

CS 241: Systems Programming

Lecture 20. Generics and Traits

Fall 2023

Prof. Stephen Checkoway

Generics

Similar to Java's generics

- ▶ e.g., `ArrayList<Integer>` and `Vec<i32>`

We can have generic

- ▶ structs
- ▶ enums
- ▶ functions/methods

Generic struct

```
struct Container<T> {  
    x: T,  
    y: T,  
    z: T,  
}
```

Each type we plug in for T
gives a new type

```
c1: Container<i32>  
c2: Container<f64>  
c3: Container<&str>
```

```
fn main() {  
    let c1 = Container { x: 10, y: 20, z: 30 };  
    let c2 = Container { x: 1.8, y: 2.3, z: -7.2 };  
    let c3 = Container { x: "abc", y: "def", z: "ghi" };  
}
```

Implementing methods

A generic impl block lists all type parameters in <> after impl

```
impl<T> Container<T> {  
    fn new(x: T, y: T, z: T) -> Self {  
        Self { x, y, z }  
    }  
  
    fn set_y(&mut self, y: T) {  
        self.y = y;  
    }  
}
```

“For any type T, implement functions/methods for Container<T>”

Type parameters can have different names

```
struct Container<T> {  
    x: T,  
    y: T,  
    z: T,  
}
```

“For any type `Typ`, implement functions/methods for `Container<Typ>`”

```
impl<Typ> Container<Typ> {  
    fn new(x: Typ, y: Typ, z: Typ) -> Self {  
        Self { x, y, z }  
    }  
  
    fn set_y(&mut self, y: Typ) {  
        self.y = y;  
    }  
}
```

Can implement methods for specific types

No generic parameter

```
impl Container<&str> {  
    fn join(&self) -> String {  
        format!("{}", self.x, self.y, self.z)  
    }  
}
```

.join() method can only be called on Container<&str>

How do we implement a `max()` method for the `Container<T>` struct that's only defined when `T` is an `i32`?

```
struct Container<T> {  
    x: T,  
    y: T,  
    z: T,  
}
```

```
// A  
impl Container {  
    fn max(&self) -> i32 { todo!() }  
}
```

```
// B  
impl<T> Container<T> {  
    fn max(&self) -> i32 { todo!() }  
}
```

```
// C  
impl<T> Container<i32> {  
    fn max(&self) -> i32 { todo!() }  
}
```

```
// D  
impl<i32> Container<T> {  
    fn max(&self) -> i32 { todo!() }  
}
```

```
// E  
impl Container<i32> {  
    fn max(&self) -> i32 { todo!() }  
}
```

A generic version of join gives an error

```
impl<T> Container<T> {  
    fn join(&self) -> String {  
        format!("{}", self.x, self.y, self.z)  
    }  
}
```

```
error[E0277]: `T` doesn't implement `std::fmt::Display`  
--> generics.rs:21:27
```

```
21 |         format!("{}", self.x, self.y, self.z)  
    |                   ^^^^^^^ `T` cannot be formatted with the  
    |                   default formatter
```

```
help: consider restricting type parameter `T`
```

```
19 | impl<T: std::fmt::Display> Container<T> {  
    |           ++++++
```

Rustc doesn't know anything about type T

We'll return to this shortly!

Generic enums

We've seen this with `Option<T>` and `Result<T, E>`

```
enum Option<T> {  
    None,  
    Some(T),  
}
```

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

Generic type aliases

```
type Result<T> = std::result::Result<T, Box<dyn std::error::Error>>;
```

Multiple generic parameters

```
enum Either<L, R> {  
    Left(L),  
    Right(R),  
}
```

Returns a reference to
an L or None

```
impl<L, R> Either<L, R> {  
    fn left(&self) -> Option<&L> {  
        match self {  
            Either::Left(x) => Some(x),  
            _ => None,  
        }  
    }  
}
```

Always returns an L
Moves it out of self

```
fn unwrap_left_or(self, default: L) -> L {  
    match self {  
        Either::Left(x) => x,  
        _ => default,  
    }  
}
```

Generic impl with fewer parameters

```
impl<T> Either<T, T> {  
    fn unwrap(self) -> T {  
        match self {  
            Either::Left(x) => x,  
            Either::Right(x) => x,  
        }  
    }  
}
```

“For any type T, implement functions/methods for Either<T, T>”

Generic functions

Type parameters go after the function name but before parameter list

```
fn do_something<T>(x: T) -> T {  
    todo!()  
}
```

Generic arguments have limited functionality

There's not a lot we can do with x; we cannot

1. Print x
2. Call methods on x (no methods defined for every type unlike Java)
3. Modify x (no way to modify a value that works for every type)

We cannot even create a new instance of T!

```
fn do_something<T>(x: T) -> T {  
    todo!()  
}
```

Given the following code, what can we say about val afterward

```
fn do_something<T>(x: T) -> T { /* ... */ }
```

```
let val = do_something("hello");
```

- A. val has type &str and can be any string
- B. val has type &str and is exactly "hello"
- C. val has some unknown type is an unknown value
- D. It's not possible to return a generic T so this is an error

Traits

Defining a trait

```
pub trait Run {  
    fn setup(&mut self);  
  
    fn run(&mut self);  
  
    fn cleanup(&mut self);  
}
```

Defines a public trait with three methods (that are public because the trait is)

Implementing a trait

```
struct Foo;
impl Run for Foo {
    fn setup(&mut self) {
        println!("Foo::setup()");
    }

    fn run(&mut self) {
        println!("Foo::run()");
    }

    fn cleanup(&mut self) {
        println!("Foo::cleanup()");
    }
}
```

Trait name

Type implementing the trait

Calling methods from the trait

To call a method from a trait, the trait must be in scope

- ▶ If it's in the same module, it's in scope
- ▶ If it's not in the same module, you need to use a use statement to bring it into scope like
use std::io::Write;
to bring the Write trait into scope

If the trait isn't in scope, we get an error

```
use std::{io, fs};

fn main() -> io::Result<()> {
    let mut file = fs::File::create("blarg.txt"?);
    writeln!(file, "Let's write some data!");
    Ok(())
}
```

writeln!() acts like println!() except it writes to something that implements std::io::Write
It returns an io::Result

A trait must be in scope to use its methods

```
error[E0599]: cannot write into `File`
```

```
--> generics.rs:171:14
```

```
171 |         writeln!(file, "Let's write some data!");  
    |         ~~~~~^ method not found in `File`
```

```
::: /Users/steve/.rustup/toolchains/stable-aarch64-apple-darwin/lib/rustlib/src/rust/library/std/src/io/mod.rs:1693:8
```

```
1693 |     fn write_fmt(&mut self, fmt: fmt::Arguments<'_>) -> Result<()> {  
    |     ~~~~~ the method is available for `File` here
```

```
note: must implement `io::Write`, `fmt::Write`, or have a `write_fmt` method
```

```
--> generics.rs:171:14
```

```
171 |         writeln!(file, "Let's write some data!");  
    |         ~~~~~^
```

= **help:** items from traits can only be used if the trait is in scope

help: the following trait is implemented but not in scope; perhaps add a `use` for it:

```
167 + use std::io::Write;
```

And here's the fix

Default methods

```
pub trait Run {  
    fn setup(&mut self) {  
        println!("Default setup()");  
    }  
  
    fn run(&mut self);  
  
    fn cleanup(&mut self) {  
        println!("Default cleanup()");  
    }  
}
```

To implement Run, we have to implement at least the run() method

Implementing a trait with default methods

```
impl Run for Foo {  
    fn setup(&mut self) {  
        println!("Foo::setup()");  
    }  
  
    fn run(&mut self) {  
        println!("Foo::run()");  
    }  
}  
  
fn main() {  
    let mut f = Foo;  
    f.setup();  
    f.run();  
    f.cleanup();  
}
```

```
Foo::setup()  
Foo::run()  
Default cleanup()
```

Trait bounds

Traits are used to constrain or **bound** the types of generic arguments

```
<T: TraitName>
```

The T type parameter is restricted to be some type that implements TraitName

A generic version of join gives an error

```
impl<T> Container<T> {  
    fn join(&self) -> String {  
        format!("{}", self.x, self.y, self.z)  
    }  
}
```

This is the issue: T doesn't implement the Display trait

```
error[E0277]: `T` doesn't implement `std::fmt::Display`  
--> generics.rs:21:27
```

```
21 |         format!("{}", self.x, self.y, self.z)  
    |                   ^^^^^^^ `T` cannot be formatted with the  
    |                   default formatter
```

```
help: consider restricting type parameter `T`
```

```
19 | impl<T: std::fmt::Display> Container<T> {  
    |           ++++++
```

This is suggesting we require T implement Display

impl with bound

```
struct Container<T> {  
    x: T,  
    y: T,  
    z: T,  
}
```

We can create Container<T>
for any type T.

```
impl<Typ> Container<Typ> {  
    fn new(x: Typ, y: Typ, z: Typ) -> Self { Self { x, y, z } }  
    fn set_y(&mut self, y: Typ) { self.y = y; }  
}
```

.join() is only defined for types
that implement Display

```
impl<D: std::fmt::Display> Container<D> {  
    fn join(&self) -> String {  
        format!("{}", self.x, self.y, self.z)  
    }  
}
```

The Clone trait defines a `fn clone(&self) -> Self` method

How do you specify the dup method's generic arguments?

```
fn dup(x: &T, n: usize) -> Vec<T> {  
    let mut result: Vec<T> = Vec::new();  
    for _ in 0..n {  
        result.push(x.clone())  
    }  
    result  
}
```

A. `fn<T> dup(x: &T, n: usize) -> Vec<T>`

B. `fn<T: Clone> dup(x: &T, n: usize) -> Vec<T>`

C. `fn dup<T: Clone>(x: &T: Clone, n: usize) -> Vec<T: Clone>`

D. `fn dup<T: Clone>(x: &T, n: usize) -> Vec<T>`

E. More than one of the above (which ones?)

A bunch of standard library traits

`std::fmt::Display` — data can be formatted via `{}` in `format!()`/`println!()`/etc

`Debug` — data can be formatted via `{:?}` in `format!()`/`println!()`/etc

`Clone` — defines `.clone()` method

`std::io::BufRead` — defines a bunch of methods, including `read_line()`

`std::io::Read` —
defines `.read()`, `.read_to_end()`, `.read_to_string()`, etc.

`std::io::Write` — defines `.write()` and `.write_all()` methods (and others)

Weird Read/Write trait behavior

If `T` implements `Write`, then `&mut T` implements `Write`

- ▶ `impl<W: Write> Write for &mut W { /* ... */ }`

If `T` implements `Read`, then `&mut T` implements `Read`

- ▶ `impl<R: Read> Read for &mut R { /* ... */ }`

For example, `File` implements `Read` and `Write` so `&mut File` implements them as well

- ▶ Functions often have generic arguments like

```
fn foo<W: Write>(writer: W)
```

You can pass a `File` or a `&mut File` (or anything else that implements `Write`)

Defining behavior in terms of generics

```
use std::io::{self, Write};

fn write_haiku<W: Write>(mut writer: W) -> io::Result<()> {
    writeln!(writer, "古池や蛙飛び込む水の音");
    writeln!(writer, "    ふるいけやかわずとびこむみずのおと");
    Ok(())
}

fn main() -> io::Result<()> {
    let mut v: Vec<u8> = Vec::new();
    write_haiku(&mut v)?; // Write into v

    write_haiku(io::stdout())?; // Write to stdout

    let mut file = std::fs::File::create("haiku.txt");
    write_haiku(&mut file)?; // Write once
    write_haiku(file)?; // Write a second time, and take ownership!
    Ok(())
}
```