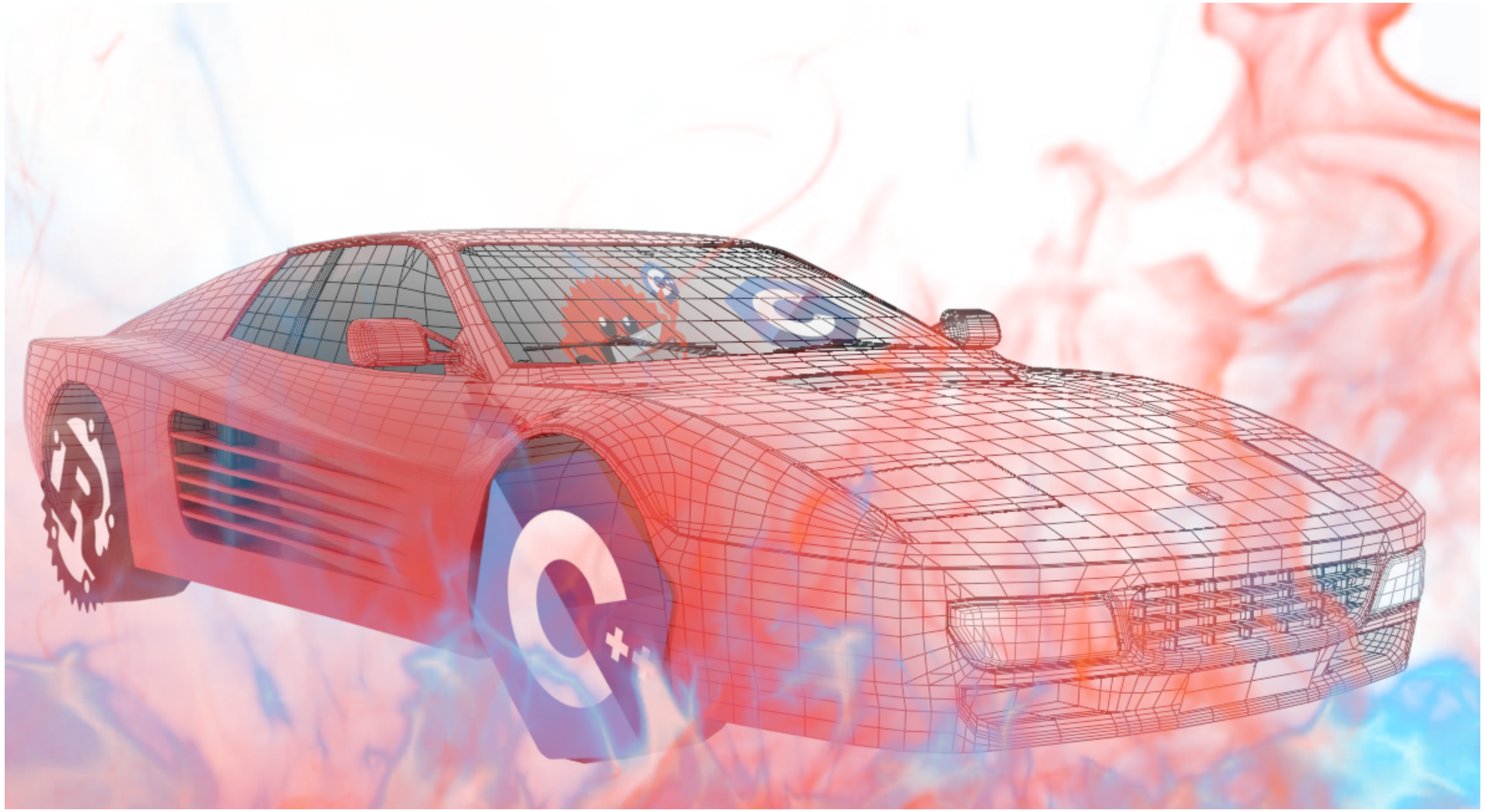
Effizientes Programmieren in C, C++ und Rust

Rust - Ownership and Borrowing

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024



Recap

1. What is your first impression of Rust?
2. What do you think about pattern matching?

Table of Contents

Ownership and Borrowing

- Topic of the lecture: understanding how simultaneous safety and efficiency is possible
- Safety = no undefined behavior↗
- Note: Before Rust, it was not widely accepted that this is even possible

developer.okta.com/blog/2022/03/18/programming-security-and-why-rust

Why is Safety Important?

- Computing the wrong result fast is not that helpful
- UB is often a security issue
- Avoiding UB is doable, but does not scale well to large teams

  $\Rightarrow$ human efficiency!
- Programming without UB is much more enjoyable!*

** author's note: we exclude masochists here for the sake of clarity*

Basic Ownership

Ownership in C++ and Rust

- Without garbage collection, another cleanup mechanism is necessary
- Solution: RAII pattern
  (resource acquisition is initialization)
  - *Single owner for each object: a variable or another object*
  - *Call destructor (in Rust: `drop`) when a variable goes out of scope*

$\Rightarrow$ Typically results in a tree structure

Difference: Destructive Moves

- C++: moved-from object still exists in "null" state

```cpp
std::string s1("abc");
std::print("{}", s1.size());  // 3
std::string s2 = std::move(s1);
std::print("{}", s1.size());  // 0
```

- Rust: compiler forbids reuse
  (this also saves a drop call!)

```rust
let s1 = String::from("abc");
println!("{}", s1.len());  // 3
let s2 = s1;
println!("{}", s1.len());
// error[E0382]: borrow of moved value: `s1`
```

Basic Ownership

Two Kinds of Objects

- Not all objects hold a resource (e.g., allocation or file descriptor)
- "Plain" structs are marked via the `Copy`-trait
  $\Rightarrow$ copy-semantics

COPY

```
let s1: i32 = 42;
let s2 = s1;
println!("{}", s1);
// 42
```

NON-COPY

```
let s1: String = String::from("42");
let s2 = s1;
println!("{}", s1);
// error[E0382]: borrow of moved value: `s1`
```

Aside: Traits

- For now, think **`trait`** = **`interface`** *(details later)*
- Marker traits without methods mark certain properties of objects

Efficiency 🚴

- Moves **and** copies are a bitwise copy of the struct members
  - No unexpected allocations like in C++
  - As efficient as it gets
  - Problematic for self-references

Basic Ownership

Copy and Clone

- Copy: struct supports automatic, cheap copies
  - *Edge case: very large structs, e.g. arrays*
- How to create copies of non-copy structs?
- Clone: allows to do non-trivial copies via a call to `.clone()`
  - *Cloning typically creates a new allocation*

$\Rightarrow$ New allocations are clearly marked since explicit calls are needed!

Drop

- Non-copy structs often need cleanup code (e.g. freeing the allocation)
- Implemented via the `Drop`-trait
- Automatically called at the end of the current lexical scope
- Early drop via `std::mem::drop`

Implementing `std::mem::drop`

```
pub fn drop<T>(_x: T) {}
// automatic drop at end of scope!
```

Yes, that's everything!'

```
let s = String::from("42");
std::mem::drop(s);
//             ^^^^^^^ s is moved into the function
println!("{}", s);
// error[E0382]: borrow of moved value: `s`
```

References and The Hierarchy of Power

References in Rust

- `&mut`: exclusive reference
  (also called "mutable" reference)
- `&`: shared reference
  (also called "immutable" reference)

- Unlike C++, references are normal values (e.g. reassignable)
- References have an associated lifetime↗

Usage of References

- Explicit syntax similar to C pointers
  - `&`/`&mut` to create a reference (borrowing)
  - `*` for dereferencing

```rust
fn longest_string(input: &[String]) -> &String {
    let mut longest: &String = &input[0];
    for s in &input[1..] {  // skip first element
        if s.len() > longest.len() {
            longest = s;  // reassign reference
        }
    }
    longest
}
```

(actually, `&str` would make more sense here)

References and The Hierarchy of Power

Refences and Typing

- References are full-blown types *(with a lifetime)*
- `mut` annotations for variables are not part of the type

  - `fn foo(mut val: &i32)`

    is exactly the same as

    `fn foo(val: &i32)`

  - `fn foo(val: &mut i32)`

    is very different to

    `fn foo(val: &i32)`

Three Modes of Data

- Owned (T): arbitrary mutation and destruction
- Exclusive access (&`mut` T): arbitrary mutation, but value must remain valid
- Shared access (&T): read-only *(except in case of interior mutability↗)*

$\Rightarrow$ Each level has strictly more restrictions than previous one

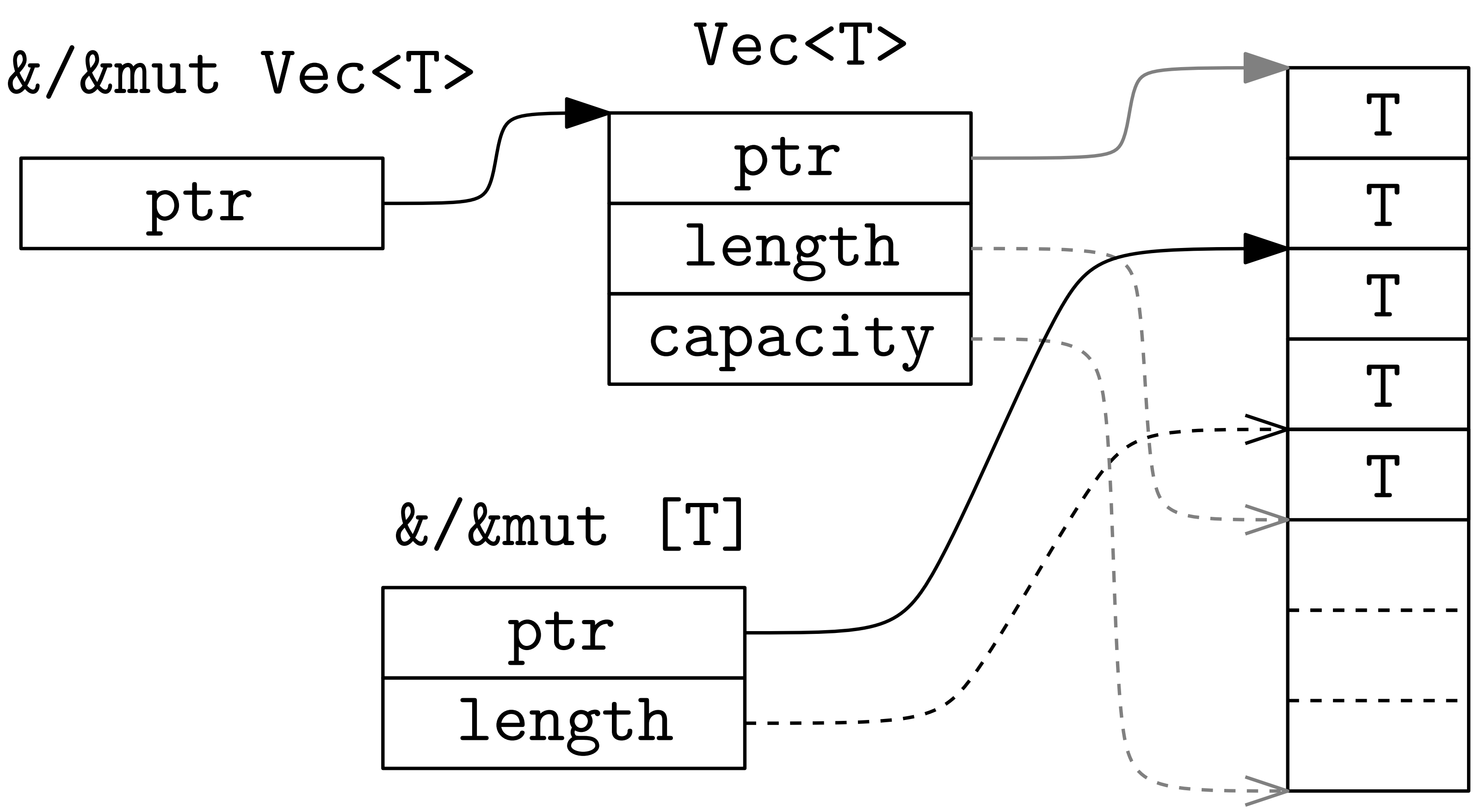
- Note: this mostly applies to C++,* too, but Rust makes it more explicit

** let's ignore `const_cast` here*

References and The Hierarchy of Power

Owned and Non-Owned Types

- `Vec<T>`: container with owned data
- `&/&mut Vec<T>`: reference to a `Vec<T>`
- `&/&mut [T]`: slice, i.e., reference to a contiguous part of the data
- *`[T]`: also a type on its own – data of unknown length (dynamically sized)*



Even More Types

- `String` and `&str` work analogous to `Vec<T>` and `&[T]`
- `Box<[T]>`, `Rc<[T]>` and `*const [T]` (smart/raw pointer to `[T]`)
- The `Fn`, `FnMut`, `FnOnce` hierarchy (traits for functions and lambdas)

$\Rightarrow$ `Using &Vec<T>` doesn't make sense,
`&[T]` is superior (compare `&i32`/`i32`)

- *Though it might appear in generic code*

Borrow Checker – Motivation

The 🐘 in the Room

- References have a lifetime and are subject to borrow checking
- &T is actually &'a T, where 'a is a (usually anonymous) lifetime
- Borrow checking (severely) restricts the valid usage of references

Borrow checking ensures that ...

- ... no reference outlives its data
- ... exclusive references are actually exclusive

THE GOLDEN RULE
OF BORROW CHECKING

> There may be either one exclusive reference or multiple shared references to the same data at any given time.

- Why does this make sense?

Borrow Checker – Motivation

The Ingredients of Safety

- No nullability
- Bounds checking for array accesses, well defined overflow
- Separating safe and unsafe code via safe wrapper APIs
  - Safe API = no possible usage can result in undefined behavior (UB)
- The borrow checker (?)
- Actually, why do we even need the borrow checker?

"Safe C++"

- Thought experiment: apply first three points to C++
  - *Add nullability checks and bounds checks*
  - *Make overflow (and similar "unnecessary" undefined behavior) well defined*
  - *Add `unsafe` keyword to mark remaining UB*
- Require safe wrappers around any unsafe code

$\Rightarrow$ We just made C++ safe*

(or did we?)

** and we are not the first to try*

Borrow Checker – Motivation

"Safe C++" – the Unsafe Parts

- Main question: What needs to be unsafe?

  ⇒ Let's look at an example!

```
int push_and_compute(std::vector<int>& input,
                     const int& val) {
    int new_element = compute_from(val);
    input.push_back(new_element);
    return compute_result(val);
}
```

```
int main() {
    std::vector<int> input = {1};
    return push_and_compute(input, input[0]);
}
```

- This code is completely fine
- ...
- right?

🙁

```
AddressSanitizer: heap-use-after-free
on address 0x602000000010
```

Borrow Checker – Motivation

"Safe C++" – the Problems

```
std::vector<int> input = {1};
return push_and_compute(input, input[0]);
```

- What happened?
  - The vector reallocated its content
  - The reference got invalidated
- Source of the problem: the referenced data changed (aka aliasing)*

⇒ Most uses of references, pointers or
iterators must be unsafe
... but this is the majority of C++ code?!

* in a multi-threaded setting, this is problematic even without data
changing its location (data races)

```
AddressSanitizer: heap-use-after-free
on address 0x602000000010
```

A Possible Solution

- Let the compiler track the origin of references to prevent invalid aliasing
- Congratulations!
  We just rediscovered the basic idea of the borrow checker

Borrow Checker – Motivation

- For most data types, mutation requires an exclusive reference

  ⇒ Prevents referenced data from changing unexpectedly
- References are invalidated if an object is moved or dropped

```rust
let mut input = vec![1];
push_and_compute(&mut input, &input[0]);
```

```
error[E0502]: cannot borrow `input` as immutable because it is also borrowed as mutable
  --> src/main.rs:11:35
   |
11 |     push_and_compute(&mut input, &input[0]);
   |     ---------------- ----------   ^^^^^ immutable borrow occurs here
   |     |                |
   |     |                mutable borrow occurs here
   |     mutable borrow later used by call
```

There may be either one exclusive reference **or** multiple shared references to the same data at any given time.

Lifetimes

- References might be function inputs or ouputs
- How does the borrow checker know the origin of returned references?

⇒ It does not, we need to specify!

```
fn which_lifetime(
    a: &i32, b: &i32, c: &i32
) -> &i32 {
    if cond1 { a }
    else if cond2 { b }
    else { c }
}
```

```
error[E0106]
: missing lifetime specifier
```

LOCALITY RULE:

All information relevant for type checking or borrow checking must be contained in the function signature.

Lifetimes

- Lifetimes describe the possible origins of a reference...
- ...or, equivalently, how long it is valid

```
fn which_lifetime<'lt>(
    a: &'lt i32, b: &'lt i32, c: &'lt i32
) -> &'lt i32 {
    if cond1 { a } else if cond2 { b } else { c }
}
```

"The returned reference might originate from a, b or c."

"The returned reference must not live longer than (any of) a, b or c."

```
fn which_lifetime<'lt>(
    a: &'lt i32, b: &i32, c: &i32
) -> Option<&'lt i32> {
    if cond(b, c) { Some(a) } else { None }
}
```

"The returned reference always originates from a."

$\Rightarrow$ Lifetimes allow to track reference origins using only function signatures*

** yes, the syntax is a bit ... special*

Lifetime Elision

- Only lifetimes relevant for the output need to be specified
- Often, lifetimes can be fully elided *(i.e. inferred by the compiler using fixed rules)*
  - Only one input lifetime
  - Methods: lifetime originates from `&self`

```
fn which_lifetime<'lt>(
    a: &'lt i32, b: &i32, c: &i32
) -> Option<&'lt i32> {
    if cond(b, c) { Some(a) } else { None }
}
```

implicitly expands to

```
fn which_lifetime<'lt, 'b, 'c>(
    a: &'lt i32, b: &'b i32, c: &'c i32
) -> Option<&'lt i32> {
    if cond(b, c) { Some(a) } else { None }
}
```

Rule of thumb: one named lifetime suffices in 95% of cases

Lifetimes

Lifetimes in Structs

```
struct ContainsReference<'ref> {
    ref_data: &'ref i32,
}
```

- This compiles...
- ...but is a bad implementation!

```
fn transform(val: &i32) -> &i32 {
    ContainsReference {
        ref_data: val
    }.get_data()
} // error: cannot return value
  // referencing temporary value
```

```
impl<'ref> ContainsReference<'ref> {
    fn get_data(&self) -> &i32 {
        self.ref_data
    }
}
```

expands to

```
fn get_data<'self>(&'self self) -> &'self i32 {
    self.ref_data
}
```

- But `&self` is not the actual origin!

Fixed version:

```
impl<'ref> ContainsReference<'ref> {
    fn get_data(&self) -> &'ref i32 {
        self.ref_data
    }
}
```

Ownership and Borrowing – Intermediate Conclusion

- Ensure memory safety by preventing aliasing
- The problem: combining sharing and mutation

More than Memory Safety

```
fn some_business_logic(data: &mut DataHolder,
                       value: &Value) {
    let old_val = *value;
    some_subroutine(data);
    assert_eq!(old_val, *value);
    // Rust guarantees this always holds
}
```

- Typical OO programs have no such guarantee
  (`data` might reference `value`)
- Rust makes mutation locally explicit
- Aliasing can also cause logic bugs!

Locality of Reasoning

	No mutation	Mutation
No sharing	👍	👍
Sharing	👍	👎

- Common adage in Rust:
  "if it compiles, it works"*

** except, of course, for logic bugs not related to aliasing,
as well as memory leaks, deadlocks, ...*

Ownership and Borrowing – Intermediate Conclusion

- Single owner principle leads to tree-shaped ownership
- Referenced data may not be mutated through other paths

$\Rightarrow$ Storing references permanently is usually a bad idea *(Generally, references should be kept "stack-downwards" of the origin)*

👍 Benefits

- Safety! Correctness!
- Lifetimes allow complex computations with references
- We can use references "fearlessly"

$\Rightarrow$ Avoids defensive copies

Drawbacks 👎

- Enforces rather restrictive data layout
- Manually specified lifetimes are tricky (lifetime errors...)

What if the data is inherently not tree-shaped?

Complex Ownership

Strategy 1: Smart Pointers and Interior Mutability

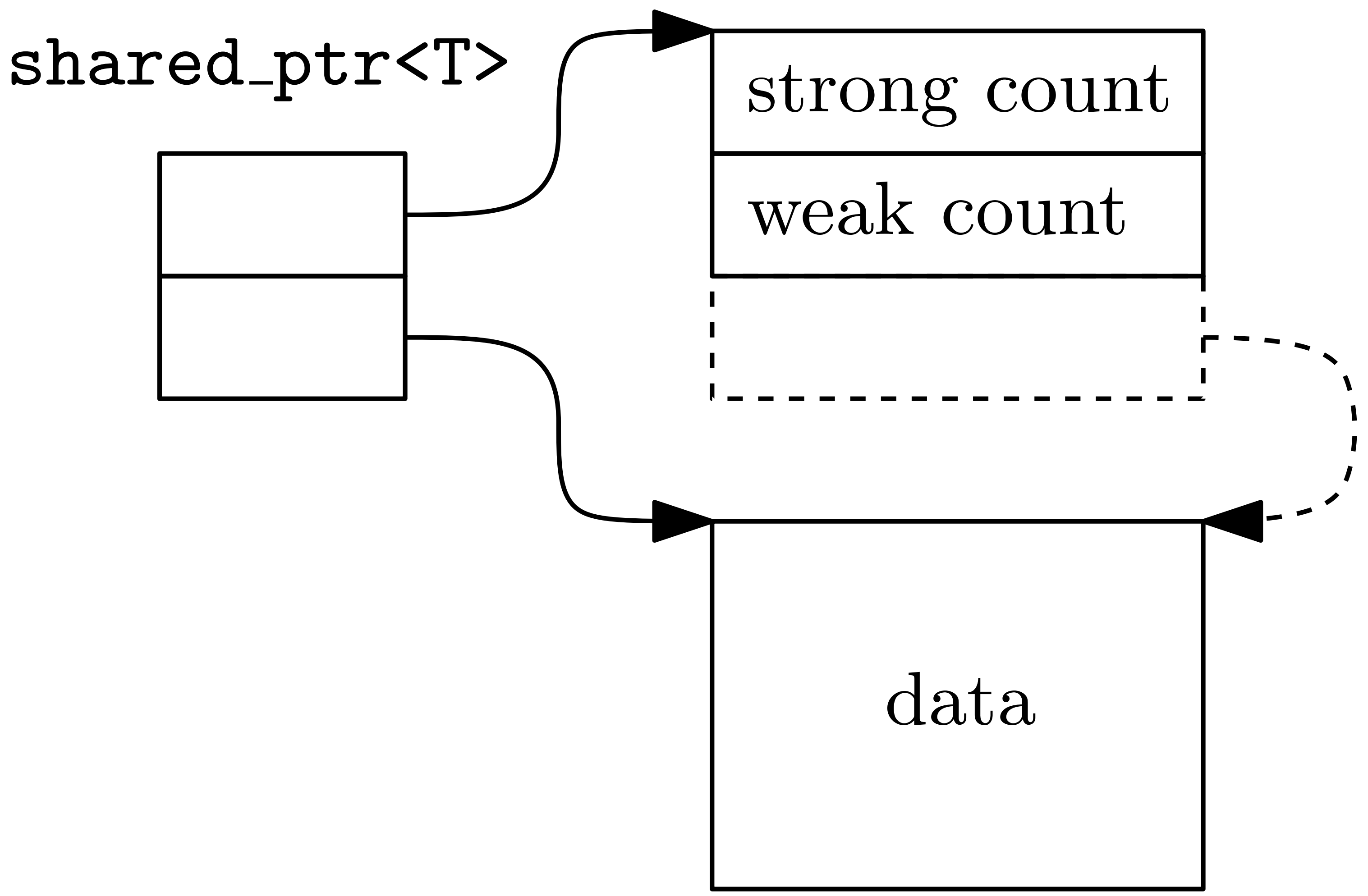
- Smart pointers support different ownership strategies

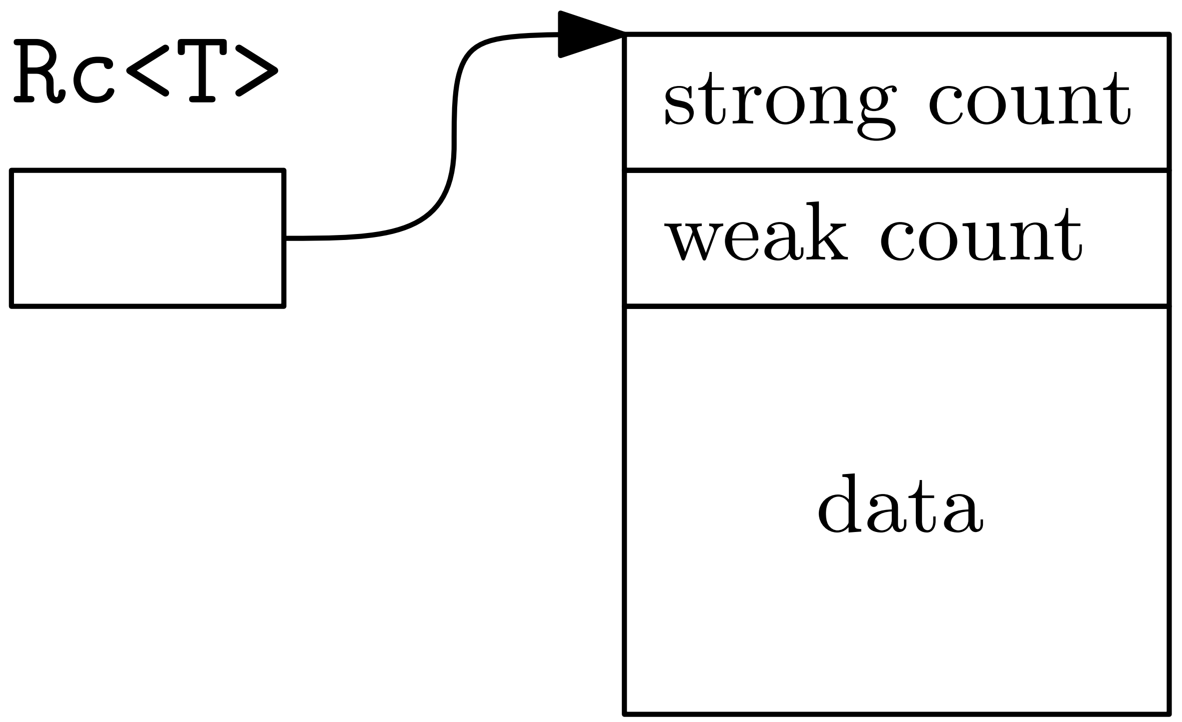
C++	Rust	
`std::unique_ptr<T>`	`Box<T>`	Heap-allocated owned data
`std::shared_ptr<T>`	`Rc<T>` *(not thread-safe)*	Reference counted, shared ownership
	`Arc<T>` *(thread-safe)*	Atomic reference counted, shared ownership

- Careful: reference counted cycles can cause memory leak!

Smart Pointers and Interior Mutability

- `Rc` allows single-threaded reference counting without synchronization overhead
  - *Not available in C++!*
- Rust always uses the same allocation for counter and data
  - 👍 Better memory layout
  - 👎 Creation requires copying the data





Smart Pointers and Interior Mutability

- Restriction: only allows shared (= immutable?) references to the data
- Solution: Interior Mutability!

Interior Mutability: pattern that allows mutation through shared references by using specialized container types

TYPES WITH INTERIOR MUTABILITY

`RefCell<T>`	checks at runtime for exclusiveness,returning an exclusive reference if possible
`Cell<T>`	allows mutation only via copies (no reference involved)
`Mutex<T>`	ensures thread-safe exclusive access via locking
`UnsafeCell<T>`	unsafe core primitive for interior mutability, basis for all the other types

Smart Pointers and Interior Mutability

Example: Updating Data Inside a Mutex

```
let data: Arc<Mutex<MyData>> = initial_arc.clone();
// ...
let mut guard = data.lock().unwrap();  // lock the mutex
       // ^--  the guard ensures exclusive access
*guard = computeUpdate(guard.borrow());
mem::drop(guard);  // unlock the mutex
// ...
```

Smart Pointers and Interior Mutability

- Reference counting
  - Efficient access
  - Memory overhead: two counters, separate allocation
- Interior Mutability
  - Efficient for small values via `Cell`, but copying inefficient for large data
  - `RefCell` needs runtime-check

👍 Benefits

- Allows safe shared ownership
- Reference counting for large immutable data chunks is efficient
- Enables some typical OO patterns (objects referencing each other)

👎 Drawbacks

- Significant memory overhead if small data chunks are shared
- Mutating large data via `RefCell` involves runtime-check
- OO patterns are often a bad idea *(in Rust)*

$\Rightarrow$ Design your data layout to avoid interior mutability, but use it where necessary

Complex Ownership

Strategy 2: Arena Allocation

- Pattern for handling many identical objects
- Idea: don't use pointers, instead use a central data structure (= arena) and IDs
- Simple variant: `Vec<T>` with indices as IDs

Avoiding the borrow checker

- No borrow checking on IDs $\Rightarrow$ no issues with sharing
- Borrow checking now happens on the arena reference
- If elements can be deleted: **ID invalidation** *("safe" version of use-after-free)*
  - Can be mitigated with runtime checks $\Rightarrow$ overhead!

    (see: generational indices, the slotmap library)

Arena Allocation

Example: A Static Graph Data Structure

```
// macro to derive basic properties for IDs
#[derive(Debug, Clone, Copy, ...)]
pub struct NodeID(pub u32);
#[derive(Debug, Clone, Copy, ...)]
pub struct EdgeID(pub u32);

struct NodeData{ first_edge: EdgeID, weight: i32 }
struct EdgeData{ target_node: NodeID, weight: i32 }

pub struct Graph { nodes: Vec<NodeData>, edges: Vec<EdgeData> }

impl Graph {
    pub fn node_weight(&self, node: NodeID) -> i32 {
        self.nodes[node.0 as usize].weight
    }

    pub fn nodes(&self) -> impl Iterator<Item=NodeID> {
        // Iterator over nodes. Remove last entry (which is a sentinel
        (0..(self.nodes.len() - 1) as u32).map(NodeID)
    }
}
```

Arena Allocation

- Safety: access via the ID needs a bounds check

  $\Rightarrow$ Provide an unsafe getter for hot code

👍 Benefits

- Elegantly avoids borrow checker issues
- Very efficient memory layout and allocation of elements
- For smaller data sets, we can use 32-bit IDs

👎 Drawbacks

- Avoiding the borrow checker means less safety: ID invalidation
- Always requires a reference to the arena
- Only with bounds checking safe
- Compared to pointers, access requires an additional offset calculation

$\Rightarrow$ Always consider using arena allocation if there are many similar objects!

Complex Ownership

Strategy 3: `unsafe` and Raw Pointers

- Sometimes we can't accept any overhead

  $\Rightarrow$ This is the raison d'être for `unsafe`

- Rust supports raw pointers that are mostly equivalent to C
  (more details in the next lecture ↗)

👍 Benefits

- No overhead
- Can be encapsulated in a safe data structure
- Sometimes there is **no other possibility**

👎 Drawbacks

- Opens the door to undefined behavior
- Some rules are even trickier than C/C++

Complex Ownership

Strategy 1: Smart Pointers and Interior Mutability

- Enables shared ownership and mutation at once
- Has inherent overhead and is not very ergonomic to use

Strategy 2: Arena Allocation

- A generally useful and quite efficient strategy
- Not always applicable (needs many similar objects)

Strategy 3: `unsafe` and Raw Pointers

- Useful if no overhead is acceptable
- Allows efficient custom data structures

Note: Arena allocation is often more efficient🚴 than raw pointers!

Generics and Traits

- Generic programming is necessary for strong abstraction support
- "Safer" variant of templates: generic parameters must be annoted with trait bounds *(similar to enforced concepts)*
- Implemented via monomorphization (separate compilation of each variant)

```rust
struct GenericData<T> {
    data: Vec<T>
}

impl<T> GenericData<T> {
    // trait bound: S must be convertible to T
    fn add<S: Into<T>>(&mut self, value: S) {
        self.data.push(value.into());
    }
}
```

Traits

- Traits define shared functionality of different types
- Includes constants, types, static functions and methods

```rust
trait AbstractVector {
    // associated constant
    const DIMENSION: usize;
    // associated type
    type Scalar;
    // associated function
    fn new_zeroed() -> Self;
    // associated method
    fn get(&self, index: usize)
        -> Option<Self::Scalar>;
}
```

Generics and Traits

Using Traits

- Generic code must specify all required properties via trait bounds
- `where` allows to conveniently add more complex bounds

```
trait AbstractVector {
    const DIMENSION: usize;
    type Scalar;
    fn new_zeroed() -> Self;
    fn get(&self, index: usize)
        -> Option<Self::Scalar>;
}
```

```
fn compute_sum<V: AbstractVector>(value: &V)
                -> V::Scalar
    // additional bounds
    where V::Scalar: AddAssign + Default
{
    // Default trait for initial zero
    let mut result = Default::default();
    for index in 0..V::DIMENSION {
        // AddAssign trait for addition
        result += value.get(index).unwrap();
    }
    result
}
```

Generics and Traits

Implementing Traits

- Traits are implemented via a specific `impl` block
- Traits can be implemented for already existing types! *(aka extension methods)*

```
trait AbstractVector {
    const DIMENSION: usize;
    type Scalar;
    fn new_zeroed() -> Self;
    fn get(&self, index: usize)
        -> Option<Self::Scalar>;
}
```

```
impl AbstractVector for f64 {
    const DIMENSION: usize = 1;
    type Scalar = f64;

    fn new_zeroed() -> Self { 0.0 }

    fn get(&self, index: usize) -> Option<f64> {
        (index == 0).then_some(*self)
    }
}
```

Orphan rule (simplified):
A trait implementation is valid if either the trait or the implementing type is local.

More on Generics

- **Three kinds** of generic parameters:
  - Lifetimes*
  - Type parameters
  - Const generics

```
struct RefToArray<'a, Content,
                  const N: usize> {
    data: &'a [Content; N],
}
```

Similarities to C++

- Type parameters + traits are turing complete
- Compilation of generics via monomorphization

  ⇒ Zero overhead! 💸
- `const` is like `constexpr`, but more restricted (for now)

Differences to C++

- Type checking happens before monomorphization
  - Enabled by the explicit bounds

  ⇒ Errors are detected earlier (at declaration instead of usage)

  ⇒ Better error messages
- Explicit bounds can be restrictive

* *lifetimes are a bit special: they are erased instead of monomorphized and allow subtyping through variance*

More on Generics

```rust
fn try_create() {
    let mut vec = Vec::<Uncloneable>::new();
    vec.resize(10, Uncloneable);
}
```

```cpp
void try_create() {
    std::vector<Uncopyable> vec;
    vec.resize(10, Uncopyable{});
}
```

```
error[E0599]: the method `resize` exists for struct `Vec<Uncloneable>`,
but its trait bounds were not satisfied
 --> <source>:7:9
  |
2 | struct Uncloneable;
  | ------------------ doesn't satisfy `Uncloneable: Clone`
...
7 |     vec.resize(10, Uncloneable);
  |         ^^^^^^
  |
  = note: the following trait bounds were not satisfied:
          `Uncloneable: Clone`
help: consider annotating `Uncloneable` with `#[derive(Clone)]`
  |
2 + #[derive(Clone)]
3 | struct Uncloneable;
  |

error: aborting due to 1 previous error
```

```
In file included from <source>:2:
In file included from /opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/vector:63:
In file included from /opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/allocator.h:46:
In file included from /opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/x86_64-linux-gnu/bits/c+
+allocator.h:33:
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/new_allocator.h:187:23: error: call to deleted
constructor of 'Uncopyable'
  187 |         { ::new((void *)__p) _Up(std::forward<_Args>(__args)...); }
      |                              ^   ~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/alloc_traits.h:537:8: note: in instantiation
of function template specialization 'std::__new_allocator<Uncopyable>::construct<Uncopyable, const Uncopyable &>' requested here
  537 |           __a.construct(__p, std::forward<_Args>(__args)...);
      |               ^
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/stl_vector.h:1832:21: note: in instantiation
of function template specialization 'std::allocator_traits<std::allocator<Uncopyable>>::construct<Uncopyable, const Uncopyable &>' requested here
 1832 |             _Alloc_traits::construct(_M_this->_M_impl, _M_ptr(),
      |                            ^
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/vector.tcc:540:25: note: in instantiation of
function template specialization 'std::vector<Uncopyable>::_Temporary_value::_Temporary_value<const Uncopyable &>' requested here
  540 |               _Temporary_value __tmp(this, __x);
      |                                ^
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/stl_vector.h:1034:4: note: in instantiation of
member function 'std::vector<Uncopyable>::_M_fill_insert' requested here
 1034 |           _M_fill_insert(end(), __new_size - size(), __x);
      |           ^
<source>:15:9: note: in instantiation of member function 'std::vector<Uncopyable>::resize' requested here
   15 |     vec.resize(10, Uncopyable{});
      |         ^
<source>:10:5: note: 'Uncopyable' has been explicitly marked deleted here
   10 |     Uncopyable(const Uncopyable& other) = delete;
      |     ^
In file included from <source>:2:
In file included from /opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/vector:65:
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/stl_uninitialized.h:90:21: error: static
assertion failed due to requirement 'is_constructible<Uncopyable, Uncopyable &&>::value': result type must be constructible from input type
   90 |       static_assert(is_constructible<_ValueType, _Tp>::value,
      |                     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/stl_uninitialized.h:182:4: note: in
instantiation of function template specialization 'std::__check_constructible<Uncopyable, Uncopyable &&>' requested here
  182 |         = _GLIBCXX_USE_ASSIGN_FOR_INIT(_ValueType2, _From);
      |           ^
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/stl_uninitialized.h:101:13: note: expanded
from macro '_GLIBCXX_USE_ASSIGN_FOR_INIT'
  101 |     && std::__check_constructible<T, U>()
      |             ^
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/stl_uninitialized.h:373:19: note: in
instantiation of function template specialization 'std::uninitialized_copy<std::move_iterator<Uncopyable *>, Uncopyable *>' requested here
  373 |       return std::uninitialized_copy(__first, __last, __result);
      |                   ^
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/stl_uninitialized.h:384:19: note: in
instantiation of function template specialization 'std::__uninitialized_copy_a<std::move_iterator<Uncopyable *>, Uncopyable *, Uncopyable>'
requested here
  384 |       return std::__uninitialized_copy_a(_GLIBCXX_MAKE_MOVE_ITERATOR(__first),
      |                   ^
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/vector.tcc:548:10: note: in instantiation of
function template specialization 'std::__uninitialized_move_a<Uncopyable *, Uncopyable *, std::allocator<Uncopyable>>' requested here
  548 |                   std::__uninitialized_move_a(__old_finish - __n,
      |                        ^
/opt/compiler-explorer/gcc-13.2.0/lib/gcc/x86_64-linux-gnu/13.2.0/../../../../include/c++/13.2.0/bits/stl_vector.h:1034:4: note: in instantiation of
member function 'std::vector<Uncopyable>::_M_fill_insert' requested here
 1034 |           _M_fill_insert(end(), __new_size - size(), __x);
      |           ^
```
*

** this is the clang error which is IMO notably better than the gcc error: at least the actual problem is stated at the beginning*

Trait Objects

- Sometimes, we really need dynamic dispatch
  - Avoid code bloat
  - Handling different types uniformly at runtime (type erasure)
  - Example: Library that supports user-defined types
- Dynamic dispatch works via traits, too
  - *Note: C++ virtual methods are a completely separate system from templates*
- Trait objects are marked with **dyn** and must be behind some kind of pointer

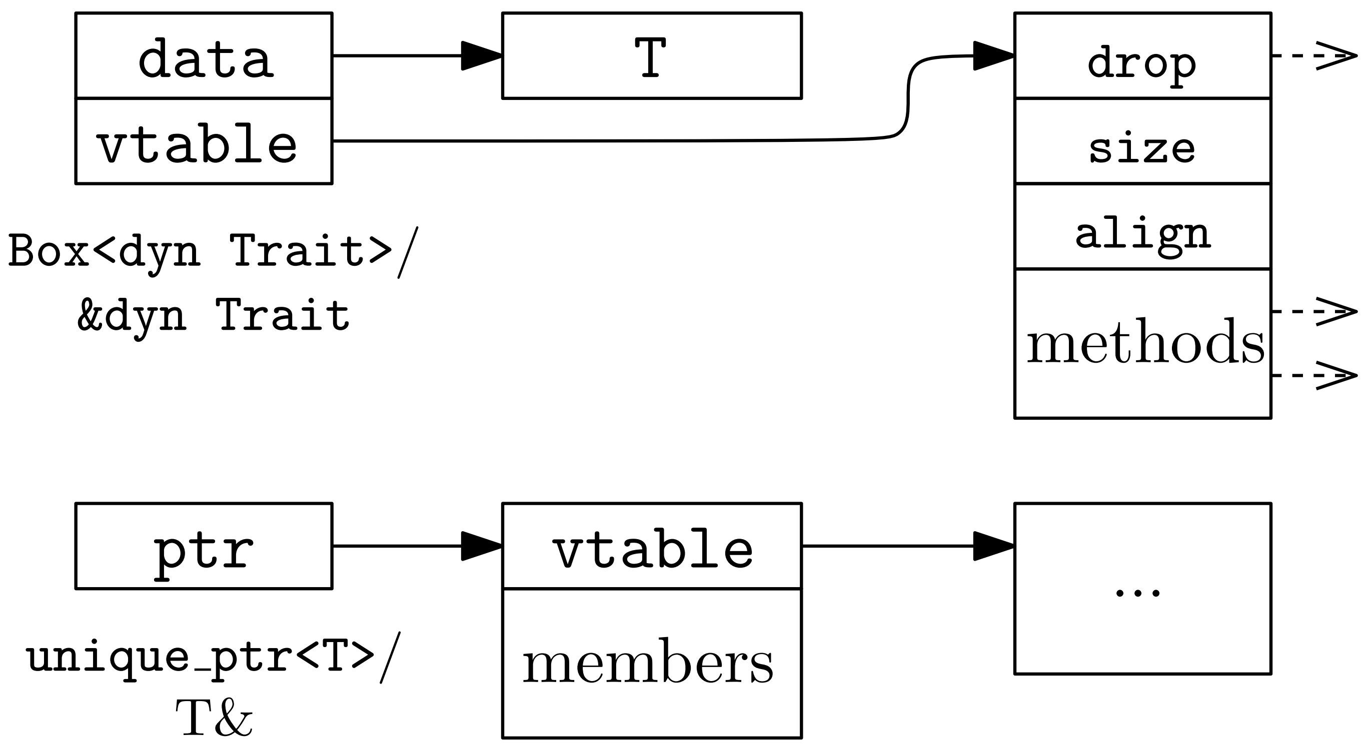
```rust
trait AbstractVectorObj {
    type Scalar;
    fn get(&self, index: usize)
        -> Option<Self::Scalar>;
}
```

```rust
let mut list = Vec::new();
let first: Box<dyn AbstractVectorObj<
    Scalar = f64>> = Box::new(1.0);
list.push(first);
let second: Box<dyn AbstractVectorObj<
    Scalar = f64>> = Box::new([24.0, 42.0]);
list.push(second);
let scalar: f64 = list[0].get(0);
```

Trait Objects

- Based on vtables as in C++
- vtable-pointer not part of the struct *(structs are plain data!)*
- Fat pointers: pointer to the trait object also contains the vtable-pointer

⇒ Restricted to functionality that can be implemented with a vtable!



Object Safety

- No associated constants
- All functions must be methods*
- No generic functions (except for lifetimes)
- Associated types must be specified*

** functions and associated types can be "excluded" from trait objects via* `where Self: Sized`

Trait Objects

- Different memory layout than C++
- vtable-pointer is created "on the fly"

👍 Benefits

- Unified system for static and dynamic polymorphism
- Zero overhead when not used

👎 Drawbacks

- Object safety can be awkward
- Many fat pointers to same object create memory overhead

Rust is Object Oriented

Support for many OO features

- Methods allow to use structs similar to objects
- Structs can contain data and behavior
- Dynamic dispatch via trait objects
- Traits can "inherit" from other traits

Encapsulation

- Visibility management via `pub`
- Hiding of implementation details and preserving invariants
- Traits for shared behavior

Rust is Not Object Oriented

The equivalent of a normal Java object in Rust:

```
Option<Arc<UnsafeCell<dyn Any>>>
```

Everything is nullable

Everything is shared (and garbage collected)

Objects can be freely mutated without checks

Runtime metadata

- Rust has much more restrictions on data layout
- Shared ownership is heavily discouraged

$\Rightarrow$ Typical OO "object soup" architecture is very unidiomatic (and unergonomic)

Rust is Functional

Support for many functional features

- Immutability by default
- Pattern matching
- Iterators with `filter`, `map`, `fold`, etc.
- Monadic types
  (`Option`, `Result`, `Iterator`, ...)

Functional Programming:
A programming paradigm that focuses on avoiding state and mutability and uses pure functions as fundamental abstraction.

Mutability and Type Safety

- Similar to the functional paradigm, Rust restricts
  mutability and avoids it where unnecessary
- A strong type system for catching as many errors at compile time as possible

Rust is Not Functional

- Functional languages tend to avoid mutability completely
- Requires according data structures and optimizations to be approximately efficient

  $\Rightarrow$ Makes many common algorithms impossible

- Rust restricts but still embraces mutability
- Rust has (intentionally) no immutable data structures in the standard library

> "[...] pure functional programming is an ingenious trick to show you can code without mutation, but Rust is an even cleverer trick to show you can just have mutation."*

* *without.boats/blog/notes-on-a-smaller-rust/*

So What?

Conclusion

- Rust combines many of the best features of OO and functional...
- ...and comes with new, previously unknown, drawbacks (lifetimes!)
- While Rust has features from multiple paradigms, both OO and functional program structure are a bad fit

Structure of Rust Programs

- Rust programs tend to use a simple procedural structure
  - Algorithms are implemented within (generic) free functions
  - Explicit flat data layout using arenas
  - Types encode invariants of the data

$\Rightarrow$ A flavor of procedural? A new paradigm?*

- *How about ownership oriented programming (OOP)? ... oh wait*

** at least, Rust has inspired some others*