Ownership types in theory and practice (in Rust)

Fritz Rehde

School of Computation, Information, and Technology
Technical University of Munich

26.01.2023

Overview

Memory safety

- definition: program is memory safe if all memory pointers or references always refer to valid memory
- examples: buffer overflows, memory leaks, double-free, use-after-free
- result: malicious exploits, incorrect program results, "random" crashes

Aliasing

aliasing: accessing same memory through different symbolic names

Aliasing

aliasing: accessing same memory through different symbolic names

C++

```
1 std::vector<int> v { 10, 11 };
2 int *vptr = &v[1]; // points into v
3 v.push_back(12);  // v buffer reallocated => vptr dangling
4 std::cout << *vptr; // bug (use-after-free)
```

Aliasing

aliasing: accessing same memory through different symbolic names

C++

```
1 std::vector<int> v { 10, 11 };
2 int *vptr = &v[1]; // points into v
3 v.push_back(12);  // v buffer reallocated => vptr dangling
4 std::cout << *vptr; // bug (use-after-free)
```

Rust

```
1 let mut v = vec![10, 11];
2 let vptr = &mut v[1]; // 1. mutable reference
3 Vec::push(&mut v, 12); // 2. mutable reference
4 println!("{}", *vptr); // compiler error
```

Observation: an action through an object (v) will also affect all of its aliases ($vptr$), even though these aliases might not "expect" a change.

Ownership types

- using ownership types:
  - limit which objects can be referenced
  - specify whether the referenced objects may be mutated or just read from
  - no garbage collection (overhead)

Ownership types: core concepts

- objects can be *owners* of other objects
- special *world* object is root of the ownership hierarchy tree
- *ownership context*: set of all objects that an object owns
- *siblings*: objects in the same ownership context
- *inside* relation: *a* is *inside b* if they are the same object or *a* is transitively owned by *b*
- *outside* relation: converse of *inside* relation

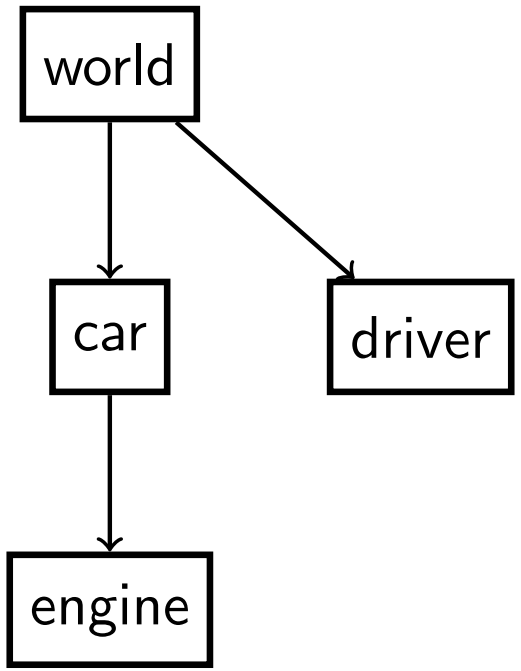


Figure: solid lines indicate "owns"

Owners-as-dominators

For a to validly reference b:

1. a is the owner of b,
2. a and b are siblings, or
3. b is outside of a.

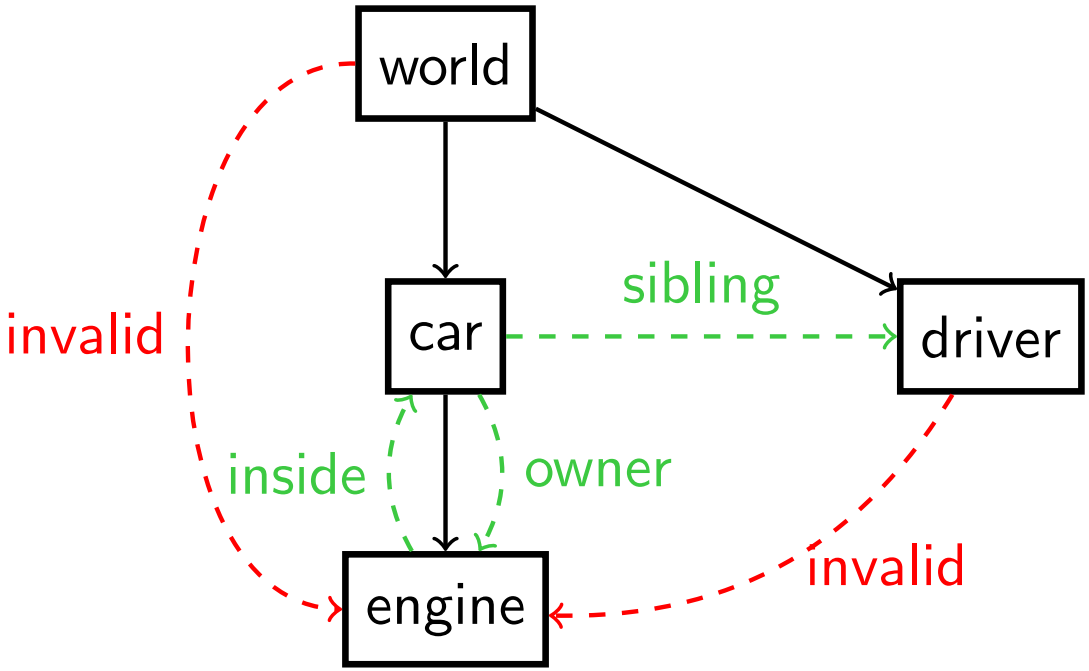
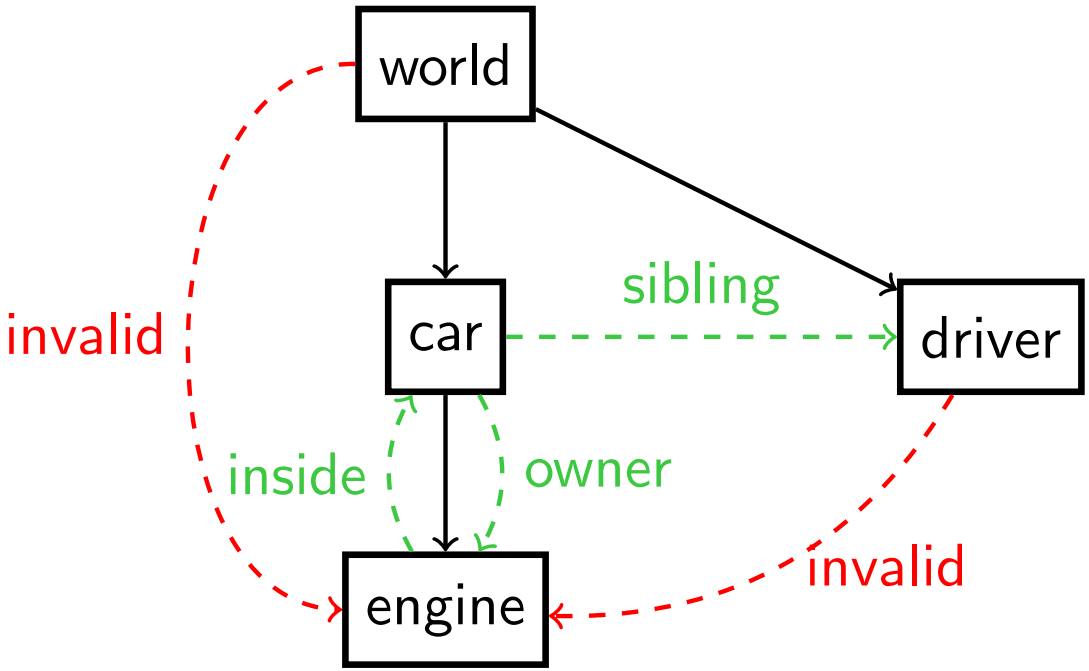


Figure: solid lines indicate "owns" and dotted lines indicate "references"

Owners-as-dominators: implications

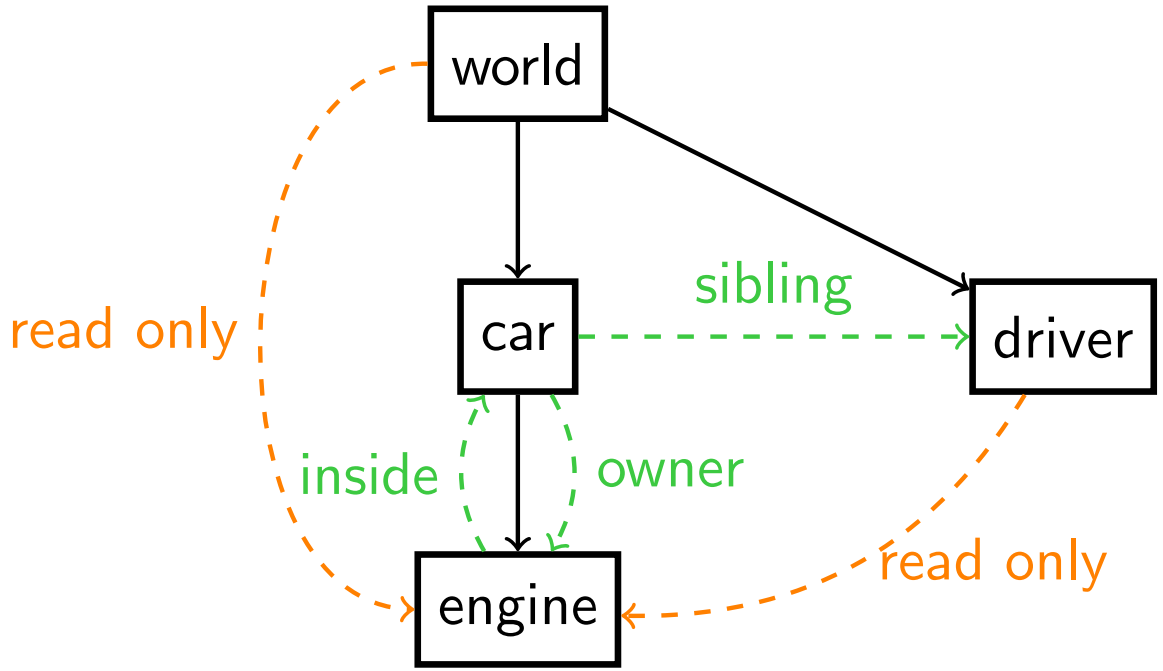
- any external reference to an object's internals must go through its owner
- objects are only protected from external access, not internal access
- model restricts *where* references can point, not *how* references are used
- doesn't allow common idioms that involve aliasing



Owners-as-modifiers

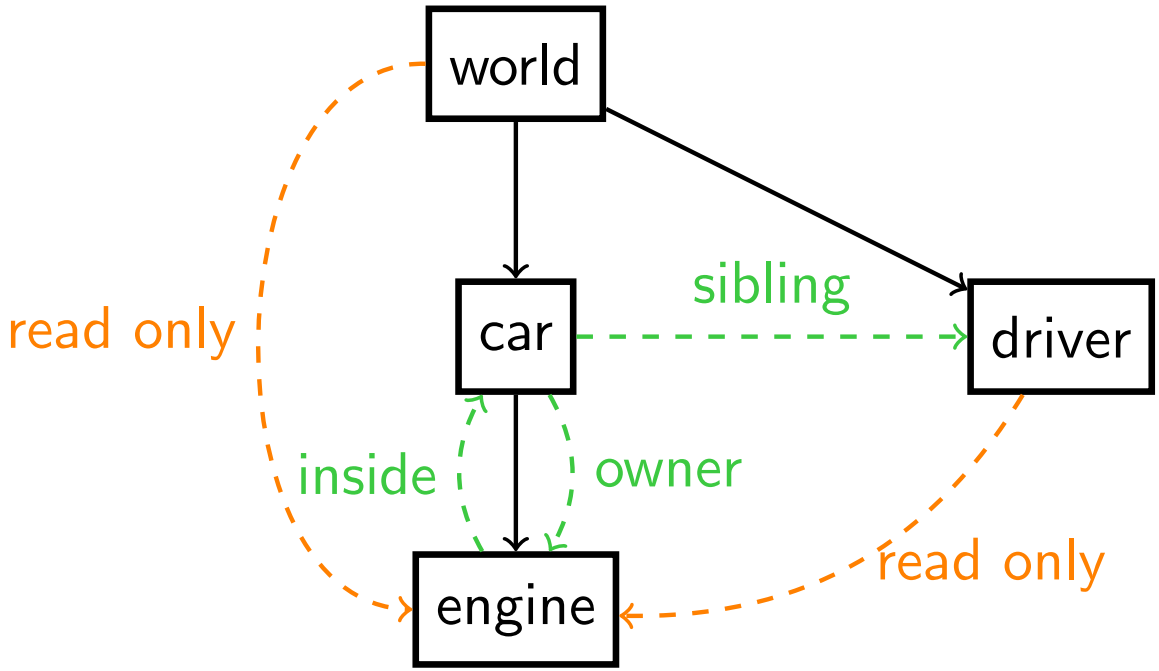
Same rules as owners-as-dominators, but *a* can also reference *b* through *r*, if:

1. *r* is a read-only reference and only pure methods (that don't modify existing objects) can be called on it.



Owners-as-modifiers: implications

- owners-as-modifiers: only the owners can modify objects
- model allows references to objects in arbitrary contexts, but restricts *how* the references can be used



From theory to implementation

- Rust's ownership system: extends owners-as-modifiers
  - immutable (read-only) and mutable references
  - rules for compiler to check validity of references

From theory to implementation

- Rust's ownership system: extends owners-as-modifiers
  - immutable (read-only) and mutable references
  - rules for compiler to check validity of references
- ownership rules in Rust:
  1. Each value has an owner.
  2. There can only be one owner at a time.
  3. When the owner goes out of scope, the value will be *dropped* (associated memory is freed).

```
1  { // a is not declared yet
2    let a = 42; // scope of a starts here
3    // do stuff with a
4  } // scope is over => a is no longer valid
```

From theory to implementation

- Rust's ownership system: extends owners-as-modifiers
  - immutable (read-only) and mutable references
  - rules for compiler to check validity of references
- ownership rules in Rust:
  1. Each value has an owner.
  2. There can only be one owner at a time.
  3. When the owner goes out of scope, the value will be *dropped* (associated memory is freed).

```
1  { // a is not declared yet
2    let a = 42; // scope of a starts here
3    // do stuff with a
4  } // scope is over => a is no longer valid
```

- Variables are immutable by default

```
1  let a = 42; // immutable
2  a = 24; // invalid
3  let mut b = 42; // mutable
4  b = 24; // valid
```

Transferring ownership

Heap-based values:

- unknown size at compile time
- copying the value at runtime could be costly
- values are "moved" (between variables) by default

```
1  fn main() {
2    let a = String::from("hello world");
3    let b = a; // ownership transfer: a is moved into b
4    println!("{}", a); // compiler error: a no longer valid
5    let c = take_ownership(b); // b is moved into function
6  }
7  fn take_ownership(s: String) -> String {
8    // do something with s
9    String::from("new hello world")
10 }
```

- create deep copies with explicit `clone()` call

Transferring ownership

Stack-based primitives:

- examples: i32, u8, f64, char, bool etc.
- small, known size at compile time
- values are copied by default

```
1  let x: u32 = 42;
2  let y: u32 = x; // x is copied instead of moved
3  println!("x: {}, y: {}", x, y); // x and y valid
```

Borrowing

- creating a reference in Rust is called *borrowing*
- *reference*: an address that points to a value that is owned by another variable
- unlike *pointers*, a reference is guaranteed to point to a valid value
- used for performing operations on values without taking ownership
- every reference has a *lifetime*: scope for which reference is valid
- lifetime of borrowed value must be at least as long as lifetime of reference

Borrowing: immutable references

- read-only/shared references $\rightarrow$ compiler disallows modifying borrowed value
- & and * operators
- borrow-checker rule: the borrowed value does not get mutated during lifetime
- unlimited amount of immutable references to same value (if no mutable references or mutations)

```
1  let s = String::from("hello");
2  let r1 = &s; // 1. reference
3  let r2 = &s; // 2. reference
4  println!("r1: {}, r2: {}", *r1, *r2);
```

Borrowing: mutable references

- borrowed value may be modified
- acquired using the `&mut` keyword
- borrowed value must be mutable with `mut` (variables are immutable by default)

```
1  let mut s = String::from("hello");
2  s.push_str(" world");
3  String::push_str(&mut s, " world"); // equivalent
```

Borrowing: mutable references

- borrow-checker rule: the borrowed value is not used at all during lifetime

```
1  let mut v = vec![10, 11];
2  let vptr = &mut v[1]; // lifetime 'a starts
3  Vec::push(&mut v, 12); // lifetime 'b starts and ends
4  println!("{}", *vptr); // lifetime 'a ends
```

Borrowing: mutable references

- borrow-checker rule: the borrowed value is not used at all during lifetime

```
1  let mut v = vec![10, 11];
2  let vptr = &mut v[1]; // lifetime 'a starts
3  Vec::push(&mut v, 12); // lifetime 'b starts and ends
4  println!("{}", *vptr); // lifetime 'a ends
```

```
1  let mut v = vec![10, 11];
2  Vec::push(&mut v, 12); // lifetime 'b starts and ends
3  let vptr = &mut v[1]; // lifetime 'a starts
4  println!("{}", *vptr); // lifetime 'a ends
```

Borrowing: examples

```
1  let a;
2  {
3    let b: u32 = 42;
4    a = &b;
5  }
6  println!("{}", *a) // dereference a
```

- Will it compile?

Borrowing: examples

```
1  let a;
2  {
3    let b: u32 = 42;
4    a = &b;
5  }
6  println!("{}", *a) // dereference a
```

- Will it compile? No.
- Why not? Borrowed value 'b' does not live as long as the reference 'a'.
- Refactored version that compiles:

Borrowing: examples

```
1  let a;
2  {
3    let b: u32 = 42;
4    a = &b;
5  }
6  println!("{}", *a) // dereference a
```

- Will it compile? No.
- Why not? Borrowed value 'b' does not live as long as the reference 'a'.
- Refactored version that compiles:

```
1  let a;
2  let b: u32 = 42;
3  a = &b;
4  println!("{}", *a) // dereference a
```

Borrowing: examples

```
1  let mut s = String::from("hello world");
2  let r = &s;
3  s = String::from("hello world again");
4  println!("{}", *r);
```

- Will it compile?

Borrowing: examples

```
1  let mut s = String::from("hello world");
2  let r = &s;
3  s = String::from("hello world again");
4  println!("{}", *r);
```

- Will it compile? No.
- Why not? Violated rule: the borrowed value does not get mutated during lifetime
- Refactored version that compiles:

Borrowing: examples

```
1  let mut s = String::from("hello world");
2  let r = &s;
3  s = String::from("hello world again");
4  println!("{}", *r);
```

- Will it compile? No.
- Why not? Violated rule: the borrowed value does not get mutated during lifetime
- Refactored version that compiles:

```
1  let mut s = String::from("hello world");
2  s = String::from("hello world again");
3  let r = &s;
4  println!("{}", *r);
```

Another implementation: C++

- move semantics closely related to constructors and assignment operators (overloaded for classes)
- different reference types based on value categories:
  - L-value references: refer to named values → copy
  - R-value references: refer to temporary results of expression evaluations → move
- *overload resolution process* will select matching overload based on reference's value category

```
1  struct S {
2    S() {} // default constructor
3    S(const S&) {...} // copy constructor
4    S(S&&) {...} // move constructor
5    S operator=(const S&) {...} // copy assignment operator
6    S operator=(S&&) {... } // move assignment operator
7    // ... more functions and data fields etc.
8  }
```

Move semantics: C++ vs Rust

C++

```
1 S s{};
2 S s_copied{s}; // copy constructor
3 S s_moved{std::move(s)}; // move constructor
4 // problem: 's' can still be used
```

`std::move`:

- "move" a named value by casting it to an R-value reference to tell compiler to use move constructor
- no formal effect on the lifetime of the moved-from object → moved-from object can still be used, but compiler can't guarantee it is valid
- no move constructor or move assignment operator implementation specification

Move semantics: C++ vs Rust

C++

```cpp
S s{};
S s_copied{s}; // copy constructor
S s_moved{std::move(s)}; // move constructor
// problem: 's' can still be used
```

Rust

```rust
let s = S::default();
let s_copied = s.clone();
let s_moved = s; // 's' is moved
// compiler prevents use of 's'
```

Rust: safer move sematics through destructive moves enforced by the compiler

Conclusions and outlook

- *owners-as-dominators* and *owners-as-modifiers* models: different strictness on *where* and *how* references may be used
- Rust:
  - ownership-related rules that are strictly enforced at compile time
  - helpful compile-time error messages instead of crashes or UB at runtime
- will ownership types be incorporated into more new or existing programming languages in the future?
- maybe upcoming presentations on C++ enhancements (Carbon and cppfront) fix some of these issues?

Lifetimes: example

```
1  fn main() {
2    let s1 = String::from("short");
3    let s2 = String::from("longest");
4    let longest = longest(&s1, &s2);
5    println!("longest string: '{}'", longest);
6  }
7  fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
8    if s1.len() > s2.len() {
9      s1
10   } else {
11     s2
12   }
13 }
```

- longest function: return type contains a borrowed value, but unclear whether it is borrowed from 's1' or 's2'
- 's1' and 's2' have to live at least as long as lifetime 'a

Lifetime elision

- lifetime annotations: `&'a T` or `&'a mut T` , where `'a` is the lifetime of the reference
- compiler uses conventions to elide lifetimes $\rightarrow$ explicit annotations not always required
- Here are the three rules:
  - Each elided lifetime in a function's arguments becomes a distinct lifetime parameter.
  - If there is exactly one input lifetime, elided or not, that lifetime is assigned to all elided lifetimes in the return values of that function.
  - If there are multiple input lifetimes, but one of them is self or mut self, the lifetime of self is assigned to all elided output lifetimes.
- Otherwise, it is an error to elide an output lifetime.

Lifetime elision: examples

```
1  fn print(s: &str); // elided
2  fn print<'a>(s: &'a str); // expanded
3
4  fn debug(lvl: u32, s: &str); // elided
5  fn debug<'a>(lvl: u32, s: &'a str); // expanded
6
7  fn substr(s: &str, until: u32) -> &str; // elided
8  fn substr<'a>(s: &'a str, until: u32) -> &'a str; // expanded
9
10 fn get_str() -> &str; // ILLEGAL, no inputs
11
12 fn frob(s: &str, t: &str) -> &str; // ILLEGAL, two inputs
13 fn frob<'a, 'b>(s: &'a str, t: &'b str) -> &str; // Expanded:
       Output lifetime is ambiguous
```

Borrow-checker rules

program is valid if all of the below hold for all references:

1. the reference can only be used while its lifetime is ongoing
2. the original referent is not used at all (for mutable references) or does not get mutated (for immutable references) until the lifetime of the newly created reference has expired.

Formal lifetime rules

- Association rule:
  - $x : \&'aT \Rightarrow scope(\text{x}) \subseteq$ 'a
  - A lifetime is a superset of the scope of its associated reference.
- Reference rule:
  - $x : \&'aT = \&y \Rightarrow$ 'a $\subseteq scope(\text{y})$
  - A lifetime associated with a reference is a subset of the scope of the referent object.
- Assignment rule:
  - $x : \&'aS = y : \&'bT \Rightarrow$ 'a $\subseteq$ 'b
  - The lifetime associated with the assignee is a subset of the lifetime associated with the assigner.