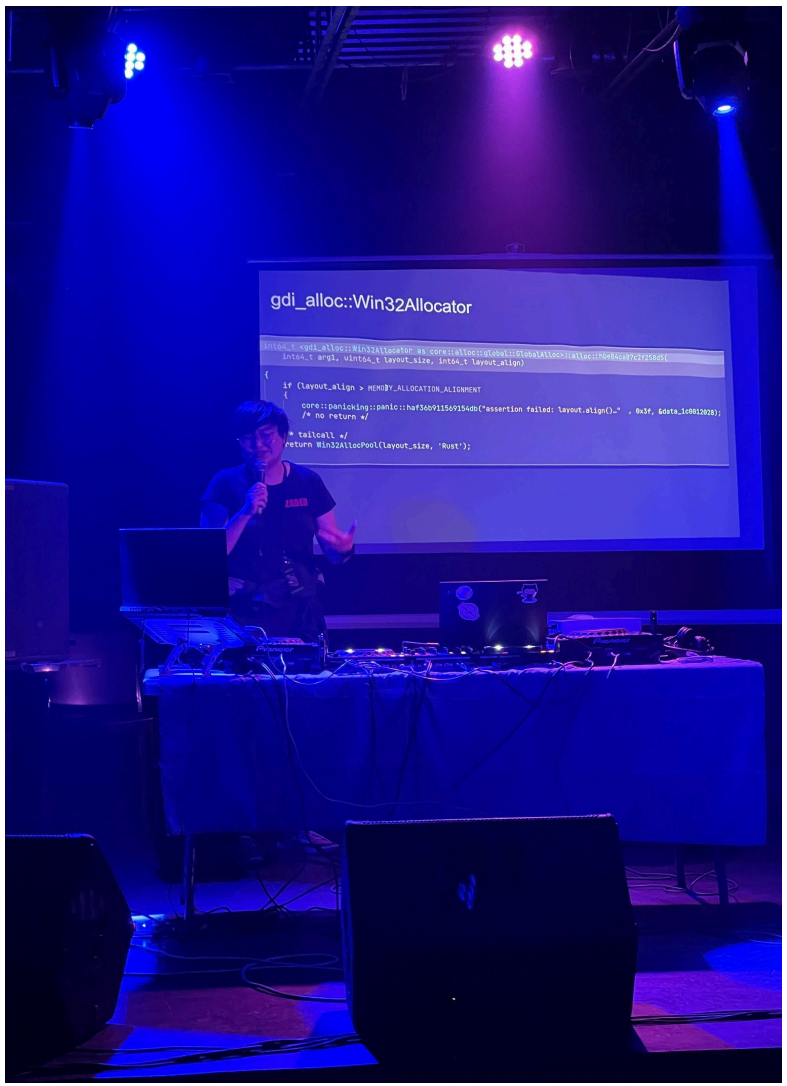
Reconstructing Rust Types

A Practical Guide for Reverse Engineers

Cindy Xiao
(she/her, they/them)
Senior Security Researcher, CrowdStrike

February 28, 2025

My Background



- I currently do malware analysis, reverse engineering, and cyber threat intelligence.
- I used to be a C/C++ developer, and I'm interested in approaching things from the developer's perspective.
- We were seeing more Rust malware in our analysis queue, and we needed practical skills to deal with them.

Some of my previous work on Rust reversing:

- 2023 - Analysis of Rust code for GDI in the Windows Kernel - Lightning Talk @ RECon
- 2023 - Rust Type Layout Helper - Binary Ninja Plugin
- 2023 - Rust String Slicer - Binary Ninja Plugin
- 2023 - Using panic metadata to recover source code information from Rust binaries - Blog Post
- 2024 - Reversing Rust Binaries: One step beyond strings - Workshop @ NorthSec
- 2024 - Reversing Rust Binaries: One step beyond strings - Workshop @ RECon

The State of Rust Reversing In 2025

Rust RE skills needed for:

- Malware
  - → Remote access tools (RATs), downloaders, loaders, ransomware, infostealers, adware, etc…
- Windows kernel
- Windows drivers
- Android libraries
- Maybe everything???

Rust is becoming increasingly popular as a *general purpose systems programming language*.

The State of Rust Reversing In 2025

We have:

- Function signatures for the standard library (e.g. shipped with IDA)
- The ability to generate function signatures for third-party libraries (e.g. rustbinsign)
- Some basic information on the metadata that you get from dumping a binary's strings

  → See 2024 - Reversing Rust Binaries: One step beyond strings - Workshop @ RECon

We don't have:

- Tools for reconstructing Rust types.
- Good systematic explanations of static Rust reversing.

The State of Rust Reversing in 2025

📖 The `rustc` Book > Codegen Options > `strip`

> Note that, at any level, removing debuginfo only necessarily impacts "friendly" introspection. `-Cstrip` cannot be relied on as a meaningful security or obfuscation measure, as disassemblers and decompilers can extract ***considerable information*** even in the absence of symbols.

(emphasis added)

haha…yes…..we can totally extract considerable information…….

What we'll cover today

- The basic building blocks of the Rust type system: The programmer's perspective
- The basic building blocks of the Rust type system: The compiler's perspective
- Constructing standard library types from the building blocks
- Features of Rust binaries that give information about type layout

Our example: The *RustyClaw* malware

The *RustyClaw* malware, first publicly reported by Cisco Talos in October 2024, is a downloader used to deliver a backdoor: 🔗 UAT-5647 targets Ukrainian and Polish entities with RomCom malware variants

We will be looking at the sample with SHA-256 hash

```
1  b1fe8fbbb0b6de0f1dcd4146d674a71c511488a9eb4538689294bd782df040df
```

🔗 Sample download from MalwareBazaar

This is an x86_32 Windows binary.

Call to `core::result::unwrap_failed`

Specifically, we will be trying to annotate the code inside *this one block* as much as possible!

- This block is located inside a function where the malware checks for the Windows version.

```
int32_t __fastcall sub_404283(char* arg1, int32_t arg2)

0040443a            }
0040435e        } else {
00404360            int32_t eax_1 = var_50.d
00404363            lpProcName = var_50:4.d
00404366            esi_1 = var_48
00404369            result = result_1
00404369
00404370            if (neg.d(eax_1) != 0x80000000) {
00404376                var_44 = eax_1
00404379                PSTR lpProcName_2 = lpProcName
0040437c                uint32_t var_3c_1 = esi_1
0040437f                var_38.d = edx_4
00404382                int32_t var_14_2 = 3
0040439f                sub_426ce9(&var_44, 0x2b, "called `Result::unwrap()` on an `Err` value",
0040439f                    &var_44, &data_428314, &data_428410)
0040439f                noreturn
00404370            }
0040435e        }
00404479        int32_t eax_8 = GetProcAddress(hModule, lpProcName)
```

Call to `core::result::unwrap_failed`

Here's a preview of the nicely annotated result.

```
BOOL __fastcall mw::is_windows7_or_below(char* data_for_hashing, int32_t length_of_data_for_hashing)

0040435e        } else {
00404360            struct std::result::Result<std::ffi::CString, std::ffi::NulError> unwrap_result =
00404360                cstring_new_result
00404363            var_4c
00404363            lpProcName = var_4c.d
00404366            esi_1 = var_48
00404369            _failure_position_in_data = result_err.Err._failure_position_in_data
00404369
00404370            if (neg.d(unwrap_result) != 0x80000000) {
00404376                result_err._discriminant = unwrap_result
00404379                result_err.Err._failure_data.buf.ptr = lpProcName
0040437c                result_err.Err._failure_data.buf.cap = esi_1
0040437f                result_err.Err._failure_data.len = edx_3
00404382                int32_t var_14_2 = 3
0040439f                core::result::unwrap_failed(error_concrete_type_data: &result_err,
0040439f                    error_vtable: &_<impl fmt::Debug for std::ffi::NulError>::_vtable,
0040439f                    panic_location: &panic_location_"src\is_windows7_or_below.rs"_line_37_col_59,
0040439f                    msg_data_ptr: "called `Result::unwrap()` on an `Err` value", msg_len: 0x2b)
0040439f                noreturn _llvm_panic() __tailcall
00404370            }
0040435e        }
00404479        BOOL (__stdcall* GetVersionExW)(OSVERSIONINFOW* lpVersionInformation) =
00404479            GetProcAddress(hModule, lpProcName)
00404479
```

Inside `core::result::unwrap_failed`

We'll also be reversing a function called inside this block.

- This actually ends up being the Rust standard library function `core::result::unwrap_failed`.

```
void __convention("regparm") sub_426ce9(int32_t arg1, int32_t arg2, int32_t arg3, int32_t arg4, int32_t arg5,

00426ce9    void __convention("regparm") sub_426ce9(int32_t arg1, int32_t arg2, int32_t arg3, int32_t arg4,
00426ce9      int32_t arg5, int32_t arg6) __noreturn

00426cff        int32_t var_44 = arg3
00426d01        int32_t var_40 = arg2
00426d0c        void* const var_24 = &data_42c228
00426d12        int32_t var_3c = arg4
00426d14        int32_t var_38 = arg5
00426d1b        int32_t* var_34 = &var_44
00426d1d        int32_t (* var_30)(int32_t* arg1, char* arg2) = sub_409b8e
00426d24        int32_t* var_2c = &var_3c
00426d27        int32_t (* var_28)(int32_t* arg1, int32_t arg2) = sub_409ba2
00426d31        int32_t var_20 = 2
00426d34        int32_t var_14 = 0
00426d38        int32_t** var_1c = &var_34
00426d3b        int32_t var_18 = 2
00426d3e        sub_426b1b(&var_24)
00426d3e        noreturn
```

Inside `core::result::unwrap_failed`

```
void core::result::unwrap_failed(void* error_concrete_type_data, struct fmt::Debug::_vtable* error_vtable, stru

00426ce9    // fn unwrap_failed(msg: &str, error: &dyn fmt::Debug) -> ! {
00426ce9    //     panic!("{msg}: {error:?}")
00426ce9    // }
00426ce9
00426ce9    void core::result::unwrap_failed(void* error_concrete_type_data,
00426ce9      struct fmt::Debug::_vtable* error_vtable, struct core::panic::Location* panic_location,
00426ce9      void* msg_data_ptr @ ecx, void* msg_len @ edx)

00426ceb        struct core::fmt::Arguments formatting_arguments
00426ceb        void* fmt
00426ceb        formatting_arguments.fmt..Some.__offset(0x4).d = fmt
00426cff        struct &str msg_data_
00426cff        msg_data_.data_ptr = msg_data_ptr
00426d01        msg_data_.length = msg_len
00426d0c        formatting_arguments.pieces._slice_data = &data_42c228
00426d12        struct &dyn fmt::Debug error_trait_object_
00426d12        error_trait_object_._concrete_type_data = error_concrete_type_data
00426d14        error_trait_object_._vtable = error_vtable
00426d1b        struct Argument formatting_arguments_data[0x2]
00426d1b        formatting_arguments_data[0].value = &msg_data_
00426d1d        formatting_arguments_data[0].formatter = _<&str as core::fmt::Display>::fmt
00426d24        formatting_arguments_data[1].value = &error_trait_object_
00426d27        formatting_arguments_data[1].formatter = _<&T as core::fmt::Debug>::fmt
00426d31        formatting_arguments.pieces._slice_len = 2
00426d34        formatting_arguments.fmt._discriminant = 0
00426d38        formatting_arguments.args._slice_data = &formatting_arguments_data
00426d3b        formatting_arguments.args._slice_len = 2
00426d3e        core::panicking::panic_fmt(&formatting_arguments, panic_location)
00426d3e        noreturn
```

The basic building blocks of the Rust type system: The programmer's perspective

Understanding Rust types, from the *source code* side.

Learning a little bit of Rust

We can't learn all of Rust today, but we will need to understand *some* Rust source code.

- Rust is a very different language from C.
  - → We need to take the source emitted by our decompiler and go one abstraction level up.
- The Rust standard library is written in Rust, and so is the Rust compiler.
  - → Sometimes, we will need to look at those when trying to figure out how something works.
  - → Reducing "magic" as much as possible.
  - → We should be able to find where in the Rust toolchain something comes from, so that we can be prepared if / when something changes.

Reading Rust

We will need to read some Rust source code today, so a crash course for C programmers on some syntax:

- A variable declaration, with a type annotation:

```
1 let counter: u64 = 0;
```

- Functions look like this:

```
1 fn transform_value(input: u64) -> u64 {
2     // Function body here
3 }
4
5 let value: u64 = transform_value(10);
```

Reading Rust

- Generics use angle brackets:

```
1 let values: Vec<u8> = Vec::new();
```

- Reference types have **&** :

```
1 fn sum_values(input: &Vec<u8>) -> i64 {
2     // Function body here
3 }
```

- To take a reference, also use **&**:

```
1 let values: Vec<u8> = vec![0, 1, 2];
2
3 let sum: i64 = sum_values(&values);
```

Basic types

- `i64` -> Signed 64-bit integer
- `u8` -> Unsigned 8-bit integer
- `f32` -> 32-bit floating point value
- `usize`, `isize` -> The size of a pointer, on whatever platform you're on (think `size_t`, `ssize_t`)
- `bool` -> True, or False. Always a size of 1 byte.
- `()` -> The empty "unit" type.
- `[u64; 128]` -> An array of unsigned 64-bit integers, with 128 entries.

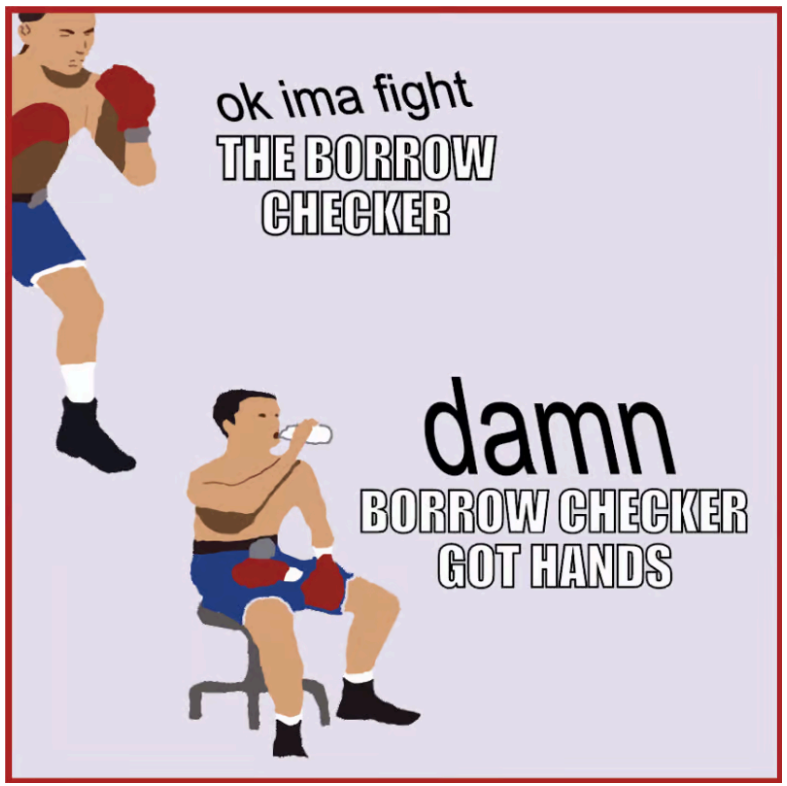
Reading Rust

Code examples in this talk are simplified for easier reading and clarity:

- No lifetime annotations
- No `pub` keyword
- No `mut` keyword
- Some namespaces are expanded for clarity

Things we won't talk about

- We will (mostly) not talk about:
  - → The borrow checker
  - → Lifetimes
  - → Mutability
  - → `unsafe`
- If you want to learn Rust as a programmer, these are important.
- However, they (mostly) don't affect type layout.



Slices

Syntax: `&[T]`, where `T` is some type.

- A special type of reference - a sized view into some collection of data.
- An `&[u8]` is a sized view into a collection of unsigned 8-bit integers (`u8`).
- This reference contains information about not only the *address* of the first element, but also about the *length* of how many elements we want to slice.
- This is essentially a pointer, but with additional length metadata.

Slices

Suppose we have an array of bytes:

```
1 let array: [u8; 5] = [0, 1, 2, 3, 4];
```

We can take a slice into that array:

```
1 let array_slice = &array[1..3];
2 println!("Slice contents from index 1 (inclusive) to 3 (exclusive): {:?}", array_slice
```

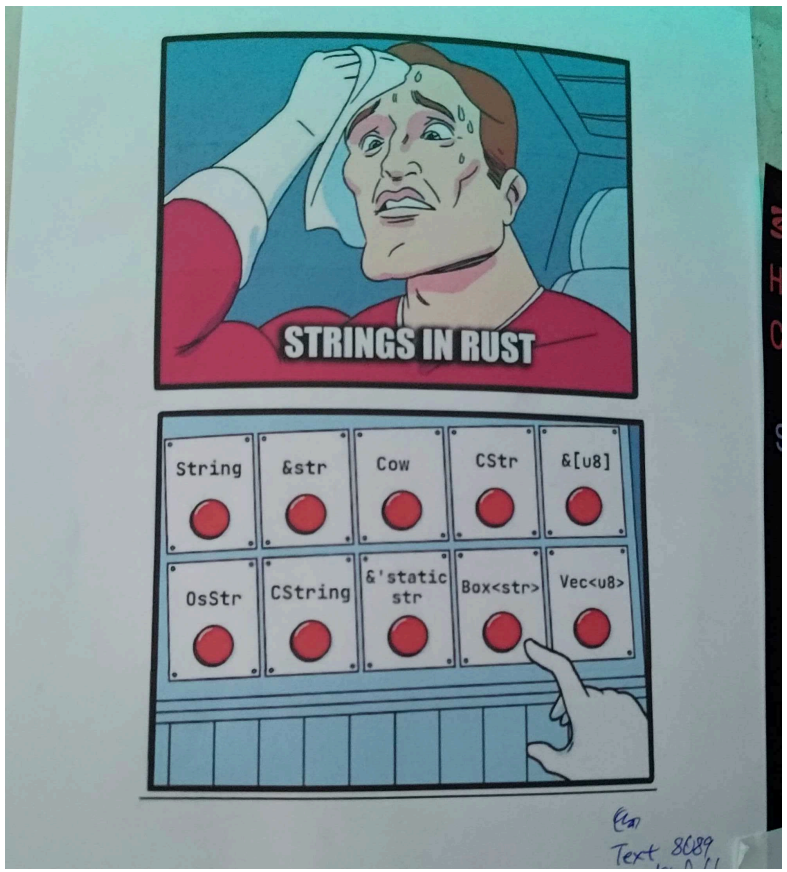
```
1 Slice contents from index 1 (inclusive) to 3 (exclusive): [1, 2]
```

And query its length:

```
1 println!("Slice length: {}", array_slice.len())
```

```
1 Slice length: 2
```

Strings



(thank you Johannes for this beautiful analog meme from 38c3)

We will be focusing on just two string types:

- The primitive type, `&str`
  - → Also called a *string slice*
  - → A reference / sized view to some static string data somewhere
- The standard library type, `std::string::String`
  - → A growable string, (usually) on the heap

The primitive `&str` type

You will see string slices (`&str`) for:

- String literals (similar to `const char*`):

```
1 let PROGRAM_NAME: &str = "tunnel_tool";
```

- Slices of the standard library type, `std::string::String`:

```
1 let full_name: String = String::from("Cindy Xiao");
2 let first_name: &str = &full_name[0..5];
3 println!("Slice of length {}, with data {}", first_name.len(), first_name);
```

```
1 Slice of length 5, with data Cindy
```

An example of **`&str`** *: Panic path metadata in binaries*

```
0x4283dc .rdata {0x428000-0x434da0} Read-only data


004283dc  char data_4283dc[0x1b] = "src\\is_windows7_or_below.rs"

004283f7                          6b-65 72 6e 65 6c 33 32 2e         kernel32.
00428400  64 6c 6c 47 65 74 56 65-72 73 69 6f 6e 45 78 57  dllGetVersionExW

00428410  struct &str data_428410 =
00428410  {
00428410      u8* _slice_data = data_4283dc {"src\is_windows7_or_below.rskernel32.dllGetVersionExW"}
00428414      usize _slice_len = 0x1b
00428418  }

00428418                              25 00 00 00 3b 00 00 00          %...;...
```

See the following for more info:

- 2023 - Rust String Slicer - Binary Ninja Plugin
- 2023 - Using panic metadata to recover source code information from Rust binaries - Blog Post

Structs, and the `std::string::String` type

- 📖 Docs: `std::string::String`

```
struct String {
    vec: Vec<u8>,
}
```

```
struct Vec<u8> {
    buf: RawVec<u8>,
    len: usize,
}
```

```
struct RawVec<u8> {
    ptr: *u8, // Some details simplified here
    cap: usize,
}
```

The `std::string::String` type

Putting it all together into a C-like representation:

```
1 struct String {
2     struct Vec<u8> {
3         struct RawVec<u8> {
4             uint8_t* ptr;
5             size_t cap;
6         } buf;
7         size_t len;
8     } vec;
9 };
```

Enums, i.e. tagged unions

```
enum std::result::Result<i64, String> {
    Ok(i64),
    Err(String),
}
```

```
struct Result<i64, String> {
    enum {
        Ok = 0,
        Err = 1,
    } discriminant;
    union {
        int64_t Ok_data;
        char* Err_data;
    } data;
}
```

Traits (i.e. “object oriented programming” in Rust)

- 📖 Docs: `core::fmt::Write`

  → “A trait for writing or formatting into Unicode-accepting buffers or streams.”

```
1 trait core::fmt::Write {
2     // Required method
3     fn write_str(&mut self, s: &str) -> Result<(), core::fmt::Error>;
4
5     // Provided methods
6     fn write_char(&mut self, c: char) -> Result<(), core::fmt::Error> { ... }
7     fn write_fmt(&mut self, args: Arguments) -> Result<(), core::fmt::Error> { ... }
8 }
```

Traits (i.e. “object oriented programming” in Rust)

```
trait core::fmt::Write {
    // Required method
    fn write_str(&mut self, s: &str) -> Result<(), core::fmt::Error>;

    // Provided methods
    fn write_char(&mut self, c: char) -> Result<(), core::fmt::Error> { ... }
    fn write_fmt(&mut self, args: Arguments) -> Result<(), core::fmt::Error> { ... }
}
```

```
impl core::fmt::Write for std::string::String {
    fn write_str(&mut self, s: &str) -> Result {
        self.push_str(s);
        Ok(())
    }

    fn write_char(&mut self, c: char) -> Result {
        self.push(c);
        Ok(())
    }
}
```

Dynamic dispatch using traits

To do dynamic dispatch without having a concrete type, Rust uses another type of reference: the *trait object*, e.g. `&dyn core::fmt::Write`.

```
fn append_woohoo(writeable_type: &mut dyn core::fmt::Write) {
    writeable_type.write_str(", woohoo").unwrap();
}

fn main() {
    let mut message = String::from("I'm at RE//verse 2025");
    println!("{}", message);

    append_woohoo(&mut message);
    println!("{}", message);
}
```

```
I'm at RE//verse 2025
I'm at RE//verse 2025, woohoo
```

Implementing destructors: The `Drop` Trait

One important trait that will be relevant for us later: The `Drop` trait.

```
1 trait Drop {
2     // Required method
3     fn drop(&mut self);
4 }
```

This is a destructor - it gets called for non-primitive types when values go out of scope!

- 📖 The Rust Reference > Destructors

Implementing the destructor for `Vec<T>`: `impl Drop for Vec<T>`

```
1  struct Vec<T> {
2      buf: RawVec<T>,
3      len: usize,
4  }
5
6  impl<T> Drop for Vec<T> {
7      fn drop(&mut self) {
8          unsafe {
9              ptr::drop_in_place(ptr::slice_from_raw_parts_mut(self.as_mut_ptr(), self.l
10         }
11     }
12 }
```

The basic building blocks of the Rust type system: The compiler's perspective

What Rust guarantees: Type Layouts

📖 The Rust Reference > Type Layouts

- The sizes of primitive scalar types (`bool`, `i64`, `f32`, etc.) are guaranteed.
- The sizes of references are guaranteed.

  → However, there are different types of references, each with a different (guaranteed) size!

  ⇾ References to primitive types with known sizes, such as `&i64`

  ⇾ Slices, such as `&str`, `&[u8]`, etc.

  ⇾ Trait objects, such as `&dyn core::fmt::Write`

What Rust guarantees: Type Layouts

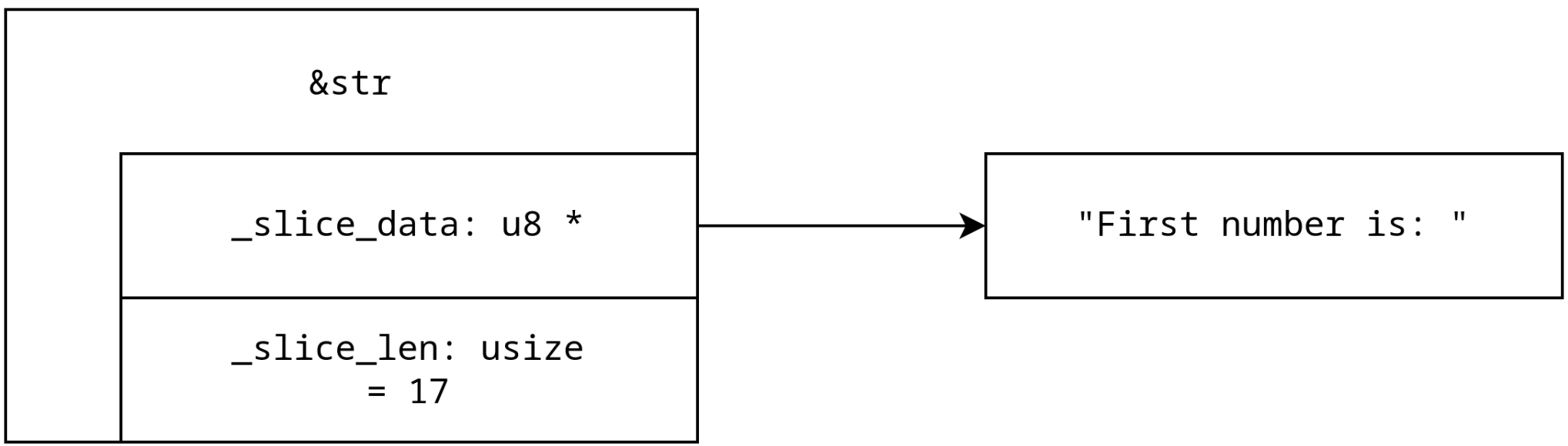
📖 The Rust Reference > Type Layouts

- ⚠ Structs and enums have no layout guarantees!
  - → No guaranteed size
  - → No guaranteed alignment
  - → No guaranteed field ordering

References: Slices

These include:

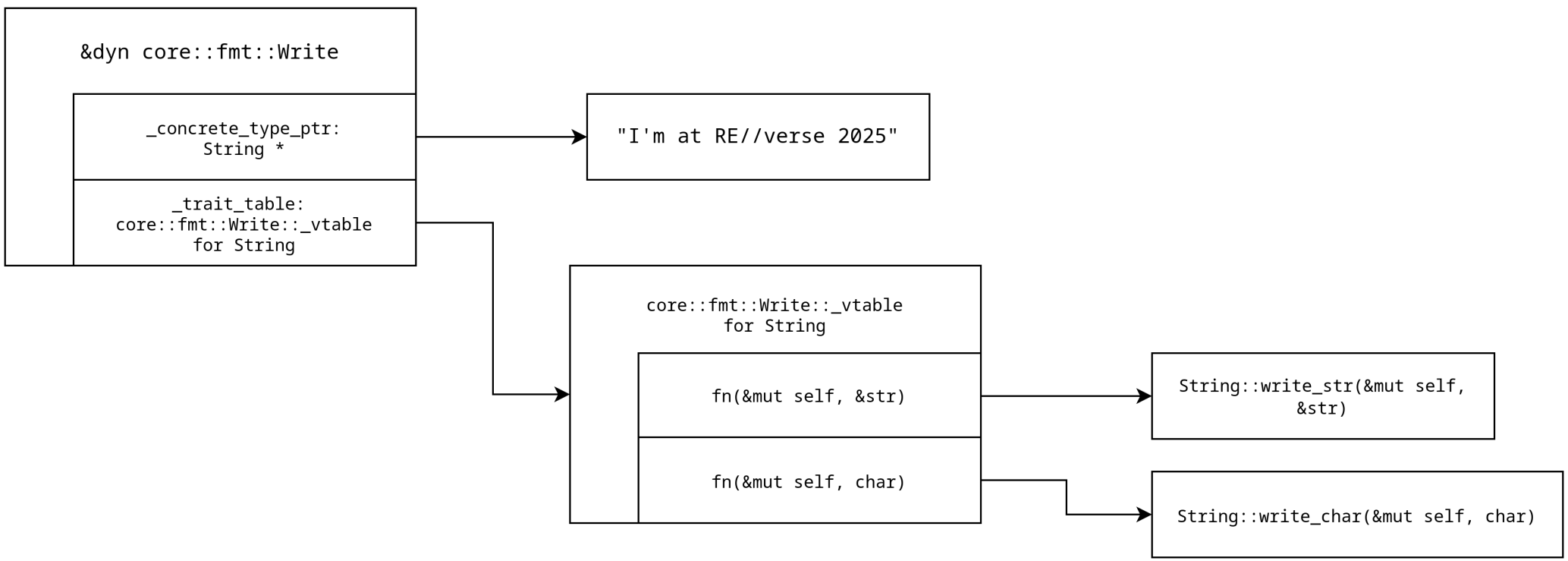
- A pointer: To the beginning of the slice
- Metadata attached to the pointer: The length of the slice



References: Trait objects

These include:

- A pointer: To the concrete type
- Metadata attached to the pointer: The trait table



What Rust guarantees: Passing types between functions

⚠ Rust's calling convention for Rust-to-Rust function calls is neither *stable*, nor even *defined*.

- Often the compiler will pick one of the platform's usual calling conventions, but this is not guaranteed.

Example from our RustyClaw binary (Windows x86_32) of (sort of) `__fastcall`:

```
BOOL __fastcall mw::is_windows7_or_below(char* data_for_hashing, int32_t length_of_data_for_hashing)
00404370  0f80ff000000        jo      0x404475

00404376  8945c0              mov     dword [ebp-0x40 {result_err._discriminant}], eax
00404379  897dc4              mov     dword [ebp-0x3c {result_err.Err._failure_data.buf.cap}], edi
0040437c  8975c8              mov     dword [ebp-0x38 {result_err.Err._failure_data.buf.ptr}], esi
0040437f  8955cc              mov     dword [ebp-0x34 {result_err.Err._failure_data.len}], edx
00404382  c745f003000000      mov     dword [ebp-0x10 {var_14_2}], 0x3
00404389  6a2b                push    0x2b
0040438b  5a                  pop     edx  {0x2b}
0040438c  b9c4ac4200          mov     ecx, data_42acc4  {"called `Result::unwrap()` on an `Err` value"}
00404391  6810844200          push    panic_location_"src\is_windows7_or_below.rs"_line_37_col_59
00404396  6814834200          push    _<impl fmt::Debug for std::ffi::NulError>::_vtable {var_18c}
0040439b  8d45c0              lea     eax, [ebp-0x40 {result_err}]
0040439e  50                  push    eax {result_err} {var_190_1}
0040439f  e845290200          call    core::result::unwrap_failed  {_llvm_panic}
{ Falls through into _llvm_panic }
{ Does not return }
```

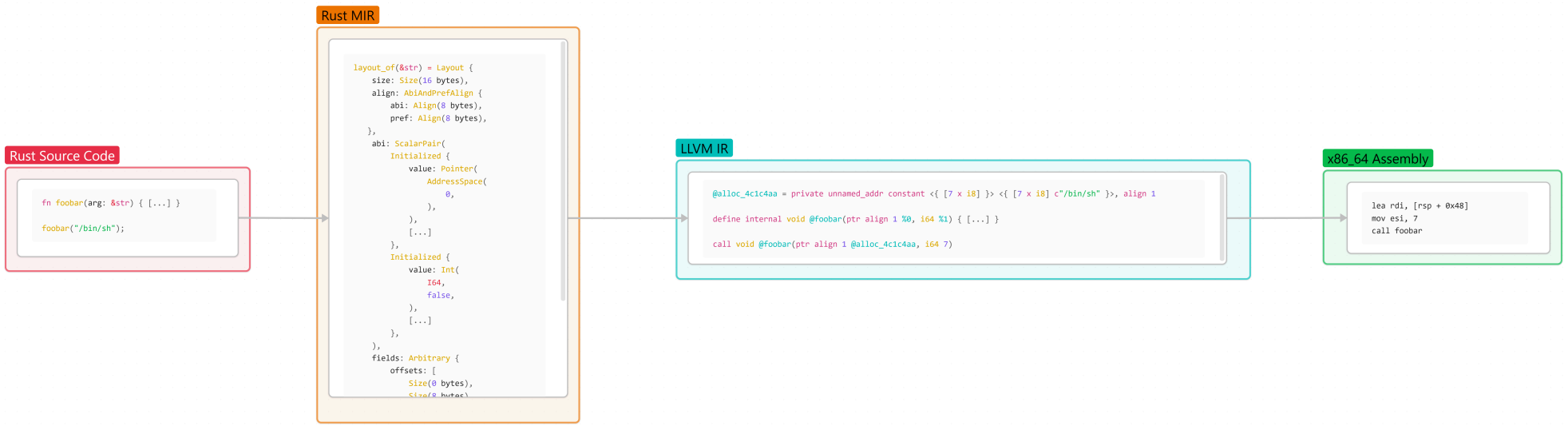
What Rust guarantees: Passing types between functions

Notice how a single **`&str`** is split across two registers here:

```
1  struct `&str`
2  {
3      uint8_t* _slice_data = "called Result::unwrap() on an Err value"
4      size_t _slice_len = 0x2b
5  };
```

```
BOOL __fastcall mw::is_windows7_or_below(char* data_for_hashing, int32_t length_of_data_for_hashing)

00404370  0f80ff000000        jo      0x404475

00404376  8945c0              mov     dword [ebp-0x40 {result_err._discriminant}], eax
00404379  897dc4              mov     dword [ebp-0x3c {result_err.Err._failure_data.buf.cap}], edi
0040437c  8975c8              mov     dword [ebp-0x38 {result_err.Err._failure_data.buf.ptr}], esi
0040437f  8955cc              mov     dword [ebp-0x34 {result_err.Err._failure_data.len}], edx
00404382  c745f003000000      mov     dword [ebp-0x10 {var_14_2}], 0x3
00404389  6a2b                push    0x2b
0040438b  5a                  pop     edx  {0x2b}
0040438c  b9c4ac4200          mov     ecx, data_42acc4  {"called `Result::unwrap()` on an `Err` value"}
00404391  6810844200          push    panic_location_"src\is_windows7_or_below.rs"_line_37_col_59
00404396  6814834200          push    _<impl fmt::Debug for std::ffi::NulError>::_vtable {var_18c}
0040439b  8d45c0              lea     eax, [ebp-0x40 {result_err}]
0040439e  50                  push    eax {result_err} {var_190_1}
0040439f  e845290200          call    core::result::unwrap_failed  {_llvm_panic}
{ Falls through into _llvm_panic }
{ Does not return }
```

References: Pointers and Metadata, through the compiler pipeline



Enums, and their discriminants

A C-like representation of our standard library `Result` type:

```
struct std::io::error::Result<Vec<u8>> {
    enum {
        Ok = 0,
        Err = 0x8000000000000000,
    } __discriminant;
    union {
        Vec<u8> Ok_data;
        std::io::error::Error* Err_data;
    } data;
}
```

Enums, and their discriminants

You will often see this idiom in decompiled Rust binaries:

```
struct std::io::error::Result<Vec<u8>> read_result;
std::fs::read::inner(&read_result, path_data, path_len);
uint64_t __discriminant = read_result.__discriminant;

if (__discriminant == 0x8000000000000000) {
    core::result::unwrap_failed("called `Result::unwrap()` on an `Err` value", 0x2b, &
    /* no return */
} else {
    /* Read was successful, do stuff with read_result here... */
}
```

- 📖 Docs: **`std::mem::discriminant`**
- 📖 The Rust Reference > Discriminants

Example: A **`Result`** *from RustyClaw*

```
int32_t __fastcall sub_404283(char* arg1, int32_t arg2)

0040435e        } else {
00404360            struct std::result::Result<T, E> result_ = var_50.d
00404363            lpProcName = var_50:4.d
00404366            esi_1 = var_48
00404369            result = result_2
00404369
00404370            if (neg.d(result_) != 0x80000000) {
00404376                result_1 = result_
00404379                PSTR lpProcName_2 = lpProcName
0040437c                uint32_t var_3c_1 = esi_1
0040437f                var_38.d = edx_4
00404382                int32_t var_14_2 = 3
0040439f                core::result::unwrap_failed(
0040439f                    "called `Result::unwrap()` on an `Err` value", 0x2b,
0040439f                    &result_1, &data_428314, &data_428410, var_184)
0040439f                noreturn
00404370            }
0040435e        }
00404479        int32_t eax_6 = GetProcAddress(hModule, lpProcName)
```

Putting basic building blocks together

Constructing the `core::fmt::Arguments` standard library type.

Constructing `core::fmt::Arguments`

Let's construct the standard library's `core::fmt::Arguments` type.

- This is useful because it puts together every basic type concept we've looked at so far - structs, references, slices, enums, etc.
- This is useful, because understanding it is required for understanding string formatting in Rust, which comes up quite often!

String formatting: Printing text with ***`println!`***

From the programmer's perspective, doing string formatting is quite easy:

```
println!("First number is: {}, second number is: {}", 86, 64);
```

```
First number is: 86, second number is: 64
```

String formatting: Printing panic strings when aborting / panicking the program

- 📖 Example from The Rust Book > Error Handling

```
1 use std::fs::File;
2
3 fn main() {
4     let greeting_file = File::open("hello.txt").unwrap();
5 }
```

```
1 thread 'main' panicked at src/main.rs:4:49:
2 called `Result::unwrap()` on an `Err` value: Os { code: 2, kind: NotFound, message: "N
```

String formatting: Printing panic strings when aborting / panicking the program

-  Source: `library/core/src/result.rs`

```
fn core::result::unwrap_failed(msg: &str, error: &dyn fmt::Debug) -> ! {
    panic!("{msg}: {error:?}")
}
```

String formatting: Peeking inside ***`println!`***

- 📖 Docs: `println!`, `std::io::stdio::_print`
- 💿 Source: `std/src/io/stdio.rs`

```
macro_rules! println {
    ($($arg:tt)*) => {{
        $crate::io::_print($crate::format_args_nl!($($arg)*));
    }};
}
```

```
fn std::io::stdio::_print(args: core::fmt::Arguments)
```

The `core::fmt::Arguments` Type

- 📖 Docs: core::fmt::Arguments
- 💿 Source: core/src/fmt/mod.rs

```
1 struct core::fmt::Arguments {
2     pieces: &[&str],                                    ①
3     fmt: Option<&[core::fmt::rt::Placeholder]>,         ②
4     args: &[core::fmt::rt::Argument],                   ③
5 }
```

① An array slice (`&[]`), containing the string literals (`&str`) to put together: `["First number is: ", ", second number is: ", "\n"]`

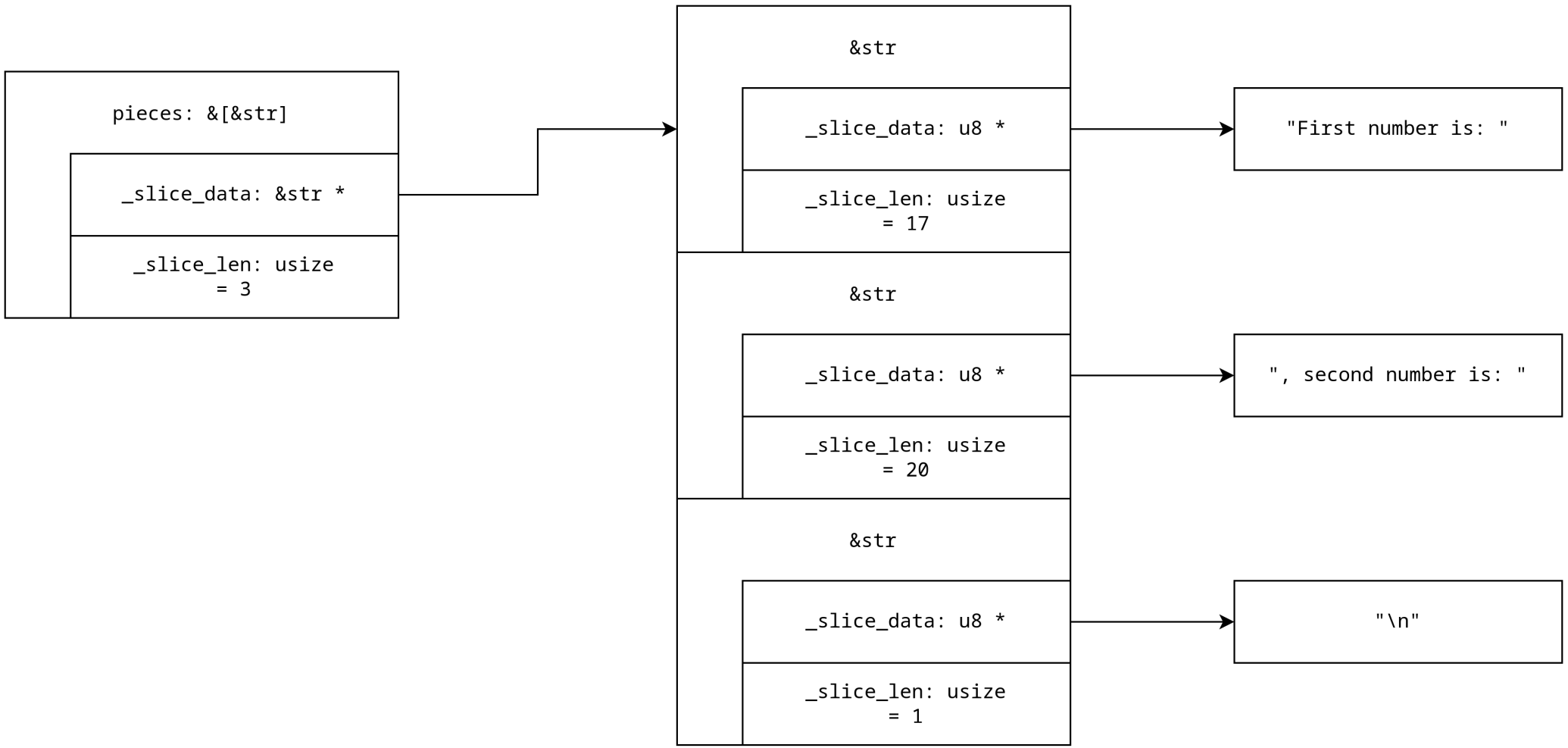
② Any formatting specifications (alignment, width, etc.)

③ An array slice (`&[]`), containing the dynamic values (`core::fmt::rt::Argument`) to display as strings: `[86, 64]`

Exploding this: `&[&str]`

- An `&[&str]` is:
  - → An array slice (`&[]`)
  - → Pointing to a collection of string slices (`&str`)

Exploding this: `&[&str]`



Exploding this: `Option<&[core::fmt::rt::Placeholder]>`

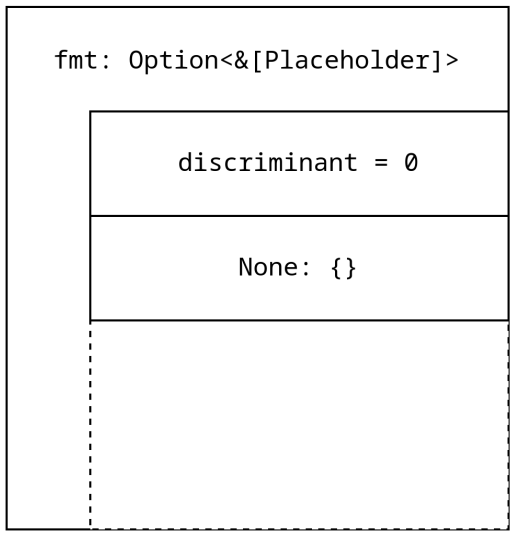
- An `Option<&[core::fmt::rt::Placeholder]>` is:
  - → A Rust enum, i.e.
  - → tagged union (`Option<>`)
  - → Which either contains nothing, or contains a `&[core::fmt::rt::Placeholder]`
- A `&[core::fmt::rt::Placeholder]` is:
  - → An array slice (`&[]`)
  - → Pointing to a collection of `core::fmt::rt::Placeholder` structs

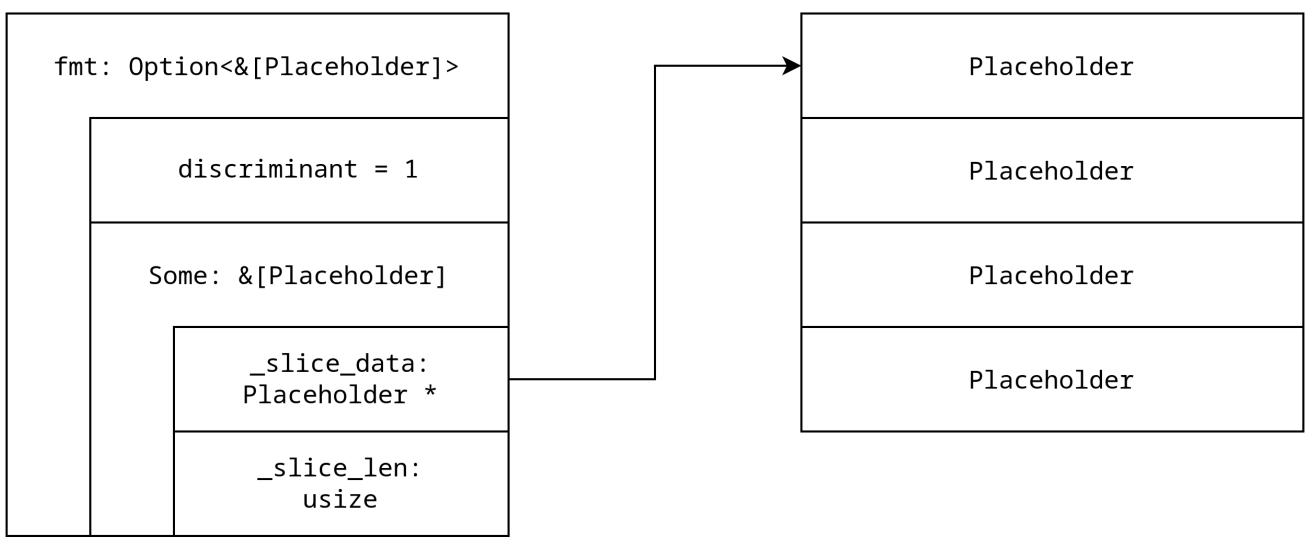
```
1  enum Option<&[core::fmt::rt::Placeholder]> {
2      None,                                          ①
3      Some(&[core::fmt::rt::Placeholder]),           ②
4  }
```

① The `None` variant of the union, when it holds no data.

② The `Some` variant of the union, when it holds an actual array slice (`&[]`) of `core::fmt::rt::Placeholder` values.

Exploding this: `Option<&[core::fmt::rt::Placeholder]>`





Exploding this: `&[core::fmt::rt::Argument]`

- A `&[core::fmt::rt::Argument]` is:
  - → An array slice (`&[]`)
  - → Containing a set of `core::fmt::rt::Argument` structs

Exploding this: `core::fmt::rt::Argument`

```
1 struct core::fmt::rt::Argument {
2     value: &Opaque,                                          ①
3     formatter: fn(_: &Opaque, _: &mut Formatter) -> Result,  ②
4 }
```

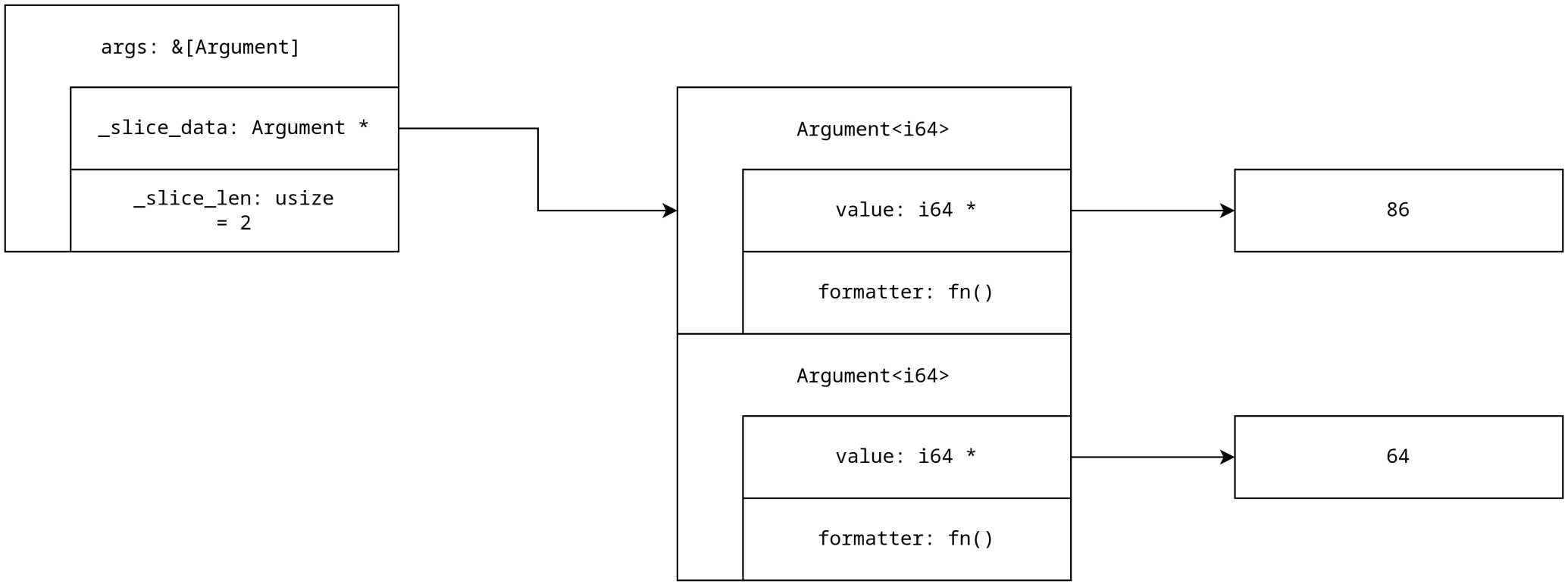
① A reference (`&`) to a value (`Opaque`) to format into a string.

② A pointer to a function (`fn()`) which does the actual string formatting.

A C-like representation of `core::fmt::rt::Argument`

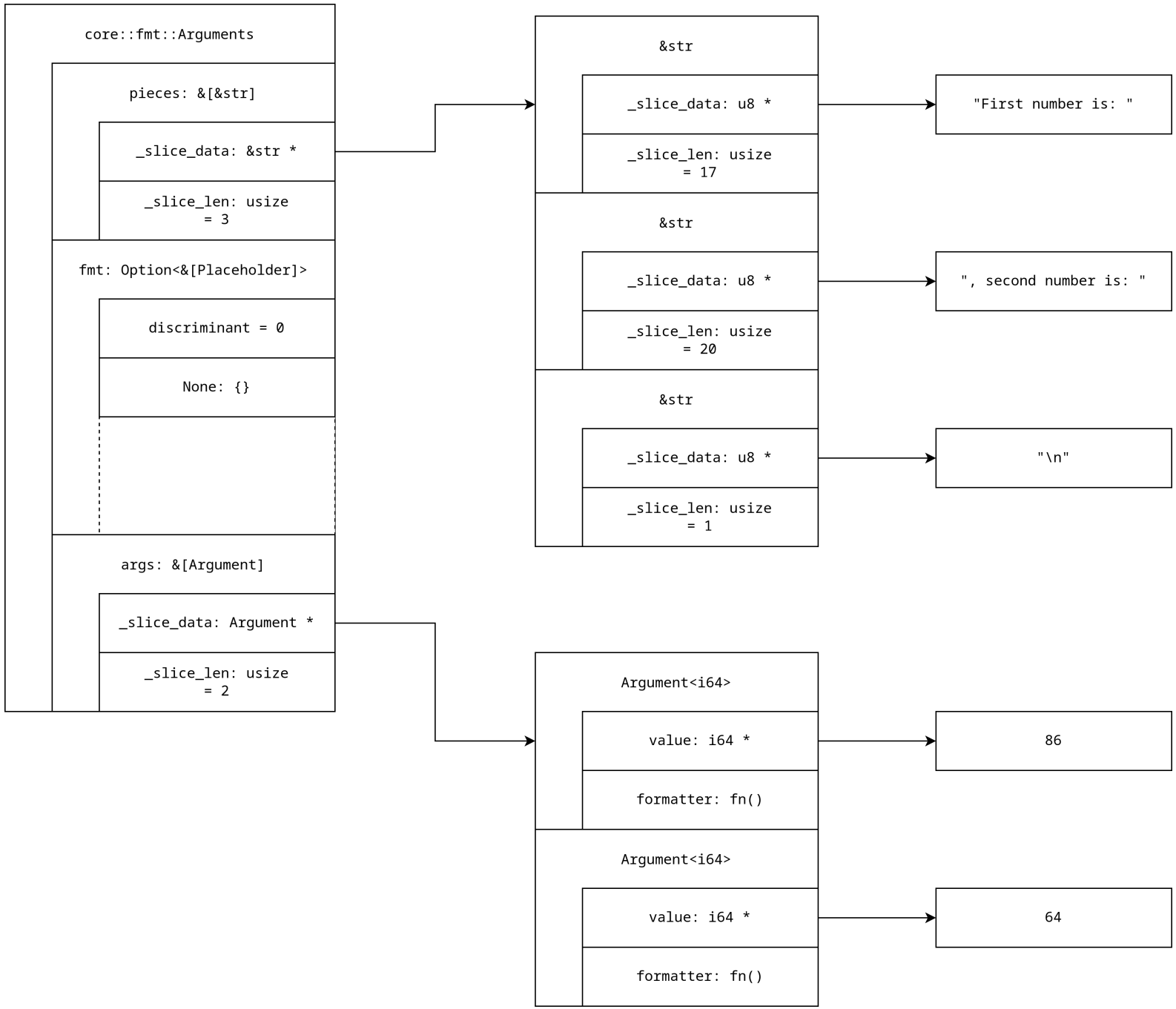
```
1  struct Argument<i64>
2  {
3      void* value;
4      Result* (* formatter)(void* value_to_format, Formatter* formatter);
5  };
```

Exploding this: `&[core::fmt::rt::Argument]`



The full explosion

```
struct core::fmt::Arguments
{
    struct &[&str] pieces
    {
        struct &str* data_ptr;
        usize length;
    },
    struct Option<&[core::fmt::rt::Placeholder]> fmt
    {
        __discriminant_type discriminant;
        union
        {
            &[core::fmt::rt::Placeholder] Some;
            struct {} None;
        }
    }
    struct &[core::fmt::rt::Argument] args
    {
        struct core::fmt::rt::Argument* data_ptr;
        usize length;
    },
};
```



Back to the RustyClaw `core::result::unwrap_failed` *example*

```
int32_t __fastcall sub_404283(char* arg1, int32_t arg2)

0040435e        } else {
00404360            struct std::result::Result<T, E> result_ = var_50.d
00404363            lpProcName = var_50:4.d
00404366            esi_1 = var_48
00404369            result = result_2
00404369
00404370            if (neg.d(result_) != 0x80000000) {
00404376                result_1 = result_
00404379                PSTR lpProcName_2 = lpProcName
0040437c                uint32_t var_3c_1 = esi_1
0040437f                var_38.d = edx_4
00404382                int32_t var_14_2 = 3
0040439f                core::result::unwrap_failed(
0040439f                    "called `Result::unwrap()` on an `Err` value", 0x2b,
0040439f                    &result_1, &data_428314, &data_428410, var_184)
0040439f                noreturn
00404370            }
0040435e        }
00404479        int32_t eax_6 = GetProcAddress(hModule, lpProcName)
```

Back to the RustyClaw ***core::result::unwrap_failed*** *example*

```
void __convention("regparm") core::result::unwrap_failed(char* msg_data_ptr, uint32_t msg_len, int32_t

00426ce9    void __convention("regparm") core::result::unwrap_failed(char* msg_data_ptr,
00426ce9      uint32_t msg_len, int32_t arg3, int32_t arg4, int32_t arg5, int32_t arg6) __noreturn

00426cff      int32_t var_44 = arg3
00426d01      uint32_t msg_len_1 = msg_len
00426d0c      void* const var_24 = &data_42c228
00426d12      int32_t var_3c = arg4
00426d14      int32_t var_38 = arg5
00426d1b      int32_t* var_34 = &var_44
00426d1d      int32_t (* var_30)(int32_t* arg1, char* arg2) = sub_409b8e
00426d24      int32_t* var_2c = &var_3c
00426d27      int32_t (* var_28)(int32_t* arg1, int32_t arg2) = sub_409ba2
00426d31      int32_t var_20 = 2
00426d34      int32_t var_14 = 0
00426d38      int32_t** var_1c = &var_34
00426d3b      int32_t var_18 = 2
00426d3e      sub_426b1b(&var_24)
00426d3e      noreturn
```

Spotting some likely ***core::fmt::Argument*** *variables*

```
void __convention("regparm") core::result::unwrap_failed(char* msg_data_ptr, uint32_t msg

00426cfd  89e3              mov     ebx, esp {var_44}
00426cff  890b              mov     dword [ebx {var_44}], ecx
00426d01  894304            mov     dword [ebx+0x4 {msg_len_1}], eax
00426d04  8d442408          lea     eax, [esp+0x8 {var_3c}]
00426d08  8d4c2420          lea     ecx, [esp+0x20 {var_24}]
00426d0c  c70128c24200      mov     dword [ecx {var_24}], data_42c228
00426d12  8938              mov     dword [eax {var_3c}], edi
00426d14  897004            mov     dword [eax+0x4 {var_38}], esi
00426d17  8d742410          lea     esi, [esp+0x10 {var_34}]
00426d1b  891e              mov     dword [esi {var_34}], ebx {var_44}
00426d1d  c746048e9b4000    mov     dword [esi+0x4 {var_30}], sub_409b8e
00426d24  894608            mov     dword [esi+0x8 {var_2c}], eax {var_3c}
00426d27  c7460ca29b4000    mov     dword [esi+0xc {var_28}], sub_409ba2
00426d2e  6a02              push    0x2 {var_48}
00426d30  58                pop     eax {var_48}  {0x2}
00426d31  894104            mov     dword [ecx+0x4 {var_20}], eax  {0x2}
00426d34  83611000          and     dword [ecx+0x10 {var_14}], 0x0
00426d38  897108            mov     dword [ecx+0x8 {var_1c}], esi {var_34}
00426d3b  89410c            mov     dword [ecx+0xc {var_18}], eax  {0x2}
00426d3e  e8d8fdffff        call    sub_426b1b
{ Does not return }
```

Defining a `core::fmt::Argument`

```
1  struct `core::fmt::Argument`
2  {
3      void* value;
4      void* (* formatter)(void* value, void* formatter_struct);
5  };
```

After applying the ***core::fmt::Argument*** *type*

```
void __convention("regparm") core::result::unwrap_failed(char* msg_data_ptr, uint32
89e3                mov     ebx, esp {var_44}
890b                mov     dword [ebx {var_44}], ecx
894304              mov     dword [ebx+0x4 {msg_len_1}], eax
8d442408            lea     eax, [esp+0x8 {var_3c}]
8d4c2420            lea     ecx, [esp+0x20 {var_24}]
c70128c24200        mov     dword [ecx {var_24}], data_42c228
8938                mov     dword [eax {var_3c}], edi
897004              mov     dword [eax+0x4 {var_38}], esi
8d742410            lea     esi, [esp+0x10 {arg_data_0}]
891e                mov     dword [esi {arg_data_0.value}], ebx {var_44}
c746048e9b4000      mov     dword [esi+0x4 {arg_data_0.formatter}], sub_409b8e
894608              mov     dword [esi+0x8 {arg_data_1.value}], eax {var_3c}
c7460ca29b4000      mov     dword [esi+0xc {arg_data_1.formatter}], sub_409ba2
6a02                push    0x2 {var_48}
58                  pop     eax {var_48}  {0x2}
894104              mov     dword [ecx+0x4 {var_20}], eax  {0x2}
83611000            and     dword [ecx+0x10 {var_14}], 0x0
897108              mov     dword [ecx+0x8 {var_1c}], esi {arg_data_0}
89410c              mov     dword [ecx+0xc {var_18}], eax  {0x2}
e8d8fdffff          call    sub_426b1b
ot return }
```

Spotting a likely ***&[core::fmt::Argument]*** *variable*

```
void __convention("regparm") core::result::unwrap_failed(char* msg_data_ptr, uint3
00426d14  897004              mov     dword [eax+0x4 {var_38}], esi
00426d17  8d742410            lea     esi, [esp+0x10 {arg_data_0}]
00426d1b  891e                mov     dword [esi {arg_data_0.value}], ebx {var_44}
00426d1d  c746048e9b4000      mov     dword [esi+0x4 {arg_data_0.formatter}], sub_409b
00426d24  894608              mov     dword [esi+0x8 {arg_data_1.value}], eax {var_3c}
00426d27  c7460ca29b4000      mov     dword [esi+0xc {arg_data_1.formatter}], sub_409b
00426d2e  6a02                push    0x2 {var_48}
00426d30  58                  pop     eax {var_48}  {0x2}
00426d31  894104              mov     dword [ecx+0x4 {var_20}], eax  {0x2}
00426d34  83611000            and     dword [ecx+0x10 {var_14}], 0x0
00426d38  897108              mov     dword [ecx+0x8 {var_1c}], esi {arg_data_0}
00426d3b  89410c              mov     dword [ecx+0xc {var_18}], eax  {0x2}
00426d3e  e8d8fdffff          call    sub_426b1b
{ Does not return }
```

Defining a `&[core::fmt::Argument]`

```
1 struct `&[core::fmt::Argument]` __packed
2 {
3     struct `core::fmt::Argument`* _slice_data;
4     usize _slice_len;
5 };
```

After applying the ***&[core::fmt::Argument]*** *type*

```
void __convention("regparm") core::result::unwrap_failed(char* msg_data_ptr, uint32_
8938            mov     dword [eax {var_3c}], edi
897004          mov     dword [eax+0x4 {var_38}], esi
8d742410        lea     esi, [esp+0x10 {arg_data_0}]
891e            mov     dword [esi {arg_data_0.value}], ebx {var_44}
c746048e9b4000  mov     dword [esi+0x4 {arg_data_0.formatter}], sub_409b8e
894608          mov     dword [esi+0x8 {arg_data_1.value}], eax {var_3c}
c7460ca29b4000  mov     dword [esi+0xc {arg_data_1.formatter}], sub_409ba2
6a02            push    0x2 {var_48}
58              pop     eax {var_48}  {0x2}
894104          mov     dword [ecx+0x4 {var_20}], eax  {0x2}
83611000        and     dword [ecx+0x10 {var_14}], 0x0
897108          mov     dword [ecx+0x8 {arg_slice._slice_data}], esi {arg_data_0}
89410c          mov     dword [ecx+0xc {arg_slice._slice_len}], eax  {0x2}
e8d8fdffff      call    sub_426b1b
t return }
```

In the decompiler after applying `&[core::fmt::Argument]`

```
void __convention("regparm") core::result::unwrap_failed(char* msg_data_ptr, uint32_t msg_

00426ce9    void __convention("regparm") core::result::unwrap_failed(char* msg_data_ptr,
00426ce9      uint32_t msg_len, int32_t arg3, int32_t arg4, int32_t arg5, int32_t arg6)
00426ce9      __noreturn

00426cff      int32_t val_to_be_formatted_0 = arg3
00426d01      uint32_t msg_len_1 = msg_len
00426d0c      void* const var_24 = &data_42c228
00426d12      int32_t val_to_be_formatted_1 = arg4
00426d14      int32_t var_38 = arg5
00426d1b      struct core::fmt::Argument arg_data_0
00426d1b      arg_data_0.value = &val_to_be_formatted_0
00426d1d      arg_data_0.formatter = sub_409b8e
00426d24      struct core::fmt::Argument arg_data_1
00426d24      arg_data_1.value = &val_to_be_formatted_1
00426d27      arg_data_1.formatter = sub_409ba2
00426d31      int32_t var_20 = 2
00426d34      int32_t var_14 = 0
00426d38      struct &[core::fmt::Argument] arg_slice
00426d38      arg_slice._slice_data = &arg_data_0
00426d3b      arg_slice._slice_len = 2
00426d3e      sub_426b1b(&var_24)
00426d3e      noreturn
```

Features of Rust binaries that give information about type layout

Allocation functions

- Heap allocations for standard library types, such as for `std::string::String`, require a global allocator to be defined.
- The Rust standard library provides a default global allocator implementation.
  - → The details of this will vary by platform.

The standard library's global allocator implementation, on Windows

-  Source: library/std/src/sys/alloc/windows.rs

```
1  fn std::sys::pal::windows::alloc::process_heap_alloc(
2      _heap: MaybeUninit<c::HANDLE>,
3      flags: u32,
4      bytes: usize,
5  ) -> *mut c_void
6  {
7      let heap = HEAP.load(Ordering::Relaxed);
8      if core::intrinsics::likely(!heap.is_null()) {
9          unsafe { HeapAlloc(heap, flags, bytes) }
10     } else {
11         process_heap_init_and_alloc(MaybeUninit::uninit(), flags, bytes)   ①
12     }
13 }
```

① Further calls to `HeapAlloc` inside this function

Deallocation functions

- When types that required heap allocations go out of scope, their destructors are called.
  - → Types that require heap deallocation implement the `Drop` trait
  - → That is, they implement a destructor!
- The Rust standard library provides a default global deallocator implementation.
  - → The details of this will vary by platform.

The standard library's global deallocator

```
1 fn __rust_dealloc(ptr: *mut u8, size: usize, align: usize);                    ①
2
3
4 fn __rdl_dealloc(ptr: *mut u8, size: usize, align: usize) {
5     unsafe { System.dealloc(ptr, Layout::from_size_align_unchecked(size, align)) }
6 }
```

① This is just a stub; it gets replaced with `__rdl_dealloc`, if you're using the default standard library allocator

The standard library's global deallocator implementation, on Windows

- Source: library/std/src/sys/alloc/windows.rs

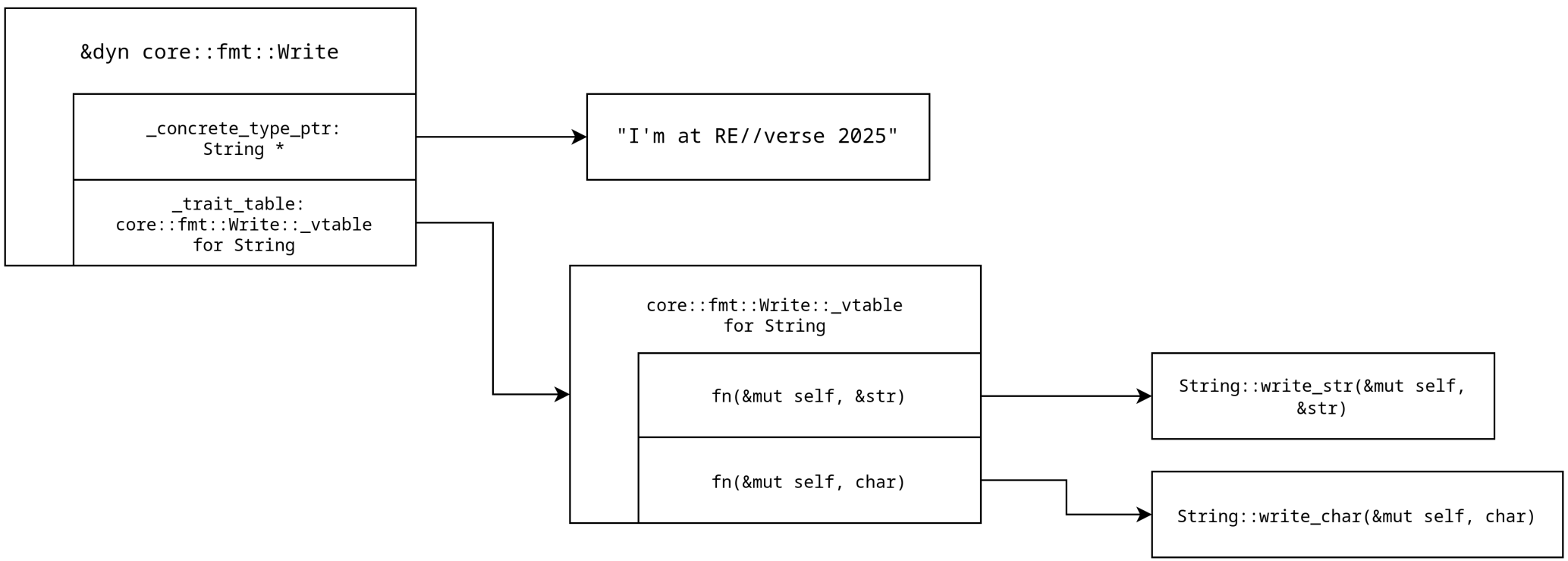
```
unsafe fn System::dealloc(&self, ptr: *mut u8, layout: Layout) {
    let block = {
        if layout.align() <= MIN_ALIGN {
            ptr
        } else {
            // The location of the start of the block is stored in the padding before

            // SAFETY: Because of the contract of `System`, `ptr` is guaranteed to be
            // and have a header readable directly before it.
            unsafe { ptr::read((ptr as *mut Header).sub(1)).0 }
        }
    };

    let heap = unsafe { get_process_heap() };
    unsafe { HeapFree(heap, 0, block.cast::<c_void>()) };
}
```

Trait objects

Trait object tables (i.e. vtables)

Recall the `&dyn core::fmt::Write` trait object. It includes:

- A pointer: To the concrete type that implements the `core::fmt::Write` trait
- Metadata attached to the pointer: A table of pointers to the concrete type's functions, which implement that trait (i.e. a vtable!)



Trait object tables (i.e. vtables)

The vtable attached to trait objects has:

- Size, alignment, and destructor information for the concrete type.
- A fixed layout, such that size, alignment, and destructor information are always in the same places in the table!

  →  Compiler source code where the vtable layout is defined and generated

Trait object tables (i.e. vtables)

```
1 struct core::fmt::Write::_vtable alloc::string::String::_vtable =
2 {
3     void* (* destructor)(void* self) = core::ptr::drop_in_place<alloc::string::String> ①
4     int64_t size = 0x18 ②
5     int64_t alignment = 0x8 ③
6     void* (* write_str)(void* self, char* str_data, uint64_t str_len) = <alloc::string: ④
7     void* (* write_char)(void* self, int32_t character) = <alloc::string::String as cor
8     void* (* write_fmt)(void* self, Arguments* args) = core::fmt::Write::write_fmt
9 }
```

① A pointer to the destructor for this concrete type.

② The size (in bytes) of the concrete type that implements this trait.

③ The alignment (in bytes) of the concrete type that implements this trait.

④ A pointer to this type's implementation of a trait method.

Finding trait object tables

- Find a function pointer, followed by 2 **`usize`** constants, followed by a function pointer
- Ensure that the first function pointer is a destructor; that is, make sure it eventually calls **`__rust_dealloc`** / **`__rdl_dealloc`**.

```
0x1400b2c30 .rdata {0x14009c000-0x1400c2390} Read-only data

1400b2c30  char const data_1400b2c30[0x3c] = "called `Result::unwrap()` on an `Err` valueErrorLayoutError", 0

1400b2c6c                                        00 00 00 00              ....

1400b2c70  void* data_1400b2c70 = core::ptr::drop_in_place<alloc::string::String>::h96044aa01ac4eaf5

1400b2c78                              18 00 00 00 00 00 00 00          ........
1400b2c80  08 00 00 00 00 00 00 00                                      ........

1400b2c88  void* data_1400b2c88 = <alloc::string::String as core::fmt::Write>::write_str::hd148d41295382305
1400b2c90  void* data_1400b2c90 = <alloc::string::String as core::fmt::Write>::write_char::h9ded5901f95d9d46
1400b2c98  void* data_1400b2c98 = core::fmt::Write::write_fmt::h8c1332eb2c2e7a8f
1400b2ca0  char const data_1400b2ca0[0x12] = "capacity overflow", 0
```

Example likely vtable from an x86_64 binary with symbols

Example: Reversing the `core::result::unwrap_failed` function

-  Source: `library/core/src/result.rs`

```
1 fn core::result::unwrap_failed(msg: &str, error: &dyn fmt::Debug) -> ! {
2     panic!("{msg}: {error:?}")
3 }
```

Printing a debug representation of a struct

You can implement the **`fmt::Debug`** trait for your type, to produce a convenient string representation of your type when debugging.

- This can be done via the printing macros (**`println!`**, **`panic!`**, etc.), via the **`{variable_name:?}`** syntax.
- You can also get the compiler to just generate a suitable one for you, by slapping the **`#[derive(Debug)]`** onto your type:

```
#[derive(Debug)]
struct Coordinates {
    x: i64,
    y: i64,
}

fn main() {
    let cursor_position = Coordinates { x: -100, y: 120 };
    println!("{cursor_position:?}");
}
```

```
Coordinates { x: -100, y: 120 }
```

The `&dyn fmt::Debug` trait object

```
1  struct &dyn fmt::Debug
2  {
3      void* _concrete_type_data;
4      struct fmt::Debug::_vtable* _vtable;
5  };
```

Call to `core::result::unwrap_failed` *in RustyClaw*

Notice how our metadata-bearing pointer (`&dyn fmt::Debug`) is split across two variables again!

```
void core::result::unwrap_failed(
    void* error_concrete_type_data,                 // `&dyn fmt::Debug`._concrete_typ
    struct fmt::Debug::_vtable* error_vtable,        // `&dyn fmt::Debug._vtable
    struct core::panic::Location* panic_location,
    void* msg_data_ptr @ ecx,                       // `&str`._slice_data
    void* msg_len @ edx                             // `&str`._slice_len
) { [...] }
```

```
core::result::unwrap_failed(
    &unwrapped_err,                                 // error_concrete_type_data (`&dyn
    &unwrapped_err_vtable,                          // error_vtable (`&dyn fmt::Debug.
    &panic_location_"src\is_windows7_or_below.rs"_line_37_col_59, // panic_location
    "called `Result::unwrap()` on an `Err` value",  // msg_data_ptr (`&str`._slice_dat
    0x2b                                            // msg_len (`&str`._slice_len)
);
```

Examining the vtable in this call

Note how this vtable likely only has one entry!

```
0x4282f8 .rdata {0x428000-0x434da0} Read-only data

004282f8  08 00 00 00 04 00 00 00  ........

00428300  void* data_428300 = sub_409ba2
00428304  void* data_428304 = sub_4012e8

00428308  0c 00 00 00 04 00 00 00  ........

00428310  void* data_428310 = sub_401b91
00428314  void* data_428314 = sub_4012e8

00428318  10 00 00 00 04 00 00 00  ........

00428320  void* data_428320 = sub_401e84
00428324  char data_428324[0x8] = "NulError"
0042832c  char data_42832c[0xd] = "src\\hasher.rs"
```

Examining the vtable in this call: Defining a vtable type

The `fmt::Debug` trait only requires the implementation of one method:

```
1 trait Debug {
2     fn fmt(&self, f: &Formatter) -> Result;
3 }
```

The vtable type will therefore look something like this:

```
1 struct fmt::Debug::_vtable __packed
2 {
3     void* (* destructor)(void* self);
4     int32_t size;
5     int32_t alignment;
6     int32_t (* fmt)(void* self, struct std::fmt::Formatter* formatter_specification);
7 };
```

Examining the vtable in this call: Defining the vtable

```
0x428300 .rdata {0x428000-0x434da0} Read-only data


00428300  void* data_428300 = sub_409ba2
00428304  void* data_428304 = sub_4012e8

00428308                          0c 00 00 00 04 00 00 00          ........

00428310  void* data_428310 = sub_401b91
00428314  struct fmt::Debug::_vtable data_428314 =
00428314  {
00428314      void* (* destructor)(void* self) = sub_4012e8
00428318      int32_t size = 0x10
0042831c      int32_t alignment = 0x4
00428320      int32_t (* fmt)(void* self, struct std::fmt::Formatter* formatter_specification) = sub_401e84
00428324  }
00428324  char data_428324[0x8] = "NulError"
0042832c  char data_42832c[0xd] = "src\\hasher.rs"
```

*Peeking inside the **fmt** implementation*

```
int32_t sub_401e84(void* self, struct std::fmt::Formatter* formatter_specification)

00401e84    int32_t sub_401e84(void* self, struct std::fmt::Formatter* formatter_specification)

00401e95        int32_t eax = formatter_specification->__offset(0x18).d
00401e98        void* self_1 = self
00401ea7        int32_t eax_1 = (*(eax + 0xc))(formatter_specification->__offset(0x14).d,
00401ea7            "NulErrorsrc\hasher.rs", 8, self_1)
00401eb3        struct std::fmt::Formatter* formatter_specification_1 = formatter_specification
00401eb6        char var_14 = eax_1.b
00401eb9        int32_t var_1c = 0
00401ebc        char var_13 = 0
00401ed6        sub_40a2db(sub_40a2db(eax_1, self + 0xc, &var_1c, sub_401297), &self_1,
00401ed6            &var_1c, sub_410cdc)
00401eea        return sub_40a411(&var_1c)
```

Defining `std::fmt::Formatter`

```
1  struct `std::fmt::Formatter` __packed
2  {
3      __padding char _0[0x14];
4      __padding char _14[4];
5      __padding char _18[4];
6  };
```

```
    struct std::fmt::Formatter __packed
    {
00  ?? ?? ?? ?? ?? ?? ?? ??
08  ?? ?? ?? ?? ?? ?? ?? ??
10  ?? ?? ?? ??
14      __offset(0x14).d
14              ?? ?? ?? ??
18      __offset(0x18).d
18  ?? ?? ?? ??
1c  };
```

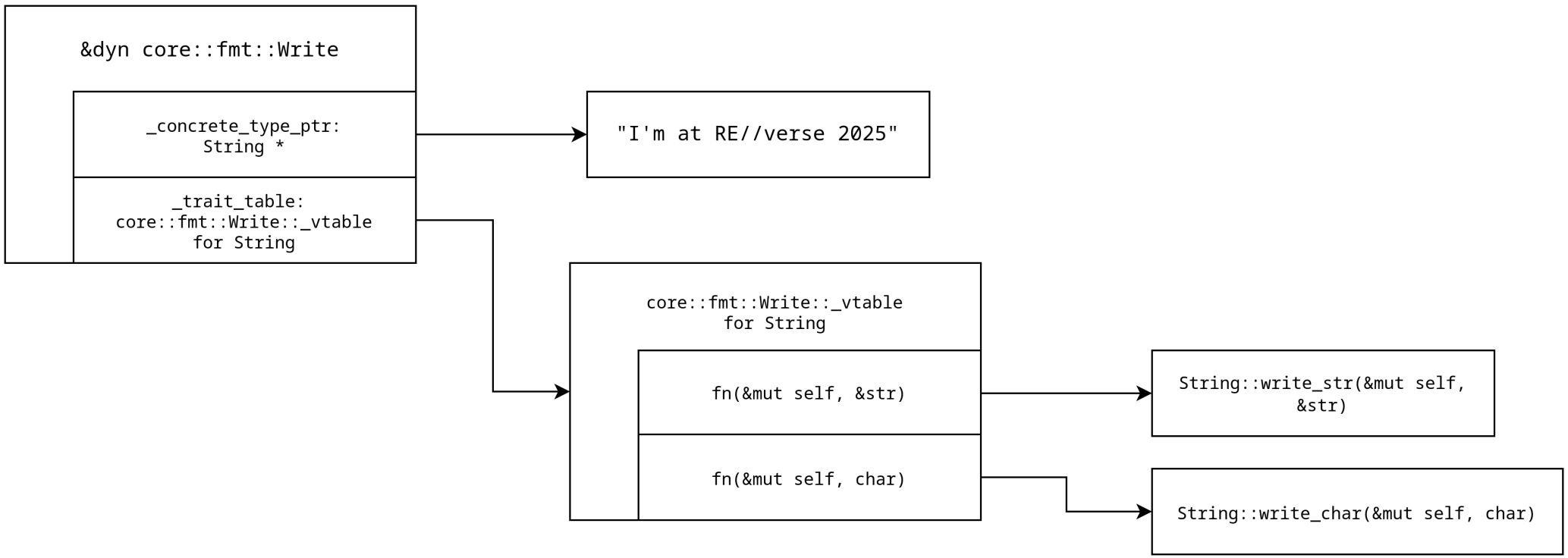
Defining `std::fmt::Formatter`

- 📖 Docs: `core::fmt::Formatter`

```
struct Formatter {
    flags: u32,
    fill: char,
    align: Alignment,
    width: Option<usize>,
    precision: Option<usize>,
    buf: &dyn core::fmt::Write,
}
```

Note our trait object, `&dyn core::fmt::Write,` here!

Looking at `&dyn core::fmt::Write`



Looking at ***&dyn core::fmt::Write***

```
1  struct `&dyn fmt::Write` __packed
2  {
3      void* _concrete_type_data;
4      struct `fmt::Write::_vtable`* _vtable;
5  };
```

Looking at `&dyn core::fmt::Write`

```
1 struct fmt::Write::_vtable __packed
2 {
3     void* (* destructor)(void* self);
4     uint32_t size;
5     uint32_t alignment;
6     int32_t (* write_str)(void* self, char* str_data, usize str_length);
7 };
```

Defining `std::fmt::Formatter`*: The* `buf: &dyn core::fmt::Write` *field*

```
1  struct `std::fmt::Formatter` __packed
2  {
3      __padding char _0[0x14];
4      struct `&dyn fmt::Write` buf;
5  };
```

```
    struct std::fmt::Formatter __packed
    {
00  ?? ?? ?? ?? ?? ?? ?? ??
08  ?? ?? ?? ?? ?? ?? ?? ??
10  ?? ?? ?? ??
14      struct &dyn fmt::Write buf;
1c  };
```

Peeking inside the `fmt` implementation again

After defining `Formatter`, `&dyn Write`, and the `Write` vtable: We can now see an `&str` being passed to `write_str`!

```
int32_t sub_401e84(void* self, struct std::fmt::Formatter* formatter_specification)

00401e84    int32_t sub_401e84(void* self, struct std::fmt::Formatter* formatter_specification)

00401e95        struct fmt::Write::_vtable* _vtable = formatter_specification->buf._vtable
00401e98        void* self_1 = self
00401ea7        int32_t eax = _vtable->write_str(
00401ea7            self: formatter_specification->buf._concrete_type_data,
00401ea7            str_data: "NulErrorsrc\hasher.rs", str_length: 8)
00401eb3        struct std::fmt::Formatter* formatter_specification_1 = formatter_specification
00401eb6        char var_14 = eax.b
00401eb9        int32_t var_1c = 0
00401ebc        char var_13 = 0
00401ed6        sub_40a2db(sub_40a2db(eax, self + 0xc, &var_1c, sub_401297), &self_1, &var_1c,
00401ed6            sub_410cdc)
00401eea        return sub_40a411(&var_1c)
```

Taking advantage of the default `fmt::Debug` trait implementation

Recall that you can just use the `#[derive(Debug)]` to get the compiler to generate a sensible `fmt::Debug` representation:

```
#[derive(Debug)]
struct Coordinates {
    x: i64,
    y: i64,
}

fn main() {
    let cursor_position = Coordinates { x: -100, y: 120 };
    println!("{cursor_position:?}");
}
```

```
Coordinates { x: -100, y: 120 }
```

This prints the name of the type, and all its fields!

A new `NulError` type

```
int32_t sub_401e84(void* self, struct std::fmt::Formatter* formatter_specification)

00401e84    int32_t sub_401e84(void* self, struct std::fmt::Formatter* formatter_specification)

00401e95        struct fmt::Write::_vtable* _vtable = formatter_specification->buf._vtable
00401e98        void* self_1 = self
00401ea7        int32_t eax = _vtable->write_str(
00401ea7            self: formatter_specification->buf._concrete_type_data,
00401ea7            str_data: "NulErrorsrc\hasher.rs", str_length: 8)
00401eb3        struct std::fmt::Formatter* formatter_specification_1 = formatter_specification
00401eb6        char var_14 = eax.b
00401eb9        int32_t var_1c = 0
00401ebc        char var_13 = 0
00401ed6        sub_40a2db(sub_40a2db(eax, self + 0xc, &var_1c, sub_401297), &self_1, &var_1c,
00401ed6            sub_410cdc)
00401eea        return sub_40a411(&var_1c)
```

*Defining a new **`NulError`** type*

We actually have the size and alignment of this type already, from the vtable!

```
0x428300 .rdata {0x428000-0x434da0} Read-only data


00428300  void* data_428300 = sub_409ba2
00428304  void* data_428304 = sub_4012e8

00428308                          0c 00 00 00 04 00 00 00          ........

00428310  void* data_428310 = sub_401b91
00428314  struct fmt::Debug::_vtable data_428314 =
00428314  {
00428314      void* (* destructor)(void* self) = sub_4012e8
00428318      int32_t size = 0x10
0042831c      int32_t alignment = 0x4
00428320      int32_t (* fmt)(void* self, struct std::fmt::Formatter* formatter_specification) = sub_401e84
00428324  }
00428324  char data_428324[0x8] = "NulError"
0042832c  char data_42832c[0xd] = "src\\hasher.rs"
```

*Defining a new **`NulError`** type*

```
int32_t sub_401e84(struct NulError* self, struct std::fmt::Formatter* formatter_specification)

00401e84    int32_t sub_401e84(struct NulError* self, struct std::fmt::Formatter* formatter_specification)

00401e95    |   struct fmt::Write::_vtable* _vtable = formatter_specification->buf._vtable
00401e98    |   struct NulError* self_1 = self
00401ea7    |   int32_t eax = _vtable->write_str(self: formatter_specification->buf._concrete_type_data,
00401ea7    |       str_data: "NulErrorsrc\hasher.rs", str_length: 8)
00401eb3    |   struct std::fmt::Formatter* formatter_specification_1 = formatter_specification
00401eb6    |   char var_14 = eax.b
00401eb9    |   int32_t var_1c = 0
00401ebc    |   char var_13 = 0
00401ed6    |   sub_40a2db(sub_40a2db(eax, &self->__offset(0xc).d, &var_1c, sub_401297), &self_1, &var_1c, sub_410cdc)
00401eea    |   return sub_40a411(&var_1c)
```

A likely culprit: The `std::ffi::NulError` type

```
1  struct NulError(usize, Vec<u8>); // Struct with anonymous fields
```

🕮 Docs: `std::ffi:NulError`

> An error indicating that an interior nul byte was found. While Rust strings may contain nul bytes in the middle, C strings can't, as that byte would effectively truncate the string. This error is created by the **`new`** method on **`CString`**.

A likely culprit: The `std::ffi::NulError` type

```
1 use std::ffi::{CString, NulError};
2
3 fn main() {
4     let err: NulError = CString::new(b"f\0oo".to_vec()).unwrap_err();
5     println!("{err:?}")
6 }
```

```
1 NulError(1, [102, 0, 111, 111])
```

```
1 struct NulError(
2     usize,   // Position of null byte in data
3     Vec<u8>  // The rest of the data
4 ); // Struct with anonymous fields
```

Defining the `std::ffi::NulError` type

```
1 struct `std::ffi::NulError` __packed
2 {
3     struct `std::vec::Vec<u8>` _string_data;
4     uint32_t _position_of_null_byte_in_string_data;
5 };
```

Defining the ***std::ffi::NulError*** *type*

```
    struct alloc::raw_vec::RawVec<u8> __packed
    {
00      uint32_t cap;
04      uint8_t* ptr;
08  };

    struct std::vec::Vec<u8>
    {
00      struct alloc::raw_vec::RawVec<u8> buf;
08      uint32_t len;
0c  };

    struct std::ffi::NulError __packed
    {
00      struct std::vec::Vec<u8> _string_data;
0c      uint32_t _position_of_null_byte_in_string_data;
10  };
```

The `_<impl fmt::Debug for std::ffi::NulError>::fmt` function

```
int32_t _<impl fmt::Debug for std::ffi::NulError>::fmt(struct std::ffi::NulError* self, struct st

00401e84    int32_t _<impl fmt::Debug for std::ffi::NulError>::fmt(
00401e84      struct std::ffi::NulError* self,
00401e84      struct std::fmt::Formatter* formatter_configuration)

00401e95      struct fmt::Write::_vtable* buf_vtable = formatter_configuration->buf._vtable
00401e98      struct std::ffi::NulError* self_1 = self
00401ea7      int32_t eax = buf_vtable->write_str(
00401ea7          self: formatter_configuration->buf._concrete_type_data,
00401ea7          str_data: "NulErrorsrc\hasher.rs", str_length: 8)
00401eb3      struct std::fmt::Formatter* formatter_configuration_1 = formatter_configuration
00401eb6      char var_14 = eax.b
00401eb9      int32_t var_1c = 0
00401ebc      char var_13 = 0
00401ed6      _invoke_formatter(
00401ed6          _invoke_formatter(eax, &self->_position_of_null_byte_in_string_data,
00401ed6              &var_1c, _<impl core::fmt::Display for usize>::fmt),
00401ed6          &self_1, &var_1c, _<impl core::fmt::Display for Vec<u8>>::fmt)
00401eea      return sub_40a411(&var_1c)
```

The full vtable for `std::ffi::NulError`

```
0x428314 .rdata {0x428000-0x434da0} Read-only data

00428304  }
00428304  struct fmt::Debug::_vtable vtable_428304 =
00428304  {
00428304      void* (* destructor)(void* self) = _<impl Drop for std::vec::Vec<u8>>::drop
00428308      int32_t size = 0xc
0042830c      int32_t alignment = 0x4
00428310      int32_t (* fmt)(void* self, struct std::fmt::Formatter* formatter_specification) = sub_401b91
00428314  }
00428314  struct fmt::Debug::_vtable _<impl fmt::Debug for std::ffi::NulError>::_vtable =
00428314  {
00428314      // NulError is laid out such that we can just use Vec<u8>'s destructor.
00428314      void* (* destructor)(void* self) = _<impl Drop for std::vec::Vec<u8>>::drop
00428318      int32_t size = 0x10
0042831c      int32_t alignment = 0x4
00428320      int32_t (* fmt)(void* self, struct std::fmt::Formatter* formatter_specification) = _<impl fmt::Debug for std::ffi::NulError>::fmt
00428324  }
00428324  char data_428324[0x8] = "NulError"
0042832c  char data_42832c[0xd] = "src\\hasher.rs"
```

Our one block of code, nicely annotated

```
BOOL __fastcall mw::is_windows7_or_below(char* data_for_hashing, int32_t length_of_data_for_hashing)

0040435e        } else {
00404360            struct std::result::Result<std::ffi::CString, std::ffi::NulError>
00404360                unwrap_result = cstring_new_result
00404363            var_4c
00404363            attempted_string_data_vec_cap = var_4c.d
00404366            attempted_string_data_vec_ptr = attempted_string_data_vec_ptr_1
00404369            _position_of_null_byte_in_string_data =
00404369                result_err.Err._position_of_null_byte_in_string_data
00404369
00404370            if (neg.d(unwrap_result) != 0x80000000) {
00404376                result_err._discriminant = unwrap_result
00404379                result_err.Err._string_data.buf.cap = attempted_string_data_vec_cap
0040437c                result_err.Err._string_data.buf.ptr = attempted_string_data_vec_ptr
0040437f                result_err.Err._string_data.len = attempted_string_data_vec_len
00404382                int32_t var_14_2 = 3
0040439f                core::result::unwrap_failed(error_concrete_type_data: &result_err,
0040439f                    error_vtable: &_<impl fmt::Debug for std::ffi::NulError>::_vtable,
0040439f                    panic_location:
0040439f                        &panic_location_"src\is_windows7_or_below.rs"_line_37_col_59,
0040439f                    msg_data_ptr: "called `Result::unwrap()` on an `Err` value",
0040439f                    msg_len: 0x2b)
0040439f                noreturn _llvm_panic() __tailcall
00404370            }
0040435e        }
```

A primer on how to explore Rust internals

There is quite a lot you can figure out just by reading!

- The Rust compiler is open source, and the Rust standard library is open source.
- There is only one production-ready compiler implementation, and only one standard library implementation.
- The Rust compiler is documented: 📖 The rustc book, 📖 Rust Compiler Development Guide
- The Rust standard library is documented: 📖 doc.rust-lang.org/std/, 📖 stdrs.dev
- The Rust language is documented: 📖 The Rust Reference, 📖 The Unsafe Rust Book (Rustonomicon)

Questions

You can also find me at:

- Mastodon: @cxiao@infosec.exchange
- Bluesky: @cxiao.net
- Website: cxiao.net

illustration of how we all feel

Acknowledgements

Thank you to:

- My colleagues Ken, Josh, Lilly, and Jörg for listening to drafts of this presentation.
- The RE//verse review board for feedback from the dry run, which significantly improved the presentation.

The slide template used here is Grant McDermott's `quarto-revealjs-clean`.

Resources

This presentation would not be possible without the huge amount of documentation, blogs, tutorials and public resources published by the Rust community.

- Matt Oswalt: Polymorphism in Rust: https://oswalt.dev/2021/06/polymorphism-in-rust/
- Marco Amann: Rust Dynamic Dispatching deep-dive: https://medium.com/digitalfrontiers/rust-dynamic-dispatching-deep-dive-236a5896e49b
- Raph Levien: Rust container cheat sheet: https://docs.google.com/presentation/d/1q-c7UAyrUlM-eZyTo1pd8SZ0qwA_wYxmPZVOQkoDmH4/edit#slide=id.p
- Mara Bos: Behind the Scenes of Rust String Formatting: format_args!(): https://blog.m-ou.se/format-args/
- Rust to Assembly: Understanding the Inner Workings of Rust: https://www.eventhelix.com/rust/
- fasterthanlime - Peeking inside a Rust Enum: https://fasterthanli.me/articles/peeking-inside-a-rust-enum
- Rust Language Cheat Sheet: https://cheats.rs/

- Primitive Type fn: ABI Compatibility of Rust-to-Rust calls: https://doc.rust-lang.org/core/primitive.fn.html#abi-compatibility

- The Rust Reference: Dynamically Sized Types: https://doc.rust-lang.org/reference/dynamically-sized-types.html
- The Rust Reference: Type Layout: https://doc.rust-lang.org/reference/type-layout.html
- The Rust Reference: Destructors: https://doc.rust-lang.org/reference/destructors.html
- Changes to `u128`/`i128` layout in 1.77 and 1.78: https://blog.rust-lang.org/2024/03/30/i128-layout-update.html

- The Rustonomicon: https://doc.rust-lang.org/nightly/nomicon/
- Exploring dynamic dispatch in Rust: https://alschwalm.com/blog/static/2017/03/07/exploring-dynamic-dispatch-in-rust/
- Rust Deep Dive: Borked Vtables and Barking Cats: https://geo-ant.github.io/blog/2023/rust-dyn-trait-objects-fat-pointers/
- About `vtable_allocation_provider`: https://www.reddit.com/r/rust/comments/11okz75/comment/jbt969m/
- https://github.com/rust-lang/rust/pull/86461/files
- https://github.com/rust-lang/rust/blob/1.83.0/compiler/rustc_middle/src/ty/vtable.rs

- How is `__rust_dealloc` function connected to `__rdl_dealloc` function?: https://users.rust-lang.org/t/how-is-rust-dealloc-function-connectted-to-rdl-dealloc-function/122159
- What is difference between a unit struct and an enum with 0 variants?: https://www.reddit.com/r/rust/comments/1hw19el/what_is_difference_between_a_unit_struct_and_an/