you may freely leak types from sibling crates.
- ⬛ behind a relevant feature flag, types may be leaked (e.g., `serde`)
- ⬛ without a feature *only* if they give a *substantial benefit*. Most commonly that is interoperability with significant other parts of the Rust ecosystem based around these types.

[1] In rare instances, e.g., high performance libraries used from embedded, you might even want to limit yourself to `core` only.

[2] For example, a `runtime` crate might be the umbrella of `runtime_rt`, `runtime_app` and `runtime_clock` As users are expected to only interact with the umbrella, siblings may leak each others types.

Libraries / UX Guidelines

Abstractions Don't Visibly Nest (M-SIMPLE-ABSTRACTIONS)

Why this entry exists:	To prevent cognitive load and a bad out of the box UX.
Version:	0.1

When designing your public types and primary API surface, avoid exposing nested or complex parametrized types to your users.

While powerful, type parameters introduce a cognitive load, even more so if the involved traits are crate-specific. Type parameters become infectious to user code holding on to these types in their fields, often come with complex trait hierarchies on their own, and might cause confusing error messages.

From the perspective of a user authoring `Foo`, where the other structs come from your crate:

```rust
struct Foo {
    service: Service // Great
    service: Service<Backend> // Acceptable
    service: Service<Backend<Store>> // Bad

    list: List<Rc<u32>> // Great, `List<T>` is simple container,
                        // other types user provided.

    matrix: Matrix4x4 // Great
    matrix: Matrix4x4<f32> // Still ok
    matrix: Matrix<f32, Const<4>, Const<4>, ArrayStorage<f32, 4, 4>> // ?!?
}
```

Visible type parameters should be avoided in *service-like* types (i.e., types mainly instantiated once per thread / application that are often passed as dependencies), in particular if the nestee originates from the same crate as the service.

Containers, smart-pointers and similar data structures obviously must expose a type parameter, e.g., `List<T>` above. Even then, care should be taken to limit the number and nesting of parameters.

To decide whether type parameter nesting should be avoided, consider these factors:

- Will the type be **named** by your users?
    - Service-level types are always expected to be named (e.g., `Library<T>`),
    - Utility types, such as the many `std::iter` types like `Chain`, `Cloned`, `Cycle`, are not expected to be named.
- Does the type primarily compose with non-user types?
- Do the used type parameters have complex bounds?
- Do the used type parameters affect inference in other types or functions?

The more of these factors apply, the bigger the cognitive burden.

As a rule of thumb, primary service API types should not nest *on their own volition*, and if they do, only 1 level deep. In other words, these APIs should not require users having to deal with an `Foo<Bar<FooBar>>`. However, if `Foo<T>` users want to bring their own `A<B<C>>` as `T` they should be free to do so.

💡 Type Magic for Better UX?

The guideline above is written with 'bread-and-butter' types in mind you might create during *normal* development activity. Its intention is to reduce friction users encounter when working with your code.

However, when designing API patterns and ecosystems at large, there might be valid reasons to introduce intricate type magic to overall *lower* the cognitive friction involved, Bevy's ECS or Axum's request handlers come to mind.

The threshold where this pays off is high though. If there is any doubt about the utility of your creative use of generics, your users might be better off without them.

Avoid Smart Pointers and Wrappers in APIs (M-AVOID-WRAPPERS)

Why this entry exists:	To reduce cognitive load and improve API ergonomics.
Version:	1.0

As a specialization of M-ABSTRACTIONS-DONT-NEST, generic wrappers and smart pointers like `Rc<T>`, `Arc<T>`, `Box<T>`, or `RefCell<T>` should be avoided in public APIs.

From a user perspective these are mostly implementation details, and introduce infectious complexity that users have to resolve. In fact, these might even be impossible to resolve once multiple crates disagree about the required type of wrapper.

If wrappers are needed internally, they should be hidden behind a clean API that uses simple types like `&T`, `&mut T`, or `T` directly. Compare:

```rust
// Good: simple API
pub fn process_data(data: &Data) -> State { ... }
pub fn store_config(config: Config) -> Result<(), Error> { ... }

// Bad: Exposing implementation details
pub fn process_shared(data: Arc<Mutex<Shared>>) -> Box<Processed> { ... }
pub fn initialize(config: Rc<RefCell<Config>>) -> Arc<Server> { ... }
```

Smart pointers in APIs are acceptable when:

- The smart pointer is fundamental to the API's purpose (e.g., a new container lib)
- The smart pointer, based on benchmarks, significantly improves performance and the complexity is justified.

Prefer Types over Generics, Generics over Dyn Traits (M-DI-HIERARCHY)

Why this entry exists: To prevent patterns that don't compose, and design lock-in.
Version: 0.1

When asking for async dependencies, prefer concrete types over generics, and generics over `dyn Trait`.

It is easy to accidentally deviate from this pattern when porting code from languages like C# that heavily rely on interfaces. Consider you are porting a service called `Database` from C# to Rust and, inspired by the original `IDatabase` interface, you naively translate it into:

```
trait Database {
    async fn update_config(&self, file: PathBuf);
    async fn store_object(&self, id: Id, obj: Object);
    async fn load_object(&self, id: Id) -> Object;
}

impl Database for MyDatabase { ... }

// Intended to be used like this:
async fn start_service(b: Rc<dyn Database>) { ... }
```

Apart from not feeling idiomatic, this approach precludes other Rust constructs that conflict with object safety, can cause issues with asynchronous code, and exposes wrappers (compare M-AVOID-WRAPPERS).

Instead, when more than one implementation is needed, this *design escalation ladder* should be followed:

If the other implementation is only concerned with providing a *sans-io* implementation for testing, implement your type as an enum, following M-MOCKABLE-SYSCALLS instead.

If users are expected to provide custom implementations, you should introduce one or more traits, and implement them for your own types *on top* of your inherent functions. Each trait should be relatively narrow, e.g., `StoreObject`, `LoadObject`. If eventually a single trait is needed it should be made a subtrait, e.g., `trait DataAccess: StoreObject + LoadObject {}`.

Code working with these traits should ideally accept them as generic type parameters as long as their use does not contribute to significant nesting (compare M-ABSTRACTIONS-DONT-NEST).

```
// Good, generic does not have infectious impact, uses only most specific trait
async fn read_database(x: impl LoadObject) { ... }

// Acceptable, unless further nesting makes this excessive.
struct MyService<T: DataAccess> {
    db: T,
}
```

Once generics become a nesting problem, `dyn Trait` can be considered. Even in this case, visible wrapping should be avoided, and custom wrappers should be preferred.

```rust
// This allows you to expand or change `DynamicDataAccess` later. You can also
// implement `DataAccess` for `DynamicDataAccess` if needed, and use it with
// regular generic functions.
struct DynamicDataAccess(Arc<dyn DataAccess>);

impl DynamicDataAccess {
    fn new<T: DataAccess + 'static>(db: T) -> Self {
        Self(Arc::new(db))
    }
}

struct MyService {
    db: DynamicDataAccess,
}
```

The generic wrapper can also be combined with the enum approach from M-MOCKABLE-SYSCALLS:

```rust
enum DataAccess {
    MyDatabase(MyDatabase),
    Mock(mock::MockCtrl),
    Dynamic(DynamicDataAccess)
}

async fn read_database(x: &DataAccess) { ... }
```

Error are Canonical Structs (M-ERRORS-CANONICAL-STRUCTS)

Why this entry exists: To harmonize the behavior of error types, and provide a consistent error handling.
Version: 1.0

Errors should be a situation-specific `struct` that contain a `Backtrace`, a possible upstream error cause, and helper methods.

Simple crates usually expose a single error type `Error`, complex crates may expose multiple types, for example `AccessError` and `ConfigurationError`. Error types should provide helper methods for additional information that allows callers to handle the error.

A simple error might look like so:

```rust
pub struct ConfigurationError {
    backtrace: Backtrace,
}

impl ConfigurationError {
    pub(crate) fn new() -> Self {
        Self { backtrace: Backtrace::capture() }
    }
}

// Impl Debug + Display
```

Where appropriate, error types should provide contextual error information, for example:

```rust
impl ConfigurationError {
    pub fn config_file(&self) -> &Path { }
}
```

If your API does mixed operations, or depends on various upstream libraries, store an `ErrorKind`. Error kinds, and more generally enum-based errors, should not be used to avoid creating separate public error types when there is otherwise no error overlap:

```rust
// Prefer this
fn download_iso() -> Result<(), DownloadError> {}
fn start_vm() -> Result<(), VmError> {}

// Over that
fn download_iso() -> Result<(), GlobalEverythingErrorEnum> {}
fn start_vm() -> Result<(), GlobalEverythingErrorEnum> {}

// However, not every function warrants a new error type. Errors
// should be general enough to be reused.
fn parse_json() -> Result<(), ParseError> {}
fn parse_toml() -> Result<(), ParseError> {}
```

If you do use an inner `ErrorKind`, that enum should not be exposed directly for future-proofing reasons, as otherwise you would expose your callers to *all* possible failure modes, even the ones you consider internal and unhandleable. Instead, expose various `is_xxx()` methods as shown below:

```rust
#[derive(Debug)]
pub(crate) enum ErrorKind {
    Io(std::io::Error),
    Protocol
}

#[derive(Debug)]
pub struct HttpError {
    kind: ErrorKind,
    backtrace: Backtrace,
}

impl HttpError {
    pub fn is_io(&self) -> bool { matches!(self.kind, ErrorKind::Io(_)) }
    pub fn is_protocol(&self) -> bool { matches!(self.kind, ErrorKind::Protocol) }
}
```

Most upstream errors don't provide a backtrace. You should capture one when creating an `Error` instance, either via one of your `Error::new()` flavors, or when implementing `From<UpstreamError> for Error {}`.

Error structs must properly implement `Display` that renders as follows:

```rust
impl Display for MyError {
    // Print a summary sentence what happened.
    // Print `self.backtrace`.
    // Print any additional upstream 'cause' information you might have.
}
```

Errors must also implement `std::error::Error`:

```rust
impl std::error::Error for MyError { }
```

Lastly, if you happen to emit lots of errors from your crate, consider creating a private `bail!()` helper macro to simplify error instantiation.

💡 When You Get Backtraces

Backtraces are an invaluable debug tool in complex or async code, since errors might *travel* far through a callstack before being surfaced.

That said, they are a *development* tool, not a *runtime* diagnostic, and by default `Backtrace::capture()` will **not** capture backtraces, as they have a large overhead, e.g., 4μs per capture on the author's PC.

Instead, Rust evaluates a set of environment variables, such as `RUST_BACKTRACE`, and only walks the call frame when explicitly asked. Otherwise it captures an empty trace, at the cost of only a few CPU instructions.

Complex Type Construction has Builders (M-INIT-BUILDER)

Why this entry exists: To future-proof type construction in complex scenarios.
Version: 0.3

Types that could support 4 or more arbitrary initialization permutations should provide builders. In other words, types with up to 2 optional initialization parameters can be constructed via inherent methods:

```rust
struct Foo;

// Supports 2 optional construction parameters, inherent methods ok.
impl Foo {
    pub fn new() -> Self { Self }
    pub fn with_a(a: A) -> Self { Self }
    pub fn with_b(b: B) -> Self { Self }
    pub fn with_a_b(a: A, b: B) -> Self { Self }
}
```

Beyond that, types should provide a builder:

```rust
impl Foo {
    pub fn new() -> Self { ... }
    pub fn builder() -> FooBuilder { ... }
}

impl FooBuilder {
    pub fn a(mut self, a: A) -> Self { ... }
    pub fn b(mut self, b: B) -> Self { ... }
    pub fn c(mut self, c: C) -> Self { ... }
    pub fn build(self) -> Foo { ... }
}
```

The proper name for a builder that builds `Foo` is `FooBuilder`. Its methods must be chainable, with the final method called `.build()`. The buildable struct must have a shortcut `Foo::builder()`, while the builder itself should *not* have a public `FooBuilder::new()`. Builder methods that set a value `x` are called `x()`, not `set_x()` or similar.

Builders and Required Parameters

Required parameters should be passed when creating the builder, not as setter methods. For builders with multiple required parameters, encapsulate them into a parameters struct and use the `deps: impl Into<Deps>` pattern to provide flexibility:

Note: A dedicated deps struct is not required if the builder has no required parameters or only a single simple parameter. However, for backward compatibility and API evolution, it's preferable to use a dedicated struct for deps even in simple cases, as it makes it easier to add new required parameters in the future without breaking existing code.

```rust
#[derive(Debug, Clone)]
pub struct FooDeps {
    pub logger: Logger,
    pub config: Config,
}

impl From<(Logger, Config)> for FooDeps { ... }
impl From<Logger> for FooDeps { ... } // In case we could use default Config instance

impl Foo {
    pub fn builder(deps: impl Into<FooDeps>) -> FooBuilder { ... }
}
```

This pattern allows for convenient usage:

- `Foo::builder(logger)` - when only the logger is needed
- `Foo::builder((logger, config))` - when both parameters are needed
- `Foo::builder(FooDeps { logger, config })` - explicit struct construction

Alternatively, you can use `fundle` to simplify the creation of `FooDeps`:

```rust
#[derive(Debug, Clone)]
#[fundle::deps]
pub struct FooDeps {
    pub logger: Logger,
    pub config: Config,
}
```

This pattern enables "dependency injection", see these docs for more details.

Runtime-Specific Builders

For types that are runtime-specific or require runtime-specific configuration, provide dedicated builder creation methods that accept the appropriate runtime parameters:

```
#[cfg(feature="smol")]
#[derive(Debug, Clone)]
pub struct SmolDeps {
    pub clock: Clock,
    pub io_context: Context,
}

#[cfg(feature="tokio")]
#[derive(Debug, Clone)]
pub struct TokioDeps {
    pub clock: Clock,
}

impl Foo {
    #[cfg(feature="smol")]
    pub fn builder_smol(deps: impl Into<SmolDeps>) -> FooBuilder { ... }

    #[cfg(feature="tokio")]
    pub fn builder_tokio(deps: impl Into<TokioDeps>) -> FooBuilder { ... }
}
```

This approach ensures type safety at compile time and makes the runtime dependency explicit in the API surface. The resulting builder methods follow the pattern `builder_{runtime}(deps)` where `{runtime}` indicates the specific runtime or execution environment.

Further Reading

- Builder pattern in Rust: self vs. &mut self, and method vs. associated function
- fundle

Complex Type Initialization Hierarchies are Cascaded (M-INIT-CASCADED)

Why this entry exists: To prevent misuse and accidental parameter mix ups.
Version: 1.0

Types that require 4+ parameters should cascade their initialization via helper types.

```
impl Deposit {
    // Easy to confuse parameters and signature generally unwieldy.
    pub fn new(bank_name: &str, customer_name: &str, currency_name: &str,
currency_amount: u64) -> Self { }
}
```

Instead of providing a long parameter list, parameters should be grouped semantically. When applying this guideline, also check if C-NEWTYPE is applicable:

```rust
impl Deposit {
    // Better, signature cleaner
    pub fn new(account: Account, amount: Currency) -> Self { }
}

impl Account {
    pub fn new_ok(bank: &str, customer: &str) -> Self { }
    pub fn new_even_better(bank: Bank, customer: Customer) -> Self { }
}
```

Services are Clone (M-SERVICES-CLONE)

Why this entry exists: To avoid composability issues when sharing common services.
Version: 1.0

Heavyweight *service* types and 'thread singletons' should implement shared-ownership `Clone` semantics, including any type you expect to be used from your `Application::init`.

Per thread, users should essentially be able to create a single resource handler instance, and have it reused by other handlers on the same thread:

```rust
impl ThreadLocal for MyThreadState {
    fn init(...) -> Self {

        // Create common service instance possibly used by many.
        let common = ServiceCommon::new();

        // Users can freely pass `common` here multiple times
        let service_1 = ServiceA::new(&common);
        let service_2 = ServiceA::new(&common);

        Self { ... }
    }
}
```

Services then simply clone their dependency and store a new *handle*, as if `ServiceCommon` were a shared-ownership smart pointer:

```rust
impl ServiceA {
    pub fn new(common: &ServiceCommon) -> Self {
        // If we only need to access `common` from `new` we don't have
        // to store it. Otherwise, make a clone we store in `Self`.
        let common = common.clone();
    }
}
```

Under the hood this `Clone` should **not** create a fat copy of the entire service. Instead, it should follow the `Arc<Inner>` pattern:

```
// Actual service containing core logic and data.
struct ServiceCommonInner {}

#[derive(Clone)]
pub ServiceCommon {
    inner: Arc<ServiceCommonInner>
}

impl ServiceCommon {
    pub fn new() {
        Self { inner: Arc::new(ServiceCommonInner::new()) }
    }

    // Method forwards ...
    pub fn foo(&self) { self.inner.foo() }
    pub fn bar(&self) { self.inner.bar() }
}
```

Accept `impl AsRef<>` Where Feasible (M-IMPL-ASREF)

Why this entry exists:	To give users flexibility calling in with their own types.
Version:	1.0

In **function** signatures, accept `impl AsRef<T>` for types that have a clear reference hierarchy, where you do not need to take ownership, or where object creation is relatively cheap.

Instead of ...	accept ...
`&str`, `String`	`impl AsRef<str>`
`&Path`, `PathBuf`	`impl AsRef<Path>`
`&[u8]`, `Vec<u8>`	`impl AsRef<[u8]>`

```
// Definitely use `AsRef`, the function does not need ownership.
fn print(x: impl AsRef<str>) {}
fn read_file(x: impl AsRef<Path>) {}
fn send_network(x: impl AsRef<[u8]>) {}

// Further analysis needed. In these cases the function wants
// ownership of some `String` or `Vec<u8>`. If those are
// "low freqency, low volume" functions `AsRef` has better ergonomics,
// otherwise accepting a `String` or `Vec<u8>` will have better
// performance.
fn new_instance(x: impl AsRef<str>) -> HoldsString {}
fn send_to_other_thread(x: impl AsRef<[u8]>) {}
```

In contrast, **types** should generally not be infected by these bounds:

```
// Generally not ok. There might be exceptions for performance
// reasons, but those should not be user visible.
struct User<T: AsRef<str>> {
    name: T
}

// Better
struct User {
    name: String
}
```

Accept `impl RangeBounds<>` Where Feasible (M-IMPL-RANGEBOUNDS)

Why this entry exists:	To give users flexibility and clarity when specifying ranges.
Version:	1.0

Functions that accept a range of numbers must use a `Range` type or trait over hand-rolled parameters:

```
// Bad
fn select_range(low: usize, high: usize) {}
fn select_range(range: (usize, usize)) {}
```

In addition, functions that can work on arbitrary ranges, should accept `impl RangeBounds<T>` rather than `Range<T>`.

```
// Callers must call with `select_range(1..3)`
fn select_range(r: Range<usize>) {}

// Callers may call as
//     select_any(1..3)
//     select_any(1..)
//     select_any(..)
fn select_any(r: impl RangeBounds<usize>) {}
```

Accept `impl 'IO'` Where Feasible ('Sans IO') (M-IMPL-IO)

Why this entry exists:	To untangle business logic from I/O logic, and have N*M composability.
Version:	0.1

Functions and types that only need to perform one-shot I/O during initialization should be written "sans-io", and accept some `impl T`, where `T` is the appropriate I/O trait, effectively outsourcing I/O work to another type:

```rust
// Bad, caller must provide a File to parse the given data. If this
// data comes from the network, it'd have to be written to disk first.
fn parse_data(file: File) {}


// Much better, accepts
// - Files,
// - TcpStreams,
// - Stdin,
// - &[u8],
// - UnixStreams,
// ... and many more.
fn parse_data(data: impl std::io::Read) {}
```

Synchronous functions should use `std::io::Read` and `std::io::Write`. Asynchronous *functions* targeting more than one runtime should use `futures::io::AsyncRead` and similar. *Types* that need to perform runtime-specific, continuous I/O should follow M-RUNTIME-ABSTRACTED instead.

Essential Functionality Should be Inherent (M-ESSENTIAL-FN-INHERENT)

Why this entry exists:	To make essential functionality easily discoverable.
Version:	1.0

Types should implement core functionality inherently. Trait implementations should forward to inherent functions, and not replace them. Instead of this

```rust
struct HttpClient {}

// Offloading essential functionality into traits means users
// will have to figure out what other traits to `use` to
// actually use this type.
impl Download for HttpClient {
    fn download_file(&self, url: impl AsRef<str>) {
        // ... logic to download a file
    }
}
```

do this:

```rust
struct HttpClient {}

impl HttpClient {
    fn download_file(&self, url: impl AsRef<str>) {
        // ... logic to download a file
    }
}

// Forward calls to inherent impls. `HttpClient` can be used
impl Download for HttpClient {
    fn download_file(&self, url: impl AsRef<str>) {
        Self::download_file(self, url)
    }
}
```

Libraries / Resilience Guidelines

I/O and System Calls Are Mockable (M-MOCKABLE-SYSCALLS)

Why this entry exists:	To make otherwise hard-to-evoke edge cases testable.
Version:	0.2

Any user-facing type doing I/O, or sys calls with side effects, should be mockable to these effects. This includes file and network access, clocks, entropy sources and seeds, and similar. More generally, any operation that is

- non-deterministic,
- reliant on external state,
- depending on the hardware or the environment,
- is otherwise fragile or not universally reproducible

should be mockable.

💡 Mocking Allocations?

Unless you write kernel code or similar, you can consider allocations to be deterministic, hardware independent and practically infallible, thus not covered by this guideline.

However, this does *not* mean you should expect there to be unlimited memory available. While it is ok to accept caller provided input as-is if your library has a *reasonable* memory complexity, memory-hungry libraries and code handling external input should provide bounded and / or chunking operations.

This guideline has several implications for libraries, they

- should not perform ad-hoc I/O, i.e., call `read("foo.txt")`
- should not rely on non-mockable I/O and sys calls
- should not create their own I/O or sys call *core* themselves
- should not offer `MyIoLibrary::default()` constructors

Instead, libraries performing I/O and sys calls should either accept some I/O *core* that is mockable already, or provide mocking functionality themselves:

```
let lib = Library::new_runtime(runtime_io); // mockable I/O functionality passed in
let (lib, mock) = Library::new_mocked(); // supports inherent mocking
```

Libraries supporting inherent mocking should implement it as follows:

```rust
pub struct Library {
    some_core: LibraryCore // Encapsulates syscalls, I/O, ... compare below.
}

impl Library {
    pub fn new() -> Self { ... }
    pub fn new_mocked() -> (Self, MockCtrl) { ... }
}
```

Behind the scenes, `LibraryCore` is a non-public enum, similar to M-RUNTIME-ABSTRACTED, that either dispatches calls to the respective sys call, or to an mocking controller.

```rust
// Dispatches calls either to the operating system, or to a
// mocking controller.
enum LibraryCore {
    Native,

    #[cfg(feature = "test-util")]
    Mocked(mock::MockCtrl)
}

impl LibraryCore {
    // Some function you'd forward to the operating system.
    fn random_u32(&self) {
        match self {
            Self::Native => unsafe { os_random_u32() }
            Self::Mocked(m) => m.random_u32()
        }
    }
}


#[cfg(feature = "test-util")]
mod mock {
    // This follows the M-SERVICES-CLONE pattern, so both `LibraryCore` and
    // the user can hold on to the same `MockCtrl` instance.
    pub struct MockCtrl {
        inner: Arc<MockCtrlInner>
    }

    // Implement required logic accordingly, usually forwarding to
    // `MockCtrlInner` below.
    impl MockCtrl {
        pub fn set_next_u32(&self, x: u32) { ... }
        pub fn random_u32(&self) { ... }
    }

    // Contains actual logic, e.g., the next random number we should return.
    struct MockCtrlInner {
        next_call: u32
    }
}
```

Runtime-aware libraries already build on top of the M-RUNTIME-ABSTRACTED pattern should extend their runtime enum instead:

```
enum Runtime {
    #[cfg(feature="tokio")]
    Tokio(tokio::Tokio),

    #[cfg(feature="smol")]
    Smol(smol::Smol)

    #[cfg(feature="test-util")]
    Mock(mock::MockCtrl)
}
```

As indicated above, most libraries supporting mocking should not accept mock controllers, but return them via parameter tuples, with the first parameter being the library instance, the second the mock controller. This is to prevent state ambiguity if multiple instances shared a single controller:

```
impl Library {
    pub fn new_mocked() -> (Self, MockCtrl) { ... } // good
    pub fn new_mocked_bad(&mut MockCtrl) -> Self { ... } // prone to misuse
}
```

Test Utilities are Feature Gated (M-TEST-UTIL)

Why this entry exists: To prevent production builds from accidentally bypassing safety checks.
Version: 0.2

Testing functionality must be guarded behind a feature flag. This includes

- mocking functionality (M-MOCKABLE-SYSCALLS),
- the ability to inspect sensitive data,
- safety check overrides,
- fake data generation.

We recommend you use a single flag only, named `test-util`. In any case, the feature(s) must clearly communicate they are for testing purposes.

```
impl HttpClient {
    pub fn get() { ... }

    #[cfg(feature = "test-util")]
    pub fn bypass_certificate_checks() { ... }
}
```

Use the Proper Type Family (M-STRONG-TYPES)

Why this entry exists: To have and maintain the right data and safety variants, at the right time.
Version: 1.0

Use the appropriate `std` type for your task. In general you should use the strongest type available, as early as possible in your API flow. Common offenders are

Do not use ...	use instead ...	Explanation
`String` *	`PathBuf` *	Anything dealing with the OS should be `Path` -like

That said, you should also follow common Rust `std` conventions. Purely numeric types at public API boundaries (e.g., `window_size()`) are expected to be regular numbers, not `Saturating<usize>` , `NonZero<usize>` , or similar.

* Including their siblings, e.g., `&str` , `Path` , ...

Don't Glob Re-Export Items (M-NO-GLOB-REEXPORTS)

Why this entry exists: To prevent accidentally leaking unintended types.
Version: 1.0

Don't `pub use foo::*` from other modules, especially not from other crates. You might accidentally export more than you want, and globs are hard to review in PRs. Re-export items individually instead:

```
pub use foo::{A, B, C};
```

Glob exports are permissible for technical reasons, like doing platform specific re-exports from a set of HAL (hardware abstraction layer) modules:

```
#[cfg(target_os = "windows")]
mod windows { /* ... */ }

#[cfg(target_os = "linux")]
mod linux { /* ... */ }

// Acceptable use of glob re-exports, this is a common pattern
// and it is clear everything is just forwarded from a single
// platform.

#[cfg(target_os = "windows")]
pub use windows::*;

#[cfg(target_os = "linux")]
pub use linux::*;
```

Avoid Statics (M-AVOID-STATICS)

Why this entry exists: To prevent consistency and correctness issues between crate versions.
Version: 1.0

Libraries should avoid `static` and thread-local items, if a consistent view of the item is relevant for correctness. Essentially, any code that would be incorrect if the static *magically* had another value must not use them. Statics only used for performance optimizations are ok.

The fundamental issue with statics in Rust is the secret duplication of state.

Consider a crate `core` with the following function:

```rust
static GLOBAL_COUNTER: AtomicUsize = AtomicUsize::new(0);

pub fn increase_counter() -> usize {
    GLOBAL_COUNTER.fetch_add(1, Ordering::Relaxed)
}
```

Now assume you have a crate `main`, calling two libraries `library_a` and `library_b`, each invoking that counter:

```rust
// Increase global static counter 2 times
library_a::count_up();
library_a::count_up();

// Increase global static counter 3 more times
library_b::count_up();
library_b::count_up();
library_b::count_up();
```

They eventually report their result:

```rust
library_a::print_counter();
library_b::print_counter();
main::print_counter();
```

At this point, what is *the* value of said counter; `0`, `2`, `3` or `5`?

The answer is, possibly any (even multiple!) of the above, depending on the crate's version resolution!

Under the hood Rust may link to multiple versions of the same crate, independently instantiated, to satisfy declared dependencies. This is especially observable during a crate's `0.x` version timeline, where each `x` constitutes a separate *major* version.

If `main`, `library_a` and `library_b` all declared the same version of `core`, e.g. `0.5`, then the reported result will be `5`, since all crates actually *see* the same version of `GLOBAL_COUNTER`.

However, if `library_a` declared `0.4` instead, then it would be linked against a separate version of `core`; thus `main` and `library_b` would agree on a value of `3`, while `library_a` reported `2`.

Although `static` items can be useful, they are particularly dangerous before a library's stabilization, and for any state where *secret duplication* would cause consistency issues when static and non-static variable use interacts. In addition, statics interfere with unit testing, and are a contention point in thread-per-core designs.

Libraries / Building Guidelines

Libraries Work Out of the Box (M-OOBE)

Why this entry exists: To be easily adoptable by the Rust ecosystem.
Version: 1.0

Libraries must *just work* on all supported platforms, with the exception of libraries that are expressly platform or target specific.

Rust crates often come with dozens of dependencies, applications with 100's. Users expect `cargo build` and `cargo install` to *just work*. Consider this installation of `bat` that pulls in ~250 dependencies:

```
Compiling writeable v0.5.5
Compiling strsim v0.11.1
Compiling litemap v0.7.5
Compiling crossbeam-utils v0.8.21
Compiling icu_properties_data v1.5.1
Compiling ident_case v1.0.1
Compiling once_cell v1.21.3
Compiling icu_normalizer_data v1.5.1
Compiling fnv v1.0.7
Compiling regex-syntax v0.8.5
Compiling anstyle v1.0.10
Compiling vcpkg v0.2.15
Compiling utf8parse v0.2.2
Compiling aho-corasick v1.1.3
Compiling utf16_iter v1.0.5
Compiling hashbrown v0.15.2
Building [==>                       ] 29/251: icu_locid_transform_data, serde, winnow,
indexma...
```

This compilation, like practically all other applications and libraries, will *just work*.

While there are tools targeting specific functionality (e.g., a Wayland compositor) or platform crates like `windows`; unless a crate is *obviously* platform specific, the expectation is that it will otherwise *just work*.

This means crates must build, ultimately

- ■ on all Tier 1 platforms,[1] and
- ■ without any additional prerequisites beyond `cargo` and `rust`.[2]

[1] It is ok to not support Tier 1 platforms "for now", but abstractions must be present so support can easily be extended. This is usually done by introducing an internal `HAL` (Hardware Abstraction Layer) module with a `dummy` fallback target.

[2] A default Rust installation will also have `cc` and a linker present.

In particular, non-platform crates must not, by default, require the user to install additional tools, or expect environment variables to compile. If tools were somehow needed (like the generation of Rust from `.proto` files) these tools should be run as part of the publishing workflow or earlier, and the resulting artifacts (e.g., `.rs` files) be contained inside the published crate.

If a dependency is known to be platform specific, the parent must use conditional (platform) compilation or opt-in feature gates.

⚠️**Libraries are Responsible for Their Dependencies.**

Imagine you author a `Copilot` crate, which in turn uses an `HttpClient`, which in turn depends on a `perl` script to compile.

Then every one of your users, and your user's users, and everyone above, would need to install Perl to compile *their* crate. In large projects you would have 100's of people who don't know or don't care about your library or Perl, encounter a cryptic compilation error, and now have to figure out how to install it on their system.

In practical terms, such behavior is largely a self-inflicted death sentence in the open source space, since the moment alternatives are available, people will switch to those that *just work*.

Native `-sys` Crates Compile Without Dependencies (M-SYS-CRATES)

Why this entry exists:	To have libraries that 'just work' on all platforms.
Version:	0.2

If you author a pair of `foo` and `foo-sys` crates wrapping a native `foo.lib`, you are likely to run into the issues described in M-OOBE.

Follow these steps to produce a crate that *just works* across platforms:

- ⬛ fully govern the build of `foo.lib` from `build.rs` inside `foo-sys`. Only use hand-crafted compilation via the cc crate, do *not* run Makefiles or external build scripts, as that will require the installation of external dependencies,
- ⬛ make all external tools optional, such as `nasm`,
- ⬛ embed the upstream source code in your crate,
- ⬛ make the embedded sources verifiable (e.g., include Git URL + hash),
- ⬛ pre-generate `bindgen` glue if possible,
- ⬛ support both static linking, and dynamic linking via libloading.

Deviations from these points can work, and can be considered on a case-by-case basis:

If the native build system is available as an *OOBE* crate, that can be used instead of `cc` invocations. The same applies to external tools.

Source code might have to be downloaded if it does not fit crates.io size limitations. In any case, only servers with an availability comparable to crates.io should be used. In addition, the specific hashes of acceptable downloads should be stored in the crate and verified.

Downloading sources can fail on hermetic build environments, therefore alternative source roots should also be specifiable (e.g., via environment variables).

Features are Additive (M-FEATURES-ADDITIVE)

Why this entry exists:	To prevent compilation breakage in large and complex projects.
Version:	1.0

All library features must be additive, and any combination must work, as long as the feature itself would work on the current platform. This implies:

- ■ You must not introduce a `no-std` feature, use a `std` feature instead
- ■ Adding any feature `foo` must not disable or modify any public item
  - Adding enum variants is fine if these enums are `#[non_exhaustive]`
- ■ Features must not rely on other features to be manually enabled
- ■ Features must not rely on their parent to skip-enable a feature in one of their children

Further Reading

- Feature Unification
- Mutually Exclusive Features

Application Guidelines

Use Mimalloc for Apps (M-MIMALLOC-APPS)

Why this entry exists:	To get significant performance for free.
Version:	0.1

Applications should set mimalloc as their global allocator. This usually results in notable performance increases along allocating hot paths; we have seen up to 25% benchmark improvements.

Changing the allocator only takes a few lines of code. Add mimalloc to your `Cargo.toml` like so:

```
[dependencies]
mimalloc = { version = "0.1" } # Or later version if available
```

Then use it from your `main.rs`:

```
use mimalloc::MiMalloc;

#[global_allocator]
static GLOBAL: MiMalloc = MiMalloc;
```

Applications may use Anyhow or Derivatives (M-APP-ERROR)

Why this entry exists:	To simplify application-level error handling.
Version:	0.1

> Note, this guideline is primarily a relaxation and clarification of M-ERRORS-CANONICAL-STRUCTS.

Applications, and crates in your own repository exclusively used from your application, may use anyhow, eyre or similar application-level error crates instead of implementing their own types.

For example, in your application crates you may just re-export and use eyre's common `Result` type, which should be able to automatically handle all third party library errors, in particular the ones following M-ERRORS-CANONICAL-STRUCTS.

```rust
use eyre::Result;

fn start_application() -> Result<()> {
    start_server()?;
    Ok(())
}
```

Once you selected your application error crate you should switch all application-level errors to that type, and you should not mix multiple application-level error types.

Libraries (crates used by more than one crate) should always follow M-ERRORS-CANONICAL-STRUCTS instead.

FFI Guidelines

Isolate DLL State Between FFI Libraries (M-ISOLATE-DLL-STATE)

Why this entry exists:	To prevent data corruption and undefined behavior.
Version:	0.1

When loading multiple Rust-based dynamic libraries (DLLs) within one application, you may only share 'portable' state between these libraries. Likewise, when authoring such libraries, you must only accept or provide 'portable' data from foreign DLLs.

Portable here means data that is safe and consistent to process regardless of its origin. By definition, this is a subset of FFI-safe types. A type is portable if it is `#[repr(C)]` (or similarly well-defined), and *all* of the following:

- It must not have any interaction with any `static` or thread local.
- It must not have any interaction with any `TypeId`.
- It must not contain any value, pointer or reference to any non-portable data (it is valid to point into portable data within non-portable data, such as sharing a reference to an ASCII string held in a `Box`).

Interaction means any computational relationship, and therefore also relates to how the type is used. Sending a `u128` between DLLs is OK, using it to exchange a transmuted `TypeId` isn't.

The underlying issue stems from the Rust compiler treating each DLL as an entirely new compilation artifact, akin to a standalone application. This means each DLL:

- has its own set of `static` and thread-local variables,
- the type layout of any `#[repr(Rust)]` type (the default) can differ between compilations,
- has its own set of unique type IDs, differing from any other DLL.

Notably, this affects:

- ⚠️any allocated instance, e.g., `String`, `Vec<u8>`, `Box<Foo>`, ...
- ⚠️any library relying on other statics, e.g., `tokio`, `log`,
- ⚠️any struct not `#[repr(C)]`,
- ⚠️any data structure relying on consistent `TypeId`.

In practice, transferring any of the above between libraries leads to data loss, state corruption, and usually undefined behavior.

Take particular note that this may also apply to types and methods that are invisible at the FFI boundary:

```rust
/// A method in DLL1 that wants to use a common service from DLL2
#[ffi_function]
fn use_common_service(common: &CommonService) {
    // This has at least two issues:
    // - `CommonService`, or ANY type nested deep within might have
    //   a different type layout in DLL2, leading to immediate
    //   undefined behavior (UB) ⚠️
    // - `do_work()` here looks like it will be invoked in DLL2, but
    //   the code executed will actually come from DLL1. This means that
    //   `do_work()` invoked here will see a data structure coming from
    //   DLL2, but will use statics from DLL1 ⚠️
    common.do_work();
}
```

Safety Guidelines

Unsafe Needs Reason, Should be Avoided (M-UNSAFE)

Why this entry exists:	To prevent undefined behavior, attack surface, and similar 'happy little accidents'.
Version:	0.2

You must have a valid reason to use `unsafe`. The only valid reasons are

1. novel abstractions, e.g., a new smart pointer or allocator,
2. performance, e.g., attempting to call `.get_unchecked()`,
3. FFI and platform calls, e.g., calling into C or the kernel, ...

Unsafe code lowers the guardrails used by the compiler, transferring some of the compiler's responsibilities to the programmer. Correctness of the resulting code relies primarily on catching all mistakes in code review, which is error-prone. Mistakes in unsafe code may introduce high-severity security vulnerabilities.

You must not use ad-hoc `unsafe` to

- shorten a performant and safe Rust program, e.g., 'simplify' enum casts via `transmute`,
- bypass `Send` and similar bounds, e.g., by doing `unsafe impl Send ...`,
- bypass lifetime requirements via `transmute` and similar.

Ad-hoc here means `unsafe` embedded in otherwise unrelated code. It is of course permissible to create properly designed, sound abstractions doing these things.

In any case, `unsafe` must follow the guidelines outlined below.

Novel Abstractions

- ⬛ Verify there is no established alternative. If there is, prefer that.
- ⬛ Your abstraction must be minimal and testable.
- ⬛ It must be hardened and tested against "adversarial code", esp.
  - If they accept closures they must become invalid (e.g., poisoned) if the closure panics
  - They must assume any safe trait is misbehaving, esp. `Deref`, `Clone` and `Drop`.
- ⬛ Any use of `unsafe` must be accompanied by plain-text reasoning outlining its safety
- ⬛ It must pass Miri, including adversarial test cases
- ⬛ It must follow all other unsafe code guidelines

Performance

- ⬛ Using `unsafe` for performance reasons should only be done after benchmarking
- ⬛ Any use of `unsafe` must be accompanied by plain-text reasoning outlining its safety. This applies to both calling `unsafe` methods, as well as providing `_unchecked` ones.

- ■ The code in question must pass Miri
- ■ You must follow the unsafe code guidelines

FFI

- ■ We recommend you use an established interop library to avoid `unsafe` constructs
- ■ You must follow the unsafe code guidelines
- ■ You must document your generated bindings to make it clear which call patterns are permissible

Further Reading

- Nomicon
- Unsafe Code Guidelines
- Miri
- "Adversarial code"

All Code Must be Sound (M-UNSOUND)

Why this entry exists:	To prevent unexpected runtime behavior, leading to potential bugs and incompatibilities.
Version:	1.0

Unsound code is seemingly *safe* code that may produce undefined behavior when called from other safe code, or on its own accord.

💡 Meaning of 'Safe'

The terms *safe* and `unsafe` are technical terms in Rust.

A function is *safe*, if its signature does not mark it `unsafe` . That said, *safe* functions can still be dangerous (e.g., `delete_database()`), and `unsafe` ones are, when properly used, usually quite benign (e.g., `vec.get_unchecked()`).

A function is therefore *unsound* if it appears *safe* (i.e., it is not marked `unsafe`), but if *any* of its calling modes would cause undefined behavior. This is to be interpreted in the strictest sense. Even if causing undefined behavior is only a 'remote, theoretical possibility' requiring 'weird code', the function is unsound.

Also see Unsafe, Unsound, Undefined.

```
// "Safely" converts types
fn unsound_ref<T>(x: &T) -> &u128 {
    unsafe { std::mem::transmute(x) }
}

// "Clever trick" to work around missing `Send` bounds.
struct AlwaysSend<T>(T);
unsafe impl<T> Send for AlwaysSend<T> {}
unsafe impl<T> Sync for AlwaysSend<T> {}
```

Unsound abstractions are never permissible. If you cannot safely encapsulate something, you must expose `unsafe` functions instead, and document proper behavior.

> No Exceptions
>
> While you may break most guidelines if you have a good enough reason, there are no exceptions in this case: unsound code is never acceptable.

💡 It's the Module Boundaries

Note that soundness boundaries equal module boundaries! It is perfectly fine, in an otherwise safe abstraction, to have safe functions that rely on behavior guaranteed elsewhere **in the same module**.

```
struct MyDevice(*const u8);

impl MyDevice {
    fn new() -> Self {
       // Properly initializes instance ...
    }

    fn get(&self) -> u8 {
        // It is perfectly fine to rely on `self.0` being valid, despite this
        // function in-and-by itself being unable to validate that.
        unsafe { *self.0 }
    }
}
```

Unsafe Implies Undefined Behavior (M-UNSAFE-IMPLIES-UB)

Why this entry exists:	To ensure semantic consistency and prevent warning fatigue.
Version:	1.0

The marker `unsafe` may only be applied to functions and traits if misuse implies the risk of undefined behavior (UB). It must not be used to mark functions that are dangerous to call for other reasons.

```rust
// Valid use of unsafe
unsafe fn print_string(x: *const String) { }

// Invalid use of unsafe
unsafe fn delete_database() { }
```

Performance Guidelines

Optimize for Throughput, Avoid Empty Cycles (M-THROUGHPUT)

Why this entry exists:	To ensure COGS savings at scale.
Version:	0.1

You should optimize your library for throughput, and one of your key metrics should be *items per CPU cycle*.

This does not mean to neglect latency—after all you can scale for throughput, but not for latency. However, in most cases you should not pay for latency with *empty cycles* that come with single-item processing, contended locks and frequent task switching.

Ideally, you should

- partition reasonable chunks of work ahead of time,
- let individual threads and tasks deal with their slice of work independently,
- sleep or yield when no work is present,
- design your own APIs for batched operations,
- perform work via batched APIs where available,
- yield within long individual items, or between chunks of batches (see M-YIELD-POINTS),
- exploit CPU caches, temporal and spatial locality.

You should not:

- hot spin to receive individual items faster,
- perform work on individual items if batching is possible,
- do work stealing or similar to balance individual items.

Shared state should only be used if the cost of sharing is less than the cost of re-computation.

Identify, Profile, Optimize the Hot Path Early (M-HOTPATH)

Why this entry exists:	To end up with high performance code.
Version:	0.1

You should, early in the development process, identify if your crate is performance or COGS relevant. If it is:

- identify hot paths and create benchmarks around them,
- regularly run a profiler collecting CPU and allocation insights,
- document or communicate the most performance sensitive areas.

For benchmarks we recommend criterion or divan. If possible, benchmarks should not only measure elapsed wall time, but also used CPU time over all threads (this unfortunately requires manual work and is not supported out of the box by the common benchmark utils).

Profiling Rust on Windows works out of the box with Intel VTune and Superluminal. However, to gain meaningful CPU insights you should enable debug symbols for benchmarks in your `Cargo.toml`:

```
[profile.bench]
debug = 1
```

Documenting the most performance sensitive areas helps other contributors take better decision. This can be as simple as sharing screenshots of your latest profiling hot spots.

Further Reading

- Performance Tips

💡 How much faster?

Some of the most common 'language related' issues we have seen include:

- frequent re-allocations, esp. cloned, growing or `format!` assembled strings,
- short lived allocations over bump allocations or similar,
- memory copy overhead that comes from cloning Strings and collections,
- repeated re-hashing of equal data structures
- the use of Rust's default hasher where collision resistance wasn't an issue

Anecdotally, we have seen ~15% benchmark gains on hot paths where only some of these `String` problems were addressed, and it appears that up to 50% could be achieved in highly optimized versions.

Long-Running Tasks Should Have Yield Points. (M-YIELD-POINTS)

Why this entry exists:	To ensure you don't starve other tasks of CPU time.
Version:	0.2

If you perform long running computations, they should contain `yield_now().await` points.

Your future might be executed in a runtime that cannot work around blocking or long-running tasks. Even then, such tasks are considered bad design and cause runtime overhead. If your complex task performs I/O regularly it will simply utilize these await points to preempt itself:

```
async fn process_items(items: &[items]) {
    // Keep processing items, the runtime will preempt you automatically.
    for i in items {
        read_item(i).await;
    }
}
```

If your task performs long-running CPU operations without intermixed I/O, it should instead cooperatively yield at regular intervals, to not starve concurrent operations:

```
async fn process_items(zip_file: File) {
    let items = zip_file.read().async;
    for i in items {
        decompress(i);
        yield_now().await;
    }
}
```

If the number and duration of your individual operations are unpredictable you should use APIs such as `has_budget_remaining()` and related APIs to query your hosting runtime.

💡**Yield how often?**

In a thread-per-core model the overhead of task switching must be balanced against the systemic effects of starving unrelated tasks.

Under the assumption that runtime task switching takes 100's of ns, in addition to the overhead of lost CPU caches, continuous execution in between should be long enough that the switching cost becomes negligible (<1%).

Thus, performing 10 - 100μs of CPU-bound work between yield points would be a good starting point.

Documentation

First Sentence is One Line; Approx. 15 Words (M-FIRST-DOC-SENTENCE)

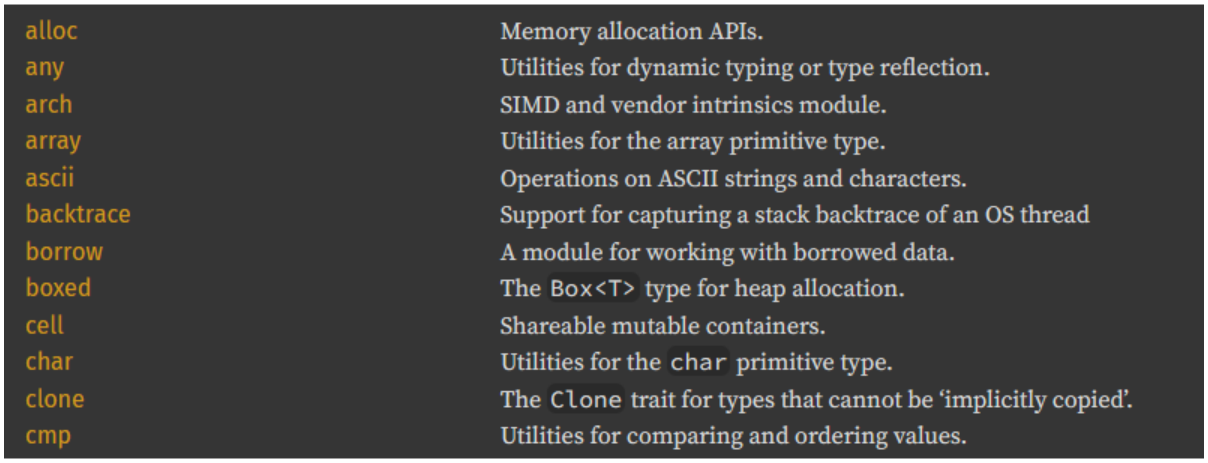
Why this entry exists: To make API docs easily skimmable.
Version: 1.0

When you document your item, the first sentence becomes the "summary sentence" that is extracted and shown in the module summary:

```rust
/// This is the summary sentence, shown in the module summary.
///
/// This is other documentation. It is only shown in that item's detail view.
/// Sentences here can be as long as you like and it won't cause any issues.
fn some_item() { }
```

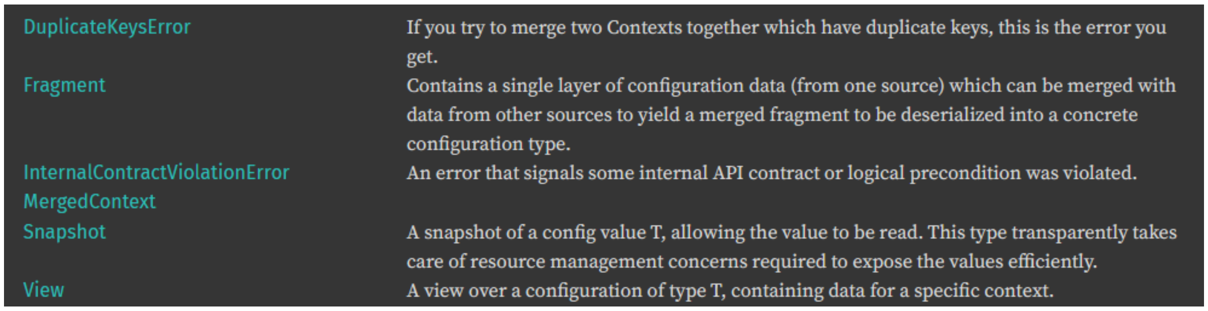
Since Rust API documentation is rendered with a fixed max width, there is a naturally preferred sentence length you should not exceed to keep things tidy on most screens.

If you keep things in a line, your docs will become easily skimmable. Compare, for example, the standard library:



alloc	Memory allocation APIs.
any	Utilities for dynamic typing or type reflection.
arch	SIMD and vendor intrinsics module.
array	Utilities for the array primitive type.
ascii	Operations on ASCII strings and characters.
backtrace	Support for capturing a stack backtrace of an OS thread
borrow	A module for working with borrowed data.
boxed	The `Box<T>` type for heap allocation.
cell	Shareable mutable containers.
char	Utilities for the `char` primitive type.
clone	The `Clone` trait for types that cannot be 'implicitly copied'.
cmp	Utilities for comparing and ordering values.

Otherwise, you might end up with *widows* and a generally unpleasant reading flow:



DuplicateKeysError	If you try to merge two Contexts together which have duplicate keys, this is the error you get.
Fragment	Contains a single layer of configuration data (from one source) which can be merged with data from other sources to yield a merged fragment to be deserialized into a concrete configuration type.
InternalContractViolationError	An error that signals some internal API contract or logical precondition was violated.
MergedContext	
Snapshot	A snapshot of a config value T, allowing the value to be read. This type transparently takes care of resource management concerns required to expose the values efficiently.
View	A view over a configuration of type T, containing data for a specific context.

As a rule of thumb, the first sentence should not exceed **15 words**.

Has Comprehensive Module Documentation (M-MODULE-DOCS)

Why this entry exists:	To allow for better API docs navigation.
Version:	1.1

Any public library module must have `//!` module documentation, and the first sentence must follow M-DOC-FIRST-SENTENCE.

```
pub mod ffi {
    //! Contains FFI abstractions.

    pub struct String {};
}
```

The rest of the module documentation should be comprehensive, i.e., cover the most relevant technical aspects of the contained items, including

- what the module contains
- when it should be used, possibly when not
- examples
- subsystem specifications (e.g., `std::fmt` also describes its formatting language)
- observable side effects, including what guarantees are made about these, if any
- relevant implementation details, e.g., the used system APIs

Great examples include:

- `std::fmt`
- `std::pin`
- `std::option`

This does not mean every module should contain all of these items. But if there is something to say about the interaction of the contained types, their module documentation is the right place.

Documentation Has Canonical Sections (M-CANONICAL-DOCS)

Why this entry exists:	To follow established and expected Rust best practices.
Version:	1.0

Public library items must contain the canonical doc sections. The summary sentence must always be present. Extended documentation and examples are strongly encouraged. The other sections must be present when applicable.

```rust
/// Summary sentence < 15 words.
///
/// Extended documentation in free form.
///
/// # Examples
/// One or more examples that show API usage like so.
///
/// # Errors
/// If fn returns `Result`, list known error conditions
///
/// # Panics
/// If fn may panic, list when this may happen
///
/// # Safety
/// If fn is `unsafe` or may otherwise cause UB, this section must list
/// all conditions a caller must uphold.
///
/// # Abort
/// If fn may abort the process, list when this may happen.
pub fn foo() {}
```

In contrast to other languages, you should not create a table of parameters. Instead parameter use is explained in plain text. In other words, do not

```rust
/// Copies a file.
///
/// # Parameters
/// - src: The source.
/// - dst: The destination.
fn copy(src: File, dst: File) {}
```

but instead:

```rust
/// Copies a file from `src` to `dst`.
fn copy(src: File, dst: File) {}
```

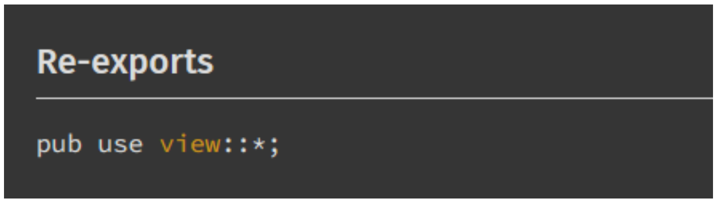
Related Reading

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

Mark `pub use` Items with `#[doc(inline)]` (M-DOC-INLINE)

Why this entry exists: To make re-exported items 'fit in' with their non re-exported siblings.
Version: 1.0

When publicly re-exporting crate items via `pub use foo::Foo` or `pub use foo::*`, they show up in an opaque re-export block. In most cases, this is not helpful to the reader:



Re-exports

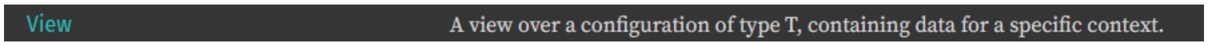
```rust
pub use view::*;
```

Instead, you should annotate them with `#[doc(inline)]` at the `use` site, for them to be inlined organically:

```
#[doc(inline)]
pub use foo::*;

// or

#[doc(inline)]
pub use foo::Foo;
```



View A view over a configuration of type T, containing data for a specific context.

This does not apply to `std` or 3rd party types; these should always be re-exported without inlining to make it clear they are external.

⚠️Still avoid glob exports

The `#[doc(inline)]` trick above does not change M-NO-GLOB-REEXPORTS; you generally should not re-export items via wildcards.

AI Guidelines

Design with AI use in Mind (M-DESIGN-FOR-AI)

Why this entry exists:	To maximize the utility you get from letting agents work in your code base.
Version:	0.1

As a general rule, making APIs easier to use for humans also makes them easier to use by AI. If you follow the guidelines in this book, you should be in good shape.

Rust's strong type system is a boon for agents, as their lack of genuine understanding can often be counterbalanced by comprehensive compiler checks, which Rust provides in abundance.

With that said, there are a few guidelines which are particularly important to help make AI coding in Rust more effective:

- **Create Idiomatic Rust API Patterns**. The more your APIs, whether public or internal, look and feel like the majority of Rust code in the world, the better it is for AI. Follow the Rust API Guidelines along with the guidelines from Library / UX.
- **Provide Thorough Docs**. Agents love good detailed docs. Include docs for all of your modules and public items in your crate. Assume the reader has a solid, but not expert, level of understanding of Rust, and that the reader understands the standard library. Follow C-CRATE-DOC, C-FAILURE, C-LINK, and M-MODULE-DOCS M-CANONICAL-DOCS.
- **Provide Thorough Examples**. Your documentation should have directly usable examples, the repository should include more elaborate ones. Follow C-EXAMPLE C-QUESTION-MARK.
- **Use Strong Types**. Avoid primitive obsession by using strong types with strict well-documented semantics. Follow C-NEWTYPE.
- **Make Your APIs Testable**. Design APIs which allow your customers to test their use of your API in unit tests. This might involve introducing some mocks, fakes, or cargo features. AI agents need to be able to iterate quickly to prove that the code they are writing that calls your API is working correctly.
- **Ensure Test Coverage**. Your own code should have good test coverage over observable behavior. This enables agents to work in a mostly hands-off mode when refactoring.

Agents & LLMs

While these guidelines have been carefully handcrafted, they have been written with LLM consumption in mind. We offer condensed versions you can include in your agent sessions when authoring or reviewing code.

Available Resources

Name	Size	Description
all 📋	~88 kb, ~22k tokens	The entire Pragmatic Rust Guidelines in one file.

FAQ

How can I contribute?

Go to our GitHub page and file an issue or PR. New guidelines should be compatible with the meta design principles.

I disagree with a guideline ...

If you find a mistake or issue, let us know. Also let us know if a *must* guideline interferes with a valid use case. If you disagree with a *should* guideline, there are usually valid exceptions we haven't documented. If you think a *should* guideline generally leads to worse outcomes, let us know.