Lecture 5: Metaprogramming

Alexander Stanovoy

April 5, 2022

alex.stanovoy@gmail.com

In this lecture

- Closures
- Intro to Metaprogramming
- Declarative macros
- Procedural macros
- Macros from standard library

Closures

Closures

You've already seen closures in homeworks and lectures:

```
let x = 4;
let equal_to_x = |z| z == x;
let y = 4;
assert!(equal_to_x(y));
```

Question: What's the difference between the closures and the functions?

Closures

You've already seen closures in homeworks and lectures:

```
let x = 4;
let equal_to_x = |z| z == x;
let y = 4;
assert!(equal_to_x(y));
```

Question: What's the difference between the closures and the functions?

A closure is an *anonymous function* that can directly *use variables from the scope* in which it is defined.

Closures

Unlike functions, closures infer input and output types since it's more convenient most of the time.

```rust
let option = Some(2);

let x = 3;
// explicit types:
let new: Option<i32> = option.map(|val: i32| -> i32 {
    val + x
});
println!("{:?}", new);  // Some(5)

let y = 10;
// inferred:
let new2 = option.map(|val| val * y);
println!("{:?}", new2);  // Some(20)
```

Let's try to duplicate `Option::map` functionality with handcrafted function.

```
fn map<X, Y>(option: Option<X>, transform: ...) -> Option<Y> {
    match option {
        Some(x) => Some(transform(x)),
        None => None,
    }
}
```

We need to fill in the `...` with something that transforms an `X` into a `Y`. What it will be?

Closures and traits

We want `transform` to be the object that is callable. In Rust, when we want to abstract over some property, we use traits!

```
fn map<X, Y, T>(option: Option<X>, transform: T) -> Option<Y>
    where T: /* the trait */ { ... }
```

Let's design it.

Closures and traits

We want `transform` to be the object that is callable. In Rust, when we want to abstract over some property, we use traits!

```
fn map<X, Y, T>(option: Option<X>, transform: T) -> Option<Y>
    where T: /* the trait */ { ... }
```

Let's design it.

- Idea: compiler generated structure that implements some trait.

Closures and traits

We want `transform` to be the object that is callable. In Rust, when we want to abstract over some property, we use traits!

```rust
fn map<X, Y, T>(option: Option<X>, transform: T) -> Option<Y>
    where T: /* the trait */ { ... }
```

Let's design it.

- Idea: compiler generated structure that implements some trait.
- Our trait will have only one function.

Closures and traits

We want `transform` to be the object that is callable. In Rust, when we want to abstract over some property, we use traits!

```rust
fn map<X, Y, T>(option: Option<X>, transform: T) -> Option<Y>
    where T: /* the trait */ { ... }
```

Let's design it.

- Idea: compiler generated structure that implements some trait.
- Our trait will have only one function.
- We'll use tuple as input type since we don't have variadics in Rust (and we don't actually need them, at least in this case).

Closures and traits

```
trait Transform<Input> {
    type Output;
    fn transform(/* self */, input: Input) -> Self::Output;
}
```

Question: Do we need `self`, `&mut self` or `&self` here?

Closures and traits

```
trait Transform<Input> {
    type Output;
    fn transform(/* self */, input: Input) -> Self::Output;
}
```

Question: Do we need `self`, `&mut self` or `&self` here?

Since the transformation should be able to incorporate arbitrary information beyond what is contained in `Input`. Without any `self` argument, the method would look like `fn transform(input: Input) -> Self::Output` and the operation could only depend on `Input` and global variables.

Closures and traits

Question: What do we need exactly: `self`, `&mut self` or `&self`?

Closures and traits

Question: What do we need exactly: `self`, `&mut self` or `&self`?

	User
`self`	Can only call method once
`&mut self`	Can call many times, only with unique access
`&self`	Can call many times, with no restrictions

Question: What do we need exactly: `self`, `&mut self` or `&self`?

	User
`self`	Can only call method once
`&mut self`	Can call many times, only with unique access
`&self`	Can call many times, with no restrictions

We usually want to chose the highest row of the table that still allows the consumers to do what they need to do.

Closures and traits

Let's start with `self`. In summary, our `map` and its trait look like:

```rust
trait Transform<Input> {
    type Output;
    fn transform(self, input: Input) -> Self::Output;
}

fn map<X, Y, T>(option: Option<X>, transform: T) -> Option<Y>
    where T: Transform<X, Output = Y>
{
    match option {
        Some(x) => Some(transform.transform(x)),
        None => None,
    }
}
```

```rust
let option = Some(2);
let x = 3;
let new: Option<i32> = map(option, Adder { x: x });
println!("{:?}", new); // Some(5)
```

Fn, FnMut, FnOnce

Rust uses `Fn`, `FnMut`, `FnOnce` traits to unify functions and closures, similar to what we've invented.

```
pub trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}

pub trait FnMut<Args>: FnOnce<Args> {
    fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait Fn<Args>: FnMut<Args> {
    fn call(&self, args: Args) -> Self::Output;
}
```

Fn, FnMut, FnOnce

Rust uses `Fn`, `FnMut`, `FnOnce` traits to unify functions and closures, similar to what we've invented.

```
pub trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}

pub trait FnMut<Args>: FnOnce<Args> {
    fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait Fn<Args>: FnMut<Args> {
    fn call(&self, args: Args) -> Self::Output;
}
```

Look carefuly as `self`. Every `FnMut` closure can implement `FnOnce` exactly the same way! Same applies to `Fn` and `FnMut`.

Fn, FnMut, FnOnce

The real `map` looks like this:

```rust
impl<T> Option<T> {
    pub fn map<U, F>(self, f: F) -> Option<U>
    where
        F: FnOnce(T) -> U,
    {
        match self {
            Some(x) => Some(f(x)),
            None => None
        }
    }
}
```

`FnOnce(T) -> U` is another name for our `Transform<X, Output = Y>` bound, and `f(x)` for `transform.transform(x)`.

Returning and accepting closures

Since the closure is a compiler-generated type, it's **non denotable**, i.e you cannot write it exact type.

```
fn return_closure() -> impl Fn() {
    || println!("hello world!")
}
```

`Fn`, `FnMut`, `FnOnce` are traits, and we can benefit from trait objects here too!

```
let c1 = || {
    println!("calculating...");
    42 * 2 - 22
};
let c2 = || 42;
let vec: Vec<&dyn Fn() -> i32> = vec![&c1, &c2];
```

Fn, FnMut, FnOnce

Basically any funtions also implement these traits!

```rust
fn cast(x: i32) -> i64 {
    (x + 1) as i64
}

fn func(f: impl FnOnce(i32) -> i64) {
    println!("f(42) = {}", f(42));
}

fn main() {
    func(cast)
}
```

So, like everything in Rust, operator `()` is defined by traits (Although you can overload it only in nightly currently).

fn

There's also *function pointers* in Rust. It's not a trait, it's an actual *type* that refers to the code, not data. Unlike closures, they cannot capture the environment.

```
fn add_one(x: usize) -> usize { x + 1 }

let ptr: fn(usize) -> usize = add_one;
assert_eq!(ptr(5), 6);

let clos: fn(usize) -> usize = |x| x + 5;
assert_eq!(clos(5), 10);

// error: mismatched types
// let y = 2;
// let clos: fn(usize) -> usize = |x| y + x + 5;
// assert_eq!(clos(5), 10);
```

Closures: capturing

Let's find out how Rust closures decide how to capture the variables.

```
struct T { ... }

fn by_value(_: T) {}
fn by_mut(_: &mut T) {}
fn by_ref(_: &T) {}
```

Closures: capturing

```rust
let x: T = ...;
let mut y: T = ...;
let mut z: T = ...;

let closure = || {
    by_ref(&x);
    by_ref(&y);
    by_ref(&z);

    // Forces `y` and `z` to be at least
    // captured by `&mut` reference
    by_mut(&mut y);
    by_mut(&mut z);

    // Forces `z` to be captured by value
    by_value(z);
};
```

Closures: capturing

This is how closure environment will look like:

```
struct Environment<'x, 'y> {
    x: &'x T,
    y: &'y mut T,
    z: T
}

/* impl of FnOnce for Environment */

let closure = Environment {
    x: &x,
    y: &mut y,
    z: z,
};
```

Closures: capturing

Since this closure implements `FnOnce`, it cannot be called twice:

```
// Ok
closure();
// error: moved due to previous call
// closure();
```

Closures: capturing

What if you need to move out a closure from the scope? In this case, you need to move all the variables even if it's enougth to have a shared reference.

```
// Returns a function that adds a fixed number
// to the argument. Reminds of Higher Order Functions
// from functional programming!
fn make_adder(x: i32) -> impl Fn(i32) -> i32 {
    |y| x + y
}
fn main() {
    let f = make_adder(3);
    println!("{}", f(1));  // 4
    println!("{}", f(10));  // 13
}
```

Closures: capturing

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:2:9
  |
2 |     |y| x + y
  |     --- ^ borrowed value does not live long enough
  |     |
  |     value captured here
3 | }
  |  -
  |  |
  |  `x` dropped here while still borrowed
  |  borrow later used here
```

Let's use **move** keyword to tell Rust we need to capture by value:

```rust
fn make_adder(x: i32) -> impl Fn(i32) -> i32 {
    // Compiles just fine!
    move |y| x + y
}

fn main() {
    let f = make_adder(3);
    println!("{}", f(1));  // 4
    println!("{}", f(10));  // 13
}
```

Closures: capturing

Going back to previous example, the closure with `move` keyword will capture all variables by value:

```rust
let closure = move || {
    by_ref(&x);
    by_ref(&y);
    by_ref(&z);

    // Forces `y` and `z` to be at least
    // captured by `&mut` reference
    by_mut(&mut y);
    by_mut(&mut z);

    // Forces `z` to be captured by value
    by_value(z);
};
```

Closures: capturing

```rust
struct Environment {
    x: T,
    y: T,
    z: T,
}
```

In Rust, there's no fine-grained capture lists like in C++11. *But do we need it*? In practice, we don't (at least lecturer doesn't know good examples).

Closure type

Every closure have **distinct type**. This implies that in this example `id0`, `id1`, `id2` and `id3` have **different types**.

```
fn id0(x: u64) -> u64 { x }
fn id1(x: u64) -> u64 { x }
fn main() {
    let id2 = || 1;
    let id3 = || 1;
}
```

And this code won't compile:

```
fn make_closure(n: u64) -> impl Fn() -> u64 {
    move || n
}

vec![make_closure(1), make_closure(2)];
vec![(|| 1), (|| 1)];  // Error: mismatched types
```

Closures and optimizations

- We create a structure for the closure, do some moves… It must be expensive!

Closures and optimizations

- We create a structure for the closure, do some moves... It must be expensive!
- Actually, the compiler knows a lot about our code and optimizes it with ease. Most closure calls are inlined and in binary is the same as code without closure.

Closures and optimizations

- We create a structure for the closure, do some moves... It must be expensive!
- Actually, the compiler knows a lot about our code and optimizes it with ease. Most closure calls are inlined and in binary is the same as code without closure.
- Zero cost abstraction!

Lambdas and C++

- In C++, lambdas work in pretty the same way as closures.

Lambdas and C++

- In C++, lambdas work in pretty the same way as closures.
- The bad thing is `std::function`. It is a general-purpose polymorphic function wrapper, type erasure object, the only way in language to store lambda.

- In C++, lambdas work in pretty the same way as closures.
- The bad thing is `std::function`. It is a general-purpose polymorphic function wrapper, type erasure object, the only way in language to store lambda.
- Inside, we create a heap allocation, and it's not efficient (Anyway, there's SOO that optimizes it, but not every time).

- In C++, lambdas work in pretty the same way as closures.
- The bad thing is `std::function`. It is a general-purpose polymorphic function wrapper, type erasure object, the only way in language to store lambda.
- Inside, we create a heap allocation, and it's not efficient (Anyway, there's SOO that optimizes it, but not every time).
- At the same time, Rust's closures are located in stack, which means great performance comparing to C++.

Lambdas and C++

- In C++, lambdas work in pretty the same way as closures.
- The bad thing is `std::function`. It is a general-purpose polymorphic function wrapper, type erasure object, the only way in language to store lambda.
- Inside, we create a heap allocation, and it’s not efficient (Anyway, there’s SOO that optimizes it, but not every time).
- At the same time, Rust’s closures are located in stack, which means great performance comparing to C++.
- If you want `std::function`, use `Box<dyn Fn(...) -> ...>` (or other needed trait).

Intro to Metaprogramming

Metaprogramming

First of all, what is Metaprogramming?

First of all, what is Metaprogramming?

Metaprogramming is a programming technique that means *“programs that manipulate programs”*. It has quite broad meaning, but we don’t need to understand them all. Instead, we’ll focus on definition *“code that generates code”*.

When do we need metaprogramming?

When do we need metaprogramming?

- Optimizations: precalculation of any values, tables...

When do we need metaprogramming?

- Optimizations: precalculation of any values, tables…
- Embedding data into source code of the program.

Metaprogramming

When do we need metaprogramming?

- Optimizations: precalculation of any values, tables...
- Embedding data into source code of the program.
- Generating boilerplate code.

Metaprogramming

When do we need metaprogramming?

- Optimizations: precalculation of any values, tables...
- Embedding data into source code of the program.
- Generating boilerplate code.
- Implementing DSLs (Domain-specific language).
- And much more!

C++ has two main approaches to metaprogramming: template metaprogramming and `#define` directive.

C++ has two main approaches to metaprogramming: template metaprogramming and `#define` directive.

- Templates in C++ are turing-complete, and that implies you can write any program in the *mathematical* sence. Template metaprogramming can be seen as some jedi technique, hard to learn and extremely rarely used in practice.

C++ and Metaprogramming

C++ has two main approaches to metaprogramming: template metaprogramming and `#define` directive.

- Templates in C++ are turing-complete, and that implies you can write any program in the *mathematical* sence. Template metaprogramming can be seen as some jedi technique, hard to learn and extremely rarely used in practice.
- Old `#define` preprocessor directive was inhereted from C. It just replaces all of the appearances of identifier, so it's not very powerful tool. Since preprocessor runs before lexical analysis, it means it can arbitrary break you code compilation.

Rust and metaprogramming

In Rust, we have one main approach to metaprogramming: macros.

[1] (Russian) How to write FizzBuzz on the interview
[2] Rust's Type System is Turing-Complete

Rust and metaprogramming

In Rust, we have one main approach to metaprogramming: **macros**.

- Macros are **extremely** powerful. They enable you to read arbitrary tokens and translate them into arbitrary Rust code!

[1] (Russian) How to write FizzBuzz on the interview
[2] Rust's Type System is Turing-Complete

Rust and metaprogramming

In Rust, we have one main approach to metaprogramming: **macros**.

- Macros are **extremely** powerful. They enable you to read arbitrary tokens and translate them into arbitrary Rust code!
- Moreover, macros can read files, send requests to databases and so on, since it is Rust code that generates Rust code.

Actually, you can even do some magic with traits and objects and emulate C++ template metaprogramming, since traits are turing-complete too! Hovewer, prefer macros instead, they're way easier.[1][2]

[1] (Russian) How to write FizzBuzz on the interview
[2] Rust's Type System is Turing-Complete

Declarative macros

Declarative macros

Before we continue, let's check the `json` macro from crate serde!

```
let value = json!({
    "code": 200,
    "success": true,
    "payload": {
        "features": [
            "serde",
            "json"
        ]
    }
});
```

Declarative macros

Let's start with the easiest example. Suppose we want to create a vector from the list of arguments. If we do this by hand, the result will be as follows:

```rust
let mut a = Vec::new();
a.push(1);
a.push(1);
a.push(1);
```

But we already know there's a macro that is doing the same!

```
let a = vec![1; 3];
let a = vec![1, 1, 1];
```

We'll implement it and call `create_vec`.

`macro_rules!`

```rust
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        vec.resize($count, $value);
        vec
    }};
    [$($value:expr),*] => {{
        let mut vec = ::std::vec::Vec::new();
        $(vec.push($value);)*
        vec
    }};
}
```

`macro_rules!` is also a macro, special for compiler. It means “I’m defining a macro”, pretty the same as `fn` keyword.

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        /* some code */
    }};
    [$($value:expr),*] => {{
        /* some code */
    }};
}
```

Our `create_vec` macro works quite like the same as matching an `enum`. We check what arguments are given and match them sequentally with patterns on the left side, called *matchers*. Then, we insert code generated on the right side, called *transcribers*.

```rust
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        /* code */
    }};
    [$($value:expr),*] => {{
        /* code */
    }};
}
```

Arguments (called *meta-variables*) of macro are specified by $ symbol. After the colon, we put the type of the argument.

`macro_rules!`

Possible types of meta-variables are:

- `expr` - an expression.
- `stmt` - a statement.
- `ty` - a type.
- `ident` - an identifier (or keyword).
- `block` - block in `{}`.
- `tt` - token tree `()`, `[]` or `{}`.
- `literal` - literal.
- And much more!

```
macro_rules! other_create_vec {
    /* variant */
    [$($value:expr),*] => {{
        /* code */
    }};
}
```

To repeat some pattern multiple times, there is "special" syntaxes like `$( )*`, `$( )+` and `$( )?`. They all mean "sequence of patterns inside" and called *repetitions*.

- `*` means "zero or more times".
- `+` means "one or more times".
- `?` means "zero or one time".

```
macro_rules! other_create_vec {
    /* variant */
    [$($value:expr),*] => {{
        /* code */
    }};
}
```

To repeat some pattern multiple times, there is “special” syntaxes like `$()*`, `$()+` and `$()?`. They all mean “sequence of patterns inside” and called *repetitions*.

- `*` means “zero or more times”.
- `+` means “one or more times”.
- `?` means “zero or one time”.

After `$()` and before repetition qualifier, you can write a separator (or don’t choose a separator). Rust allows you not to write it at the end, just like, for instance, in definition of `enum`.

Just to understand it a bit better, let's write some strange looking macro.

```
macro_rules! example {
    {$($value1:expr),* => $($value2:expr),*} => {};
}

// Note that we've used '{}' on definition but
// in this line 'example' is used with []!
example![1, 2, 3 => 3, 2];
```

Just to understand it a bit better, let's write some strange looking macro.

```
macro_rules! example {
    {($($value1:expr),* => $($value2:expr),*)} => {};
}

example![(1, 2, 3 => 3, 2)];
```

macro_rules!

Just to understand it a bit better, let's write an example macro.

```rust
macro_rules! example {
    {$(($($value1:expr),* => $($value2:expr),*)),*} => {{}};
}

example![(1, 2, 3 => 3 + 3, 2), ("hello" => [1, 2, 3], (22, 42))];
```

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        vec.resize($count, $value);
        vec
    }};
    // Note this example!
    [] => { ::std::vec::Vec::new() };
    [$($value:expr),*] => {{
        let mut vec = ::std::vec::Vec::new();
        $(vec.push($value);)*
        vec
    }};
}
```

We use first {} to mark the beginning of macro block with code. The second is needed to show that we'll use multiple statements inside the block, and this two are actually inserted in the place of macro invocation since Rust wants only one

macro_rules!

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        /* code */
    }};
    /* variant */
}
```

We have to use a fully specified type for a `Vec` because our macro is actually placed in the place where it's used.

macro_rules!

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        /* code */
    }};
    /* variant */
}
```

We have to use a fully specified type for a `Vec` because our macro is actually placed in the place where it's used.

- The `::` means "from list of imported crates". If `std` is not imported, we'll receive an error.

macro_rules!

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        /* code */
    }};
    /* variant */
}
```

We have to use a fully specified type for a `Vec` because our macro is actually placed in the place where it's used.

- The `::` means "from list of imported crates". If `std` is not imported, we'll receive an error.
- We can also use `$crate` pattern: for instance, if our crate is called `example`, the name `$crate::module::Example` will change to `example::module::Example`.

macro_rules!

```rust
macro_rules! new_s {
    [] => { nested::S {} };
}
pub mod example {
    pub mod nested {
        pub struct S;
    }
    pub fn test() -> nested::S {
        new_s!()
    }
}

assert_eq!(vec![1; 3], create_vec![1; 3]);
assert_eq!(vec![1, 1, 1], create_vec![1, 1, 1]);
example::test();
// error: use of undeclared crate or module `nested`
// new_s!();
```

macro_rules!

```
macro_rules! create_vec {
    [$value:expr; $count:expr] => {{
        let mut vec = ::std::vec::Vec::new();
        vec.resize($count, $value);
        vec
    }};
    [$($value:expr),*] => {{
        let mut vec = ::std::vec::Vec::new();
        $(vec.push($value);)*
        vec
    }};
}
```

Inside code, we can insert our meta-variables by writing `$META_VAR_NAME`. If we want to expand variadic pattern, we use `$()*`, `$()+` or `$()?`. This forces pattern to expand exactly zero or more times, more than zero times or not more than one time, or you'll get error.

macro_rules!

```
macro_rules! create_vec {
    [$($value:expr),*] => {{
        let mut vec = ::std::vec::Vec::new();
        $(vec.push($value);)*
        vec
    }};
}
```

You can see that this macro will expand to the series of pushes, and it's not effective, since we'll reallocate multiple times. As in many metaprogramming tools, you cannot just get the size of the `$value`, you need to calculate it.

macro_rules!

```
macro_rules! count {
    () => (0usize);
    ($x:tt $($xs:tt)*) => (1usize + count!($($xs)*));
}
```

To solve this, we'll create a macro that returns 0 when there's no arguments and `1 + count($tail)` when there's more arguments.

Note that our macro is recursive!

macro_rules!

```
macro_rules! create_vec {
    [$($value:expr),*] => {{
        let mut vec = ::std::vec::Vec::with_capacity(
            count!($($value)*)
        );
        $(vec.push($value);)*
        vec
    }};
}
```

To solve this, we'll create a macro that returns 0 when there's no arguments and `1 + count($tail)` when there's more arguments.

macro_rules!

Question: Do we have any limits on macro recursion?

Question: Do we have any limits on macro recursion?

Yes. If not, user will see strange compiler segmentation faults. To extend the limit, use `recursion_limit` attribute:

Question: Do we have any limits on macro recursion?

Yes. To extend the limit, use `recursion_limit` attribute:

```
#![recursion_limit = "300"]
macro_rules! count {
    () => (0usize);
    ($x:tt $($xs:tt)*) => (1usize + count!($($xs)*));
}

// This will fail without recursion_limit!
count!(0 1 2 /* ... */ 254 255)
```

The default value of `recursion_limit` attribute is 128. Please note that this attribute applies to all compile-time recursive operations, including dereference and `const` functions.

cargo-expand

Macros are quite difficult to write and debug. One of the tools that can help you with that is `expand`, installable by `cargo install cargo-expand`. If we use `cargo expand`, the following call:

```
count!(0 1 2 3 4);
```

Expands to:

```
1usize + (1usize + (1usize + (1usize + (1usize + 0usize))))
```

macro_rules!

Macro are so powerful that you can write a lot of stuff inside!

```rust
macro_rules! funny {
    [sentence in English with value $value:expr] => {{
        println!("wow, the value is {}!", $value);
    }};
}

funny![sentence in English with value 42];
```

Of course, there's some limitations on syntax. For instance, you cannot use `|` as a separator after `expr` since Rust actually builds AST before macro expansion, and `|` is an operator, therefore compiler won't know where is separator and where is next expression.

The precise rules are not the material of the lecture.

Macros hygiene

Macros are *hygienic*. That means what is written in macro won't affect code in the call site. For instance, `#define` allows us to make the following mistake:

```
#define FIVE(value) (value * 5)
// Fails!
// assert(FIVE(2 + 3) == 25);
```

This happens because `#define` is not hygienic. In Rust:

```
macro_rules! five_times {
    ($x:expr) => (5 * $x);
}
// Works just fine!
assert_eq!(25, five_times!(2 + 3));
```

Macros hygiene

When we say Rust macros are hygienic, we mean that a declarative macro (generally) cannot affect variables that aren't explicitly passed to it.

```
macro_rules! let_foo {
    ($x:expr) => {
        let foo = $x; }
}
let foo = 1;
// expands to let foo = 2;
let_foo!(2);
// ...But instead, the compiler will even complain
// that the 'let foo' in the macro is an unused variable!
assert_eq!(foo, 1);
```

You can, most of the time, think of macro identifiers as existing in their own universe that is separate from that of the code they expand into.

Macros hygiene

This hygienic separation does not apply beyond variable identifiers. Declarative macros do share a namespace for types, modules, and functions with the call site.

Macros hygiene

This hygienic separation does not apply beyond variable identifiers. Declarative macros do share a namespace for types, modules, and functions with the call site.

This means your macro can define new functions that can be called in the invoking scope, add new implementations to a type defined elsewhere (and not passed in), introduce a new module that can then be accessed where the macro was invoked, and so on.

Macros hygiene

You can explicitly choose to share identifiers between a macro and its caller if you specifically want the macro to affect a variable in the caller's scope.

```
macro_rules! please_set {
    ($i:ident, $x:expr) => {
        $i = $x;
    }
}
let mut x = 1;
please_set!(x, x + 1);
assert_eq!(x, 2);
```

Macros visibility

Unlike pretty much everything else in Rust, declarative macros only exist in the source code after they are declared. If you try to use a macro that you define further down in the file, this will not work!

```rust
// Does not compile!
fn main() {
    count!(0 1 2 3 4 5);
}

macro_rules! count {
    /* macro */
}
```

Macros visibility

Macros are not visible in modules and **pub** keyword does not affect them. If you want to make your macro visible for users, use `#[macro_export]` on macro: it's pretty the same as putting macro in the root of the crate and marking it as **pub**

```rust
mod a {
    mod b {
        // Now visible to the end user
        #[macro_export]
        macro_rules! count {
            () => (0usize);
            ($x:tt $($xs:tt)*) => (1usize + count!($($xs)*));
        }
    }
}
fn main() {
    count!(0 1 2 3 4 5);
}
```

Procedural macros


YO DAWG I HEARD YOU LIKE RUST
SO I BUILT RUST INTO RUST SO YOU
CAN WRITE RUST THAT WRITES RUST

Procedural macros is a Rust code which accepts token stream as an input and outputs Rust code.

This is how `yew` crate (wow, React on Rust!) generates `html` by using procedural macros:

```
html! {
    <div>
        <div class="panel">
            { "Hello, World!" }
        </div>
    </div>
}
```

Procedural macros

There are 3 types of procedural macros:

- Function-like macros - `custom!(...)`. Invoked just like declarative macros, but are much more powerful!

Procedural macros

There are 3 types of procedural macros:

- Function-like macros - `custom!(...)`. Invoked just like declarative macros, but are much more powerful!
- Derive macros - `#[derive(CustomDerive)]`. Accepts `struct`, `enum` or `union` and produces some code depending on input, *adding* it to the source near the target. There's also *derive helper attributes* that exist to give clue to the derive macro.

There are 3 types of procedural macros:

- Function-like macros - `custom!(...)`. Invoked just like declarative macros, but are much more powerful!
- Derive macros - `#[derive(CustomDerive)]`. Accepts `struct`, `enum` or `union` and produces some code depending on input, *adding* it to the source near the target. There's also *derive helper attributes* that exist to give clue to the derive macro.
- Attribute macros - `#[CustomAttribute]`. They define new outer attributes which can be attached to items. They are used to transform the item.

Procedural macros

Procedural macro must be written in special crate:

```toml
[lib]
proc-macro = true
```

An example of the simple, function-like procedural macro:

```rust
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn make_answer(_item: TokenStream) -> TokenStream {
    "fn answer() -> u32 { 42 }".parse().unwrap()
}

// adds 'fn answer() -> u32 { 42 }'
// to the code!
make_answer!();
```

Procedural macros hygiene

As you can see, procedural macros are *unhygienic*. This means they behave as if the output token stream was simply written inline to the code it's next to.

Procedural macros: `TokenStream`

First of all, what is `TokenStream`?

`TokenStream` is a sequence of `TokenTree`.

```
struct TokenStream(Vec<TokenTree>);

pub enum TokenTree {
    Ident(Ident),  // An identifier
    Punct(Punct),  // A punctuation
    Literal(Literal),  // A literal
    Group(Group),  // An another TokenSteam inside braces
}
```

As you can see, any input to procedural macro is *balanced bracket sequence*.

Procedural macros: `TokenStream`

Question: What `TokenStream` this line produces?

```
let r = five_times!(2 + 3);
```

Procedural macros: TokenStream

Question: What TokenStream this line produces?

```rust
let r = five_times!(2 + 3);
```

The output TokenStream:

```
Ident("let"),
Ident("r"),
Punct("="),
Group(
    Literal(5),
    Punct("*"),
    Group(
        Literal(2),
        Punct("+"),
        Literal(3),
    )
)
```

Procedural macros: TokenStream

Let's check use this in real procedural macro!

```rust
#[proc_macro]
pub fn foo(body: TokenStream) -> TokenStream {
    for tt in body.into_iter() {
        match tt {
            TokenTree::Ident(_) => println!("Ident"),
            TokenTree::Punct(_) => println!("Punct"),
            TokenTree::Literal(_) => println!("Literal"),
            _ => {}
        }
    }
    return TokenStream::new();
}
```

Procedural macros: `TokenStream`

```
foo! {
    bar = "123";
}
```

The output:

```
Ident
Punct
Literal
Punct
```

Procedural macros: `TokenStream`

`Group` variant is another `TokenStream` inside the braces.

```
foo!( foo { 2 + 2 } bar );
```

Procedural macros: `TokenStream`

`Group` variant is another `TokenStream` inside the braces.

```
foo!( foo { 2 + 2 } bar );
```

- `foo` is an `Ident("foo")`.

Procedural macros: `TokenStream`

`Group` variant is another `TokenStream` inside the braces.

```
foo!( foo { 2 + 2 } bar );
```

- `foo` is an `Ident("foo")`.
- `bar` is an `Ident("bar")`.

Procedural macros: `TokenStream`

`Group` variant is another `TokenStream` inside the braces.

```
foo!( foo { 2 + 2 } bar );
```

- `foo` is an `Ident("foo")`.
- `bar` is an `Ident("bar")`.
- `{2 + 2}` is a `Group("{}", TokenSteam)`, where `TokenStream` consists of `Literal("2")`, `Punct("+")` and `Literal("2")`,

Procedural macros: TokenStream

```
// 1, 1i32, 1.2, 1.2e-1f64, "foo", r#"bar"#, 'a', b'a'
struct Literal(String, Span);

// foo, r#foo, struct, _
struct Ident(String, Span);

// . , : # $ + -
struct Punct(char, Spacing, Span);

// What is that?
enum Spacing { Alone, Joint }
```

Procedural macros: `Spacing`

What is a `Spacing`?

```
struct Punct(char, Spacing, Span);
//                 ^^^^^^^ - The usage
enum Spacing { Alone, Joint }

foo! { foo::bar }
//         ^^
```

The first `:` is `Punct(":", Alone)`, and the second is `Punct(":", Joint)`.

It's not very convenient to parse Rust code as a `TokenStream`. To solve this problem, there exists `syn` crate which parses `TokenStream` assuming that the input is a valid Rust code.

Let's use it to write our first derive macro! We'll also use a helper attribute to show to what field we want dereference.

```rust
#[derive(Deref)]
struct IntWrapper {
    #[deref]
    inner: i32,
}
```

```
#[proc_macro_derive(Deref, attributes(deref))]
pub fn derive_deref(stream: TokenStream) -> TokenStream {
    let input: DeriveInput = parse_macro_input!(stream);
    let struct_ = match input.data {
        syn::Data::Struct(struct_) => struct_,
        _ => panic!("only structs are supported" ),
    };
    let named_fields = match struct_.fields {
        syn::Fields::Named(fields) => fields.named,
        _ => panic!("only structs with named fields are supported"),
    };

    /* 1 */
```

```
/* 1 */

let deref_field = named_fields
.into_iter()
.filter(|field| {
    field.attrs.iter().find(|attr| {
        attr.path.segments.first()
            .map_or(false, |seg| seg.ident == "deref")
        }
    ).is_some()
})
.next()
.expect("field with #[deref] is not found" );

/* 2 */
```

```
    /* 2 */

    format!(
        r#"impl ::std::ops::Deref for {} {{
            type Target = {};
            fn deref(&self) -> &Self::Target {{
            &self.{} }}
        }}"#,
        input.ident,
        deref_field.ty.to_token_stream(),
        deref_field.ident.unwrap(),
    ).parse().unwrap()
}
```

This macro is not perfect and does not cover a lot of cases, for instance:

```
struct Wrapper<T> {
    inner: T,
}

struct Wrapper<T>
where
    T: MyTrait
{
    inner: T,
}
```

Actually, `format!` is not the best way to return our `impl`. Instead, we can use crate `quote` that enables us to write the Rust code to the output with pleasure.

```
    /* 2 */
    let output = quote! {
        impl ::std::ops::Deref for #ident {
        type Target = #target;
        fn deref(&self) -> &Self::Target {
                &self.#field_ident
            }
        }
    };
    output.into()
}
```

Procedural macros: `Span`

What is a `Span`?

`Span` how the compiler ties generated code back to the source code that generated that code.

`Span` how the compiler ties generated code back to the source code that generated that code.

We've written a macro that depends on correctness of input. For instance, user can write multiple `#[deref]`. Technically, the compiler error occurs *inside* the macro, and user won't understand what's wrong.

```
#[derive(Deref)]
struct IntWrapper {
    #[deref]
    inner1: i32,
    #[deref]
    inner2: i32
}
```

Procedural macros: Span

`Span` how the compiler ties generated code back to the source code that generated that code.

We've written a macro that depends on correctness of input. For instance, user can write multiple `#[deref]`. Technically, the compiler error occurs *inside* the macro, and user won't understand what's wrong.

```
#[derive(Deref)]
struct IntWrapper {
    #[deref]
    inner1: i32,
    #[deref]
    inner2: i32
}
```

But we'd like the compiler to point the user at the `#[deref]` in their code, and that's what spans let us do.

Procedural macros: Span

```rust
if deref_fields.len() > 1 {
    let mut error = quote! {
        compile_error!("found >= 2 fields with #[deref]");
    };
    error.extend(quote_spanned! {
        deref_fields[0].span() => {
            compile_error!("the first #[deref] field");
        }
    });
    error.extend(quote_spanned! {
        deref_fields[1].span() => {
            compile_error!("the second #[deref] field");
        }
    });
    return error.into();
}
```

Procedural macros: Span

```
error: found >= 2 fields with #[deref]
  --> src/main.rs:10:10
10 | #[derive(Deref)]
   |          ^^^^^
error: the first #[deref] field
  --> src/main.rs:12:5
12 | #[deref]
   | ^
error: the second #[deref] field
  --> src/main.rs:14:5
   |
14 | #[deref]
   | ^
```

`compile_error!`

`compile_error!` emits an error when the code is *compiled*. As you can see, we write multiple `compile_error!` to the output, and when user code compiles, the user-friendly error occures.

Procedural macros hygiene

The hygiene of macros output code depends on the `Span` of output code:

- `Span::call_site()` - the name will work just like it was in the user code, i.e call site (the default value).
- `Span::mixed_site()` - the same as hygiene of declarative macros.
- `Span::def_site()` - the name will be seen only in file with macro (only in nightly).

Procedural macros hygiene

For instance, this code will behave just like it was in the user code:

```
#[proc_macro]
pub fn define_foo(_stream: TokenStream) -> TokenStream {
    let output = quote_spanned! {
        Span::call_site() => let foo = "foo";
    };
    output.into()
}
```

Macros from standard library

Conditional compilation

The conditional compilation is also implemented on macros:

```rust
#[cfg(unix)]
pub fn bytes2path(bytes: &[u8]) -> CargoResult<PathBuf> {
    use std::os::unix::prelude::*;
    Ok(PathBuf::from(OsStr::from_bytes(bytes)))
}

#[cfg(windows)]
pub fn bytes2path(bytes: &[u8]) -> CargoResult<PathBuf> {
    use std::str;
    match str::from_utf8(bytes) {
        Ok(s) => Ok(PathBuf::from(s)),
        Err(..) => Err(failure::format_err!(
            "invalid non-unicode path"
        )),
    }
}
```

Conditional compilation

The conditional compilation is also implemented on macros:

```
#[cfg(unix)]
pub fn bytes2path(bytes: &[u8]) -> CargoResult<PathBuf> {
    use std::os::unix::prelude::*;
    Ok(PathBuf::from(OsStr::from_bytes(bytes)))
}

#[cfg(windows)]
pub fn bytes2path(bytes: &[u8]) -> CargoResult<PathBuf> {
    use std::str;
    match str::from_utf8(bytes) {
        Ok(s) => Ok(PathBuf::from(s)),
        Err(..) => Err(failure::format_err!(
            "invalid non-unicode path"
        )),
    }
}
```

Conditional compilation

The `cfg!` macro evaluates boolean combinations of configuration flags at compile-time.

When you use it as *attribute macro*, it will remove the code conditionally. When you use it as a *function-like macro*, it will just evaluate the condition without removing the code.

```
let my_directory = if cfg!(windows) {
    "windows-specific-directory"
} else {
    "unix-directory"
};
```

Conditional compilation

You've already seen this pattern in homeworks:

```rust
#[cfg(test)]
mod tests {
    use some_lib::test_specific_function;
    use super::*;

    #[test]
    fn test_foo() { ... }
}
```

- `#[test]` will never exist in the binary.
- `#[cfg(test)]` groups unit-tests and removes unused imports.

Conditional compilation

Macros `env!` and `option`$_e$*nv!allowsustogetenvironmentvariablesatcompiletime*.

```rust
fn main() {
    let compile_time_path = env!("PATH");
    println!(
        "PATH at *compile* time:\n{}",
        compile_time_path,
    );
}
```

Conditional compilation

Macros `env!` and `option_env!` allows us to get environment variables at compile time.

```
fn main() {
    let compile_time_path = env!("PATH");
    println!(
        "PATH at *compile* time:\n{}",
        compile_time_path,
    );
}
```

stringify!

Macro **stringify!** stringifies its arguments:

```
let one_plus_one = stringify!(1 + 1);
assert_eq!(one_plus_one, "1 + 1");
```

`include_str!` and `include_bytes!`

Macros `include_str!` and `include_bytes!` include file directly to our binary as `&str` or `&[u8]`:

```rust
fn main() {
    let my_str = include_str!("spanish.in");
    assert_eq!(my_str, "adiós\n");
    print!("{}", my_str);
}
```

Attributes

Rust contains a lot of atttibute macros, useful for your code.[3]

- `allow`, `warn`, `deny`, `forbid` — Alters the default lint level.
- `deprecated` — Generates deprecation notices.
- `must_use` — Generates a lint for unused values.
- `inline` — Hint to inline code.
- `cold` — Hint that a function is unlikely to be called.
- `no_std` - Removes std from the prelude.
- And much more!

[3] Attributes - The Rust Reference

Conclusion

- We studied Rust closures and found their unique features.
- Studied declarative and procedural macros.
- Highlighted their weak and strong sides.
- And looked at standard library macros.