5 Generic Types

Thomas Prokosch
November 13, 2018

Outline

1. Traits
2. Clone and Copy
3. Associated types
4. Display and Debug
5. Trait objects
6. Summary

C++ Ambitions

In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for [...]. And further: What you do use, you couldn't hand code any better.

– Bjarne Stroustrup, Abstraction and the C++ Machine Model (2004)

Traits

Traits define common behavior

Traits are

- a **set of methods** implemented for a particular data type
- modeled after Haskell's type classes
- similar to the traits in Scala, and to interfaces in Java, Kotlin, and Go

```
use std::f32::{self, consts::PI};
struct Square { width: u64 }
struct Circle { diameter: u64 }

trait Area {
    fn area(&self) -> f32;
}
impl Area for Square {
    fn area(&self) -> f32 {
        (self.width * self.width) as f32 }
}
impl Area for Circle {
    fn area(&self) -> f32 {
        ((self.diameter as f32)/2.).powi(2) * PI }
}

let (s, c) = (Square { width: 2 }, Circle { diameter: 2 });
println!("area of 2-square: {}, 2-circle: {}", s.area(), c.area());
```

Traits are constraints

Traits

- can be **used like types**.
- are a kind of **type constraint**:
  We don't know anything about the **actual type** except that it implements some methods: The methods defined by the trait.
  Rust calls these constraints defined by traits trait bounds.
- use the same **syntax as lifetime** specifiers.

Syntax

Base syntax `identifier<T: Trait>(t: &T)`

Using `where` `identifier<T>(t: &T) where T: Trait`
in longer signatures (line break!)

T	type variable
Trait	trait (`T: Trait` means: T is of type Trait)
t	variable of type T

Generic functions and trait bounds

Generic function with **trait bound**:

```
fn larger<T: std::cmp::PartialOrd>(a: T, b: T) -> T {
    if a > b { a }
    else     { b }
}

println!("larger(5, 3) = {}", larger(5, 3));
println!("larger(a, c) = {}", larger('a', 'c'));
```

Let’s have a look at `std::cmp::PartialOrd`!

`std::cmp::PartialOrd`

```
pub trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where Rhs: Sized {
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    fn lt(&self, other: &Rhs) -> bool { ... }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

Observations:

1. **Traits**, like types, can be **parametrized** with type variables.
2. There is not only `&self` but also `Self`. → next slide
3. Some functions are declared, some offer definitions.
   → default implementations
4. There exists a trait `Sized` → markers
5. **Additional syntax:** `trait Trait1: Trait2 ...` → supertraits

The type of `&self`

`&self` in **method declarations** is shorthand: Refers to special type `Self`. `Self` means the **current data type**.

shorthand	actual type
&self	self: &Self
&mut self	self: &mut Self
self	self: Self
	self: Box<Self>

`self: Box<Self>` means that the method can only be called when the **data type is contained** in a `Box`.

- The last two variants are rarely used.
- **Calling methods:** Only `self` is used (idiomatic!)

Special type `Self`

- Use when a second element of the **same type** is needed.
- `Self` and the actual type can be used (almost) **interchangably**.

```
enum Color { Red, Green, Blue }
let (r, b) = (Color::Red, Color::Blue);
println!("Color matches: {}", r == r);

impl std::cmp::PartialEq for Color {
    fn eq(&self, other: &Self) -> bool {
        match (self, other) {
            (Color::Red,   Color::Red)   |
            (Color::Green, Color::Green) |
            (Color::Blue,  Color::Blue) => true,
            _                           => false
        }
    }
}
```

Default implementations

Sometimes, functions can be expressed in terms of other functions. For example, $a \neq b \equiv \neg(a = b)$ and $a = b \equiv \neg(a \neq b)$.

```
pub trait PartialEq<Rhs = Self>
where Rhs: Sized {
    fn eq(&self, other: &Rhs) -> bool { !self.ne(other) }
    fn ne(&self, other: &Rhs) -> bool { !self.eq(other) }
}
```

(Remark: `PartialEq` requires symmetry, transitivity, **no reflexivity**)

- Only one of the two methods **requires** an implementation.
- Both methods **can** be implemented if needed (e.g. better speed).

Assume, `PartialEq::eq()` is implemented for `Vec<>`:

- Checks all elements, only then return result
- What happens when using default implementation for checking inequality…?

Static dispatch

Traits are statically dispatched:

- Traits as a generic type stand in for a concrete type.
- Every instantiated concrete type gets their **own copy** of trait functions: Monomorphization.
  — "What you do use, you couldn't hand code any better."
  The abstraction is **completely erased** from the binary.

```
fn print_hash<T: Hash>(t: &T) {
    println!("Hash: {}", t.hash()) }
print_hash(&true);            // instantiates T = bool
print_hash(&12_i64);          // instantiates T = i64

// The compiled code:
__print_hash_bool(&true);     // specialized bool version
__print_hash_i64(&12_i64);    // specialized i64 version
```

Static and dynamic dispatch

static dispatch	**dynamic dispatch**
call target fixed at **compile time**	call target determined at **run time**
fast code	pointer indirections, cache misses
expressiveness somewhat limited	**run-time decisions** possible
long compilation time (flattening)	code generation more complex
large binary size (flattening)	**smaller binary** size

C++ Compilation time and memory requirements are an issue with heavily templated code. C++ templates utilize **static dispatch only**.

Rust Supports **static** dispatch (default) and **type-safe dynamic** dispatch.

Multiple trait bounds

Sometimes, we need **more than one trait**:

```
<Trait1 + Trait2>
```

- Traits combined with +
  - **satisfy both** `Trait1` and `Trait2`
  - are generally **independent** from each other (dependence: supertraits)
- Syntax also for combining **traits with lifetimes**

```
fn print_larger<T>(a: T, b: T)
where T: std::cmp::PartialOrd + std::fmt::Display {
    println!("Given {} and {}, {} is larger.",
             &a, &b, if a>b { &a } else { &b });
}
print_larger(5, 4);
```

Supertraits

Supertraits establish **dependencies between traits**: Defining a trait may require functions defined in other traits.

```
pub trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;

    fn max(self, other: Self) -> Self { ... }
    fn min(self, other: Self) -> Self { ... }
}
```

A total order `Ord`

- is a special case of a partial order `PartialOrd`, and
- any two elements must be comparable (with `Eq`).

Clone and Copy

Markers

- Traits, even when implementing the same functions, are distinct.
- Marker traits implement **no functions**, used to **distinguish types**.

```
pub trait Sized { }
```

trait	meaning	implemented on
`Copy`	copy values by copying bits	machine types on stack
`Sized`	constant size (compile-time)	non-dynamically sized
`Send`	transferable between threads	most non-pointer types
`Sync`	thread-sharable references	`&T` is `Send`

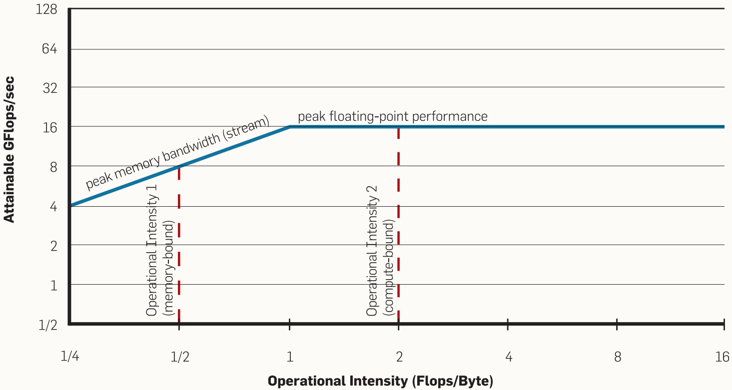
These traits are **automatically implemented** (by the compiler) if appropriate.

Roofline model

Only two reasons why computer programs take time:

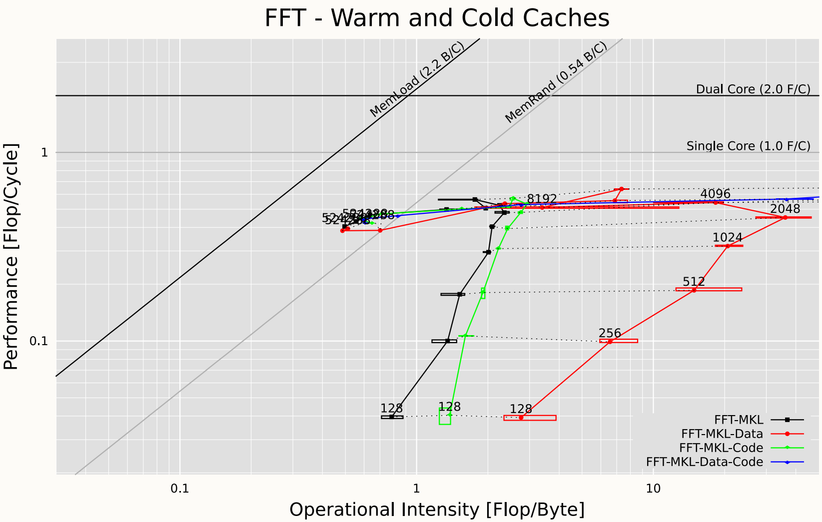
- They compute: Compute-bound application.
- They wait for data: Memory-bound application.
- Only **one of the reasons** fits in a particular situation.

Dependent on hardware and implementation, usually memory-bound.



source: Williams et al (2009) Roofline

Fast Fourier Transform (varying problem size)



source: Steinmann (2012) Applying the Roofline Model

`Copy`

`Copy` and `Clone` seem similar. However,
`Copy` is a **fast operation**:

- works on small data types
- efficiently implemented in hardware
  usually: single instruction
- little impact on **memory caches**
  values are on stack: only few memory pages touched
- **saves space**: no references/pointers
- makes life **much** easier

This is why `Copy` is implemented automatically.

Clone

`Clone` is an **expensive operation**:

- larger data, possibly spread over multiple memory pages
- usually **bit-copying not possible**: costly operations
- more likely that implementation will be **memory-bound**

Cloning is **explicit**:

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

Think twice before you

- implement `Clone`
- call `fn clone()`

Largest element in a slice

```
fn largest<T>(list: &[T]) -> T
where T: std::cmp::PartialOrd + std::clone::Clone {
    let mut largest = &list[0];       // largest: &T
    for item in list.iter() {         // item:    &T
        if item > largest { largest = item } }
    largest.clone()    // clone as late as possible
}

let ns = vec![34, 50, 25, 100, 65];
println!("Largest number: {}", largest(&ns));

let cs: Vec<char> = "Rust".chars().collect();
println!("Largest char:   {}", largest(&cs));
```

Associated types

The need for associated types

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

fn distance<N, E, G: Graph<N, E>>
    (graph: &G, start: &N, end: &N) -> u32 { ... }
```

Why do we need to pass type parameter `E` to `fn distance()`? Distance calculation works without edges!
Types `N`, `E` are **always the same** for a particular `Graph`!

Solution: Associated types.

Declaring associated types

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

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

fn distance<G: Graph>
    (graph: &G, start: &G::N, end: &G::N) -> u32 { ... }
```

Implementing associated types

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

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

    fn has_edge... // rest as usual
}
```

- `type` keyword does **not** introduce a type alias
- **concrete types** are assigned to **associated types** using =
- associated types also used when implementing `trait Iterator`

Example for iterators: Fibonacci sequence (Rust by Example)

Display and Debug

Adding complex numbers ℂ

Want to achieve:

```
struct Complex(f64, f64);

let p1 = Complex(1., 2.);
let p2 = Complex(2., 1.);
let input = format!("Adding {} to {} =", p1, p2);
println!("{} {}", input, p1+p2);
```

Fails to compile:

1. We cannot `Add` `Complex` to `Complex`.

   ```
   an implementation of ‘std::ops::Add‘ might be
   missing for ‘({integer}, {integer})‘
   ```

2. We cannot `Display` `Complex`.

Operator overloading

```
use std::ops::Add;
struct Complex(f64, f64);

impl Add for Complex {
    type Output = Complex;

    // take ownership of self and other
    fn add(self, other: Complex) -> Complex {
        Complex(self.0 + other.0, self.1 + other.1)
    }
}

let p1 = Complex(1., 2.);
let p2 = Complex(2., 1.);
let input = format!("Adding {} to {} =", p1, p2);
println!("{} {}", input, p1+p2);   // still no Display
```

Manually implementing `Display` and `Debug`

- `Display` **must be** and
- `Debug` **can be** defined **manually**

using the `Formatter` builder API:

- `fn fmt(&self, f: &mut Formatter) -> Result<(), Error>`

```
use std::fmt;
struct ... { x: ... }

impl fmt::Display for ... {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({})", self.x)
    }
}
```

`Display` trait for complex numbers

```
use std::fmt;
struct Complex(f64, f64);

impl fmt::Display for Complex {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{}+{}i", self.0, self.1)
    }
}

println!("Got {} and {}",
         Complex(1., 2.), Complex(2., 1.));
```

```
Got 1+2i and 2+1i
```

Derivable traits

- `Debug` is essentially the **same** as `Display`
- Too much **boilerplate** for debug output nobody sees anyway…
- Computer should do the work:

  `#[derive(Debug)]`

Automatically derivable traits

`Debug`	programmer output
`PartialEq`, `Eq`	equality comparisons
`PartialOrd`, `Ord`	ordering comparisons
`Copy`, `Clone`	duplicating values
`Default`	generating default values
`Hash`	hashing values

Formatting traits and the `derive` directive

Interesting formatting traits with their syntax:

interpolation with	interpreted by
{}	std::fmt::Display
{:?}	std::fmt::Debug
{:#?}	std::fmt::Debug for pretty printing
{:p}	std::fmt::Pointer

```
#[derive(Debug)]
struct Point { x: i32, y: i32 }
let origin = Point { x: 0, y: 0 };
println!("The origin is: {:#?}", origin);
```

```
The origin is: Point {
    x: 0,
    y: 0
}
```

Trait objects

Heterogeneous data store

```
trait Draw { fn draw(&self); }

struct Square;
impl Draw for Square { fn draw(&self) {println!("Square")}}
struct Circle;
impl Draw for Circle { fn draw(&self) {println!("Circle")}}

let objs = vec![Square, Circle, Square];
for o in objs { o.draw(); }
```

What happens? Why?

Heterogeneous data store

```
trait Draw { fn draw(&self); }

struct Square;
impl Draw for Square { fn draw(&self) {println!("Square")}}
struct Circle;
impl Draw for Circle { fn draw(&self) {println!("Circle")}}

let objs = vec![Square, Circle, Square];
for o in objs { o.draw(); }
```

What happens? Why?

```
error[E0308]: mismatched types
14 | let objs = vec![Square, Circle, Square];
   |                         ^^^^^^
expected struct ‘main::Square‘, found struct ‘main::Circle‘
```

→ Types may have **different size**!

Trait objects

Different size → Use `Box<...>`**!**

```
let components: Vec<Box<Draw>>;
```

- We have invented trait objects!
- Actual type depends on **run-time** information, so **no monomorphization** possible
- **Dynamic dispatch** must be used

Review: Genericity

Generics	`<T>` create **parametrized** data types
Traits	`<T: Trait>` define **common operations** on distinct data types and define constraints on input data
Trait objects	`Box<Trait>` are **diverse objects** (of unknown type) with common operations **Purpose:** abstraction over common behavior; checked at run-time when concrete type (substituted for the trait) is known

Meaning of trait objects to a compiler

Traits

- **static dispatch**
  The compiler knows at compile-time which function to call.
- monomorphization possible

Trait objects

- **type-safe** through compile-time checks
  Compiler checks whether a particular data type actually implements the methods we want to call at run-time. (Trait!)
- **dynamic dispatch**
  Actual type of values is only known at run-time.
  - Advantage: Flexibility.
  - Disadvantage: No inlining, thus only little code optimization.

The `dyn` keyword in **Rust 1.27** aims to emphasize dynamic dispatch:

```
let components: Vec<Box<dyn Draw>>;
```

What traits can be made into trait objects?

What happens here?

```
let v: Vec<Box<Clone>> = Vec::new();
println!("Output: {}", v.len());
```

1. The program does not compile, keyword `dyn` is missing.
2. The program does not compile, we cannot use trait `Clone` here.
3. The output is zero, no elements in `v`.
4. The output is non-zero, `Box` takes up space.

What traits can be made into trait objects?

What happens here?

```
let v: Vec<Box<Clone>> = Vec::new();
println!("Output: {}", v.len());
```

2 The program does not compile, we cannot use trait `Clone` here.

Requirements to convert a trait into a trait object

- All methods of the trait have a return type which is **not** `Self`.
- No methods of the trait have **generic type** parameters.

Here: `fn std::clone::Clone::clone(&self) -> Self;`

Lifetime of trait objects

Automatic lifetime annotation

`Box<T>` → `Box<T + 'static>`

Problem:

- **trait object returned** from a member function
- usually, `&self` cannot be `'static`

```
struct Store(i32);
impl Store {
    fn giveback(&self) -> Box<Fn(i32) -> i32> {
        Box::new(|a| self.0+a)    // Fn: next time
    }
}
```

Possible solutions:

1. annotate lifetimes of `Box` object and `&self` (same lifetime)
2. `move` consumes closed values: `Box::new(move |a| self.0+a)`

Abstract return types with `impl`

`impl` keyword for returning abstract types since Rust 1.26

- similar to trait objects (without the box)
- existential type
- re-introduction of **static dispatch** (Box uses dynamic dispatch)
- **Usage:** Replace

  `fn ... -> Box<Trait>`

  with

  `fn ... -> impl Trait`

  and remove the function calls related to `Box`.

```
fn giveback() -> impl Fn(i32) -> i32 {
    let local = 5;
    move |a| local+a
}
```

Summary

Is Rust object-oriented?

- objects contain **data and behavior**
- **encapsulation** hides implementation details (via modules and the `pub` keyword)
- **polymorphism** (via traits)
- **no** notion of structural inheritance but Rust includes
  - inheritance in the type system (subtyping with variance)

Uses of traits

Traits are a unifying concept:

- **conditional APIs** — function only valid if input satisfies constraints (trait bounds)
- **extension methods** — methods to an externally defined type
- **markers** — `Sized`, `Send`,...
- **overloading**, operator overloading — `Add`, `Clone`,...
- **closures** — implemented using traits, discussed next time

Summary

Today's goals:

1. Learn about and get to know some traits.
2. Use boxed values with trait objects.
3. Understand performance issues related to memory operations.

We have learned about…

- traits, trait bounds, and supertraits
- implementing, deriving, and marker traits
- trait objects and their limits
- the type `Self`
- operator overloading
- static and dynamic dispatch
- roofline model