Introduction to borrow checking in C++

Serbian C++ User Group meetup,
Belgrade, August 2023, Beograđanka

Goran Aranđelović
goran.arandjelovic@gmail.com
goran@cppserbia.com

Rust Ownership, References and Borrowing

Ownership

- Each value in Rust has an owner
- There can only be one owner at a time
- When the owner goes out of scope, the value will be dropped

References

- &T -> non-mutable
- &mut T -> mutable

References

```rust
fn main() {
  let s1 = String::from("hello");

  let len = calculate_length(&s1);

  println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
  s.len()
}
```

References

```rust
fn main() {
  let mut s = String::from("hello");

  change(&mut s);
}

fn change(some_string: &mut String) {
  some_string.push_str(", world");
}
```

References

- At any given time, you can have either one mutable reference or any number of immutable references
- References must always be valid

Borrowing

- Action of creating a reference is called *borrowing*

Borrowing

```rust
fn main() {
  let s = String::from("hello");

  change(&s);
}

fn change(some_string: &String) {
  some_string.push_str(", world");
}
```

Borrowing

```rust
fn main() {
  let mut s = String::from("hello");

  change(&mut s);
}

fn change(some_string: &mut String) {
  some_string.push_str(", world");
}
```

Borrowing

```rust
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s; // error

println!("{}, {}", r1, r2);
```

Borrowing

```rust
let mut s = String::from("hello");

let r1 = &s; // ok
let r2 = &s; // ok
let r3 = &mut s; // error

println!("{}, {}, and {}", r1, r2, r3);
```

Data race

A data race happens when these three behaviors occur:

- Two or more pointers access the same data at the same time
- At least one of the pointers is being used to write to the data
- There’s no mechanism being used to synchronize access to the data

Remember References?

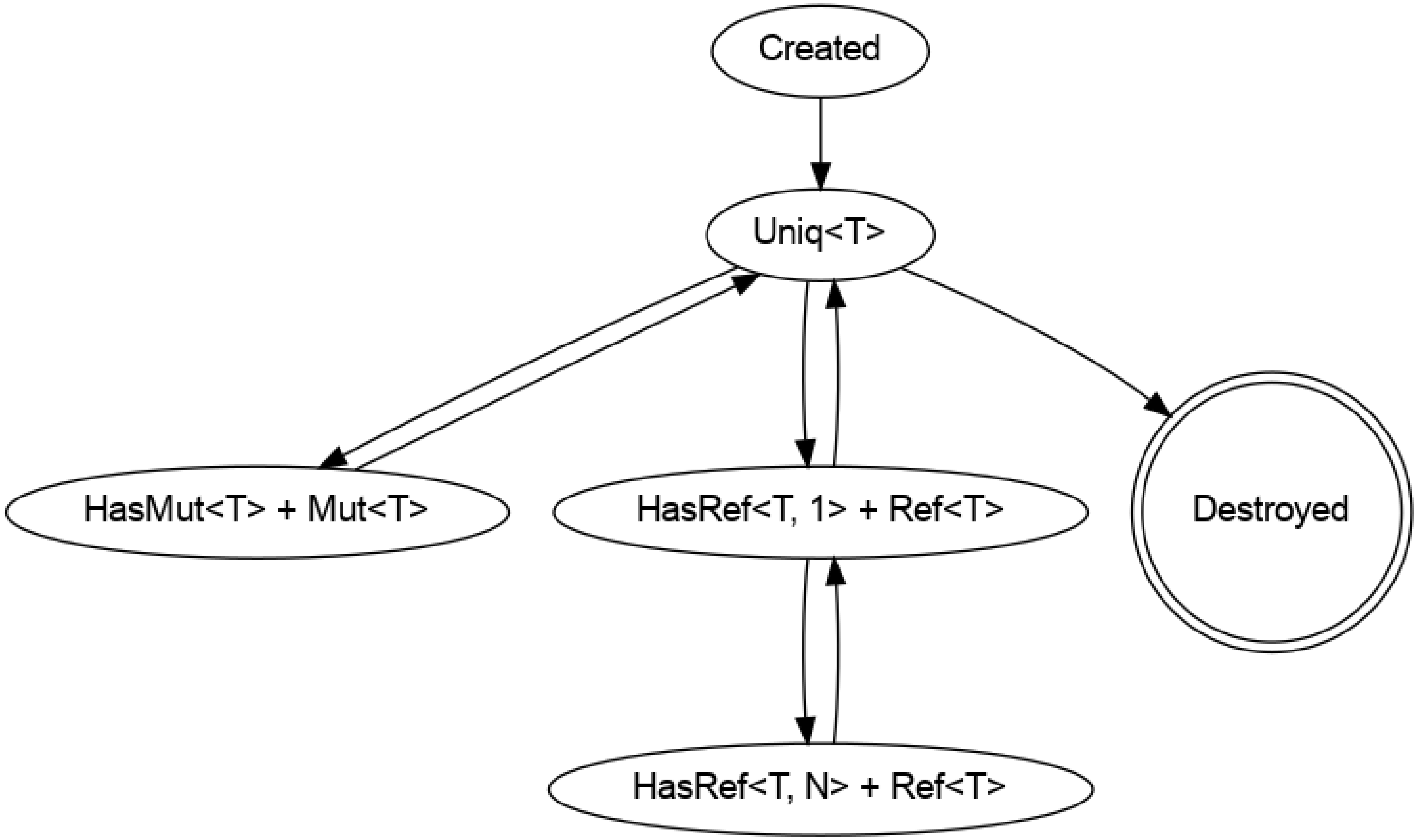
- At any given time, you can have either one mutable reference or any number of immutable references (no data race)
- References must always be valid (no use-after-free)

Borrow checking in C++

Borrow checking in C++

- Lifetime safety - wg21.link/p1179
- Google's Chromium team analyzed if it's possible to implement borrow checker in C++
- Various independent implementations of (runtime) borrow checking

Borrow checking in C++ — State transition



Author: palmer@chromium.org

Borrow checking in C++ — Corresponding types

Define each state as a type:

```
class Uniq<T> {};
class HasMut<T> {};
class HasRef<T, unsigned> {};
```

Borrow checking in C++ — Corresponding types

References require an explicit lifetime, so they also are defined as types:

```
class MutRef<T> {};
class Ref<T> {};
```

Borrow checking in C++ — Corresponding types

State transitions free functions from a uniquely owned state to the corresponding references:

```
std::pair<HasMut<T>, MutRef<T>> mut(Uniq<T>);
std::pair<HasRef<T, 1>, Ref<T>> ref(Uniq<T>);
```

Borrow checking in C++ — Corresponding types

State transitions free functions back to unique ownership:

```
Uniq<T> consume(HasMut<T>, MutRef<T>);
Uniq<T> consume(HasRef<T, 1>, Ref<T>);
```

Ignoring HasRef<T, N> where N > 1 for simplicity.

Borrow checking in C++ — The failures

- Destructors
- Non-destructive moves
- Left-over states

Borrow checking in C++ — The failures
Destructors

Object must always be returned to the Uniq<T> state/type before it is destroyed. To do otherwise violates the guarantees of the borrow checker. **To do that, HasMut<T> and HasRef<T> can not be (publicly) destructible.**

- If they are destructible but **do not destroy** the underlying T, then we can leak the object beyond the lifetime of the references and unique ownership.
- If they are destructible and **do destroy** the underlying T, then we destroy T while there is an outstanding reference to it, which creates a UAF scenario.

By ensuring the types HasMut<T> and HasRef<T> are not (publicly) destructible, we would ensure that they are converted back to Uniq<T> eventually, or their usage would fail to compile.

Borrow checking in C++ — The failures
Non-destructive moves

- C++ passes ownership though moves, however C++ requires that the destructor of the moved-from object is still run when the object leaves scope
- Rust moves objects destructively. When ownership (always of a uniquely owned object) is transferred, the destructor on the moved-from object would not run, the contents of the object are mem-copied into the destination object and the moved-from object is considered to no longer exist to the compiler
- Each state transition in C++ borrow checker implementation would leave behind a moved-from object representing the old state, which must be destroyed. Because the moved-from object lives outside of the state transition functions, **it requires a public destructor**. This contradicts borrow checker's desired guarantees

Borrow checking in C++ — The failures
Left-over states

- Coming back from HasRef<T> to Uniq<T> creates new (different) Uniq<T> object
- What about original Uniq<T>? It's a left-over state and we must be sure that nobody will use it for some other state transition (which would unfortunately require runtime check)
- Therefore, going from Uniq<T> to HasRef<T> **must destroy** Uniq<T> state
- If we choose not to destroy it and leave it assignable, we generate another problem: Some other object could be returned into that state, because Uniq<T> is bound to a particular type T, not the object itself

Borrow checking in C++ — The failures
Reference lifetime problem

- Rust compiler guarantees references lifetimes. If we pass/borrow mutable reference to a function, Rust ensures that the reference will be returned back.

- In order to mimic the same lifetime rules for references, C++ function that receives MutRef<T> must also return it to the caller. That means putting additional burden on the API

Borrow checking in C++ — Runtime

- Runtime borrow checking is possible
- There are a few good prototypes out there (e.g. https://github.com/shuaimu/borrow-cpp/)
- Some compilers and tool-chains already support UAF (use-after-free) analysis to some extent
- Using a runtime implementation + static analysis can help us prevent the most common errors

References (not the Rust ones)

- Lifetime safety - wg21.link/p1179
- Borrowing Trouble: The Difficulties Of A C++ Borrow-Checker
- Destructive move | N4034
- Sean Parent - About move
- Rust - References and Borrowing
- Bjørn Reese - Almost Affine
- Example of stateful metaprogramming

Thank you