INTRO TO RUST LANG MACROS

Benjamin Owad, David Rudo, and Connor Tsui

Macros Spotted!

We've seen a few macros so far...

- `println!()`
- `assert!()` and `assert_eq!()`
- `panic!()` and `todo!()`
- `vec![]`

Hiding in Plain Sight

A few things that we haven't explicitly called macros are actually macros in disguise!

- `#[derive(Debug)]`
- `#[cfg(test)]` and `#[test]`

Macros!

- 3 Levels of Metaprogramming
- Declarative Macros
- The `vec![]` Macro
- Procedural Macros

Metaprogramming

Metaprogramming is essentially writing code that writes code.

More precisely, it is a programming technique where computer programs treat other programs as their data.

- This can mean generating a program from a program (code generation)
- Or it could mean a program modifying itself
- *Today we will focus on the latter example*

Level 1: Metaprogramming in C

C's metaprogramming is mostly restricted to C macros.

- C compilers like `gcc` and `clang` come with a *C preprocessor*
- However you define the macro is how it is expanded
- Simple source code / text expansion

C Macros

You use the `define` directive to create object-like macros in C. Every invocation of the macro that is defined "expands" to its definition.

```
#define PAGE_SIZE 4096
void *page = malloc(PAGE_SIZE);
//       -> = malloc(4096);
```

```
#define NUMBERS 1, \
                2, \
                3
int x[] = { NUMBERS };
     →  = { 1, 2, 3 };
```

```
#define MAX(X, Y)  ((X) > (Y) ? (X) : (Y))
x = MAX(a, b);          →  x = ((a) > (b) ? (a) : (b));
y = MAX(1, 2);          →  y = ((1) > (2) ? (1) : (2));
z = MAX(a + 28, *p);    →  z = ((a + 28) > (*p) ? (a + 28) : (*p));
```

What's Wrong With This Picture?

```
#define TWICE(x) 2*x

TWICE(3)    →    2*3
```

- What happens when you write `TWICE(1 + x)`?
  - Expands to `2 * 1 + x`
  - We probably wanted `2 * (1 + x)`...

`#define`

- The `define` directive can be powerful, but can also be easily misused
- Generally* C macros cannot go beyond what a programmer could write themselves manually
- C macros are unaware* of types, scope, and even variable names
- *`#define` is literally just string manipulation*

Level 2: C++ Metaprogramming

C++ is a much more expressive language than C, so how does it approach metaprogramming differently?

- C++ is mostly a superset of C, so it still has `#define` 😔
- C++ has a feature called *Templates* that allows for metaprogramming

C++ Templates

C++ Templates offer a solution to writing generic code:

```cpp
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

max(10, 20);
max(10.11, 20.22);
```

- C++ templates will generate a version of the templated function for every type that needs to use the function
- Seem familiar?
    - Rust Monomorphization!

C++ Error Messages

What happens if we're not so careful with the types?

```
max(10, 20.0);
```

```
test.cpp: In function 'int main()':
test.cpp:12:8: error: no matching function for call to 'max(int, double)'
   12 |     max(10, 20.2);
      |     ~~~^~~~~~~~~~
test.cpp:5:3: note: candidate: 'template<class T> T max(T, T)'
    5 | T max(T a, T b) {
      |   ^~~
test.cpp:5:3: note:   template argument deduction/substitution failed:
test.cpp:12:8: note:   deduced conflicting types for parameter 'T' ('int' and 'double')
   12 |     max(10, 20.2);
      |     ~~~^~~~~~~~~~
```

- Not terrible...

C++ ERROR Messages

What about this?

```
#include <vector>
#include <algorithm>

int main() {
    int a;
    std::vector<std::vector<int>> v;
    std::find(v.begin(), v.end(), a);
}
```

- *Brace for impact*

```
In file included from /usr/local/include/c++/12.2.0/bits/stl_algobase.h:71,
                 from /usr/local/include/c++/12.2.0/vector:60,
                 from /tmp/aa5vrlEi5A.cpp:1:
/usr/local/include/c++/12.2.0/bits/predefined_ops.h: In instantiation of 'bool
__gnu_cxx::__ops::_Iter_equals_val<_Value>::operator()(_Iterator) [with _Iterator =
__gnu_cxx::__normal_iterator<std::vector<int>*, std::vector<std::vector<int> > >; _Value = const int]':
/usr/local/include/c++/12.2.0/bits/stl_algobase.h:2067:14:   required from '_RandomAccessIterator std::__find_if(_RandomAccessIterator, _RandomAccessIterator, _Predicate, random_access_iterator_tag) [with _RandomAccessIterator =
__gnu_cxx::__normal_iterator<vector<int>*, vector<vector<int> > >; _Predicate = __gnu_cxx::__ops::_Iter_equals_val<const int>]'
/usr/local/include/c++/12.2.0/bits/stl_algobase.h:2112:23:   required from '_Iterator std::__find_if(_Iterator, _Iterator, _Predicate) [with _Iterator = __gnu_cxx::__normal_iterator<vector<int>*, vector<vector<int> > >; _Predicate = __gnu_cxx::__ops::_Iter_equals_val<const int>]'
/usr/local/include/c++/12.2.0/bits/stl_algo.h:3851:28:   required from '_IIter std::find(_IIter, _IIter, const _Tp&) [with _IIter = __gnu_cxx::__normal_iterator<vector<int>*, vector<vector<int> > >; _Tp = int]'
/tmp/aa5vrlEi5A.cpp:7:14:   required from here
/usr/local/include/c++/12.2.0/bits/predefined_ops.h:270:24: error: no match for 'operator==' (operand types are 'std::vector<int>' and 'const int')
  270 |         { return *__it == _M_value; }
      |                  ~~~~~~^~~~~~~~~~~
In file included from /usr/local/include/c++/12.2.0/bits/stl_algobase.h:67:
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:1213:5: note: candidate: 'template<class _IteratorL, class _IteratorR, class _Container> bool __gnu_cxx::operator==(const __normal_iterator<_IteratorL, _Container>&, const __normal_iterator<_IteratorR, _Container>&)'
 1213 |     operator==(const __normal_iterator<_IteratorL, _Container>& __lhs,
      |     ^~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:1213:5: note:   template argument deduction/substitution failed:
/usr/local/include/c++/12.2.0/bits/predefined_ops.h:270:24: note:   'std::vector<int>' is not derived from 'const __gnu_cxx::__normal_iterator<_IteratorL, _Container>'
  270 |         { return *__it == _M_value; }
      |                  ~~~~~~^~~~~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:1221:5: note: candidate: 'template<class _Iterator, class _Container> bool __gnu_cxx::operator==(const __normal_iterator<_Iterator, _Container>&, const __normal_iterator<_Iterator, _Container>&)'
 1221 |     operator==(const __normal_iterator<_Iterator, _Container>& __lhs,
      |     ^~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:1221:5: note:   template argument deduction/substitution failed:
/usr/local/include/c++/12.2.0/bits/predefined_ops.h:270:24: note:   'std::vector<int>' is not derived from 'const __gnu_cxx::__normal_iterator<_Iterator, _Container>'
  270 |         { return *__it == _M_value; }
      |                  ~~~~~~^~~~~~~~~~~
In file included from /usr/local/include/c++/12.2.0/x86_64-linux-gnu/bits/c++allocator.h:33,
                 from /usr/local/include/c++/12.2.0/bits/allocator.h:46,
                 from /usr/local/include/c++/12.2.0/vector:61:
/usr/local/include/c++/12.2.0/bits/new_allocator.h:196:9: note: candidate: 'template<class _Up> bool std::operator==(const __new_allocator<int>&, const __new_allocator<_Tp>&)'
  196 |         operator==(const __new_allocator&, const __new_allocator<_Up>&)
      |         ^~~~~~~~
/usr/local/include/c++/12.2.0/bits/new_allocator.h:196:9: note:   template argument deduction/substitution failed:
/usr/local/include/c++/12.2.0/bits/predefined_ops.h:270:24: note:   mismatched types 'const std::__new_allocator<_Tp>' and 'const int'
  270 |         { return *__it == _M_value; }
      |                  ~~~~~~^~~~~~~~~~~
In file included from /usr/local/include/c++/12.2.0/bits/stl_algobase.h:64:
/usr/local/include/c++/12.2.0/bits/stl_pair.h:640:5: note: candidate: 'template<class _T1, class _T2> constexpr bool std::operator==(const pair<_T1, _T2>&, const pair<_T1, _T2>&)'
  640 |     operator==(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
      |     ^~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_pair.h:640:5: note:   template argument deduction/substitution failed:
/usr/local/include/c++/12.2.0/bits/predefined_ops.h:270:24: note:   'std::vector<int>' is not derived from 'const std::pair<_T1, _T2>'
  270 |         { return *__it == _M_value; }
      |                  ~~~~~~^~~~~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:444:5: note: candidate: 'template<class _Iterator> constexpr bool std::operator==(const reverse_iterator<_Iterator>&, const reverse_iterator<_Iterator>&)'
  444 |     operator==(const reverse_iterator<_Iterator>& __x,
      |     ^~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:444:5: note:   template argument deduction/substitution failed:
/usr/local/include/c++/12.2.0/bits/predefined_ops.h:270:24: note:   'std::vector<int>' is not derived from 'const std::reverse_iterator<_Iterator>'
  270 |         { return *__it == _M_value; }
      |                  ~~~~~~^~~~~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:489:5: note: candidate: 'template<class _IteratorL, class _IteratorR> constexpr bool std::operator==(const reverse_iterator<_Iterator>&, const reverse_iterator<_IteratorR>&)'
  489 |     operator==(const reverse_iterator<_IteratorL>& __x,
      |     ^~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:489:5: note:   template argument deduction/substitution failed:
/usr/local/include/c++/12.2.0/bits/predefined_ops.h:270:24: note:   'std::vector<int>' is not derived from 'const std::reverse_iterator<_Iterator>'
  270 |         { return *__it == _M_value; }
      |                  ~~~~~~^~~~~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:1656:5: note: candidate: 'template<class _IteratorL, class _IteratorR> constexpr bool std::operator==(const move_iterator<_IteratorL>&, const move_iterator<_IteratorR>&)'
 1656 |     operator==(const move_iterator<_IteratorL>& __x,
      |     ^~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:1656:5: note:   template argument deduction/substitution failed:
/usr/local/include/c++/12.2.0/bits/predefined_ops.h:270:24: note:   'std::vector<int>' is not derived from 'const std::move_iterator<_IteratorL>'
  270 |         { return *__it == _M_value; }
      |                  ~~~~~~^~~~~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:1726:5: note: candidate: 'template<class _Iterator> constexpr bool std::operator==(const move_iterator<_IteratorL>&, const move_iterator<_IteratorL>&)'
 1726 |     operator==(const move_iterator<_Iterator>& __x,
      |     ^~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_iterator.h:1726:5: note:   template argument deduction/substitution failed:
/usr/local/include/c++/12.2.0/bits/predefined_ops.h:270:24: note:   'std::vector<int>' is not derived from 'const std::move_iterator<_IteratorL>'
  270 |         { return *__it == _M_value; }
      |                  ~~~~~~^~~~~~~~~~~
/usr/local/include/c++/12.2.0/bits/allocator.h:219:5: note: candidate: 'template<class _T1, class _T2> bool std::operator==(const allocator<_Tp1>&, const allocator<_T2>&)'
  219 |     operator==(const allocator<_T1>&, const allocator<_T2>&)
      |     ^~~~~~~~
/usr/local/include/c++/12.2.0/bits/allocator.h:219:5: note:   template argument deduction/substitution failed:
/usr/local/include/c++/12.2.0/bits/predefined_ops.h:270:24: note:   'std::vector<int>' is not derived from 'const std::allocator<_Tp1>'
  270 |         { return *__it == _M_value; }
      |                  ~~~~~~^~~~~~~~~~~
In file included from /usr/local/include/c++/12.2.0/vector:64:
/usr/local/include/c++/12.2.0/bits/stl_vector.h:2035:5: note: candidate: 'template<class _Tp, class _Alloc> bool std::operator==(const vector<_Tp, _Alloc>&, const vector<_Tp, _Alloc>&)'
 2035 |     operator==(const vector<_Tp, _Alloc>& __x, const vector<_Tp, _Alloc>& __y)
      |     ^~~~~~~~
/usr/local/include/c++/12.2.0/bits/stl_vector.h:2035:5: note:   template argument deduction/substitution failed:
```

C++ Templates

- C++ Templates are more powerful than Rust Generics (for now)
    - *C++ Templates are even Turing-complete!*
- Powerful, but not exactly the most ergonomic
- Very similar to Rust Generics in nature, can be different in practice

Level 3: Rust Metaprogramming

We've already seen metaprogramming in Rust through Generics.

```rust
use std::cmp::PartialOrd;

fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

- Rust generates monomorphized versions of this function for each type that we need it for!

`#[derive(...)]`

We have also seen metaprogramming through the `derive` attribute, which can generate trait implementations for us.

```rust
#[derive(Debug)]
struct Student {
    andrew_id: String,
    attendance: Vec<bool>,
    grade: u8,
    stress_level: u64,
}
```

- Same idea, the `Debug` trait is implemented for us at compile time

`println!(...)`

Finally, we've seen function-like macros:

```
println!("hello");
println!("hello {}", name);
let v = vec![1, 2, 3];
assert_eq!(2 + 2, 4, "Math broken?");
```

- The main difference between these function-like macros and normal functions is the variadic parameters (on top of other things)

Rust Macros

- Macros are expanded before the compiler interprets the meaning of code
- This means that Rust macros can:
  - Implement a trait on some type
  - Statically evaluate code
  - *Modify the Abstract Syntax Tree*
- Macros are expanded into the abstract syntax tree at compile time, whereas functions are called at runtime

Macros Everywhere?

Macros are strictly more powerful than functions because of their ability to execute during compile time.

So why not just use macros everywhere?

- Macro definitions are far more complex than function definitions
  - *You're writing Rust code that writes Rust code!*
- Macros are much more difficult to read, understand, and maintain

2 Types of Macros

Rust has 2 main types of macros.

- Declarative Macros
- Procedural Macros (*3 subcategories*)
  - Custom `#[derive]` macros
  - Attribute-like macros
  - Function-like macros operating on tokens

Declarative Macros

Declarative macros are the most widely used form of macros in Rust.

- At a high level, declarative macros allow you to write something similar to a `match` expression
- The only difference is that instead of `match`ing expressions, we `match` Rust source code

`macro_rules!`

We use the `macro_rules!` construct to define declarative macros.

```
// This is a simple macro named `say_hello`
macro_rules! say_hello {
    // `()` indicates that the macro takes no argument
    () => {
        // The macro will expand into the contents of this block
        println!("Hello!")
    };
}

fn main() {
    // This call will expand into `println!("Hello")`
    say_hello!()
}
```

Matching Identifiers

We can match literal tokens in `macro_rules!`:

```
macro_rules! create_function {
    ($func_name:ident) => {
        fn $func_name() {
            // The `stringify!` macro converts an `ident` into a string
            println!("You called {:?}()", stringify!($func_name));
        }
    };
}

// Create functions named `foo` and `bar` with the `create_function` macro
create_function!(foo);
create_function!(bar);
```

- The `ident` designator is used for variable / function names
- The `create_function` macro takes 1 argument of designator `ident`
- It will create a function named `$func_name`

Matching Identifiers (`ident`)

```
macro_rules! create_function {
    ($func_name:ident) => {
        fn $func_name() {
            println!("You called {:?}()", stringify!($func_name));
        }
    };
}

create_function!(foo);
create_function!(bar);

fn main() {
    foo();
    bar();
}
```

```
You called "foo"()
You called "bar"()
```

Matching Expressions

We can also match valid Rust expressions:

```
macro_rules! print_result {
    ($expression:expr) => {
        // `stringify!` will convert the expression *as it is* into a string.
        println!("{:?} = {:?}", stringify!($expression), $expression);
    };
}
```

- The `expr` designator is used for expressions
- The `print_result` macro takes 1 argument of designator `expr`
- It will print the expression and as well as the evaluated result

Matching Expressions (`expr`)

```rust
macro_rules! print_result {
    ($expression:expr) => {
        println!("{:?} = {:?}", stringify!($expression), $expression);
    };
}

fn main() {
    print_result!(1u32 + 1);

    // Recall that blocks are expressions too!
    print_result!({
        let x = 1u32;
        x * x + 2 * x - 1
    });
}
```

```
"1u32 + 1" = 2
"{ let x = 1u32; x * x + 2 * x - 1 }" = 2
```

Variadic Arguments

Macros can match to any number of code patterns:

```
macro_rules! test {
    // Any template can be used!
    ($left:expr; and $right:expr) => {
        println!("{:?} and {:?} is {:?}", stringify!($left), stringify!($right),
                 $left && $right)
    };
//   ^ each arm must end with a semicolon.

    ($left:expr; or $right:expr) => {
        println!("{:?} or {:?} is {:?}", stringify!($left), stringify!($right),
                 $left || $right)
    };
}
```

- `test!` will compare `$left` and `$right` in different ways depending on how you invoke it

Variadic Arguments

```
macro_rules! test {
    ($left:expr; and $right:expr) => {
        println!("{:?} and {:?} is {:?}", stringify!($left), stringify!($right),
                 $left && $right)
    };
    ($left:expr; or $right:expr) => {
        println!("{:?} or {:?} is {:?}", stringify!($left), stringify!($right),
                 $left || $right)
    };
}

fn main() {
    test!(1i32 + 1 == 2i32; and 2i32 * 2 == 4i32);
    test!(true; or false);
}
```

```
"1i32 + 1 == 2i32" and "2i32 * 2 == 4i32" is true
"true" or "false" is true
```

Repeating Arguments

If we want to match to *any* number of repeated arguments, we can use `+` and `*`.

```
macro_rules! find_min {
    // Base case of one argument
    ($x:expr) => ($x);

    // $x, followed by at least one $y
    ( $x:expr, $( $y:expr ),+ ) => (
        // Call `find_min!` on the tail
        std::cmp::min($x, find_min!($($y),+))
    )
}
```

- `find_min!` will calculate the minimum of any number of arguments
- `+` indicates an expression can repeat at least once
- `*` indicates an expression can repeat zero or more times

Repeating Arguments (`+` and `*`)

```
macro_rules! find_min {
    ($x:expr) => ($x);

    ( $x:expr, $( $y:expr ),+ ) => (
        std::cmp::min($x, find_min!($($y),+))
    )
}

fn main() {
    println!("{}", find_min!(1));
    println!("{}", find_min!(1 + 2, 2));
    println!("{}", find_min!(5, 2 * 3, 4));
}
```

```
1
2
4
```

The `vec![]` Macro

Recall that we can use the `vec![]` macro to easily create vectors.

```rust
let v: Vec<u32> = vec![1, 2, 3];
```

- Let's try to implement this ourselves!

`vec![]` Behavior

```rust
#[test]
fn test_empty() {
    let v: Vec<u32> = vec![];
    assert!(v.is_empty());
}

#[test]
fn single() {
    let x: Vec<u32> = vec![42];
    assert!(!x.is_empty());
    assert_eq!(x.len(), 1);
    assert_eq!(x[0], 42);
}

#[test]
fn double() {
    let x: Vec<u32> = vec![42, 43];
    assert!(!x.is_empty());
    assert_eq!(x.len(), 2);
    assert_eq!(x[0], 42);
    assert_eq!(x[1], 43);
}
```

The Empty Vector

To define the empty vector, we can just use `Vec::new()`.

```
macro_rules! vec {
    () => {
        Vec::new()
    };
}

#[test]
fn test_empty() {
    let v: Vec<u32> = vec![];
    assert!(v.is_empty());
}
```

One Element

If we want a vector with one element, let's just push it onto the empty vector!

```
macro_rules! vec {
    () => {
        Vec::new()
    };

    ($element:expr) => {
        let mut v = Vec::new();
        v.push($element);
        v
    };
}
```

Expansion

```
    // <-- snip -->

    ($element:expr) => {
        let mut v = Vec::new();
        v.push($element);
        v
    };
```

We get a compiler error when we try to compile this:

```
error: expected expression, found `let` statement
  --> src/lib.rs:7:9
   |
7  |         let mut v = Vec::new();
   |         ^^^
...
20 |     let x: Vec<u32> = vec![42];
   |                       --------- in this macro invocation
   |
```

Statements

The issue was that we were expanding to 3 statements, but what we really wanted to do was expand to an expression.

```
let v: Vec<u32> = vec![1];

// Would expand to:
let v: Vec<u32> = let mut v = Vec::new(); // ...
```

- Let's just make sure we return an expression instead of a series of statements!

Expressions

It is typical to see double brackets in macro definitions for this very reason.

```rust
macro_rules! vec {
    () => { Vec::new() };

    ($element:expr) => {{
        let mut v = Vec::new();
        v.push($element);
        v
    }};
}

#[test]
fn single() {
    let x: Vec<u32> = vec![42];
    assert!(!x.is_empty());
    assert_eq!(x.len(), 1);
    assert_eq!(x[0], 42);
}
```

Commas in `vec!`

Let's implement comma-separated elements in `vec![]`.

```
macro_rules! vec {
    // <-- snip -->

    ($e1:expr, $e2:expr) => {{
        let mut v = Vec::new();
        v.push($e1);
        v.push($e2);
        v
    }};
}

let x: Vec<u32> = vec![42, 43];
```

- Might as well just hard code 2 elements separated by a comma...
- Is this a good idea?

Repeating Elements in `vec!`

Instead of copying and pasting `v.push($ex)` however many times, we can use the `+` symbol to indicate 1 or more repeated arguments.

```
macro_rules! vec {
    // <-- snip -->

    ( $( $element:expr ),+ ) => {{
        let mut v = Vec::new();
        $(
            v.push($element);
        )+
        v
    }};
}
```

Cleanup

Instead of having separate branches for all of these cases, we can combine them into a single `*` repeating branch!

```
macro_rules! vec {
    ($($element:expr),*) => {{
        let mut v = Vec::new();
        $(v.push($element);)*
        v
    }};
}
```

- *The real standard library* `vec![]` *has a few more features, including allocating memory up front and supporting array notation instantiation*

Captures

Here is a list of captures you can match on in `macro_rules!`:

- `item`: an item, like a function, `struct`, module, etc
- `block`: a block (surrounded by `{}`)
- `stmt`: a statement
- `pat`: a pattern
- `expr`: an expression
- `ty`: a type
- `ident`: an identifier
- `path`: a module path (`::std::mem::replace`, `transmute::<_, int>`)
- `meta`: a meta item, the things that go inside `#[...]` and `#![...]` attributes
- `tt`: a single token tree

Recurrences

If we had more time, we would go through this recurrence example:

```
fn main() {
    let fib  = recurrence![a[n]: u64 = 0, 1;    a[n-1] + a[n-2]         ];
    let trib = recurrence![a[n]: u64 = 0, 0, 1; a[n-1] + a[n-2] + a[n-3]];

    for e in fib.take(20) {
        print!("{} ", e)
    }
    for e in trib.take(20) {
        print!("{} ", e)
    }
}
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
0 0 1 1 2 4 7 13 24 44 81 149 274 504 927 1705 3136 5768 10609 19513
```

- Source: The Little book of Rust Macros

Procedural Macros

The second form of macros is the *procedural* macro.

- Procedural macros take code as input, operate on that code, and produce code as output
- The 3 types of procedural macros are:
  - Custom `#[derive]` macros
  - Attribute-like macros
  - Function-like macros operating on tokens

`TokenStream`

At a high level, procedural macros take on this form:

```rust
use proc_macro;

#[some_attribute]
pub fn some_name(input: TokenStream) -> TokenStream {
    // Do something with the TokenStream
}
```

Custom `#[derive]` Macros

Here's a very high-level example of creating a custom `derive` macro:

```rust
use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}
```

Custom `#[derive]` Macros

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!", stringify!(#name));
            }
        }
    };
    gen.into()
}
```

Using a Custom `#[derive]` Macro

Instead of manually writing out this code...

```rust
use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro! My name is Pancakes!");
    }
}

fn main() {
    Pancakes::hello_macro();
}
```

Using a Custom `#[derive]` Macro

We can now use our new `#[derive(HelloMacro)]` Macro!

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

Attribute-like Macros

Attribute-like macros allow you to create new attributes.

- Different from just the `derive` attribute
- Can be applied to items other than structs and enums

Attribute-like Macros

Suppose you have an attribute named `route` that annotates functions when using a web app framework:

```
#[route(GET, "/")]
fn index() {
    // <-- snip -->
}
```

The signature of the `route` definition function would look like this:

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
    // <-- snip -->
}
```

- You can imagine different behavior for different HTTP methods and paths!

Function-like Procedural Macros

Function-like Procedural macros define macros that look like function calls.

- Different from declarative macros because they operate on a `TokenStream`
- Recall that `macro_rules!` is just a pattern matching machine

Function-like Procedural Macros

A good example of a function-like procedural macro is a theoretical `sql!` macro:

```
let sql = sql!(SELECT * FROM posts WHERE id = 1);

#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
    // <-- snip -->
}
```

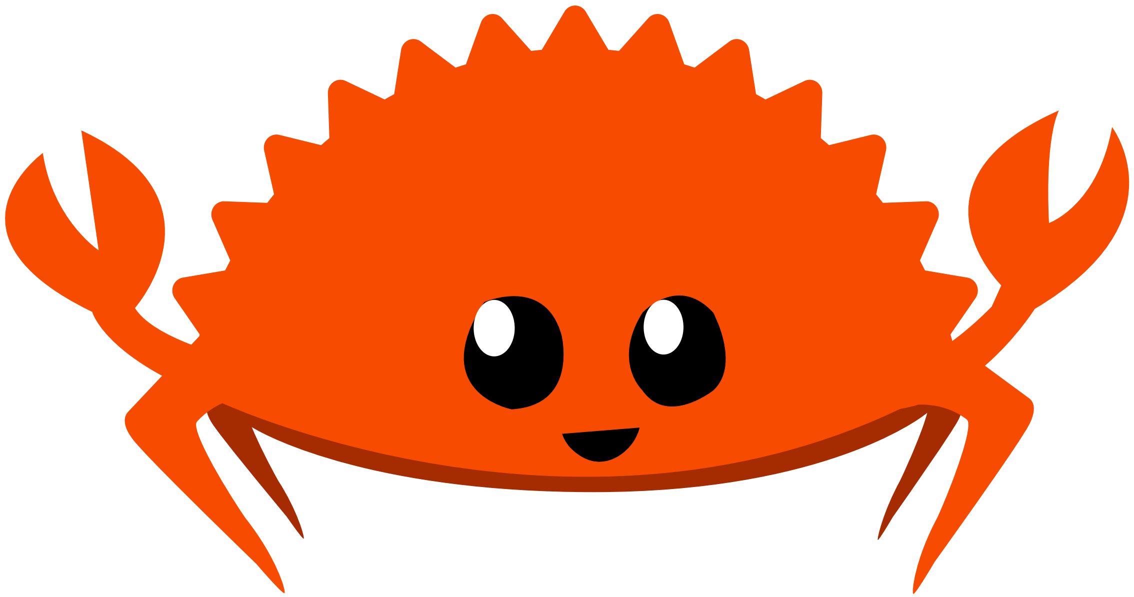
- This `sql` macro would parse the query inside to check the syntax
- *You would never do SQL bindings like this in practice*

Summary: Macros

- Rust Macros are *incredibly* powerful
- Both similar and very different to normal Rust code
- Very niche use cases and not super easy to write

The End!

We've reached the end of our prepared content!



Reflection

We can reflect on what we've learned this semester from our course description.

Students will be able to:

- Read, write, and reason about Rust code
- Know common Rust types and collections
- Understand the Ownership system and Borrow Checker
- Use advanced Rust features like iterators, closures, and lifetimes
- Make use of advanced patterns like parallelism, concurrency, and `unsafe`

The Cycle Begins Again

Looking back at our very first lecture, we asked a question...

Why Rust?

Why Rust?

If there are only a few things to take away from this course:

- Rust is Different
- Rust is Modern
- Rust is Fast
- Rust is Safe
- Rust is not a silver bullet
- Rust is *Important*

Thank You!

- Thanks for sticking with us this semester!