CHAPTER 20

Object-Oriented Programming

In this chapter, you will learn:

- How to implement destructors, that is, methods to run when an object is deallocated
- How to implement operator overloading, that is, methods that allow use of Rust operators with user-defined types
- How to implement infallible or fallible type converters
- Why Rust does not use data inheritance and how to do without it
- What are static dispatch and dynamic dispatch, how they are implemented, and when to use them

Peculiarities of Rust Object-Orientation

In previous chapters, we have already seen that Rust supports some concepts of object-oriented programming, like methods, privacy, and trait inheritance.

In this chapter–addressed mainly to those who already know the object-oriented paradigm–for any concept of object-oriented programming, the corresponding Rust idiom will be introduced.

C. Milanesi, *Beginning Rust*, https://doi.org/10.1007/978-1-4842-7208-4_20

Destructors

We already saw that types can have constructors. Well, they also have *destructors*. For a type, a destructor is that method that is automatically invoked on an object of that type, when that object is deallocated.

Let's see the purpose of destructors. If an object can reserve some system resources, like file handles or database connections, such resources should be automatically released when that object is destroyed, to avoid a bug named *resource leak*. Here is an example, in which actual reservation, use, and release of resources are replaced by prints on the console:

```
struct CommunicationChannel {
    address: String,
    port: u16,
}
impl Drop for CommunicationChannel {
    fn drop(&mut self) {
        println!("Closing port {}:{}", self.address, self.port);
    }
}
impl CommunicationChannel {
fn create(address: &str, port: u16) -> CommunicationChannel
    {
    println!("Opening port {}:{}", address, port);
        CommunicationChannel {
            address: address.to_string(),
            port: port,
        }
    }
    fn send(&self, msg: &str) {
        println!("Sent to {}:{} the message '{}'",
            self.address, self.port, msg);
    }
}
```

```
let channel_a = CommunicationChannel::create("usb4", 879);
channel_a.send("Message 1");
{
    let channel_b = CommunicationChannel::create("eth1", 12000);
    channel_b.send("Message 2");
}
channel_a.send("Message 3");
```

This program will print:

```
Opening port usb4:879
Sent to usb4:879 the message 'Message 1'
Opening port eth1:12000
Sent to eth1:12000 the message 'Message 2'
Closing port eth1:12000
Sent to usb4:879 the message 'Message 3'
Closing port usb4:879
```

The second statement implements the trait **`Drop`** for the newly declared type `CommunicationChannel`. Such trait, defined by the Rust language, has the peculiar property that its only method, named **`drop`**, is automatically invoked exactly when the object is deallocated, and therefore it is a *destructor*. In general, to create a destructor for a type, it is enough to implement the `Drop` trait for such type. You can implement the `Drop` trait only for the types declared inside your program, because that trait is not declared by your own code.

The third statement of the previous program is a block that defines two methods for our struct: the `create` constructor and the `send` operation.

At last, there is the application code, which uses the `CommunicationChannel` type. A communication channel is created, and such creation prints the first line of output. A message is sent, and that operation prints the second line. Then there is a nested block, in which another communication channel is created, printing the third line, and a message is sent through such channel, printing the fourth line.

So far, there is nothing new. But now, the nested block ends. That causes the `channel_b` variable to be destroyed, so its `drop` method is invoked, and that prints the fifth line.

Now, after the nested block has ended, another message is sent to the `channel_a` variable, causing it to print the last-but-one line. At last, the `channel_a` variable is destroyed, and that prints the last line.

As it will be seen in Chapter 22, in Rust, memory is already released by the language, and therefore there is no need to invoke functions similar to the `free` function of C language, or to the `delete` operator of C++ language. But other resources are not automatically released. Therefore destructors are most useful to release resources like file handles, communication handles, GUI windows, graphical resources, and synchronization primitives. If you use a library to handle such resources, that library should already contain a proper implementation of the `Drop` trait for any of its types that handle a resource.

Another use of destructors is to better understand how memory is managed, like this code shows:

```
struct S ( i32 );
impl Drop for S {
    fn drop(&mut self) {
        println!("Dropped {}", self.0);
    }
}
let _a = S (1);
let _b = S (2);
let _c = S (3);
{
    let _d = S (4);
    let _e = S (5);
    let _f = S (6);
    println!("INNER");
}
println!("OUTER");
```

It will print:

```
INNER
Dropped 6
Dropped 5
Dropped 4
```

```
OUTER
Dropped 3
Dropped 2
Dropped 1
```

Notice that objects are destroyed in exactly the opposite order than their declaration order, and just when they get out of their block.

This program is similar to the previous one, but its variables have been replaced by variable placeholders:

```
struct S ( i32 );
impl Drop for S {
    fn drop(&mut self) {
        println!("Dropped {}", self.0);
    }
}
let _ = S (1);
let _ = S (2);
let _ = S (3);
{
    let _ = S (4);
    let _ = S (5);
    let _ = S (6);
    println!("INNER");
}
println!("OUTER");
```

It will print:

```
Dropped 1
Dropped 2
Dropped 3
Dropped 4
Dropped 5
Dropped 6
INNER
OUTER
```

Variable placeholders do not allocate objects lasting for the entire block; they allocate temporary objects. Temporary objects are destroyed at the end of their statements, that is, when a semicolon in encountered. Therefore, all the objects are dropped in the same statement in which they are allocated.

This program is equivalent to the following one:

```
struct S ( i32 );
impl Drop for S {
    fn drop(&mut self) {
        println!("Dropped {}", self.0);
    }
}
S (1); S (2); S (3);
{
    S (4); S (5); S (6);
    println!("INNER");
}
println!("OUTER");
```

Operator Overloading

In many object-oriented languages, it is possible to redefine the language operators for the user-defined types. Also Rust allows us to redefine most of its operators. Though, differing from C++, some operators are not redefinable.

A typical use of operator overloading is for creating new numeric types not defined by the language, like the *complex* numbers of higher mathematics. Here, some features of these types of numbers will be implemented using stepwise refinement. The first draft is the following program:

```
struct Complex {
    re: f64,
    im: f64,
}

fn add_complex(lhs: Complex, rhs: Complex) -> Complex {
    Complex { re: lhs.re + rhs.re, im: lhs.im + rhs.im }
}
```

```
let z1 = Complex { re: 3.8, im: -2.1 };
let z2 = Complex { re: -1.5, im: 8.6 };
let z3 = add_complex(z1, z2);
print!("{} + {}i", z3.re, z3.im);
```

It will print: `2.3 + 6.5i`.

Complex numbers are actually a pair of real numbers: one item of the couple is named *real part*, *re* for short, and the other item is named *imaginary part*, *im* for short. The addition between any two complex numbers is defined to be a number whose real part is the sum of the real parts of the addends, and whose imaginary part is the sum of the imaginary parts of the addends. The words `lhs` and `rhs` stand, respectively, for *left-hand side* and *right-hand side* for the addition operation.

Now we want to replace the expression `add_complex(z1, z2)` with the expression `z1 + z2`. In Rust, the expression `a + b` is just syntax sugar for the call `a.add(b)`, where add is a function belonging to the `Add` trait, declared in the standard library. Here is the declaration of that trait:

```
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

Let's explain it. When you add two numbers, there may be three distinct types around: the type of the first addend, the type of the second addend, and the type of the result. They may be the same type, as is reasonable when we add two real numbers or two complex numbers, but in general there may be cases in which you want to write the expression `c = a + b`, where `a`, `b`, and `c` have three different types.

The first addend, that is, the left-hand side of the operation, is the current object, or `self`. Its type is `Self`.

The type of the second addend, that is, the right-hand side of the operation, is specified using a type parameter named `Rhs` (for right-hand side). Usually, when we monomorphize a generic function or a generic type, we must specify all its generic parameters. So, as we want to implement an addition of a `Complex` object with another `Complex` object, we should implement the `Add<Complex>` trait.

Though, Rust allows us to specify a default value for generic parameters. In the preceding definition, the default value for the `Rhs` generic parameter is `Self`. So, if we are implementing the `Add` trait, without specifying a generic parameter, it means that we are

actually implementing the `Add<Self>` trait. And in particular, if we are implementing the `Add` trait for the `Complex` type, it means that we are implementing the `Add<Complex>` trait for the `Complex` type.

The type of the operation result is specified as the `Output` associated type. Such type alias is used as the return value type of the `add` method.

So, in our code, it is enough to implement the `Add` trait for the `Complex` type, with `Self` implicitly specified as the `Rhs` generic parameter, and again `Self` explicitly specified as the `Output` associated type. It is what this program does:

```
struct Complex {
    re: f64,
    im: f64,
}

impl std::ops::Add for Complex {
    type Output = Self;
    fn add(self, rhs: Self) -> Self {
        Self { re: self.re + rhs.re, im: self.im + rhs.im }
    }
}

let z1 = Complex { re: 3.8, im: -2.1 };
let z2 = Complex { re: -1.5, im: 8.6 };
use std::ops::Add;
let z3 = z1.add(z2);
print!("{} + {}i", z3.re, z3.im);
```

You can obtain an equivalent program if you replace the last-but-two and last-but-one statements with this statement:

```
let z3 = z1 + z2;
```

So, we got our operator overloaded. Now, to make a somewhat more realistic example, let's encapsulate the `Complex` type in a module. There are two possibilities: to let any user freely access the fields `re` and `im`, with no data hiding; or to make those fields private, forcing the use of public functions to access them. For the `Complex` type, there is no invariant to keep, so we can keep the fields public, like in this complete example.

Notice that, in the following examples in this section, the `complex` module is outside of the `main` function:

```
mod complex {
    pub struct Complex {
        pub re: f64,
        pub im: f64,
    }

    impl std::ops::Add for Complex {
        type Output = Self;
        fn add(self, rhs: Self) -> Self {
            Self { re: self.re + rhs.re, im: self.im + rhs.im }
        }
    }
}

fn main () {
    use complex::Complex;
    let z1 = Complex { re: 3.8, im: -2.1 };
    let z2 = Complex { re: -1.5, im: 8.6 };
    let z3 = z1 + z2;
    print!("{} + {}i", z3.re, z3.im);
}
```

Here, we see the `Complex` type has been encapsulated in the `complex` module, so it would have become implicitly private, were it not for the `pub` keyword in the second line. Its two fields also would have become implicitly private, were it not for their `pub` keyword.

Then, the `use` directive is specified to avoid having to explicitly write `complex::Complex` every time that type name is to be specified.

Notice that the trait implementation has no `pub` keywords (and they are even forbidden there), because the implementations of a trait are implicitly public as long as that trait is accessible.

Now, let's apply data hiding, making private the fields of `Complex`, with this code:

```
mod complex {
    pub struct Complex {
        re: f64,
        im: f64,
    }

    impl Complex {
        pub fn from_re_im(re: f64, im: f64) -> Self {
            Self { re, im }
        }
        pub fn re(&self) -> &f64 { &self.re }
        pub fn im(&self) -> &f64 { &self.im }
    }
    impl std::ops::Add for Complex {
        type Output = Self;
        fn add(self, rhs: Self) -> Self::Output {
            Self { re: self.re + rhs.re, im: self.im + rhs.im }
        }
    }
}

fn main() {
    use complex::Complex;
    let z1 = Complex::from_re_im(3.8, -2.1);
    let z2 = Complex::from_re_im(-1.5, 8.6);
    let z3 = z1 + z2;
    print!("{} + {}i", z3.re(), z3.im());
}
```

Notice that the structure itself has remained public, but its fields are private now, so we need some additional functions to access them: the constructor `from_re_im`, and the getter methods `re` and `im`. They must be marked as public.

At last, let's consider the case that we want to generalize our type to use other numeric types for its components: not only `f64`, but also `f32` or even some user-defined numeric type. We must declare a generic type `Complex<Num>` with `Num`, a type that can be added to another `Num` object, returning a `Num` value. Here is the code:

```
mod complex {
    pub struct Complex<Num> {
        re: Num,
        im: Num,
    }

    impl<Num> Complex<Num> {
            pub fn from_re_im(re: Num, im: Num) -> Self {
            Self { re, im }
        }
        pub fn re(&self) -> &Num { &self.re }
        pub fn im(&self) -> &Num { &self.im }
    }

    impl<Num> std::ops::Add for Complex<Num>
    where Num: std::ops::Add<Output = Num> {
        type Output = Self;
        fn add(self, rhs: Self) -> Self::Output {
            Self { re: self.re + rhs.re, im: self.im + rhs.im }
        }
    }
}

fn main() {
    use complex::Complex;
    let z1 = Complex::from_re_im(3.8, -2.1);
    let z2 = Complex::from_re_im(-1.5, 8.6);
    let z3 = z1 + z2;
    print!("{} + {}i", z3.re(), z3.im());
}
```

Notice the following changes, with respect to the previous version:

- The six occurrences of `f64` have been replaced by `Num`.
- The two occurrences of `impl` have been replaced by `impl<Num>`.
- The three occurrences of `Complex` in the `complex` module have been replaced by `Complex<Num>`.
- In the implementation of the `Add` trait, a type parameter bound has been added: `where Num: std::ops::Add<Output = Num>`. It means that to implement this trait, we require knowing how to add an object of type `Num` with an object of type `Num` (implicitly specified), generating a result of type `Num`.

We have seen how to overload only the addition operator, though many other Rust operators can be overloaded. Each of them has its own trait, with its own method. The main ones are listed in the following table:

Operator	Trait	Method
+ (addition)	Add	add
- (subtraction)	Sub	sub
- (unary negation)	Neg	neg
* (multiplication)	Mul	mul
/ (division)	Div	div
% (remainder)	Rem	rem
+= (addition assignment)	AddAssign	add_assign
-= (subtraction assignment)	SubAssign	sub_assign
*= (multiplication assignment)	MulAssign	mul_assign
/= (division assignment)	DivAssign	div_assign
%= (remainder assignment)	RemAssign	rem_assign

Other operators that can be overloaded are the bit-twiddling operators: !, &, |, ^, <<, >>, &=, |=, ^=, <<=, >>=.

The `Index` trait and the `IndexMut` trait allow us to define the indexing operations with square brackets, for immutable objects and mutable objects, respectively.

The function call itself also can be overloaded, to implement smart pointers, using three traits: `Fn`, which takes the argument by immutable reference; `FnMut`, which takes the argument by mutable reference; and `FnOnce`, which takes the argument by value.

Rust does not allow redefining the following operators: `=`, `==`, `!=` , `<`, `<=`, `>`, `>=`. Actually, the behavior of such operators can be changed too, but using different mechanisms.

Infallible Type Converters

In many object-oriented languages, it is possible to declare type converters. Let's see an example that shows the need of such converters.

Using primitive types, we can write:

```
let ln = 1_f64 / 3_f64;
let sn = ln as f32;
print!("{}", sn);
```

It will print: `0.33333334`. This is possible because the `as` keyword allows converting between primitive types. Though, if we define custom types:

```
struct LargeNumber (f64);
struct SmallNumber (f32);
let ln = LargeNumber (1. / 3.);
let sn = ln as SmallNumber;
print!("{}", sn.0);
```

This generates the compilation error: `non-primitive cast: `LargeNumber` as `SmallNumber``. This is caused by the use of the `as` keyword to convert between nonprimitive types. This notation is not allowed, and anyway the compiler doesn't know how to perform such a conversion.

To allow conversions between arbitrary types, Rust provides four different techniques, which make use of the standard library traits `Into`, `From`, `TryInto`, and `TryFrom`. Let's see an example of the first one:

```
struct LargeNumber (f64);
struct SmallNumber (f32);

impl Into<SmallNumber> for LargeNumber {
    fn into(self) -> SmallNumber {
        SmallNumber (self.0 as f32)
    }
}

let ln = LargeNumber (1. / 3.);
let sn: SmallNumber = ln.into();
print!("{}", sn.0);
```

It will print: `0.33333334`. In the last-but-one line, the `into` method is applied to the value of the `ln` variable, and it returns the conversion of that value into a value assigned to the `sn` variable. Notice that the type of the value returned by `into` must be inferred from the context.

To be able to write this statement, the previous `impl` statement is required. It implements the generic `Into` trait for the `LargeNumber` type, which is the source type of the conversion. Such implementation monomorphizes the implementation for the `SmallNumber` type, which is the destination of the conversion.

Here, both the source type and the destination type are user-defined types. This solution can also be applied if the source type *or* the destination type is an external type, that is, a primitive type or a type defined in another crate. However, it is not allowed to convert from an external type to an external type.

This solution works, but it has the drawback of requiring us to call the `into` function in a context in which the destination type can be inferred.

The other allowed solution uses the `From` standard library trait, as this example shows:

```
struct LargeNumber (f64);
struct SmallNumber (f32);

impl From<LargeNumber> for SmallNumber {
    fn from(source: LargeNumber) -> Self {
        Self (source.0 as f32)
    }
}
```

```
let ln = LargeNumber (1. / 3.);
let sn = SmallNumber::from(ln);
print!("{}", sn.0);
```

Here, in the last-but-one statement, the function `from`, associated with the `SmallNumber` type is called to convert its argument `ln`. Type inference is not required, because the destination type is specified anyway.

To be able to write this statement, the previous `impl` statement implements the generic `From` trait for the `SmallNumber` type, which is the destination type of the conversion. Such implementation monomorphizes the implementation for the `LargeNumber` type, which is the source of the conversion. Notice that the source and destination types have been swapped, with respect to the implementation of `Into`.

Also this solution allows that the source or destination type is external, but not both.

The `From` solution has an additional advantage, with respect to the `Into` solution. In the previous example, try to replace the last-but-one statement with this one:

```
let sn: SmallNumber = ln.into();
```

You will get a valid, equivalent program. Actually, when you implement the `From` trait, a default implementation of the `Into` trait is implicitly declared. Therefore, it is recommended to implement only the `From` trait, and then freely use the `from` or `into` functions. You should implement the `Into` trait only when, for some reason, you require that the `From` trait is not implemented, or when you are not satisfied by the default implementation of `Into`.

These solutions have a possible drawback, though. They are not fallible. Whenever a conversion of such kind is called, it must always return a value. But, in some cases you may want a conversion to succeed for some values and fail for some other values.

Fallible Type Converters

To support fallible type conversions, the standard library declares the `TryInto` and the `TryFrom` traits are available. Let's see an example using the last one:

```
struct LargeNumber (f64);
struct SmallNumber (f32);
```

```
impl TryFrom<LargeNumber> for SmallNumber {
    type Error = String;
    fn try_from(source: LargeNumber) -> Result<Self, Self::Error> {
        if source.0.abs() > f32::MAX as f64 {
            Err("too large number".to_string())
        }
        else if source.0 != 0.
        && source.0.abs() < f32::MIN_POSITIVE as f64 {
            Err("too small number".to_string())
        }
        else {
            let result = source.0 as f32;
            if result as f64 == source.0 {
                Ok(Self (result))
            }
            else {
                Err("precision loss".to_string())
            }
        }
    }
}

fn show_result(n: Result<SmallNumber, String>) {
    match n {
        Ok(x) => println!("Converted to {}", x.0),
        Err(msg) => println!("Conversion error: {}", msg),
    }
}

let ln = LargeNumber (1.5);
show_result(SmallNumber::try_from(ln));
let ln = LargeNumber (1. / 3.);
show_result(SmallNumber::try_from(ln));
let ln = LargeNumber (1e50);
show_result(SmallNumber::try_from(ln));
let ln = LargeNumber (1e-50);
show_result(SmallNumber::try_from(ln));
```

It will print:

```
Converted to 1.5
Conversion error: precision loss
Conversion error: too large number
Conversion error: too small number
```

In the last eight lines, four objects of type `LargeNumber` are created, all named `ln`. For each of them, the `try_from` function is called; the result of that conversion attempt is passed to the `show_result` function, which prints a result message.

The first variable contains the 64-bit number `1.5`, which can be translated to a 32-bit number without loss of precision. Therefore, it is converted successfully.

The second variable contains the 64-bit number obtained dividing one by three. That number cannot be represented exactly in a binary representation; therefore it is represented in an approximate way. However, a 64-bit representation has more significant digits than a 32-bit representation, so the two approximate representations are different. So, the conversion fails with the error message: `precision loss`.

The third variable contains a 64-bit number larger than the maximum value that can be represented by a 32-bit number. So, the conversion fails with an error message: `too large number`.

The fourth variable contains a 64-bit number that is nearer to zero than the minimum positive value that can be represented by a 32-bit number. So, the conversion fails with an error message: `too small number`.

To get such behavior, the `try_from` method, instead of returning a value of type `Self`, returns a value of type `Result<Self, Self::Error>`. In case that routine succeeds in the conversion, an `Ok` variant is returned. In case it fails, an `Err` variant is returned. The type of that error must be specified as the `Error` associated type of the trait implementation. In our example, such error type was `String`.

Composition Instead of Inheritance

At the beginning of the object-oriented programming era, inheritance appeared as a panacea, a technique that could be applied to any problem. In fact, there are three kinds of inheritance: data inheritance, method implementation inheritance, and method interface inheritance. As years went by, it was realized that data inheritance created

more problems than it solved, so Rust does not support it. In place of data inheritance, Rust uses composition.

Let's consider an example. Assume we already have a type representing a text to draw on the graphical screen, and we want to create a type representing a text surrounded by a rectangle. For simplicity, instead of drawing the text, we will print it on the console; and instead of drawing the rectangle, we will surround the text in brackets. Here is our code:

```
struct Text { characters: String }
impl Text {
    fn from(text: &str) -> Text {
        Text { characters: text.to_string() }
    }
    fn draw(&self) {
        print!("{}", self.characters);
    }
}
let greeting = Text::from("Hello");
greeting.draw();
struct BoxedText {
    text: Text,
    first: char,
    last: char,
}
impl BoxedText {
    fn with_text_and_borders(
        text: &str, first: char, last: char)
        -> BoxedText
    {
        BoxedText {
            text: Text::from(text),
            first: first,
            last: last,
        }
    }
```

```
    fn draw(&self) {
        print!("{}", self.first);
        self.text.draw();
        print!("{}", self.last);
    }
}
let boxed_greeting =
    BoxedText::with_text_and_borders("Hi", '[', ']');
print!(", ");
boxed_greeting.draw();
```

It will print: `Hello, [Hi]`.

The first statement defines a struct representing a text, characterized only by the string that contains its characters.

The second statement defines two methods for this struct: `from`, which is a constructor; and `draw`, which prints the string contained in the object. So, we can create a `Text` object, and we can draw it using such a method.

Now assume that you want to capitalize on this struct and its associated methods to create a text enclosed in a box, represented by a struct named `BoxedText`. That is what inheritance usually is touted for.

In Rust, instead of using inheritance, you create a `BoxedText` struct that *contains* an object of `Text` type and some additional fields. In this case, there are two fields representing the character to print before and after the base text.

Then, you declare the implementation of the methods you want to encapsulate in the `BoxedText` type, that is, to apply to objects of `BoxedText` type. They are a constructor, named `with_text_and_borders`, and a function to draw the current object, named `draw`.

The constructor gets all the information it needs and initializes a new object with that information. In particular, the field having type `Text` is initialized by invoking the `Text::from` constructor.

The method to draw the current object has the same signature as the similar method associated with `Text` objects, but this is just a coincidence, due to the fact that such methods do similar things. They could have different names, different argument types, or different return value types. The body of this `draw` method, though, contains an invocation of the `draw` method of `Text`.

Finally, an object having type `BoxedText` is created, and its method `draw` is invoked, having as a result that the string `[Hi]` is printed.

In this program, reuse happens in the following places:

- The first field of `struct BoxedText` is `text: Text`. It reuses that data structure.
- The constructor of `BoxedText` contains the expression `Text::from(text)`. It reuses the constructor of the `Text` type.
- The body of the `draw` method of the `BoxedText` type contains the statement `self.text.draw();`. It reuses the `draw` method associated with the `Text` type.

Static Dispatch

In the previous section, we have seen how to define the `Text` and `BoxedText` types so that the latter reuses some data and code written for the former.

Now assume you need to write a function capable of drawing several kinds of text objects. In particular, if it receives as an argument an object of `Text` type, it should invoke the `draw` method associated with the `Text` type; while, if it receives as argument an object of `BoxedText` type, it should invoke the `draw` method associated to the `BoxedText` type.

If Rust were a dynamically typed language, this function would be:

```
fn draw_text(txt) {
    txt.draw();
}
```

Of course, this is not allowed in Rust, because the type of the `txt` argument must be specified explicitly.

Rust allows two nonequivalent solutions for this problem. One is this:

```
trait Draw {
    fn draw(&self);
}
struct Text { characters: String }
impl Text {
    fn from(text: &str) -> Text {
        Text { characters: text.to_string() }
    }
}
```

```
impl Draw for Text {
    fn draw(&self) {
        print!("{}", self.characters);
    }
}
struct BoxedText {
    text: Text,
    first: char,
    last: char,
}
impl BoxedText {
    fn with_text_and_borders(
        text: &str, first: char, last: char)
        -> BoxedText
    {
        BoxedText {
            text: Text::from(text),
            first: first,
            last: last,
        }
    }
}
impl Draw for BoxedText {
    fn draw(&self) {
        print!("{}", self.first);
        self.text.draw();
        print!("{}", self.last);
    }
}
let greeting = Text::from("Hello");
let boxed_greeting =
    BoxedText::with_text_and_borders("Hi", '[', ']');
// SOLUTION 1 //
fn draw_text<T>(txt: T) where T: Draw {
    txt.draw();
}
```

```
draw_text(greeting);
print!(", ");
draw_text(boxed_greeting);
```

It will print: `Hello, [Hi]`.

The last three lines are the ones that print the resulting text. The `draw_text(greeting)` statement receives an object having type `Text` and prints `Hello`; and the `draw_text(boxed_greeting)` statement receives an object having type `BoxedText` and prints `[Hi]`. Such a generic function is defined just before. These last six lines are the first solution to the problem. The preceding part of the program is in common with the second solution.

But let's start to examine this program from the beginning.

First, the `Draw` trait is declared, as the capability of an object to be drawn.

Then the `Text` and `BoxedText` types are declared, with their associated methods, similarly to the example of the previous section. Though, only the two constructors `Text::from` and `BoxedText::with_text_and_borders` are kept as inherent implementations; the two `draw` functions, instead, now are implementations of the `Draw` trait.

Regarding the previous example, we said that the two `draw` methods had the same signature by coincidence, and they could have different signatures just as well. In this last example, instead, this equality is no more a coincidence; such functions are now used to implement the `Draw` trait, so they must have the same signature of the function declared in such trait.

After having declared the two types, their associated functions, and their implementations of the `Draw` trait, the two variables `greeting` and `boxed_greeting` are created and initialized, one for each of those types.

Then there is the first solution. The `draw_text` generic function receives an argument of `T` type, where `T` is any type that implements the `Draw` trait. Because of this, it is allowed to invoke the `draw` function on that argument.

So, whenever the compiler encounters an invocation of the `draw_text` function, it determines the type of the argument, and checks if such type implements the `Draw` trait. In case that trait is not implemented, a compilation error is generated. Otherwise, a monomorphized version of the `draw_text` function is generated. In this concrete function, the `T` type is replaced by the type of the argument, and the invocation of the `draw` generic method in the body of the function is replaced by an invocation of the implementation of `draw` for the type of the argument.

This technique is named ***static dispatch***. In programming language theory, *dispatch* means choosing which function to invoke when there are several functions with the same name. In this program there are two functions named `draw`, so a dispatch is required to choose between them. In this program, this choice is performed by the compiler, at compile time, so this dispatch is *static*.

Dynamic Dispatch

The previous program can be changed a little, by replacing the last seven lines with the following ones:

```
// SOLUTION 1/bis //
fn draw_text<T>(txt: &T) where T: Draw {
    txt.draw();
}
draw_text(&greeting);
print!(", ");
draw_text(&boxed_greeting);
```

This version has substantially the same behavior of the previous version. Now, the `draw` function receives its argument by reference, and so an "&" character has been added in its signature, and other "&" characters have been added at the two invocations of this function.

This solution is still a case of static dispatch. So, we see that static dispatch can work both with pass-by-value and with pass-by-reference.

The previous program can be changed further, by replacing the last seven lines with the following ones:

```
// SOLUTION 2 //
fn draw_text(txt: &dyn Draw) {
    txt.draw();
}
draw_text(&greeting);
print!(", ");
draw_text(&boxed_greeting);
```

Also this program has the same external behavior as before, but it uses another kind of technique. Now only the `draw_text` signature has been changed. The `T` generic parameter and the `where` clause have been removed, and the argument has type `&dyn Draw` instead of `&T`.

Here, the new **`dyn`** keyword is used. It means that this type is not a usual reference, but it is something new. It is a *dynamic-dispatch reference*. So now, instead of a generic function, we have a concrete function, which gets as argument a *dynamic-dispatch* reference. Though, the calls to this function haven't changed: one passes as argument a reference to a `Text`, and the other a reference to a `BoxedText`. So such references are automatically converted to dynamic-dispatch references.

Let's see the purpose of this dynamic-dispatch reference argument. When the `draw_text` method is called using the `&greeting` argument, the method receives a pointer to the `greeting` object, but it also receives some indication that the type of this object has implemented the `Draw` trait for the `Text` type. Instead, when the `&boxed_greeting` argument is used, the `draw_text` method, in addition to the pointer to the `boxed_greeting` object, receives some indication that the type of this object has implemented the `Draw` trait for the `BoxedText` type. Having such additional information, the `draw_text` method is able to invoke the appropriate `draw` method at runtime.

So, `&dyn Draw` is a kind of pointer capable of choosing the right `draw` method to invoke, according to the type of the referenced object. This is a kind of dispatch, but it happens at runtime, so it is a *dynamic dispatch*.

Dynamic dispatch is handled in C++ by using the `virtual` keyword, albeit with a slightly different mechanism.

Implementation of Dynamic-Dispatch References

Get back to the program showing the last solution to the dispatch problem, and replace the last seven lines (from the one starting with `// SOLUTION`) with the following code:

```
// PRINTING OBJECT SIZES
use std::mem::size_of_val;
print!("{} {} {}, {} {} {}, ",
    size_of_val(&greeting),
    size_of_val(&&greeting),
    size_of_val(&&&greeting),
```

```
    size_of_val(&boxed_greeting),
    size_of_val(&&boxed_greeting),
    size_of_val(&&&boxed_greeting));
fn draw_text(txt: &dyn Draw) {
    print!("{} {} {} ",
        size_of_val(txt),
        size_of_val(&txt),
        size_of_val(&&txt));
    txt.draw();
}
draw_text(&greeting);
print!(", ");
draw_text(&boxed_greeting);
```

The resulting program, in a 64-bit target, will print: `24 8 8, 32 8 8, 24 16 8 Hello, 32 16 8 [Hi]`.

Remember that the `size_of_val` standard library generic function gets a reference to an object of any type, and returns the size in bytes of that object.

First, the `greeting` variable is processed. Its type is `Text`, which contains only a `String` object. We already discovered that `String` objects occupy 24 bytes in the stack, plus a variable buffer in the heap. That buffer is not taken into account by the `size_of_val` function. The `size_of_val` function requires a reference, so the expression `size_of_val(greeting)` would be illegal.

The size of a reference to a `Text` is printed, and then the size of a reference to a reference to a `Text`. They are ordinary references, so they occupy 8 bytes.

Then, the `boxed_greeting` variable is processed in the same way. This struct contains a `Text` and two `char` objects. Each `char` occupies 4 bytes, and so it is 24 + 4 + 4 = 32 bytes. Its references are normal references too.

Then, near the end of the program, the expression `&greeting`, having type `&Text`, is passed as an argument to the `draw_text` function, where it is used to initialize the argument `txt`, having type `&dyn Draw`.

The `txt` argument is a kind of reference, so it is possible to evaluate the expression `size_of_val(txt)`. It will return the size of the referenced object. But which type has the object referenced to by an object of type `&dyn Draw`? Of course, it is not `Draw`, because `Draw` is not a type. Actually, this cannot be said at compile time. It depends on the object referenced at runtime by the expression used to initialize the `txt` argument. The first

time that the `draw_text` function is invoked, the `txt` argument receives a reference to a `Text` object, so 24 is printed.

If you jump forward in the output, going after the comma, you see that the second time that the `draw_text` function is invoked, the `txt` argument receives a reference to a `BoxedText` object, so 32 is printed.

Going back to the invocation using `greeting`, we see that the value of the expression `size_of_val(&txt)` is 16. This may be surprising. This expression is the size of an object having type `&dyn Draw`, initialized by an object having type `&Text`. So we use a normal 8-byte reference to initialize a 16-byte dynamic-dispatch reference. Why is a dynamic-dispatch reference so large? The mechanism of dynamic dispatch lies in how it is initialized.

Actually any dynamic-dispatch reference has two fields. The first one is a copy of the reference used to initialize it, and the second one is a pointer used to choose the proper version of the `draw` function, or any other function needing dynamic dispatch. It is named `virtual table pointer`. This name comes from C++.

At last, the size of a reference to a dynamic-dispatch reference is printed. This is a normal reference, occupying eight bytes.

The same numbers are printed for the references to the `boxed_greeting` variable.

Static vs. Dynamic Dispatch

So you can use static or dynamic dispatch. Which should you use?

Like any instance of static-versus-dynamic dilemma, where *static* means *compile-time*, and *dynamic* means *runtime*, static requires a somewhat longer compilation time, and generates somewhat faster code; but if not enough information is available to the compiler, the dynamic solution is the only possible one.

Assume that, for the example programs shown before, there is the following requirement. The user is asked to enter a string. If that string is `b`, a boxed text should be printed; and for any other input, a nonboxed text should be printed.

Using the static dispatch, the final part of the program (replacing the lines from `// PRINTING OBJECT SIZES`) becomes:

```
// SOLUTION 1/ter //
fn draw_text<T>(txt: T) where T: Draw {
    txt.draw();
}
```

```
let mut input = String::new();
std::io::stdin().read_line(&mut input).unwrap();
let to_box = input.trim() == "b";
if to_box {
    draw_text(boxed_greeting);
} else {
    draw_text(greeting);
}
```

If you run it and then you press b and Enter, the text `[Hi]` will be printed. If instead you type nothing or any other text and then Enter, the text `Hello` will be printed.

Instead, using the dynamic dispatch, you have this equivalent code:

```
// SOLUTION 2/bis //
fn draw_text(txt: &dyn Draw) {
    txt.draw();
}
let mut input = String::new();
std::io::stdin().read_line(&mut input).unwrap();
let to_box = input.trim() == "b";
draw_text(if to_box { &boxed_greeting } else { &greeting });
```

The static dispatch forces you to write several function invocations, one for each type to use for the argument to pass to `draw_text`. Instead, the dynamic dispatch allows you to write just one call to `draw_text`, and let the dynamic dispatch mechanism perform the choice.

In addition, static dispatch uses generic functions, and this technique may create code bloat, so it may end up being slower.