3-3: Operator overloading and formatting (Theory)

Artem Pavlov, TII, Abu Dhabi, 05.05.2024

Special marker traits

- Copy: gives a type 'copy semantics' instead of 'move semantics'
- Sized: types with a constant size known at compile time
- Sync: types for which it is safe to share references between threads
- Send: types that can be transferred across thread boundaries

Custom destructor with Drop

```rust
pub struct Foo(u32);

impl Drop for Foo {
    fn drop(&mut self) {
        let val = self.0;
        println!("Dropped Foo with {val}");
    }
}
```

Formatting traits

- Formatting traits reside in the std::fmt module: https://doc.rust-lang.org/std/fmt/index.html
- Formatting options support a number of modifiers
- For example, {:08X} prints integer in upper hex with zero padding up to 8 digits

Formatting options and associated traits

- nothing ⇒ Display
- ? ⇒ Debug
- x? ⇒ Debug with lower-case hexadecimal integers
- X? ⇒ Debug with upper-case hexadecimal integers
- p ⇒ Pointer
- x ⇒ LowerHex
- X ⇒ UpperHex
- o ⇒ Octal
- b ⇒ Binary
- e ⇒ LowerExp
- E ⇒ UpperExp

Formatter

- Contains information about requested formatting
- Can be used with the write! macro
- Contains convenience methods for implementation of common formatting cases
- https://doc.rust-lang.org/std/fmt/struct.Formatter.html

Operator overloading

- Operators in Rust are just syntax sugar for traits defined in the std::ops module: https://doc.rust-lang.org/std/ops/index.html
- For example, a + b is desugared to a.add(b)
- Operator traits can be implemented for a custom type, which makes it possible to use operators with it
- Many operators have pairs, e.g. Add and AddAsing

Ops traits and references

- Non-asign arithmetic traits take owned values
- But we can use references as “owned” values
- For example, we have the following impls:
  - impl Add<&i32> for &i32
  - impl Add<&i32> for i32
  - impl<'a> Add<i32> for &'a i32

Indexing

- Indexing (i.e. the “square bracket” operator) is controlled by the Index and IndexMut traits
- These traits can be implemented for custom types and allow to perform indexing with arbitrary types

Deref coercions

- Deref and DerefMut are the special traits which enable “deref coercions”
- If T implements Deref<Target = U>, and v is a value of type T, then:
  - In immutable contexts, *v (where T is neither a reference nor a raw pointer) is equivalent to *Deref::deref(&v).
  - Values of type &T are coerced to values of type &U
  - T implicitly implements all the methods of the type U which take the &self receiver.

Questions?