PREV
10. Enums and Patterns

NEXT
12. Operator Overloading

Chapter 11. Traits and Generics

> *[A] computer scientist tends to be able to deal with nonuniform structures—case 1, case 2, case 3—while a mathematician will tend to want one unifying axiom that governs an entire system.*
>
> — Donald Knuth

One of the great discoveries in programming is that it's possible to write code that operates on values of many different types, *even types that haven't been invented yet.* Here are two examples:

- `Vec<T>` is generic: you can create a vector of any type of value, including types defined in your program that the authors of `Vec` never anticipated.

- Many things have `.write()` methods, including `File`s and `TcpStream`s. Your code can take a writer by reference, any writer, and send data to it. Your code doesn't have to care what type of writer it is. Later, if someone adds a new type of writer, your code will already support it.

Of course, this capability is hardly new with Rust. It's called *polymorphism*, and it was the hot new programming language technology of the 1970s. By now it's effectively universal. Rust supports polymorph-

ism with two related features: traits and generics. These concepts will be familiar to many programmers, but Rust takes a fresh approach inspired by Haskell's typeclasses.

Traits are Rust's take on interfaces or abstract base classes. At first, they look just like interfaces in Java or C#. The trait for writing bytes is called `std::io::Write`, and its definition in the standard library starts out like this:

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    ...
}
```

This trait offers several methods; we've shown only the first three.

The standard types `File` and `TcpStream` both implement `std::io::Write`. So does `Vec<u8>`. All three types provide methods named `.write()`, `.flush()`, and so on. Code that uses a writer without caring about its type looks like this:

```
use std::io::Write;

fn say_hello(out: &mut dyn Write) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

The type of `out` is `&mut dyn Write`, meaning "a mutable reference to any value that implements the `Write` trait." We can pass `say_hello` a mutable reference to any such value:

```
use std::fs::File;
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?;  // works

let mut bytes = vec![];
say_hello(&mut bytes)?;  // also works
assert_eq!(bytes, b"hello world\n");
```

This chapter begins by showing how traits are used, how they work, and how to define your own. But there is more to traits than we've hinted at so far. We'll use them to add extension methods to existing types, even built-in types like `str` and `bool`. We'll explain why adding a trait to a type costs no extra memory and how to use traits without virtual method call overhead. We'll see that built-in traits are the hook into the language that Rust provides for operator overloading and other features. And we'll cover the

`Self` type, associated functions, and associated types, three features Rust lifted from Haskell that elegantly solve problems that other languages address with workarounds and hacks.

Generics are the other flavor of polymorphism in Rust. Like a C++ template, a generic function or type can be used with values of many different types.

```
/// Given two values, pick whichever one is less.
fn min<T: Ord>(value1: T, value2: T) -> T {
    if value1 <= value2 {
        value1
    } else {
        value2
    }
}
```

The `<T: Ord>` in this function means that `min` can be used with arguments of any type `T` that implements the `Ord` trait—that is, any ordered type. A requirement like this is called a *bound*, because it sets limits on which types `T` could possibly be. The compiler generates custom machine code for each type `T` that you actually use.

Generics and traits are closely related: generic functions use traits in bounds to spell out what types of arguments they can be applied to. So we'll also talk about how `&mut dyn Write` and `<T: Write>` are similar, how they're different, and how to choose between these two ways of using traits.

Using Traits

A trait is a feature that any given type may or may not support. Most often, a trait represents a capability: something a type can do.

- A value that implements `std::io::Write` can write out bytes.
- A value that implements `std::iter::Iterator` can produce a sequence of values.
- A value that implements `std::clone::Clone` can make clones of itself in memory.
- A value that implements `std::fmt::Debug` can be printed using `println!()` with the `{:?}` format specifier.

Those four traits are all part of Rust's standard library, and many standard types implement them. For example:

- `std::fs::File` implements the `Write` trait; it writes bytes to a local file. `std::net::TcpStream` writes to a network connection. `Vec<u8>` also implements `Write`. Each `.write()` call on a vector of bytes appends some data to the end.

- `Range<i32>` (the type of `0..10`) implements the `Iterator` trait, as do some iterator types associated with slices, hash tables, and so on.
- Most standard library types implement `Clone`. The exceptions are mainly types like `TcpStream` that represent more than just data in memory.
- Likewise, most standard library types support `Debug`.

There is one unusual rule about trait methods: the trait itself must be in scope. Otherwise, all its methods are hidden.

```
let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?;  // error: no method named `write_all`
```

In this case, the compiler prints a friendly error message that suggests adding `use std::io::Write;` and indeed that fixes the problem:

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?;  // ok
```

Rust has this rule because, as we'll see later in this chapter, you can use traits to add new methods to any type—even standard library types like `u32` and `str`. Third-party crates can do the same thing. Clearly, this could lead to naming conflicts! But since Rust makes you import the traits you plan to use, crates are free to take advantage of this superpower. To get a conflict, you'd have to import two traits that add a method with the same name to the same type. This is rare in practice. (If you do run into a conflict, you can spell out what you want using fully qualified method syntax, covered later in the chapter.)

The reason `Clone` and `Iterator` methods work without any special imports is that they're always in scope by default: they're part of the standard prelude, names that Rust automatically imports into every module. In fact, the prelude is mostly a carefully chosen selection of traits. We'll cover many of them in Chapter 13.

C++ and C# programmers will already have noticed that trait methods are like virtual methods. Still, calls like the one shown above are fast, as fast as any other method call. Simply put, there's no polymorphism here. It's obvious that `buf` is a vector, not a file or a network connection. The compiler can emit a simple call to `Vec<u8>::write()`. It can even inline the method. (C++ and C# will often do the same, although the possibility of subclassing sometimes precludes this.) Only calls through `&mut dyn Write` incur the overhead of a dynamic dispatch, also known as a virtual method call, which is indicated by the `dyn` keyword in the type. `dyn Write` is known as a *trait object*; we'll look at the technical details of trait objects, and how they compare to generic functions, below.

Trait Objects

There are two ways of using traits to write polymorphic code in Rust: trait objects and generics. We'll present trait objects first and turn to generics in the next section.

Rust doesn't permit variables of type `dyn Write`:

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: dyn Write = buf;  // error: `Write` does not have a constant size
```

A variable's size has to be known at compile time, and types that implement `Write` can be any size.

This may be surprising if you're coming from C# or Java, but the reason is simple. In Java, a variable of type `OutputStream` (the Java standard interface analogous to `std::io::Write`) is a reference to any object that implements `OutputStream`. The fact that it's a reference goes without saying. It's the same with interfaces in C# and most other languages.

What we want in Rust is the same thing, but in Rust, references are explicit:

```
let mut buf: Vec<u8> = vec![];
let writer: &mut dyn Write = &mut buf;  // ok
```

A reference to a trait type, like `writer`, is called a *trait object*. Like any other reference, a trait object points to some value, it has a lifetime, and it can be either `mut` or shared.

What makes a trait object different is that Rust usually doesn't know the type of the referent at compile time. So a trait object includes a little extra information about the referent's type. This is strictly for Rust's own use behind the scenes: when you call `writer.write(data)`, Rust needs the type information to dynamically call the right `write` method depending on the type of `*writer`. You can't query the type information directly, and Rust does not support downcasting from the trait object `&mut dyn Write` back to a concrete type like `Vec<u8>`.

TRAIT OBJECT LAYOUT

In memory, a trait object is a fat pointer consisting of a pointer to the value, plus a pointer to a table representing that value's type. Each trait object therefore takes up two machine words, as shown in Figure 11-1.

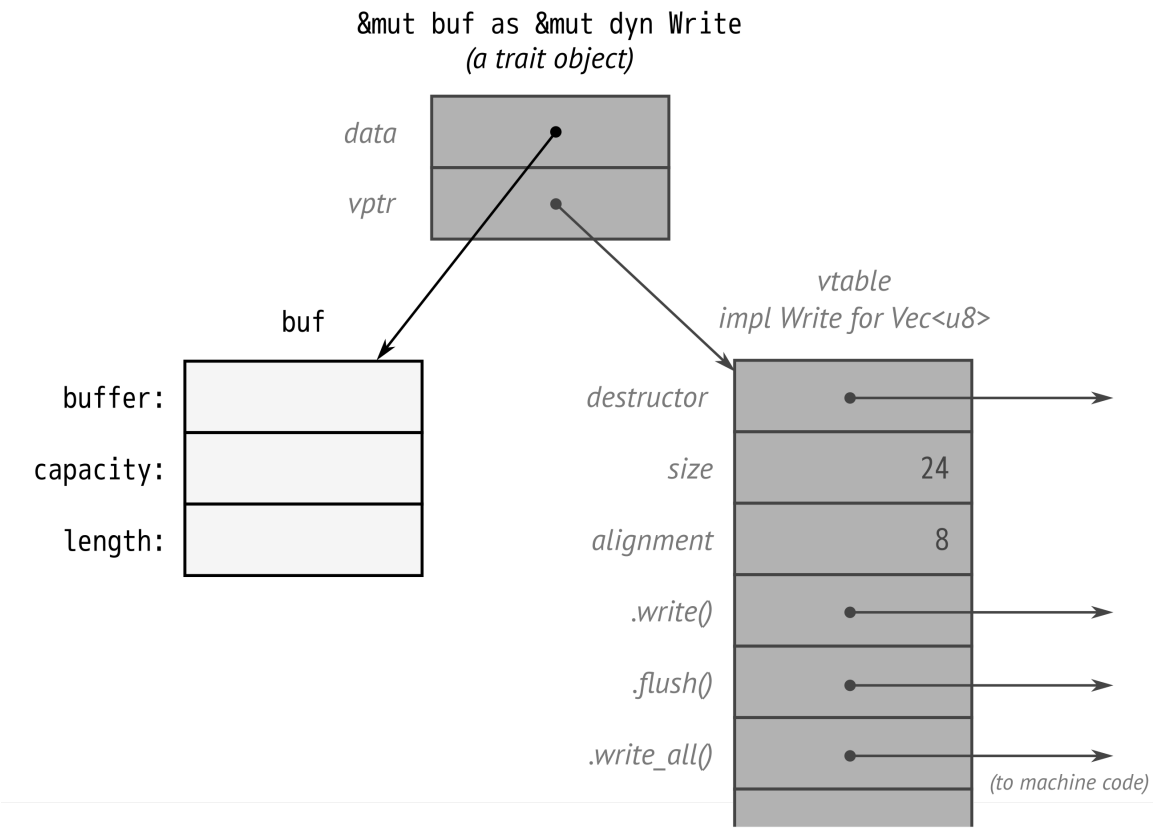

Figure 11-1. Trait objects in memory

C++ has this kind of run-time type information as well. It's called a *virtual table*, or *vtable*. In Rust, as in C++, the vtable is generated once, at compile time, and shared by all objects of the same type. Everything shown in dark gray in Figure 11-1, including the vtable, is a private implementation detail of Rust. Again, these aren't fields and data structures that you can access directly. Instead, the language automatically uses the vtable when you call a method of a trait object, to determine which implementation to call.

Seasoned C++ programmers will notice that Rust and C++ use memory a bit differently. In C++, the vtable pointer, or *vptr*, is stored as part of the struct. Rust uses fat pointers instead. The struct itself contains nothing but its fields. This way, a struct can implement dozens of traits without containing dozens of vptrs. Even types like `i32`, which aren't big enough to accommodate a vptr, can implement traits.

Rust automatically converts ordinary references into trait objects when needed. This is why we're able to pass `&mut local_file` to `say_hello` in this example:

```
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?;
```

The type of `&mut local_file` is `&mut File`, and the type of the argument to `say_hello` is `&mut dyn Write`. Since a `File` is a kind of writer, Rust allows this, automatically converting the plain reference to a trait object.

Likewise, Rust will happily convert a `Box<File>` to a `Box<dyn Write>`, a value that owns a writer in the heap:

```
let w: Box<dyn Write> = Box::new(local_file);
```

`Box<dyn Write>`, like `&mut dyn Write`, is a fat pointer: it contains the address of the writer itself and the address of the vtable. The same goes for other pointer types, like `Rc<dyn Write>`.

This kind of conversion is the only way to create a trait object. What the computer is actually doing here is very simple. At the point where the conversion happens, Rust knows the referent's true type (in this case, `File`), so it just adds the address of the appropriate vtable, turning the regular pointer into a fat pointer.

Generic Functions and Type Parameters

At the beginning of this chapter, we showed a `say_hello()` function that took a trait object as an argument. Let's rewrite that function as a generic function:

```
fn say_hello<W: Write>(out: &mut W) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

Only the type signature has changed:

```
fn say_hello(out: &mut dyn Write)     // plain function

fn say_hello<W: Write>(out: &mut W)   // generic function
```

The phrase `<W: Write>` is what makes the function generic. This is a *type parameter*. It means that throughout the body of this function, `W` stands for some type that implements the `Write` trait. Type parameters are usually single uppercase letters, by convention.

Which type `W` stands for depends on how the generic function is used:

```
say_hello(&mut local_file)?;  // calls say_hello::<File>
say_hello(&mut bytes)?;       // calls say_hello::<Vec<u8>>
```

When you pass `&mut local_file` to the generic `say_hello()` function, you're calling `say_hello::<File>()`. Rust generates machine code for this function that calls `File::write_all()` and `File::flush()`. When you pass `&mut bytes`, you're calling `say_hello::<Vec<u8>>()`. Rust gen-

erates separate machine code for this version of the function, calling the corresponding `Vec<u8>` methods. In both cases, Rust infers the type `W` from the type of the argument. This process is known as *monomorphization*, and the compiler handles it all automatically.

You can always spell out the type parameters:

```
say_hello::<File>(&mut local_file)?;
```

but it's seldom necessary, because Rust can usually deduce the type parameters by looking at the arguments. Here, the `say_hello` generic function expects a `&mut W` argument, and we're passing it a `&mut File`, so Rust infers that `W = File`.

If the generic function you're calling doesn't have any arguments that provide useful clues, you may have to spell it out:

```
// calling a generic method collect<C>() that takes no arguments
let v1 = (0 .. 1000).collect();  // error: can't infer type
let v2 = (0 .. 1000).collect::<Vec<i32>>(); // ok
```

Sometimes we need multiple abilities from a type parameter. For example, if we want to print out the top ten most common values in a vector, we'll need for those values to be printable:

```
use std::fmt::Debug;

fn top_ten<T: Debug>(values: &Vec<T>) { ... }
```

But this isn't good enough. How are we planning to determine which values are the most common? The usual way is to use the values as keys in a hash table. That means the values need to support the `Hash` and `Eq` operations. The bounds on `T` must include these as well as `Debug`. The syntax for this uses the `+` sign:

```
use std::hash::Hash;
use std::fmt::Debug;

fn top_ten<T: Debug + Hash + Eq>(values: &Vec<T>) { ... }
```

Some types implement `Debug`, some implement `Hash`, some support `Eq`; and a few, like `u32` and `String`, implement all three, as shown in Figure 11-2.

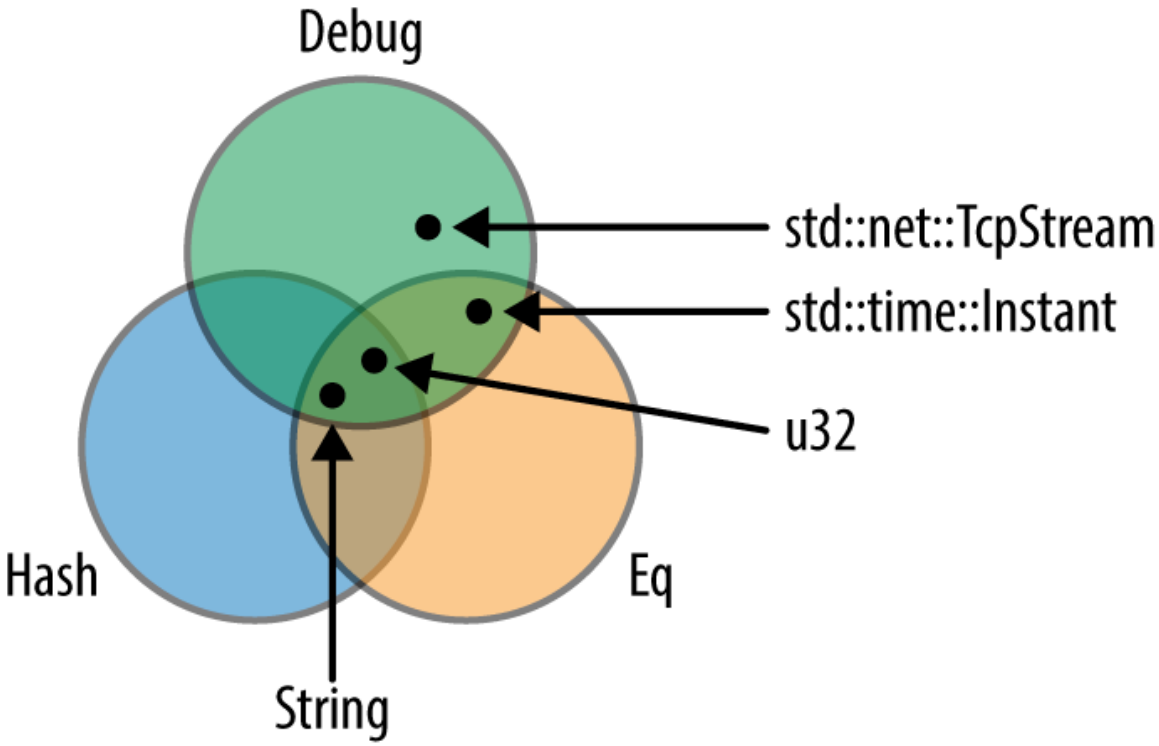

Figure 11-2. Traits as sets of types

It's also possible for a type parameter to have no bounds at all, but you can't do much with a value if you haven't specified any bounds for it. You can move it. You can put it into a box or vector. That's about it.

Generic functions can have multiple type parameters:

```
/// Run a query on a large, partitioned data set.
/// See <http://research.google.com/archive/mapreduce.html>.
fn run_query<M: Mapper + Serialize, R: Reducer + Serialize>(
    data: &DataSet, map: M, reduce: R) -> Results
{ ... }
```

As this example shows, the bounds can get to be so long that they are hard on the eyes. Rust provides an alternative syntax using the keyword `where`:

```
fn run_query<M, R>(data: &DataSet, map: M, reduce: R) -> Results
    where M: Mapper + Serialize,
          R: Reducer + Serialize
{ ... }
```

The type parameters `M` and `R` are still declared up front, but the bounds are moved to separate lines. This kind of `where` clause is also allowed on generic structs, enums, type aliases, and methods—anywhere bounds are permitted.

Of course, an alternative to `where` clauses is to keep it simple: find a way to write the program without using generics quite so intensively.

"Receiving References as Function Arguments" introduced the syntax for lifetime parameters. A generic function can have both lifetime parameters and type parameters. Lifetime parameters come first.

```
/// Return a reference to the point in `candidates` that's
/// closest to the `target` point.

fn nearest<'t, 'c, P>(target: &'t P, candidates: &'c [P]) -> &'c P
    where P: MeasureDistance
{
    ...
}
```

This function takes two arguments, `target` and `candidates`. Both are references, and we give them distinct lifetimes `'t` and `'c` (as discussed in "Distinct Lifetime Parameters"). Furthermore, the function works with any type `P` that implements the `MeasureDistance` trait, so we might use it on `Point2d` values in one program and `Point3d` values in another.

Lifetimes never have any impact on machine code. Two calls to `nearest()` using the same type `P`, but different lifetimes, will call the same compiled function. Only differing types cause Rust to compile multiple copies of a generic function.

Of course, functions are not the only kind of generic code in Rust.

- We've already covered generic types in "Generic Structs" and "Generic Enums".
- An individual method can be generic, even if the type it's defined on is not generic:

```
impl PancakeStack {
    fn push<T: Topping>(&mut self, goop: T) -> PancakeResult<()> {
        goop.pour(&self);
        self.absorb_topping(goop)
    }
}
```

- Type aliases can be generic, too:

```
type PancakeResult<T> = Result<T, PancakeError>;
```

- We'll cover generic traits later in this chapter.

All the features introduced in this section—bounds, `where` clauses, lifetime parameters, and so forth—can be used on all generic items, not just functions.

Which to Use

The choice of whether to use trait objects or generic code is subtle. Since both features are based on traits, they have a lot in common.

Trait objects are the right choice whenever you need a collection of values of mixed types, all together. While is technically possible to make generic salad:

```
trait Vegetable {
    ...
}

struct Salad<V: Vegetable> {
    veggies: Vec<V>
}
```

this is a rather severe design. Each such salad consists entirely of a single type of vegetable. Not everyone is cut out for this sort of thing. One of your authors once paid $14 for a `Salad<IcebergLettuce>` and has never quite gotten over the experience.

How can we build a better salad? Since `Vegetable` values can be all different sizes, we can't ask Rust for a `Vec<dyn Vegetable>`:

```
struct Salad {
    veggies: Vec<dyn Vegetable>  // error: `dyn Vegetable` does
                                 // not have a constant size
}
```

Trait objects are the solution:

```
struct Salad {
    veggies: Vec<Box<dyn Vegetable>>
}
```

Each `Box<dyn Vegetable>` can own any type of vegetable, but the box itself has a constant size—two pointers—suitable for storing in a vector. Apart from the unfortunate mixed metaphor of having boxes in one's food, this is precisely what's called for, and it would work out just as well for shapes in a drawing app, monsters in a game, pluggable routing algorithms in a network router, and so on.

Another possible reason to use trait objects is to reduce the total amount of compiled code. Rust may have to compile a generic function many times, once for each type it's used with. This could make the binary large, a phenomenon called *code bloat* in C++ circles. These days, memory is plentiful, and most of us have the luxury of ignoring code size; but constrained environments do exist.

Outside of situations involving salad or low-resource environments, generics have three important advantages over trait objects, with the result that in Rust, generics are the more common choice.

The first advantage is speed. Note the absence of the `dyn` keyword in generic function signatures. Because you specify the types at compile time, either explicitly or through type inference, the compiler knows exactly which `write` method to call. The `dyn` keyword isn't used because there are no trait objects - and thus no dynamic dispatch - involved.

The generic `min()` function shown in the introduction is just as fast as if we had written separate functions `min_u8`, `min_i64`, `min_string`, and so on. The compiler can inline it, like any other function, so in a release build, a call to `min::<i32>` is likely just two or three instructions. A call with constant arguments, like `min(5, 3)`, will be even faster: Rust can evaluate it at compile time, so that there's no run-time cost at all.

Or consider this generic function call:

```
let mut sink = std::io::sink();
say_hello(&mut sink)?;
```

`std::io::sink()` returns a writer of type `Sink` that quietly discards all bytes written to it.

When Rust generates machine code for this, it could emit code that calls `Sink::write_all`, checks for errors, then calls `Sink::flush`. That's what the body of the generic function says to do.

Or, Rust could look at those methods and realize the following:

- `Sink::write_all()` does nothing.
- `Sink::flush()` does nothing.
- Neither method ever returns an error.

In short, Rust has all the information it needs to optimize away this function call entirely.

Compare that to the behavior with trait objects. Rust never knows what type of value a trait object points to until run time. So even if you pass a `Sink`, the overhead of calling virtual methods and checking for errors still applies.

The second advantage of generics is that not every trait can support trait objects. Traits support several features, such as associated functions, that work only with generics: they rule out trait objects entirely. We'll point out these features as we come to them.

The third advantage of generics is that it's easy to bound a generic type parameter with several traits at once, as our `top_ten` function did when it required its `T` parameter to implement `Debug + Hash + Eq`. Trait objects can't do this: types like `&mut (dyn Debug + Hash + Eq)` aren't supported in Rust. (You can work around this with subtraits, defined later in this chapter, but it's a bit involved.)

Defining and Implementing Traits

Defining a trait is simple. Give it a name and list the type signatures of the trait methods. If we're writing a game, we might have a trait like this:

```
/// A trait for characters, items, and scenery -
/// anything in the game world that's visible on screen.
trait Visible {
    /// Render this object on the given canvas.
    fn draw(&self, canvas: &mut Canvas);

    /// Return true if clicking at (x, y) should
    /// select this object.
    fn hit_test(&self, x: i32, y: i32) -> bool;
}
```

To implement a trait, use the syntax `impl TraitName for Type`:

```
impl Visible for Broom {
    fn draw(&self, canvas: &mut Canvas) {
        for y in self.y - self.height - 1 .. self.y {
            canvas.write_at(self.x, y, '|');
        }
        canvas.write_at(self.x, self.y, 'M');
    }

    fn hit_test(&self, x: i32, y: i32) -> bool {
        self.x == x
        && self.y - self.height - 1 <= y
        && y <= self.y
    }
}
```

Note that this `impl` contains an implementation for each method of the `Visible` trait, and nothing else. Everything defined in a trait `impl` must actually be a feature of the trait; if we wanted to add a helper method in support of `Broom::draw()`, we would have to define it in a separate `impl` block:

```
impl Broom {
    /// Helper function used by Broom::draw() below.
    fn broomstick_range(&self) -> Range<i32> {
        self.y - self.height - 1 .. self.y
    }
}
```

These helper functions can be used within the trait `impl` blocks:

```
impl Visible for Broom {
    fn draw(&self, canvas: &mut Canvas) {
        for y in self.broomstick_range() {
            ...
        }
        ...
    }
    ...
}
```

Default Methods

The `Sink` writer type we discussed earlier can be implemented in a few lines of code. First, we define the type:

```
/// A Writer that ignores whatever data you write to it.
pub struct Sink;
```

`Sink` is an empty struct, since we don't need to store any data in it. Next, we provide an implementation of the `Write` trait for `Sink`:

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

impl Write for Sink {
    fn write(&mut self, buf: &[u8]) -> Result<usize> {
        // Claim to have successfully written the whole buffer.
        Ok(buf.len())
    }

    fn flush(&mut self) -> Result<()> {
        Ok(())
    }
}
```

So far, this is very much like the `Visible` trait. But we have also seen that the `Write` trait has a `write_all` method:

```
    let mut out = Sink;
    out.write_all(b"hello world\n")?;
```

Why does Rust let us `impl Write for Sink` without defining this method? The answer is that the standard library's definition of the `Write` trait contains a *default implementation* for `write_all`:

```
    trait Write {
        fn write(&mut self, buf: &[u8]) -> Result<usize>;
        fn flush(&mut self) -> Result<()>;

        fn write_all(&mut self, buf: &[u8]) -> Result<()> {
            let mut bytes_written = 0;
            while bytes_written < buf.len() {
                bytes_written += self.write(&buf[bytes_written..])?;
            }
            Ok(())
        }

        ...
    }
```

The `write` and `flush` methods are the basic methods that every writer must implement. A writer may also implement `write_all`, but if not, the default implementation shown above will be used.

Your own traits can include default implementations using the same syntax.

The most dramatic use of default methods in the standard library is the `Iterator` trait, which has one required method (`.next()`) and dozens of default methods. Chapter 15 explains why.

Traits and Other People's Types

Rust lets you implement any trait on any type, as long as either the trait or the type is introduced in the current crate.

This means that any time you want to add a method to any type, you can use a trait to do it:

```
    trait IsEmoji {
        fn is_emoji(&self) -> bool;
    }

    /// Implement IsEmoji for the built-in character type.
    impl IsEmoji for char {
```

```
    fn is_emoji(&self) -> bool {
        ...
    }
}

assert_eq!('$'.is_emoji(), false);
```

Like any other trait method, this new `is_emoji` method is only visible when `IsEmoji` is in scope.

The sole purpose of this particular trait is to add a method to an existing type, `char`. This is called an *extension trait*. Of course, you can add this trait to types, too, by writing `impl IsEmoji for str { ... }` and so forth.

You can even use a generic `impl` block to add an extension trait to a whole family of types at once. This trait could be implemented on any type:

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

/// Trait for values to which you can send HTML.
trait WriteHtml {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()>;
}
```

Implementing the trait for all writers makes it an extension trait, adding a method to all Rust writers:

```
/// You can write HTML to any std::io writer.
impl<W: Write> WriteHtml for W {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()> {
        ...
    }
}
```

The line `impl<W: Write> WriteHtml for W` means "for every type `W` that implements `Write`, here's an implementation of `WriteHtml` for `W`."

The `serde` library offers a nice example of how useful it can be to implement user-defined traits on standard types. `serde` is a serialization library. That is, you can use it to write Rust data structures to disk and reload them later. The library defines a trait, `Serialize`, that's implemented for every data type the library supports. So in the `serde` source code, there is code implementing `Serialize` for `bool`, `i8`, `i16`, `i32`, array and tuple types, and so on, through all the standard data structures like `Vec` and `HashMap`.

The upshot of all this is that `serde` adds a `.serialize()` method to all these types. It can be used like this:

```
use serde::Serialize;
use serde_json;

pub fn save_configuration(config: &HashMap<String, String>)
    -> std::io::Result<()>
{
    // Create a JSON serializer to write the data to a file.
    let writer = File::create(config_filename())?;
    let mut serializer = serde_json::Serializer::new(writer);

    // The serde `.serialize()` method does the rest.
    config.serialize(&mut serializer)?;

    Ok(())
}
```

We said earlier that when you implement a trait, either the trait or the type must be new in the current crate. This is called the *orphan rule*. It helps Rust ensure that trait implementations are unique. Your code can't `impl Write for u8`, because both `Write` and `u8` are defined in the standard library. If Rust let crates do that, there could be multiple implementations of `Write` for `u8`, in different crates, and Rust would have no reasonable way to decide which implementation to use for a given method call.

(C++ has a similar uniqueness restriction: the One Definition Rule. In typical C++ fashion, it isn't enforced by the compiler, except in the simplest cases, and you get undefined behavior if you break it.)

Self in Traits

A trait can use the keyword `Self` as a type. The standard `Clone` trait, for example, looks like this (slightly simplified):

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

Using `Self` as the return type here means that the type of `x.clone()` is the same as the type of `x`, whatever that might be. If `x` is a `String`, then the type of `x.clone()` is `String`—not `dyn Clone` or any other cloneable type.

Likewise, if we define this trait:

```
pub trait Spliceable {
    fn splice(&self, other: &Self) -> Self;
}
```

with two implementations:

```
impl Spliceable for CherryTree {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}

impl Spliceable for Mammoth {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}
```

then inside the first `impl`, `Self` is simply an alias for `CherryTree`, and in the second, it's an alias for `Mammoth`. This means that we can splice together two cherry trees or two mammoths, not that we can create a mammoth-cherry hybrid. The type of `self` and the type of `other` must match.

A trait that uses the `Self` type is incompatible with trait objects:

```
// error: the trait `Spliceable` cannot be made into an object
fn splice_anything(left: &dyn Spliceable, right: &dyn Spliceable) {
    let combo = left.splice(right);
    // ...
}
```

The reason is something we'll see again and again as we dig into the advanced features of traits. Rust rejects this code because it has no way to type-check the call `left.splice(right)`. The whole point of trait objects is that the type isn't known until run time. Rust has no way to know at compile time if `left` and `right` will be the same type, as required.

Trait objects are really intended for the simplest kinds of traits, the kinds that could be implemented using interfaces in Java or abstract base classes in C++. The more advanced features of traits are useful, but they can't coexist with trait objects because with trait objects, you lose the type information Rust needs to type-check your program.

Now, had we wanted genetically improbable splicing, we could have designed a trait-object-friendly trait:

```
pub trait MegaSpliceable {
    fn splice(&self, other: &dyn MegaSpliceable) -> Box<dyn MegaSpliceable>;
}
```

This trait is compatible with trait objects. There's no problem type-checking calls to this `.splice()` method because the type of the argument `other` is not required to match the type of `self`, as long as both

types are `MegaSpliceable`.

Subtraits

We can declare that a trait is an extension of another trait:

```
/// Someone in the game world, either the player or some other
/// pixie, gargoyle, squirrel, ogre, etc.
trait Creature: Visible {
    fn position(&self) -> (i32, i32);
    fn facing(&self) -> Direction;
    ...
}
```

The phrase `trait Creature: Visible` means that all creatures are visible. Every type that implements `Creature` must also implement the `Visible` trait:

```
impl Visible for Broom {
    ...
}

impl Creature for Broom {
    ...
}
```

We can implement the two traits in either order, but it's an error to implement `Creature` for a type without also implementing `Visible`. Here, we say that `Creature` is a *subtrait* of `Visible`, and that `Creature` is `Visible`'s *supertrait*.

Subtraits resemble subinterfaces in Java or C#, in that users can assume that any value that implements a subtrait implements its supertrait as well. But in Rust, a subtrait does not inherit the associated items of its supertrait; each trait still needs to be in scope if you want to call its methods.

In fact, Rust's subtraits are really just a shorthand for a bound on `Self`. A definition of `Creature` like this is exactly equivalent the one above:

```
trait Creature where Self: Visible {
    ...
}
```

Type-Associated Functions

In most object-oriented languages, interfaces can't include static methods or constructors, but traits can include type-associated functions, Rust's analog to static methods:

```
trait StringSet {
    /// Return a new empty set.
    fn new() -> Self;

    /// Return a set that contains all the strings in `strings`.
    fn from_slice(strings: &[&str]) -> Self;

    /// Find out if this set contains a particular `value`.
    fn contains(&self, string: &str) -> bool;

    /// Add a string to this set.
    fn add(&mut self, string: &str);
}
```

Every type that implements the `StringSet` trait must implement these four associated functions. The first two, `new()` and `from_slice()`, don't take a `self` argument. They serve as constructors. In non-generic code, these functions can be called using `::` syntax, just like any other type-associated function:

```
// Create sets of two hypothetical types that impl StringSet:
let set1 = SortedStringSet::new();
let set2 = HashedStringSet::new();
```

In generic code, it's the same, except the type is often a type variable, as in the call to `S::new()` shown here:

```
/// Return the set of words in `document` that aren't in `wordlist`.
fn unknown_words<S: StringSet>(document: &[String], wordlist: &S) -> S {
    let mut unknowns = S::new();
    for word in document {
        if !wordlist.contains(word) {
            unknowns.add(word);
        }
    }
    unknowns
}
```

Like Java and C# interfaces, trait objects don't support type-associated functions. If you want to use `&dyn StringSet` trait objects, you must change the trait, adding the bound `where Self: Sized` to each associated function that doesn't take a `self` argument by reference:

```
trait StringSet {
    fn new() -> Self
```

```
        where Self: Sized;

    fn from_slice(strings: &[&str]) -> Self
        where Self: Sized;

    fn contains(&self, string: &str) -> bool;

    fn add(&mut self, string: &str);
}
```

This bound tells Rust that trait objects are excused from supporting this particular associated function. With these additions, `StringSet` trait objects are allowed; they still don't support `new` or `from_slice`, but you can create them and use them to call `.contains()` and `.add()`. The same trick works for any other method that is incompatible with trait objects. (We will forgo the rather tedious technical explanation of why this works, but the `Sized` trait is covered in Chapter 13.)

Fully Qualified Method Calls

All the ways for calling trait methods we've seen so far rely on Rust filling in some missing pieces for you. For example, if you write

```
"hello".to_string()
```

it's understood that `to_string` refers to the `to_string` method of the `ToString` trait, of which we're calling the `str` types's implementation. So there are four players in this game: the trait, the method of that trait, the implementation of that method, and the value to which that implementation is being applied. It's great that we don't have to spell all that out every time we want to call a method. But in some cases you need a way to say exactly what you mean. Fully qualified method calls fit the bill.

First of all, it helps to know that a method is just a special kind of function. These two calls are equivalent:

```
"hello".to_string()

str::to_string("hello")
```

The second form looks exactly like a associated function call. This works even though the `to_string` method takes a `self` argument. Simply pass `self` as the function's first argument.

Since `to_string` is a method of the standard `ToString` trait, there are two more forms you can use:

```
ToString::to_string("hello")

<str as ToString>::to_string("hello")
```

All four of these method calls do exactly the same thing. Most often, you'll just write `value.method()`. The other forms are *qualified* method calls. They specify the type or trait that a method is associated with. The last form, with the angle brackets, specifies both: a *fully qualified* method call.

When you write `"hello".to_string()`, using the `.` operator, you don't say exactly which `to_string` method you're calling. Rust has a method lookup algorithm that figures this out, depending on the types, deref coercions, and so on. With fully qualified calls, you can say exactly which method you mean, and that can help in a few odd cases:

- When two methods have the same name. The classic hokey example is the `Outlaw` with two `.draw()` methods from two different traits, one for drawing it on the screen and one for interacting with the law:

```
outlaw.draw();  // error: draw on screen or draw pistol?

Visible::draw(&outlaw);  // ok: draw on screen
HasPistol::draw(&outlaw);  // ok: corral
```

  Normally you're better off just renaming one of the methods, but sometimes you can't.

- When the type of the `self` argument can't be inferred:

```
let zero = 0;  // type unspecified; could be `i8`, `u8`, ...

zero.abs();  // error: can't call method `abs`
             // on ambiguous numeric type

i64::abs(zero);  // ok
```

- When using the function itself as a function value:

```
let words: Vec<String> =
    line.split_whitespace()  // iterator produces &str values
        .map(ToString::to_string)  // ok
        .collect();
```

- When calling trait methods in macros. We'll explain in Chapter 21.

Fully qualified syntax also works for associated functions. In the previous section, we wrote `S::new()` to create a new set in a generic function. We could also have written `StringSet::new()` or `<S as StringSet>::new()`.

Traits That Define Relationships Between Types

So far, every trait we've looked at stands alone: a trait is a set of methods that types can implement. Traits can also be used in situations where there are multiple types that have to work together. They can describe relationships between types.

- The `std::iter::Iterator` trait relates each iterator type with the type of value it produces.
- The `std::ops::Mul` trait relates types that can be multiplied. In the expression `a * b`, the values `a` and `b` can be either the same type, or different types.
- The `rand` crate includes both a trait for random number generators (`rand::Rng`), and a trait for types that can be randomly generated (`rand::Distribution`). The traits themselves define exactly how these types work together.

You won't need to create traits like these every day, but you'll come across them throughout the standard library and in third-party crates. In this section, we'll show how each of these examples is implemented, picking up relevant Rust language features as we need them. The key skill here is the ability to read traits and method signatures and figure out what they say about the types involved.

Associated Types (or How Iterators Work)

We'll start with iterators. By now every object-oriented language has some sort of built-in support for iterators, objects that represent the traversal of some sequence of values.

Rust has a standard `Iterator` trait, defined like this:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

The first feature of this trait, `type Item;`, is an *associated type*. Each type that implements `Iterator` must specify what type of item it produces.

The second feature, the `next()` method, uses the associated type in its return value. `next()` returns an `Option<Self::Item>`: either `Some(item)`, the next value in the sequence, or `None` when there are no more values to visit. The type is written as `Self::Item`, not just plain `Item`, because `Item` is a feature of each type of iterator, not a standalone type. As always, `self` and the `Self` type show up explicitly in the code everywhere their fields, methods, and so on are used.

Here's what it looks like to implement `Iterator` for a type:

```
// (code from the std::env standard library module)
impl Iterator for Args {
    type Item = String;

    fn next(&mut self) -> Option<String> {
        ...
    }
    ...
}
```

`std::env::Args` is the type of iterator returned by the standard library function `std::env::args()` that we used in Chapter 2 to access command-line arguments. It produces `String` values, so the `impl` declares `type Item = String;`.

Generic code can use associated types:

```
/// Loop over an iterator, storing the values in a new vector.
fn collect_into_vector<I: Iterator>(iter: I) -> Vec<I::Item> {
    let mut results = Vec::new();
    for value in iter {
        results.push(value);
    }
    results
}
```

Inside the body of this function, Rust infers the type of `value` for us, which is nice; but we must spell out the return type of `collect_into_vector`, and the `Item` associated type is the only way to do that. (`Vec<I>` would be simply wrong: we would be claiming to return a vector of iterators!)

The preceding example is not code that you would write out yourself, because after reading Chapter 15, you'll know that iterators already have a standard method that does this: `iter.collect()`. So let's look at one more example before moving on.

```
/// Print out all the values produced by an iterator
fn dump<I>(iter: I)
    where I: Iterator
{
    for (index, value) in iter.enumerate() {
        println!("{}: {:?}", index, value);   // error
    }
}
```

This almost works. There is just one problem: `value` might not be a printable type.

```
error: `<I as Iterator>::Item` doesn't implement `Debug`
  |
8 |         println!("{}: {:?}", index, value);   // error
  |                                     ^^^^^
  |                           `<I as Iterator>::Item` cannot be formatted
  |                           using `{:?}` because it doesn't implement `Debug`
  |
  = help: the trait `Debug` is not implemented for `<I as Iterator>::Item`
  = note: required by `std::fmt::Debug::fmt`
help: consider further restricting the associated type
  |
5 |     where I: Iterator, <I as Iterator>::Item: Debug
  |                        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

The error message is slightly obfuscated by Rust's use of the syntax `<I as Iterator>::Item`, which is an explicit but verbose way of saying `I::Item`. This is valid Rust syntax, but you'll rarely actually need to write a type out that way.

The gist of the error message is that to make this generic function compile, we must ensure that `I::Item` implements the `Debug` trait, the trait for formatting values with `{:?}`. As the error message suggests, we can do this by placing a bound on `I::Item`:

```
use std::fmt::Debug;

fn dump<I>(iter: I)
    where I: Iterator, I::Item: Debug
{
    ...
}
```

Or, we could write, "`I` must be an iterator over `String` values":

```
fn dump<I>(iter: I)
    where I: Iterator<Item=String>
{
    ...
}
```

`Iterator<Item=String>` is itself a trait. If you think of `Iterator` as the set of all iterator types, then `Iterator<Item=String>` is a subset of `Iterator`: the set of iterator types that produce `String`s. This syntax can be used anywhere the name of a trait can be used, including trait object types:

```
fn dump(iter: &mut dyn Iterator<Item=String>) {
    for (index, s) in iter.enumerate() {
        println!("{}: {:?}", index, s);
```

```
        }
    }
```

Traits with associated types, like `Iterator`, are compatible with trait methods, but only if all the associated types are spelled out, as shown here. Otherwise, the type of `s` could be anything, and again, Rust would have no way to type-check this code.

We've shown a lot of examples involving iterators. It's hard not to; they're by far the most prominent use of associated types. But associated types are generally useful whenever a trait needs to cover more than just methods.

- In a thread pool library, a `Task` trait, representing a unit of work, could have an associated `Output` type.
- A `Pattern` trait, representing a way of searching a string, could have an associated `Match` type, representing all the information gathered by matching the pattern to the string.

  ```
  trait Pattern {
      type Match;

      fn search(&self, string: &str) -> Option<Self::Match>;
  }

  /// You can search a string for a particular character.
  impl Pattern for char {
      /// A "match" is just the location where the
      /// character was found.
      type Match = usize;

      fn search(&self, string: &str) -> Option<usize> {
          ...
      }
  }
  ```

  If you're familiar with regular expressions, it's easy to see how `impl Pattern for RegExp` would have a more elaborate `Match` type, probably a struct that would include the start and length of the match, the locations where parenthesized groups matched, and so on.

- A library for working with relational databases might have a `DatabaseConnection` trait with associated types representing transactions, cursors, prepared statements, and so on.

Associated types are perfect for cases where each implementation has *one* specific related type: each type of `Task` produces a particular type of `Output`; each type of `Pattern` looks for a particular type of `Match`. However, as we'll see, some relationships among types are not like this.

Generic Traits (or How Operator Overloading Works)

Multiplication in Rust uses this trait:

```
/// std::ops::Mul, the trait for types that support `*`.
pub trait Mul<RHS> {
    /// The resulting type after applying the `*` operator
    type Output;

    /// The method for the `*` operator
    fn mul(self, rhs: RHS) -> Self::Output;
}
```

`Mul` is a generic trait. The type parameter, `RHS`, is short for *right-hand side*.

The type parameter here means the same thing that it means on a struct or function: `Mul` is a generic trait, and its instances `Mul<f64>`, `Mul<String>`, `Mul<Size>`, etc. are all different traits, just as `min::<i32>` and `min::<String>` are different functions and `Vec<i32>` and `Vec<String>` are different types.

A single type—say, `WindowSize`—can implement both `Mul<f64>` and `Mul<i32>`, and many more. You would then be able to multiply a `WindowSize` by many other types. Each implementation would have its own associated `Output` type.

Generic traits get a special dispensation when it comes to the orphan rule: you can implement a foreign trait for a foreign type, so long as one of the trait's type parameters is a type defined in the current crate. So, if you've defined `WindowSize` yourself, you can implement `Mul<WindowSize>` for `f64`, even though you didn't define either `Mul` or `f64`. These implementations can even be generic, such as `impl<T> Mul<WindowSize> for Vec<T>`. This works because there's no way any other crate could define `Mul<WindowSize>` on anything, and thus no way a conflict among implementations could arise. (We introduced the orphan rule back in "Traits and Other People's Types".) This is how crates like `nalgebra` define arithmetic operations on vectors.

The trait shown above is missing one minor detail. The real `Mul` trait looks like this:

```
pub trait Mul<RHS=Self> {
    ...
}
```

The syntax `RHS=Self` means that `RHS` defaults to `Self`. If I write `impl Mul for Complex`, without specifying `Mul`'s type parameter, it means `impl Mul<Complex> for Complex`. In a bound, if I write `where T: Mul`, it means `where T: Mul<T>`.

In Rust, the expression `lhs * rhs` is shorthand for `Mul::mul(lhs, rhs)`. So overloading the `*` operator in Rust is as simple as implementing the `Mul` trait. We'll show examples in the next chapter.

impl Trait

As you might imagine, combinations of many generic types can get messy. For example, combining just a few iterators using standard library combinators rapidly turns your return type into an eyesore.

```
use std::iter;
use std::vec::IntoIter;
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) ->
    iter::Cycle<iter::Chain<IntoIter<u8>, IntoIter<u8>>> {
        v.into_iter().chain(u.into_iter()).cycle()
}
```

We could easily replace this hairy return type with a trait object:

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> Box<dyn Iterator<Item=u8>> {
    Box::new(v.into_iter().chain(u.into_iter()).cycle())
}
```

However, taking the overhead of dynamic dispatch and an unavoidable heap allocation every time this function is called just to avoid an ugly type signature doesn't seem like a good trade, in most cases.

Rust has a feature called `impl Trait` designed for precisely this situation. `impl Trait` allows us to "erase" the type of a return value, specifying only the trait or traits it implements, without dynamic dispatch or a heap allocation.

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> impl Iterator<Item=u8> {
    v.into_iter().chain(u.into_iter()).cycle()
}
```

Now, rather than specifying a particular nested type of iterator combinator structs, `cyclical_zip`'s signature just states that it returns some kind of iterator over `u8`. The return type expresses the intent of the function, rather than its implementation details.

This has definitely cleaned up the code and made it more readable, but `impl Trait` is more than just a convenient shorthand. Using `impl Trait` means that you can change the actual type being returned in the future as long as it still implements `Iterator<Item=u8>`, and any code calling the function will continue to compile without an issue. This provides a lot of flexibility for library authors, because only the relevant functionality is encoded in the type signature.

For example, if the first version of a library uses iterator combinators as above, but a better algorithm for the same process is discovered, the library author can use different combinators or even make a custom type that implements `Iterator` and users of the library can get the performance improvements without changing their code at all.

It might be tempting to use `impl Trait` to approximate a statically-dispatched version of the factory pattern that's commonly used in object oriented languages. For example, you might define a trait like this:

```
trait Shape {
    fn new() -> Self;
    fn area(&self) -> f64;
}
```

After implementing it for a few types, you might want to different `Shape`s depending on a runtime value, like a string that a user enters. This doesn't work with `impl Shape` as the return type:

```
fn make_shape(shape: &str) -> impl Shape {
    match shape {
        "circle" => Circle::new(),
        "triangle" => Triangle::new(), // error: incompatible types
        "shape" => Rectangle::new(),
    }
}
```

From the perspective of the caller, a function like this doesn't make much sense. `impl Trait` is a form of static dispatch, so the compiler has to know the type being returned from the function at compile time in order to allocate the right amount of space on the stack and correctly access fields and methods on that type. Here, it could be `Circle`, `Triangle`, or `Rectangle`, which could all take up different amounts of space and all have different implementations of `area()`.

It's important to note that Rust doesn't allow trait methods to use `impl Trait` return values. Supporting this will require some improvements in the languages's type system. Until that work is done, only free functions and functions associated with specific types can use `impl Trait` returns.

`impl Trait` can also be used in functions that take generic arguments. For instance, this simple generic function:

```
fn print<T: Display>(val: T) {
    println!("{}", val);
}
```

is identical to this version using `impl Trait`:

```
fn print(val: impl Display) {
    println!("{}", val);
}
```

with one important exception. Using generics allows callers of the function to specify the type of the generic arguments, like `print::<i32>(42)`, while using `impl Trait` does not.

Each `impl Trait` argument is assigned its own anonymous type parameter, so `impl Trait` for arguments is limited to only the simplest generic functions, with no relationships between the types of arguments.

Associated Consts

Like structs and enums, traits can have associated constants. You can declare a trait with an associated constant using the same syntax as for a struct or enum:

```
trait Greet {
    const GREETING: &'static str = "Hello";
    fn greet(&self) -> String;
}
```

Associated consts in traits have a special power, though. Like associated types and functions, you can declare them but not give them a value.

```
trait Float {
    const ZERO: Self;
    const ONE: Self;
}
```

Then, implementors of the trait can define these values.

```
impl Float for f32 {
    const ZERO: f32 = 0.0;
    const ONE: f32 = 1.0;
}

impl Float for f64 {
    const ZERO: f64 = 0.0;
    const ONE: f64 = 1.0;
}
```

This allows you to write generic code that uses these values:

```
fn add_one<T: Float + Add<Output=T>>(value: T) -> T {
    value + T::ONE
}
```

Note that associated constants can't be used with trait objects, since the compiler relies on type information about the implementation in order to pick the right value at compile time.

Even a simple trait with no behavior at all, like `Float`, can give enough information about a type, in combination with a few operators, to implement common mathematical functions like Fibonacci.

```
fn fib<T: Float + Add<Output=T>>(n: usize) -> T {
    match n {
        0 => T::ZERO,
        1 => T::ONE,
        n => fib::<T>(n - 1) + fib::<T>(n - 2)
    }
}
```

In the last two sections, we've shown different ways traits can describe relationships between types. All of these can also be seen as ways of avoiding virtual method overhead and downcasts, since they allow Rust to know more concrete types at compile time.

Reverse-Engineering Bounds

Writing generic code can be a real slog when there's no single trait that does everything you need. Suppose we have written this non-generic function to do some computation:

```
fn dot(v1: &[i64], v2: &[i64]) -> i64 {
    let mut total = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Now we want to use the same code with floating-point values. We might try something like this:

```
fn dot<N>(v1: &[N], v2: &[N]) -> N {
    let mut total: N = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

No such luck: Rust complains about the use of `*` and the type of `0`. We can require `N` to be a type that supports `+` and `*` using the `Add` and `Mul` traits. Our use of `0` needs to change, though, because `0` is always an

integer in Rust; the corresponding floating-point value is `0.0`. Fortunately, there is a standard `Default` trait for types that have default values. For numeric types, the default is always 0.

```
use std::ops::{Add, Mul};

fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

This is closer, but still does not quite work:

```
error: mismatched types
  |
5 | fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
  |        - this type parameter
...
8 |         total = total + v1[i] * v2[i];
  |                         ^^^^^^^^^^^^^ expected type parameter `N`,
  |                                       found associated type
  |
  = note: expected type parameter `N`
            found associated type `<N as Mul>::Output`
help: consider further restricting this bound
  |
5 | fn dot<N: Add + Mul + Default + Mul<Output = N>>(v1: &[N], v2: &[N]) -> N {
  |                                 ^^^^^^^^^^^^^^^^
```

Our new code assumes that multiplying two values of type `N` produces another value of type `N`. This isn't necessarily the case. You can overload the multiplication operator to return whatever type you want. We need to somehow tell Rust that this generic function only works with types that have the normal flavor of multiplication, where multiplying `N * N` returns an `N`. The suggestion in the error message is *almost* right: we can do this by replacing `Mul` with `Mul<Output=N>`, and the same for `Add`:

```
fn dot<N: Add<Output=N> + Mul<Output=N> + Default>(v1: &[N], v2: &[N]) -> N
{
    ...
}
```

At this point, the bounds are starting to pile up, making the code hard to read. Let's move the bounds into a `where` clause:

```
fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default
{
    ...
}
```

Great. But Rust still complains about this line of code:

```
error: cannot move out of type `[N]`, a non-copy slice
  |
8 |         total = total + v1[i] * v2[i];
  |                         ^^^^^
  |                         |
  |                         cannot move out of here
  |                         move occurs because `v1[_]` has type `N`,
  |                         which does not implement the `Copy` trait
```

Since we haven't required `N` to be a copyable type, Rust interprets `v1[i]` as an attempt to move a value out of the slice, which is forbidden. But we don't want to modify the slice at all; we just want to copy the values out to operate on them. Fortunately, all of Rust's built-in numeric types implement `Copy`, so we can simply add that to our constraints on N:

```
where N: Add<Output=N> + Mul<Output=N> + Default + Copy
```

With this, the code compiles and runs. The final code looks like this:

```
use std::ops::{Add, Mul};

fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default + Copy
{

    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}

#[test]
fn test_dot() {
    assert_eq!(dot(&[1, 2, 3, 4], &[1, 1, 1, 1]), 10);
    assert_eq!(dot(&[53.0, 7.0], &[1.0, 5.0]), 88.0);
}
```

This occasionally happens in Rust: there is a period of intense arguing with the compiler, at the end of which the code looks rather nice, as if it had been a breeze to write, and runs beautifully.

What we've been doing here is reverse-engineering the bounds on `N`, using the compiler to guide and check our work. The reason it was a bit of a pain is that there wasn't a single `Number` trait in the standard library that included all the operators and methods we wanted to use. As it happens, there's a popular open source crate called `num` that defines such a trait! Had we known, we could have added `num` to our *Cargo.toml* and written:

```
use num::Num;

fn dot<N: Num + Copy>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::zero();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Just as in object-oriented programming, the right interface makes everything nice, in generic programming, the right trait makes everything nice.

Still, why go to all this trouble? Why didn't Rust's designers make the generics more like C++ templates, where the constraints are left implicit in the code, à la "duck typing?"

One advantage of Rust's approach is forward compatibility of generic code. You can change the implementation of a public generic function or method, and if you didn't change the signature, you haven't broken any of its users.

Another advantage of bounds is that when you do get a compiler error, at least the compiler can tell you where the trouble is. C++ compiler error messages involving templates can be much longer than Rust's, pointing at many different lines of code, because the compiler has no way to tell who's to blame for a problem: the template—or its caller, which might also be a template—or *that* template's caller...

Perhaps the most important advantage of writing out the bounds explicitly is simply that they are there, in the code and in the documentation. You can look at the signature of a generic function in Rust and see exactly what kind of arguments it accepts. The same can't be said for templates. The work that goes into fully documenting argument types in C++ libraries like Boost is even *more* arduous than what we went through here. The Boost developers don't have a compiler that checks their work.

Traits as a Foundation

Traits are one of the main organizing features in Rust, and with good reason. There's nothing better to design a program or library around than a good interface.

This chapter was a blizzard of syntax, rules, and explanations. Now that we've laid a foundation, we can start talking about the many ways traits and generics are used in Rust code. The fact is, we've only begun to scratch the surface. The next two chapters cover common traits provided by the standard library. Upcoming chapters cover closures, iterators, input/output, and concurrency. Traits and generics play a central role in all of these topics.

PREV
10. Enums and Patterns

NEXT
12. Operator Overloading