Linting with Dylint

Samuel Moelius
sam.moelius@trailofbits.com

Overview

- Rust linting
- Five categories of lints Dylint is good for
- `cargo-dylint` and `dylint-link`
- Example Dylint lint: `try_io_result`
- Resources
- Future work

```
cargo install cargo-dylint dylint-link
```

Rust linting

Lint

- From Wikipedia's Lint (software):

  Lint, or a linter, is a static code analysis tool used to flag programming errors, bugs, stylistic errors and suspicious constructs.

Rust linters

- The Rust compiler itself (`rustc`) includes many lints, e.g., `unreachable_code`, `unused_imports`, etc.
- Clippy is “a collection of lints to catch common mistakes and improve your Rust code”.
- 👉 Dylint 👈 runs lints from dynamic libraries named by the user, allowing developers to maintain their own personal lint collections.

  ☝ The subject of this talk

A Rust lint is a Rust lint is a Rust lint...

- A Dylint lint...
  - Is essentially no different than a Rust compiler lint...
  - Is essentially no different than a Clippy lint.
- Each use the same *unstable(!)* Rust compiler APIs.*

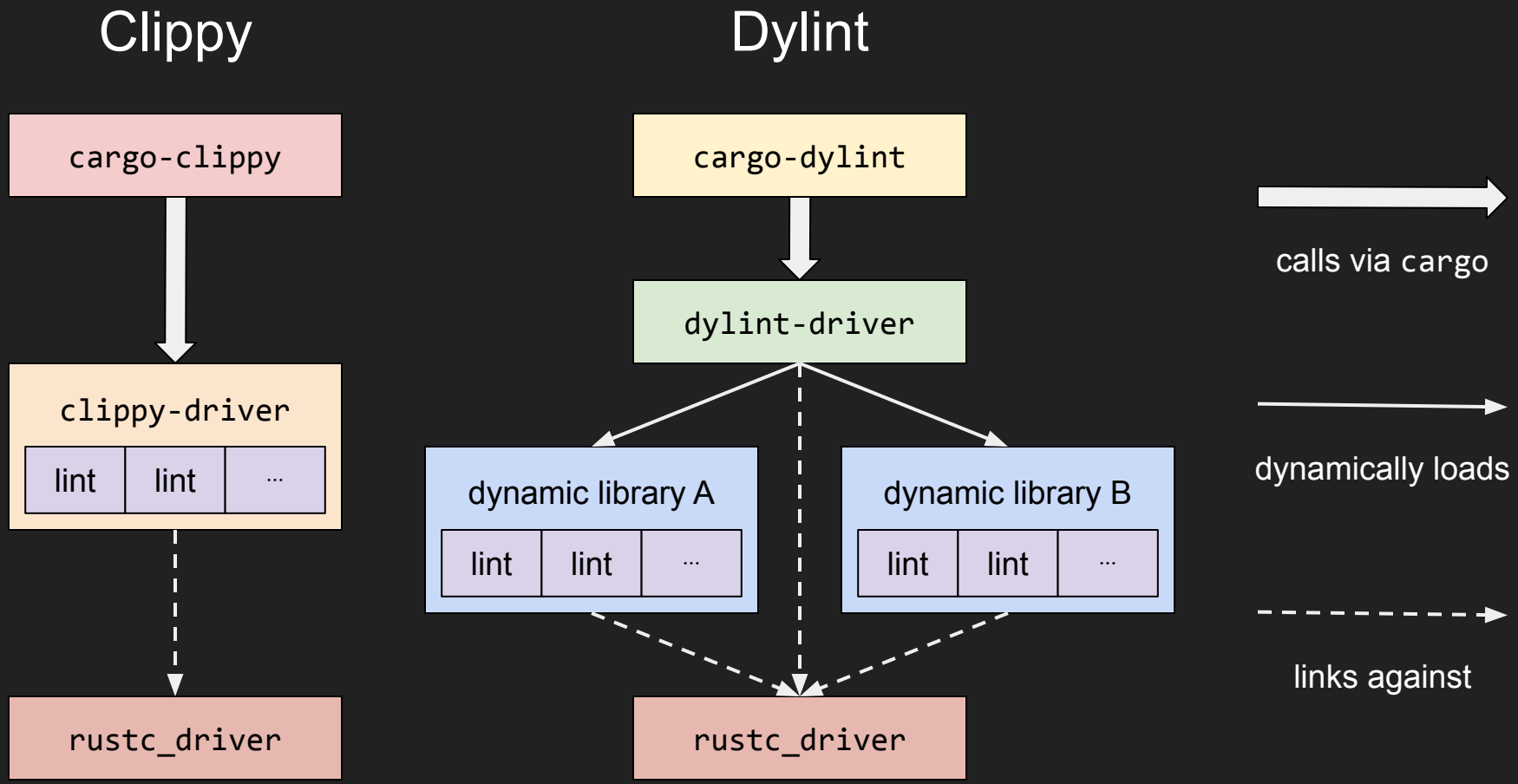
* Marker (due largely to Fridtjof Stoldt aka @xFrednet) is an attempt to create a stable linting interface on top of the Rust compiler APIs.

Rust Compiler APIs are unstable

- The Rust compiler APIs can change from one day to the next.
- An example from nightly-2024-03-04 to nightly-2024-03-05:

```
@@ -179,7 +174,7 @@ fn format_project<T: FormatHandler>(
179     struct FormatContext<'a, T: FormatHandler> {
180         krate: &'a ast::Crate,
181         report: FormatReport,
182 -       parse_session: ParseSess,
183         config: &'a Config,
184         handler: &'a mut T,
185     }
```

```
@@ -179,7 +174,7 @@ fn format_project<T: FormatHandler>(
174     struct FormatContext<'a, T: FormatHandler> {
175         krate: &'a ast::Crate,
176         report: FormatReport,
177 +       psess: ParseSess,
178         config: &'a Config,
179         handler: &'a mut T,
180     }
```


Clippy
Dylint
cargo-clippy
clippy-driver
lint
lint
...
rustc_driver
cargo-dylint
dylint-driver
dynamic library A
lint
lint
...
dynamic library B
lint
lint
...
rustc_driver
calls via cargo
dynamically loads
links against

Stages of a compiled program

- AST - abstract syntax tree
  - Comments and whitespace have been removed, major syntactic constructs (e.g., functions, statements) have been identified
- HIR - high-level intermediate representation
  - Names have been resolved, types have been checked, ...

Types of lints

- Pre-expansion
  - Run on the AST before macros are expanded
- Early
  - Run on the AST after macros have been expanded
- Late
  - Run on the HIR, i.e., after names have been resolved, types have been checked, ...

Types of lints

- Pre-expansion
  - Run on the AST before macros are expanded
- Early
  - Run on the AST after macros have been expanded
- Late
  - Run on the HIR, i.e., after names have been resolved, types have been checked, ...

When starting a new lint, you nearly always want a late lint.

Five categories of lints Dylint is good for

“I have an idea for a lint...

Why would I write a Dylint lint?

Why not just submit a pull request to Clippy?”

Five categories of lints Dylint is good for

- Lints involving third-party crates
  - Clippy has a policy of not linting third-party APIs

Five categories of lints Dylint is good for

- Lints involving third-party crates
- Obscure or exceedingly complex lints
  - To save Clippy maintainers the maintenance burden

Five categories of lints Dylint is good for

- Lints involving third-party crates
- Obscure or exceedingly complex lints
- Project-specific lints
  - E.g., lints involving a project's internal functions

Five categories of lints Dylint is good for

- Lints involving third-party crates
- Obscure or exceedingly complex lints
- Project-specific lints
- Lints with a high false-positive rate
  - …

Five cate

- Lints in
- Obscur
- Project-
- Lints wi
  - ...

Five categories of lints Dylint is good for

- Lints involving third-party crates
- Obscure or exceedingly complex lints
- Project-specific lints
- Lints with a high false-positive rate

Five categories of lints Dylint is good for

- Lints involving third-party crates
- Obscure or exceedingly complex lints
- Project-specific lints
- Lints with a high false-positive rate
- Proprietary lints
  - E.g., analyses whose details you would prefer not to share

cargo-dylint and dylint-link

`cargo-dylint` and `dylint-link`

- `cargo-dylint` allows you to, e.g.:
  - run the lints in a library
  - list the lints in a library
  - create new a new library package
  - upgrade a library package to a more recent toolchain
- `dylint-link` builds libraries with filenames that Dylint recognizes

$

```
$ cargo dylint new my_new_lint
```

```
$ cargo dylint new my_new_lint
$
```

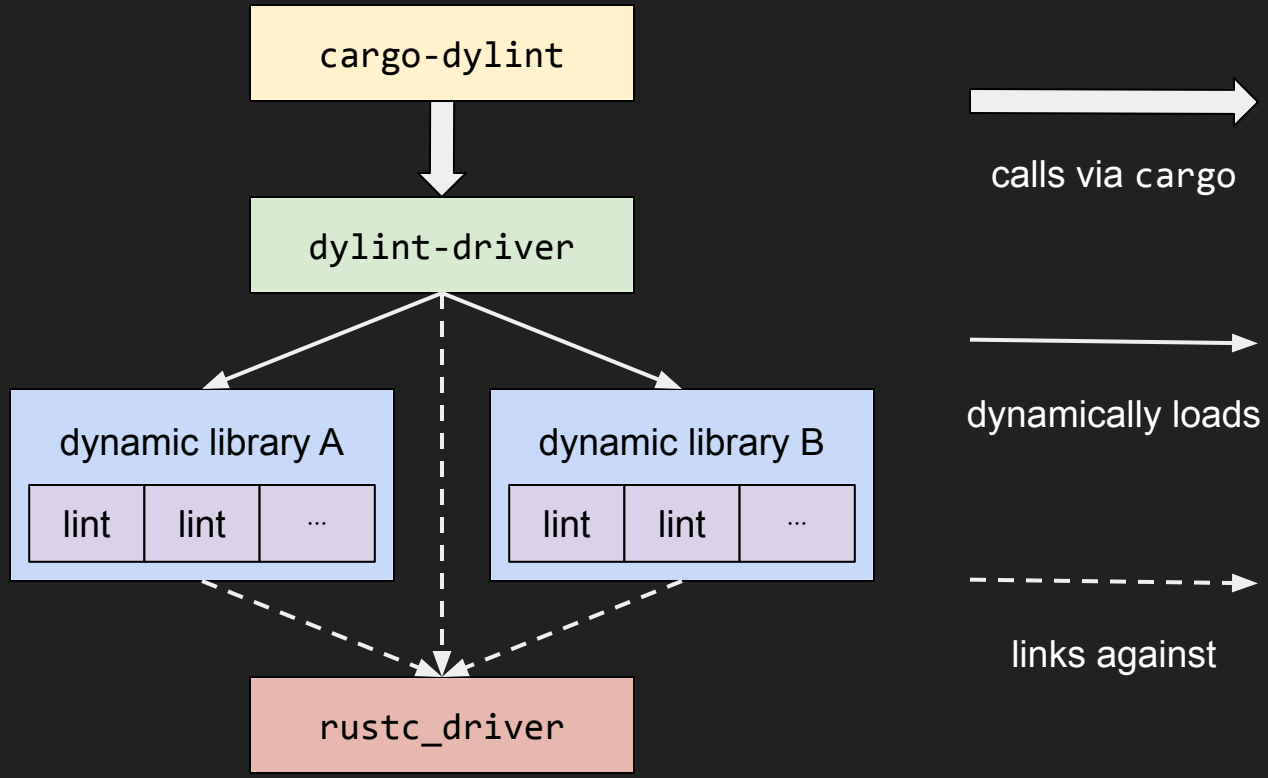
```
$ cargo dylint new my_new_lint
$ tree -a my_new_lint
```

```
$ cargo dylint new my_new_lint
$ tree -a my_new_lint
my_new_lint
├── .cargo
│   └── config.toml
├── .gitignore
├── Cargo.toml
├── README.md
├── rust-toolchain
├── src
│   └── lib.rs
└── ui
    ├── main.rs
    └── main.stderr
```

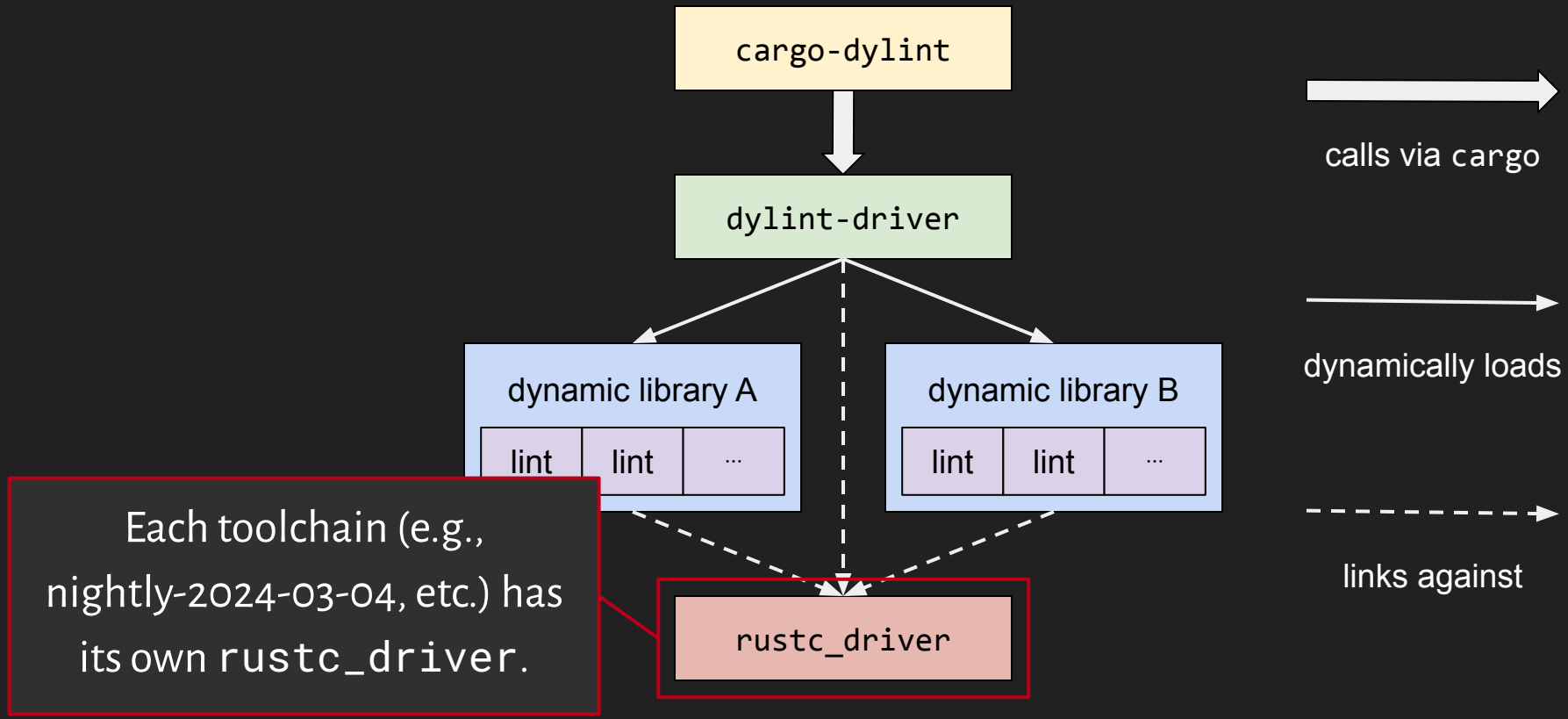
```
$ cargo dylint new my_new_lint
$ tree -a my_new_lint
my_new_lint
├── .cargo
│   └── config.toml
├── .gitignore
├── Cargo.to
├── README.m
├── rust-toolchain
├── src
│   └── lib.rs
└── ui
    ├── main.rs
    └── main.stderr
```

```
[target.'cfg(all())']
rustflags = ["-C", "linker=dylint-link"]
```

Dylint

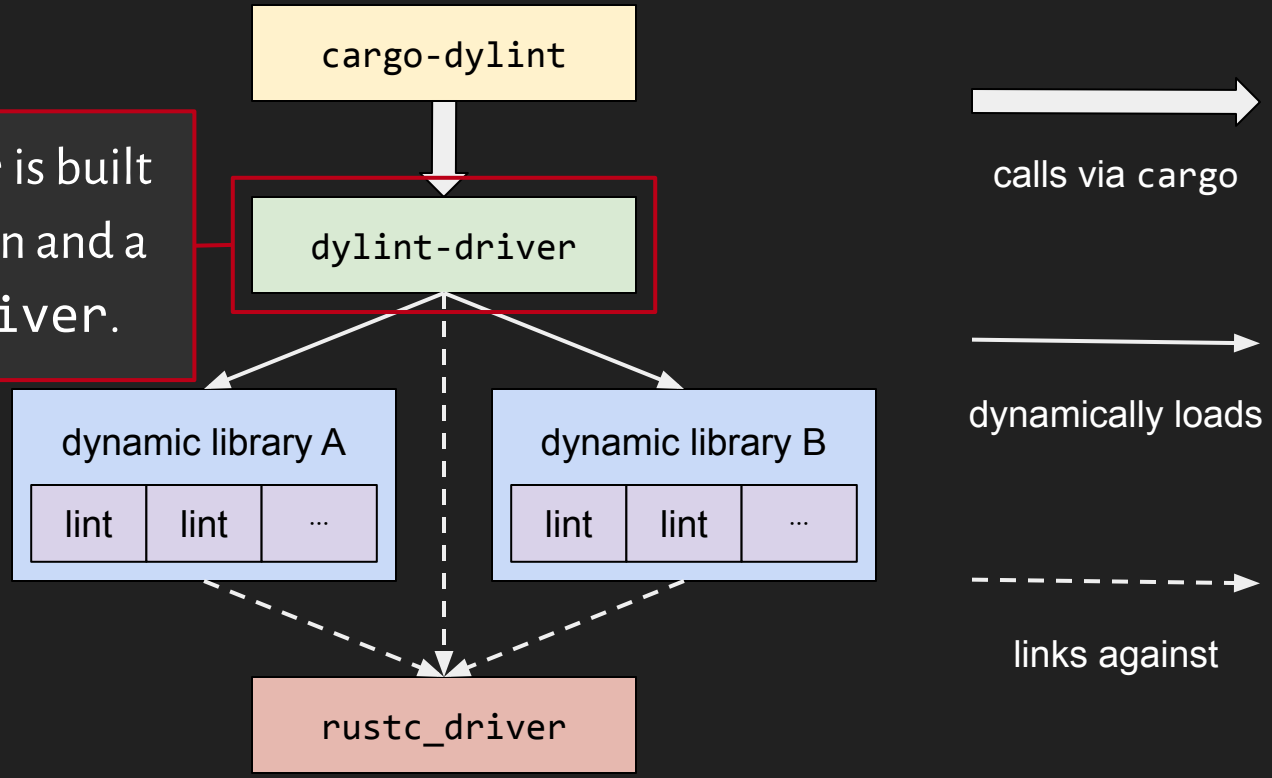

cargo-dylint
dylint-driver
dynamic library A
lint
lint
...
dynamic library B
lint
lint
...
rustc_driver
calls via cargo
dynamically loads
links against

Dylint

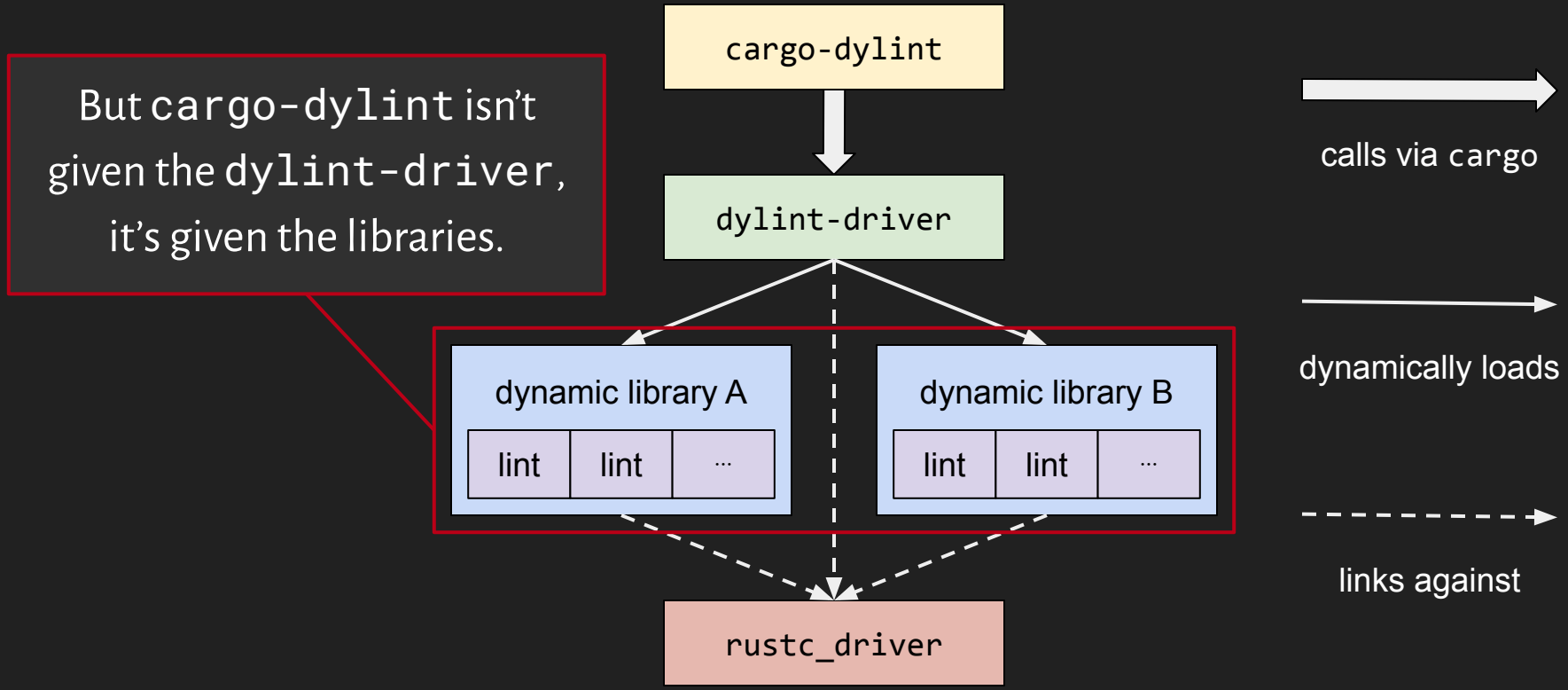

cargo-dylint
dylint-driver
dynamic library A
lint
lint
···
dynamic library B
lint
lint
···
rustc_driver
calls via cargo
dynamically loads
links against
Each toolchain (e.g., nightly-2024-03-04, etc.) has its own rustc_driver.

Dylint

Each `dylint-driver` is built for a particular toolchain and a particular `rustc_driver`.

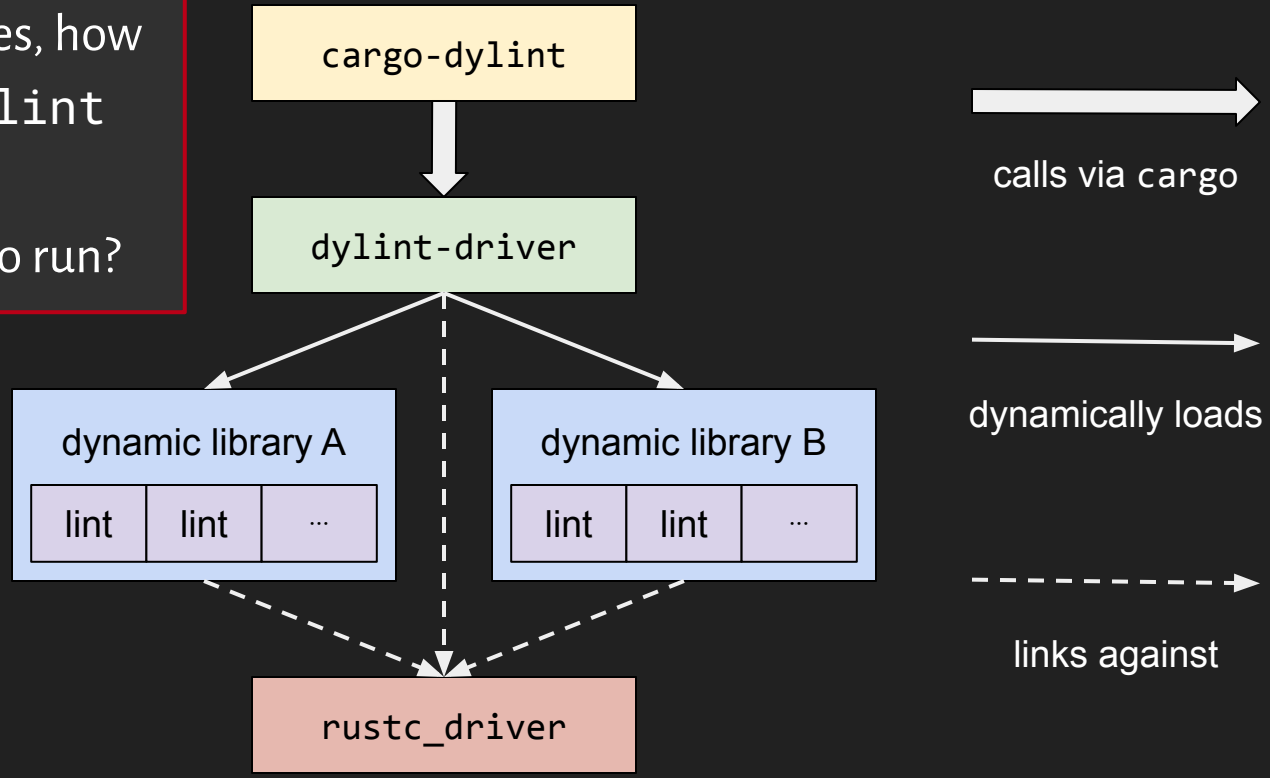


Dylint


But cargo-dylint isn't given the dylint-driver, it's given the libraries.
cargo-dylint
dylint-driver
dynamic library A
lint
lint
...
dynamic library B
lint
lint
...
rustc_driver
calls via cargo
dynamically loads
links against

Dylint

Given a set of libraries, how should `cargo-dylint` know which `dylint-driver` to run?



Form of a Dylint library filename

```
DLL_PREFIX LIBRARY_NAME '@' TOOLCHAIN DLL_SUFFIX
```

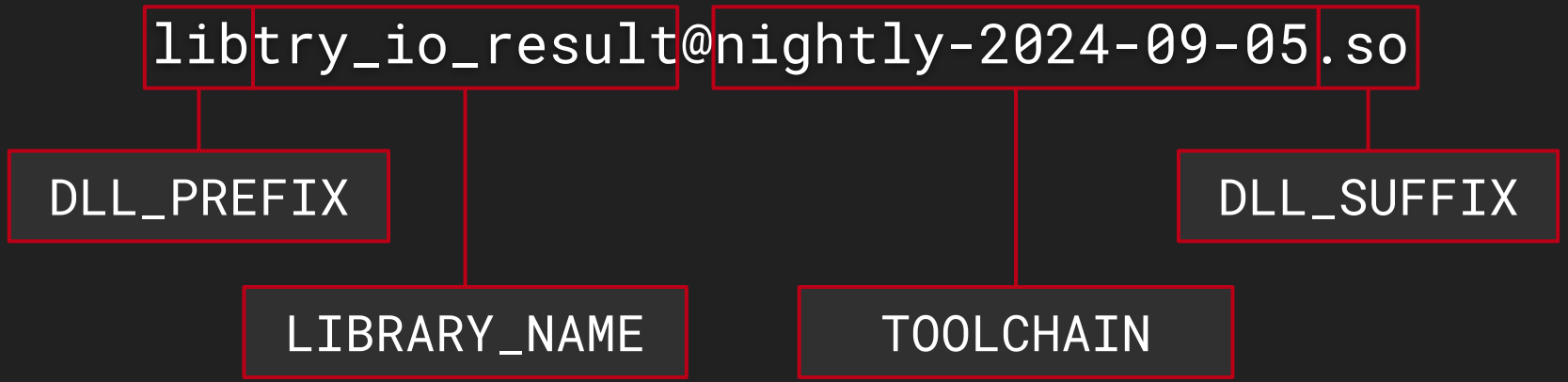
Concrete example on Linux:

```
libtry_io_result@nightly-2024-09-05.so
```

Form of a Dylint library filename

```
DLL_PREFIX LIBRARY_NAME '@' TOOLCHAIN DLL_SUFFIX
```

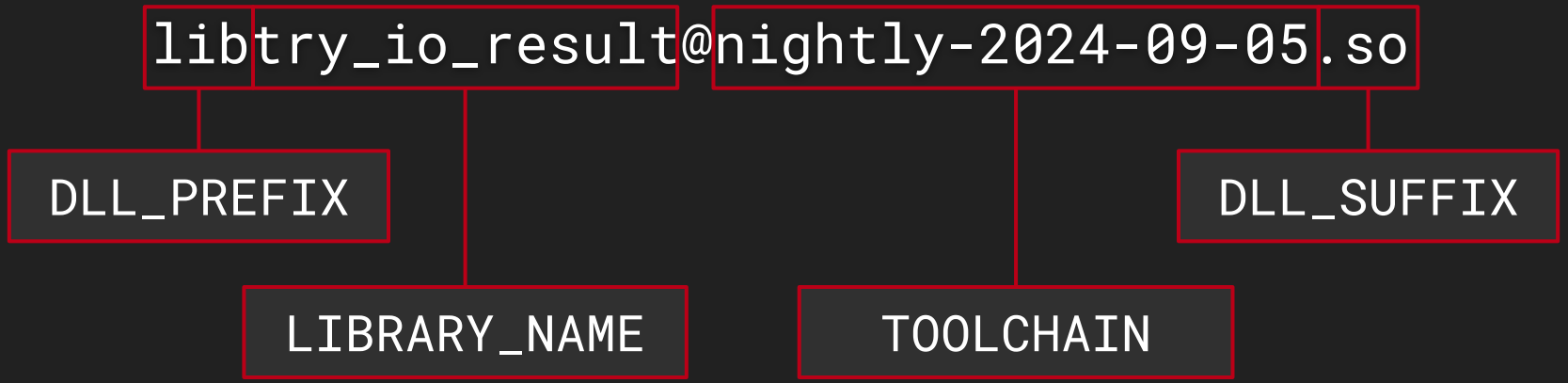
Concrete example on Linux:



Form of a Dylint library filename

```
DLL_PREFIX LIBRARY_NAME '@' TOOLCHAIN DLL_SUFFIX
```

Concrete example on Linux:



`dylint-link` creates a copy of your library with a filename of this form.

Example Dylint lint: `try_io_result`

https://github.com/trailofbits/dylint/tree/master/examples/restriction/try_io_result

$

```
$ cargo amaze --maximize=wow-factor
```

```
$ cargo amaze --maximize=wow-factor
```

```
$ cargo amaze --maximize=wow-factor
Error: No such file or directory (os error 2)
$
```

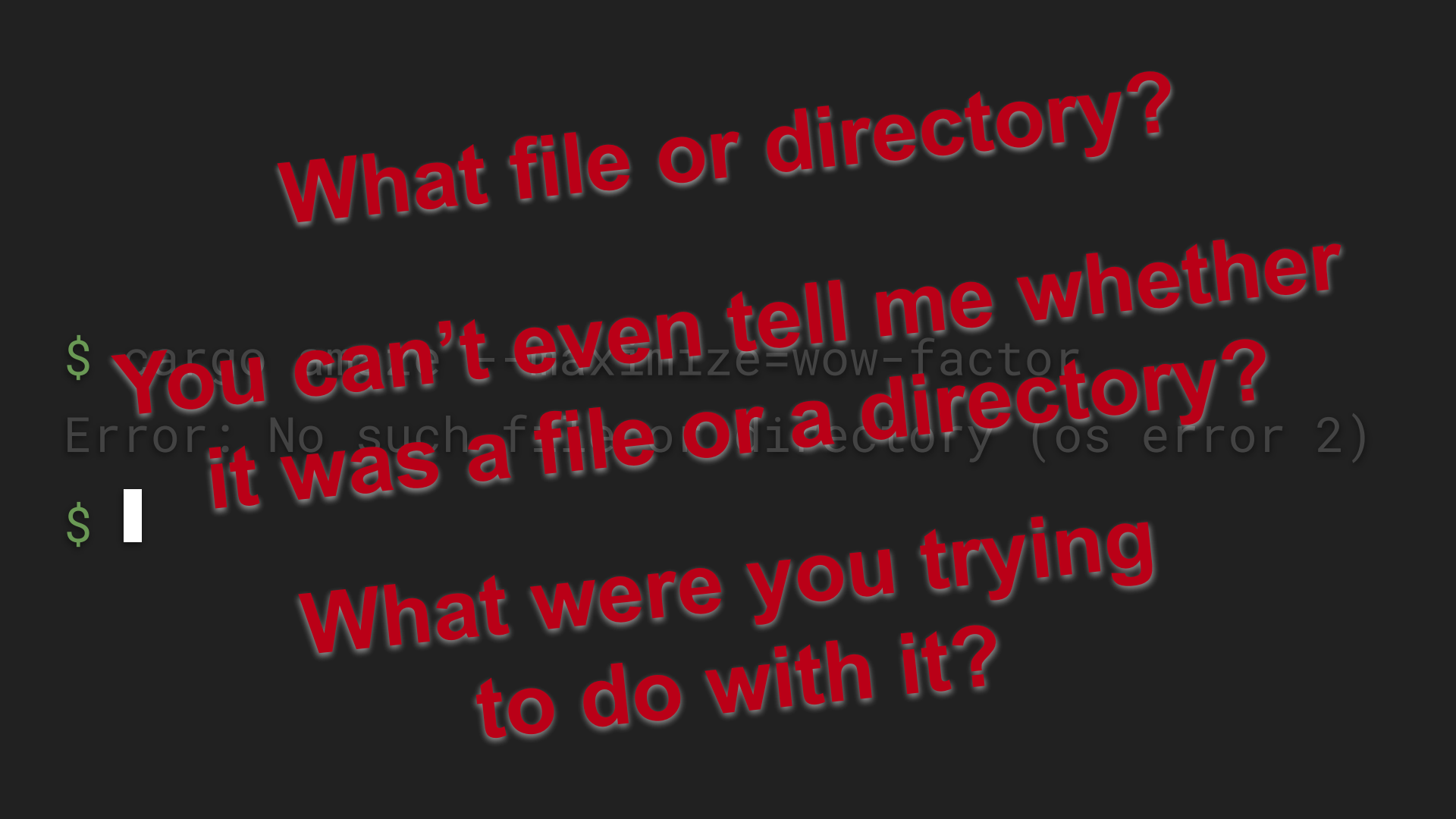
What file or directory?

```
$ cargo amaze --maximize=wow-factor
Error: No such file or directory (os error 2)
$ 
```

What file or directory?

```
$ cargo amaze --maximize=wow-factor
Error: No such file or directory (os error 2)
$
```

What were you trying to do with it?


What file or directory?
You can't even tell me whether it was a file or a directory?
What were you trying to do with it?
$ cargo cmake --maximize=wow-factor
Error: No such file or directory (os error 2)
$

```rust
fn foo() -> anyhow::Result<()> {
    let _ = File::open("/nonexistent")?;
    Ok(())
}
```

A function's return type allows additional information to be returned, but it goes unused.

```
fn foo() -> anyhow::Result<()> {
    let _ = File::open("/nonexistent")?;
    Ok(())
}
```

Something like `File::open` returns a non-descript `std::io::Result`.

```rust
fn foo() -> anyhow::Result<()> {
    let _ = File::open("/nonexistent")?;
    Ok(())
}
```

```rust
fn foo() -> anyhow::Result<()> {
    let _ = File::open("/nonexistent")?;

    Ok(())
}
```

```rust
fn foo() -> anyhow::Result<()> {
    let _ = File::open("/nonexistent")
        .with_context(|| "could not open `/nonexistent`")?;
    Ok(())
}
```

```
cargo dylint --git https://github.com/trailofbits/dylint
    --pattern examples/restriction/try_io_result
```

$

```
$ git clone https://github.com/trailofbits/dylint
```

```
$ git clone https://github.com/trailofbits/dylint
$
```

```
$ git clone https://github.com/trailofbits/dylint
$ cd dylint/examples/restriction
```

```
$ git clone https://github.com/trailofbits/dylint
$ cd dylint/examples/restriction
$ 
```

```
$ git clone https://github.com/trailofbits/dylint
$ cd dylint/examples/restriction
$ tree -a try_io_result
```

```
$ git clone https://github.com/trailofbits/dylint
$ cd dylint/examples/restriction
$ tree -a try_io_result
try_io_result
├── Cargo.toml
├── README.md
├── src
│   └── lib.rs
└── ui
    ├── main.rs
    └── main.stderr
```

```
$ git clone https://github.com/trailofbits/dylint
$ cd dylint/examples/restriction
$ tree -a try_io_result
try_io_result
├── Cargo.toml
├── README.md
├── src
│   └── lib.rs
└── ui
    ├── main.rs
    └── main.stderr
```

Structure of `try_io_result/src/lib.rs`

```
[features, extern crate declarations, and imports]

dylint_linting::declare_late_lint! {
    ...
}

impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    ...
}

fn is_io_result(cx: &LateContext<'_>, ty: Ty) -> bool { ... }

#[test]
fn ui() {
   dylint_testing::ui_test_examples(env!("CARGO_PKG_NAME"));
}
```

Structure of `try_io_result/src/lib.rs`

```
[features, extern crate declarations, and imports]
```

```
dylint_linting::declare_late_lint! {
    ...
}
```

2

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    ...
}
```

3

```
fn is_io_result(cx: &LateContext<'_>, ty: Ty) -> bool { ... }
```

```
#[test]
fn ui() {
   dylint_testing::ui_test_examples(env!("CARGO_PKG_NAME"));
}
```

1

Structure of `try_io_result/src/lib.rs`

```
[features, extern crate declarations, and imports]

dylint_linting::declare_late_lint! {
    ...
}

impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    ...
}

fn is_io_result(cx: &LateContext<'_>, ty: Ty) -> bool { ... }

#[test]
fn ui() {
   dylint_testing::ui_test_examples(env!("CARGO_PKG_NAME"));
}
```

Structure of `try_io_result/src/lib.rs`

```
[features, extern crate declarations, and imports]

dylint_linting::declare_late_lint! {
    ...
}

impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    ...
}

fn is_io_result(cx: &LateContext<'_>, ty: Ty) -> bool { ... }

#[test]
fn ui() {
    dylint_testing::ui_test_examples(env!("CARGO_PKG_NAME"));
}
```

Mostly boilerplate.

Suffice it to say `is_io_result` determines whether `ty` refers to a `std::io::Result`.

Structure of `try_io_result/src/lib.rs`

```
[features, extern crate declarations, and imports]

dylint_linting::declare_late_lint! {
    ...
}

impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    ...
}

fn is_io_result(cx: &LateContext<'_>, ty: Ty) -> bool { ... }

#[test]
fn ui() {
   dylint_testing::ui_test_examples(env!("CARGO_PKG_NAME"));
}
```

Structure of `try_io_result/src/lib.rs`

```
[features, extern crate declarations, and imports]

dylint_linting::declare_late_lint! {
    ...
}

impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    ...
}

fn is_io_result(cx: &LateContext<'_>, ty: Ty) -> bool { ... }

#[test]
fn ui() {
    dylint_testing::ui_test_examples(env!("CARGO_PKG_NAME"));
}
```

1

`try_io_result/ui/main.rs` (slightly reduced)

try_io_result/ui/main.rs (slightly reduced)

```
[...]

fn foo() -> anyhow::Result<()> {
    let _ = File::open("/nonexistent")?;
    Ok(())
}

fn foo_with_context() -> anyhow::Result<()> {
    let _ = File::open("/nonexistent").with_context(|| "could not open `/nonexistent`")?;
    Ok(())
}

fn bar() -> io::Result<()> {
    let _ = File::open("/nonexistent")?;
    Ok(())
}
```

try_io_result/ui/main.rs (slightly reduced)

```
[...]

fn foo() -> anyhow::Result<()> {
    let _ = File::open("/nonexistent")?;
    Ok(())
}

fn foo_with_context() -> anyhow::Result<()> {
    let _ = File::open("/nonexistent").with_context(|| "could not open `/nonexistent`")?;
    Ok(())
}

fn bar() -> io::Result<()> {
    let _ = File::open("/nonexistent")?;
    Ok(())
}
```

✘ Should flag

✔ Should not flag

✔ Should not flag

try_io_result/ui/main.stderr

try_io_result/ui/main.stderr

```
warning: returning a `std::io::Result` could discard relevant context (e.g., files or paths involved)
  --> $DIR/main.rs:9:13
   |
LL |     let _ = File::open("/nonexistent")?;
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = help: return a type that includes relevant context
   = note: `#[warn(try_io_result)]` on by default

warning: 1 warning emitted
```

Structure of `try_io_result/src/lib.rs`

```
[features, extern crate declarations, and imports]

dylint_linting::declare_late_lint! {
    ...
}

impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    ...
}

fn is_io_result(cx: &LateContext<'_>, ty: Ty) -> bool { ... }

#[test]
fn ui() {
   dylint_testing::ui_test_examples(env!("CARGO_PKG_NAME"));
}
```

Structure of `try_io_result/src/lib.rs`

```
[features, extern crate declarations, and imports]

dylint_linting::declare_late_lint! {
    ...
}

impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    ...
}

fn is_io_result(cx: &LateContext<'_>, ty: Ty) -> bool { ... }

#[test]
fn ui() {
   dylint_testing::ui_test_examples(env!("CARGO_PKG_NAME"));
}
```

declare_late_lint! for try_io_result

```
dylint_linting::declare_late_lint! {
    /// ### What it does
    /// Checks for `?` operators applied to values of type `std::io::Result`.
    ///
    /// ### Why is this bad?
    /// Returning a `std::io::Result` could mean relevant context (e.g., files or paths
    /// involved) is lost. The problem is discussed under "Verbose IO errors" in Yoshua Wuyts'
    /// [Error Handling Survey].
    ///
    /// ### Known problems
    /// No interprocedural analysis is done. ...
    [...]
    pub TRY_IO_RESULT,
    Warn,
    "`?` operator applied to `std::io::Result`"
}
```

declare_late_lint! for try_io_result

```
dylint_linting::declare_late_lint! {
    /// ### What it does
    /// Checks for `?` operators applied to values of type `std::io::Result`.
    ///
    /// ### Why is this bad?
    /// Return                                                    paths
    /// involv                                                    oshua Wuyts'
    /// [Error
    ///
    /// ### Kn
    /// No interprocedural analysis is done. ...
    [...]
    pub TRY_IO_RESULT,
    Warn,
    "`?` operator applied to `std::io::Result`"
}
```

Under the hood,

`dylint_linting::declare_late_lint!`

declares a `Lint` struct and prepares it to be run as a late lint.

declare_late_lint! for try_io_result

```
dylint_linting::declare_late_lint! {
    /// ### What it does
    /// Checks for `?` operators applied to values of type `std::io::Result`.
    ///
    /// ### Why is this bad?
    /// Returning a `std::io::Result` could mean relevant context (e.g., files or paths
    /// involved) is lost. The problem is discussed under "Verbose IO errors" in Yoshua Wuyts'
    /// [Error Handling Survey].
    ///
    /// ### Known problems
    /// No interprocedural analysis is done. ...
    [...]
    pub TRY_IO_RESULT,
    Warn,
    "`?` operator applied to `std::io::Result`"
}
```

declare_late_lint! for try_io_result

```
dylint_linting::declare_late_lint! {
    /// ### What it does
    /// Checks for `?` operators applied to values of type `std::io::Result`.
    ///
    /// ### Why is this bad?
    /// Returning a `std::io::Result` could mean relevant context (e.g., files or paths
    /// involved) is lost. The problem is discussed under "Verbose IO errors" in Yoshua Wuyts'
    /// [Error Handling Survey].
    ///
    /// ### Known problems
    /// No interprocedural analysis is done. ...
    [...]
    pub TRY_IO_RESULT,
    Warn,
    "`?` operator applied to `std::io::Result`"
}
```

Rustdoc comment describing the lint

declare_late_lint! for try_io_result

```
dylint_linting::declare_late_lint! {
    /// ### What it does
    /// Checks for `?` operators applied to values of type `std::io::Result`.
    ///
    /// ### Why is this bad?
    /// Returning a `std::io::Result` could mean relevant context (e.g., files or paths
    /// involved) is lost. The problem is discussed under "Verbose IO errors" in Yoshua Wuyts'
    /// [Error Handling Survey].
    ///
    /// ### Known problems
    /// No interprocedural analysis is done. ...
    [...]
    pub TRY_IO_RESULT,
    Warn,
    "`?` operator applied to `std::io::Result`"
}
```

declare_late_lint! for try_io_result

```
dylint_linting::declare_late_lint! {
    /// ### What it does
    /// Checks for `?` operators applied to values of type `std::io::Result`.
    ///
    /// ### Why is this bad?
    /// Returning a `std::io::Result` could mean relevant context (e.g., files or paths
    /// involved) is lost. The problem is discussed under "Verbose IO errors" in Yoshua Wuyts'
    /// [Error Handling Survey].
    ///
    /// ### Known problems
    /// No interprocedural analysis is done. ...
    [...]
    pub TRY_IO_RESULT,
    Warn,
    "`?` operator applied to `std::io::Result`"
}
```

Lint name

Lint level (usually `Warn`)

Short description for when a library's contents are listed

Structure of `try_io_result/src/lib.rs`

```
[features, extern crate declarations, and imports]

dylint_linting::declare_late_lint! {
    ...
}

impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    ...
}

fn is_io_result(cx: &LateContext<'_>, ty: Ty) -> bool { ... }

#[test]
fn ui() {
   dylint_testing::ui_test_examples(env!("CARGO_PKG_NAME"));
}
```

Structure of `try_io_result/src/lib.rs`

```
[features, extern crate declarations, and imports]

dylint_linting::declare_late_lint! {
    ...
}

impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    ...
}

fn is_io_result(cx: &LateContext<'_>, ty: Ty) -> bool { ... }

#[test]
fn ui() {
   dylint_testing::ui_test_examples(env!("CARGO_PKG_NAME"));
}
```

`LateLintPass` implementation for `try_io_result`

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if let ExprKind::Match(scrutinee, _, MatchSource::TryDesugar(_)) = expr.kind
            && let ExprKind::Call(callee, [arg]) = scrutinee.kind
            && let ExprKind::Path(path) = &callee.kind
            && matches!(path, QPath::LangItem(LangItem::TryTraitBranch, _))
            && let arg_ty = cx.typeck_results().node_type(arg.hir_id)
            && is_io_result(cx, arg_ty)
            && let body_owner_hir_id = cx.tcx.hir().enclosing_body_owner(expr.hir_id)
            && let body = cx.tcx.hir().body_owned_by(body_owner_hir_id)
            && let body_ty = cx.typeck_results().expr_ty(body.value)
            // smoelius: If the body's return type is `std::io::Result`, do not flag, because
            // the return type cannot carry any additional information.
            && !is_io_result(cx, body_ty)
        {
            (Continued)
```

`LateLintPass` implementation for `try_io_result`

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if let ExprKind::Match(scrutinee, _, MatchSource::TryDesugar(_)) = expr.kind
                                                                        nd
                                                                        tBranch, _))
            && let arg_ty = cx.typeck_results().node_type(arg.hir_id)
            && is_io_result(cx, arg_ty)
            && let body_owner_hir_id = cx.tcx.hir().enclosing_body_owner(expr.hir_id)
            && let body = cx.tcx.hir().body_owned_by(body_owner_hir_id)
            && let body_ty = cx.typeck_results().expr_ty(body.value)
            // smoelius: If the body's return type is `std::io::Result`, do not flag, because
            // the return type cannot carry any additional information.
            && !is_io_result(cx, body_ty)
        {
            (Continued)
```

Called on each expression in the crate being checked.

`LateLintPass` implementation for `try_io_result`

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if let ExprKind::Match(scrutinee, _, MatchSource::TryDesugar(_)) = expr.kind
            && let ExprKind::Call(callee, [arg]) = scrutinee.kind
            && let ExprKind::Path(path) = &callee.kind
            && matches!(path, QPath::LangItem(LangItem::TryTraitBranch, _))
            && let arg_ty = cx.typeck_results().node_type(arg.hir_id)
            && is_io_result(cx, arg_ty)
            && let body_owner_hir_id = cx.tcx.hir().enclosing_body_owner(expr.hir_id)
            && let body = cx.tcx.hir().body_owned_by(body_owner_hir_id)
            && let body_ty = cx.typeck_results().expr_ty(body.value)
            // smoelius: If the body's return type is `std::io::Result`, do not flag, because
            // the return type cannot carry any additional information.
            && !is_io_result(cx, body_ty)
        {
            (Continued)
```

LateLintPass implementation for try_io_result

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if let ExprKind::Match(scrutinee, _, MatchSource::TryDesugar(_)) = expr.kind
            && let ExprKind::Call(callee, [arg]) = scrutinee.kind
            && let ExprKind::Path(path) = &callee.kind
            && matches!(path, QPath::LangItem(LangItem::TryTraitBranch, _))
            && let arg_ty = cx.typeck_results().node_type(arg.hir_id)
            && is_io_resul
            && let body_ow                                    ner(expr.hir_id)
            && let body = cx.tcx.hir().body_owned_by(body_owner_hir_id)
            && let body_ty = cx.typeck_results().expr_ty(body.value)
            // smoelius: If the body's return type is `std::io::Result`, do not flag, because
            // the return type cannot carry any additional information.
            && !is_io_result(cx, body_ty)
        {
            (Continued)
```

Is the expression an application of ?.

`LateLintPass` implementation for `try_io_result`

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if let ExprKind::Match(scrutinee, _, MatchSource::TryDesugar(_)) = expr.kind
            && let ExprKind::Call(callee, [arg]) = scrutinee.kind
            && let ExprKind::Path(path) = &callee.kind
            && matches!(path, QPath::LangItem(LangItem::TryTraitBranch, _))
            && let arg_ty = cx.typeck_results().node_type(arg.hir_id)
            && is_io_result(cx, arg_ty)
            && let body_owner_hir_id = cx.tcx.hir().enclosing_body_owner(expr.hir_id)
            && let body = cx.tcx.hir().body_owned_by(body_owner_hir_id)
            && let body_ty = cx.typeck_results().expr_ty(body.value)
            // smoelius: If the body's return type is `std::io::Result`, do not flag, because
            // the return type cannot carry any additional information.
            && !is_io_result(cx, body_ty)
        {
            (Continued)
```

LateLintPass implementation for try_io_result

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if let ExprKind::Match(scrutinee, _, MatchSource::TryDesugar(_)) = expr.kind
            && let ExprKind::Call(callee, [arg]) = scrutinee.kind
            && let ExprKind::Path(path) = &callee.kind
            && matches!(pa                                          tBranch, _))
            && let arg_ty                                           ir_id)
            && is_io_resul
            && let body_owner_hir_id = cx.tcx.hir().enclosing_body_owner(expr.hir_id)
            && let body = cx.tcx.hir().body_owned_by(body_owner_hir_id)
            && let body_ty = cx.typeck_results().expr_ty(body.value)
            // smoelius: If the body's return type is `std::io::Result`, do not flag, because
            // the return type cannot carry any additional information.
            && !is_io_result(cx, body_ty)
        {
            (Continued)
```

Notably, `arg` is the expression to which the ? was applied.

`LateLintPass` implementation for `try_io_result`

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if let ExprKind::Match(scrutinee, _, MatchSource::TryDesugar(_)) = expr.kind
            && let ExprKind::Call(callee, [arg]) = scrutinee.kind
            && let ExprKind::Path(path) = &callee.kind
            && matches!(path, QPath::LangItem(LangItem::TryTraitBranch, _))
            && let arg_ty = cx.typeck_results().node_type(arg.hir_id)
            && is_io_result(cx, arg_ty)
            && let body_owner_hir_id = cx.tcx.hir().enclosing_body_owner(expr.hir_id)
            && let body = cx.tcx.hir().body_owned_by(body_owner_hir_id)
            && let body_ty = cx.typeck_results().expr_ty(body.value)
            // smoelius: If the body's return type is `std::io::Result`, do not flag, because
            // the return type cannot carry any additional information.
            && !is_io_result(cx, body_ty)
        {
            (Continued)
```

LateLintPass implementation for try_io_result

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if let ExprKind::Match(scrutinee, _, MatchSource::TryDesugar(_)) = expr.kind
            && let ExprKind::Call(callee, [arg]) = scrutinee.kind
            && let ExprKind::Path(path) = &callee.kind
            && matches!(path, QPath::LangItem(LangItem::TryTraitBranch, _))
            && let arg_ty = cx.typeck_results().node_type(arg.hir_id)
            && is_io_result(cx, arg_ty)
            && let body_owner_hir_id = cx.tcx.hir().enclosing_body_owner(expr.hir_id)
            && let                                                    _hir_id)
            && let                                                    alue)
            // smoelius: If the body's return type is `std::io::Result`, do not flag, because
            // the return type cannot carry any additional information.
            && !is_io_result(cx, body_ty)
        {
            (Continued)
```

Is `arg`'s type `std::io::Result`?

`LateLintPass` implementation for `try_io_result`

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if let ExprKind::Match(scrutinee, _, MatchSource::TryDesugar(_)) = expr.kind
            && let ExprKind::Call(callee, [arg]) = scrutinee.kind
            && let ExprKind::Path(path) = &callee.kind
            && matches!(path, QPath::LangItem(LangItem::TryTraitBranch, _))
            && let arg_ty = cx.typeck_results().node_type(arg.hir_id)
            && is_io_result(cx, arg_ty)
            && let body_owner_hir_id = cx.tcx.hir().enclosing_body_owner(expr.hir_id)
            && let body = cx.tcx.hir().body_owned_by(body_owner_hir_id)
            && let body_ty = cx.typeck_results().expr_ty(body.value)
            // smoelius: If the body's return type is `std::io::Result`, do not flag, because
            // the return type cannot carry any additional information.
            && !is_io_result(cx, body_ty)
        {
            (Continued)
```

`LateLintPass` implementation for `try_io_result`

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if let ExprKind::Mat                                                    pr.kind
            && let ExprKind:
            && let ExprKind:
            && matches!(path
            && let arg_ty =
            && is_io_result(cx, arg_ty)
            && let body_owner_hir_id = cx.tcx.hir().enclosing_body_owner(expr.hir_id)
            && let body = cx.tcx.hir().body_owned_by(body_owner_hir_id)
            && let body_ty = cx.typeck_results().expr_ty(body.value)
            // smoelius: If the body's return type is `std::io::Result`, do not flag, because
            // the return type cannot carry any additional information.
            && !is_io_result(cx, body_ty)
        {
            (Continued)
```

Is the type of the body from which the result is returned `std::io::Result`? If so, don't flag.

`LateLintPass` implementation for `try_io_result`

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if let ExprKind::Match(scrutinee, _, MatchSource::TryDesugar(_)) = expr.kind
            && let ExprKind::Call(callee, [arg]) = scrutinee.kind
            && let ExprKind::Path(path) = &callee.kind
            && matches!(path, QPath::LangItem(LangItem::TryTraitBranch, _))
            && let arg_ty = cx.typeck_results().node_type(arg.hir_id)
            && is_io_result(cx, arg_ty)
            && let body_owner_hir_id = cx.tcx.hir().enclosing_body_owner(expr.hir_id)
            && let body = cx.tcx.hir().body_owned_by(body_owner_hir_id)
            && let body_ty = cx.typeck_results().expr_ty(body.value)
            // smoelius: If the body's return type is `std::io::Result`, do not flag, because
            // the return type cannot carry any additional information.
            && !is_io_result(cx, body_ty)
        {
            (Continued)
```

`LateLintPass` implementation for `try_io_result`

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if [...]
        {
            span_lint_and_help(
                cx,
                TRY_IO_RESULT,
                expr.span,
                "returning a `std::io::Result` could discard relevant context (e.g., files \
                 or paths involved)",
                None,
                "return a type that includes relevant context",
            );
        }
    }
}
```

`LateLintPass` implementation for `try_io_result`

```
impl<'tcx> LateLintPass<'tcx> for TryIoResult {
    fn check_expr(&mut self, cx: &LateContext<'tcx>, expr: &'tcx Expr<'_>) {
        if [...]
        {
            span_lint_and_help(
                cx,
                TRY_IO_RESULT,
                expr.span,
                "returning a `std::io::Result` could discard relevant context (e.g., files \
                 or paths involved)",
                None,
                "return a type that includes relevant context",
            );
        }
    }
}
```

Lint name

Span to highlight

Warning message

Help message

try_io_result/ui/main.stderr

```
warning: returning a `std::io::Result` could discard relevant context (e.g., files or paths involved)
  --> $DIR/main.rs:9:13
   |
LL |     let _ = File::open("/nonexistent")?;
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = help: return a type that includes relevant context
   = note: `#[warn(try_io_result)]` on by default

warning: 1 warning emitted
```

try_io_result/ui/main.stderr

```
warning: returning a `std::io::Result` could discard relevant context (e.g., files or paths involved)
  --> $DIR/main.rs:9:13
   |
LL |     let _ = File::open("/nonexistent")?;
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = help: return a type that includes relevant context
   = note: `#[warn(try_io_result)]` on by default

warning: 1 warning emitted
```

Warning message

Span to highlight

Help message

Resources

Resources (1 of 2)

- Clippy Documentation: Adding a new lint
- Clippy's `author` attribute
- Rust Compiler Development Guide

(Continued)

Resources (2 of 2)

- `rustc_lint::LateContext`
- `rustc_middle::ty::typeck_results::TypeckResults`
  - Returned by `rustc_lint::LateContext::typeck_results`
- `rustc_middle::ty::context::TyCtxt`
  - Type of `rustc_lint::LateContext`'s `tcx` field
- `rustc_middle::hir::map::Map`
  - Returned by `rustc_middle::ty::context::TyCtxt::hir`
- `clippy_utils`
  - Generously provided by the Clippy team! 🙏

Future work

Future work

- Make writing Rust lints easier, generally!
- Automatically infer and apply fixes required for API changes
  - Clippy is mirrored in the Rust repo.
  - Fixes required for API changes are in Clippy's commit history.
  - Would it be possible to extract them and apply them to other code relying on the same APIs?

Dylint

https://github.com/trailofbits/dylint

Thank you. Questions?

Samuel Moelius

sam.moelius@trailofbits.com

TRAIL OF BITS