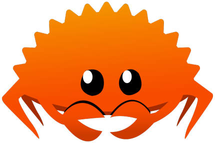
# Rust’s advanced type system


Lukas Prokop


June 8, 2020


RustGraz community

# Prologue

## How is `assert_eq!` implemented? Does it typecheck at compile-time?

# macros

```
macro_rules! assert_eq {
  ($left:expr, $right:expr) => ({
    match (&$left, &$right) {
      (left_val, right_val) => {
        if !(*left_val == *right_val) {
          // The reborrows below are intentional.
          // Without them, the stack slot for the
          // borrow is initialized even before the
          // values are compared, leading to a
          // noticeable slow down.
          panic!(r#"assertion failed: `(left == right)`
left: `{:?}`,
right: `{:?}`"#, &*left_val, &*right_val)
        }
      }
    }
  });
```

```
  ($left:expr, $right:expr,) => ({
    $crate::assert_eq!($left, $right)
  });
  ($left:expr, $right:expr, $($arg:tt)+) => ({
    match (&($left), &($right)) {
      (left_val, right_val) => {
        if !(*left_val == *right_val) {
          // The reborrows below are intentional.
          // Without them, the stack slot for the
          // borrow is initialized even before the
          // values are compared, leading to a
          // noticeable slow down.
          panic!(r#"assertion failed: `(left == right)`
left: `{:?}`,
right: `{:?}`: {}"#, &*left_val, &*right_val,
                  $crate::format_args!($($arg)+))
        }
      }
    }
  });
}
```

# Dialogue

# Recap: traits

- Semantically like a contract. Methods and constants.
- No subtyping, no inheritance, inspired by Haskell typeclasses.
- Trait can be implemented iff trait or type is local (trait coherence)
- implementation in `impl Trait for Type` block

```
use std::os::unix::net;
use std::fs;

trait Sendable {
    fn to_socket(&self, s: net::UnixStream);
    fn to_file(&self, l: fs::File);
}
```

# Recap: generics

- Replace a type by a *type argument*
- Actual type will be inserted upon instantiation
- Implemented with monomorphisation
- Trait bounds to require implementation of a trait

```
fn send<T: Sendable>(sender: T, req: Request) {
    if req.header.dest() == Resources::FILE {
      sender.to_file(req.log_file);
    } else {
      sender.to_socket(req.socket);
    }
}
```

`Trait1` requires implementation of `Trait2`

# Require implementation of another trait

- If `Trait1` is implemented, then `Trait2` must be implemented
- `trait Trait1: Trait2`
- Concept that comes closest to OOP subtyping, but no inheritance

```
trait JSONToFile: JSONSerializable {
  fn json_to_file(&self, s: fs::File)
    -> io::Result<()>;
}
```

# Require trait implementation (example)

```rust
use std::{io,fs};
use std::io::{BufWriter,Write};
use std::fs::File;

trait JSONSerializable {
  fn to_json(&self) -> Vec<u8>;
}

trait JSONToFile: JSONSerializable {
  fn json_to_file(&self, s: fs::File)
    -> io::Result<()>;
}
```

# Require trait implementation (example)

```rust
struct MyType { val: u32 }

impl JSONSerializable for MyType {
  fn to_json(&self) -> Vec<u8> {
    let mut bytes = Vec::new();
    for v in br#"{"type":"int", "value":"#.iter() {
      bytes.push(*v);
    }
    for v in format!("{}", self.val).bytes() {
      bytes.push(v);
    }
    bytes.push(b'}');
    bytes
  }
}
```

# Require trait implementation (example)

```
impl JSONToFile for MyType {
  fn json_to_file(&self, fd: fs::File)
    -> io::Result<()>
  {
    let mut writer = BufWriter::new(fd);
    writer.write(self.to_json().as_slice())?;
    Ok(())
  }
}
```

# Require trait implementation (example)

```rust
fn write_json<T: JSONToFile>
  (w: T, filepath: &str) -> io::Result<()>
{
  let fd = File::create(filepath)?;
  w.json_to_file(fd)
}

fn main() -> io::Result<()> {
  write_json(
    MyType { val: 42u32 },
    "example.json"
  )
}
```

# Which operators are overwritable?

# Non-overloadable operators

There is a defined set of operator traits. The following operators cannot be overwritten:

- `?` for error handling
- `||` as *lazy boolean or*
- `&&` as *lazy boolean and*
- `=` as *assignment operator*
- `&v` `&&v` `&&&v` … to get a reference
- `&mut v` `&&mut v` … to get a mutable reference

## n-ary references

```
fn print_number(s: &&u32) {
  println!("{}", **s);  // prints “42”
}

fn main() {
  print_number(&&42);
}
```

```
fn print_number(s: &&u32) {
  println!("{}", s);  // w/o “**”
}

fn main() {
  print_number(&&42);
}
```

Does it compile?

```rust
fn print_number(s: &&u32) {
  println!("{}", s);  // w/o “**”
}

fn main() {
  print_number(&&42);
}
```

Does it compile? Yes, auto-dereferencing.

```c
#include <stdio.h>
#include <stdint.h>

void print_number(uint32_t **s) {
  printf("%u", **s);
}
int main() {
  print_number(&&42);
  return 0;
}
```

Does it compile?

```c
#include <stdio.h>
#include <stdint.h>

void print_number(uint32_t **s) {
  printf("%u", **s);
}
int main() {
  print_number(&&42);
  return 0;
}
```

Does it compile? No.

```
main.c:8:18: error: expected identifier
  print_number(&&42);
                 ^
1 error generated.
```

Let's consider one level less. The error message becomes more explicit.

```c
#include <stdio.h>
#include <stdint.h>

void print_number(uint32_t *s) {
  printf("%u", *s);
}
int main() {
  print_number(&42);
  return 0;
}
```

Does it compile? No.

```
main.c:8:16: error: cannot take the address of
                    an rvalue of type 'int'
  print_number(&42);
               ^~~
1 error generated.
```

# n-ary references

```c
#include <stdio.h>
#include <stdint.h>

void print_number(uint32_t *s) {
  printf("%u", *s);
}
int main() {
  uint32_t val = 42;
  print_number(&val);
  return 0;
}
```

Does it compile?

```c
#include <stdio.h>
#include <stdint.h>

void print_number(uint32_t *s) {
  printf("%u", *s);
}
int main() {
  uint32_t val = 42;
  print_number(&val);
  return 0;
}
```

Does it compile? Yes.

**Summary:**

- In rust, you can take references to constant values.
- In C, you cannot (unless you assign them).

# Overloadable operators

This the defined (and exhaustive) set of operator traits.

- implement `std::ops::Neg` for `-`
- implement `std::ops::Not` for `!`
- implement `std::ops::Add` for `+`
- implement `std::ops::Sub` for `-`
- implement `std::ops::Mul` for `*`
- implement `std::ops::Div` for `/`
- implement `std::ops::Rem` for `%`

# Overloadable operators

- implement `std::ops::BitAnd` for `&`
- implement `std::ops::BitOr` for `|`
- implement `std::ops::BitXor` for `^`
- implement `std::ops::Shl` for `<<`
- implement `std::ops::Shr` for `>>`
- implement `std::cmp::PartialEq::eq` for `==`
- implement `std::cmp::PartialEq::ne` for `!=`
- implement `std::cmp::PartialOrd::gt` for `>`
- implement `std::cmp::PartialOrd::lt` for `<`
- implement `std::cmp::PartialOrd::ge` for `>=`
- implement `std::cmp::PartialOrd::le` for `<=`

# Overloadable operators

- implement `std::ops::AddAssign` for `+=`
- implement `std::ops::SubAssign` for `-=`
- implement `std::ops::MulAssign` for `*=`
- implement `std::ops::DivAssign` for `/=`
- implement `std::ops::RemAssign` for `%=`
- implement `std::ops::BitAndAssign` for `&=`
- implement `std::ops::BitOrAssign` for `|=`
- implement `std::ops::BitXorAssign` for `^=`
- implement `std::ops::ShlAssign` for `<<=`
- implement `std::ops::ShrAssign` for `>>=`

# Overloadable operators

- implement `std::ops::Index` for indexing `v[i]`
- implement `std::ops::IndexMut` for mutable indexing
- implement `std::ops::Deref` for `* ** ***` …
- implement `std::ops::DerefMut` for `*mut **mut ***mut` …

# Deref example

**Goal:**

- Let v be value 42 (wrapped by custom type).
- v is represented as `42`
- &v is represented as `(42)`
- &&v is represented as `((42))`
- &&&v is represented as `(((42))) …`

## Deref example

**Goal:**

- Let v be value 42 (wrapped by custom type).
- v is represented as 42
- &v is represented as (42)
- &&v is represented as ((42))
- &&&v is represented as (((42))) ...

**Problem:**

- We cannot overload &v, but *v
- Let v be (((((((42)))))))
- Let *v be ((((((42))))))
- Let **v be (((((42))))) ...

# Deref example

```rust
use std::fmt;
use std::ops::Deref;

struct Wrapped<T: fmt::Display> {
  value: T,
  depth: usize,
}

impl<T: fmt::Display> fmt::Display for Wrapped<T> {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f, "{}{}{}", "(".repeat(self.depth),
           self.value, ")".repeat(self.depth))
  }
}
```

# Deref example

```rust
impl<T: fmt::Display> Deref for Wrapped<T> {
  type Target = T;

  fn deref(&self) -> &Self::Target {
    &Wrapped { value: self.value, depth: self.depth - 1 }
  }
}

fn main() {
    let v = Wrapped{ value: 1, depth: 8 };
    println!("{}", v);
    println!("{}", ****v);
    println!("{}", ********v);
}
```

Does it compile?

# Deref example

```rust
impl<T: fmt::Display> Deref for Wrapped<T> {
  type Target = T;

  fn deref(&self) -> &Self::Target {
    &Wrapped { value: self.value, depth: self.depth - 1 }
  }
}

fn main() {
    let v = Wrapped{ value: 1, depth: 8 };
    println!("{}", v);
    println!("{}", ****v);
    println!("{}", ********v);
}
```

Does it compile? No.

# Deref example

```
error[E0614]: type `{integer}` cannot be dereferenced
  --> src/main.rs:27:22
   |
27 |     println!("{}", ****v);
   |                     ^^^

error[E0614]: type `{integer}` cannot be dereferenced
  --> src/main.rs:28:26
   |
28 |     println!("{}", ********v);
   |                         ^^^
```

Why?

```rust
struct Wrapped<T: fmt::Display> {
  value: T,
  depth: usize,
}

impl<T: fmt::Display> Deref for Wrapped<T> {
  type Target = T;

  fn deref(&self) -> &Self::Target {
    &Wrapped { value: self.value,
               depth: self.depth - 1 }
  }
}
```

# Deref example (hint for error)

```
struct Wrapped<T: fmt::Display> {
    value: T,
    depth: usize,
}

impl<T: fmt::Display> Deref for Wrapped<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &Wrapped { value: self.value,
                   depth: self.depth - 1 }
    }
}
```

T is **u32**, thus `Self::Target` as well.

# Deref example

**My learning process:**

- Where can we store the depth information?
- `fn deref(&self) -> &Self::Target` uses `&self` (c.f. `&mut self`). Does not permit mutation.
- We also cannot create new object, because where do we store it? (switch to heap objects like Rc would be possible)

# Deref example (fixed)

```rust
fn main() {
    let v1 = Wrapped{ value: 42, depth: 0 };
    let v2 = Wrapped{ value: &v1, depth: 1 };
    let v3 = Wrapped{ value: &v2, depth: 2 };
    let v4 = Wrapped{ value: &v3, depth: 3 };
    println!("{}", v1);  // "42"
    println!("{}", v2);  // "(42)"
    println!("{}", *v2); // "42"
}
```

# Deref example (fixed)

```rust
use std::fmt;
use std::ops::Deref;

struct Wrapped<T: fmt::Display> {
  value: T,
  depth: usize,
}

impl<T: fmt::Display> Deref for Wrapped<T> {
  type Target = T;

  fn deref(&self) -> &Self::Target {
    &Wrapped { value: self.value,
               depth: self.depth - 1 }
  }
}
```

# Deref example (fixed)

```rust
impl<T: fmt::Display> fmt::Display for Wrapped<T> {
    fn fmt(&self, f: &mut fmt::Formatter)
       -> fmt::Result {
        write!(f,
               "{}{}{}",
               "(".repeat(self.depth),
               self.value,
               ")".repeat(self.depth))
    }
}

impl<T: fmt::Display> Deref for Wrapped<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.value
    }
}
```

# Trait objects

# Method dispatch

**Cliff hanger, last time**: Can we take references to traits?

Let T be a trait. We call `T.method()`.
Where do we find the implementation of `method`?

- **static dispatch:** monomorphization, like C++ templates, preferred dispatch
- **dynamic dispatch:** trait objects

One application example for dynamic dispatch: What about a vector of objects implementing a trait; `Vec<Trait>`?

- **static dispatch:** wrap each possible type with **`enum`**, unextensible to external types
- **dynamic dispatch:** trait object

# Monomorphisation example

```rust
struct Wrapped { val: u32 }

trait Numeric {
  fn as_u32(&self) -> u32;
}

impl Numeric for Wrapped {
  fn as_u32(&self) -> u32 {
    self.val
  }
}
```

# Monomorphisation example

```rust
fn print_int<T: Numeric>(obj: T) {
    println!("{:x}", obj.as_u32());
}

fn main() {
    let v = Wrapped { val: 42 };
    print_int(v);  // "2a"
}
```

# Trial and error

Can we provide `&Wrapped` for `Numeric`?

```rust
fn print_int<T: Numeric>(obj: T) {
    println!("{:x}", obj.as_u32());
}

fn main() {
    let v = Wrapped { val: 42 };
    print_int(&v);
}
```

# Trial and error

```
error[E0277]: the trait bound `&Wrapped: Numeric` is not satisfied
  --> src/main.rs:20:15
   |
14 | fn print_int<T: Numeric>(obj: T) {
   |    ---------    ------- required by this bound in `print_int`
...
20 |     print_int(&v);
   |              -^
   |               |
   | the trait `Numeric` is not implemented for `&Wrapped`
   | help: consider removing the leading `&`-reference
   |
   = help: the following implementations were found:
            <Wrapped as Numeric>
```

# Trial and error

```
error[E0277]: the trait bound `&Wrapped: Numeric` is not satisfied
  --> src/main.rs:20:15
   |
14 | fn print_int<T: Numeric>(obj: T) {
   |    ---------    ------- required by this bound in `print_int`
...
20 |     print_int(&v);
   |               -^
   |               |
   | the trait `Numeric` is not implemented for `&Wrapped`
   | help: consider removing the leading `&`-reference
   |
   = help: the following implementations were found:
             <Wrapped as Numeric>
```

Apparently, we can implement `Numeric` for `&Wrapped` as well.

# Trial and error

```rust
struct Wrapped { val: u32 }
trait Numeric {
  fn as_u32(&self) -> u32;
}

impl Numeric for Wrapped {
  fn as_u32(&self) -> u32 { self.val }
}
impl Numeric for &Wrapped {
  fn as_u32(&self) -> u32 { (**self).val }
}

fn print_int<T: Numeric>(obj: T) {
  println!("{:x}", obj.as_u32());
}
fn main() {
  let v = Wrapped { val: 42 };
  print_int(&v); // "2a"
}
```

What about `&Wrapped` *and* `&Numeric`?

```
fn print_int<T: &Numeric>(obj: T) {
    println!("{:x}", obj.as_u32());
}

fn main() {
    let v = Wrapped { val: 42 };
    print_int(&v);
}
```

# Trial and error

```
error: expected one of `!`, `(`, `,`, `=`,
       `>`, `?`, `for`, lifetime, or path, found `&`
  --> src/main.rs:14:17
   |
14 | fn print_int<T: &Numeric>(obj: T) {
   |                 ^ expected one of 9 possible tokens

error: aborting due to previous error
```

**Recap:** Let's switch syntax. Can we use `Numeric` as type?

```
fn print_int(obj: Numeric) {
    println!("{:x}", obj.as_u32());
}

fn main() {
    let v = Wrapped { val: 42 };
    print_int(v);
}
```

**Recap:** Let's switch syntax. Can we use `Numeric` as type?

```
fn print_int(obj: Numeric) {
    println!("{:x}", obj.as_u32());
}

fn main() {
    let v = Wrapped { val: 42 };
    print_int(v);
}
```

No, we need to use the `impl` keyword!

**Recap:** use `impl`! This is the equivalent syntax to `fn print_int<T: Numeric>(obj: T)`.

```
fn print_int(obj: impl Numeric) {
    println!("{:x}", obj.as_u32());
}

fn main() {
    let v = Wrapped { val: 42 };
    print_int(v);
}
```

But it **does work** if we use `&Numeric` as type!

```rust
fn print_int(obj: &Numeric) {
    println!("{:x}", obj.as_u32());
}

fn main() {
    let v = Wrapped { val: 42 };
    print_int(&v);    // "2a"
}
```

… so, what is a trait object?

# Trait objects

**Idea:**

- We generate an separate object from an object maintaining pointers to the actual implementation of the methods specified in the trait (and *only those*)!
- Exactly like Golang's function call with argument of interface type
- Similar to C++'s vtable
- Runtime overhead, no inlining of function calls

**Syntax and history:**

- `dyn` keyword: `dyn Trait` is a type referring to any trait object implementing `Trait`
- Since Rust 1.27. Is `Foo` a struct or a trait?

```
Box<Foo>    became   Box<dyn Foo>
&Foo        became   &dyn Foo
&mut Foo    became   &mut dyn Foo
```

**Requirements** for traits to generate trait objects (“object safe”):

- All return types must not be Self.
- No generic type parameters.

Standard library's `Clone` trait is **not** object-safe:

```
pub trait Clone {
    fn clone(&self) -> Self;
}
```

Standard library's `Clone` trait is **not** object-safe:

```
pub trait Clone {
    fn clone(&self) -> Self;
}
```

Thus, `dyn Clone` is not permitted.

# Trait object example

```rust
trait Named {
    fn name(&self) -> String;
}

struct Student { name: String }
struct Teacher { name: String }
```

# Trait object example

```rust
impl Named for Student {
    fn name(&self) -> String {
        let mut s = String::from("student ");
        s.push_str(&self.name);
        s
    }
}

impl Named for Teacher {
    fn name(&self) -> String {
        let mut s = String::from("teacher ");
        s.push_str(&self.name);
        s
    }
}
```

# Trait object example

```
fn main() {
  let s1 = Student { name: String::from("Lukas") };
  let s2 = Student { name: String::from("Anita") };
  let t1 = Teacher { name: String::from("Sensei") };
  println!("{}\n{}\n{}", s1.name(), s2.name(), t1.name());
}
```

```
student Lukas
student Anita
teacher Sensei
```

# Trait object example

```rust
struct Container {
    elements: Vec<dyn Named>,
}
```

```
error[E0277]: the size for values of type
  `(dyn Named + 'static)` cannot be known
  at compilation time
  --> src/main.rs:25:5
   |
25 |     elements: Vec<dyn Named>,
   |     ^^^^^^^^^^^^^^^^^^^^^^^^
   |     doesn't have a size known at compile-time
   |
   = help: the trait `std::marker::Sized` is not implemented
          for `(dyn Named + 'static)`
   = note: required by `std::vec::Vec`
```

# Trait object example

```
struct Container {
  elements: Vec<Box<dyn Named>>,
}

fn main() {
  let c = Container {
    elements: vec![
      Box::new(s1), Box::new(s2),
       Box::new(t1)
  ]};

  println!("{}", c.elements[0].name());
  // "student Lukas"
}
```

# Trait object example

```
struct Container<T: Named> {
    elements: Vec<Box<T>>,
}

fn main() {
  let c = Container {
    elements: vec![
      Box::new(s1), Box::new(s2),
       Box::new(t1)
  ]};
  println!("{}", c.elements[0].name());
}
```

What's the difference?

# Trait object example

```
error[E0308]: mismatched types
  --> src/main.rs:35:61
   |
35 | elements: vec![Box::new(s1), Box::new(s2), Box::new(t1)]
   |   expected struct `Student`, found struct `Teacher` ^^

error: aborting due to previous error
```

# Trait object example

```
error[E0308]: mismatched types
  --> src/main.rs:35:61
   |
35 | elements: vec![Box::new(s1), Box::new(s2), Box::new(t1)]
   |   expected struct `Student`, found struct `Teacher` ^^

error: aborting due to previous error
```

# Trait object coercion

- Casts for trait objects are possible, but rarely necessary
- Use **as** keyword for coercion
- `&obj as &Trait`
- Allows to tests for object safety

```
let v = vec![1, 2, 3];
let o = &v as &Clone;
```

# The *type keyword* or *associated types*

# Associated types

```rust
trait Graph<N, E> {
    fn has_edge(&self, &N, &N) -> bool;
    fn edges(&self, &N) -> Vec<E>;
    // Etc.
}
```

A Graph generic over any node type and edge type.

```rust
fn distance<N, E, G: Graph<N, E>>(
  graph: &G, start: &N, end: &N
) -> u32 {
  // implementation of distance function
}
```

```
trait Graph {
  type N;
  type E;

  fn has_edge(&self, &Self::N, &Self::N) -> bool;
  fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

# Associated types

- Associated types are declared with the `type` keyword within a trait
- Binds a type to some instance of a graph
- In our example: Graph is a trait with two associated types $N$ and $E$

# Associated types

- Associated types are declared with the `type` keyword within a trait
- Binds a type to some instance of a graph
- In our example: Graph is a trait with two associated types *N* and *E*

# Associated types

```rust
struct Node;
struct Edge;
struct MyGraph;

impl Graph for MyGraph {
  type N = Node;
  type E = Edge;

  fn has_edge(&self, n1: &Node, n2: &Node)
     -> bool
  {
    true
  }

  fn edges(&self, n: &Node)
    -> Vec<Edge>
  {
    Vec::new()
  }
}
```

# Associated types

## Associated types and coercion into a trait object:

```rust
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph>;
```

```
error: the value of the associated type `E`
       (from the trait `main::Graph`) must
       be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
24:44 error: the value of the associated type `N`
             (from the trait `main::Graph`) must
             be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

# Associated types

**Solution:** explicit assignment

```
let graph = MyGraph;
let obj = Box::new(graph) as
  Box<Graph<N=Node, E=Edge>>;
```

## Add trait via stdlib implementation

```
#[lang = "add"]
#[stable(feature = "rust1", since = "1.0.0")]
#[rustc_on_unimplemented(
  on(all(_Self = "{integer}", Rhs = "{float}"), message = "cannot add
  on(all(_Self = "{float}", Rhs = "{integer}"), message = "cannot add
  message = "cannot add `{Rhs}` to `{Self}`",
  label = "no implementation for `{Self} + {Rhs}`"
)]
#[doc(alias = "+")]
pub trait Add<Rhs = Self> {
  /// The resulting type after applying the `+` operator.
  #[stable(feature = "rust1", since = "1.0.0")]
  type Output;

  /// Performs the `+` operation.
  #[must_use]
  #[stable(feature = "rust1", since = "1.0.0")]
  fn add(self, rhs: Rhs) -> Self::Output;
}
```

# TypeId

`TypeId` is a crate in stdlib that allows you to reason about types (obviously at compile time) in a limited manner. One example:

```rust
use std::any::{Any, TypeId};

fn is_string<T: ?Sized + Any>(_s: &T) -> bool {
    TypeId::of::<String>() == TypeId::of::<T>()
}

assert_eq!(is_string(&0), false);
assert_eq!(is_string(&"cookie monster".to_string()), true);
```

# Epilogue

# Quiz

**What is an associated type?**

**What is dynamic dispatching?**

**When do you use the dyn keyword?**

# Quiz

**What is an associated type?**

A type local to a trait

**What is dynamic dispatching?**

**When do you use the dyn keyword?**

# Quiz

**What is an associated type?**

A type local to a trait

**What is dynamic dispatching?**

An object is generated containing only pointers to the trait method implementations

**When do you use the dyn keyword?**

# Quiz

**What is an associated type?**

A type local to a trait

**What is dynamic dispatching?**

An object is generated containing only pointers to the trait method implementations

**When do you use the dyn keyword?**

To refer to a trait object

# Quiz on macros (skipped last time)

**How can you syntactically recognize macros?**

**Which two kinds of macros exist?**

**What kind of procedural macros exist?**

**Define macro hygiene**

**Which repetition specifiers exist in macros?**

# Quiz on macros (skipped last time)

**How can you syntactically recognize macros?**

`macro!()`

**Which two kinds of macros exist?**

**What kind of procedural macros exist?**

**Define macro hygiene**

**Which repetition specifiers exist in macros?**

# Quiz on macros (skipped last time)

**How can you syntactically recognize macros?**

`macro!()`

**Which two kinds of macros exist?**

declarative & procedural

**What kind of procedural macros exist?**

**Define macro hygiene**

**Which repetition specifiers exist in macros?**

# Quiz on macros (skipped last time)

**How can you syntactically recognize macros?**

`macro!()`

**Which two kinds of macros exist?**

declarative & procedural

**What kind of procedural macros exist?**

derive macros, attribute-like macros, function-like macros

**Define macro hygiene**

**Which repetition specifiers exist in macros?**

# Quiz on macros (skipped last time)

**How can you syntactically recognize macros?**

`macro!()`

**Which two kinds of macros exist?**

declarative & procedural

**What kind of procedural macros exist?**

derive macros, attribute-like macros, function-like macros

**Define macro hygiene**

Local scope does not get polluted by variables introduced in macro

**Which repetition specifiers exist in macros?**

# Quiz on macros (skipped last time)

**How can you syntactically recognize macros?**

`macro!()`

**Which two kinds of macros exist?**

declarative & procedural

**What kind of procedural macros exist?**

derive macros, attribute-like macros, function-like macros

**Define macro hygiene**

Local scope does not get polluted by variables introduced in macro

**Which repetition specifiers exist in macros?**

0–infinity: `*`

0–1: `?`

1–infinity: `+`

# Next time

**Covid19 disclaimer:** Once, more than 15 people are allowed to meet and the majority is fine with it, we are going to schedule an offline meeting. In the following, we are going to hold a *Hacker Jeopardy* (finally).

| | |
|---|---|
| Next meetup | Wed, 2020/06/24 |
| Topic | Lifetimes, anonymous functions and modularization |

# Next time

For the ambitious ones:

- Rust lifetimes are inspired by Cyclone's memory regions
- I will talk about Cyclone next time
- Cyclone: A safe dialect of C (2002)
- Cyclone homepage
- Recommendation: read the Cyclone paper before next time

# Thank you!