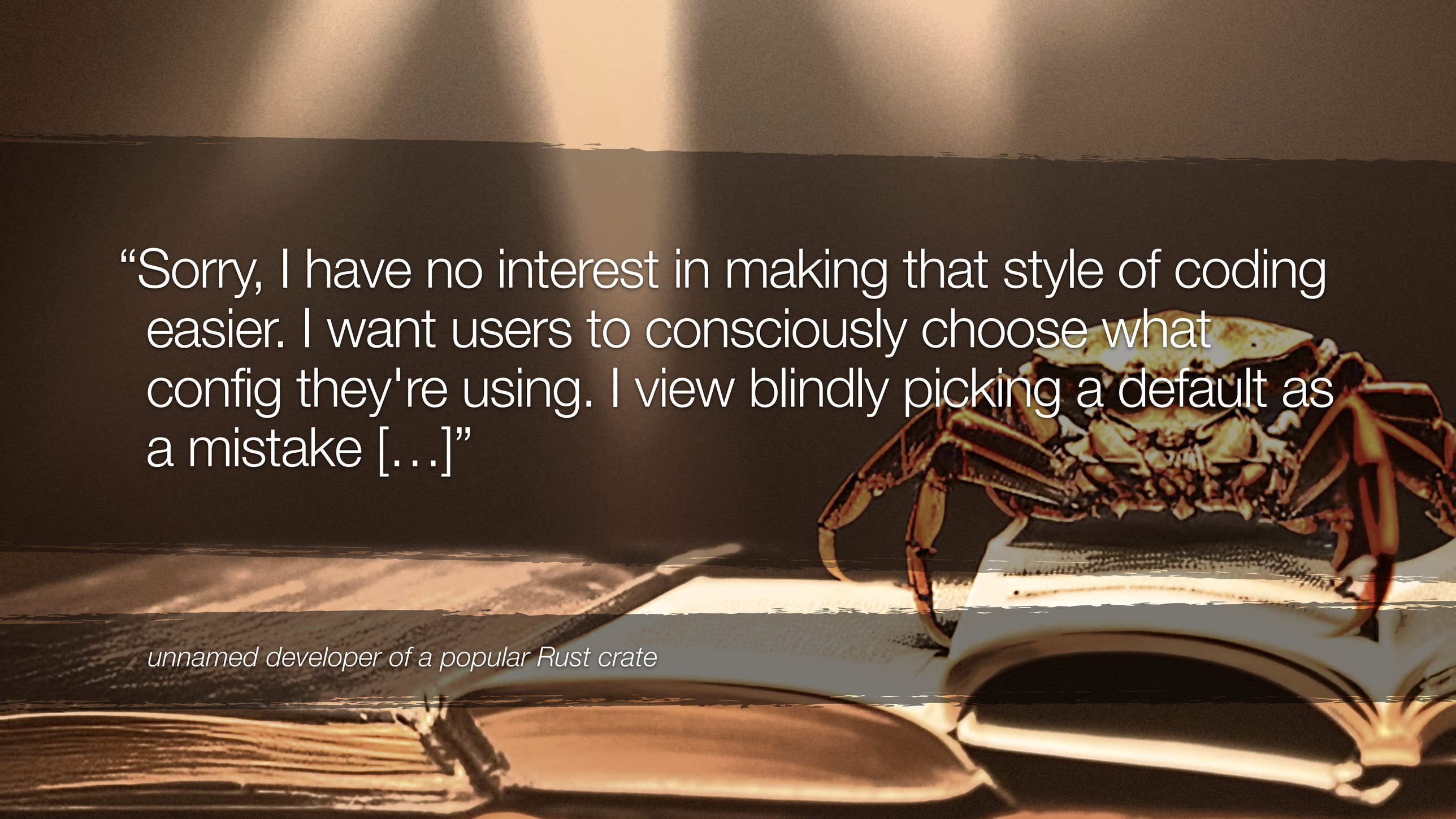
Rust API Design Learnings

Lessons learned from building Rust libraries

Armin @mitsuhiko Ronacher

Who am I

- Armin Ronacher
- twitter.com/@mitsuhiko | hachyderm.io/@mitsuhiko
- Python since time immemorial, Rust since 2012
- Python: Flask, Jinja, Werkzeug, …
- Rust: Insta, MiniJinja, Console, Indicatif, Similar



“Sorry, I have no interest in making that style of coding easier. I want users to consciously choose what config they're using. I view blindly picking a default as a mistake […]”

unnamed developer of a popular Rust crate

APIs are Important

- A library's author's true success metrics are:
  - how successful all users are in using the API
  - the quality of the output that users achieve by using the API
  - the percentage of users making the correct choices

Your User Matters

- When you build a library you should treat it like any other thing
- Define success metrics
- Measure yourself

But we are Flying Blind

- Library developers typically fly blind
- The only metrics we have is download stats, which mostly correlate with CI setups, and not true utilization
- User frustration is often the only other form of feedback we get
- We need extrapolation from user surveys and interviews
- In the absence of this, personal frustration and issues is a good proxy

Values: Metrics without Measuring

- If we have trouble measuring, metrics are not useless
- Metrics often express what we believe is important
- Values can steer us

Values and Metrics

My Values

- Concise: easy to get started
- Good Defaults: easy to get started, trivial to stay on the golden path as it changes
- Small Surface Area: enable room to breath and innovate, without breaking users
- Backwards compatible: avoid unnecessary churn to keep users on the golden path

The Golden Path

The Golden Path

- An opinionated path for how to build
- That path might change over time
- Change requires adjustment by users
- Fast change means users being left behind
- Measuring success: users on the golden path (not churning, not staying on old versions, not hating the upgrade experience, not using old patterns)

Defaults Matter

Use Defaults to Fight Cargo Cult

- Defaults are hard and of two types:
  - Absolute defaults that cannot be changed (i32::default() -> 0)
  - Defaults that allow a level of flexibility (Default Hasher: SipHash)
- For defaults to allow flexibility, care has to be taken:
  - Set rules and expectations about stability
  - Aim for some level of change

Good Defaults

- Default Hasher:
  - Hasher is documented to be non portable
  - Hasher is documented to change
  - No expectation around cross-version/process stability
- A better hasher can be picked, all code ever written benefits at once

Cargo Cult

- Imagine mandatory hasher
- People would cargo cult some default hasher that they see elsewhere or in the docs.
- New hasher comes around, lots of code stuck with the old choice.

```rust
use std::collections::HashMap;
use std::hash::BuildHasherDefault;
use std::hash::fxhash::FxHasher;

type FxBuildHasher = BuildHasherDefault<FxHasher>;

struct ThingCollection {
    extra: HashMap<String, Thing, FxBuildHasher>,
}
```

Defaults and Protocols

- What if this hash becomes part of a protocol?
- If you have an API that drives a protocol, consider that protocol to consider defaults
- This approach can only be guidance, a lot of situations do not allow it.

```
fn calculate_checksum<C: Default>(bytes: &[u8]) -> String;
fn check_checksum(bytes: &[u8], sig: &str) -> bool;

>> calculate_checksum(&"foobar")
"sha256:c3ab8ff13720e8ad9047dd39466b3c8974e592c2fa383d4a3960714caef0c4f2"

>> check_checksum(&"foobar", "sha256:c3ab8ff…")
true
```

Less is More

More API = More Problems

- The larger the surface, the more of it ends up used
- Less commonly used APIs have the most leaky abstractions
- Inhibits future change: "does someone even use this?"

Hide API Behind Common Abstractions

- Developers are used to these patterns, they are worth exploring:
  - Into<T>
  - AsRef<T>
- Careful: surface area stays large, but large bound to common and simple patterns

Into

- Common pairs:
  - Into<String>
  - Into<Cow<'_, T>>
  - Into<YourRuntimeType>

```rust
/// Adds a global variable.
pub fn add_global<N, V>(&mut self, name: N, value: V)
where
    N: Into<Cow<'source, str>>,
    V: Into<Value>,
{
    self.globals.insert(name.into(), value.into());
}
```

- ToString can be sometimes an interesting alternative to Into<String>

AsRef<T>

- Related in Into, but for borrowing
- Abstracts over
  - &String/&str/&Cow<'_, str>
  - &PathBuf/&Path
  - &[u8]/&Vec<u8>/&String/&str

```rust
pub fn snapshot_path<P: AsRef<Path>>(&mut self, path: P) {
    self.snapshot_path = path.as_ref().to_path_buf();
}

pub fn input_file<P: AsRef<Path>>(&mut self, p: P) {
    self.input_file = Some(p.as_ref().to_path_buf());
}
```

Monomorphization & Compile Times

- Rust loves to inline
- All those different types create duplicated generated code
- Example: isolate conversions and call into shared functions to reduce the total amount of copied code.

```rust
pub fn render<S: Serialize>(&self, ctx: S) -> Result<String, Error> {
    self._render(Value::from_serializable(&ctx))
}

fn _render(&self, root: Value) -> Result<String, Error> {
    let mut rv = String::new();
    self._eval(root, &mut Output::with_string(&mut rv))
        .map(|_| rv)
}

fn _eval(&self, root: Value, out: &mut Output) -> Result<Option<Value>, Error> {
    Vm::new(self.env).eval(
        &self.compiled.instructions,
        root,
        &self.compiled.blocks,
        out,
        self.initial_auto_escape,
    )
}
```

Hide the Onion but create the Onion

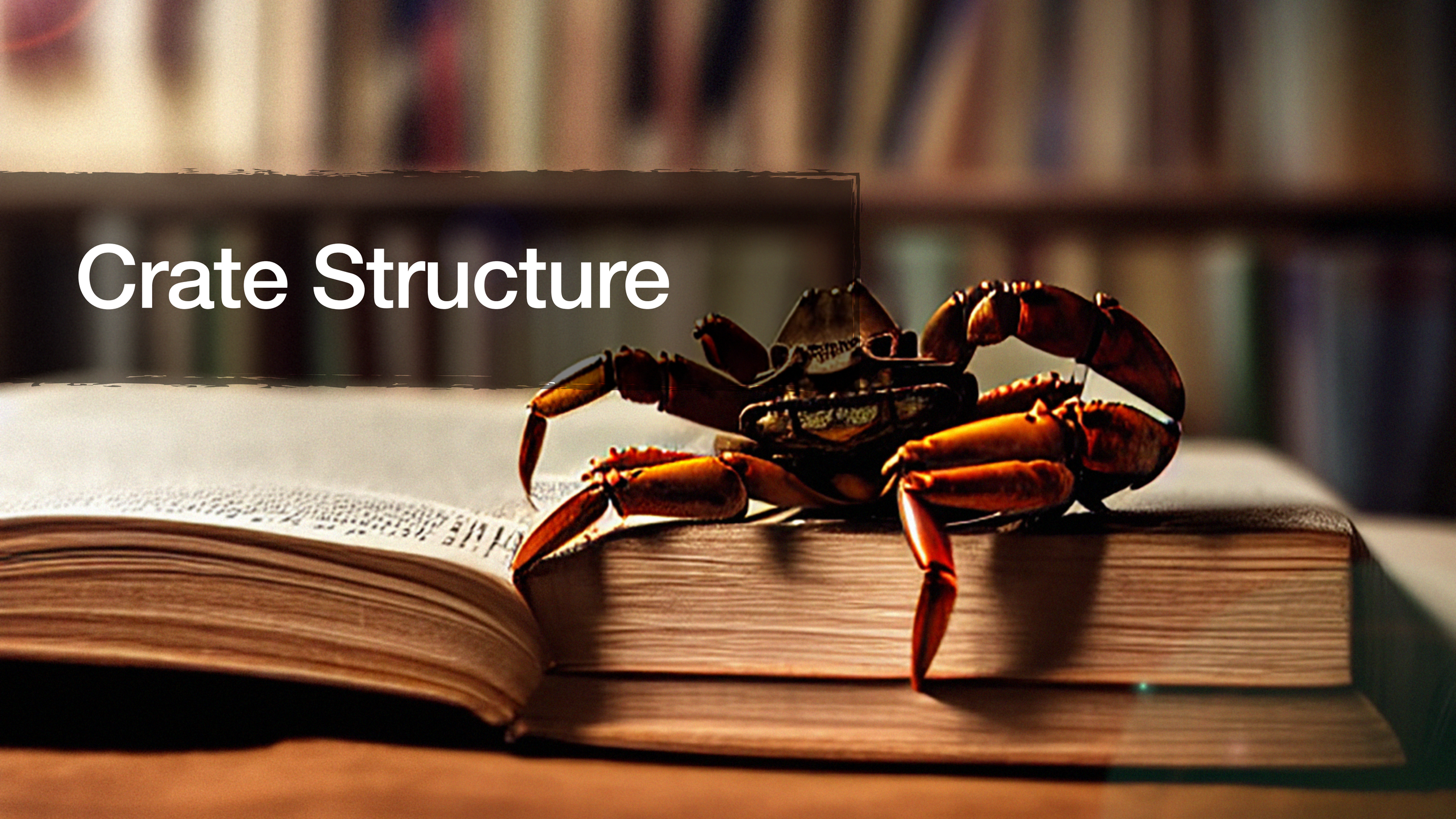
- Good APIs are Layered Like Onions
- Only provide the outermost layer first
- Keeps the inner layers flexibility to change
- Over time, consider exposing internal layers under separate stability guarantees

Layer 2 and 3

- Example: CompiledTemplate is entirely private, so is the CodeGenerator or the parser.
- It's still layered, and over time some functionality *could* be exposed.

```
impl<'source> CompiledTemplate<'source> {
    fn from_name_and_source_impl(
        name: &'source str,
        source: &'source str,
    ) -> Result<CompiledTemplate<'source>, Error> {
        value::with_value_optimization(|| {
            let ast = ok!(parse(source, name));
            let mut gen = CodeGenerator::new(name, source);
            gen.compile_stmt(&ast);
            let (instructions, blocks) = gen.finish();
            Ok(CompiledTemplate {
                instructions,
                blocks,
            })
        })
    }
}
```

Crate Structure



Explicit Exports

- Hide your internal structure, re-export sensibly
- Your folder structure does not matter to your users

```rust
pub use self::defaults::{default_auto_escape_callback, escape_formatter};
pub use self::environment::Environment;
pub use self::error::{Error, ErrorKind};
pub use self::expression::Expression;
pub use self::output::Output;
pub use self::template::Template;
pub use self::utils::{AutoEscape, HtmlEscape};

#[cfg(feature = "source")]
pub use self::source::Source;

pub use self::vm::State;
```

Explicit Fake Modules

- Consider creating modules on the spot for utilities
- For instance "insta" has utility functions and types that are rarely useful. The ones I subscribe stability to are re-exported under a specific module.

```rust
pub mod internals {
    pub use crate::content::Content;
    #[cfg(feature = "filters")]
    pub use crate::filters::Filters;
    pub use crate::runtime::AutoName;
    pub use crate::settings::SettingsBindDropGuard;
    pub use crate::snapshot::{MetaData, SnapshotContents};
    #[cfg(feature = "redactions")]
    pub use crate::{
        redaction::{ContentPath, Redaction},
        settings::Redactions,
    };
}
```

Public but Hidden

- Sometimes stuff needs to be public, but you don't want anyone to use it.
- Common example: utility functionality for macros.
- Here both __context and __context_pair! are public but hidden

```
#[macro_export]
macro_rules! context {
    () => { $crate::__context::build($crate::__context::make()) };
    ( $($key:ident $(=> $value:expr)?),* $(,)? ) => {{
        let mut ctx = $crate::__context::make();
        $( $crate::__context_pair!(ctx, $key $(, $value)?); )*
        $crate::__context::build(ctx)
    }}
}

#[macro_export]
#[doc(hidden)]
macro_rules! __context_pair {
    ($ctx:ident, $key:ident) => {{
        $crate::__context_pair!($ctx, $key, $key);
    }};
    ($ctx:ident, $key:ident, $value:expr) => {
        $crate::__context::add(
            &mut $ctx,
            stringify!($key),
            $crate::value::Value::from_serializable(&$value),
        );
    };
}
```

Traits

Traits are Tricky

- Traits are super useful, but they are tricky
- Fall into two categories:
  - Sealed (user should not implement)
  - Open (user should implement)

Sealed Traits

- Not really supported, doc hidden and hackery
- Example in MiniJinja: want to abstract over types, but I don't really want to let the user do that.

```
26 implementations
pub trait ArgType<'a> {
    /// The output type of this argument.
    type Output;

    #[doc(hidden)]
    fn from_value(value: Option<&'a Value>) -> Result<Self::Output, Error>;

    #[doc(hidden)]
    fn from_value_owned(_value: Value) -> Result<Self::Output, Error> {
        Err(Error::new(
            ErrorKind::InvalidOperation,
            "type conversion is not legal in this situation (implicit borrow)",
        ))
    }

    #[doc(hidden)]
    fn from_state_and_value(
        _state: Option<&'a State>,
        value: Option<&'a Value>,
    ) -> Result<(Self::Output, usize), Error> {
        Ok((ok!(Self::from_value(value)), 1))
    }
}
```

Full Seal

- Uses a private zero sized marker type somewhere
- User cannot implement or invoke as the type is private

```
6 implementations
pub trait Function<Rv, Args>: Send + Sync + 'static {
    #[doc(hidden)]
    fn invoke(&self, args: Args, _: SealedMarker) -> Rv;
}
```

Traits are Hard to Discover

- I avoid traits unless I know abstraction over implementations is necessary
- Did you notice that BTreeMap and HashMap are not expressed via traits?
- The usefulness of abstraction even for interchangeable types is sometimes unclear
- You can always add traits later

Common Traits

Debug

- Put it on all public types
- Consider it on your internal types behind a feature flag
- Super valuable for dbg!() and co

```rust
/// An if/else condition.
#[cfg_attr(feature = "internal_debug", derive(Debug))]
1 implementation
pub struct IfCond<'a> {
    pub expr: Expr<'a>,
    pub true_body: Vec<Stmt<'a>>,
    pub false_body: Vec<Stmt<'a>>,
}
```

Display

- Makes the type have a representation in format!()
- It also gives it the `.to_string()` method
- Certain types need it in the contract (eg: all errors)
- Recommendation: avoid in most cases unless you implement a custom integer, string etc.

Copy and Clone

- Once granted, impossible to take away
- Neither can be universally provided
- Clone: really useful, consider adding
  - If you ever feel you need to take it away, consider Arc<T> internally
- Copy: might inhibit future change, but really useful
  - Some types regrettably do not have Copy (eg: Range) and people hate it

Sync and Send

- I cannot give recommendations
- The only one I have: non Send/Sync types are not that bad
- Consider them seriously

Lifetimes

Lifetimes and Libraries

- Try to avoid too clever setups
- Consider "Session" abstractions where people only need to temporarily hold on to data.

```rust
/// A debugging session for DWARF debugging information.
2 implementations
pub struct DwarfDebugSession<'data> {
    cell: SelfCell<Box<DwarfSections<'data>>, DwarfInfo<'data>>,
    bcsymbolmap: Option<Arc<BcSymbolMap<'data>>>,
}
```

```rust
fn execute(matches: &ArgMatches) -> Result<()> {
    let path = matches.value_of("path").unwrap_or("a.out");
    let view = ByteView::open(path).context("failed to open file")?;
    let object = Object::parse(&view).context("failed to parse file")?;
    let session = object.debug_session().context("failed to process file")?;
    let symbol_map = object.symbol_map();
```

Borrowing to Self

- Rust is really bad at this, sometimes you build yourself into a corner
- Best tool I found to date for this is the self_cell crate
- Buffer can be held into itself

```
self_cell! {
    struct LoadedTemplate {
        owner: (String, String),
        #[covariant]
        dependent: CompiledTemplate,
    }
}
```

```
let owner = (name.clone(), source);
let tmpl = ok!(LoadedTemplate::try_new(
    owner,
    |(name, source)| -> Result<_, Error> {
        CompiledTemplate::from_name_and_source(name.as_str(), source)
    }
));
```

Erroring

Panic vs Error

- Try to avoid panics
- If you do need to panic, consider #[track_caller]

```
impl Stack {
    pub fn push(&mut self, arg: Value) {
        self.values.push(arg);
    }

    #[track_caller]
    pub fn pop(&mut self) -> Value {
        self.values.pop().unwrap()
    }

    // ...
}
```

Errors Matter

- Spend some time designing your errors
- Errors deserve attention just as much as your other types
- A talk all by itself, so here the basics:
  - Implement std::error::Error on your errors
  - Implement source() if you think someone might want to peak into

Questions!