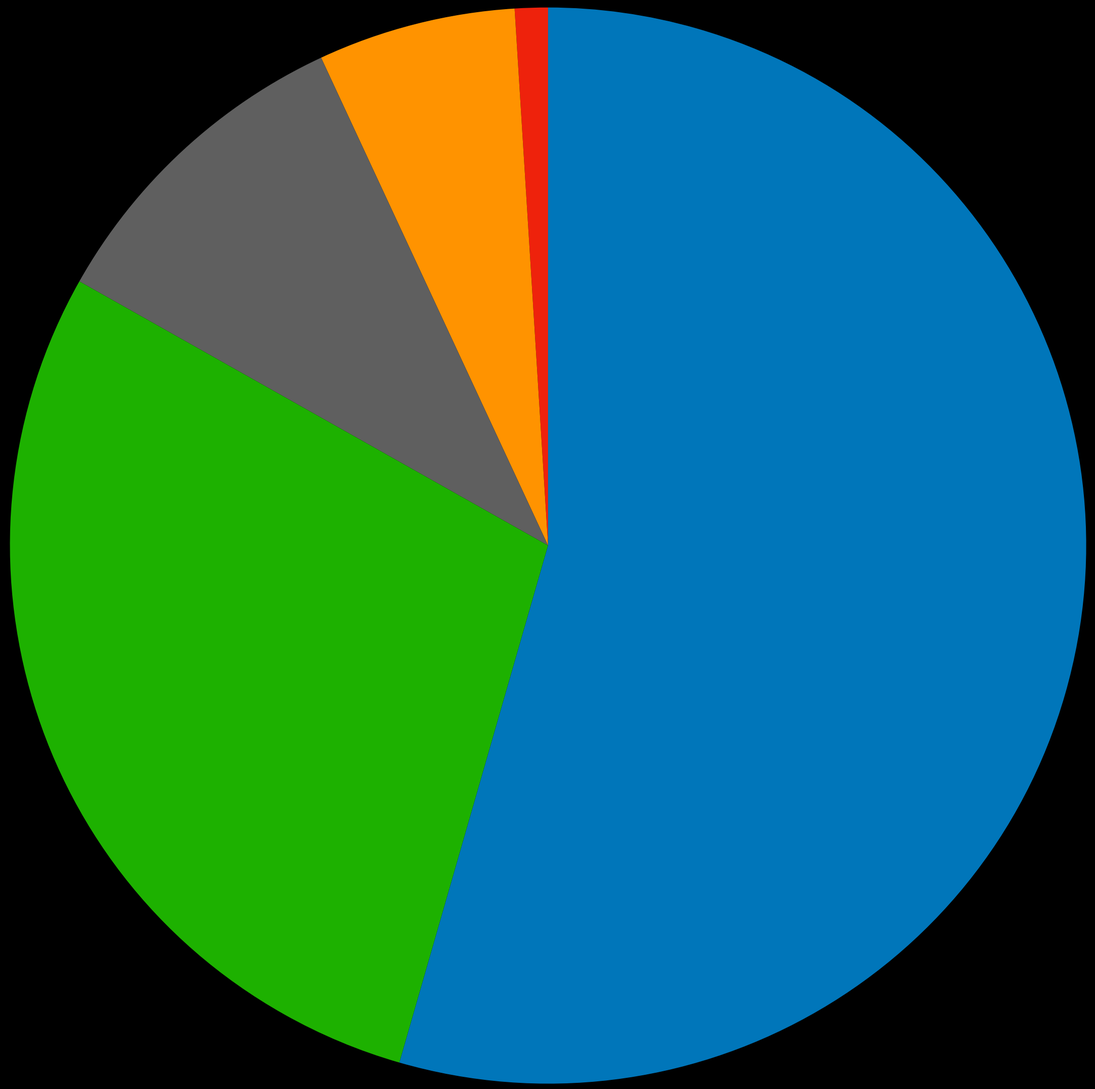
Demistifying Rust Debugging

RustLab, Florence
November 3rd, 2025

Vitaly Bragilevsky,
Developer Advocate at JetBrains

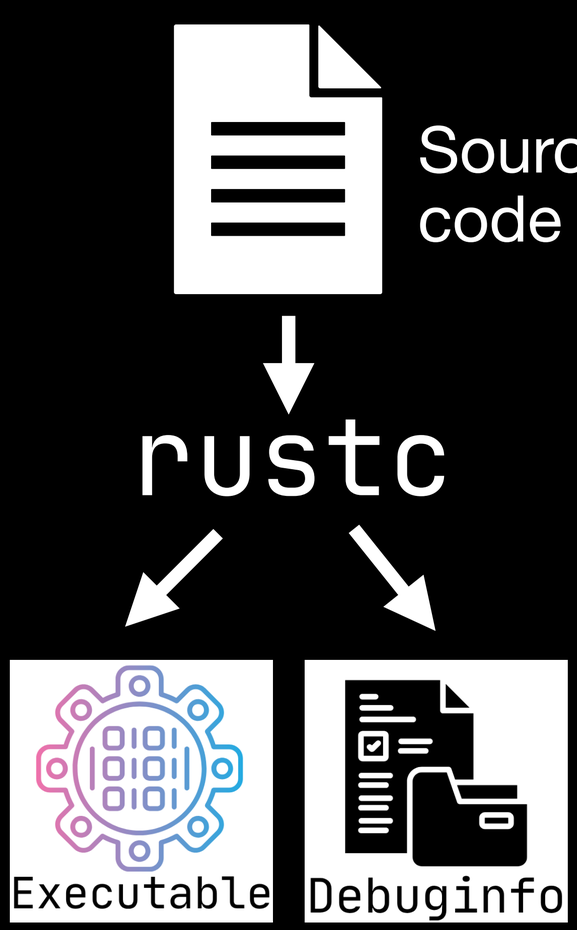
RustRover
JETBRAINS IDE

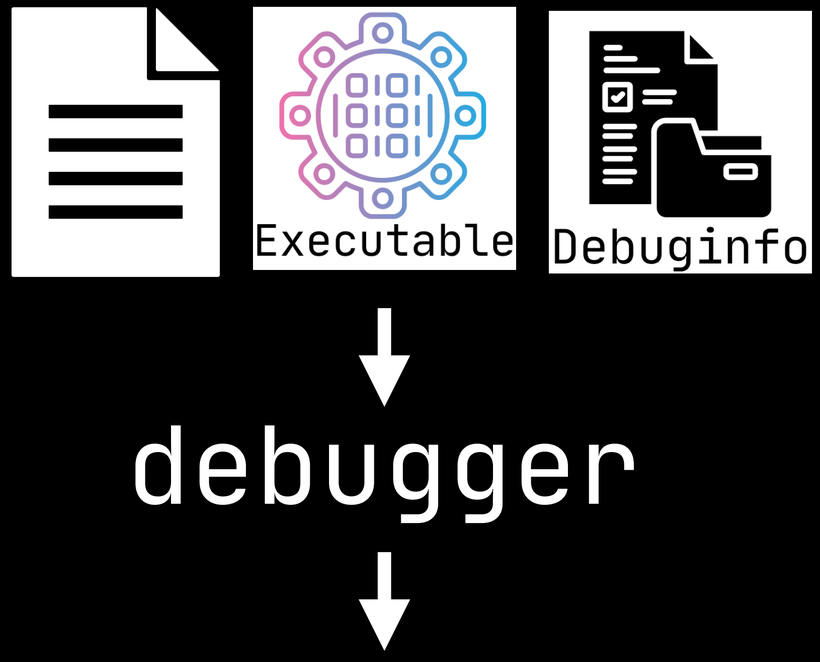
How do you usually debug your Rust code?



println!/dbg!
In IDE
In console
I don't debug at all
Other

Debugging native executables 101



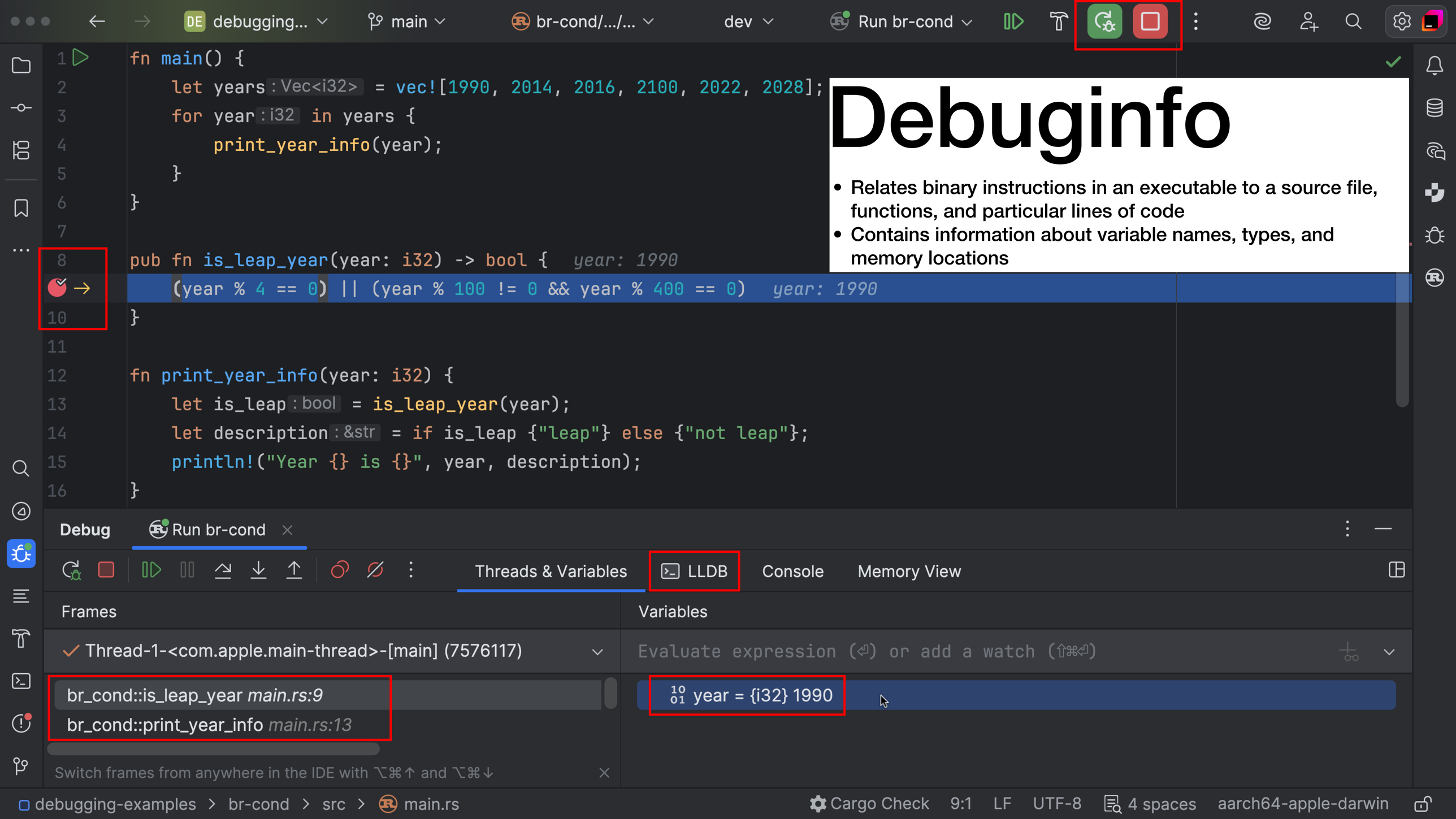


- Execute
- Suspend → explore current state

Debugging native executables 101

What can be done with a debugger?

- Step-by-step (line-by-line) execution
- Breakpoints (incl. conditional, non-suspending)
- Run to cursor (in UI debugging)
- Watches (incl. suspending on changes)
- Exploring stack frames, threads, variables, CPU registers, memory
- Changing values at runtime
- Replaying code fragments


debugging...
main
br-cond/.../...
dev
Run br-cond
fn main() {
let years : Vec<i32> = vec![1990, 2014, 2016, 2100, 2022, 2028];
for year : i32 in years {
print_year_info(year);
}
}
pub fn is_leap_year(year: i32) -> bool { year: 1990
(year % 4 == 0) || (year % 100 != 0 && year % 400 == 0) year: 1990
}
fn print_year_info(year: i32) {
let is_leap : bool = is_leap_year(year);
let description : &str = if is_leap {"leap"} else {"not leap"};
println!("Year {} is {}", year, description);
}
Debuginfo
Relates binary instructions in an executable to a source file, functions, and particular lines of code
Contains information about variable names, types, and memory locations
Debug
Run br-cond
Threads & Variables
LLDB
Console
Memory View
Frames
Variables
Thread-1-<com.apple.main-thread>-[main] (7576117)
Evaluate expression (⏎) or add a watch (⇧⌘⏎)
br_cond::is_leap_year main.rs:9
br_cond::print_year_info main.rs:13
year = {i32} 1990
Switch frames from anywhere in the IDE with ⌥⌘↑ and ⌥⌘↓
debugging-examples
br-cond
src
main.rs
Cargo Check
9:1
LF
UTF-8
4 spaces
aarch64-apple-darwin

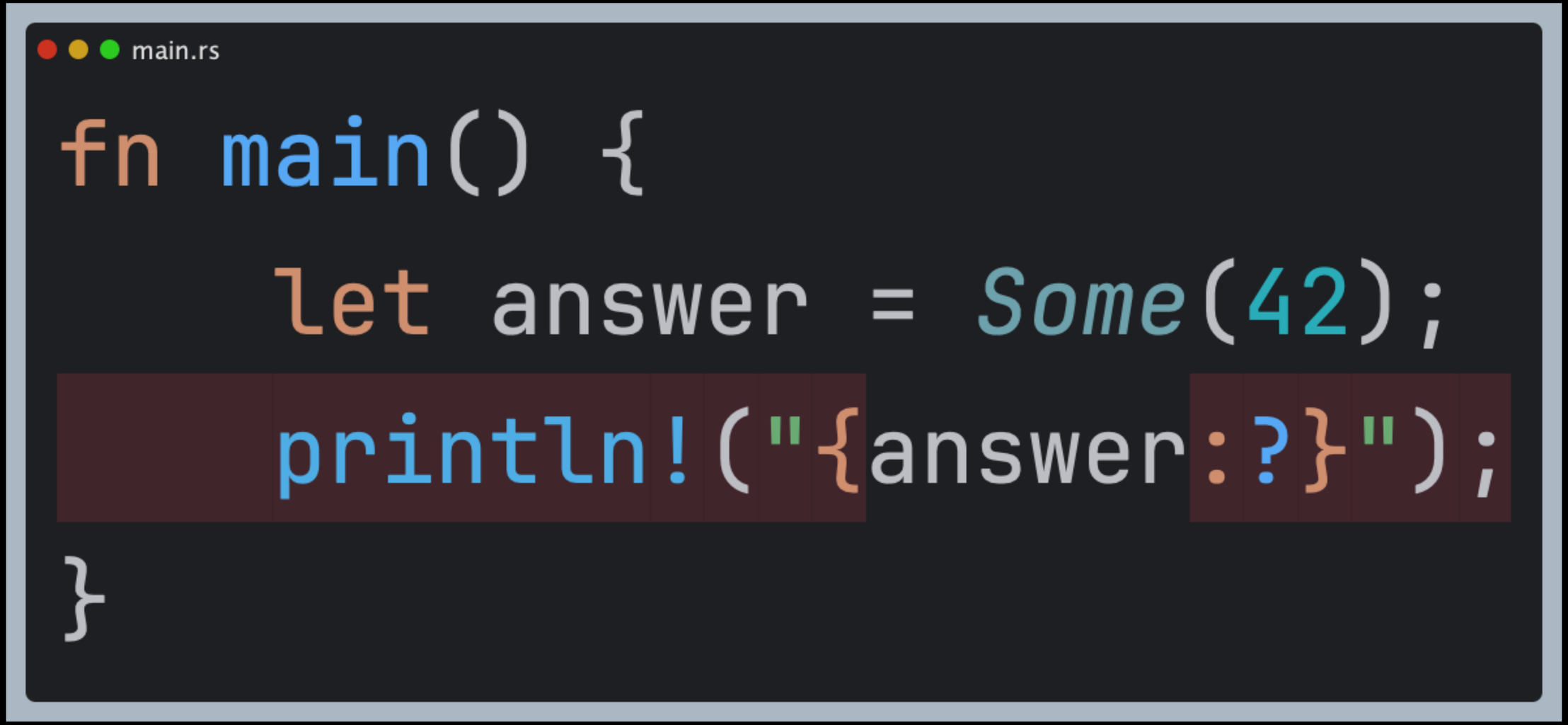
Debugging native executables: a bit deeper

- Debuginfo formats:
  - Windows: PDB-files (program database, a separate file)
  - Linux/macOS: DWARF (included in an executable)
- Low-level debuggers: gdb, lldb, WinDbg (with GUI)
- Wrappers for gdb/lldb:
  - rust-lldb/rust-gdb (come with rustc)
  - CodeLLDB (VS Code, Zed, Neovim,…)
  - RustRover debugger

NB: Debuginfo is language-agnostic

NB: low-level debuggers were never meant to support Rust

Simple example



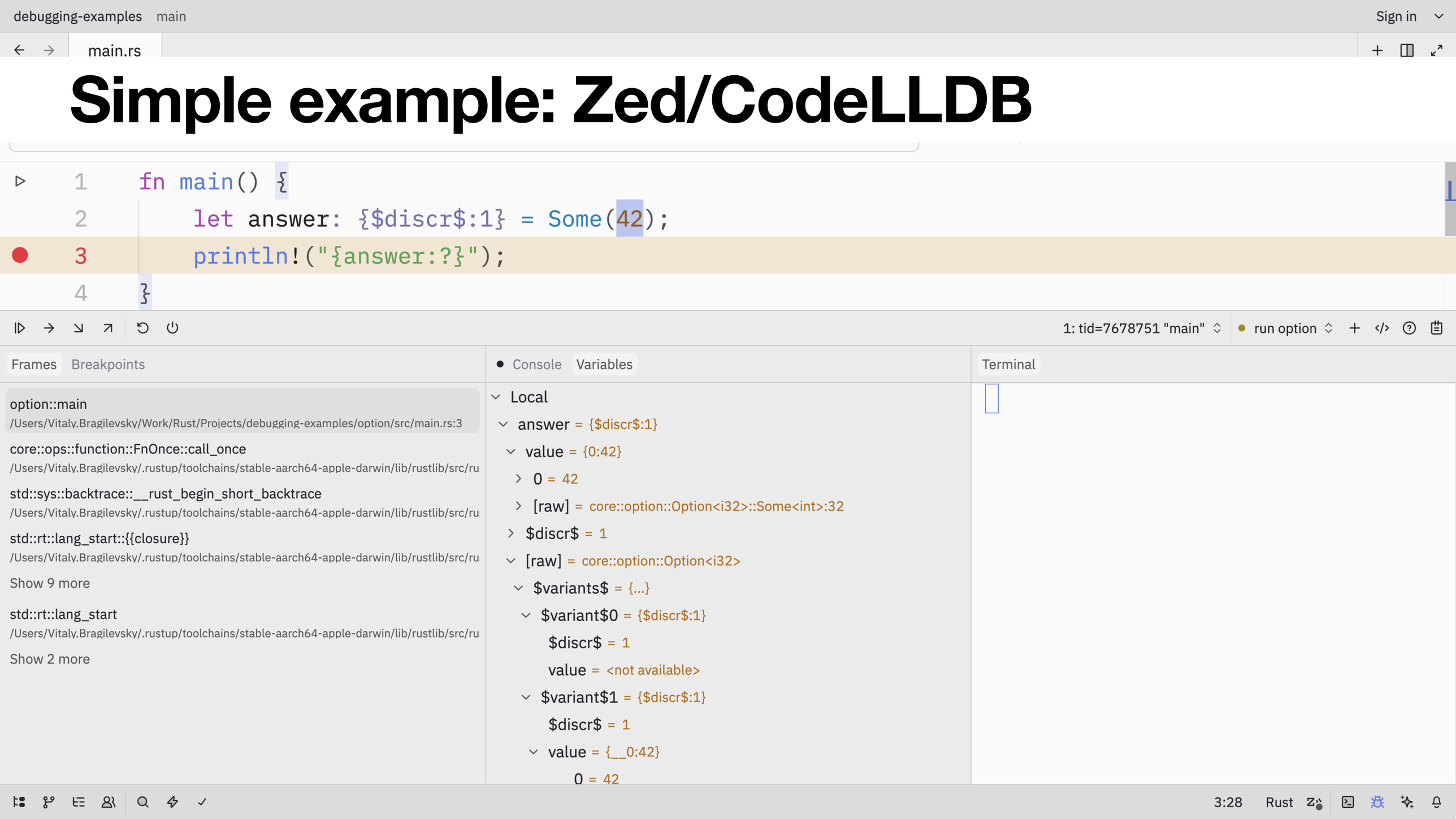
Simple example: vanilla lldb

```
$ lldb target/debug/option
(core::option::Option<i32>) {/Rust/Projects/debugging-examples/target/
  $variants$ = {
    $variant$0 = ($discr$ = 1,
     value = core::option::Option<i32>::None<int>:32
             @ 0x000000016fdfe538)eason = breakpoint 1.1
    $variant$1 = {tion`option::main::h2d75a6b059e1f431 at main.rs:3:5
      $discr$ = 1
      value = (__0 = 42)
    }tion::Option<i32>) {
  }ts$ = {
  variant$0 = ($discr$ = 1, value = core::option::Option<i32>::None<int>:32 @ 0x000000016fdfe538)
  }variant$1 = {
    $discr$ = 1
}   value = (__0 = 42)
  }
 }
}
```

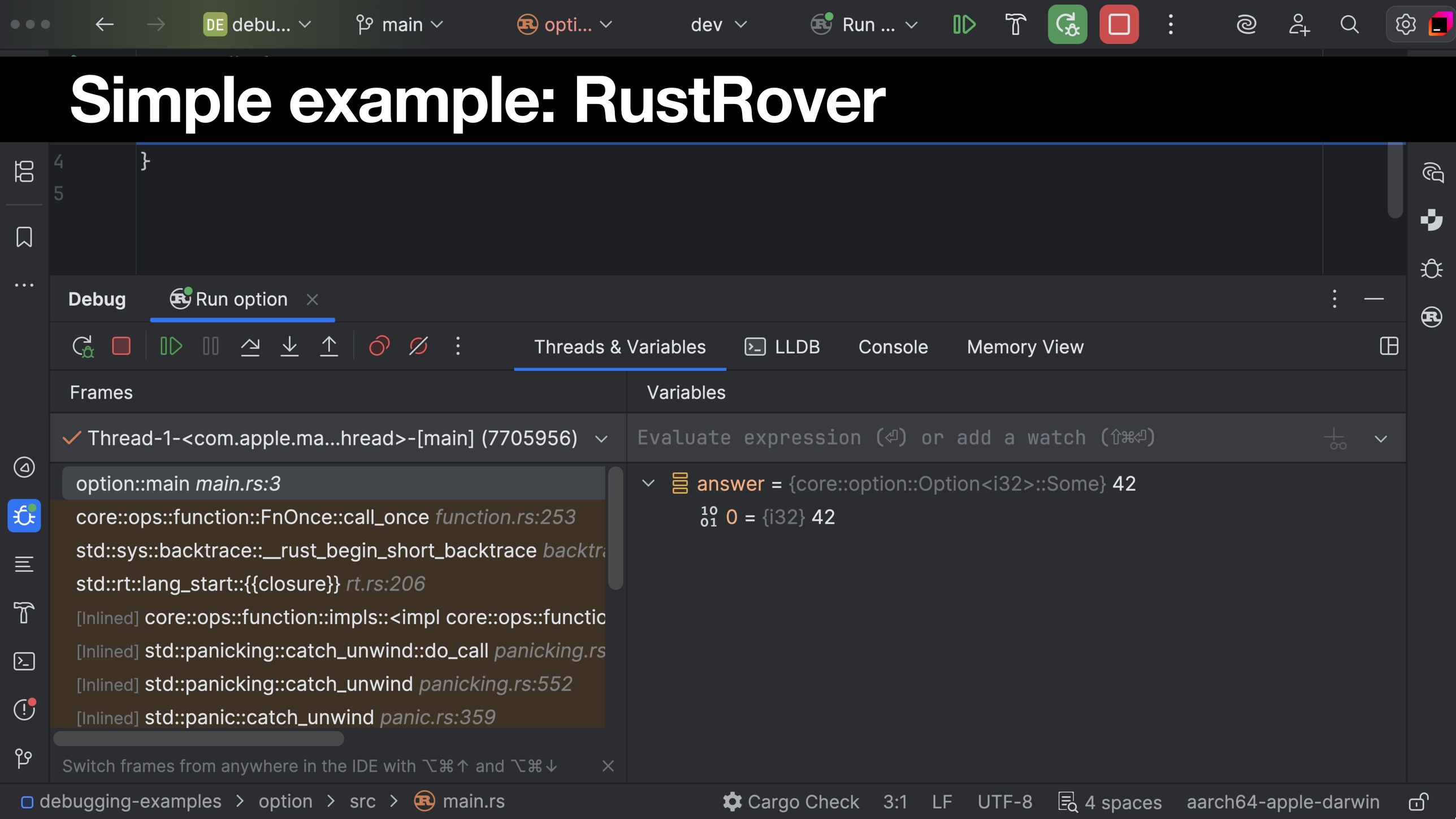
Simple example: rust-lldb

```
$ rust-lldb target/debug/option
(lldb) command script import "/Users/Vitaly.Bragilevsky/.rustup/toolchains/stable-aarch64-apple-darwin/lib/rustlib/etc/lldb_lookup.py"
(lldb) command source -s 0 '/Users/Vitaly.Bragilevsky/.rustup/toolchains/stable-aarch64-apple-darwin/lib/rustlib/etc/lldb_commands'
Executing commands in '/Users/Vitaly.Bragilevsky/.rustup/toolchains/stable-aarch64-apple-darwin/lib/rustlib/etc/lldb_commands'.
(lldb) <…SKIPPED…>
(lldb) target create "target/debug/option"
Current executable set to '/Users/Vitaly.Bragilevsky/Work/Rust/Projects/debugging-examples/target/debug/option' (arm64).
(lldb) b main.rs:3
Breakpoint 1: 2 locations.
(lldb) run
Process 24823 launched: '/Users/Vitaly.Bragilevsky/Work/Rust/Projects/debugging-examples/target/debug/option' (arm64)
Process 24823 stopped
* thread #1, name = 'main', queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100000be8 option`option::main::h2d75a6b059e1f431 at main.rs:3:5
   1   fn main() {
   2       let answer = Some(42);
-> 3       println!("{answer:?}");
   4   }
(lldb) print answer
(core::option::Option<i32>) {
  value = {
    0 = 42
  }
  $discr$ = 1
}
```

```
(core::option::Option<i32>) {
  $variants$ = {
    $variant$0 = ($discr$ = 1, value = core::option::Option<i32>::None<int>:32 @ 0x000000016fdfe538)
    $variant$1 = {
      $discr$ = 1
      value = (__0 = 42)
    }
  }
}
```


debugging-examples main
Sign in
main.rs
Simple example: Zed/CodeLLDB
1 fn main() {
2 let answer: {$discr$:1} = Some(42);
3 println!("{answer:?}");
4 }
1: tid=7678751 "main"
run option
Frames
Breakpoints
option::main
/Users/Vitaly.Bragilevsky/Work/Rust/Projects/debugging-examples/option/src/main.rs:3
core::ops::function::FnOnce::call_once
/Users/Vitaly.Bragilevsky/.rustup/toolchains/stable-aarch64-apple-darwin/lib/rustlib/src/ru
std::sys::backtrace::__rust_begin_short_backtrace
/Users/Vitaly.Bragilevsky/.rustup/toolchains/stable-aarch64-apple-darwin/lib/rustlib/src/ru
std::rt::lang_start::{{closure}}
/Users/Vitaly.Bragilevsky/.rustup/toolchains/stable-aarch64-apple-darwin/lib/rustlib/src/ru
Show 9 more
std::rt::lang_start
/Users/Vitaly.Bragilevsky/.rustup/toolchains/stable-aarch64-apple-darwin/lib/rustlib/src/ru
Show 2 more
Console
Variables
Terminal
Local
answer = {$discr$:1}
value = {0:42}
0 = 42
[raw] = core::option::Option<i32>::Some<int>:32
$discr$ = 1
[raw] = core::option::Option<i32>
$variants$ = {...}
$variant$0 = {$discr$:1}
$discr$ = 1
value = <not available>
$variant$1 = {$discr$:1}
$discr$ = 1
value = {__0:42}
0 = 42
3:28
Rust

Simple example: RustRover



What’s the difference here?

- Vanilla LLDB doesn’t know anything about Rust specifics
- Rust-LLDB provides pretty-printers for Rust data types
- RustRover uses its own fork of LLDB with extensions for Rust Type System
- CodeLLDB used to have another fork of LLDB, but not anymore

Brief history of the LLDB forks

- Tom Tromey wrote the original implementation of Rust support for both GDB and LLDB
- There were issues with pushing it to the upstream GDB/LLDB
- Vadim Chugunov (CodeLLDB) maintained this fork (for a long time, but then decided to stop (in late 2024):
  - it’s too hard to keep up with changes in both `rustc` and the upstream debugger;
  - the upstream debugger is **good enough** in Rust support (along with Zed and Swift).
- JetBrains still maintains our own (private) fork based on Tom Tromey's and Vadim Chugunov’s work and uses it across all our products (RustRover, CLion, Rider) – this fork is tightly coupled with our UI debugger components.

More about pretty-printers for LLDB

Example: Printing a Rust vector (1)

```
class StdVecSyntheticProvider:
    """Pretty-printer for alloc::vec::Vec<T>

    struct Vec<T> { buf: RawVec<T>, len: usize }
    rust 1.75: struct RawVec<T> { ptr: Unique<T>, cap: usize, ... }
    rust 1.76: struct RawVec<T> { ptr: Unique<T>, cap: Cap(usize), ... }
    rust 1.31.1: struct Unique<T: ?Sized> { pointer: NonZero<*const T>, ... }
    rust 1.33.0: struct Unique<T: ?Sized> { pointer: *const T, ... }
    rust 1.62.0: struct Unique<T: ?Sized> { pointer: NonNull<T>, ... }
    struct NonZero<T>(T)
    struct NonNull<T> { pointer: *const T }
    “""

    def __init__(self, valobj: SBValue, _dict: LLDBOpaque):
       self.valobj = valobj
       self.update()
```

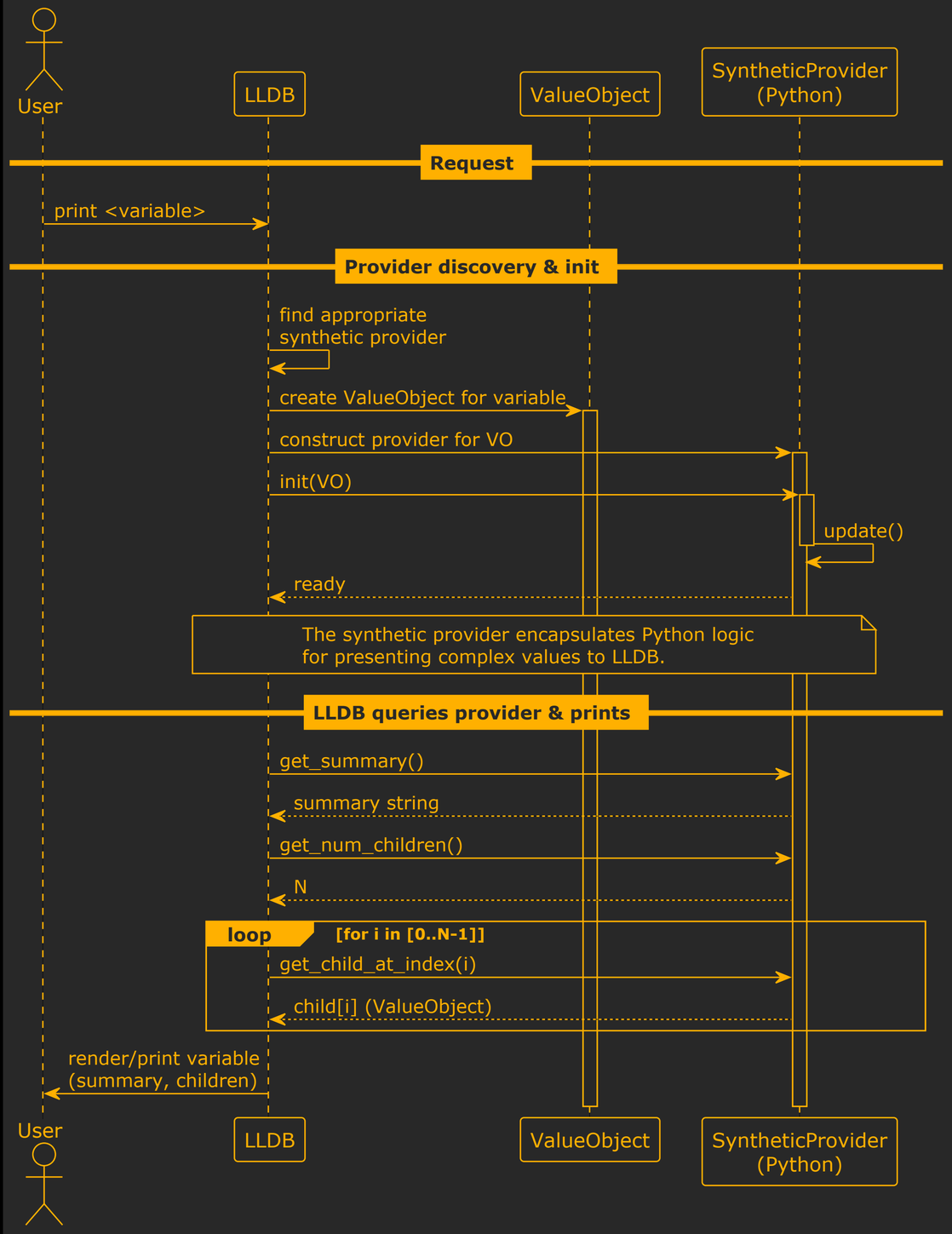
More about pretty-printers for LLDB

Example: Printing a Rust vector (2)

```
def update(self):
    self.length = self.valobj.GetChildMemberWithName("len").GetValueAsUnsigned()
    self.buf = self.valobj.GetChildMemberWithName("buf").GetChildMemberWithName(
        "inner"
    )

    self.data_ptr = unwrap_unique_or_non_null(
        self.buf.GetChildMemberWithName("ptr")
    )

    self.element_type = self.valobj.GetType().GetTemplateArgumentType(0)

    if not self.element_type.IsValid():
        element_name = get_template_args(self.valobj.GetTypeName())[0]
        self.element_type = self.valobj.target.FindFirstType(element_name)

    self.element_type_size = self.element_type.GetByteSize()
```

LLDB pretty-printers flow

- Pretty-printer can hide something unnecessary and evaluate some new information based on the actual **value object** provided.
- `rust-lldb` is basically:
  - Vanilla LLDB
  - Provider discovery scripts
  - Pretty-printers for value objects



Hey! I already have `Debug`/`Display` instances for my type.

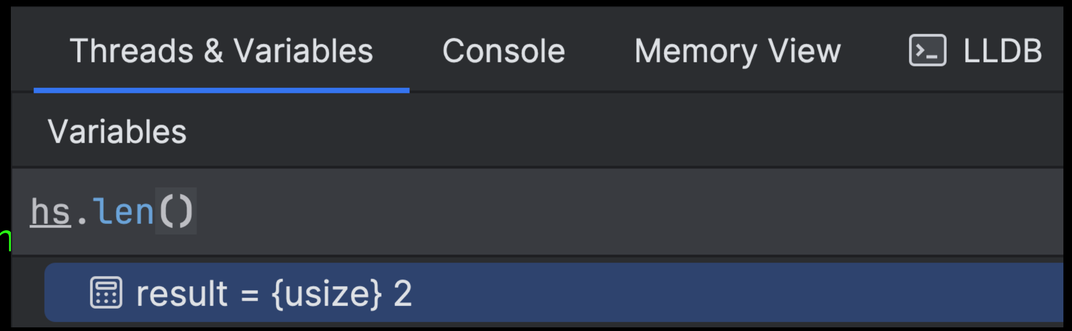
Why not using them?

Why we can’t use Debug/Display instances

At least for now

- Rust compiler doesn’t generate debug info regarding traits implemented for a type (see #33014 in Rust repo on GitHub).
- Even if it does, there are many issues with evaluating expressions in the context of a program in a debugging session.

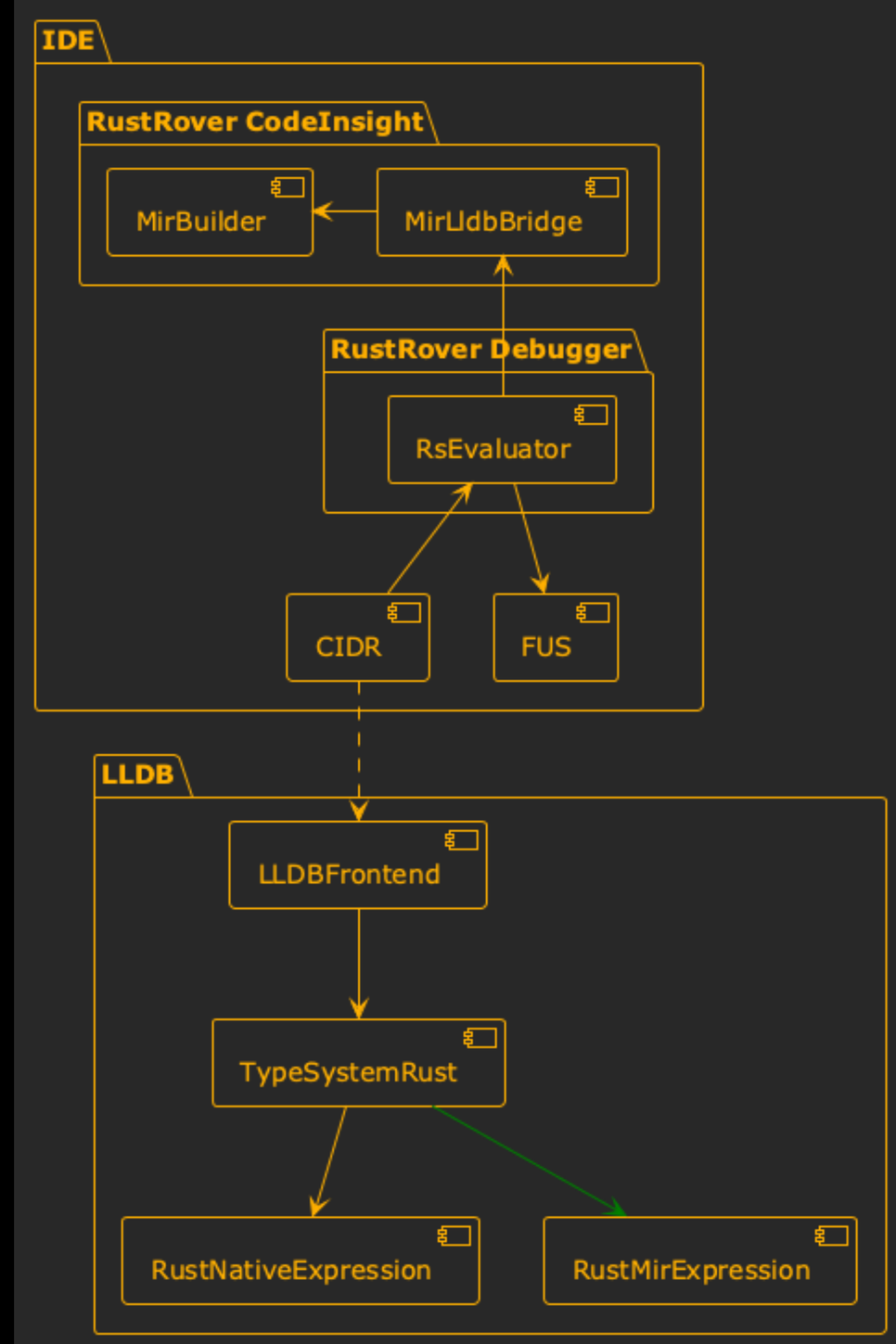
```
Process 27818 stopped
* thread #1, name = 'main', queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100005ba4 option`option::main::h2d75a6b059e1f431 at main.rs:9:23
   6        let mut hs = HashSet::new();
   7        hs.insert(1);
   8        hs.insert(2);
-> 9        println!("{}", hs.len())
   10   }
(lldb) print hs.len()
error: <user expression 0>:1:4: no member named 'len' in
std::hash::random::RandomState>'
    1 | hs.len()
      | ~~ ^
```



Expression evaluation

In a debugging session

- We type an expression
- This expression is parsed
- Some simple (native) expressions can be evaluated directly by debugger, but not all of them – debugger simply doesn’t have enough information
- So we might need to run a compiler for our expression during the debugging session!
- In RustRover, we recently started doing partial compilation to MIR to be able to evaluate more expressions within a debugging session



Quick troubleshooting guide

- Why don’t I see my variable?
- Why doesn’t the debugger stop at my breakpoint?
- Why do I see some garbage instead of my variable?
- Why can’t I evaluate my expression?
- Why is the debugger so slow?

Maybe, I'll continue println!/dbg!
my variables…

Arguments in favour of interactive debugging

- Fine-grained control and inspection
- Non-intrusive workflow
- Easily explore control flow
- Dive into legacy or 3rd party code
- Shorten edit–compile–run–print–analyze loop
- Build a better mental model of your data
- Debugging asynchronous code interactively can be easier (but not right now!)

If you use a debugger and report bugs, the debugger experience will be improved eventually.

Let’s do that together!