Structuring data and organizing types

SSP-RS — Safe System Programming (in Rust)

Samuel Tardieu Stefano Zacchiroli

2024-10-15

More on composite types

Reminder: arrays, vectors and slices

Rust distinguishes between arrays, vectors and slices:

- An **array** contains a fixed number of elements of the same type.
- A **vector** is a growable/shrinkable collection of elements of the same type.
- A **slice** is a collection of consecutive elements whose size is known at run time only.

Thanks to the `Deref` trait (more on this later), a reference to an array or a vector can be automatically made into a reference to a slice.

```
1  let my_array: [u32; 4] = [10, 20, 30, 40];
2
3  let mut my_vec: Vec<u32> = vec![10, 20, 30, 40];
4  my_vec.push(50);
5
6  let slice1: &[u32] = &my_array;    // This slice has size 4
7  let slice2: &[u32] = &my_vec;      // This slice has size 5
```

A reference to a slice encodes both the memory location and the number of elements (**fat pointer**).

Slices and pattern matching

Slices can be used in a pattern matching environment:

```
fn describe(slice: &[u32]) {
  match slice {
    []          => println!("Empty"),
    [x]         => println!("One unique element {x}"),
    [x, y]      => println!("Two elements {x} and {y}"),
    [x, y, ..] => println!("More than two elements, starting with {x} and {y}"),
  }
}

fn main() {
  describe(&vec![10, 20, 30, 40]);
}
```

outputs:

```
More than two elements, starting with 10 and 20
```

Reminder: strings

`String` instances contain valid UTF-8 strings. `str` is to `String` what `[T]` is to `Vec<T>`, *i.e.,* a memory area containing a string whose size is known at run time only:

```
1  let s1: &str = "Hello, world!";      // Points to read-only memory
2  let s2: String = String::from(s1);   // Stored on the heap after copying the content of s1
3  let s3: &str = &s2;                  // Points to the heap data owned by s2
```

Similar to a slice, a reference to a `str` encodes both the memory location and the size of the string in bytes using a fat pointer.

Tuples

A tuple groups several elements together, possibly of different types:

```
fn main() {
  let x: (String, u32) = (String::from("John Doe"), 23);
  println!("Name: {}", x.0);
  println!("Age: {}", x.1);
}
```

Elements in a tuple are accessed with `.0`, `.1`, etc. A tuple can also be used in a pattern matching environment, for example a `let`:

```
fn main() {
  let x: (String, u32) = (String::from("John Doe"), 23);
  let (name, age) = x;      // x is deconstructed here and ceases to exist
  println!("Name = {name}, age = {age}");
}
```

Structures

We have encountered structures in the lab sessions, but haven't formally studied them yet. They allow the grouping of several values, possibly of different types, together:

```
#[derive(Clone, Debug)]
pub struct Person {
  pub name: String,
  pub age: Option<u8>,
}
```

A structure can derive some traits (properties),[1] such as `Clone` (resp. `Debug`): if all fields of the structure are implicitly copiable (resp. displayable), the structure will also be implicitly copiable (resp. displayable):

```
fn main() {
  let person_a = Person { name: String::from("John Doe"), age: Some(23) };
  let person_b = person_a.clone();        // Because of #[derive(Clone)]
  println!("Person B = {person_b:?}");  // Because of #[derive(Debug)]
}
```

[1] more on this later in this lecture

Another name for structures

Let's assume that type `T1` can take n_1 possible values, and type `T2` can take n_2 possible values.

```
pub struct S {
  pub t1: T1,
  pub t2: T2,
}
```

How many distinct values can `S` take?

`S` can take any combination of a `T1` value with a `T2` value, which makes a total of $n_1 \times n_2$ values. Structures are hence called **product types** in type theory.

The following structure can take $256 \times 2 \times 2 = 1024$ possible values:

```
pub struct S {
  pub x: u8,
  pub b: bool,
  pub c: bool,
}
```

Structures and pattern matching

Structures can be used in a pattern matching environment. In this context, a field name alone (`age`) means `age: age`. `_` means “don’t care”, and `..` means “and the other fields if any”.

```
fn describe(person: &Person) {
  match person {
    Person { age: Some(18), .. }     => println!("A recent adult"),
    Person { name, age: None }       => println!("An ageless person named {name}"),
    Person { name: _, age: Some(a) } => println!("A {a} year old person"),
  }
}
```

Note how we are able to match `person` (of type `&Person`) against a `Person` like in the pattern `Person { name, age: None }`. How can a reference ever match a non-reference?

Let’s talk a bit more about pattern matching.

Interlude: pattern matching

Pattern matching tries to unify the pattern (below, `Some(s)`) with the data (`o`) it is matched against:

```
fn describe(o: Option<String>) {
  match o {
    Some(s) => println!("o contained a string: {s}"),
    None    => println!("o contained Nothing"),
  }
  // We want to do something else with o but it doesnt exist anymore :(
}
```

However, right after the `match` expression, `o` does no longer exist as it has been deconstructed.

What if we just want to *peek* inside the option and not deconstruct it?

Interlude: pattern matching and ref

By using the `ref` keyword, we can indicate that we just want to obtain a reference to the value that was unified with our variable instead of getting the value itself.

```
1  fn describe(o: Option<String>) {
2    match o {
3      Some(ref s) => println!("o contained a string: {s}"),  // s is a &String
4      None        => println!("o contained Nothing"),
5    }
6    println!("o still exists here, it has not been deconstructed: {o:?}");
7  }
```

In the example above, `s` is of type `&String` instead of type `String`, because of the `ref s`. Note that `ref mut` also exists and would have given a `&mut String` if `o` was mutable.

Interlude: pattern matching and non-reference patterns

There is a more ergonomic way than `ref` to get a reference on the inside of the option:

```
fn describe(o: Option<String>) {
  match &o {   // Note the "&" here
    Some(s) => println!("o contained a string: {s}"),  // s is a &String
    None    => println!("o contained Nothing"),
  }
  println!("o still exists here, it has not been deconstructed: {o:?}");
}
```

Since `Some(s)` can never match an `&Option<String>` as it is not a reference, the Rust compiler tries adding a `&` in front of it as well as a `ref` before all free variables (or `&mut`/`ref mut` if the target is a mutable reference):

- `Some(s)` is transformed into `&Some(ref s)`
- `None` is transformed into `&None` (no free variable to add `ref` to here)

Now the `match` branches are each correctly typed and may match the type of `&o`, that is `&Option<String>`.

Interlude: pattern matching and a mutable example

We want to write a function which takes a `&mut Option<String>` and appends a “!” character to its content if it is not None. Any ideas?

```
fn maybe_append(o: &mut Option<String>) {
  if let Some(s) = o {    // s is of type &mut String
    s.push('!');
  }
}

fn main() {
  let mut o: Option<String> = Some(String::from("Hello, world"));
  maybe_append(&mut o);
  println!("{o:?}");      // Displays: Some("Hello, world!");
}
```

The `maybe_append` function is transformed by the compiler into

```
fn maybe_append(o: &mut Option<String>) {
  if let &mut Some(ref mut s) = o {  // s is of type &mut String
    s.push('!');
  }
}
```

Interlude: back to our structure

By applying the rules we just saw, we now understand that

```
fn describe(person: &Person) {
  match person {
    Person { age: Some(18), .. }     => println!("A recent adult"),
    Person { name, age: None }       => println!("An ageless person named {name}"),
    Person { name: _, age: Some(a) } => println!("A {a} year old person"),
  }
}
```

is transformed by the compiler into

```
fn describe(person: &Person) {
  match person {
    &Person { age: Some(18), .. }         => println!("A recent adult"),
    &Person { ref name, age: None }       => println!("An ageless person named {name}"),
    &Person { name: _, age: Some(ref a) } => println!("A {a} year old person"),
  }
}
```

`ref name` alone in a structure pattern is a shortcut for `name: ref name`.

Implementation blocks

A structure, as all compound types, can be accompanied by implementation blocks:

```
impl Person {
  pub fn new(name: &str) -> Self {
    Person { name: String::from(name), age: None }
  }

  pub fn set_age(&mut self, age: u8) {
    self.age = Some(age);
  }
}

fn main() {
  let mut person = Person::new("John Doe");
  person.set_age(23);   // Or person.age = Some(23)
}
```

In an implementation block, `Self` designates the type on which the implementation is defined. Functions taking `self`, `&self`, `&mut self` or similar expressions are called **methods**. Other functions are called **associated functions**.

Implementation blocks (cont.)

Methods can be used to retrieve or set private fields:

```rust
pub struct Person {
  name: String,        // Note the absence of "pub" on those two fields. They are
  age: Option<u8>,     // accessible only by the current module and its descendants.
}

impl Person {
  pub fn new(name: &str) -> Self { Person { name: String::from(name), age: None } }

  pub fn name(&self) -> &str { &self.name }

  // We do not provide a setter for "name" as we do not want it modified

  pub fn age(&self) -> Option<u8> { self.age }

  pub fn set_age(&mut self, age: u8) { self.age = Some(age) }
}
```

Lifetime rules guarantee that the reference returned by the `.name()` method cannot be used beyond the lifetime of the `Person` it was obtained from.

Structures as tuples

Instead of using named fields surrounded by “{}”, a structure can use a nameless tuple shape:

```
1  pub struct BytePair(u8, u8);   // Store two bytes, without explicit field names
2
3  impl BytePair {
4    pub fn new(left: u8, right: u8) -> Self {    // Here Self = BytePair
5      BytePair(left, right)
6    }
7
8    pub fn left(&self) -> u8 { self.0 }
9    pub fn right(&self) -> u8 { self.1 }
10 }
11
12 let p = BytePair(1, 42);
13 println!("pair ({}, {})", p.left(), p.right());  // Displays "pair (1, 42)"
```

Enumerations

Enumerations represent exclusive variants. Each variant can be structure-like and store additional fields:

```
pub enum Human {
  Toddler,
  Teen,
  Adult { name: String, age: u8 },   // named fields .name and .age
  Elder(String, u8),                 // positional fields .0 and .1
}
```

Pattern matching can be used the same way as on other types:

```
impl Human {
  pub fn describe(&self) {
    match self {
      Human::Toddler | Human::Teen =>
        println!("A small creature"),
      Human::Adult { name: n, .. } | Human::Elder(n, _) =>
        println!("An older person named {n}"),
    }
  }
}
```

Another name for enumerations

If type `T1` can take n_1 possible values, and type `T2` can take n_2 possible values, how many different values can `E` take?

```
pub enum E {
  Variant1(T1),
  Variant2(T2),
}
```

Since `E` can take any of `T1` values with variant 1, or (exclusively) any of `T2` values with variant 2, it can take a total of $n_1 + n_2$ values. Enumerations are called **sum types** in type theory.

The following type can take $1 + 256 + 2 + 2 = 261$ different values:

```
pub enum E {
  Variant1,              // Only 1 value, itself
  Variant2(u8),          // 256 values
  Variant3(bool),        // 2 values: true or false
  Variant4(bool),        // 2 values: true or false
}
```

Some thoughts on the number of possible values

How many distinct values can the following types take?

```rust
pub struct S1 {
  x: u8,
  y: (),
}
```

`S1` can take $256 \times 1 = 256$ values since `()` unique value is `()`.

```rust
pub struct S2 {
  x: (),
  y: (),
}
```

`S2` can take only $1 \times 1 = 1$ value

```rust
pub enum Empty {}
```

`Empty` has 0 possible values. You can use the `Empty` type, but you cannot instantiate it.

```rust
pub enum E1 {
  Variant1,
  Variant2(u8),
  Variant3(Empty)
}
```

`E1` can take $1 + 256 + 0 = 257$ values. You cannot build variant 3.

```rust
pub struct S3 {
  x: u8,
  y: Empty,
}
```

`S3` can take $256 \times 0 = 0$ values. You cannot build an instance of S3 since you will not be able to build a value to initialize the `y` field.

Memory representation

Here are some examples of the size used in memory to represent some types:

- `u8`: 1 byte
- `i8`: 1 byte
- `(u16, u16)`: 4 bytes
- `Vec<T>`: 3 machine words (address, size, capacity) (+ the heap storage for the elements)
- `&i32` (reference): 1 machine word, *i.e.,* 8 bytes on a 64 bit computer
- `&str` (fat pointer): 2 machine words (address and size), *i.e.,* 16 bytes on a 64 bit computer
- `Empty`: 0 bytes
- `()`: 0 bytes

`Empty` and `()` are called **zero-sized types (ZST)**. Rust standard collections handles them specially: a `Vec<()>` will never allocate any memory on the heap.

Memory representation of a reference

How many distinct values can a Rust reference `&u8` take on a 64 bit computer?

In Rust, a reference points onto a valid object and cannot be `NULL` (0). It has $2^{64} - 1$ values as it will never contain the all-0 value.

How many distinct values can an `Option<&u8>` take on a 64 bit computer?

An `Option<&u8>` can be `None` (1 value) or `Some(&x)` if `x` is an `u8` ($2^{64} - 1$ values). The answer is: `Option<&u8>` can take 2^{64} values.

Rust will apply its **null-pointer optimization (NPO)** there: it guarantees that for every type `T`, `Option<&T>` has the same size as `&T`. `None` will be represented in memory with the 0 value, which is the only invalid reference for every type. `Some(&x)` will be represented the same way as `&x`, which cannot be all 0.

This will become important when interfacing with other languages: `NULL` will become `None`, and a non-null pointer will become `Some(…)`, and vice-versa (cf. future lecture on FFI).

Other high-level data structures

In addition to `Vec<T>`, the standard library contains high-level data structures in the `std::collections` module:

- `BinaryHeap<T>`: a binary heap, or maximum priority queue
- `BinaryMap<K, V>`: a key-value dictionary with `O(log(N))` lookups and sorted keys
- `BinarySet<T>`: a set with `O(log(N))` lookups and sorted keys
- `HashMap<K, V>`: a key-value dictionary with `O(1)` lookups
- `HashSet<T>`: a set with `O(1)` lookups
- `LinkedList<T>`: a doubly-linked list with `O(1)` pushing and popping at either end
- `VecDeque<T>`: a vector which can be pushed to or popped from both ends

All those are safe compound types, as well as the other types we have seen so far.

Traits and generics

Polymorphism

Code reuse is a desirable property in all non-trivial software systems. We will see in a future lecture how packages, crates, and the module system helps with in-the-large, *across-projects code reuse*.

Other Rust features support *within-project code reuse*. Here are a couple of examples we have already encountered in passing:

1. `Vec<T>` is a **generic** data structure: you can create a vector containing values of any type `T`, including types that do not exist yet (and hence did not exist when `Vec<T>` was implemented).
2. Several types have a `.fmt(...)` method that is invoked to pretty print their values when passed to `println!()` using `{}`. You can also invoke the method yourself, without caring about the type. Those types share the `Display` **trait**.

Generics and *traits* are two forms of **polymorphism** supported by Rust.

In this lecture we are going to learn about them and how they help reusing Rust code with either minimal or absent performance penalty (so called **"zero-cost abstractions"**).

Hello, traits world!

Traits, inspired by Haskell's type classes, are Rust's take on features like Java interfaces and Python abstract base classes. Here's the trait `std::io::Write`:

```rust
trait Write { // something a byte-stream can be written to (a "byte-oriented sink")
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;
    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }  // default impl., calling write()
    ... // more methods available
}
```

Some of the standard types that **implement** this trait are: `File`, `TcpStream`, `Vec<u8>`.
Therefore they all provide the shown methods (and more).

```rust
use std::fs::File;
let mut local_file = File::create("hello.txt")?;
local_file.write_all(b"hello world\n")?;
local_file.flush()
```

```rust
let mut bytes = vec![];
bytes.write_all(b"hello world\n")?;
bytes.flush()?;
assert_eq!(bytes, b"hello world\n");
```

Traits and types

A trait usually denotes *something a type can do*, i.e., a **capability**.

Unlike *classes* in OOP (Object-Oriented Programming), traits do not induce a strict hierarchy on code that implements them. Rather, a type can implement multiple traits and a trait can be implemented by multiple types, in a N-M cardinality relationship.

Some examples from the standard library:

Trait	Capability	Implemented by (e.g.)
`std::io::Write`	can be written bytes to	`File`, `TcpStream`, `Vec<u8>`
`std::iter::Iterator`	can produce a sequence of values	`Range<i32>`
`std::clone::Clone`	can make copies of itself	most stdlib types
`std::fmt::Debug`	can be printed using `{:?}`	most stdlib types

Traits as adapters

Traits are more than inert *interfaces*. Traits can also provide (overridable) **default implementations** for methods, that can rely on other trait methods (with or without default implementations).

This makes traits suitable for implementing the **adapter design pattern** to lift small APIs (the non-default methods that implementers must provide) to larger APIs (all trait methods: default or not).

```
1  trait CmpToZero {
2      fn is_zero(&self) -> bool;  // no implementation, must be implemented by `impl` type
3      fn is_not_zero(&self) -> bool {  // default implementation (can be overridden)
4          !self.is_zero()
5      }
6  }
7
8  struct Point { x: i32, y: i32 }
9  impl CmpToZero for Point {
10     fn is_zero(&self) -> bool {
11         (self.x == 0) && (self.y == 0)
12     }
13 }
14
15 assert!((Point { x: 1, y: -1 }).is_not_zero());  // use default implementation
```

Traits and imports

One common gotcha of using Rust traits is that, unlike inherited methods in OOP languages, **trait methods are not in scope by default**.

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

Let's welcome the compiler explaining this to us with a super-helpful error message:

```
3    |     buf.write_all(b"hello")?; // error: no method named `write_all`
     |         ^^^^^^^^^ method not found in `Vec<u8>`
[...]
1540 |     fn write_all(&mut self, mut buf: &[u8]) -> Result<()> {
     |        --------- the method is available for `Vec<u8>` here
     |
     = help: items from traits can only be used if the trait is in scope
help: the following trait is implemented but not in scope; perhaps add a `use` for it:
     |
1    | use std::io::Write;
```

(Yes, some traits are imported by default nonetheless, so that you don't need `use` for popular traits like `Clone` and `Iterator`.)

Trait arguments

Types that implement a trait share a capability. A natural need is hence to write *generic code* that works on all values sharing a capability.
How do we write a function that accepts **any type that implements a given trait**?

```
fn check(val: &impl CmpToZero) {
    if val.is_zero() { println!("Zero") } else { println!("Not zero") };
}
```

`&impl CmpToZero` means “any type that implements the `CmpToZero` trait”.

We can even be more demanding, and request that a given argument implements **multiple traits at once** (i.e., it’s a logical AND, not OR, so that your code can make assumptions about available capabilities):

```
fn f(t: &(impl Trait1 + Trait2), u: &(impl Trait3 + Trait4)) -> usize {}
```

Monomorphization

So when we use `&impl` values we perform **virtual method calls** like in other OOP languages (incurring the cost of dynamic dispatch, virtual method tables, etc.), right?

Nope!

When you use `&impl` function parameters:

- The *function* is instantiated with the *actual type* of the passed value (**monomorphization**).
- If the function is called (statically) with N different types, N different copies of it will be instantiated at *compile time*.
- At *runtime* the type-appropriate version of the function is called, with zero overhead w.r.t. a normal function (i.e., one without `&impl` args).

Trait objects

But Rust *also* supports OOP-style virtual methods via **trait objects**, denoted as `dyn Trait`. A value of type `dyn Trait` has a **dynamic type** known only at runtime; at compile time we only know that the type implements *Trait*.

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

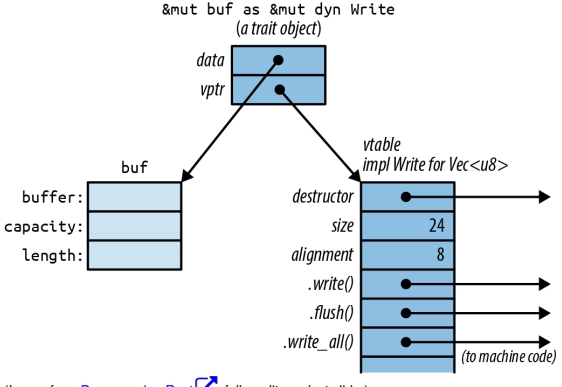
Why is that?

In languages like Java most values are accessed indirectly via fixed-size references. In Rust references are fixed-size too, but they are not the default value to manipulate values and referencing is explicit:

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

Welcome `writer`, our first a trait object.

Memory layout of trait objects



(Image from Programming Rust, full credits on last slide.)

- In memory a trait object is a *fat pointer* consisting of two pointers
- One pointer to the object value (with the common trifecta: buffer, capacity, length)
- One pointer to a **virtual table** (vtable), used for the dynamic dispatch of method invocations on the trait object.
  - Similar to C++: vtable is generated once at compile time and shared by all objects of the same type.
  - Unlike C++: decoupling of vptr from the struct, to avoid struct size inflation when implementing many traits.

(You have encountered other fat pointers in the past: `&str` and `&[]`. They are the same kind of beast, except those were data+length pointers vs data+vtable in this case.)

Trait object arguments

Functions can expect trait object arguments, which are then automatically created by the compiler when needed:

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

use std::fs::File;
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?;  // automatic trait object creation: &mut File -> &mut dyn Write

let mut bytes = vec![];
say_hello(&mut bytes)?;
assert_eq!(bytes, b"hello world\n");  // &mut Vec<u8> -> &mut dyn Write
```

Creating a trait object implies creating the fat pointer, making it point on the one side to the actual value and on the other to the right vtable for its actual type.

Generic functions and type parameters

We can rewrite our `say_hello` function as a **generic** (AKA polymorphic) **function** like this:

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

The function is now (explicitly) parametric over a **type parameter** `W`. In the rest of the definition `W` stands for *some* type, which might vary across function invocations.

In source code, only the function *signature* has changed.
At runtime, the trait object is gone and we get monomorphization back! (like with `&impl`).

Thanks to **type inference**, in most cases you do not need to spell out the actual type `W` upon invocation, it is automatically inferred. But *you can* spell it out if needed, with the **turbofish syntax** `::<type>`:

```
// say_hello(&mut local_file);          // implicitly inferred type parameter
   say_hello::<File>(&mut local_file)?;  // explicit type parameter File for W
```

Trait bounds

In `fn say_hello<W: Write>(...)`, the postfix `: Write` notation is a **trait bound** on `W`, which means "a type `W` that (at least) implements the `Write` trait" (i.e., in type theory, `W` must be a *sub-type* of `Write`).

For multiple traits the syntax we have used before:

```
fn f(t: &(impl Trait1 + Trait2), u: &(impl Trait3 + Trait4)) -> usize { ... }
```

with trait bounds becomes:

```
fn f<T: Trait1 + Trait2, U: Trait3 + Trait4>(t: &T, u: &U) -> usize { ... }
```

Where clauses can be used to make function signatures more readable when you have multiple arguments and/or traits:

```
fn f<T, U>(t: &T, u: &U) -> usize
    where T: Trait1 + Trait2,      // same trait bounds as before
          U: Trait3 + Trait4
{ ... }
```

Trait objects vs generics

Advantages of generics over trait objects:

- Speed (no dynamic dispatch, zero-cost abstraction)
- Multiple trait bounds are easier to express (`&mut (dyn Trait1 + Trait2 + Trait3)` does not work)

Rule of thumb:

- Use trait objects when you need to group together values of mixed types (that share a trait). The additional flexibility could be worth the additional trait object/fat pointer.
- Use trait objects if compile-time bloat becomes an issue (and the additional fat pointer at runtime is not).
- Use generics otherwise.

Standard traits

Standard traits

The Rust stdlib defines several *noteworthy traits*. Here are some of them:

Traits	Purpose
`Debug`, `Display`	formatting
`[Partial]Eq`, `[Partial]Ord`	comparison
`Sized`	(marker trait) values that have a fixed compile-time size
`Copy`, `Clone`	values that can be copied
`Deref`, `DerefMut`	smart pointers
`AsRef`, `AsRefMut`	cheap reference-to-reference conversions
`[Try]From`, `[Try]Into`	type conversions
`Drop`	custom destructor hooks
`Send`, `Sync`	concurrency

You have already used some of them in the past and you will use more in the future.
Let's review some of them, in light of what you now know about traits.

Formatting traits — Display

The `Display` trait provides **formatting to string** capabilities and is meant for **user-facing output**.

```rust
use std::fmt;

struct Point { x: i32, y: i32 };

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

let origin = Point { x: 0, y: 0 };

println!("The origin is: {}", origin);
println!("The origin is: {origin}");  // same but nicer with interpolation syntax
```

```
The origin is: (0, 0)
The origin is: (0, 0)
```

Automatic trait derivation and the Debug trait

- Some traits can be automatically derived for you (= “implemented automatically at compile time”) using the `#[derive]` macro, provided that trait-specific preconditions are fulfilled.
- The `Debug` trait is similar to `Display` but meant for **programmer-facing output**.
- `Debug` can be derived automatically for compound types, *provided that all fields implement* `Debug` (which is the case for most stdlib types).

```rust
#[derive(Debug)]
struct Point { x: i32, y: i32 }
let c = Point { x: 0, y: 0 };
println!("c = {c}");      // user-facing output (Display)
println!("c = {c:?}");    // programmer-facing output (Debug)
println!("c = {c:#?}");   // Debug output with pretty printing (e.g., struct indentation)
```

```
c = (0, 0)
c = Point { x: 0, y: 0 }
c = Point {
  x: 0,
  y: 0,
}
```

Comparisons traits

A tour of common gotchas when unit testing your Rust code:

Take #1:

```
1  struct Point { x: i32, y: i32 }
2
3  fn main() {
4      let p1 = Point { x: 1, y: 2 };
5      let p2 = Point { y: 2, x: 1 };
6      assert_eq!(p1, p2);
7  }
```

```
error[E0277]: `Point` doesn't implement `Debug`
```

Makes sense, in case of test failure the test harness needs to be able to show (to a programmer) the values involved in the failed assertion.

Let's add `#[derive(Debug)]` and try again. (Note: derivation preconditions are respected.)

Comparisons traits (cont.)

Take #2:

```
#[derive(Debug)]
struct Point { x: i32, y: i32 }
fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { y: 2, x: 1 };
    assert_eq!(p1, p2);
}
```

```
error[E0369]: binary operation `==` cannot be applied to type `Point`
2 | struct Point {
  | ^^^^^^^^^^^^ must implement `PartialEq<_>`
```

`PartialEq` is the trait denoting values that can be **compared for equality**. Implementers must provide a `fn eq(&self, other: &Rhs) -> bool` method, which is implicitly invoked by the `==` operator. It makes sense for such a requirement to exists for, well, an *equality* assertion.

`PartialEq` can be derived on structs and enums, provided all sub-fields are comparable as well. So let's derive it!

Comparisons traits (cont.)

Take #3:

```rust
#[derive(Debug, PartialEq)]
struct Point { x: i32, y: i32 }
fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { y: 2, x: 1 };
    assert_eq!(p1, p2);
}
```

It works!

N.B. the comparison is *partial* in the mathematical sense of a partial equivalence relation, which allows for non-comparable values like `NaN != NaN` for floats. Implementers must ensure that `a != b` if and only if `!(a == b)` (which does not hold for floats).

The stricter `Eq` trait exists as well (and is not implemented for floats).

`Ord` and `PartialOrd` are analogous traits for **(partial) order relations**.

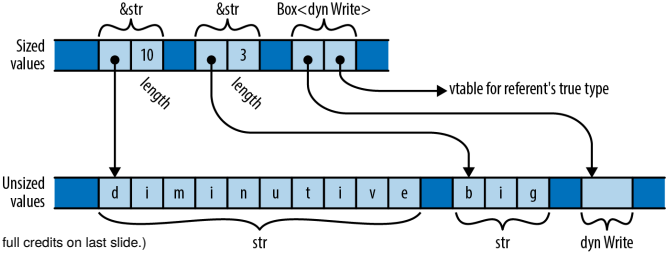
Sized

A **sized type** is a type whose values have all the same size in memory. Most Rust types are sized: numbers (obviously), enums (which always takes enough space for its largest variant), even `Vec<T>` (think of the triple pointer/capacity/length, *not* of the referenced heap memory).

`Sized` is a **marker trait**, i.e., an empty trait which is just an indication for the type system.

All sized types implement the `Sized` trait automatically, you cannot implement it yourself.

Very few **unsized type** exists, e.g.: string slices `str` and array slices `[T]` (without `&`).



(Image from Programming Rust, full credits on last slide.)

Sized (cont.)

- You can't store unsized values in variables or pass them as arguments.
- Basically the only think you can do with unsized values is reference them via (sized) pointers or references.
- Given there's so little you can do with unsized values, **type variables are bound by default to Sized**.
  - When you write `<T>`, Rust interprets it implicitly as `<T: Sized>`.
  - In those rare cases you need to relax the constraint you can write `<T: ?Sized>`, meaning "not necessarily sized" (mnemonic: *questionably sized*).
  - (To actually work in practice your code will need to always access unsized values through (sized) references of some kind.)

Clone

`Clone` indicates that values of a given type can be copied via the `clone` method: [2] [3]

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

- Expectation: the new copy is *completely independent* from the old one, which usually requires transitive/"deep copying" to implement `clone()` properly.
  - E.g., cloning a `Vec<String>` requires copying both the vector itself and all contained strings.
- `Clone` is derivable if all fields are `Clone`, making the common need of *recursive cloning* easy:

```
1 #[derive(Clone, Debug, PartialEq)]
2 struct Point { x: i32, y: i32 }
3
4 let x = Point { x: 0, y: 0 };
5 let y = x.clone();
6 assert_eq!(x, y);
```

[2] the `Self` type in traits can be used to reference the type of the implementer

[3] `T1: T2` makes T1 a **subtrait** ot T2, i.e., all T1 values are also T2, T1 is more specific, T2 more general

Copy

`Copy`—which you have encountered already on types that are exempt to the move rule—is another marker trait, whose full definition is just:

```
trait Copy: Clone { }
```

You can implement (and even automatically derive) `Copy` for your types, but only if they can be safely copied with a **shallow byte-for-byte copy**. E.g., they cannot own any heap-allocated memory. Think carefully before making types `Copy`, as byte copies might become expensive for large structures.

If you really decide to, here's an example:

```
1 #[derive(Clone, Copy)]  // when derived together, Clone is implemented using memory copy
2 struct S { a: u32 };
3 let x = S { a: 42 };
4 let y = x;  // no move occurs here
5 // both x and y can still be used here (y is a copy of x)
```

Q: can you implement `Copy` for `Vec`?

Boxes

The `Box<T>` type is a *generic type* whose values own heap memory.

```
let t = (44, "cats");
let b = Box::new(t);  // allocate enough heap memory to hold t; move t to b
// when b goes out of scope, drop b and free t's heap memory
```

- `Box` is generic and uses `T: ?Sized` as trait bound.
- Hence you can put most Rust values (including *unsized* ones!) in a box.
- (`Box` values themselves are `Sized` though, phew.)

Q: can `Box<T>` be `Copy`?

Nope! For the same reason `Vec<T>` cannot.
As a rule of thumb: any type whose values need special handling when dropped (freeing heap memory in the case of `Box`) cannot be `Copy`. (It is indeed forbidden by the compiler to implement both `Drop` and `Copy`.)

Deref

Boxes are generic smart pointers that: (1) move data to the heap upon creation, (2) free heap memory upon destruction. It would be nice to manipulate them as regular Rust references! That is precisely what the `Deref`/`DerefMut` traits allow you to do:

```
trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
trait DerefMut: Deref {  // subtrait: if you can borrow mutably, you can borrow readonly
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

Given that: (1) the `deref` method is called implicitly upon dereferencing (`*` operator) and (2) smart pointer types like `Box` implement the `Deref` trait, the following works out of the (errrm….) box:

```
use std::ops::Deref;  // to pull deref() in scope
let b = Box::new(42);
assert_eq!(*(b.deref()), 42);  // ugh
assert_eq!(*b, 42);            // much nicer!
```

`DerefMut` is the mutable counterpart of `Deref`.

Deref coercion

- Let's reconsider the signature: `fn deref(&self) -> &Self::Target`.
- It can be interpreted as a *conversion* function from references on the LHS to references on the RHS, and is generally expected to be a *cheap* conversion (just follow some pointers).
- Rust takes the liberty of automatically invoking the `deref` method in situations where, *without applying it*, a type error would arise.
- This process is called **deref coercion**[4] and can be repeated multiple times, recursively, e.g.:

```
1 let s = Box::new(String::from("Hello, World!"));
2 assert_eq!(s.find('!'), Some(12));
```

A box does not have a `find` method, so naively line 2 should not type check. To make it type check the compiler first applies a deref coercion `Box -> &String` and then *another* deref coercion `&String -> &str` (`find` is indeed provided by string slices).

[4] In type theory a **coercion** is a function that can be applied to convert values of one type to another, in order to make the type checker happy.

From and Into

You have already encountered `From` in the error-handling lecture. It goes hand-in-hand with `Into` and, together, the two traits provide **conversion capabilities** that *consume* one value (taking owernship) to produce another of a different type (returning ownership of the new value to the caller).

```
trait Into<T>: Sized {
    fn into(self) -> T;
}
trait From<T>: Sized {
    fn from(other: T) -> Self;
}
```

`Into` is generally used to **make your functions more flexible** in what they accept (replacing function overloading in OOP languages), e.g.:

```
use std::net::Ipv4Addr;
fn ping<A>(address: A) -> std::io::Result<bool>
    where A: Into<Ipv4Addr>  // do not insist on receiving an IPv4 address,
                             // anything that can be *converted into one* would do
{
    let ipv4_address = address.into();
    ...
}
```

From and Into (cont.)

`From` is generally used to provide **constructors** (similar to the use of factories and static methods in OOP languages) of your values from different types, e.g.:

```
1 // Ipv4Addr implements From<[u8;4]> and From<u32>
2 let addr1 = Ipv4Addr::from([66, 146, 219, 98]);  // No need to spell out actual type parameters,
3 let addr2 = Ipv4Addr::from(0xd076eb94_u32);      // thanks to type inference.
```

From/Into *contract:*

- From/Into conversions are not expected to be *cheap*; it is perfectly fine to copy memory and parse data upon `from`/`into`. For **cheap conversions** check out the following traits: `AsRef`, `AsMut`, `Borrow`, `BorrowMut`.
- From/Into conversions are expected to be *infallible* and the trait interface only allows them to panic upon failure. For **fallible conversions** there are the companion `TryFrom`/`TryInto` traits, whose main methods return `Result<Self, Self::Error>`.

Conveniently, `Into` is implemented automatically when the complementary `From` trait is implemented.

More on generics

Generics and type restrictions

The notion of generic code is not specific to traits. It just often happens that your code cannot be *fully* generic and needs to restrict what it is generic on. As traits are the Rust way of structuring the type "hierarchy", *trait bounds* is what we use to specify those constraints.

E.g., a generic function to return the minimum of two values needs to be able to compare them:

```
fn min<T: Ord>(value1: T, value2: T) -> T {
    if value1 <= value2 {
        value1
    } else {
        value2
    }
}
```

Generic structs

Some code can be *very* generic though. For instance you have already use the `Vec<T>` type; vectors can contain almost[5] any type.

Of course you can also define your own generic types. **Generic structs** are quite common, here is an example:

```
1 pub struct Queue<T> {
2     older: Vec<T>,
3     younger: Vec<T>
4 }
```

for any (sized) type `T` you can have a `Queue<T>`. The type parameter `T` is reused in the definition as we have seen in the case of traits: a `Queue<u32>` will have `younger` and `older` fields, each of type `Vec<u32>`.

[5] Remember that a type parameter `T` is bound by default by `<T: Sized>`

Generic structs (cont.)

`impl` blocks for generic types can implement **generic associated functions** that exist for all `T`:

```
impl<T> Queue<T> {  // note the double <T>
    pub fn new() -> Self {
        Queue { older: Vec::new(), younger: Vec::new() }
    }
    pub fn push(&mut self, t: T) {
        self.younger.push(t);
    }
}
let mut q1: Queue<bool> = Queue::new();
let mut q2: Queue<f64> = Queue::new();
```

as well as **type-specific associated functions** that exist only for some `T`:

```
impl Queue<f64> {
    fn total(&self) -> f64 { ... }
}

// assert_eq!(q1.total(), 0_f64);
// error[E0599]: no method named `total` found for struct `Queue<bool>` in the current scope
assert_eq!(q2.total(), 0_f64);
```

Generic traits and operator overloading

Operator overloading is implemented in Rust on top of traits.

- Some operators[6] are mapped to method invocations, e.g., `a + b` becomes `a.add(b)`;
- The involved methods are defined in trait interfaces;
- Custom types can implement the relevant traits to overload the meaning of an existing operator.

We have already seen an example of this for the `Deref` trait and the `*` dereferencing operator. As another example, here is the trait that Rust uses for the addition operator `+`:

```
1 pub trait Add<RHS> {  // RHS (right hand side) is the type parameter for the right operand
2     type Output;  // resulting type after addition
3     fn add(self, rhs: RHS) -> Self::Output;
4 }
```

`Add` is a **generic trait**. It can be implemented multiple times for each actual type `RHS`. For operator overloading in general you will implement it once for each right operand type you want to support.

For a fun complete example see: https://doc.rust-lang.org/rust-by-example/trait/ops.html.

[6] See https://doc.rust-lang.org/std/ops/index.html#traits for a mapping between operators and traits.

Iterators

Traits as type relationships

- Most of the traits seen so far say something about a specific type
  - e.g., `Display` that its values can be pretty-printed, `Clone` that they can be cloned.
- Some traits can establish **relationships between types**.
  - *Generic traits*, like `std::ops::Add` for addition overloading, are one way to do so: they establish a relationship between the LHS and RHS operand types.
- **Associated types** are types defined *within* a trait (rather than as a type parameter) that must be provided by each trait implementation.
- Iterator and iterator types are a prime example of trait associated types, let's have a look.

The Iterator trait

Most modern languages provide an idiomatic way to iterate over value sequences. The Rust way of doing so is based on the `Iterator` trait:

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // ... many default methods, discussed later in this lecture
}
```

- `Item` is an *associated type*, representing what you receive at each iteration *step*.
- `next` is called to obtain the next element in the iteration; `None` means you reached the end.
  - Note the mutable borrow via the `&mut self` argument, needed because iterators are *stateful*.
- Note that the trait interface does not define *how to obtain* an iterator from something that *is not* an iterator itself, only what you can do with an iterator.

Iterator implementation — example

```rust
struct Counter { value: u8 }

impl Iterator for Counter {
    type Item = u8;

    fn next(&mut self) -> Option<Self::Item> {
        match self.value {
            255 => None,
            _ => { self.value += 1; Some(self.value) }
        }
    }
}
fn main() {
    let mut c = Counter { value: 0 };
    while let Some(v) = c.next() {
        println!("v = {v:?}");
    }
}
```

```
v = 1
v = 2
...
v = 255
```

IntoIterator and for loops

Types that have a natural way of being iterated over are called **iterables**. They can implement the companion `IntoIterator` trait to provide an easy way for *obtaining* an iterator.

```
trait IntoIterator where Self::IntoIter: Iterator<Item=Self::Item> {
    type Item;                // required: iteration item type
    type IntoIter: Iterator; // required: iterator type
    fn into_iter(self) -> Self::IntoIter;  // an iterator please!
}
```

Rust's for loops use `IntoIterator` transparently, hence the following:

```
let salad = vec!["tomato", "lettuce", "onion"];  // Vec<T> implements IntoIterator
for ingredient in &salad {
    println!("{}", ingredient);
}
```

becomes:

```
let mut iterator = (&salad).into_iter();  // an iterator please!
while let Some(ingredient) = iterator.next() {
    println!("{}", ingredient);
}
```

N.B.: iterators automatically implements `IntoIterator`, so you can pass them to `for` too.

Obtaining iterators

- Most collection and slice types provide `iter` and `iter_mut` methods to obtain iterators on contained values.
- `iter` returns an iterator over shared references; `iter_mut` over mutable ones

```
let v = vec![4, 20];
let mut iterator = v.iter();  // request an iterator over &i32 values
assert_eq!(iterator.next(), Some(&4));
assert_eq!(iterator.next(), Some(&20));
assert_eq!(iterator.next(), None);
```

```
let mut v = vec![4, 20];
let mut iterator = v.iter_mut(); // request an iterator over &mut i32 values
let e: &mut i32 = iterator.next().unwrap();
*e = 42;
assert_eq!(v, [42, 20]);
```

- This is just a naming convention. If there is more than one way to iterate over a sequence-like type, the documentation will provide guidance. E.g., `&str` does not have `.iter()` and provides separate `.bytes()` and `.chars()` methods instead.

Iterating over collections

Most collection types provide different implementations of `IntoIterator`, depending on how you reference the collection:

- Given a *shared reference* (to the collection object), you obtain an iterator producing *shared references to the contained items*.
- Given a *mutable reference*, an iterator producing mutable references to the contained items.
- Given a collection *passed by value*, an iterator that *takes ownership* of the collection (consuming it) and produces the contained items.

Given that `for in` implicitly uses `IntoIterator`, mind the difference between:

```
for element in &collection { ... }      // shared refs
for element in &mut collection { ... }  // mutable refs
for element in collection { ... }       // contained items (consuming collection!)
```

The iterator language

Iterator adapters

- The Iterator trait requires very little from implementers (`Item` and `next`) and provides in exchange many methods with default implementations.
- Taken together those methods form a **DSL** (Domain Specific Language) **for iterator manipulation** that can express concisely programming logic that is then executed in an efficient manner.
- In the following we will take a brief example-based tour of the most important iterators methods, focusing on two categories:
  - **Iterator adapters,** which consume iterators to build new ones adding useful behaviour (similar to filters in pipeline architectures, like UNIX CLI), and
  - **Iterator consumers,** which consume iterator to produce a “final” result (similar to sinks in pipeline architectures).

Lambda expressions — preview

- **Lambda expressions** (from the λ-calculus) are expressions that denote anonymous functions.
- They are also called **closures** in the common case where they capture context from the enclosing environment.
- You will learn all about closures in a future lecture, but as lambda expressions are frequently used together with iterator adapters, we anticipate here their basic syntax.
- `|x| x + 42` is a valid Rust expression, denoting an **anonymous function** that takes an argument `x` as input and returns the sum of it and 42 as output.
- Multiple arguments can be accepted on the LHS with the `|x, y, z|` syntax.
- A code block `{ ... }` can be used on the RHS if you need to execute instructions before returning a value.

```
1  assert_eq!((|x| x + 40)(2), 42);  // anonymous lambda expression applied right away
2
3  let prod = |x, y| x * y;  // lambda expression bound to a name
4  assert_eq!(prod(6, 7), 42);
5
6  let peek_next = |x| { println!("got {x}"); x + 1 };  // lambda expression with a block
7  assert_eq!(peek_next(41), 42);  // prints "got 41" as side effect
```

map

The `map` adapter takes a function and consumes an iterator to produce a new one where each element is the result of applying the function to the corresponding element of the original iterator.

```
fn map<B, F>(self, f: F) -> impl Iterator<Item=B>
    where Self: Sized, F: FnMut(Self::Item) -> B;
```

```
1 let text = " ponies \n giraffes\niguanas \nsquid".to_string();
2 let v: Vec<&str> = text.lines()
3     .map(str::trim)
4     .collect();
5 assert_eq!(v, ["ponies", "giraffes", "iguanas", "squid"]);
```

filter

`filter` produces a new iterator where only the elements satisfying a given predicate (passed as a function argument to `filter`) are produced.

```
fn filter<P>(self, predicate: P) -> impl Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

```
1 let text = " ponies \n giraffes\niguanas \nsquid".to_string();
2 let v: Vec<&str> = text.lines()
3     .map(str::trim)
4     .filter(|s| *s != "iguanas")
5     .collect();
6 assert_eq!(v, ["ponies", "giraffes", "squid"]);
```

filter_map

`filter_map` combines `filter` and `map` into a single more expressive adapter:

```
fn filter_map<B, F>(self, f: F) -> impl Iterator<Item=B>
    where Self: Sized, F: FnMut(Self::Item) -> Option<B>;
```

```
use std::str::FromStr;
let text = "1\nfrond .25 289\n3.1415 estuary\n";
for number in text.split_whitespace()
    .filter_map(|w| f64::from_str(w).ok())
{
    print!("{:4.2}, ", number.sqrt());
}  // prints: "1.00, 0.50, 17.00, 1.77, "
```

which could be preferable over multiple `filter`/`map` passes:

```
text.split_whitespace()
    .map(|w| f64::from_str(w))
    .filter(|r| r.is_ok())
    .map(|r| r.unwrap())
```

`flat_map` is a similar adapter, where each `f` call can return multiple elements.

flatten

`flatten` concatenates together the items produced by multiple iterators:

```
fn flatten(self) -> impl Iterator<Item=Self::Item::Item>
    where Self::Item: IntoIterator;
```

```
let data = vec![vec![1, 2, 3, 4], vec![5, 6]];
let flattened = data.into_iter().flatten().collect::<Vec<u8>>();
assert_eq!(flattened, &[1, 2, 3, 4, 5, 6]);
```

Q: why does the following work:

```
assert_eq!(vec![None, Some("day"), None, Some("one")].into_iter()
           .flatten()
           .collect::<Vec<_>>(),
           vec!["day", "one"]);
```

Because `Option` comes with a surprisingly useful `.iter()` method!

```
let x = Some(4);
assert_eq!(x.iter().next(), Some(&4));
let x: Option<u32> = None;
assert_eq!(x.iter().next(), None);
```

take and take_while

`take` and `take_while` allow to cut an iterator short.

```
fn take(self, n: usize) -> impl Iterator<Item=Self::Item>
    where Self: Sized;

fn take_while<P>(self, predicate: P) -> impl Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

```
let message = "To: author\r\n\
               From: superego <editor@example.com>\r\n\
               \r\n\
               Did you get any writing done today?\r\n\
               When will you stop wasting time plotting fractals?\r\n";
for header in message.lines().take_while(|l| !l.is_empty()) {
    println!("{}" , header);
}
```

```
To: author
From: superego <editor@example.com>
```

`skip` and `skip_while` are the complementary adapters to skip initial items.

rev and reversible iterators

Some iterators (not all: why?), called **reversible iterators**, can produce elements from either end of the underlying sequence. They implement the subtrait:

```
trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;
}
```

Reversible iterators provide a `rev` adapter to reverse iteration order:

```
fn rev(self) -> impl Iterator<Item=Self>
    where Self: Sized + DoubleEndedIterator;
```

```
1 let meals = ["breakfast", "lunch", "dinner"];
2 let mut iter = meals.iter().rev();
3 assert_eq!(iter.next(), Some(&"dinner"));
4 assert_eq!(iter.next(), Some(&"lunch"));
5 assert_eq!(iter.next(), Some(&"breakfast"));
6 assert_eq!(iter.next(), None);
```

chain, enumerate, zip

`chain` concatenates multiple iterators.

```
let v: Vec<i32> = (1..4).chain(vec![20, 30, 40]).collect();
assert_eq!(v, [1, 2, 3, 20, 30, 40]);
```

`enumerate` adds an integer enumeration on the left in a pair:

```
let a = ['a', 'b'];
let mut iter = a.iter().enumerate();
assert_eq!(iter.next(), Some((0, &'a')));
assert_eq!(iter.next(), Some((1, &'b')));
assert_eq!(iter.next(), None);
```

`zip` combines two iterators into one over item pairs:

```
let a1 = [1, 2];
let a2 = [4, 5, 6];
let mut iter = a1.iter().zip(a2.iter());  // stop at the end of the shortest one
assert_eq!(iter.next(), Some((&1, &4)));
assert_eq!(iter.next(), Some((&2, &5)));
assert_eq!(iter.next(), None);
```

cloned, copied

Due to Rust memory model restrictions, you might encounter situations in which you need to `iter.map(|item| item.clone())` (but beware of the runtime cost!).

For convenience, a predefined `.cloned()` adapter to do so exists:

```
1 let a = ['1', '2', '3', 'ϖ'];
2 assert_eq!(a.iter().next(), Some(&'1'));
3 assert_eq!(a.iter().cloned().next(), Some('1'));
```

`.copied()` is similar, for copying (`Copy` types only) instead of cloning (`Clone`).

Consuming iterators

- Several **iterator consumer** methods (or sinks) exist on iterators, restricted as needed by appropriate trait bounds.
- Here is just an overview, with a few examples in the following slides:

Purpose	Method(s)	Bounds
length measurement	`count`	
arithmetic operations	`sum`, `product`	`Sum`, `Product`
least/greatest element	`min`, `max`	`Ord`
quantifiers	`any`, `all`	
general recursion	`fold`, `rfold`	
selection	`nth`, `last`, `find`	
collection	`collect`	

- For more—on both iterator consumers and adapters—refer to the documentation of the `Iterator` trait and that of the itertools crate that provides even more advanced iterator manipulation functionalities.

Consuming iterators — examples

```rust
fn factorial(n: u64) -> u64 {
    (1..=n).product()
}
assert_eq!(factorial(20), 2_432_902_008_176_640_000);
```

```rust
assert_eq!([-2, 0, 1, 0, -2, -5].iter().min(), Some(&-5));
```

```rust
let id = "Iterator";
assert!( id.chars().any(char::is_uppercase));
assert!(!id.chars().all(char::is_uppercase));
```

```rust
fn fold<A, F>(self, init: A, f: F) -> A
    where Self: Sized, F: FnMut(A, Self::Item) -> A;
```

```rust
let a = [5, 6, 7, 8, 9, 10];

assert_eq!(a.iter().fold(0, |n, _| n + 1), 6);        // count
assert_eq!(a.iter().fold(0, |n, i| n + i), 45);       // sum
assert_eq!(a.iter().fold(1, |n, i| n * i), 151_200);  // product
assert_eq!(a.iter().cloned().fold(i32::min_value(), std::cmp::max), 10);  // max
```

Takeaways

Takeaways

- Structures and enumerations complete the palette of Rust compound types.
- They can have associated implementation blocks, providing abstract data types.
- Code reuse is supported in Rust by two forms of polymorphism: generics and traits.
- Traits provide a way to define standard interfaces for capabilities shared by multiple types and implement them.
- Generics allow to write code once and use it in the context of different types, with no performance penalties thanks to monomorphization.
- A rich collection of traits already exist in the stdlib, providing a consistent way of addressing common needs.
- Iterators, implemented as traits, provide an idiomatic way of iterating sequence-like types, including collections.
- Iterators also provide an expressive DSL that can express powerful computations in a concise manner wit no performance penalties.

Credits

- Some slides have been adapted from the SE302b course at Télécom Paris, by G. Duc and S. Tardieu.
- Some examples and images have been adapted from Chapters 11, 13, and 15 of the Programming Rust book.