Effective Rust

35 Specific Ways to Improve Your Rust Code

David Drysdale

A printed and ebook version of this book is published by O'Reilly Media:

Release date: April 2024
Page count: 280 pages
ISBN: 9781098151409

Available at:

- US Amazon.com
- GB Amazon.co.uk
- US Barnes & Noble
- GB Waterstones
- and many other book stores.

Preface

> "The code is more what you'd call *guidelines* than actual rules." – Hector Barbossa

In the crowded landscape of modern programming languages, Rust is different. Rust offers the speed of a compiled language, the efficiency of a non-garbage-collected language, and the type safety of a functional language—as well as a unique solution to memory safety problems. As a result, Rust regularly polls as the most loved programming language.

The strength and consistency of Rust's type system means that if a Rust program compiles, there is already a decent chance that it will work—a phenomenon previously observed only with more academic, less accessible languages such as Haskell. If a Rust program compiles, it will also work *safely*.

This safety—both type safety and memory safety—does come with a cost, though. Despite the quality of the basic documentation, Rust has a reputation for having a steep on-ramp, where newcomers have to go through the initiation rituals of fighting the borrow checker, redesigning their data structures, and being befuddled by lifetimes. A Rust program that compiles may have a good chance of working the first time, but the struggle to get it to compile is real—even with the Rust compiler's remarkably helpful error diagnostics.

Who This Book Is For

This book tries to help with these areas where programmers struggle, even if they already have experience with an existing compiled language like C++. As such—and in common with other *Effective <Language>* books—this book is intended to be the *second* book that a newcomer to Rust might need, after they have already encountered the basics elsewhere—for example, in *The Rust Programming Language* (Steve Klabnik and Carol Nichols, No Starch Press) or *Programming Rust* (Jim Blandy et al., O'Reilly).

However, Rust's safety leads to a slightly different slant to the Items here, particularly when compared to Scott Meyers's original *Effective C++* series. The C++ language was (and is) full of footguns, so *Effective C++* focused on a collection of advice for avoiding those footguns, based on real-world experience creating software in C++. Significantly, it contained *guidelines* not rules, because guidelines have exceptions—providing the detailed rationale for a guideline allows readers to decide for themselves whether their particular scenario warranted breaking the rule.

The general style of giving advice together with the *reasons* for that advice is preserved here. However, since Rust is remarkably free of footguns, the Items here concentrate more on the

concepts that Rust introduces. Many Items have titles like "*Understand…*" and "*Familiarize yourself with…*", and help on the journey toward writing fluent, idiomatic Rust.

Rust's safety also leads to a complete absence of Items titled "*Never…*". If you really should never do something, the compiler will generally prevent you from doing it.

Rust Version

The text is written for the 2018 edition of Rust, using the stable toolchain. Rust's back-compatibility promises mean that any later edition of Rust, including the 2021 edition, will still support code written for the 2018 edition, even if that later edition introduces breaking changes. Rust is now also stable enough that the differences between the 2018 and 2021 editions are minor; none of the code in the book needs altering to be 2021-edition compliant (but Item 19 includes one exception in which a later version of Rust allows new behavior that wasn't previously possible).

The Items here do not cover any aspects of Rust's `async` functionality, as this involves more advanced concepts and less stable toolchain support—there's already enough ground to cover with synchronous Rust. Perhaps an *Effective Async Rust* will emerge in the future…

The specific `rustc` version used for code fragments and error messages is 1.70. The code fragments are unlikely to need changes for later versions, but the error messages may vary with your particular compiler version. The error messages included in the text have also been manually edited to fit within the width constraints of the book but are otherwise as produced by the compiler.

The text has a number of references to and comparisons with other statically typed languages, such as Java, Go, and C++, to help readers with experience in those languages orient themselves. (C++ is probably the closest equivalent language, particularly when C++11's move semantics come into play.)

Navigating This Book

The Items that make up the book are divided into six chapters:

- *Chapter 1 — Types*: Suggestions that revolve around Rust's core type system
- *Chapter 2 — Traits*: Suggestions for working with Rust's traits
- *Chapter 3 — Concepts*: Core ideas that form the design of Rust
- *Chapter 4 — Dependencies*: Advice for working with Rust's package ecosystem
- *Chapter 5 — Tooling*: Suggestions for improving your codebase by going beyond just the Rust compiler

- *Chapter 6 — Beyond Standard Rust*: Suggestions for when you have to work beyond Rust's standard, safe environment

Although the "Concepts" chapter is arguably more fundamental than the "Types" and "Traits" chapters, it is deliberately placed later in the book so that readers who are reading from beginning to end can build up some confidence first.

Conventions Used in This Book

The following typographical conventions are used in this book:

- *Italic*: Indicates new terms, URLs, email addresses, filenames, and file extensions.
- `Constant width`: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

The following markers are used to identify code that isn't right in some way.

Marker	Meaning
	This code does not compile!
	This code does not produce the desired behavior.

Acknowledgments

My thanks go to the people who helped make this book possible:

- The technical reviewers who gave expert and detailed feedback on all aspects of the text: Pietro Albini, Jess Males, Mike Capp, and especially Carol Nichols.
- My editors at O'Reilly: Jeff Bleiel, Brian Guerin, and Katie Tozer.
- Tiziano Santoro, from whom I originally learned many things about Rust.
- Danny Elfanbaum, who provided vital technical assistance for dealing with the AsciiDoc formatting of the book.
- Diligent readers of the original web version of the book, in particular:
  - Julian Rosse, who spotted dozens of typos and other errors in the online text.
  - Martin Disch, who pointed out potential improvements and inaccuracies in several Items.

    - Chris Fleetwood, Sergey Kaunov, Clifford Matthews, Remo Senekowitsch, Kirill Zaborsky, and an anonymous Proton Mail user, who pointed out mistakes in the text.
- My family, who coped with many weekends when I was distracted by writing.

Types

This first chapter of this book covers advice that revolves around Rust's type system. This type system is more expressive than that of other mainstream languages; it has more in common with "academic" languages such as OCaml or Haskell.

One core part of this is Rust's `enum` type, which is considerably more expressive than the enumeration types in other languages and which allows for *algebraic data types*.

The Items in this chapter cover the fundamental types that the language provides and how to combine them into data structures that precisely express the semantics of your program. This concept of encoding behavior into the type system helps to reduce the amount of checking and error path code that's required, because invalid states are rejected by the toolchain at compile time rather than by the program at runtime.

This chapter also describes some of the ubiquitous data structures that are provided by Rust's standard library: `Option`s, `Result`s, `Error`s and `Iterator`s. Familiarity with these standard tools will help you write idiomatic Rust that is efficient and compact—in particular, they allow use of Rust's question mark operator, which supports error handling that is unobtrusive but still type-safe.

Note that Items that involve Rust *traits* are covered in the following chapter, but there is necessarily a degree of overlap with the Items in this chapter, because traits describe the behavior of types.

Item 1: Use the type system to express your data structures

> "who called them programers and not type writers" – @thingskatedid

This Item provides a quick tour of Rust's type system, starting with the fundamental types that the compiler makes available, then moving on to the various ways that values can be combined into data structures.

Rust's `enum` type then takes a starring role. Although the basic version is equivalent to what other languages provide, the ability to combine `enum` variants with data fields allows for enhanced flexibility and expressivity.

Fundamental Types

The basics of Rust's type system are pretty familiar to anyone coming from another statically typed programming language (such as C++, Go, or Java). There's a collection of integer types with specific sizes, both signed (`i8` , `i16` , `i32` , `i64` , `i128`) and unsigned (`u8` , `u16` , `u32` , `u64` , `u128`).

There are also signed (`isize`) and unsigned (`usize`) integers whose sizes match the pointer size on the target system. However, you won't be doing much in the way of converting between pointers and integers with Rust, so that size equivalence isn't really relevant. However, standard collections return their size as a `usize` (from `.len()`), so collection indexing means that `usize` values are quite common—which is obviously fine from a capacity perspective, as there can't be more items in an in-memory collection than there are memory addresses on the system.

The integral types do give us the first hint that Rust is a stricter world than C++. In Rust, attempting to put a larger integer type (`i32`) into a smaller integer type (`i16`) generates a compile-time error:

```
let x: i32 = 42;
let y: i16 = x;
```

```
error[E0308]: mismatched types
  --> src/main.rs:18:18
   |
18 |     let y: i16 = x;
   |            ---   ^ expected `i16`, found `i32`
   |            |
   |            expected due to this
   |
help: you can convert an `i32` to an `i16` and panic if the converted value
      doesn't fit
   |
18 |     let y: i16 = x.try_into().unwrap();
   |                   ++++++++++++++++++++
```

This is reassuring: Rust is not going to sit there quietly while the programmer does things that are risky. Although we can see that the values involved in this particular conversion would be just fine, the compiler has to allow for the possibility of values where the conversion is *not* fine:

```
let x: i32 = 66_000;
let y: i16 = x; // What would this value be?
```

The error output also gives an early indication that while Rust has stronger rules, it also has helpful compiler messages that point the way to how to comply with the rules. The suggested solution raises the question of how to handle situations where the conversion would have to alter the value to fit, and we'll have more to say on both error handling (Item 4) and using `panic!` (Item 18) later.

Rust also doesn't allow some things that might appear "safe", such as putting a value from a smaller integer type into a larger integer type:

```
let x = 42i32; // Integer literal with type suffix
let y: i64 = x;
```

```
error[E0308]: mismatched types
  --> src/main.rs:36:18
   |
36 |     let y: i64 = x;
   |            ---   ^ expected `i64`, found `i32`
   |            |
   |            expected due to this
   |
help: you can convert an `i32` to an `i64`
   |
36 |     let y: i64 = x.into();
   |                   +++++++
```

Here, the suggested solution doesn't raise the specter of error handling, but the conversion does still need to be explicit. We'll discuss type conversions in more detail later (Item 5).

Continuing with the unsurprising primitive types, Rust has a `bool` type, floating point types (`f32`, `f64`), and a unit type `()` (like C's `void`).

More interesting is the `char` character type, which holds a Unicode value (similar to Go's `rune` type). Although this is stored as four bytes internally, there are again no silent conversions to or from a 32-bit integer.

This precision in the type system forces you to be explicit about what you're trying to express—a `u32` value is different from a `char`, which in turn is different from a sequence of UTF-8 bytes, which in turn is different from a sequence of arbitrary bytes, and it's up to you to specify exactly which you mean.[1] Joel Spolsky's famous blog post can help you understand which you need.

Of course, there are helper methods that allow you to convert between these different types, but their signatures force you to handle (or explicitly ignore) the possibility of failure. For example, a Unicode code point can always be represented in 32 bits,[2] so `'a' as u32` is allowed, but the other direction is trickier (as there are some `u32` values that are not valid Unicode code points):

- `char::from_u32`: Returns an `Option<char>`, forcing the caller to handle the failure case.
- `char::from_u32_unchecked`: Makes the assumption of validity but has the potential to result in undefined behavior if that assumption turns out not to be true. The function is marked `unsafe` as a result, forcing the caller to use `unsafe` too (Item 16).

Aggregate Types

Moving on to aggregate types, Rust has a variety of ways to combine related values. Most of these are familiar equivalents to the aggregation mechanisms available in other languages:

- *Arrays*: Hold multiple instances of a single type, where the number of instances is known at compile time. For example, `[u32; 4]` is four 4-byte integers in a row.
- *Tuples*: Hold instances of multiple heterogeneous types, where the number of elements and their types are known at compile time, for example, `(WidgetOffset, WidgetSize, WidgetColor)`. If the types in the tuple aren't distinctive—for example, `(i32, i32, &'static str, bool)`—it's better to give each element a name and use a struct.
- *Structs*: Also hold instances of heterogeneous types known at compile time but allow both the overall type and the individual fields to be referred to by name.

Rust also includes the *tuple struct*, which is a crossbreed of a `struct` and a tuple: there's a name for the overall type but no names for the individual fields—they are referred to by number instead: `s.0`, `s.1`, and so on:

```
/// Struct with two unnamed fields.
struct TextMatch(usize, String);

// Construct by providing the contents in order.
let m = TextMatch(12, "needle".to_owned());

// Access by field number.
assert_eq!(m.0, 12);
```

enums

This brings us to the jewel in the crown of Rust's type system, the `enum`. With the basic form of an `enum`, it's hard to see what there is to get excited about. As with other languages, the `enum` allows you to specify a set of mutually exclusive values, possibly with a numeric value attached:

```
enum HttpResultCode {
    Ok = 200,
    NotFound = 404,
    Teapot = 418,
}

let code = HttpResultCode::NotFound;
assert_eq!(code as i32, 404);
```

Because each `enum` definition creates a distinct type, this can be used to improve readability and maintainability of functions that take `bool` arguments. Instead of:

```
print_page(/* both_sides= */ true, /* color= */ false);
```

a version that uses a pair of `enum`s:

```
pub enum Sides {
    Both,
    Single,
}

pub enum Output {
    BlackAndWhite,
    Color,
}

pub fn print_page(sides: Sides, color: Output) {
    // ...
}
```

is more type-safe and easier to read at the point of invocation:

```
print_page(Sides::Both, Output::BlackAndWhite);
```

Unlike the `bool` version, if a library user were to accidentally flip the order of the arguments, the compiler would immediately complain:

```
error[E0308]: arguments to this function are incorrect
   --> src/main.rs:104:9
    |
104 | print_page(Output::BlackAndWhite, Sides::Single);
    | ^^^^^^^^^^ --------------------  ------------- expected `enums::Output`,
    |            |                                   found `enums::Sides`
    |            |
    |            expected `enums::Sides`, found `enums::Output`
    |
note: function defined here
   --> src/main.rs:145:12
    |
145 |     pub fn print_page(sides: Sides, color: Output) {
    |            ^^^^^^^^^^ ------------  -------------
help: swap these arguments
    |
104 | print_page(Sides::Single, Output::BlackAndWhite);
    |             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Using the newtype pattern—see Item 6—to wrap a `bool` also achieves type safety and maintainability; it's generally best to use the newtype pattern if the semantics will always be Boolean, and to use an `enum` if there's a chance that a new alternative—e.g., `Sides::BothAlternateOrientation` —could arise in the future.

The type safety of Rust's `enum` s continues with the `match` expression:

```
let msg = match code {
    HttpResultCode::Ok => "Ok",
    HttpResultCode::NotFound => "Not found",
    // forgot to deal with the all-important "I'm a teapot" code
};
```

```
error[E0004]: non-exhaustive patterns: `HttpResultCode::Teapot` not covered
  --> src/main.rs:44:21
   |
44 |     let msg = match code {
   |                     ^^^^ pattern `HttpResultCode::Teapot` not covered
   |
note: `HttpResultCode` defined here
  --> src/main.rs:10:5
   |
7  | enum HttpResultCode {
   |      --------------
...
10 |     Teapot = 418,
   |     ^^^^^^ not covered
   = note: the matched value is of type `HttpResultCode`
help: ensure that all possible cases are being handled by adding a match arm
      with a wildcard pattern or an explicit pattern as shown
   |
46 ~         HttpResultCode::NotFound => "Not found",
47 ~         HttpResultCode::Teapot => todo!(),
   |
```

The compiler forces the programmer to consider *all* of the possibilities that are represented by the `enum`,[3] even if the result is just to add a default arm `_ => {}`. (Note that modern C++ compilers can and do warn about missing `switch` arms for `enum`s as well.)

`enum`s with Fields

The true power of Rust's `enum` feature comes from the fact that each variant can have data that comes along with it, making it an aggregate type that acts as an *algebraic data type* (ADT). This is less familiar to programmers of mainstream languages; in C/C++ terms, it's like a combination of an `enum` with a `union` —only type-safe.

This means that the invariants of the program's data structures can be encoded into Rust's type system; states that don't comply with those invariants won't even compile. A well-designed `enum` makes the creator's intent clear to humans as well as to the compiler:

```
use std::collections::{HashMap, HashSet};

pub enum SchedulerState {
    Inert,
    Pending(HashSet<Job>),
    Running(HashMap<CpuId, Vec<Job>>),
}
```

Just from the type definition, it's reasonable to guess that `Job`s get queued up in the `Pending` state until the scheduler is fully active, at which point they're assigned to some per-CPU pool.

This highlights the central theme of this Item, which is to use Rust's type system to express the concepts that are associated with the design of your software.

A dead giveaway for when this is *not* happening is a comment that explains when some field or parameter is valid:

```
pub struct DisplayProps {
    pub x: u32,
    pub y: u32,
    pub monochrome: bool,
    // `fg_color` must be (0, 0, 0) if `monochrome` is true.
    pub fg_color: RgbColor,
}
```

This is a prime candidate for replacement with an `enum` holding data:

```
pub enum Color {
    Monochrome,
    Foreground(RgbColor),
}

pub struct DisplayProps {
    pub x: u32,
    pub y: u32,
    pub color: Color,
}
```

This small example illustrates a key piece of advice: **make invalid states inexpressible in your types**. Types that support only valid combinations of values mean that whole classes of errors are rejected by the compiler, leading to smaller and safer code.

Ubiquitous `enum` Types

Returning to the power of the `enum`, there are two concepts that are so common that Rust's standard library includes built-in `enum` types to express them; these types are ubiquitous in Rust code.

`Option<T>`

The first concept is that of an `Option`: either there's a value of a particular type (`Some(T)`) or there isn't (`None`). **Always use `Option` for values that can be absent**; never fall back to using sentinel values (-1, `nullptr`, ...) to try to express the same concept in-band.

There is one subtle point to consider, though. If you're dealing with a *collection* of things, you need to decide whether having zero things in the collection is the same as not having a

collection. For most situations, the distinction doesn't arise and you can go ahead and use (say) `Vec<Thing>` : a count of zero things implies an absence of things.

However, there are definitely other rare scenarios where the two cases need to be distinguished with `Option<Vec<Thing>>` —for example, a cryptographic system might need to distinguish between "payload transported separately" and "empty payload provided". (This is related to the debates around the `NULL` marker for columns in SQL.)

Similarly, what's the best choice for a `String` that might be absent? Does `""` or `None` make more sense to indicate the absence of a value? Either way works, but `Option<String>` clearly communicates the possibility that this value may be absent.

`Result<T, E>`

The second common concept arises from error processing: if a function fails, how should that failure be reported? Historically, special sentinel values (e.g., `-errno` return values from Linux system calls) or global variables (`errno` for POSIX systems) were used. More recently, languages that support multiple or tuple return values (such as Go) from functions may have a convention of returning a `(result, error)` pair, assuming the existence of some suitable "zero" value for the `result` when the `error` is non-"zero".

In Rust, there's an `enum` for just this purpose: **always encode the result of an operation that might fail as a `Result<T, E>`** . The `T` type holds the successful result (in the `Ok` variant), and the `E` type holds error details (in the `Err` variant) on failure.

Using the standard type makes the intent of the design clear. It also allows the use of standard transformations (Item 3) and error processing (Item 4), which in turn makes it possible to streamline error processing with the `?` operator as well.

---

[1] The situation gets muddier still if the filesystem is involved, since filenames on popular platforms are somewhere in between arbitrary bytes and UTF-8 sequences: see the `std::ffi::OsString` documentation.

[2] Technically, a *Unicode scalar value* rather than a code point.

[3] The need to consider all possibilities also means that adding a new variant to an existing `enum` in a library is a *breaking change* (Item 21): library clients will need to change their code to cope with the new variant. If an `enum` is really just a C-like list of related numerical values, this behavior can be avoided by marking it as a `non_exhaustive` `enum` ; see Item 21.

Item 2: Use the type system to express common behavior

Item 1 discussed how to express data structures in the type system; this Item moves on to discuss the encoding of *behavior* in Rust's type system.

The mechanisms described in this Item will generally feel familiar, as they all have direct analogs in other languages:

- *Functions*: The universal mechanism for associating a chunk of code with a name and a parameter list.
- *Methods*: Functions that are associated with an instance of a particular data structure. Methods are common in programming languages created after object-orientation arose as a programming paradigm.
- *Function pointers*: Supported by most languages in the C family, including C++ and Go, as a mechanism that allows an extra level of indirection when invoking other code.
- *Closures*: Originally most common in the Lisp family of languages but have been retrofitted to many popular programming languages, including C++ (since C++11) and Java (since Java 8).
- *Traits*: Describe collections of related functionality that all apply to the same underlying item. Traits have rough equivalents in many other languages, including abstract classes in C++ and interfaces in Go and Java.

Of course, all of these mechanisms have Rust-specific details that this Item will cover.

Of the preceding list, traits have the most significance for this book, as they describe so much of the behavior provided by the Rust compiler and standard library. Chapter 2 focuses on Items that give advice on designing and implementing traits, but their pervasiveness means that they crop up frequently in the other Items in this chapter too.

Functions and Methods

As with every other programming language, Rust uses *functions* to organize code into named chunks for reuse, with inputs to the code expressed as parameters. As with every other statically typed language, the types of the parameters and the return value are explicitly specified:

```
/// Return `x` divided by `y`.
fn div(x: f64, y: f64) -> f64 {
    if y == 0.0 {
        // Terminate the function and return a value.
        return f64::NAN;
    }
    // The last expression in the function body is implicitly returned.
    x / y
}

/// Function called just for its side effects, with no return value.
/// Can also write the return value as `-> ()`.
fn show(x: f64) {
    println!("x = {x}");
}
```

If a function is intimately involved with a particular data structure, it is expressed as a *method*. A method acts on an item of that type, identified by `self`, and is included within an `impl DataStructure` block. This encapsulates related data and code together in an object-oriented way that's similar to other languages; however, in Rust, methods can be added to `enum` types as well as to `struct` types, in keeping with the pervasive nature of Rust's `enum` (Item 1):

```
enum Shape {
    Rectangle { width: f64, height: f64 },
    Circle { radius: f64 },
}

impl Shape {
    pub fn area(&self) -> f64 {
        match self {
            Shape::Rectangle { width, height } => width * height,
            Shape::Circle { radius } => std::f64::consts::PI * radius * radius,
        }
    }
}
```

The name of a method creates a label for the behavior it encodes, and the method signature gives type information for its inputs and outputs. The first input for a method will be some variant of `self`, indicating what the method might do to the data structure:

- A `&self` parameter indicates that the contents of the data structure may be read from but will not be modified.
- A `&mut self` parameter indicates that the method might modify the contents of the data structure.
- A `self` parameter indicates that the method consumes the data structure.

Function Pointers

The previous section described how to associate a name (and a parameter list) with some code. However, invoking a function always results in the same code being executed; all that changes from invocation to invocation is the data that the function operates on. That covers a lot of possible scenarios, but what if the *code* needs to vary at runtime?

The simplest behavioral abstraction that allows this is the *function pointer*: a pointer to (just) some code, with a type that reflects the signature of the function:

```
fn sum(x: i32, y: i32) -> i32 {
    x + y
}
// Explicit coercion to `fn` type is required...
let op: fn(i32, i32) -> i32 = sum;
```

The type is checked at compile time, so by the time the program runs, the value is just the size of a pointer. Function pointers have no other data associated with them, so they can be treated as values in various ways:

```
// `fn` types implement `Copy`
let op1 = op;
let op2 = op;
// `fn` types implement `Eq`
assert!(op1 == op2);
// `fn` implements `std::fmt::Pointer`, used by the {:p} format specifier.
println!("op = {:p}", op);
// Example output: "op = 0x101e9aeb0"
```

One technical detail to watch out for: explicit coercion to a `fn` type is needed, because just using the name of a function *doesn't* give you something of `fn` type:

```
let op1 = sum;
let op2 = sum;
// Both op1 and op2 are of a type that cannot be named in user code,
// and this internal type does not implement `Eq`.
assert!(op1 == op2);
```

```
error[E0369]: binary operation `==` cannot be applied to type
              `fn(i32, i32) -> i32 {main::sum}`
   --> src/main.rs:102:17
    |
102 |     assert!(op1 == op2);
    |             --- ^^ --- fn(i32, i32) -> i32 {main::sum}
    |             |
    |             fn(i32, i32) -> i32 {main::sum}
    |
help: use parentheses to call these
    |
102 |     assert!(op1(/* i32 */, /* i32 */) == op2(/* i32 */, /* i32 */));
    |                ++++++++++++++++++++       ++++++++++++++++++++
```

Instead, the compiler error indicates that the type is something like `fn(i32, i32) -> i32 {main::sum}`, a type that's entirely internal to the compiler (i.e., could not be written in user code) and that identifies the specific function as well as its signature. To put it another way, the *type* of `sum` encodes both the function's signature *and* its location for optimization reasons; this type can be automatically *coerced* (Item 5) to a `fn` type.

Closures

The bare function pointers are limiting, because the only inputs available to the invoked function are those that are explicitly passed as parameter values. For example, consider some code that modifies every element of a slice using a function pointer:

```
// In real code, an `Iterator` method would be more appropriate.
pub fn modify_all(data: &mut [u32], mutator: fn(u32) -> u32) {
    for value in data {
        *value = mutator(*value);
    }
}
```

This works for a simple mutation of the slice:

```
fn add2(v: u32) -> u32 {
    v + 2
}
let mut data = vec![1, 2, 3];
modify_all(&mut data, add2);
assert_eq!(data, vec![3, 4, 5]);
```

However, if the modification relies on any additional state, it's not possible to implicitly pass that into the function pointer:

```
let amount_to_add = 3;
fn add_n(v: u32) -> u32 {
    v + amount_to_add
}
let mut data = vec![1, 2, 3];
modify_all(&mut data, add_n);
assert_eq!(data, vec![3, 4, 5]);
```

```
error[E0434]: can't capture dynamic environment in a fn item
   --> src/main.rs:125:13
    |
125 |         v + amount_to_add
    |             ^^^^^^^^^^^^^
    |
    = help: use the `|| { ... }` closure form instead
```

The error message points to the right tool for the job: a *closure*. A closure is a chunk of code that looks like the body of a function definition (a *lambda expression*), except for the following:

- It can be built as part of an expression, and so it need not have a name associated with it.
- The input parameters are given in vertical bars `|param1, param2|` (their associated types can usually be automatically deduced by the compiler).
- It can capture parts of the environment around it:

```
let amount_to_add = 3;
let add_n = |y| {
    // a closure capturing `amount_to_add`
    y + amount_to_add
};
let z = add_n(5);
assert_eq!(z, 8);
```

To (roughly) understand how the capture works, imagine that the compiler creates a one-off, internal type that holds all of the parts of the environment that get mentioned in the lambda expression. When the closure is created, an instance of this ephemeral type is created to hold the relevant values, and when the closure is invoked, that instance is used as additional context:

```
let amount_to_add = 3;
// *Rough* equivalent to a capturing closure.
struct InternalContext<'a> {
    // references to captured variables
    amount_to_add: &'a u32,
}
impl<'a> InternalContext<'a> {
    fn internal_op(&self, y: u32) -> u32 {
        // body of the lambda expression
        y + *self.amount_to_add
    }
}
let add_n = InternalContext {
    amount_to_add: &amount_to_add,
};
let z = add_n.internal_op(5);
assert_eq!(z, 8);
```

The values that are held in this notional context are often references (Item 8) as here, but they can also be mutable references to things in the environment, or values that are moved out of the environment altogether (by using the `move` keyword before the input parameters).

Returning to the `modify_all` example, a closure can't be used where a function pointer is expected:

```
error[E0308]: mismatched types
   --> src/main.rs:199:31
    |
199 |         modify_all(&mut data, |y| y + amount_to_add);
    |         ----------            ^^^^^^^^^^^^^^^^^^^^ expected fn pointer,
    |         |                                          found closure
    |         |
    |         arguments to this function are incorrect
    |
    = note: expected fn pointer `fn(u32) -> u32`
                  found closure `[closure@src/main.rs:199:31: 199:34]`
note: closures can only be coerced to `fn` types if they do not capture any
      variables
   --> src/main.rs:199:39
    |
199 |         modify_all(&mut data, |y| y + amount_to_add);
    |                                      ^^^^^^^^^^^^^ `amount_to_add`
    |                                                    captured here
note: function defined here
   --> src/main.rs:60:12
    |
60  |     pub fn modify_all(data: &mut [u32], mutator: fn(u32) -> u32) {
    |            ^^^^^^^^^^                  -----------------------
```

Instead, the code that receives the closure has to accept an instance of one of the `Fn*` traits:

```
pub fn modify_all<F>(data: &mut [u32], mut mutator: F)
where
    F: FnMut(u32) -> u32,
{
    for value in data {
        *value = mutator(*value);
    }
}
```

Rust has three different `Fn*` traits, which between them express some distinctions around this environment-capturing behavior:

- `FnOnce` : Describes a closure that can be called only *once*. If some part of the environment is `move` d into the closure's context, and the closure's body subsequently moves it *out* of the closure's context, then those moves can happen only once—there's no other copy of the source item to `move` from—and so the closure can be invoked only once.
- `FnMut` : Describes a closure that can be called repeatedly and that can make changes to its environment because it *mutably* borrows from the environment.
- `Fn` : Describes a closure that can be called repeatedly and that only borrows values from the environment immutably.

The compiler *automatically* implements the appropriate subset of these `Fn*` traits for any lambda expression in the code; it's not possible to manually implement any of these traits (unlike C++'s `operator()` overload).[1]

Returning to the preceding rough mental model of closures, which of the traits the compiler auto-implements roughly corresponds to whether the captured environmental context has these elements:

- `FnOnce` : Any moved values
- `FnMut` : Any mutable references to values (`&mut T`)
- `Fn` : Only normal references to values (`&T`)

The latter two traits in this list each have a trait bound of the preceding trait, which makes sense when you consider the things that *use* the closures:

- If something expects to call a closure only once (indicated by receiving a `FnOnce`), it's OK to pass it a closure that's capable of being repeatedly called (`FnMut`).
- If something expects to repeatedly call a closure that might mutate its environment (indicated by receiving a `FnMut`), it's OK to pass it a closure that *doesn't* need to mutate its environment (`Fn`).

The bare function pointer type `fn` also notionally belongs at the end of this list; any (not-`unsafe`) `fn` type automatically implements all of the `Fn*` traits, because it borrows nothing from the environment.

As a result, when writing code that accepts closures, **use the most general `Fn*` trait that works**, to allow the greatest flexibility for callers—for example, accept `FnOnce` for closures that are used only once. The same reasoning also leads to advice to **prefer `Fn*` trait bounds over bare function pointers (`fn`)**.

Traits

The `Fn*` traits are more flexible than bare function pointers, but they can still describe only the behavior of a single function, and even then only in terms of the function's signature.

However, they are themselves examples of another mechanism for describing behavior in Rust's type system, the *trait*. A trait defines a set of related functions that some underlying item makes publicly available; moreover, the functions are typically (but don't have to be) *methods*, taking some variant of `self` as their first argument.

Each function in a trait also has a *name*, providing a label that allows the compiler to disambiguate functions with the same signature, and more importantly, that allows programmers to deduce the intent of the function.

A Rust trait is roughly analogous to an "interface" in Go and Java, or to an "abstract class" (all virtual methods, no data members) in C++. Implementations of the trait must provide all the functions (but note that the trait definition can include a default implementation; Item 13) and can also have associated data that those implementations make use of. This means that code and data gets encapsulated together in a common abstraction, in a *somewhat* object-oriented (OO) manner.

Code that accepts a `struct` and calls functions on it is constrained to only ever work with that specific type. If there are multiple types that implement common behavior, then it is more flexible to define a trait that encapsulates that common behavior, and have the code make use of the trait's functions rather than functions involving a specific `struct`.

This leads to the same kind of advice that turns up for other OO-influenced languages:[2] **prefer accepting trait types over concrete types** if future flexibility is anticipated.

Sometimes, there is some behavior that you want to distinguish in the type system, but it cannot be expressed as some specific function signature in a trait definition. For example, consider a `Sort` trait for sorting collections; an implementation might be *stable* (elements that compare the same will appear in the same order before and after the sort), but there's no way to express this in the `sort` method arguments.

In this case, it's still worth using the type system to track this requirement, using a *marker trait*:

```
pub trait Sort {
    /// Rearrange contents into sorted order.
    fn sort(&mut self);
}

/// Marker trait to indicate that a [`Sort`] sorts stably.
pub trait StableSort: Sort {}
```

A marker trait has no functions, but an implementation still has to declare that it is implementing the trait—which acts as a promise from the implementer: "I solemnly swear that my implementation sorts stably". Code that relies on a stable sort can then specify the `StableSort` trait bound, relying on the honor system to preserve its invariants. **Use marker traits to distinguish behaviors that cannot be expressed in the trait function signatures**.

Once behavior has been encapsulated into Rust's type system as a trait, it can be used in two ways:

- As a *trait bound*, which constrains what types are acceptable for a generic data type or function at compile time
- As a *trait object*, which constrains what types can be stored or passed to a function at runtime

The following sections describe these two possibilities, and Item 12 gives more detail about the trade-offs between them.

Trait bounds

A *trait bound* indicates that generic code that is parameterized by some type `T` can be used only when that type `T` implements some specific trait. The presence of the trait bound means that the implementation of the generic can use the functions from that trait, secure in the knowledge that the compiler will ensure that any `T` that compiles does indeed have those functions. This check happens at compile time, when the generic is *monomorphized*—converted from the generic code that deals with an arbitrary type `T` into specific code that deals with one particular `SomeType` (what C++ would call *template instantiation*).

This restriction on the target type `T` is *explicit*, encoded in the trait bounds: the trait can be implemented only by types that satisfy the trait bounds. This contrasts with the equivalent situation in C++, where the constraints on the type `T` used in a `template<typename T>` are *implicit*:[3] C++ template code still compiles only if all of the referenced functions are available at compile time, but the checks are purely based on function name and signature. (This "duck typing" can lead to confusion; a C++ template that uses `t.pop()` might compile for a `T` type parameter of either `Stack` or `Balloon` —which is unlikely to be desired behavior.)

The need for explicit trait bounds also means that a large fraction of generics use trait bounds. To see why this is, turn the observation around and consider what can be done with

a `struct Thing<T>` where there are *no* trait bounds on `T`. Without a trait bound, the `Thing` can perform only operations that apply to *any* type `T`—basically just moving or dropping the value. This in turn allows for generic containers, collections, and smart pointers, but not much else. Anything that *uses* the type `T` is going to need a trait bound:

```
pub fn dump_sorted<T>(mut collection: T)
where
    T: Sort + IntoIterator,
    T::Item: std::fmt::Debug,
{
    // Next line requires `T: Sort` trait bound.
    collection.sort();
    // Next line requires `T: IntoIterator` trait bound.
    for item in collection {
        // Next line requires `T::Item : Debug` trait bound
        println!("{:?}", item);
    }
}
```

So the advice here is to **use trait bounds to express requirements on the types used in generics**, but it's easy advice to follow—the compiler will force you to comply with it regardless.

Trait objects

A *trait object* is the other way to make use of the encapsulation defined by a trait, but here, different possible implementations of the trait are chosen at runtime rather than compile time. This *dynamic dispatch* is analogous to using virtual functions in C++, and under the covers, Rust has "vtable" objects that are *roughly* analogous to those in C++.

This dynamic aspect of trait objects also means that they always have to be handled indirectly, via a reference (e.g., `&dyn Trait`) or a pointer (e.g., `Box<dyn Trait>`) of some kind. The reason is that the size of the object implementing the trait isn't known at compile time—it could be a giant `struct` or a tiny `enum`—so there's no way to allocate the right amount of space for a bare trait object.

Not knowing the size of the concrete object also means that traits used as trait objects cannot have functions that return the `Self` type or arguments (other than the *receiver*—the object on which the method is being invoked) that use `Self`. The reason is that the compiled-in-advance code that uses the trait object would have no idea how big that `Self` might be.

A trait that has a generic function `fn some_fn<T>(t:T)` allows for the possibility of an infinite number of implemented functions, for all of the different types `T` that might exist. This is fine for a trait used as a trait bound, because the infinite set of *possibly* invoked generic functions becomes a finite set of *actually* invoked generic functions at compile time.

The same is not true for a trait object: the code available at compile time has to cope with all possible `T`s that might arrive at runtime.

These two restrictions—no use of `Self` and no generic functions—are combined in the concept of *object safety*. Only object-safe traits can be used as trait objects.

---

[1] At least not in stable Rust at the time of writing. The `unboxed_closures` and `fn_traits` experimental features may change this in the future.

[2] For example, Joshua Bloch's *Effective Java* (3rd edition, Addison-Wesley) includes Item 64: Refer to objects by their interfaces.

[3] The addition of *concepts* in C++20 allows explicit specification of constraints on template types, but the checks are still performed only when the template is instantiated, not when it is declared.

Item 3: Prefer `Option` and `Result` transforms over explicit `match` expressions

Item 1 expounded the virtues of `enum` and showed how `match` expressions force the programmer to take all possibilities into account. Item 1 also introduced the two ubiquitous `enum`s that the Rust standard library provides:

- `Option<T>`: To express that a value (of type `T`) may or may not be present
- `Result<T, E>`: For when an operation to return a value (of type `T`) may not succeed and may instead return an error (of type `E`)

This Item explores situations where you should try to avoid explicit `match` expressions for these particular `enum`s, preferring instead to use various transformation methods that the standard library provides for these types. Using these transformation methods (which are typically themselves implemented as `match` expressions under the covers) leads to code that is more compact and idiomatic and has clearer intent.

The first situation where a `match` is unnecessary is when only the value is relevant and the absence of value (and any associated error) can just be ignored:

```
struct S {
    field: Option<i32>,
}

let s = S { field: Some(42) };
match &s.field {
    Some(i) => println!("field is {i}"),
    None => {}
}
```

For this situation, an `if let` expression is one line shorter and, more importantly, clearer:

```
if let Some(i) = &s.field {
    println!("field is {i}");
}
```

However, most of the time the programmer needs to provide the corresponding `else` arm: the absence of a value (`Option::None`), possibly with an associated error (`Result::Err(e)`), is something that the programmer needs to deal with. Designing software to cope with failure paths is hard, and most of that is essential complexity that no amount of syntactic support can help with—specifically, deciding what should happen if an operation fails.

In some situations, the right decision is to perform an ostrich maneuver—put our heads in the sand and explicitly not cope with failure. You can't *completely* ignore the error arm, because Rust requires that the code deal with both variants of the `Error` `enum`, but you can

choose to treat a failure as fatal. Performing a `panic!` on failure means that the program terminates, but the rest of the code can then be written with the assumption of success. Doing this with an explicit `match` would be needlessly verbose:

```
let result = std::fs::File::open("/etc/passwd");
let f = match result {
    Ok(f) => f,
    Err(_e) => panic!("Failed to open /etc/passwd!"),
};
// Assume `f` is a valid `std::fs::File` from here onward.
```

Both `Option` and `Result` provide a pair of methods that extract their inner value and `panic!` if it's absent: `unwrap` and `expect`. The latter allows the error message on failure to be personalized, but in either case, the resulting code is shorter and simpler—error handling is delegated to the `.unwrap()` suffix (but is still present):

```
let f = std::fs::File::open("/etc/passwd").unwrap();
```

Be clear, though: these helper functions still `panic!`, so choosing to use them is the same as choosing to `panic!` (Item 18).

However, in many situations, the right decision for error handling is to defer the decision to somebody else. This is particularly true when writing a library, where the code may be used in all sorts of different environments that can't be foreseen by the library author. To make that somebody else's job easier, **prefer `Result` to `Option`** for expressing errors, even though this may involve conversions between different error types (Item 4).

Of course, this opens up the question, What counts as an error? In this example, failing to open a file is definitely an error, and the details of that error (no such file? permission denied?) can help the user decide what to do next. On the other hand, failing to retrieve the `first()` element of a slice because that slice is empty isn't really an error, and so it is expressed as an `Option` return type in the standard library. Choosing between the two possibilities requires judgment, but lean toward `Result` if an error might communicate anything useful.

`Result` also has a `#[must_use]` attribute to nudge library users in the right direction—if the code using the returned `Result` ignores it, the compiler will generate a warning:

```
warning: unused `Result` that must be used
  --> src/main.rs:63:5
   |
63 |     f.set_len(0); // Truncate the file
   |     ^^^^^^^^^^^^
   |
   = note: this `Result` may be an `Err` variant, which should be handled
   = note: `#[warn(unused_must_use)]` on by default
help: use `let _ = ...` to ignore the resulting value
   |
63 |     let _ = f.set_len(0); // Truncate the file
   |     +++++++
```

Explicitly using a `match` allows an error to propagate, but at the cost of some visible boilerplate (reminiscent of Go):

```
pub fn find_user(username: &str) -> Result<UserId, std::io::Error> {
    let f = match std::fs::File::open("/etc/passwd") {
        Ok(f) => f,
        Err(e) => return Err(From::from(e)),
    };
    // ...
}
```

The key ingredient for reducing boilerplate is Rust's question mark operator, `?`. This piece of syntactic sugar takes care of matching the `Err` arm, transforming the error type if necessary, and building the `return Err(...)` expression, all in a single character:

```
pub fn find_user(username: &str) -> Result<UserId, std::io::Error> {
    let f = std::fs::File::open("/etc/passwd")?;
    // ...
}
```

Newcomers to Rust sometimes find this disconcerting: the question mark can be hard to spot on first glance, leading to disquiet as to how the code can possibly work. However, even with a single character, the type system is still at work, ensuring that all of the possibilities expressed in the relevant types (Item 1) are covered—leaving the programmer to focus on the mainline code path without distractions.

What's more, there's generally no cost to these apparent method invocations: they are all generic functions marked as `#[inline]`, so the generated code will typically compile to machine code that's identical to the manual version.

These two factors taken together mean that you should **prefer `Option` and `Result` transforms over explicit `match` expressions**.

In the previous example, the error types lined up: both the inner and outer methods expressed errors as `std::io::Error`. That's often not the case: one function may

accumulate errors from a variety of different sublibraries, each of which uses different error types.

Error mapping in general is discussed in Item 4, but for now, just be aware that a manual mapping:

```
pub fn find_user(username: &str) -> Result<UserId, String> {
    let f = match std::fs::File::open("/etc/passwd") {
        Ok(f) => f,
        Err(e) => {
            return Err(format!("Failed to open password file: {:?}", e))
        }
    };
    // ...
}
```

can be more succinctly and idiomatically expressed with the following `.map_err()` transformation:

```
pub fn find_user(username: &str) -> Result<UserId, String> {
    let f = std::fs::File::open("/etc/passwd")
        .map_err(|e| format!("Failed to open password file: {:?}", e))?;
    // ...
}
```

Better still, even this may not be necessary—if the outer error type can be created from the inner error type via an implementation of the `From` standard trait (Item 10), then the compiler will automatically perform the conversion without the need for a call to `.map_err()`.

These kinds of transformations generalize more widely. The question mark operator is a big hammer; use transformation methods on `Option` and `Result` types to maneuver them into a position where they can be a nail.

The standard library provides a wide variety of these transformation methods to make this possible. Figure 1-1 shows some of the most common methods (rounded white rectangles) that transform between the relevant types (gray rectangles).[1] In line with Item 18, methods that can `panic!` are marked with an asterisk.

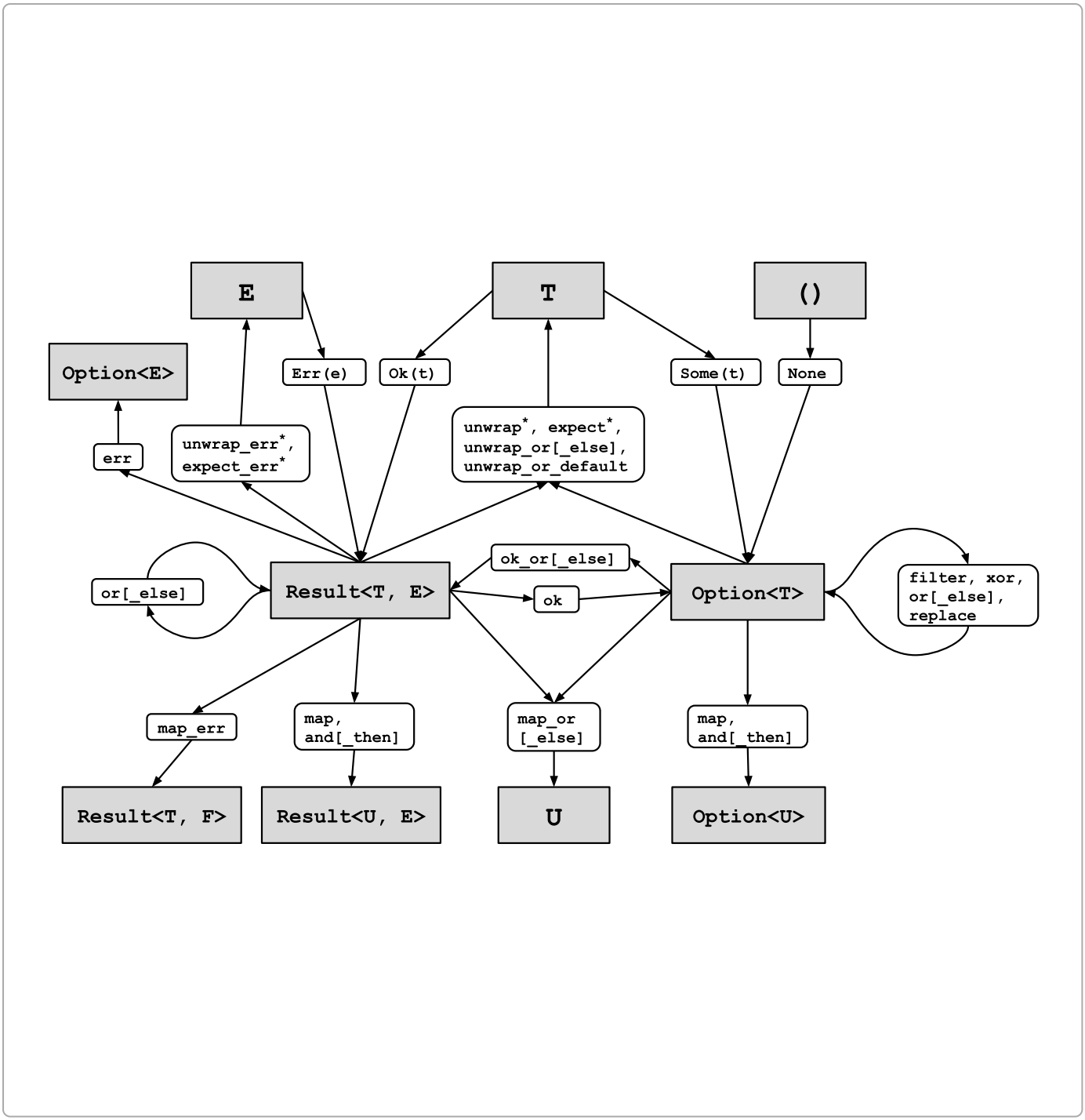


Figure 1-1. `Option` and `Result` transformations

One common situation the diagram doesn't cover deals with references. For example, consider a structure that optionally holds some data:

```
struct InputData {
    payload: Option<Vec<u8>>,
}
```

A method on this `struct` that tries to pass the payload to an encryption function with signature `(&[u8]) -> Vec<u8>` fails if there's a naive attempt to take a reference:

```
impl InputData {
    pub fn encrypted(&self) -> Vec<u8> {
        encrypt(&self.payload.unwrap_or(vec![]))
    }
}
```

```
error[E0507]: cannot move out of `self.payload` which is behind a shared
              reference
  --> src/main.rs:15:18
   |
15 |     encrypt(&self.payload.unwrap_or(vec![]))
   |              ^^^^^^^^^^^^ move occurs because `self.payload` has type
   |                           `Option<Vec<u8>>`, which does not implement the
   |                           `Copy` trait
```

The right tool for this is the `as_ref()` method on `Option`.[2] This method converts a reference-to-an-`Option` into an `Option`-of-a-reference:

```
pub fn encrypted(&self) -> Vec<u8> {
    encrypt(self.payload.as_ref().unwrap_or(&vec![]))
}
```

Things to Remember

- Get used to the transformations of `Option` and `Result`, and prefer `Result` to `Option`. Use `.as_ref()` as needed when transformations involve references.
- Use these transformations in preference to explicit `match` operations on `Option` and `Result`.
- In particular, use these transformations to convert result types into a form where the `?` operator applies.

---

[1] The online version of this diagram is clickable; each box links to the relevant documentation.

[2] Note that this method is separate from the `AsRef` trait, even though the method name is the same.

Item 4: Prefer idiomatic `Error` types

Item 3 described how to use the transformations that the standard library provides for the `Option` and `Result` types to allow concise, idiomatic handling of result types using the `?` operator. It stopped short of discussing how best to handle the variety of different error types `E` that arise as the second type argument of a `Result<T, E>`; that's the subject of this Item.

This is relevant only when there *are* a variety of different error types in play. If all of the different errors that a function encounters are already of the same type, it can just return that type. When there are errors of different types, there's a decision to make about whether the suberror type information should be preserved.

The `Error` Trait

It's always good to understand what the standard traits (Item 10) involve, and the relevant trait here is `std::error::Error`. The `E` type parameter for a `Result` doesn't *have* to be a type that implements `Error`, but it's a common convention that allows wrappers to express appropriate trait bounds—so **prefer to implement `Error` for your error types**.

The first thing to notice is that the only hard requirement for `Error` types is the trait bounds: any type that implements `Error` also has to implement the following traits:

- The `Display` trait, meaning that it can be `format!` ed with `{}`
- The `Debug` trait, meaning that it can be `format!` ed with `{:?}`

In other words, it should be possible to display `Error` types to both the user and the programmer.

The only method in the trait is `source()`,[1] which allows an `Error` type to expose an inner, nested error. This method is optional—it comes with a default implementation (Item 13) returning `None`, indicating that inner error information isn't available.

One final thing to note: if you're writing code for a `no_std` environment (Item 33), it may not be possible to implement `Error`—the `Error` trait is currently implemented in `std`, not `core`, and so is not available.[2]

Minimal Errors

If nested error information isn't needed, then an implementation of the `Error` type need not be much more than a `String` —one rare occasion where a "stringly typed" variable might be appropriate. It does need to be a *little* more than a `String` though; while it's possible to use `String` as the `E` type parameter:

```
pub fn find_user(username: &str) -> Result<UserId, String> {
    let f = std::fs::File::open("/etc/passwd")
        .map_err(|e| format!("Failed to open password file: {:?}", e))?;
    // ...
}
```

a `String` doesn't implement `Error`, which we'd prefer so that other areas of code can deal with `Error`s. It's not possible to `impl Error` for `String`, because neither the trait nor the type belong to us (the so-called *orphan rule*):

```
impl std::error::Error for String {}
```

```
error[E0117]: only traits defined in the current crate can be implemented for
              types defined outside of the crate
  --> src/main.rs:18:5
   |
18 |     impl std::error::Error for String {}
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^------
   |     |                          |
   |     |                          `String` is not defined in the current
crate
   |     impl doesn't use only types from inside the current crate
   |
   = note: define and implement a trait or new type instead
```

A *type alias* doesn't help either, because it doesn't create a new type and so doesn't change the error message:

```
pub type MyError = String;

impl std::error::Error for MyError {}
```

```
error[E0117]: only traits defined in the current crate can be implemented for
              types defined outside of the crate
  --> src/main.rs:41:5
   |
41 |     impl std::error::Error for MyError {}
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^-------
   |     |                          |
   |     |                          `String` is not defined in the current
crate
   |     impl doesn't use only types from inside the current crate
   |
   = note: define and implement a trait or new type instead
```

As usual, the compiler error message gives a hint to solving the problem. Defining a tuple struct that wraps the `String` type (the "newtype pattern", Item 6) allows the `Error` trait to be implemented, provided that `Debug` and `Display` are implemented too:

```
#[derive(Debug)]
pub struct MyError(String);

impl std::fmt::Display for MyError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}", self.0)
    }
}

impl std::error::Error for MyError {}

pub fn find_user(username: &str) -> Result<UserId, MyError> {
    let f = std::fs::File::open("/etc/passwd").map_err(|e| {
        MyError(format!("Failed to open password file: {:?}", e))
    })?;
    // ...
}
```

For convenience, it may make sense to implement the `From<String>` trait to allow string values to be easily converted into `MyError` instances (Item 5):

```
impl From<String> for MyError {
    fn from(msg: String) -> Self {
        Self(msg)
    }
}
```

When it encounters the question mark operator (`?`), the compiler will automatically apply any relevant `From` trait implementations that are needed to reach the destination error return type. This allows further minimization:

```
pub fn find_user(username: &str) -> Result<UserId, MyError> {
    let f = std::fs::File::open("/etc/passwd")
        .map_err(|e| format!("Failed to open password file: {:?}", e))?;
    // ...
}
```

The error path here covers the following steps:

- `File::open` returns an error of type `std::io::Error`.
- `format!` converts this to a `String`, using the `Debug` implementation of `std::io::Error`.
- `?` makes the compiler look for and use a `From` implementation that can take it from `String` to `MyError`.

Nested Errors

The alternative scenario is where the content of nested errors is important enough that it should be preserved and made available to the caller.

Consider a library function that attempts to return the first line of a file as a string, as long as the line is not too long. A moment's thought reveals (at least) three distinct types of failure that could occur:

- The file might not exist or might be inaccessible for reading.
- The file might contain data that isn't valid UTF-8 and so can't be converted into a `String`.
- The file might have a first line that is too long.

In line with Item 1, you can use the type system to express and encompass all of these possibilities as an `enum`:

```
#[derive(Debug)]
pub enum MyError {
    Io(std::io::Error),
    Utf8(std::string::FromUtf8Error),
    General(String),
}
```

This `enum` definition includes a `derive(Debug)`, but to satisfy the `Error` trait, a `Display` implementation is also needed:

```
impl std::fmt::Display for MyError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            MyError::Io(e) => write!(f, "IO error: {}", e),
            MyError::Utf8(e) => write!(f, "UTF-8 error: {}", e),
            MyError::General(s) => write!(f, "General error: {}", s),
        }
    }
}
```

It also makes sense to override the default `source()` implementation for easy access to nested errors:

```
use std::error::Error;

impl Error for MyError {
    fn source(&self) -> Option<&(dyn Error + 'static)> {
        match self {
            MyError::Io(e) => Some(e),
            MyError::Utf8(e) => Some(e),
            MyError::General(_) => None,
        }
    }
}
```

The use of an `enum` allows the error handling to be concise while still preserving all of the type information across different classes of error:

```
use std::io::BufRead; // for `.read_until()`

/// Maximum supported line length.
const MAX_LEN: usize = 1024;

/// Return the first line of the given file.
pub fn first_line(filename: &str) -> Result<String, MyError> {
    let file = std::fs::File::open(filename).map_err(MyError::Io)?;
    let mut reader = std::io::BufReader::new(file);

    // (A real implementation could just use `reader.read_line()`)
    let mut buf = vec![];
    let len = reader.read_until(b'\n', &mut buf).map_err(MyError::Io)?;
    let result = String::from_utf8(buf).map_err(MyError::Utf8)?;
    if result.len() > MAX_LEN {
        return Err(MyError::General(format!("Line too long: {}", len)));
    }
    Ok(result)
}
```

It's also a good idea to implement the `From` trait for all of the suberror types (Item 5):

```
impl From<std::io::Error> for MyError {
    fn from(e: std::io::Error) -> Self {
        Self::Io(e)
    }
}
impl From<std::string::FromUtf8Error> for MyError {
    fn from(e: std::string::FromUtf8Error) -> Self {
        Self::Utf8(e)
    }
}
```

This prevents library users from suffering under the orphan rules themselves: they aren't allowed to implement `From` on `MyError`, because both the trait and the struct are external to them.

Better still, implementing `From` allows for even more concision, because the question mark operator will automatically perform any necessary `From` conversions, removing the need for `.map_err()`:

```
use std::io::BufRead; // for `.read_until()`

/// Maximum supported line length.
pub const MAX_LEN: usize = 1024;

/// Return the first line of the given file.
pub fn first_line(filename: &str) -> Result<String, MyError> {
    let file = std::fs::File::open(filename)?; // `From<std::io::Error>`
    let mut reader = std::io::BufReader::new(file);
    let mut buf = vec![];
    let len = reader.read_until(b'\n', &mut buf)?; // `From<std::io::Error>`
    let result = String::from_utf8(buf)?; // `From<string::FromUtf8Error>`
    if result.len() > MAX_LEN {
        return Err(MyError::General(format!("Line too long: {}", len)));
    }
    Ok(result)
}
```

Writing a complete error type can involve a fair amount of boilerplate, which makes it a good candidate for automation via a *`derive` macro* (Item 28). However, there's no need to write such a macro yourself: **consider using the `thiserror` crate** from David Tolnay, which provides a high-quality, widely used implementation of just such a macro. The code generated by `thiserror` is also careful to avoid making any `thiserror` types visible in the generated API, which in turn means that the concerns associated with Item 24 don't apply.

Trait Objects

The first approach to nested errors threw away all of the suberror detail, just preserving some string output (`format!("{:?}", err)`). The second approach preserved the full type

information for all possible suberrors but required a full enumeration of all possible types of suberror.

This raises the question, Is there a middle ground between these two approaches, preserving suberror information without needing to manually include every possible error type?

Encoding the suberror information as a *trait object* avoids the need for an `enum` variant for every possibility but erases the details of the specific underlying error types. The receiver of such an object would have access to the methods of the `Error` trait and its trait bounds—`source()`, `Display::fmt()`, and `Debug::fmt()`, in turn—but wouldn't know the original static type of the suberror:

```
#[derive(Debug)]
pub enum WrappedError {
    Wrapped(Box<dyn Error>),
    General(String),
}

impl std::fmt::Display for WrappedError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Self::Wrapped(e) => write!(f, "Inner error: {}", e),
            Self::General(s) => write!(f, "{}", s),
        }
    }
}
```

It turns out that this *is* possible, but it's surprisingly subtle. Part of the difficulty comes from the object safety constraints on trait objects (Item 12), but Rust's *coherence rules* also come into play, which (roughly) say that there can be at most one implementation of a trait for a type.

A putative `WrappedError` type would naively be expected to implement both of the following:

- The `Error` trait, because it is an error itself.
- The `From<Error>` trait, to allow suberrors to be easily wrapped.

That means that a `WrappedError` can be created `from` an inner `WrappedError`, as `WrappedError` implements `Error`, and that clashes with the blanket reflexive implementation of `From`:

```
impl Error for WrappedError {}

impl<E: 'static + Error> From<E> for WrappedError {
    fn from(e: E) -> Self {
        Self::Wrapped(Box::new(e))
    }
}
```

```
error[E0119]: conflicting implementations of trait `From<WrappedError>` for
              type `WrappedError`
   --> src/main.rs:279:5
    |
279 |     impl<E: 'static + Error> From<E> for WrappedError {
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    |
    = note: conflicting implementation in crate `core`:
            - impl<T> From<T> for T;
```

David Tolnay's `anyhow` is a crate that has already solved these problems (by adding an extra level of indirection via `Box`) and that adds other helpful features (such as stack traces) besides. As a result, it is rapidly becoming the standard recommendation for error handling—a recommendation seconded here: **consider using the `anyhow` crate for error handling in applications**.

Libraries Versus Applications

The final advice from the previous section included the qualification "...for error handling *in applications*". That's because there's often a distinction between code that's written for reuse in a library and code that forms a top-level application.[3]

Code that's written for a library can't predict the environment in which the code is used, so it's preferable to emit concrete, detailed error information and leave the caller to figure out how to use that information. This leans toward the `enum`-style nested errors described previously (and also avoids a dependency on `anyhow` in the public API of the library, see Item 24).

However, application code typically needs to concentrate more on how to present errors to the user. It also potentially has to cope with all of the different error types emitted by all of the libraries that are present in its dependency graph (Item 25). As such, a more dynamic error type (such as `anyhow::Error`) makes error handling simpler and more consistent across the application.

Things to Remember

- The standard `Error` trait requires little of you, so prefer to implement it for your error types.
- When dealing with heterogeneous underlying error types, decide whether it's necessary to preserve those types.
  - If not, consider using `anyhow` to wrap suberrors in application code.

    - If so, encode them in an `enum` and provide conversions. Consider using `thiserror` to help with this.
- Consider using the `anyhow` crate for convenient idiomatic error handling in application code.
- It's your decision, but whatever you decide, encode it in the type system (Item 1).

---

[1] Or at least the only nondeprecated, stable method.

[2] At the time of writing, `Error` has been moved to `core` but is not yet available in stable Rust.

[3] This section is inspired by Nick Groenen's "Rust: Structuring and Handling Errors in 2020" article.

Item 5: Understand type conversions

Rust type conversions fall into three categories:

- *Manual*: User-defined type conversions provided by implementing the `From` and `Into` traits
- *Semi-automatic*: Explicit *casts* between values using the `as` keyword
- *Automatic*: Implicit *coercion* into a new type

The majority of this Item focuses on the first of these, manual conversions of types, because the latter two mostly don't apply to conversions of user-defined types. There are a couple of exceptions to this, so sections at the end of the Item discuss casting and coercion—including how they can apply to a user-defined type.

Note that in contrast to many older languages, Rust does not perform automatic conversion between numeric types. This even applies to "safe" transformations of integral types:

```
let x: u32 = 2;
let y: u64 = x;
```

```
error[E0308]: mismatched types
  --> src/main.rs:70:18
   |
70 |     let y: u64 = x;
   |            ---   ^ expected `u64`, found `u32`
   |            |
   |            expected due to this
   |
help: you can convert a `u32` to a `u64`
   |
70 |     let y: u64 = x.into();
   |                   +++++++
```

User-Defined Type Conversions

As with other features of the language (Item 10), the ability to perform conversions between values of different user-defined types is encapsulated as a standard trait—or rather, as a set of related generic traits.

The four relevant traits that express the ability to convert values of a type are as follows:

- `From<T>` : Items of this type can be built from items of type `T` , and the conversion always succeeds.

- `TryFrom<T>` : Items of this type can be built from items of type `T` , but the conversion might not succeed.
- `Into<T>` : Items of this type can be converted into items of type `T` , and the conversion always succeeds.
- `TryInto<T>` : Items of this type can be converted into items of type `T` , but the conversion might not succeed.

Given the discussion in Item 1 about expressing things in the type system, it's no surprise to discover that the difference with the `Try...` variants is that the sole trait method returns a `Result` rather than a guaranteed new item. The `Try...` trait definitions also require an associated type that gives the type of the error `E` emitted for failure situations.

The first piece of advice is therefore to **implement (just) the `Try...` trait if it's possible for a conversion to fail**, in line with Item 4. The alternative is to ignore the possibility of error (e.g., with `.unwrap()`), but that needs to be a deliberate choice, and in most cases it's best to leave that choice to the caller.

The type conversion traits have an obvious symmetry: if a type `T` can be transformed `into` a type `U` (via `Into<U>`), isn't that the same as it being possible to create an item of type `U` by transforming `from` an item of type `T` (via `From<T>`)?

This is indeed the case, and it leads to the second piece of advice: **implement the `From` trait for conversions**. The Rust standard library had to pick just one of the two possibilities, in order to prevent the system from spiraling around in dizzy circles,[1] and it came down on the side of automatically providing `Into` from a `From` implementation.

If you're consuming one of these two traits, as a trait bound on a new generic of your own, then the advice is reversed: **use the `Into` trait for trait bounds**. That way, the bound will be satisfied both by things that directly implement `Into` *and* by things that only directly implement `From` .

This automatic conversion is highlighted by the documentation for `From` and `Into` , but it's worth reading the relevant part of the standard library code too, which is a *blanket trait implementation*:

```
impl<T, U> Into<U> for T
where
    U: From<T>,
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

Translating a trait specification into words can help with understanding more complex trait bounds. In this case, it's fairly simple: "I can implement `Into<U>` for a type `T` whenever `U` already implements `From<T>` ".

The standard library also includes various implementations of these conversion traits for standard library types. As you'd expect, there are `From` implementations for integral conversions where the destination type includes all possible values of the source type (`From<u32> for u64`), and `TryFrom` implementations when the source might not fit in the destination (`TryFrom<u64> for u32`).

There are also various other blanket trait implementations in addition to the `Into` version previously shown; these are mostly for *smart pointer* types, allowing the smart pointer to be automatically constructed from an instance of the type that it holds. This means that generic methods that accept smart pointer parameters can also be called with plain old items; more on this to come and in Item 8.

The `TryFrom` trait also has a blanket implementation for any type that already implements the `Into` trait in the opposite direction—which automatically includes (as shown previously) any type that implements `From` in the same direction. In other words, if you can infallibly convert a `T` into a `U`, you can also fallibly obtain a `U` from a `T`; as this conversion will always succeed, the associated error type is the helpfully named `Infallible`.[2]

There's also one very specific generic implementation of `From` that sticks out, the *reflexive implementation*:

```
impl<T> From<T> for T {
    fn from(t: T) -> T {
        t
    }
}
```

Translated into words, this just says that “given a `T`, I can get a `T`.” That's such an obvious "well, duh" that it's worth stopping to understand why this is useful.

Consider a simple newtype `struct` (Item 6) and a function that operates on it (ignoring that this function would be better expressed as a method):

```
/// Integer value from an IANA-controlled range.
#[derive(Clone, Copy, Debug)]
pub struct IanaAllocated(pub u64);

/// Indicate whether value is reserved.
pub fn is_iana_reserved(s: IanaAllocated) -> bool {
    s.0 == 0 || s.0 == 65535
}
```

This function can be invoked with instances of the `struct`:

```
let s = IanaAllocated(1);
println!("{:?} reserved? {}", s, is_iana_reserved(s));
// output: "IanaAllocated(1) reserved? false"
```

but even if `From<u64>` is implemented for the newtype wrapper:

```
impl From<u64> for IanaAllocated {
    fn from(v: u64) -> Self {
        Self(v)
    }
}
```

the function can't be directly invoked for `u64` values:

```
if is_iana_reserved(42) {
    // ...
}
```

```
error[E0308]: mismatched types
  --> src/main.rs:77:25
   |
77 |     if is_iana_reserved(42) {
   |        ---------------- ^^ expected `IanaAllocated`, found integer
   |        |
   |        arguments to this function are incorrect
   |
note: function defined here
  --> src/main.rs:7:8
   |
7  | pub fn is_iana_reserved(s: IanaAllocated) -> bool {
   |        ^^^^^^^^^^^^^^^^ ----------------
help: try wrapping the expression in `IanaAllocated`
   |
77 |     if is_iana_reserved(IanaAllocated(42)) {
   |                         ++++++++++++++  +
```

However, a generic version of the function that accepts (and explicitly converts) anything satisfying `Into<IanaAllocated>`:

```
pub fn is_iana_reserved<T>(s: T) -> bool
where
    T: Into<IanaAllocated>,
{
    let s = s.into();
    s.0 == 0 || s.0 == 65535
}
```

allows this use:

```
if is_iana_reserved(42) {
    // ...
}
```

With this trait bound in place, the reflexive trait implementation of `From<T>` makes more sense: it means that the generic function copes with items that are already `IanaAllocated`

instances, no conversion needed.

This pattern also explains why (and how) Rust code sometimes *appears* to be doing implicit casts between types: the combination of `From<T>` implementations and `Into<T>` trait bounds leads to code that appears to magically convert at the call site (but is still doing safe, explicit, conversions under the covers). This pattern becomes even more powerful when combined with reference types and their related conversion traits; more in Item 8.

Casts

Rust includes the `as` keyword to perform explicit *casts* between some pairs of types.

The pairs of types that can be converted in this way constitute a fairly limited set, and the only user-defined types it includes are "C-like" `enum` s (those that have just an associated integer value). General integral conversions are included, though, giving an alternative to `into()`:

```
let x: u32 = 9;
let y = x as u64;
let z: u64 = x.into();
```

The `as` version also allows lossy conversions:[3]

```
let x: u32 = 9;
let y = x as u16;
```

which would be rejected by the `from` / `into` versions:

```
error[E0277]: the trait bound `u16: From<u32>` is not satisfied
   --> src/main.rs:136:20
    |
136 |     let y: u16 = x.into();
    |                    ^^^^ the trait `From<u32>` is not implemented for
`u16`
    |
    = help: the following other types implement trait `From<T>`:
              <u16 as From<NonZeroU16>>
              <u16 as From<bool>>
              <u16 as From<u8>>
    = note: required for `u32` to implement `Into<u16>`
```

For consistency and safety, you should **prefer `from` / `into` conversions over `as` casts**, unless you understand and need the precise casting semantics (e.g., for C interoperability). This advice can be reinforced by Clippy (Item 29), which includes several lints about `as` conversions; however, these lints are disabled by default.

Coercion

The explicit `as` casts described in the previous section are a superset of the implicit *coercions* that the compiler will silently perform: any coercion can be forced with an explicit `as`, but the converse is not true. In particular, the integral conversions performed in the previous section are not coercions and so will always require `as`.

Most coercions involve silent conversions of pointer and reference types in ways that are sensible and convenient for the programmer, such as converting the following:

- A mutable reference to an immutable reference (so you can use a `&mut T` as the argument to a function that takes a `&T`)
- A reference to a raw pointer (this isn't `unsafe`—the unsafety happens at the point where you're foolish enough to *dereference* a raw pointer)
- A closure that happens to not capture any variables into a bare function pointer (Item 2)
- An array to a slice
- A concrete item to a trait object, for a trait that the concrete item implements
- An item lifetime to a "shorter" one (Item 14)[4]

There are only two coercions whose behavior can be affected by user-defined types. The first happens when a user-defined type implements the `Deref` or the `DerefMut` trait. These traits indicate that the user-defined type is acting as a *smart pointer* of some sort (Item 8), and in this case the compiler will coerce a reference to the smart pointer item into being a reference to an item of the type that the smart pointer contains (indicated by its `Target`).

The second coercion of a user-defined type happens when a concrete item is converted to a *trait object*. This operation builds a fat pointer to the item; this pointer is fat because it includes both a pointer to the item's location in memory and a pointer to the *vtable* for the concrete type's implementation of the trait—see Item 8.

---

[1] More properly known as the *trait coherence rules*.

[2] For now—this is likely to be replaced with the `!` "never" type in a future version of Rust.

[3] Allowing lossy conversions in Rust was probably a mistake, and there have been discussions around trying to remove this behavior.

[4] Rust refers to these conversions as "subtyping", but it's quite different from the definition of "subtyping" used in object-oriented languages.

Item 6: Embrace the newtype pattern

Item 1 described *tuple structs*, where the fields of a `struct` have no names and are instead referred to by number (`self.0`). This Item focuses on tuple structs that have a single entry of some existing type, thus creating a new type that can hold exactly the same range of values as the enclosed type. This pattern is sufficiently pervasive in Rust that it deserves its own Item and has its own name: the *newtype pattern*.

The simplest use of the newtype pattern is to indicate additional semantics for a type, over and above its normal behavior. To illustrate this, imagine a project that's going to send a satellite to Mars.[1] It's a big project, so different groups have built different parts of the project. One group has handled the code for the rocket engines:

```
/// Fire the thrusters. Returns generated impulse in pound-force seconds.
pub fn thruster_impulse(direction: Direction) -> f64 {
    // ...
    return 42.0;
}
```

while a different group handles the inertial guidance system:

```
/// Update trajectory model for impulse, provided in Newton seconds.
pub fn update_trajectory(force: f64) {
    // ...
}
```

Eventually these different parts need to be joined together:

```
let thruster_force: f64 = thruster_impulse(direction);
let new_direction = update_trajectory(thruster_force);
```

Ruh-roh.[2]

Rust includes a *type alias* feature, which allows the different groups to make their intentions clearer:

```
/// Units for force.
pub type PoundForceSeconds = f64;

/// Fire the thrusters. Returns generated impulse.
pub fn thruster_impulse(direction: Direction) -> PoundForceSeconds {
    // ...
    return 42.0;
}
```

```
/// Units for force.
pub type NewtonSeconds = f64;

/// Update trajectory model for impulse.
pub fn update_trajectory(force: NewtonSeconds) {
    // ...
}
```

However, the type aliases are effectively just documentation; they're a stronger hint than the doc comments of the previous version, but nothing stops a `PoundForceSeconds` value being used where a `NewtonSeconds` value is expected:

```
let thruster_force: PoundForceSeconds = thruster_impulse(direction);
let new_direction = update_trajectory(thruster_force);
```

Ruh-roh once more.

This is the point where the newtype pattern helps:

```
/// Units for force.
pub struct PoundForceSeconds(pub f64);

/// Fire the thrusters. Returns generated impulse.
pub fn thruster_impulse(direction: Direction) -> PoundForceSeconds {
    // ...
    return PoundForceSeconds(42.0);
}


/// Units for force.
pub struct NewtonSeconds(pub f64);

/// Update trajectory model for impulse.
pub fn update_trajectory(force: NewtonSeconds) {
    // ...
}
```

As the name implies, a newtype is a new type, and as such the compiler objects when there's a mismatch of types—here attempting to pass a `PoundForceSeconds` value to something that expects a `NewtonSeconds` value:

```
let thruster_force: PoundForceSeconds = thruster_impulse(direction);
let new_direction = update_trajectory(thruster_force);
```

```
error[E0308]: mismatched types
  --> src/main.rs:76:43
   |
76 |     let new_direction = update_trajectory(thruster_force);
   |                         ----------------- ^^^^^^^^^^^^^^ expected
   |                         |        `NewtonSeconds`, found
`PoundForceSeconds`
   |                         |
   |                         arguments to this function are incorrect
   |
note: function defined here
  --> src/main.rs:66:8
   |
66 | pub fn update_trajectory(force: NewtonSeconds) {
   |        ^^^^^^^^^^^^^^^^ --------------------
help: call `Into::into` on this expression to convert `PoundForceSeconds` into
      `NewtonSeconds`
   |
76 |     let new_direction = update_trajectory(thruster_force.into());
   |                                                          +++++++
```

As described in Item 5, adding an implementation of the standard `From` trait:

```
impl From<PoundForceSeconds> for NewtonSeconds {
    fn from(val: PoundForceSeconds) -> NewtonSeconds {
        NewtonSeconds(4.448222 * val.0)
    }
}
```

allows the necessary unit—and type—conversion to be performed with `.into()`:

```
let thruster_force: PoundForceSeconds = thruster_impulse(direction);
let new_direction = update_trajectory(thruster_force.into());
```

The same pattern of using a newtype to mark additional "unit" semantics for a type can also help to make purely Boolean arguments less ambiguous. Revisiting the example from Item 1, using newtypes makes the meaning of arguments clear:

```
struct DoubleSided(pub bool);

struct ColorOutput(pub bool);

fn print_page(sides: DoubleSided, color: ColorOutput) {
    // ...
}


print_page(DoubleSided(true), ColorOutput(false));
```

If size efficiency or binary compatibility is a concern, then the `#[repr(transparent)]` attribute ensures that a newtype has the same representation in memory as the inner type.

That's the simple use of newtype, and it's a specific example of Item 1—encoding semantics into the type system, so that the compiler takes care of policing those semantics.

Bypassing the Orphan Rule for Traits

The other common, but more subtle, scenario that requires the newtype pattern revolves around Rust's orphan rule. Roughly speaking, this says that a crate can implement a trait for a type only if one of the following conditions holds:

- The crate has defined the trait
- The crate has defined the type

Attempting to implement a foreign trait for a foreign type:

```
use std::fmt;

impl fmt::Display for rand::rngs::StdRng {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> Result<(), fmt::Error> {
        write!(f, "<StdRng instance>")
    }
}
```

leads to a compiler error (which in turn points the way back to newtypes):

```
error[E0117]: only traits defined in the current crate can be implemented for
              types defined outside of the crate
   --> src/main.rs:146:1
    |
146 | impl fmt::Display for rand::rngs::StdRng {
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    | |                       |
    | |                       `StdRng` is not defined in the current crate
    | impl doesn't use only types from inside the current crate
    |
    = note: define and implement a trait or new type instead
```

The reason for this restriction is due to the risk of ambiguity: if two different crates in the dependency graph (Item 25) were *both* to (say) `impl std::fmt::Display for rand::rngs::StdRng`, then the compiler/linker has no way to choose between them.

This can frequently lead to frustration: for example, if you're trying to serialize data that includes a type from another crate, the orphan rule prevents you from writing `impl serde::Serialize for somecrate::SomeType`.[3]

But the newtype pattern means that you're defining a *new* type, which is part of the current crate, and so the second part of the orphan trait rule applies. Implementing a foreign trait is now possible:

```
struct MyRng(rand::rngs::StdRng);

impl fmt::Display for MyRng {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> Result<(), fmt::Error> {
        write!(f, "<MyRng instance>")
    }
}
```

Newtype Limitations

The newtype pattern solves these two classes of problems—preventing unit conversions and bypassing the orphan rule—but it does come with some awkwardness: every operation that involves the newtype needs to forward to the inner type.

On a trivial level, that means that the code has to use `thing.0` throughout, rather than just `thing`, but that's easy, and the compiler will tell you where it's needed.

The more significant awkwardness is that any trait implementations on the inner type are lost: because the newtype is a new type, the existing inner implementation doesn't apply.

For derivable traits, this just means that the newtype declaration ends up with lots of `derive`s:

```
#[derive(Debug, Copy, Clone, Eq, PartialEq, Ord, PartialOrd)]
pub struct NewType(InnerType);
```

However, for more sophisticated traits, some forwarding boilerplate is needed to recover the inner type's implementation, for example:

```
use std::fmt;
impl fmt::Display for NewType {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> Result<(), fmt::Error> {
        self.0.fmt(f)
    }
}
```

---

[1] Specifically, the Mars Climate Orbiter.

[2] See "Mars Climate Orbiter" on Wikipedia for more on the cause of failure.

[3] This is a sufficiently common problem for `serde` that it includes a mechanism to help.

Item 7: Use builders for complex types

This Item describes the builder pattern, where complex data structures have an associated *builder* type that makes it easier for users to create instances of the data structure.

Rust insists that all fields in a `struct` must be filled in when a new instance of that `struct` is created. This keeps the code safe by ensuring that there are never any uninitialized values but does lead to more verbose boilerplate code than is ideal.

For example, any optional fields have to be explicitly marked as absent with `None`:

```
/// Phone number in E164 format.
#[derive(Debug, Clone)]
pub struct PhoneNumberE164(pub String);

#[derive(Debug, Default)]
pub struct Details {
    pub given_name: String,
    pub preferred_name: Option<String>,
    pub middle_name: Option<String>,
    pub family_name: String,
    pub mobile_phone: Option<PhoneNumberE164>,
}

// ...

let dizzy = Details {
    given_name: "Dizzy".to_owned(),
    preferred_name: None,
    middle_name: None,
    family_name: "Mixer".to_owned(),
    mobile_phone: None,
};
```

This boilerplate code is also brittle, in the sense that a future change that adds a new field to the `struct` requires an update to every place that builds the structure.

The boilerplate can be significantly reduced by implementing and using the `Default` trait, as described in Item 10:

```
let dizzy = Details {
    given_name: "Dizzy".to_owned(),
    family_name: "Mixer".to_owned(),
    ..Default::default()
};
```

Using `Default` also helps reduce the changes needed when a new field is added, provided that the new field is itself of a type that implements `Default`.

That's a more general concern: the automatically derived implementation of `Default` works only if all of the field types implement the `Default` trait. If there's a field that doesn't play along, the `derive` step doesn't work:

```
#[derive(Debug, Default)]
pub struct Details {
    pub given_name: String,
    pub preferred_name: Option<String>,
    pub middle_name: Option<String>,
    pub family_name: String,
    pub mobile_phone: Option<PhoneNumberE164>,
    pub date_of_birth: time::Date,
    pub last_seen: Option<time::OffsetDateTime>,
}
```

```
error[E0277]: the trait bound `Date: Default` is not satisfied
  --> src/main.rs:48:9
   |
41 |     #[derive(Debug, Default)]
   |                     ------- in this derive macro expansion
...
48 |         pub date_of_birth: time::Date,
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `Default` is not
   |                                       implemented for `Date`
   |
   = note: this error originates in the derive macro `Default`
```

The code can't implement `Default` for `chrono::Utc` because of the orphan rule; but even if it could, it wouldn't be helpful—using a default value for date of birth is going to be wrong almost all of the time.

The absence of `Default` means that all of the fields have to be filled out manually:

```
let bob = Details {
    given_name: "Robert".to_owned(),
    preferred_name: Some("Bob".to_owned()),
    middle_name: Some("the".to_owned()),
    family_name: "Builder".to_owned(),
    mobile_phone: None,
    date_of_birth: time::Date::from_calendar_date(
        1998,
        time::Month::November,
        28,
    )
    .unwrap(),
    last_seen: None,
};
```

These ergonomics can be improved if you **implement the builder pattern for complex data structures**.

The simplest variant of the builder pattern is a separate `struct` that holds the information needed to construct the item. For simplicity, the example will hold an instance of the item itself:

```
pub struct DetailsBuilder(Details);

impl DetailsBuilder {
    /// Start building a new [`Details`] object.
    pub fn new(
        given_name: &str,
        family_name: &str,
        date_of_birth: time::Date,
    ) -> Self {
        DetailsBuilder(Details {
            given_name: given_name.to_owned(),
            preferred_name: None,
            middle_name: None,
            family_name: family_name.to_owned(),
            mobile_phone: None,
            date_of_birth,
            last_seen: None,
        })
    }
}
```

The builder type can then be equipped with helper methods that fill out the nascent item's fields. Each such method consumes `self` but emits a new `Self`, allowing different construction methods to be chained:

```
/// Set the preferred name.
pub fn preferred_name(mut self, preferred_name: &str) -> Self {
    self.0.preferred_name = Some(preferred_name.to_owned());
    self
}

/// Set the middle name.
pub fn middle_name(mut self, middle_name: &str) -> Self {
    self.0.middle_name = Some(middle_name.to_owned());
    self
}
```

These helper methods can be more helpful than just simple setters:

```
/// Update the `last_seen` field to the current date/time.
pub fn just_seen(mut self) -> Self {
    self.0.last_seen = Some(time::OffsetDateTime::now_utc());
    self
}
```

The final method to be invoked for the builder consumes the builder and emits the built item:

```
/// Consume the builder object and return a fully built [`Details`]
/// object.
pub fn build(self) -> Details {
    self.0
}
```

Overall, this allows clients of the builder to have a more ergonomic building experience:

```
let also_bob = DetailsBuilder::new(
    "Robert",
    "Builder",
    time::Date::from_calendar_date(1998, time::Month::November, 28)
        .unwrap(),
)
.middle_name("the")
.preferred_name("Bob")
.just_seen()
.build();
```

The all-consuming nature of this style of builder leads to a couple of wrinkles. The first is that separating out stages of the build process can't be done on its own:

```
let builder = DetailsBuilder::new(
    "Robert",
    "Builder",
    time::Date::from_calendar_date(1998, time::Month::November, 28)
        .unwrap(),
);
if informal {
    builder.preferred_name("Bob");
}
let bob = builder.build();
```

```
error[E0382]: use of moved value: `builder`
   --> src/main.rs:256:15
    |
247 |     let builder = DetailsBuilder::new(
    |         ------- move occurs because `builder` has type `DetailsBuilder`,
    |                 which does not implement the `Copy` trait
...
254 |         builder.preferred_name("Bob");
    |                 ---------------------- `builder` moved due to this method
    |                                        call
255 |     }
256 |     let bob = builder.build();
    |               ^^^^^^^ value used here after move
    |
note: `DetailsBuilder::preferred_name` takes ownership of the receiver `self`,
      which moves `builder`
   --> src/main.rs:60:35
    |
27  |     pub fn preferred_name(mut self, preferred_name: &str) -> Self {
    |                           ^^^^
```

This can be worked around by assigning the consumed builder back to the same variable:

```
let mut builder = DetailsBuilder::new(
    "Robert",
    "Builder",
    time::Date::from_calendar_date(1998, time::Month::November, 28)
        .unwrap(),
);
if informal {
    builder = builder.preferred_name("Bob");
}
let bob = builder.build();
```

The other downside to the all-consuming nature of this builder is that only one item can be built; trying to create multiple instances by repeatedly calling `build()` on the same builder falls foul of the compiler, as you'd expect:

```
let smithy = DetailsBuilder::new(
    "Agent",
    "Smith",
    time::Date::from_calendar_date(1999, time::Month::June, 11).unwrap(),
);
let clones = vec![smithy.build(), smithy.build(), smithy.build()];
```

```
error[E0382]: use of moved value: `smithy`
   --> src/main.rs:159:39
    |
154 |   let smithy = DetailsBuilder::new(
    |       ------ move occurs because `smithy` has type
`base::DetailsBuilder`,
    |              which does not implement the `Copy` trait
...
159 |   let clones = vec![smithy.build(), smithy.build(), smithy.build()];
    |                            -------  ^^^^^^ value used here after move
    |                            |
    |                            `smithy` moved due to this method call
```

An alternative approach is for the builder's methods to take a `&mut self` and emit a `&mut Self`:

```
/// Update the `last_seen` field to the current date/time.
pub fn just_seen(&mut self) -> &mut Self {
    self.0.last_seen = Some(time::OffsetDateTime::now_utc());
    self
}
```

This removes the need for self-assignment in separate build stages:

```
let mut builder = DetailsBuilder::new(
    "Robert",
    "Builder",
    time::Date::from_calendar_date(1998, time::Month::November, 28)
        .unwrap(),
);
if informal {
    builder.preferred_name("Bob"); // no `builder = ...`
}
let bob = builder.build();
```

However, this version makes it impossible to chain the construction of the builder together with invocation of its setter methods:

```
let builder = DetailsBuilder::new(
    "Robert",
    "Builder",
    time::Date::from_calendar_date(1998, time::Month::November, 28)
        .unwrap(),
)
.middle_name("the")
.just_seen();
let bob = builder.build();
```

```
error[E0716]: temporary value dropped while borrowed
   --> src/main.rs:265:19
    |
265 |       let builder = DetailsBuilder::new(
    |  ___________________^
266 | |         "Robert",
267 | |         "Builder",
268 | |         time::Date::from_calendar_date(1998, time::Month::November, 28)
269 | |             .unwrap(),
270 | |     )
    | |_____^ creates a temporary value which is freed while still in use
271 |       .middle_name("the")
272 |       .just_seen();
    |                   - temporary value is freed at the end of this statement
273 |       let bob = builder.build();
    |                 --------------- borrow later used here
    |
    = note: consider using a `let` binding to create a longer lived value
```

As indicated by the compiler error, you can work around this by `let`ting the builder item have a name:

```
let mut builder = DetailsBuilder::new(
    "Robert",
    "Builder",
    time::Date::from_calendar_date(1998, time::Month::November, 28)
        .unwrap(),
);
builder.middle_name("the").just_seen();
if informal {
    builder.preferred_name("Bob");
}
let bob = builder.build();
```

This mutating builder variant also allows for building multiple items. The signature of the `build()` method has to *not* consume `self` and so must be as follows:

```
/// Construct a fully built [`Details`] object.
pub fn build(&self) -> Details {
    // ...
}
```

The implementation of this repeatable `build()` method then has to construct a fresh item on each invocation. If the underlying item implements `Clone`, this is easy—the builder can hold a template and `clone()` it for each build. If the underlying item *doesn't* implement `Clone`, then the builder needs to have enough state to be able to manually construct an instance of the underlying item on each call to `build()`.

With any style of builder pattern, the boilerplate code is now confined to one place—the builder—rather than being needed at every place that uses the underlying type.

The boilerplate that remains can potentially be reduced still further by use of a macro (Item 28), but if you go down this road, you should also check whether there's an existing crate (such as the `derive_builder` crate, in particular) that provides what's needed—assuming that you're happy to take a dependency on it (Item 25).

Item 8: Familiarize yourself with reference and pointer types

For programming in general, a *reference* is a way to indirectly access some data structure, separately from whatever variable owns that data structure. In practice, this is usually implemented as a *pointer*: a number whose value is the address in memory of the data structure.

A modern CPU will typically police a few constraints on pointers—the memory address should be in a valid range of memory (whether virtual or physical) and may need to be *aligned* (e.g., a 4-byte integer value might be accessible only if its address is a multiple of 4).

However, higher-level programming languages usually encode more information about pointers in their type systems. In C-derived languages, including Rust, pointers have a type that indicates what kind of data structure is expected to be present at the pointed-to memory address. This allows the code to interpret the contents of memory at that address and in the memory following that address.

This basic level of pointer information—putative memory location and expected data structure layout—is represented in Rust as a *raw pointer*. However, safe Rust code does not use raw pointers, because Rust provides richer reference and pointer types that provide additional safety guarantees and constraints. These reference and pointer types are the subject of this Item; raw pointers are relegated to Item 16 (which discusses `unsafe` code).

Rust References

The most ubiquitous pointer-like type in Rust is the *reference*, with a type that is written as `&T` for some type `T`. Although this is a pointer value under the covers, the compiler ensures that various rules around its use are observed: it must always point to a valid, correctly aligned instance of the relevant type `T`, whose lifetime (Item 14) extends beyond its use, and it must satisfy the borrow checking rules (Item 15). These additional constraints are always implied by the term *reference* in Rust, and so the bare term *pointer* is generally rare.

The constraint that a Rust reference must point to a valid, correctly aligned item is shared by C++'s reference types. However, C++ has no concept of lifetimes and so allows footguns with dangling references:[1]

```
// C++
const int& dangle() {
  int x = 32; // on the stack, overwritten later
  return x; // return reference to stack variable!
}
```

Rust's borrowing and lifetime checks mean that the equivalent code doesn't even compile:

```
fn dangle() -> &'static i64 {
    let x: i64 = 32; // on the stack
    &x
}
```

```
error[E0515]: cannot return reference to local variable `x`
   --> src/main.rs:477:5
    |
477 |     &x
    |     ^^ returns a reference to data owned by the current function
```

A Rust reference `&T` allows read-only access to the underlying item (roughly equivalent to C++'s `const T&`). A mutable reference that also allows the underlying item to be modified is written as `&mut T` and is also subject to the borrow checking rules discussed in Item 15. This naming pattern reflects a slightly different mindset between Rust and C++:

- In Rust, the default variant is read-only, and writable types are marked specially (with `mut`).
- In C++, the default variant is writable, and read-only types are marked specially (with `const`).

The compiler converts Rust code that uses references into machine code that uses simple pointers, which are eight bytes in size on a 64-bit platform (which this Item assumes throughout). For example, a pair of local variables together with references to them:

```
pub struct Point {
    pub x: u32,
    pub y: u32,
}

let pt = Point { x: 1, y: 2 };
let x = 0u64;
let ref_x = &x;
let ref_pt = &pt;
```

might end up laid out on the stack as shown in Figure 1-2.

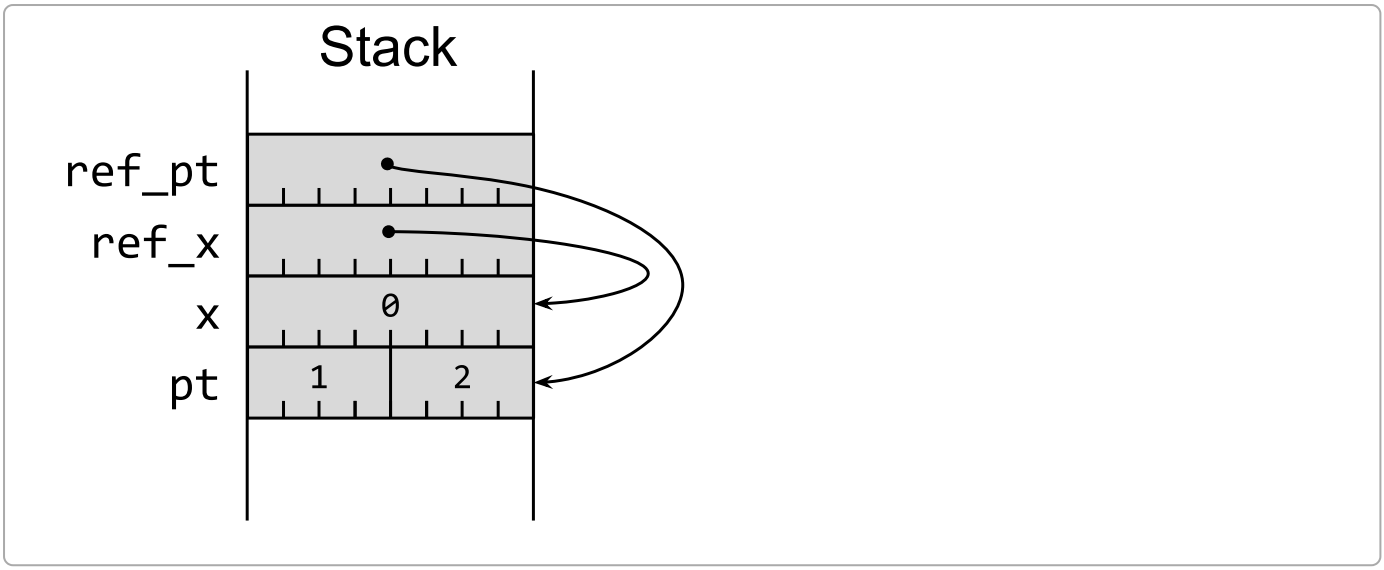


Figure 1-2. Stack layout with pointers to local variables

A Rust reference can refer to items that are located either on the stack or on the heap. Rust allocates items on the stack by default, but the `Box<T>` pointer type (roughly equivalent to C++'s `std::unique_ptr<T>`) forces allocation to occur on the heap, which in turn means that the allocated item can outlive the scope of the current block. Under the covers, `Box<T>` is also a simple eight-byte pointer value:

```
let box_pt = Box::new(Point { x: 10, y: 20 });
```

This is depicted in Figure 1-3.

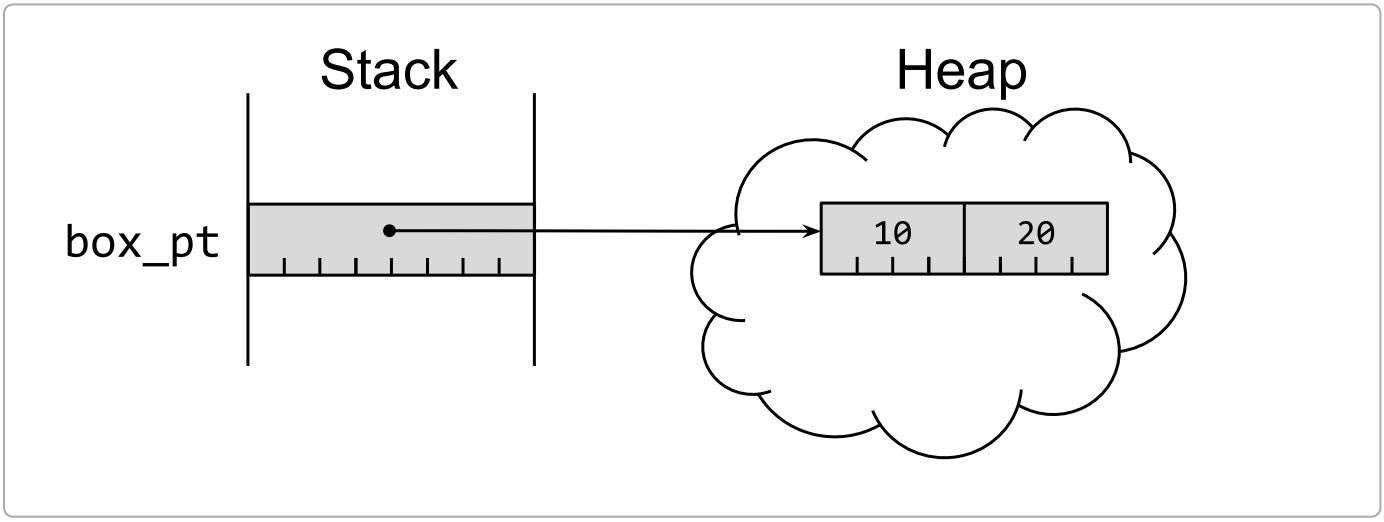


Figure 1-3. Stack `Box` pointer to `struct` on heap

Pointer Traits

A method that expects a reference argument like `&Point` can also be fed a `&Box<Point>`:

```
fn show(pt: &Point) {
    println!("({}, {})", pt.x, pt.y);
}
show(ref_pt);
show(&box_pt);
```

```
(1, 2)
(10, 20)
```

This is possible because `Box<T>` implements the `Deref` trait, with `Target = T`. An implementation of this trait for some type means that the trait's `deref()` method can be used to create a reference to the `Target` type. There's also an equivalent `DerefMut` trait, which emits a *mutable* reference to the `Target` type.

The `Deref` / `DerefMut` traits are somewhat special, because the Rust compiler has specific behavior when dealing with types that implement them. When the compiler encounters a dereferencing expression (e.g., `*x`), it looks for and uses an implementation of one of these traits, depending on whether the dereference requires mutable access or not. This *Deref coercion* allows various smart pointer types to behave like normal references and is one of the few mechanisms that allow implicit type conversion in Rust (as described in Item 5).

As a technical aside, it's worth understanding why the `Deref` traits can't be generic (`Deref<Target>`) for the destination type. If they were, then it would be possible for some type `ConfusedPtr` to implement both `Deref<TypeA>` and `Deref<TypeB>`, and that would leave the compiler unable to deduce a single unique type for an expression like `*x`. So instead, the destination type is encoded as the associated type named `Target`.

This technical aside provides a contrast to two other standard pointer traits, the `AsRef` and `AsMut` traits. These traits don't induce special behavior in the compiler but allow conversions to a reference or mutable reference via an explicit call to their trait functions (`as_ref()` and `as_mut()`, respectively). The destination type for these conversions *is* encoded as a type parameter (e.g., `AsRef<Point>`), which means that a single container type can support multiple destinations.

For example, the standard `String` type implements the `Deref` trait with `Target = str`, meaning that an expression like `&my_string` can be coerced to type `&str`. But it also implements the following:

- `AsRef<[u8]>`, allowing conversion to a byte slice `&[u8]`
- `AsRef<OsStr>`, allowing conversion to an OS string
- `AsRef<Path>`, allowing conversion to a filesystem path
- `AsRef<str>`, allowing conversion to a string slice `&str` (as with `Deref`)

Fat Pointer Types

Rust has two built-in *fat pointer* types: slices and trait objects. These are types that act as pointers but hold additional information about the thing they are pointing to.

Slices

The first fat pointer type is the *slice*: a reference to a subset of some contiguous collection of values. It's built from a (non-owning) simple pointer, together with a length field, making it twice the size of a simple pointer (16 bytes on a 64-bit platform). The type of a slice is written as `&[T]`—a reference to `[T]`, which is the notional type for a contiguous collection of values of type `T`.

The notional type `[T]` can't be instantiated, but there are two common containers that embody it. The first is the *array*: a contiguous collection of values having a size that is known at compile time—an array with five values will always have five values. A slice can therefore refer to a subset of an array (as depicted in Figure 1-4):

```
let array: [u64; 5] = [0, 1, 2, 3, 4];
let slice = &array[1..3];
```

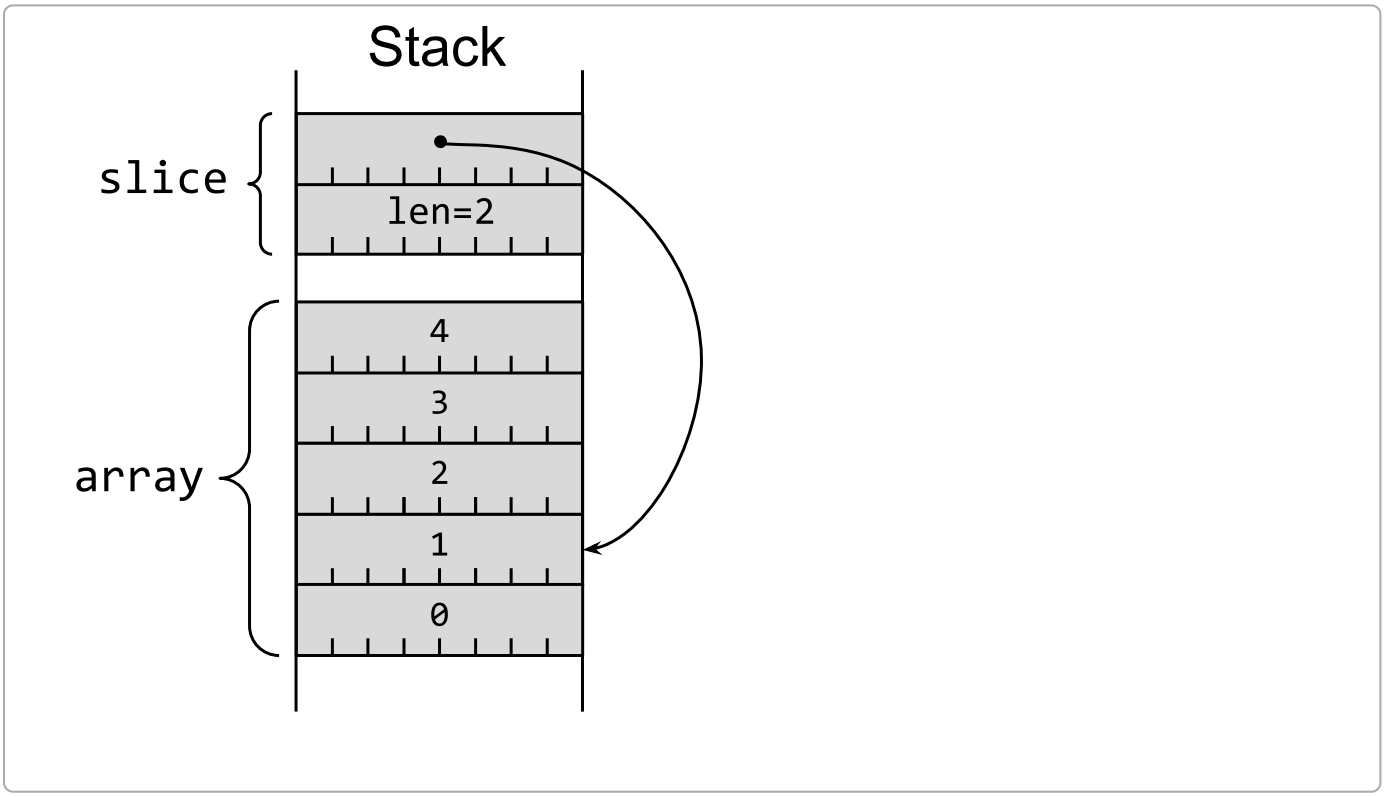


Figure 1-4. Stack slice pointing into a stack array

The other common container for contiguous values is a `Vec<T>`. This holds a contiguous collection of values like an array, but unlike an array, the number of values in the `Vec` can grow (e.g., with `push(value)`) or shrink (e.g., with `pop()`).

The contents of the `Vec` are kept on the heap (which allows for this variation in size) but are always contiguous, and so a slice can refer to a subset of a vector, as shown in Figure 1-5:

```
let mut vector = Vec::<u64>::with_capacity(8);
for i in 0..5 {
    vector.push(i);
}
let vslice = &vector[1..3];
```

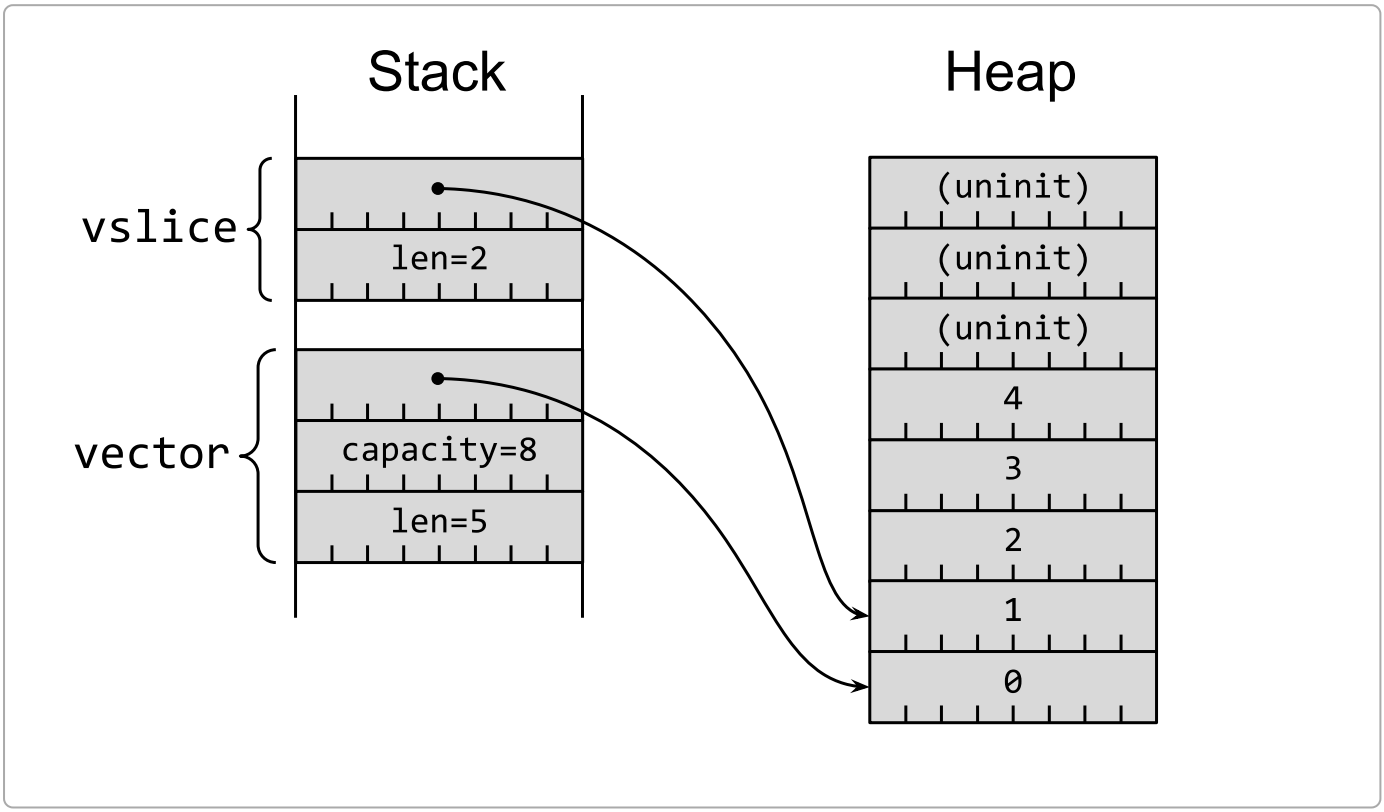


Figure 1-5. Stack slice pointing into `Vec` contents on the heap

There's quite a lot going on under the covers for the expression `&vector[1..3]`, so it's worth breaking it down into its components:

- The `1..3` part is a range expression; the compiler converts this into an instance of the `Range<usize>` type, which holds an inclusive lower bound and an exclusive upper bound.
- The `Range` type implements the `SliceIndex<T>` trait, which describes indexing operations on slices of an arbitrary type `T` (so the `Output` type is `[T]`).
- The `vector[ ]` part is an indexing expression; the compiler converts this into an invocation of the `Index` trait's `index` method on `vector`, together with a dereference (i.e., `*vector.index( )`).[2]
- `vector[1..3]` therefore invokes `Vec<T>`'s implementation of `Index<I>`, which requires `I` to be an instance of `SliceIndex<[u64]>`. This works because `Range<usize>` implements `SliceIndex<[T]>` for any `T`, including `u64`.
- `&vector[1..3]` undoes the dereference, resulting in a final expression type of `&[u64]`.

Trait objects

The second built-in fat pointer type is a *trait object*: a reference to some item that implements a particular trait. It's built from a simple pointer to the item, together with an internal pointer to the type's *vtable*, giving a size of 16 bytes (on a 64-bit platform). The vtable for a type's implementation of a trait holds function pointers for each of the method implementations, allowing dynamic dispatch at runtime (Item 12).[3]

So a simple trait:

```
trait Calculate {
    fn add(&self, l: u64, r: u64) -> u64;
    fn mul(&self, l: u64, r: u64) -> u64;
}
```

with a `struct` that implements it:

```
struct Modulo(pub u64);

impl Calculate for Modulo {
    fn add(&self, l: u64, r: u64) -> u64 {
        (l + r) % self.0
    }
    fn mul(&self, l: u64, r: u64) -> u64 {
        (l * r) % self.0
    }
}

let mod3 = Modulo(3);
```

can be converted to a trait object of type `&dyn Trait`. The `dyn` keyword highlights the fact that dynamic dispatch is involved:

```
// Need an explicit type to force dynamic dispatch.
let tobj: &dyn Calculate = &mod3;
let result = tobj.add(2, 2);
assert_eq!(result, 1);
```

The equivalent memory layout is shown in Figure 1-6.

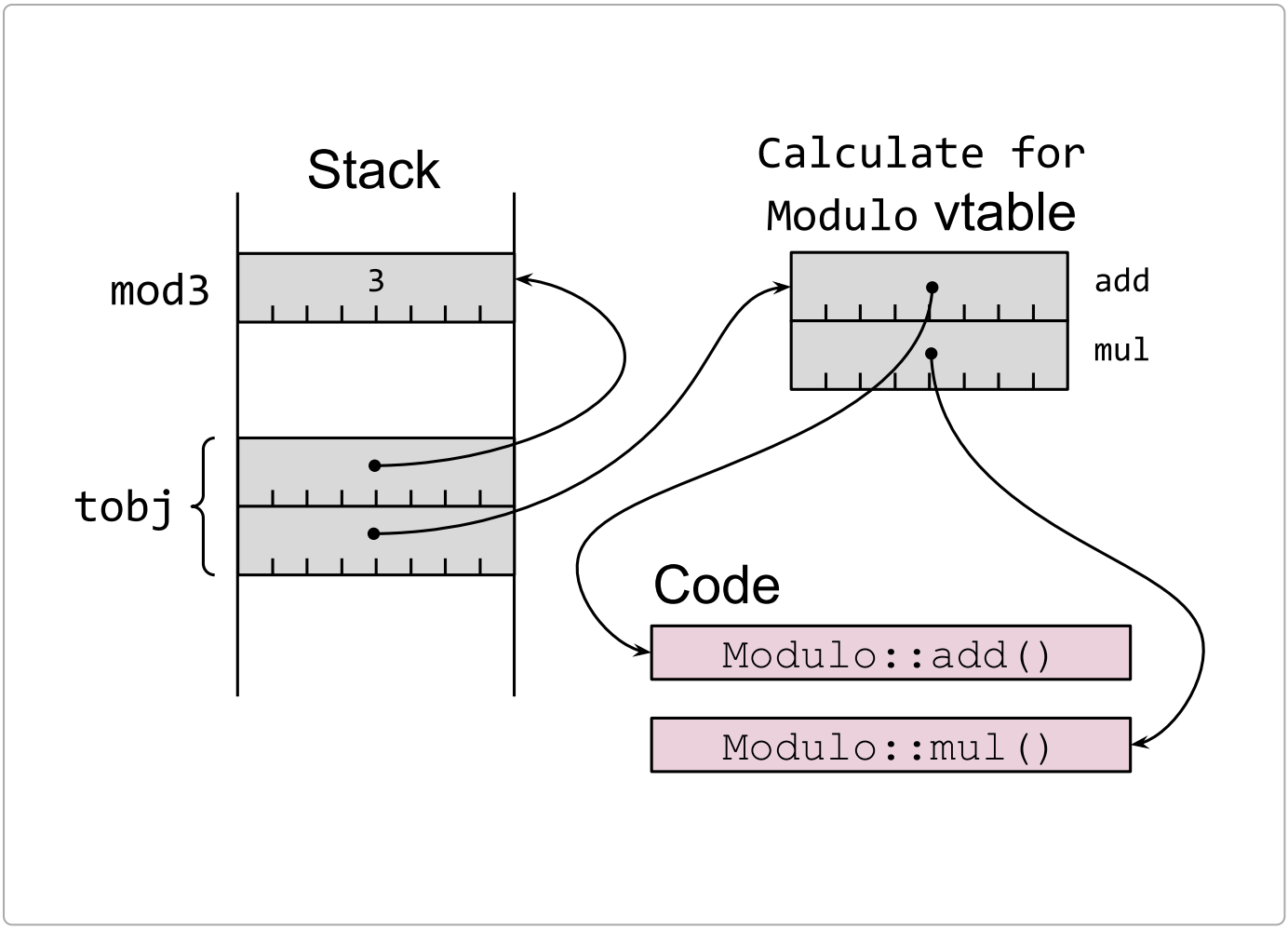


Figure 1-6. Trait object with pointers to concrete item and vtable

Code that holds a trait object can invoke the methods of the trait via the function pointers in the vtable, passing in the item pointer as the `&self` parameter; see Item 12 for more information and advice.

More Pointer Traits

A previous section described two pairs of traits (`Deref` / `DerefMut` , `AsRef` / `AsMut`) that are used when dealing with types that can be easily converted into references. There are a few more standard traits that can also come into play when working with pointer-like types, whether from the standard library or user defined.

The simplest of these is the `Pointer` trait, which formats a pointer value for output. This can be helpful for low-level debugging, and the compiler will reach for this trait automatically when it encounters the `{:p}` format specifier.

More intriguing are the `Borrow` and `BorrowMut` traits, which each have a single method (`borrow` and `borrow_mut` , respectively). This method has the same signature as the equivalent `AsRef` / `AsMut` trait methods.

The key difference in intents between these traits is visible via the blanket implementations that the standard library provides. Given an arbitrary Rust reference `&T`, there is a blanket implementation of both `AsRef` and `Borrow`; likewise, for a mutable reference `&mut T`, there's a blanket implementation of both `AsMut` and `BorrowMut`.

However, `Borrow` also has a blanket implementation for (non-reference) types: `impl<T> Borrow<T> for T`.

This means that a method accepting the `Borrow` trait can cope equally with instances of `T` as well as references-to-`T`:

```
fn add_four<T: std::borrow::Borrow<i32>>(v: T) -> i32 {
    v.borrow() + 4
}
assert_eq!(add_four(&2), 6);
assert_eq!(add_four(2), 6);
```

The standard library's container types have more realistic uses of `Borrow`. For example, `HashMap::get` uses `Borrow` to allow convenient retrieval of entries whether keyed by value or by reference.

The `ToOwned` trait builds on the `Borrow` trait, adding a `to_owned()` method that produces a new owned item of the underlying type. This is a generalization of the `Clone` trait: where `Clone` specifically requires a Rust reference `&T`, `ToOwned` instead copes with things that implement `Borrow`.

This gives a couple of possibilities for handling both references and moved items in a unified way:

- A function that operates on references to some type can accept `Borrow` so that it can also be called with moved items as well as references.
- A function that operates on owned items of some type can accept `ToOwned` so that it can also be called with references to items as well as moved items; any references passed to it will be replicated into a locally owned item.

Although it's not a pointer type, the `Cow` type is worth mentioning at this point, because it provides an alternative way of dealing with the same kind of situation. `Cow` is an `enum` that can hold either owned data or a reference to borrowed data. The peculiar name stands for "clone-on-write": a `Cow` input can remain as borrowed data right up to the point where it needs to be modified, but it becomes an owned copy at the point where the data needs to be altered.

Smart Pointer Types

The Rust standard library includes a variety of types that act like pointers to some degree or another, mediated by the standard library traits previously described. These *smart pointer* types each come with some particular semantics and guarantees, which has the advantage that the right combination of them can give fine-grained control over the pointer's behavior, but has the disadvantage that the resulting types can seem overwhelming at first (`Rc<RefCell<Vec<T>>>` , anyone?).

The first smart pointer type is `Rc<T>` , which is a reference-counted pointer to an item (roughly analogous to C++'s `std::shared_ptr<T>`). It implements all of the pointer-related traits and so acts like a `Box<T>` in many ways.

This is useful for data structures where the same item can be reached in different ways, but it removes one of Rust's core rules around ownership—that each item has only one owner. Relaxing this rule means that it is now possible to leak data: if item A has an `Rc` pointer to item B, and item B has an `Rc` pointer to A, then the pair will never be dropped.[4] To put it another way: you need `Rc` to support cyclical data structures, but the downside is that there are now cycles in your data structures.

The risk of leaks can be ameliorated in some cases by the related `Weak<T>` type, which holds a non-owning reference to the underlying item (roughly analogous to C++'s `std::weak_ptr<T>`). Holding a weak reference doesn't prevent the underlying item from being dropped (when all strong references are removed), so making use of the `Weak<T>` involves an upgrade to an `Rc<T>` —which can fail.

Under the hood, `Rc` is (currently) implemented as a pair of reference counts together with the referenced item, all stored on the heap (as depicted in Figure 1-7):

```
use std::rc::Rc;
let rc1: Rc<u64> = Rc::new(42);
let rc2 = rc1.clone();
let wk = Rc::downgrade(&rc1);
```

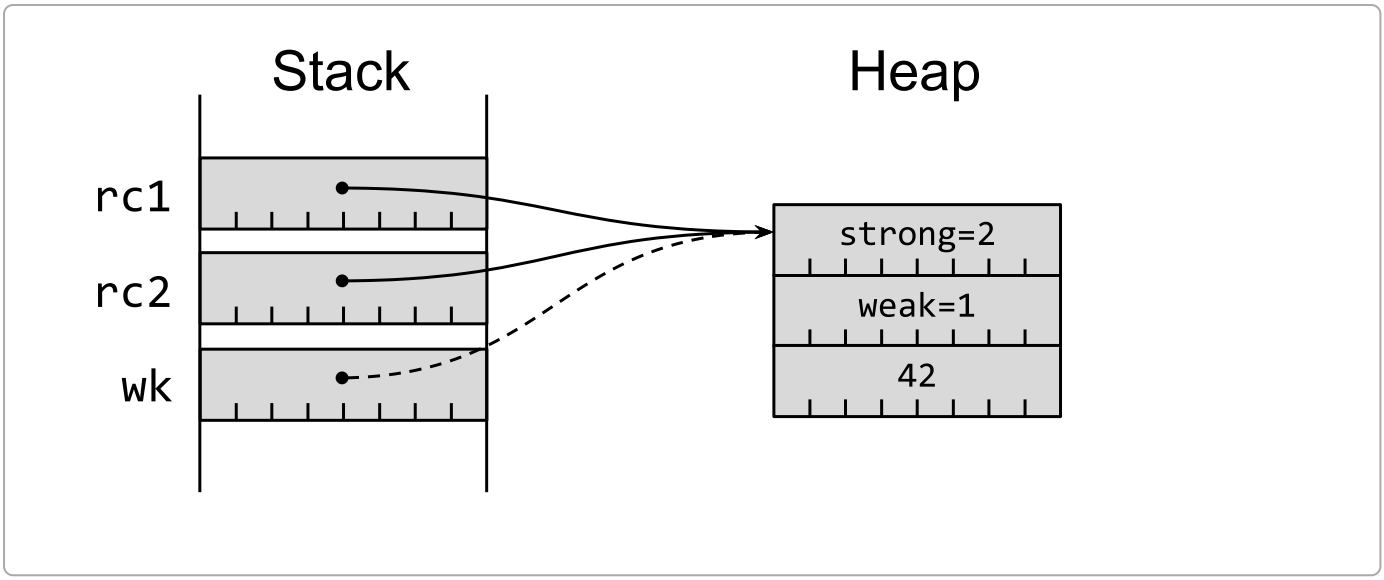


Figure 1-7. `Rc` and `Weak` pointers all referring to the same heap item

The underlying item is dropped when the strong reference count drops to zero, but the bookkeeping structure is dropped only when the weak reference count also drops to zero.

An `Rc` on its own gives you the ability to reach an item in different ways, but when you reach that item, you can modify it (via `get_mut`) only if there are no other ways to reach the item—i.e., there are no other extant `Rc` or `Weak` references to the same item. That's hard to arrange, so `Rc` is often combined with `RefCell`.

The next smart pointer type, `RefCell<T>`, relaxes the rule (Item 15) that an item can be mutated only by its owner or by code that holds the (only) mutable reference to the item. This *interior mutability* allows for greater flexibility—for example, allowing trait implementations that mutate internals even when the method signature allows only `&self`. However, it also incurs costs: as well as the extra storage overhead (an extra `isize` to track current borrows, as shown in Figure 1-8), the normal borrow checks are moved from compile time to runtime:

```
use std::cell::RefCell;
let rc: RefCell<u64> = RefCell::new(42);
let b1 = rc.borrow();
let b2 = rc.borrow();
```

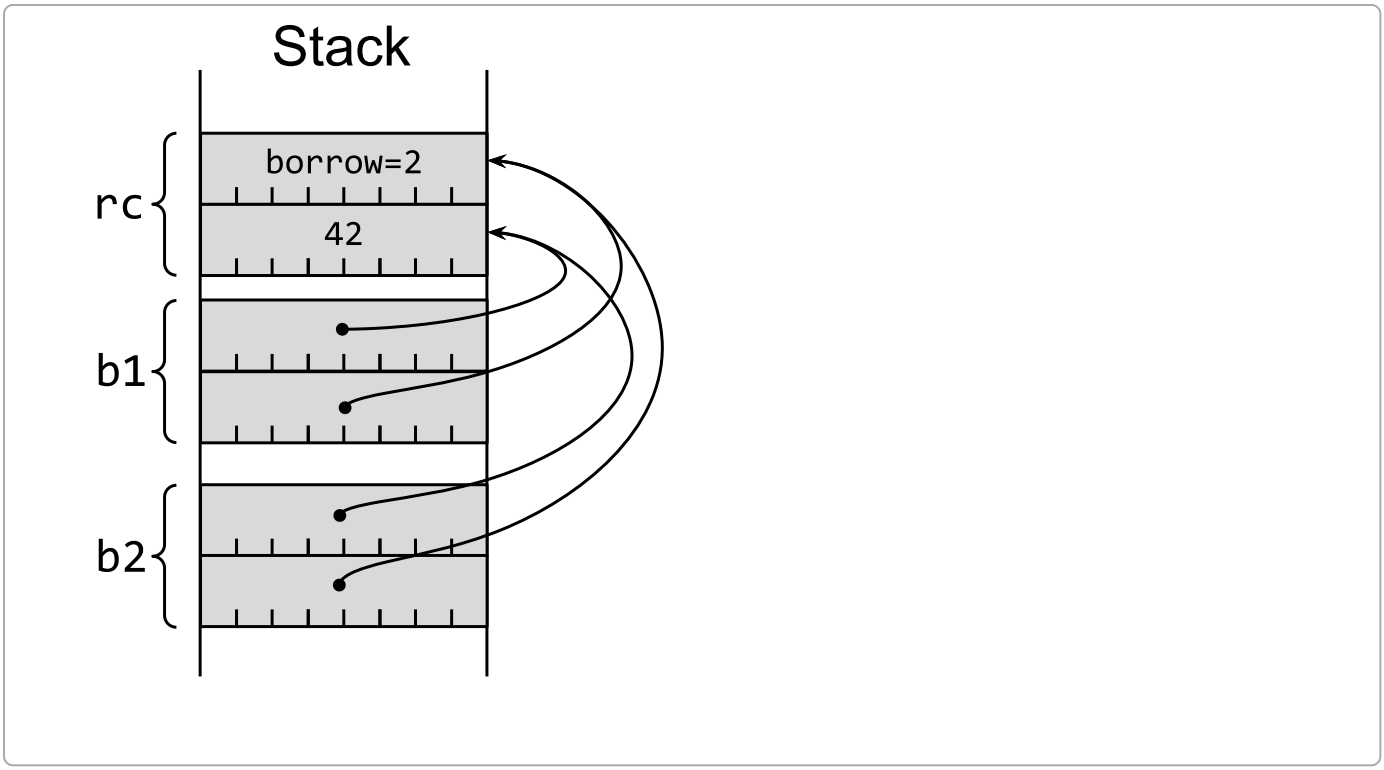


Figure 1-8. `Ref` borrows referring to a `RefCell` container

The runtime nature of these checks means that the `RefCell` user has to choose between two options, neither pleasant:

- Accept that borrowing is an operation that might fail, and cope with `Result` values from `try_borrow[_mut]`
- Use the allegedly infallible borrowing methods `borrow[_mut]`, and accept the risk of a `panic!` at runtime (Item 18) if the borrow rules have not been complied with

In either case, this runtime checking means that `RefCell` itself implements none of the standard pointer traits; instead, its access operations return a `Ref<T>` or `RefMut<T>` smart pointer type that does implement those traits.

If the underlying type `T` implements the `Copy` trait (indicating that a fast bit-for-bit copy produces a valid item; see Item 10), then the `Cell<T>` type allows interior mutation with less overhead—the `get(&self)` method copies out the current value, and the `set(&self, val)` method copies in a new value. The `Cell` type is used internally by both the `Rc` and `RefCell` implementations, for shared tracking of counters that can be mutated without a `&mut self`.

The smart pointer types described so far are suitable only for single-threaded use; their implementations assume that there is no concurrent access to their internals. If this is not the case, then smart pointers that include additional synchronization overhead are needed.

The thread-safe equivalent of `Rc<T>` is `Arc<T>`, which uses atomic counters to ensure that the reference counts remain accurate. Like `Rc`, `Arc` implements all of the various pointer-related traits.

However, `Arc` on its own does not allow any kind of mutable access to the underlying item. This is covered by the `Mutex` type, which ensures that only one thread has access—whether mutably or immutably—to the underlying item. As with `RefCell`, `Mutex` itself does not implement any pointer traits, but its `lock()` operation returns a value of a type that does: `MutexGuard`, which implements `Deref[Mut]`.

If there are likely to be more readers than writers, the `RwLock` type is preferable, as it allows multiple readers access to the underlying item in parallel, provided that there isn't currently a (single) writer.

In either case, Rust's borrowing and threading rules force the use of one of these synchronization containers in multithreaded code (but this guards against only *some* of the problems of shared-state concurrency; see Item 17).

The same strategy—see what the compiler rejects and what it suggests instead—can sometimes be applied with the other smart pointer types. However, it's faster and less frustrating to understand what the behavior of the different smart pointers implies. To borrow (pun intended) an example from the first edition of the Rust book:

- `Rc<RefCell<Vec<T>>>` holds a vector (`Vec`) with shared ownership (`Rc`), where the vector can be mutated—but only as a whole vector.
- `Rc<Vec<RefCell<T>>>` also holds a vector with shared ownership, but here each individual entry in the vector can be mutated independently of the others.

The types involved precisely describe these behaviors.

---

[1] Albeit with a warning from modern compilers.

[2] The equivalent trait for mutable expressions is `IndexMut`.

[3] This is somewhat simplified; a full vtable also includes information about the size and alignment of the type, together with a `drop()` function pointer so that the underlying object can be safely dropped.

[4] Note that this doesn't affect Rust's memory safety guarantees: the items are still safe, just inaccessible.

Item 9: Consider using iterator transforms instead of explicit loops

The humble loop has had a long journey of increasing convenience and increasing abstraction. The B language (the precursor to C) had only `while (condition) { ... }`, but with the arrival of C, the common scenario of iterating through indexes of an array became more convenient with the addition of the `for` loop:

```
// C code
int i;
for (i = 0; i < len; i++) {
  Item item = collection[i];
  // body
}
```

The early versions of C++ further improved convenience and scoping by allowing the loop variable declaration to be embedded in the `for` statement (this was also adopted by C in C99):

```
// C++98 code
for (int i = 0; i < len; i++) {
  Item item = collection[i];
  // ...
}
```

Most modern languages abstract the idea of the loop further: the core function of a loop is often to move to the next item of some container. Tracking the logistics that are required to reach that item (`index++` or `++it`) is mostly an irrelevant detail. This realization produced two core concepts:

- *Iterators*: A type whose purpose is to repeatedly emit the next item of a container, until exhausted[1]
- *For-each loops*: A compact loop expression for iterating over all of the items in a container, binding a loop variable to the *item* rather than to the details of reaching that item

These concepts allow for loop code that's shorter and (more importantly) clearer about what's intended:

```
// C++11 code
for (Item& item : collection) {
  // ...
}
```

Once these concepts were available, they were so obviously powerful that they were quickly retrofitted to those languages that didn't already have them (e.g., for-each loops were added to Java 1.5 and C++11).

Rust includes iterators and for-each–style loops, but it also includes the next step in abstraction: allowing the whole loop to be expressed as an *iterator transform* (sometimes also referred to as an *iterator adaptor*). As with Item 3's discussion of `Option` and `Result`, this Item will attempt to show how these iterator transforms can be used instead of explicit loops, and will give guidance as to when it's a good idea. In particular, iterator transforms can be more efficient than an explicit loop, because the compiler can skip the bounds checks it might otherwise need to perform.

By the end of this Item, a C-like explicit loop to sum the squares of the first five even items of a vector:

```
let values: Vec<u64> = vec![1, 1, 2, 3, 5 /* ... */];

let mut even_sum_squares = 0;
let mut even_count = 0;
for i in 0..values.len() {
    if values[i] % 2 != 0 {
        continue;
    }
    even_sum_squares += values[i] * values[i];
    even_count += 1;
    if even_count == 5 {
        break;
    }
}
```

should start to feel more natural expressed as a functional-style expression:

```
let even_sum_squares: u64 = values
    .iter()
    .filter(|x| *x % 2 == 0)
    .take(5)
    .map(|x| x * x)
    .sum();
```

Iterator transformation expressions like this can roughly be broken down into three parts:

- An initial source iterator, from an instance of a type that implements one of Rust's iterator traits
- A sequence of iterator transforms
- A final consumer method to combine the results of the iteration into a final value

The first two of these parts effectively move functionality out of the loop body and into the `for` expression; the last removes the need for the `for` statement altogether.

Iterator Traits

The core `Iterator` trait has a very simple interface: a single method `next` that yields `Some` items until it doesn't (`None`). The type of the emitted items is given by the trait's associated `Item` type.

Collections that allow iteration over their contents—what would be called *iterables* in other languages—implement the `IntoIterator` trait; the `into_iter` method of this trait consumes `Self` and emits an `Iterator` in its stead. The compiler will automatically use this trait for expressions of the form:

```
for item in collection {
    // body
}
```

effectively converting them to code roughly like:

```
let mut iter = collection.into_iter();
loop {
    let item: Thing = match iter.next() {
        Some(item) => item,
        None => break,
    };
    // body
}
```

or more succinctly and more idiomatically:

```
let mut iter = collection.into_iter();
while let Some(item) = iter.next() {
    // body
}
```

To keep things running smoothly, there's also an implementation of `IntoIterator` for any `Iterator` , which just returns `self` ; after all, it's easy to convert an `Iterator` into an `Iterator` !

This initial form is a consuming iterator, using up the collection as it's created:

```
let collection = vec![Thing(0), Thing(1), Thing(2), Thing(3)];
for item in collection {
    println!("Consumed item {item:?}");
}
```

Any attempt to use the collection after it's been iterated over fails:

```
println!("Collection = {collection:?}");
```

```
error[E0382]: borrow of moved value: `collection`
   --> src/main.rs:171:28
    |
163 |   let collection = vec![Thing(0), Thing(1), Thing(2), Thing(3)];
    |       ---------- move occurs because `collection` has type `Vec<Thing>`,
    |                  which does not implement the `Copy` trait
164 |   for item in collection {
    |               ---------- `collection` moved due to this implicit call to
    |                           `.into_iter()`
...
171 |   println!("Collection = {collection:?}");
    |                          ^^^^^^^^^^^^^^ value borrowed here after move
    |
note: `into_iter` takes ownership of the receiver `self`, which moves
      `collection`
```

While simple to understand, this all-consuming behavior is often undesired; some kind of *borrow* of the iterated items is needed.

To ensure that behavior is clear, the examples here use a `Thing` type that does *not* implement `Copy` (Item 10), as that would hide questions of ownership (Item 15)—the compiler would silently make copies everywhere:

```
// Deliberately not `Copy`
#[derive(Clone, Debug, Eq, PartialEq)]
struct Thing(u64);

let collection = vec![Thing(0), Thing(1), Thing(2), Thing(3)];
```

If the collection being iterated over is prefixed with `&`:

```
for item in &collection {
    println!("{}", item.0);
}
println!("collection still around {collection:?}");
```

then the Rust compiler will look for an implementation of `IntoIterator` for the type `&Collection`. Properly designed collection types will provide such an implementation; this implementation will still consume `Self`, but now `Self` is `&Collection` rather than `Collection`, and the associated `Item` type will be a reference `&Thing`.

This leaves the collection intact after iteration, and the equivalent expanded code is as follows:

```
let mut iter = (&collection).into_iter();
while let Some(item) = iter.next() {
    println!("{}", item.0);
}
```

If it makes sense to provide iteration over mutable references,[2] then a similar pattern applies for `for item in &mut collection`: the compiler looks for and uses an implementation of `IntoIterator` for `&mut Collection`, with each `Item` being of type `&mut Thing`.

By convention, standard containers also provide an `iter()` method that returns an iterator over references to the underlying item, and an equivalent `iter_mut()` method, if appropriate, with the same behavior as just described. These methods can be used in `for` loops but have a more obvious benefit when used as the start of an iterator transformation:

```
let result: u64 = (&collection).into_iter().map(|thing| thing.0).sum();
```

becomes:

```
let result: u64 = collection.iter().map(|thing| thing.0).sum();
```

Iterator Transforms

The `Iterator` trait has a single required method (`next`) but also provides default implementations (Item 13) of a large number of other methods that perform transformations on an iterator.

Some of these transformations affect the overall iteration process:

- `take(n)`: Restricts an iterator to emitting at most `n` items.
- `skip(n)`: Skips over the first `n` elements of the iterator.
- `step_by(n)`: Converts an iterator so it emits only every nth item.
- `chain(other)`: Glues together two iterators, to build a combined iterator that moves through one then the other.
- `cycle()`: Converts an iterator that terminates into one that repeats forever, starting at the beginning again whenever it reaches the end. (The iterator must support `Clone` to allow this.)
- `rev()`: Reverses the direction of an iterator. (The iterator must implement the `DoubleEndedIterator` trait, which has an additional `next_back` required method.)

Other transformations affect the nature of the `Item` that's the subject of the `Iterator`:

- `map(|item| {...})`: Repeatedly applies a closure to transform each item in turn. This is the most general version; several of the following entries in this list are convenience variants that could be equivalently implemented as a `map`.
- `cloned()`: Produces a clone of all of the items in the original iterator; this is particularly useful with iterators over `&Item` references. (This obviously requires the underlying `Item` type to implement `Clone`.)

- `copied()` : Produces a copy of all of the items in the original iterator; this is particularly useful with iterators over `&Item` references. (This obviously requires the underlying `Item` type to implement `Copy` , but it is likely to be faster than `cloned()` , if that's the case.)
- `enumerate()` : Converts an iterator over items to be an iterator over `(usize, Item)` pairs, providing an index to the items in the iterator.
- `zip(it)` : Joins an iterator with a second iterator, to produce a combined iterator that emits pairs of items, one from each of the original iterators, until the shorter of the two iterators is finished.

Yet other transformations perform filtering on the `Item` s being emitted by the `Iterator` :

- `filter(|item| {...})` : Applies a `bool` -returning closure to each item reference to determine whether it should be passed through.
- `take_while()` : Emits an initial subrange of the iterator, based on a predicate. Mirror image of `skip_while` .
- `skip_while()` : Emits a final subrange of the iterator, based on a predicate. Mirror image of `take_while` .

The `flatten()` method deals with an iterator whose items are themselves iterators, flattening the result. On its own, this doesn't seem that helpful, but it becomes much more useful when combined with the observation that both `Option` and `Result` act as iterators: they produce either zero (for `None` , `Err(e)`) or one (for `Some(v)` , `Ok(v)`) items. This means that `flatten` ing a stream of `Option` / `Result` values is a simple way to extract just the valid values, ignoring the rest.

Taken as a whole, these methods allow iterators to be transformed so that they produce exactly the sequence of elements that are needed for most situations.

Iterator Consumers

The previous two sections described how to obtain an iterator and how to transform it into exactly the right shape for precise iteration. This precisely targeted iteration could happen as an explicit for-each loop:

```
let mut even_sum_squares = 0;
for value in values.iter().filter(|x| *x % 2 == 0).take(5) {
    even_sum_squares += value * value;
}
```

However, the large collection of `Iterator` methods includes many that allow an iteration to be consumed in a single method call, removing the need for an explicit `for` loop.

The most general of these methods is `for_each(|item| {...})`, which runs a closure for each item produced by the `Iterator`. This can do *most* of the things that an explicit `for` loop can do (the exceptions are described in a later section), but its generality also makes it a little awkward to use—the closure needs to use mutable references to external state in order to emit anything:

```
let mut even_sum_squares = 0;
values
    .iter()
    .filter(|x| *x % 2 == 0)
    .take(5)
    .for_each(|value| {
        // closure needs a mutable reference to state elsewhere
        even_sum_squares += value * value;
    });
```

However, if the body of the `for` loop matches one of a number of common patterns, there are more specific iterator-consuming methods that are clearer, shorter, and more idiomatic.

These patterns include shortcuts for building a single value out of the collection:

- `sum()`: Sums a collection of numeric values (integers or floats).
- `product()`: Multiplies a collection of numeric values.
- `min()`: Finds the minimum value of a collection, relative to the `Item`'s `Ord` implementation (see Item 10).
- `max()`: Finds the maximum value of a collection, relative to the `Item`'s `Ord` implementation (see Item 10).
- `min_by(f)`: Finds the minimum value of a collection, relative to a user-specified comparison function `f`.
- `max_by(f)`: Finds the maximum value of a collection, relative to a user-specified comparison function `f`.
- `reduce(f)`: Builds an accumulated value of the `Item` type by running a closure at each step that takes the value accumulated so far and the current item. This is a more general operation that encompasses the previous methods.
- `fold(f)`: Builds an accumulated value of an arbitrary type (not just the `Iterator::Item` type) by running a closure at each step that takes the value accumulated so far and the current item. This is a generalization of `reduce`.
- `scan(init, f)`: Builds an accumulated value of an arbitrary type by running a closure at each step that takes a mutable reference to some internal state and the current item. This is a slightly different generalization of `reduce`.

There are also methods for *selecting* a single value out of the collection:

- `find(p)`: Finds the first item that satisfies a predicate.
- `position(p)`: Also finds the first item satisfying a predicate, but this time it returns the index of the item.
- `nth(n)`: Returns the `n`th element of the iterator, if available.

There are methods for testing against every item in the collection:

- `any(p)` : Indicates whether a predicate is `true` for *any* item in the collection.
- `all(p)` : Indicates whether a predicate is `true` for *all* items in the collection.

In either case, iteration will terminate early if the relevant counterexample is found.

There are methods that allow for the possibility of failure in the closures used with each item. In each case, if a closure returns a failure for an item, the iteration is terminated and the operation as a whole returns the first failure:

- `try_for_each(f)` : Behaves like `for_each` , but the closure can fail
- `try_fold(f)` : Behaves like `fold` , but the closure can fail
- `try_find(f)` : Behaves like `find` , but the closure can fail

Finally, there are methods that accumulate all of the iterated items into a new collection. The most important of these is `collect()` , which can be used to build a new instance of any collection type that implements the `FromIterator` trait.

The `FromIterator` trait is implemented for all of the standard library collection types (`Vec` , `HashMap` , `BTreeSet` , etc.), but this ubiquity also means that you often have to use explicit types, because otherwise the compiler can't figure out whether you're trying to assemble (say) a `Vec<i32>` or `HashSet<i32>` :

```
use std::collections::HashSet;

// Build collections of even numbers.  Type must be specified, because
// the expression is the same for either type.
let myvec: Vec<i32> = (0..10).into_iter().filter(|x| x % 2 == 0).collect();
let h: HashSet<i32> = (0..10).into_iter().filter(|x| x % 2 == 0).collect();
```

This example also illustrates the use of range expressions to generate the initial data to be iterated over.

Other (more obscure) collection-producing methods include the following:

- `unzip()` : Divides an iterator of pairs into two collections
- `partition(p)` : Splits an iterator into two collections based on a predicate that is applied to each item

This Item has touched on a wide selection of `Iterator` methods, but this is only a subset of the methods available; for more information, consult the iterator documentation or read Chapter 15 of *Programming Rust*, 2nd edition (O'Reilly), which has extensive coverage of the possibilities.

This rich collection of iterator transformations is there to be used. It produces code that is more idiomatic, more compact, and has clearer intent.

Expressing loops as iterator transformations can also produce code that is more efficient. In the interests of safety, Rust performs *bounds checking* on access to contiguous containers such as vectors and slices; an attempt to access a value beyond the bounds of the collection triggers a panic rather than an access to invalid data. An old-style loop that accesses container values (e.g., `values[i]`) *might* be subject to these runtime checks, whereas an iterator that produces one value after another is already known to be within range.

However, it's also the case that an old-style loop might *not* be subject to additional bounds checks compared to the equivalent iterator transformation. The Rust compiler and optimizer is very good at analyzing the code surrounding a slice access to determine whether it's safe to skip the bounds checks; Sergey "Shnatsel" Davidoff's 2023 article explores the subtleties involved.

Building Collections from `Result` Values

The previous section described the use of `collect()` to build collections from iterators, but `collect()` also has a particularly helpful feature when dealing with `Result` values.

Consider an attempt to convert a vector of `i64` values into bytes (`u8`), with the optimistic expectation that they will all fit:

```
// In the 2021 edition of Rust, `TryFrom` is in the prelude, so this
// `use` statement is no longer needed.
use std::convert::TryFrom;

let inputs: Vec<i64> = vec![0, 1, 2, 3, 4];
let result: Vec<u8> = inputs
    .into_iter()
    .map(|v| <u8>::try_from(v).unwrap())
    .collect();
```

This works until some unexpected input comes along:

```
let inputs: Vec<i64> = vec![0, 1, 2, 3, 4, 512];
```

and causes a runtime failure:

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value:
TryFromIntError(())', iterators/src/main.rs:266:36
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Following the advice given in Item 3, we want to keep the `Result` type in play and use the `?` operator to make any failure the problem of the calling code. The obvious modification to emit the `Result` doesn't really help:

```
let result: Vec<Result<u8, _>> =
    inputs.into_iter().map(|v| <u8>::try_from(v)).collect();
// Now what?  Still need to iterate to extract results and detect errors.
```

However, there's an alternative version of `collect()`, which can assemble a `Result` holding a `Vec`, instead of a `Vec` holding `Result`s.

Forcing use of this version requires the turbofish (`::<Result<Vec<_>, _>>`):

```
let result: Vec<u8> = inputs
    .into_iter()
    .map(|v| <u8>::try_from(v))
    .collect::<Result<Vec<_>, _>>()?;
```

Combining this with the question mark operator gives useful behavior:

- If the iteration encounters an error value, that error value is emitted to the caller and iteration stops.
- If no errors are encountered, the remainder of the code can deal with a sensible collection of values of the right type.

Loop Transformation

The aim of this Item is to convince you that many explicit loops can be regarded as something to be converted to iterator transformations. This can feel somewhat unnatural for programmers who aren't used to it, so let's walk through a transformation step by step.

Starting with a very C-like explicit loop to sum the squares of the first five even items of a vector:

```
let mut even_sum_squares = 0;
let mut even_count = 0;
for i in 0..values.len() {
    if values[i] % 2 != 0 {
        continue;
    }
    even_sum_squares += values[i] * values[i];
    even_count += 1;
    if even_count == 5 {
        break;
    }
}
```

The first step is to replace vector indexing with direct use of an iterator in a for-each loop:

```
let mut even_sum_squares = 0;
let mut even_count = 0;
for value in values.iter() {
    if value % 2 != 0 {
        continue;
    }
    even_sum_squares += value * value;
    even_count += 1;
    if even_count == 5 {
        break;
    }
}
```

An initial arm of the loop that uses `continue` to skip over some items is naturally expressed as a `filter()`:

```
let mut even_sum_squares = 0;
let mut even_count = 0;
for value in values.iter().filter(|x| *x % 2 == 0) {
    even_sum_squares += value * value;
    even_count += 1;
    if even_count == 5 {
        break;
    }
}
```

Next, the early exit from the loop once five even items have been spotted maps to a `take(5)`:

```
let mut even_sum_squares = 0;
for value in values.iter().filter(|x| *x % 2 == 0).take(5) {
    even_sum_squares += value * value;
}
```

Every iteration of the loop uses only the item squared, in the `value * value` combination, which makes it an ideal target for a `map()`:

```
let mut even_sum_squares = 0;
for val_sqr in values.iter().filter(|x| *x % 2 == 0).take(5).map(|x| x * x)
{
    even_sum_squares += val_sqr;
}
```

These refactorings of the original loop result in a loop body that's the perfect nail to fit under the hammer of the `sum()` method:

```
let even_sum_squares: u64 = values
    .iter()
    .filter(|x| *x % 2 == 0)
    .take(5)
    .map(|x| x * x)
    .sum();
```

When Explicit Is Better

This Item has highlighted the advantages of iterator transformations, particularly with respect to concision and clarity. So when are iterator transformations *not* appropriate or idiomatic?

- If the loop body is large and/or multifunctional, it makes sense to keep it as an explicit body rather than squeezing it into a closure.
- If the loop body involves error conditions that result in early termination of the surrounding function, these are often best kept explicit—the `try_..()` methods help only a little. However, `collect()` 's ability to convert a collection of `Result` values into a `Result` holding a collection of values often allows error conditions to still be handled with the `?` operator.
- If performance is vital, an iterator transform that involves a closure *should* get optimized so that it is just as fast as the equivalent explicit code. But if performance of a core loop is that important, *measure* different variants and tune appropriately:
    - Be careful to ensure that your measurements reflect real-world performance—the compiler's optimizer can give overoptimistic results on test data (as described in Item 30).
    - The Godbolt compiler explorer is an amazing tool for exploring what the compiler spits out.

Most importantly, don't convert a loop into an iteration transformation if the conversion is forced or awkward. This is a matter of taste to be sure—but be aware that your taste is likely to change as you become more familiar with the functional style.

---

[1] In fact, the iterator can be more general—the idea of emitting next items until completion need not be associated with a container.

[2] This method can't be provided if a mutation to the item might invalidate the container's internal guarantees. For example, changing the item's contents in a way that alters its `Hash` value would invalidate the internal data structures of a `HashMap` .

Traits

The second core pillar of Rust's type system is the use of traits, which allow the encoding of behavior that is common across distinct types. A `trait` is roughly equivalent to an interface type in other languages, but they are also tied to Rust's *generics* (Item 12), to allow interface reuse without runtime overhead.

The Items in this chapter describe the standard traits that the Rust compiler and the Rust toolchain make available, and provide advice on how to design and use trait-encoded behavior.

Item 10: Familiarize yourself with standard traits

Rust encodes key behavioral aspects of its type system in the type system itself, through a collection of fine-grained standard traits that describe those behaviors (see Item 2).

Many of these traits will seem familiar to programmers coming from C++, corresponding to concepts such as copy-constructors, destructors, equality and assignment operators, etc.

As in C++, it's often a good idea to implement many of these traits for your own types; the Rust compiler will give you helpful error messages if some operation needs one of these traits for your type and it isn't present.

Implementing such a large collection of traits may seem daunting, but most of the common ones can be automatically applied to user-defined types, using `derive` macros. These `derive` macros generate code with the "obvious" implementation of the trait for that type (e.g., field-by-field comparison for `Eq` on a `struct`); this normally requires that all constituent parts also implement the trait. The auto-generated implementation is *usually* what you want, but there are occasional exceptions discussed in each trait's section that follows.

The use of the `derive` macros does lead to type definitions like:

```
#[derive(Clone, Copy, Debug, PartialEq, Eq, PartialOrd, Ord, Hash)]
enum MyBooleanOption {
    Off,
    On,
}
```

where auto-generated implementations are triggered for eight different traits.

This fine-grained specification of behavior can be disconcerting at first, but it's important to be familiar with the most common of these standard traits so that the available behaviors of a type definition can be immediately understood.

Common Standard Traits

This section discusses the most commonly encountered standard traits. Here are rough one-sentence summaries of each:

- `Clone` : Items of this type can make a copy of themselves when asked, by running user-defined code.

- `Copy` : If the compiler makes a bit-for-bit copy of this item's memory representation (without running any user-defined code), the result is a valid new item.
- `Default` : It's possible to make new instances of this type with sensible default values.
- `PartialEq` : There's a partial equivalence relation for items of this type—any two items can be definitively compared, but it may not always be true that `x==x` .
- `Eq` : There's an equivalence relation for items of this type—any two items can be definitively compared, and it *is* always true that `x==x` .
- `PartialOrd` : *Some* items of this type can be compared and ordered.
- `Ord` : *All* items of this type can be compared and ordered.
- `Hash` : Items of this type can produce a stable hash of their contents when asked.
- `Debug` : Items of this type can be displayed to programmers.
- `Display` : Items of this type can be displayed to users.

These traits can all be `derive` d for user-defined types, with the exception of `Display` (included here because of its overlap with `Debug`). However, there are occasions when a manual implementation—or no implementation—is preferable.

The following sections discuss each of these common traits in more detail.

`Clone`

The `Clone` trait indicates that it's possible to make a new copy of an item, by calling the `clone()` method. This is roughly equivalent to C++'s copy-constructor but is more explicit: the compiler will never silently invoke this method on its own (read on to the next section for that).

`Clone` can be `derive` d for a type if all of the item's fields implement `Clone` themselves. The `derive` d implementation clones an aggregate type by cloning each of its members in turn; again, this is roughly equivalent to a default copy-constructor in C++. This makes the trait opt-in (by adding `#[derive(Clone)]`), in contrast to the opt-out behavior in C++ (`MyType(const MyType&) = delete;`).

This is such a common and useful operation that it's more interesting to investigate the situations where you shouldn't or can't implement `Clone` , or where the default `derive` implementation isn't appropriate.

- You *shouldn't* implement `Clone` if the item embodies unique access to some resource (such as an RAII type; Item 11), or when there's another reason to restrict copies (e.g., if the item holds cryptographic key material).
- You *can't* implement `Clone` if some component of your type is un- `Clone` able in turn. Examples include the following:
  - Fields that are mutable references (`&mut T`), because the borrow checker (Item 15) allows only a single mutable reference at a time.

    - Standard library types that fall into the previous category, such as `MutexGuard` (embodies unique access) or `Mutex` (restricts copies for thread safety).
- You should *manually implement* `Clone` if there is anything about your item that won't be captured by a (recursive) field-by-field copy or if there is additional bookkeeping associated with item lifetimes. For example, consider a type that tracks the number of extant items at runtime for metrics purposes; a manual `Clone` implementation can ensure the counter is kept accurate.

Copy

The `Copy` trait has a trivial declaration:

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

There are no methods in this trait, meaning that it is a *marker trait* (as described in Item 2): it's used to indicate some constraint on the type that's not directly expressed in the type system.

In the case of `Copy`, the meaning of this marker is that a bit-for-bit copy of the memory holding an item gives a correct new item. Effectively, this trait is a marker that says that a type is a "plain old data" (POD) type.

This also means that the `Clone` trait bound can be slightly confusing: although a `Copy` type has to implement `Clone`, when an instance of the type is copied, the `clone()` method is *not* invoked—the compiler builds the new item without any involvement of user-defined code.

In contrast to user-defined marker traits (Item 2), `Copy` has a special significance to the compiler (as do several of the other marker traits in `std::marker`) over and above being available for trait bounds—it shifts the compiler from *move semantics* to *copy semantics*.

With move semantics for the assignment operator, what the right hand giveth, the left hand taketh away:

```
#[derive(Debug, Clone)]
struct KeyId(u32);

let k = KeyId(42);
let k2 = k; // value moves out of k into k2
println!("k = {k:?}");
```

```
error[E0382]: borrow of moved value: `k`
  --> src/main.rs:60:23
   |
58 |         let k = KeyId(42);
   |             - move occurs because `k` has type `main::KeyId`, which does
   |               not implement the `Copy` trait
59 |         let k2 = k; // value moves out of k into k2
   |                  - value moved here
60 |         println!("k = {k:?}");
   |                       ^^^^^ value borrowed here after move
   |
   = note: this error originates in the macro `$crate::format_args_nl`
help: consider cloning the value if the performance cost is acceptable
   |
59 |         let k2 = k.clone(); // value moves out of k into k2
   |                   ++++++++
```

With copy semantics, the original item lives on:

```
#[derive(Debug, Clone, Copy)]
struct KeyId(u32);

let k = KeyId(42);
let k2 = k; // value bitwise copied from k to k2
println!("k = {k:?}");
```

This makes `Copy` one of the most important traits to watch out for: it fundamentally changes the behavior of assignments—including parameters for method invocations.

In this respect, there are again overlaps with C++'s copy-constructors, but it's worth emphasizing a key distinction: in Rust there is no way to get the compiler to silently invoke user-defined code—it's either explicit (a call to `.clone()`) or it's not user-defined (a bitwise copy).

Because `Copy` has a `Clone` trait bound, it's possible to `.clone()` any `Copy`-able item. However, it's not a good idea: a bitwise copy will always be faster than invoking a trait method. Clippy (Item 29) will warn you about this:

```
let k3 = k.clone();
```

```
warning: using `clone` on type `KeyId` which implements the `Copy` trait
  --> src/main.rs:79:14
   |
79 |     let k3 = k.clone();
   |              ^^^^^^^^^ help: try removing the `clone` call: `k`
   |
```

As with `Clone`, it's worth exploring when you should or should not implement `Copy`:

- The obvious: **don't implement `Copy` if a bitwise copy doesn't produce a valid item**. That's likely to be the case if `Clone` needed a manual implementation rather than an automatically `derive` d implementation.
- It may be a bad idea to implement `Copy` if your type is large. The basic promise of `Copy` is that a bitwise copy is valid; however, this often goes hand in hand with an assumption that making the copy is fast. If that's not the case, skipping `Copy` prevents accidental slow copies.
- You *can't* implement `Copy` if some component of your type is un- `Copy` able in turn.
- If all of the components of your type are `Copy` able, then it's usually worth deriving `Copy` . The compiler has an off-by-default lint `missing_copy_implementations` that points out opportunities for this.

Default

The `Default` trait defines a *default constructor*, via a `default()` method. This trait can be `derive` d for user-defined types, provided that all of the subtypes involved have a `Default` implementation of their own; if they don't, you'll have to implement the trait manually. Continuing the comparison with C++, notice that a default constructor has to be explicitly triggered—the compiler does not create one automatically.

The `Default` trait can also be `derive` d for `enum` types, as long as there's a `#[default]` attribute to give the compiler a hint as to which variant is, well, default:

```
#[derive(Default)]
enum IceCreamFlavor {
    Chocolate,
    Strawberry,
    #[default]
    Vanilla,
}
```

The most useful aspect of the `Default` trait is its combination with *struct update syntax*. This syntax allows `struct` fields to be initialized by copying or moving their contents from an existing instance of the same `struct` , for any fields that aren't explicitly initialized. The template to copy from is given at the end of the initialization, after `..` , and the `Default` trait provides an ideal template to use:

```
#[derive(Default)]
struct Color {
    red: u8,
    green: u8,
    blue: u8,
    alpha: u8,
}

let c = Color {
    red: 128,
    ..Default::default()
};
```

This makes it much easier to initialize structures with lots of fields, only some of which have nondefault values. (The builder pattern, Item 7, may also be appropriate for these situations.)

`PartialEq` and `Eq`

The `PartialEq` and `Eq` traits allow you to define equality for user-defined types. These traits have special significance because if they're present, the compiler will automatically use them for equality (`==`) checks, similarly to `operator==` in C++. The default `derive` implementation does this with a recursive field-by-field comparison.

The `Eq` version is just a marker trait extension of `PartialEq` that adds the assumption of *reflexivity*: any type `T` that claims to support `Eq` should ensure that `x == x` is true for any `x: T`.

This is sufficiently odd to immediately raise the question, When wouldn't `x == x`? The primary rationale behind this split relates to floating point numbers,[1] and specifically to the special "not a number" value NaN (`f32::NAN` / `f64::NAN` in Rust). The floating point specifications require that nothing compares equal to NaN, *including NaN itself*; the `PartialEq` trait is the knock-on effect of this.

For user-defined types that don't have any float-related peculiarities, you should **implement `Eq` whenever you implement `PartialEq`**. The full `Eq` trait is also required if you want to use the type as the key in a `HashMap` (as well as the `Hash` trait).

You should implement `PartialEq` manually if your type contains any fields that do not affect the item's identity, such as internal caches and other performance optimizations. (Any manual implementation will also be used for `Eq` if it is defined, because `Eq` is just a marker trait that has no methods of its own.)

`PartialOrd` and `Ord`

The ordering traits `PartialOrd` and `Ord` allow comparisons between two items of a type, returning `Less`, `Greater`, or `Equal`. The traits require equivalent equality traits to be implemented (`PartialOrd` requires `PartialEq`; `Ord` requires `Eq`), and the two have to agree with each other (watch out for this with manual implementations in particular).

As with the equality traits, the comparison traits have special significance because the compiler will automatically use them for comparison operations (`<`, `>`, `<=`, `>=`).

The default implementation produced by `derive` compares fields (or `enum` variants) lexicographically in the order they're defined, so if this isn't correct, you'll need to implement the traits manually (or reorder the fields).

Unlike `PartialEq`, the `PartialOrd` trait does correspond to a variety of real situations. For example, it could be used to express a subset relationship among collections:[2] `{1, 2}` is a subset of `{1, 2, 4}`, but `{1, 3}` is not a subset of `{2, 4}`, nor vice versa.

However, even if a partial order does accurately model the behavior of your type, **be wary of implementing just `PartialOrd` and not `Ord`** (a rare occasion that contradicts the advice in Item 2 to encode behavior in the type system)—it can lead to surprising results:

```
// Inherit the `PartialOrd` behavior from `f32`.
#[derive(PartialOrd, PartialEq)]
struct Oddity(f32);

// Input data with NaN values is likely to give unexpected results.
let x = Oddity(f32::NAN);
let y = Oddity(f32::NAN);

// A self-comparison looks like it should always be true, but it may not be.
if x <= x {
    println!("This line doesn't get executed!");
}

// Programmers are also unlikely to write code that covers all possible
// comparison arms; if the types involved implemented `Ord`, then the
// second two arms could be combined.
if x <= y {
    println!("y is bigger"); // Not hit.
} else if y < x {
    println!("x is bigger"); // Not hit.
} else {
    println!("Neither is bigger");
}
```

Hash

The `Hash` trait is used to produce a single value that has a high probability of being different for different items. This hash value is used as the basis for hash-bucket-based data structures like `HashMap` and `HashSet`; as such, the type of the keys in these data structures must implement `Hash` (and `Eq`).

Flipping this around, it's essential that the "same" items (as per `Eq`) always produce the same hash: if `x == y` (via `Eq`), then it must always be true that `hash(x) == hash(y)`. **If you have a manual `Eq` implementation, check whether you also need a manual implementation of `Hash` to comply with this requirement.**

Debug and Display

The `Debug` and `Display` traits allow a type to specify how it should be included in output, for either normal (`{}` format argument) or debugging purposes (`{:?}` format argument), roughly analogous to an `operator<<` overload for `iostream` in C++.

The differences between the intents of the two traits go beyond which format specifier is needed, though:

- `Debug` can be automatically derived, `Display` can only be manually implemented.
- The layout of `Debug` output may change between different Rust versions. If the output will ever be parsed by other code, use `Display`.
- `Debug` is programmer-oriented; `Display` is user-oriented. A thought experiment that helps with this is to consider what would happen if the program was localized to a language that the authors don't speak— `Display` is appropriate if the content should be translated, `Debug` if not.

As a general rule, **add an automatically generated `Debug` implementation for your types** unless they contain sensitive information (personal details, cryptographic material, etc.). To make this advice easier to comply with, the Rust compiler includes a `missing_debug_implementations` lint that points out types without `Debug`. This lint is disabled by default but can be enabled for your code with either of the following:

```
#![warn(missing_debug_implementations)]
```

```
#![deny(missing_debug_implementations)]
```

If the automatically generated implementation of `Debug` would emit voluminous amounts of detail, then it may be more appropriate to include a manual implementation of `Debug` that summarizes the type's contents.

Implement `Display` if your types are designed to be shown to end users in textual output.

Standard Traits Covered Elsewhere

In addition to the common traits described in the previous section, the standard library also includes other standard traits that are less ubiquitous. Of these additional standard traits, the following are the most important, but they are covered in other Items and so are not covered here in depth:

- `Fn`, `FnOnce`, and `FnMut`: Items implementing these traits represent closures that can be invoked. See Item 2.
- `Error`: Items implementing this trait represent error information that can be displayed to users or programmers, and that *may* hold nested suberror information. See Item 4.
- `Drop`: Items implementing this trait perform processing when they are destroyed, which is essential for RAII patterns. See Item 11.
- `From` and `TryFrom`: Items implementing these traits can be automatically created from items of some other type but with a possibility of failure in the latter case. See Item 5.
- `Deref` and `DerefMut`: Items implementing these traits are pointer-like objects that can be dereferenced to get access to an inner item. See Item 8.
- `Iterator` and friends: Items implementing these traits represent collections that can be iterated over. See Item 9.
- `Send`: Items implementing this trait are safe to transfer between multiple threads. See Item 17.
- `Sync`: Items implementing this trait are safe to be referenced by multiple threads. See Item 17.

None of these traits are `derive` able.

Operator Overloads

The final category of standard traits relates to operator overloads, where Rust allows various built-in unary and binary operators to be overloaded for user-defined types, by implementing various standard traits from the `std::ops` module. These traits are not derivable and are typically needed only for types that represent "algebraic" objects, where there is a natural interpretation of these operators.

However, experience from C++ has shown that it's best to **avoid overloading operators for unrelated types** as it often leads to code that is hard to maintain and has unexpected performance properties (e.g., `x + y` silently invokes an expensive O(N) method).

To comply with the principle of least astonishment, if you implement any operator overloads, you should **implement a coherent set of operator overloads**. For example, if `x`

`+ y` has an overload (`Add`), and `-y` (`Neg`) does too, then you should also implement `x - y` (`Sub`) and make sure it gives the same answer as `x + (-y)` .

The items passed to the operator overload traits are moved, which means that non- `Copy` types will be consumed by default. Adding implementations for `&'a MyType` can help with this but requires more boilerplate to cover all of the possibilities (e.g., there are 4 = 2 × 2 possibilities for combining reference/non-reference arguments to a binary operator).

Summary

This item has covered a lot of ground, so some tables that summarize the standard traits that have been touched on are in order. First, Table 2-1 covers the traits that this Item covers in depth, all of which can be automatically `derive` d *except* `Display` .

Table 2-1. Common standard traits

Trait	Compiler Use	Bound	Methods
`Clone`			`clone`
`Copy`	`let y = x;`	`Clone`	Marker trait
`Default`			`default`
`PartialEq`	`x == y`		`eq`
`Eq`	`x == y`	`PartialEq`	Marker trait
`PartialOrd`	`x < y, x <= y,...`	`PartialEq`	`partial_cmp`
`Ord`	`x < y, x <= y,...`	`Eq + PartialOrd`	`cmp`
`Hash`			`hash`
`Debug`	`format!("{:?}", x)`		`fmt`
`Display`	`format!("{}", x)`		`fmt`

The operator overloads are summarized in Table 2-2.[3] None of these can be `derive` d.

Table 2-2. Operator overload traits

Trait	Compiler Use	Bound	Methods
`Add`	`x + y`		`add`
`AddAssign`	`x += y`		`add_assign`
`BitAnd`	`x & y`		`bitand`
`BitAndAssign`	`x &= y`		`bitand_assign`
`BitOr`	`x | y`		`bitor`
`BitOrAssign`	`x |= y`		`bitor_assign`

Trait	Compiler Use	Bound	Methods
BitXor	x ^ y		bitxor
BitXorAssign	x ^= y		bitxor_assign
Div	x / y		div
DivAssign	x /= y		div_assign
Mul	x * y		mul
MulAssign	x *= y		mul_assign
Neg	-x		neg
Not	!x		not
Rem	x % y		rem
RemAssign	x %= y		rem_assign
Shl	x << y		shl
ShlAssign	x <<= y		shl_assign
Shr	x >> y		shr
ShrAssign	x >>= y		shr_assign
Sub	x - y		sub
SubAssign	x -= y		sub_assign

For completeness, the standard traits that are covered in other items are included in Table 2-3; none of these traits are `derive` able (but `Send` and `Sync` may be automatically implemented by the compiler).

Table 2-3. Standard traits described in other Items

Trait	Item	Compiler Use	Bound	Methods
Fn	Item 2	x(a)	FnMut	call
FnMut	Item 2	x(a)	FnOnce	call_mut
FnOnce	Item 2	x(a)		call_once
Error	Item 4		Display + Debug	[source]
From	Item 5			from
TryFrom	Item 5			try_from
Into	Item 5			into

Trait	Item	Compiler Use	Bound	Methods
`TryInto`	Item 5			`try_into`
`AsRef`	Item 8			`as_ref`
`AsMut`	Item 8			`as_mut`
`Borrow`	Item 8			`borrow`
`BorrowMut`	Item 8		`Borrow`	`borrow_mut`
`ToOwned`	Item 8			`to_owned`
`Deref`	Item 8	`*x`, `&x`		`deref`
`DerefMut`	Item 8	`*x`, `&mut x`	`Deref`	`deref_mut`
`Index`	Item 8	`x[idx]`		`index`
`IndexMut`	Item 8	`x[idx] = ...`	`Index`	`index_mut`
`Pointer`	Item 8	`format("{:p}", x)`		`fmt`
`Iterator`	Item 9			`next`
`IntoIterator`	Item 9	`for y in x`		`into_iter`
`FromIterator`	Item 9			`from_iter`
`ExactSizeIterator`	Item 9		`Iterator`	(`size_hint`)
`DoubleEndedIterator`	Item 9		`Iterator`	`next_back`
`Drop`	Item 11	`}` (end of scope)		`drop`
`Sized`	Item 12			Marker trait
`Send`	Item 17	cross-thread transfer		Marker trait

Trait	Item	Compiler Use	Bound	Methods
`Sync`	Item 17	cross-thread use		Marker trait

---

[1] Of course, comparing floats for equality is always a dangerous game, as there is typically no guarantee that rounded calculations will produce a result that is bit-for-bit identical to the number you first thought of.

[2] More generally, any lattice structure also has a partial order.

[3] Some of the names here are a little cryptic—e.g., `Rem` for remainder and `Shl` for shift left—but the `std::ops` documentation makes the intended use clear.

Item 11: Implement the `Drop` trait for RAII patterns

> "Never send a human to do a machine's job." – Agent Smith

RAII stands for "Resource Acquisition Is Initialization" which is a programming pattern where the lifetime of a value is exactly tied to the lifecycle of some additional resource. The RAII pattern was popularized by the C++ programming language and is one of C++'s biggest contributions to programming.

The correlation between the lifetime of a value and the lifecycle of a resource is encoded in an RAII type:

- The type's *constructor* acquires access to some resource
- The type's *destructor* releases access to that resource

The result of this is that the RAII type has an *invariant*: access to the underlying resource is available if and only if the item exists. Because the compiler ensures that local variables are destroyed at scope exit, this in turn means that the underlying resources are also released at scope exit.[1]

This is particularly helpful for *maintainability*: if a subsequent change to the code alters the control flow, item and resource lifetimes are still correct. To see this, consider some code that manually locks and unlocks a mutex, without using the RAII pattern; this code is in C++, because Rust's `Mutex` doesn't allow this kind of error-prone usage!

```
// C++ code
class ThreadSafeInt {
 public:
  ThreadSafeInt(int v) : value_(v) {}

  void add(int delta) {
    mu_.lock();
    // ... more code here
    value_ += delta;
    // ... more code here
    mu_.unlock();
  }
```

A modification to catch an error condition with an early exit leaves the mutex locked:

```
// C++ code
void add_with_modification(int delta) {
  mu_.lock();
  // ... more code here
  value_ += delta;
  // Check for overflow.
  if (value_ > MAX_INT) {
    // Oops, forgot to unlock() before exit
    return;
  }
  // ... more code here
  mu_.unlock();
}
```

However, encapsulating the locking behavior into an RAII class:

```
// C++ code (real code should use std::lock_guard or similar)
class MutexLock {
 public:
  MutexLock(Mutex* mu) : mu_(mu) { mu_->lock(); }
  ~MutexLock()                   { mu_->unlock(); }
 private:
  Mutex* mu_;
};
```

means the equivalent code is safe for this kind of modification:

```
// C++ code
void add_with_modification(int delta) {
  MutexLock with_lock(&mu_);
  // ... more code here
  value_ += delta;
  // Check for overflow.
  if (value_ > MAX_INT) {
    return; // Safe, with_lock unlocks on the way out
  }
  // ... more code here
}
```

In C++, RAII patterns were often originally used for memory management, to ensure that manual allocation (`new` , `malloc()`) and deallocation (`delete` , `free()`) operations were kept in sync. A general version of this memory management was added to the C++ standard library in C++11: the `std::unique_ptr<T>` type ensures that a single place has "ownership" of memory but allows a pointer to the memory to be "borrowed" for ephemeral use (`ptr.get()`).

In Rust, this behavior for memory pointers is built into the language (Item 15), but the general principle of RAII is still useful for other kinds of resources.[2] **Implement `Drop` for any types that hold resources that must be released**, such as the following:

- Access to operating system resources. For Unix-derived systems, this usually means something that holds a *file descriptor*; failing to release these correctly will hold onto system resources (and will also eventually lead to the program hitting the per-process file descriptor limit).
- Access to synchronization resources. The standard library already includes memory synchronization primitives, but other resources (e.g., file locks, database locks, etc.) may need similar encapsulation.
- Access to raw memory, for `unsafe` types that deal with low-level memory management (e.g., for foreign function interface [FFI] functionality).

The most obvious instance of RAII in the Rust standard library is the `MutexGuard` item returned by `Mutex::lock()` operations, which tend to be widely used for programs that use the shared-state parallelism discussed in Item 17. This is roughly analogous to the final C++ example shown earlier, but in Rust the `MutexGuard` item acts as a proxy to the mutex-protected data in addition to being an RAII item for the held lock:

```
use std::sync::Mutex;

struct ThreadSafeInt {
    value: Mutex<i32>,
}

impl ThreadSafeInt {
    fn new(val: i32) -> Self {
        Self {
            value: Mutex::new(val),
        }
    }
    fn add(&self, delta: i32) {
        let mut v = self.value.lock().unwrap();
        *v += delta;
    }
}
```

Item 17 advises against holding locks for large sections of code; to ensure this, **use blocks to restrict the scope of RAII items**. This leads to slightly odd indentation, but it's worth it for the added safety and lifetime precision:

```
impl ThreadSafeInt {
    fn add_with_extras(&self, delta: i32) {
        // ... more code here that doesn't need the lock
        {
            let mut v = self.value.lock().unwrap();
            *v += delta;
        }
        // ... more code here that doesn't need the lock
    }
}
```

Having proselytized the uses of the RAII pattern, an explanation of how to implement it is in order. The `Drop` trait allows you to add user-defined behavior to the destruction of an item. This trait has a single method, `drop`, which the compiler runs just before the memory holding the item is released:

```
#[derive(Debug)]
struct MyStruct(i32);

impl Drop for MyStruct {
    fn drop(&mut self) {
        println!("Dropping {self:?}");
        // Code to release resources owned by the item would go here.
    }
}
```

The `drop` method is specially reserved for the compiler and can't be manually invoked:

```
x.drop();
```

```
error[E0040]: explicit use of destructor method
  --> src/main.rs:70:7
   |
70 |     x.drop();
   |     --^^^^--
   |     | |
   |     | explicit destructor calls not allowed
   |     help: consider using `drop` function: `drop(x)`
```

It's worth understanding a little bit about the technical details here. Notice that the `Drop::drop` method has a signature of `drop(&mut self)` rather than `drop(self)`: it takes a mutable reference to the item rather than having the item moved into the method. If `Drop::drop` acted like a normal method, that would mean the item would still be available for use afterward—even though all of its internal state has been tidied up and resources released!

```
{
    // If calling `drop` were allowed...
    x.drop(); // (does not compile)

    // `x` would still be available afterwards.
    x.0 += 1;
}
// Also, what would happen when `x` goes out of scope?
```

The compiler suggested a straightforward alternative, which is to call the `drop()` function to manually drop an item. This function *does* take a moved argument, and the implementation of `drop(_item: T)` is just an empty body `{ }`—so the moved item is dropped when that scope's closing brace is reached.

Notice also that the signature of the `drop(&mut self)` method has no return type, which means that it has no way to signal failure. If an attempt to release resources can fail, then you should probably have a separate `release` method that returns a `Result`, so it's possible for users to detect this failure.

Regardless of the technical details, the `drop` method is nevertheless the key place for implementing RAII patterns; its implementation is the ideal place to release resources associated with an item.

---

[1] This also means that RAII as a technique is mostly available only in languages that have a predictable time of destruction, which rules out most garbage-collected languages (although Go's `defer` statement achieves some of the same ends).

[2] RAII is also still useful for memory management in low-level `unsafe` code, but that is (mostly) beyond the scope of this book.

Item 12: Understand the trade-offs between generics and trait objects

Item 2 described the use of traits to encapsulate behavior in the type system, as a collection of related methods, and observed that there are two ways to make use of traits: as *trait bounds* for *generics* or in *trait objects*. This Item explores the trade-offs between these two possibilities.

As a running example, consider a trait that covers functionality for displaying graphical objects:

```
#[derive(Debug, Copy, Clone)]
pub struct Point {
    x: i64,
    y: i64,
}

#[derive(Debug, Copy, Clone)]
pub struct Bounds {
    top_left: Point,
    bottom_right: Point,
}

/// Calculate the overlap between two rectangles, or `None` if there is no
/// overlap.
fn overlap(a: Bounds, b: Bounds) -> Option<Bounds> {
    // ...
}

/// Trait for objects that can be drawn graphically.
pub trait Draw {
    /// Return the bounding rectangle that encompasses the object.
    fn bounds(&self) -> Bounds;

    // ...
}
```

Generics

Rust's generics are roughly equivalent to C++'s templates: they allow the programmer to write code that works for some arbitrary type `T`, and specific uses of the generic code are generated at compile time—a process known as *monomorphization* in Rust, and *template instantiation* in C++. Unlike C++, Rust explicitly encodes the expectations for the type `T` in the type system, in the form of trait bounds for the generic.

For the example, a generic function that uses the trait's `bounds()` method has an explicit `Draw` trait bound:

```
/// Indicate whether an object is on-screen.
pub fn on_screen<T>(draw: &T) -> bool
where
    T: Draw,
{
    overlap(SCREEN_BOUNDS, draw.bounds()).is_some()
}
```

This can also be written more compactly by putting the trait bound after the generic parameter:

```
pub fn on_screen<T: Draw>(draw: &T) -> bool {
    overlap(SCREEN_BOUNDS, draw.bounds()).is_some()
}
```

or by using `impl Trait` as the type of the argument:[1]

```
pub fn on_screen(draw: &impl Draw) -> bool {
    overlap(SCREEN_BOUNDS, draw.bounds()).is_some()
}
```

If a type implements the trait:

```
#[derive(Clone)] // no `Debug`
struct Square {
    top_left: Point,
    size: i64,
}

impl Draw for Square {
    fn bounds(&self) -> Bounds {
        Bounds {
            top_left: self.top_left,
            bottom_right: Point {
                x: self.top_left.x + self.size,
                y: self.top_left.y + self.size,
            },
        }
    }
}
```

then instances of that type can be passed to the generic function, monomorphizing it to produce code that's specific to one particular type:

```
let square = Square {
    top_left: Point { x: 1, y: 2 },
    size: 2,
};
// Calls `on_screen::<Square>(&Square) -> bool`
let visible = on_screen(&square);
```

If the same generic function is used with a different type that implements the relevant trait bound:

```
#[derive(Clone, Debug)]
struct Circle {
    center: Point,
    radius: i64,
}

impl Draw for Circle {
    fn bounds(&self) -> Bounds {
        // ...
    }
}
```

then different monomorphized code is used:

```
let circle = Circle {
    center: Point { x: 3, y: 4 },
    radius: 1,
};
// Calls `on_screen::<Circle>(&Circle) -> bool`
let visible = on_screen(&circle);
```

In other words, the programmer writes a single generic function, but the compiler outputs a different monomorphized version of that function for every different type that the function is invoked with.

Trait Objects

In comparison, trait objects are fat pointers (Item 8) that combine a pointer to the underlying concrete item with a pointer to a vtable that in turn holds function pointers for all of the trait implementation's methods, as depicted in Figure 2-1:

```
let square = Square {
    top_left: Point { x: 1, y: 2 },
    size: 2,
};
let draw: &dyn Draw = &square;
```

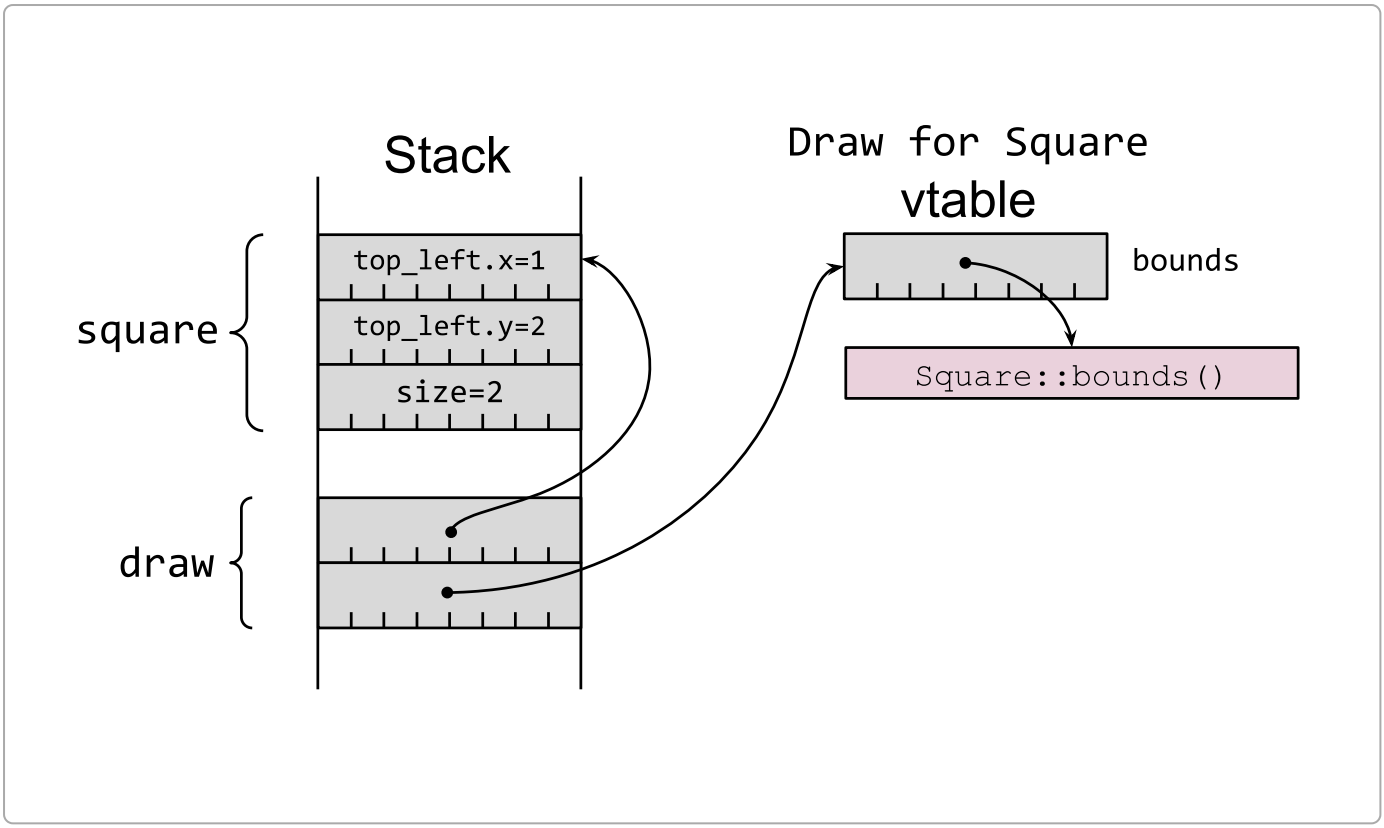


Figure 2-1. Trait object layout, with pointers to concrete item and vtable

This means that a function that accepts a trait object doesn't need to be generic and doesn't need monomorphization: the programmer writes a function using trait objects, and the compiler outputs only a single version of that function, which can accept trait objects that come from multiple input types:

```
/// Indicate whether an object is on-screen.
pub fn on_screen(draw: &dyn Draw) -> bool {
    overlap(SCREEN_BOUNDS, draw.bounds()).is_some()
}


// Calls `on_screen(&dyn Draw) -> bool`.
let visible = on_screen(&square);
// Also calls `on_screen(&dyn Draw) -> bool`.
let visible = on_screen(&circle);
```

Basic Comparisons

These basic facts already allow some immediate comparisons between the two possibilities:

- Generics are likely to lead to bigger code sizes, because the compiler generates a fresh copy (`on_screen::<T>(&T)`) of the code for every type `T` that uses the generic version of the `on_screen` function. In contrast, the trait object version (`on_screen(&dyn T)`) of the function needs only a single instance.

- Invoking a trait method from a generic will generally be ever-so-slightly faster than invoking it from code that uses a trait object, because the latter needs to perform two dereferences to find the location of the code (trait object to vtable, vtable to implementation location).
- Compile times for generics are likely to be longer, as the compiler is building more code and the linker has more work to do to fold duplicates.

In most situations, these aren't significant differences—you should use optimization-related concerns as a primary decision driver only if you've measured the impact and found that it has a genuine effect (a speed bottleneck or a problematic occupancy increase).

A more significant difference is that generic trait bounds can be used to conditionally make different functionality available, depending on whether the type parameter implements *multiple* traits:

```
// The `area` function is available for all containers holding things
// that implement `Draw`.
fn area<T>(draw: &T) -> i64
where
    T: Draw,
{
    let bounds = draw.bounds();
    (bounds.bottom_right.x - bounds.top_left.x)
        * (bounds.bottom_right.y - bounds.top_left.y)
}

// The `show` method is available only if `Debug` is also implemented.
fn show<T>(draw: &T)
where
    T: Debug + Draw,
{
    println!("{:?} has bounds {:?}", draw, draw.bounds());
}


let square = Square {
    top_left: Point { x: 1, y: 2 },
    size: 2,
};
let circle = Circle {
    center: Point { x: 3, y: 4 },
    radius: 1,
};

// Both `Square` and `Circle` implement `Draw`.
println!("area(square) = {}", area(&square));
println!("area(circle) = {}", area(&circle));

// `Circle` implements `Debug`.
show(&circle);

// `Square` does not implement `Debug`, so this wouldn't compile:
// show(&square);
```

A trait object encodes the implementation vtable only for a single trait, so doing something equivalent is much more awkward. For example, a combination `DebugDraw` trait could be defined for the `show()` case, together with a blanket implementation to make life easier:

```
trait DebugDraw: Debug + Draw {}

/// Blanket implementation applies whenever the individual traits
/// are implemented.
impl<T: Debug + Draw> DebugDraw for T {}
```

However, if there are multiple combinations of distinct traits, it's clear that the combinatorics of this approach rapidly become unwieldy.

More Trait Bounds

In addition to using trait bounds to restrict what type parameters are acceptable for a generic function, you can also apply them to trait definitions themselves:

```
/// Anything that implements `Shape` must also implement `Draw`.
trait Shape: Draw {
    /// Render that portion of the shape that falls within `bounds`.
    fn render_in(&self, bounds: Bounds);

    /// Render the shape.
    fn render(&self) {
        // Default implementation renders that portion of the shape
        // that falls within the screen area.
        if let Some(visible) = overlap(SCREEN_BOUNDS, self.bounds()) {
            self.render_in(visible);
        }
    }
}
```

In this example, the `render()` method's default implementation (Item 13) makes use of the trait bound, relying on the availability of the `bounds()` method from `Draw`.

Programmers coming from object-oriented languages often confuse trait bounds with inheritance, under the mistaken impression that a trait bound like this means that a `Shape` *is-a* `Draw`. That's not the case: the relationship between the two types is better expressed as `Shape` *also-implements* `Draw`.

Under the covers, trait objects for traits that have trait bounds:

```
let square = Square {
    top_left: Point { x: 1, y: 2 },
    size: 2,
};
let draw: &dyn Draw = &square;
let shape: &dyn Shape = &square;
```

have a single combined vtable that includes the methods of the top-level trait, plus the methods of all of the trait bounds. This is shown in Figure 2-2: the vtable for `Shape` includes the `bounds` method from the `Draw` trait, as well as the two methods from the `Shape` trait itself.

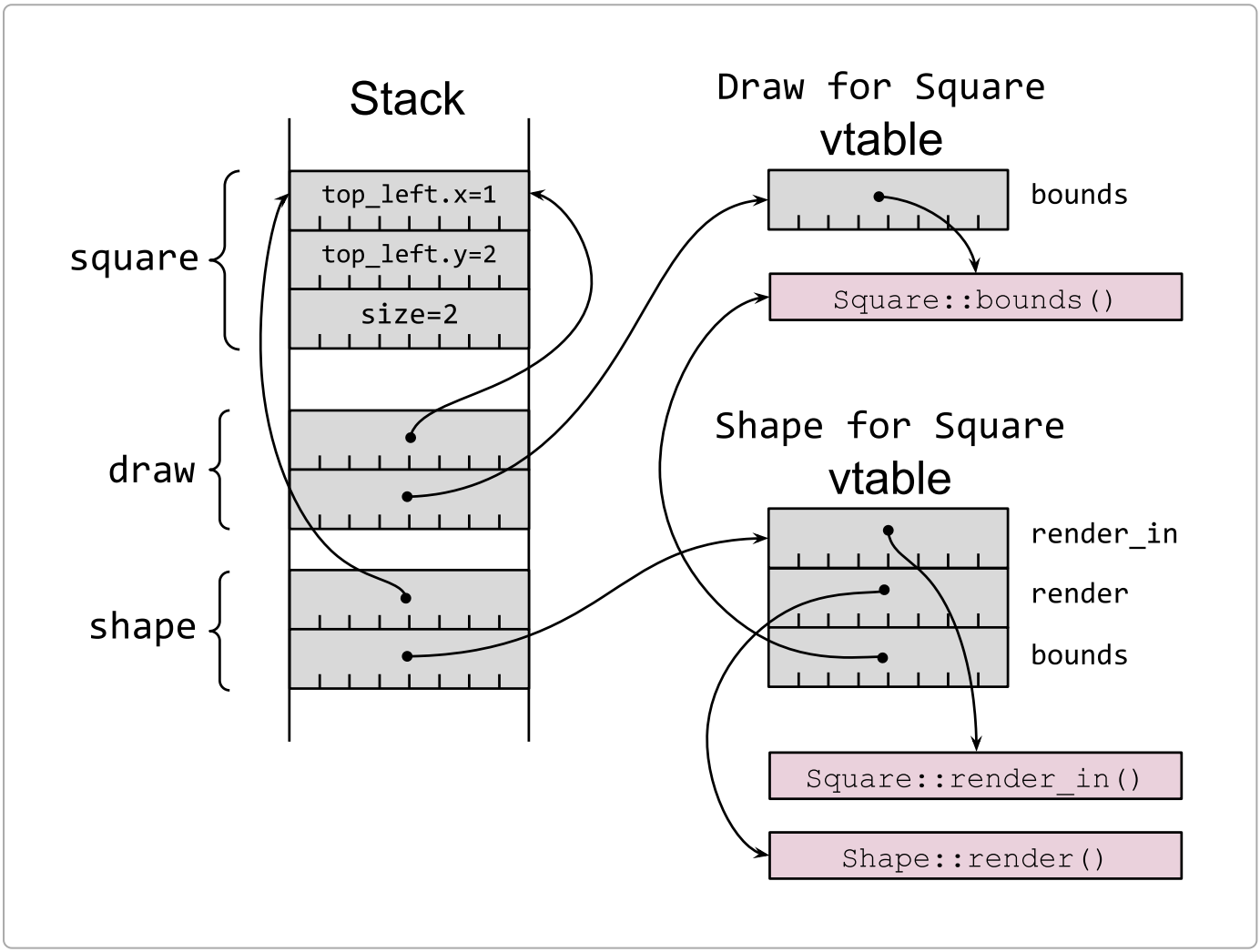


Figure 2-2. Trait objects for trait bounds, with distinct vtables for `Draw` and `Shape`

At the time of writing (and as of Rust 1.70), this means that there is no way to "upcast" from `Shape` to `Draw`, because the (pure) `Draw` vtable can't be recovered at runtime; there is no way to convert between related trait objects, which in turn means there is no Liskov substitution. However, this is likely to change in later versions of Rust—see Item 19 for more on this.

Repeating the same point in different words, a method that accepts a `Shape` trait object has the following characteristics:

- It *can* make use of methods from `Draw` (because `Shape` also-implements `Draw`, and because the relevant function pointers are present in the `Shape` vtable).
- It *cannot* (yet) pass the trait object onto another method that expects a `Draw` trait object (because `Shape` is-not `Draw`, and because the `Draw` vtable isn't available).

In contrast, a generic method that accepts items that implement `Shape` has these characteristics:

- It *can* use methods from `Draw`.
- It *can* pass the item on to another generic method that has a `Draw` trait bound, because the trait bound is monomorphized at compile time to use the `Draw` methods of the concrete type.

Trait Object Safety

Another restriction on trait objects is the requirement for *object safety*: only traits that comply with the following two rules can be used as trait objects:

- The trait's methods must not be generic.
- The trait's methods must not involve a type that includes `Self`, except for the receiver (the object on which the method is invoked).

The first restriction is easy to understand: a generic method `f` is really an infinite set of methods, potentially encompassing `f::<i16>`, `f::<i32>`, `f::<i64>`, `f::<u8>`, etc. The trait object's vtable, on the other hand, is very much a finite collection of function pointers, and so it's not possible to fit the infinite set of monomorphized implementations into it.

The second restriction is a little bit more subtle but tends to be the restriction that's hit more often in practice—traits that impose `Copy` or `Clone` trait bounds (Item 10) immediately fall under this rule, because they return `Self`. To see why it's disallowed, consider code that has a trait object in its hands; what happens if that code calls (say) `let y = x.clone()`? The calling code needs to reserve enough space for `y` on the stack, but it has no idea of the size of `y` because `Self` is an arbitrary type. As a result, return types that mention `Self` lead to a trait that is not object safe.[2]

There is an exception to this second restriction. A method returning some `Self`-related type does not affect object safety if `Self` comes with an explicit restriction to types whose size is known at compile time, indicated by the `Sized` marker trait as a trait bound:

```
/// A `Stamp` can be copied and drawn multiple times.
trait Stamp: Draw {
    fn make_copy(&self) -> Self
    where
        Self: Sized;
}


let square = Square {
    top_left: Point { x: 1, y: 2 },
    size: 2,
};

// `Square` implements `Stamp`, so it can call `make_copy()`.
let copy = square.make_copy();

// Because the `Self`-returning method has a `Sized` trait bound,
// creating a `Stamp` trait object is possible.
let stamp: &dyn Stamp = &square;
```

This trait bound means that the method can't be used with trait objects anyway, because trait objects refer to something that's of unknown size (`dyn Trait`), and so the method is irrelevant for object safety:

```
// However, the method can't be invoked via a trait object.
let copy = stamp.make_copy();


error: the `make_copy` method cannot be invoked on a trait object
   --> src/main.rs:397:22
    |
353 |         Self: Sized;
    |               ----- this has a `Sized` requirement
...
397 |     let copy = stamp.make_copy();
    |                      ^^^^^^^^^
```

Trade-Offs

The balance of factors so far suggests that you should **prefer generics to trait objects**, but there are situations where trait objects are the right tool for the job.

The first is a practical consideration: if generated code size or compilation time is a concern, then trait objects will perform better (as described earlier in this Item).

A more theoretical aspect that leads toward trait objects is that they fundamentally involve *type erasure*: information about the concrete type is lost in the conversion to a trait object. This can be a downside (see Item 19), but it can also be useful because it allows for

collections of heterogeneous objects—because the code *just* relies on the methods of the trait, it can invoke and combine the methods of items that have different concrete types.

The traditional OO example of rendering a list of shapes is one example of this: the same `render()` method could be used for squares, circles, ellipses, and stars in the same loop:

```
let shapes: Vec<&dyn Shape> = vec![&square, &circle];
for shape in shapes {
    shape.render()
}
```

A much more obscure potential advantage for trait objects is when the available types are not known at compile time. If new code is dynamically loaded at runtime (e.g., via `dlopen(3)`), then items that implement traits in the new code can be invoked only via a trait object, because there's no source code to monomorphize over.

---

[1] Using "`impl Trait` in argument position" isn't exactly equivalent to the previous two versions, because it removes the ability for a caller to explicitly specify the type parameter with something like `on_screen::<Circle>(&c)`.

[2] At the time of writing, the restriction on methods that return `Self` includes types like `Box<Self>` that *could* be safely stored on the stack; this restriction might be relaxed in the future.

Item 13: Use default implementations to minimize required trait methods

The designer of a trait has two different audiences to consider: the programmers who will be *implementing* the trait and those who will be *using* the trait. These two audiences lead to a degree of tension in the trait design:

- To make the implementor's life easier, it's better for a trait to have the absolute minimum number of methods to achieve its purpose.
- To make the user's life more convenient, it's helpful to provide a range of variant methods that cover all of the common ways that the trait might be used.

This tension can be balanced by including the wider range of methods that makes the user's life easier, but with *default implementations* provided for any methods that can be built from other, more primitive, operations on the interface.

A simple example of this is the `is_empty()` method for an `ExactSizeIterator`, which is an `Iterator` that knows how many things it is iterating over.[1] This method has a default implementation that relies on the `len()` trait method:

```
fn is_empty(&self) -> bool {
    self.len() == 0
}
```

The existence of a default implementation is just that: a default. If an implementation of the trait has a different way of determining whether the iterator is empty, it can replace the default `is_empty()` with its own.

This approach leads to trait definitions that have a small number of *required methods*, plus a much larger number of default-implemented methods. An implementor for the trait has to implement only the former and gets all of the latter for free.

It's also an approach that is widely followed by the Rust standard library; perhaps the best example there is the `Iterator` trait, which has a single required method (`next()`) but includes a panoply of pre-provided methods (Item 9), over 50 at the time of writing.

Trait methods can impose *trait bounds*, indicating that a method is only available if the types involved implement particular traits. The `Iterator` trait also shows that this is useful in combination with default method implementations. For example, the `cloned()` iterator method has a trait bound and a default implementation:

```
fn cloned<'a, T>(self) -> Cloned<Self>
where
    T: 'a + Clone,
    Self: Sized + Iterator<Item = &'a T>,
{
    Cloned::new(self)
}
```

In other words, the `cloned()` method is available only if the underlying `Item` type implements `Clone`; when it does, the implementation is automatically available.

The final observation about trait methods with default implementations is that new ones can *usually* be safely added to a trait even after an initial version of the trait is released. An addition like this preserves backward compatibility (see Item 21) for users and implementors of the trait, as long as the new method name does not clash with the name of a method from some other trait that the type implements. [2]

So follow the example of the standard library and provide a minimal API surface for implementors but a convenient and comprehensive API for users, by adding methods with default implementations (and trait bounds as appropriate).

---

[1] The `is_empty()` method is currently a nightly-only experimental function.

[2] If the new method happens to match a method of the same name in the *concrete* type, then the concrete method—known as an *inherent implementation*—will be used ahead of the trait method. The trait method can be explicitly selected instead by casting: `<Concrete as Trait>::method()`.

Concepts

The first two chapters of this book covered Rust's types and traits, which helps provide the vocabulary needed to work with some of the *concepts* involved in writing Rust code—the subject of this chapter.

The borrow checker and lifetime checks are central to what makes Rust unique; they are also a common stumbling block for newcomers to Rust and so are the focus of the first two Items in this chapter.

The other Items in this chapter cover concepts that are easier to grasp but are nevertheless a bit different from writing code in other languages. This includes the following:

- Advice on Rust's `unsafe` mode and how to avoid it (Item 16)
- Good news and bad news about writing multithreaded code in Rust (Item 17)
- Advice on avoiding runtime aborts (Item 18)
- Information about Rust's approach to reflection (Item 19)
- Advice on balancing optimization against maintainability (Item 20)

It's a good idea to try to align your code with the consequences of these concepts. It's *possible* to re-create (some of) the behavior of C/C++ in Rust, but why bother to use Rust if you do?

Item 14: Understand lifetimes

This Item describes Rust's *lifetimes*, which are a more precise formulation of a concept that existed in previous compiled languages like C and C++—in practice if not in theory. Lifetimes are a required input for the *borrow checker* described in Item 15; taken together, these features form the heart of Rust's memory safety guarantees.

Introduction to the Stack

Lifetimes are fundamentally related to the *stack*, so a quick introduction/reminder is in order.

While a program is running, the memory that it uses is divided up into different chunks, sometimes called *segments*. Some of these chunks are a fixed size, such as the ones that hold the program code or the program's global data, but two of the chunks—the *heap* and the *stack*—change size as the program runs. To allow for this, they are typically arranged at opposite ends of the program's virtual memory space, so one can grow downward and the other can grow upward (at least until your program runs out of memory and crashes), as summarized in Figure 3-1.

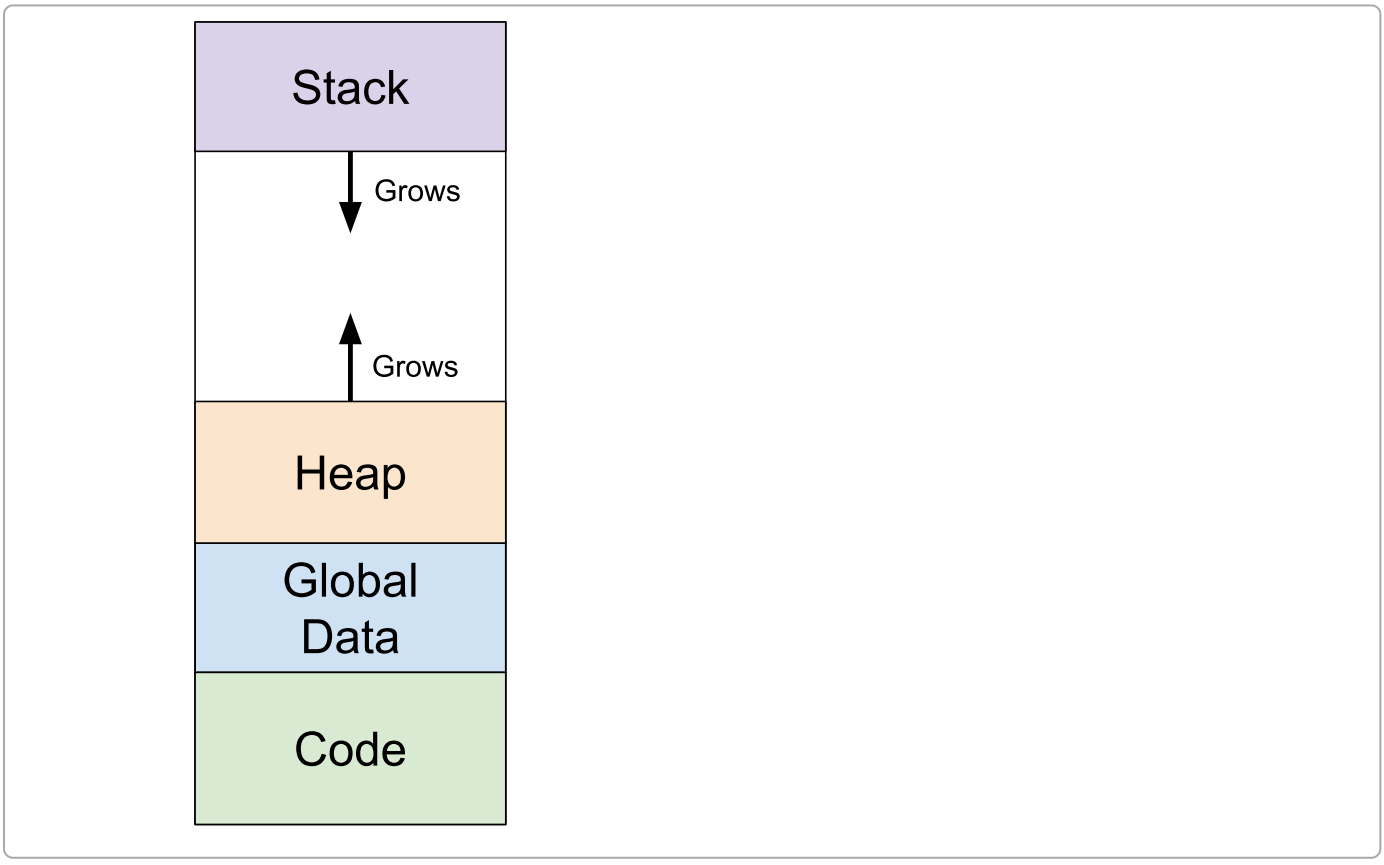


Figure 3-1. Program memory layout, including heap growing up and stack growing down

Of these two dynamically sized chunks, the stack is used to hold state related to the currently executing function. This state can include these elements:

- The parameters passed to the function
- The local variables used in the function
- Temporary values calculated within the function
- The return address within the code of the function's caller

When a function `f()` is called, a new stack frame is added to the stack, beyond where the stack frame for the calling function ends, and the CPU normally updates a register—the *stack pointer*—to point to the new stack frame.

When the inner function `f()` returns, the stack pointer is reset to where it was before the call, which will be the caller's stack frame, intact and unmodified.

If the caller subsequently invokes a different function `g()`, the process happens again, which means that the stack frame for `g()` will reuse the same area of memory that `f()` previously used (as depicted in Figure 3-2):

```
fn caller() -> u64 {
    let x = 42u64;
    let y = 19u64;
    f(x) + g(y)
}

fn f(f_param: u64) -> u64 {
    let two = 2u64;
    f_param + two
}

fn g(g_param: u64) -> u64 {
    let arr = [2u64, 3u64];
    g_param + arr[1]
}
```

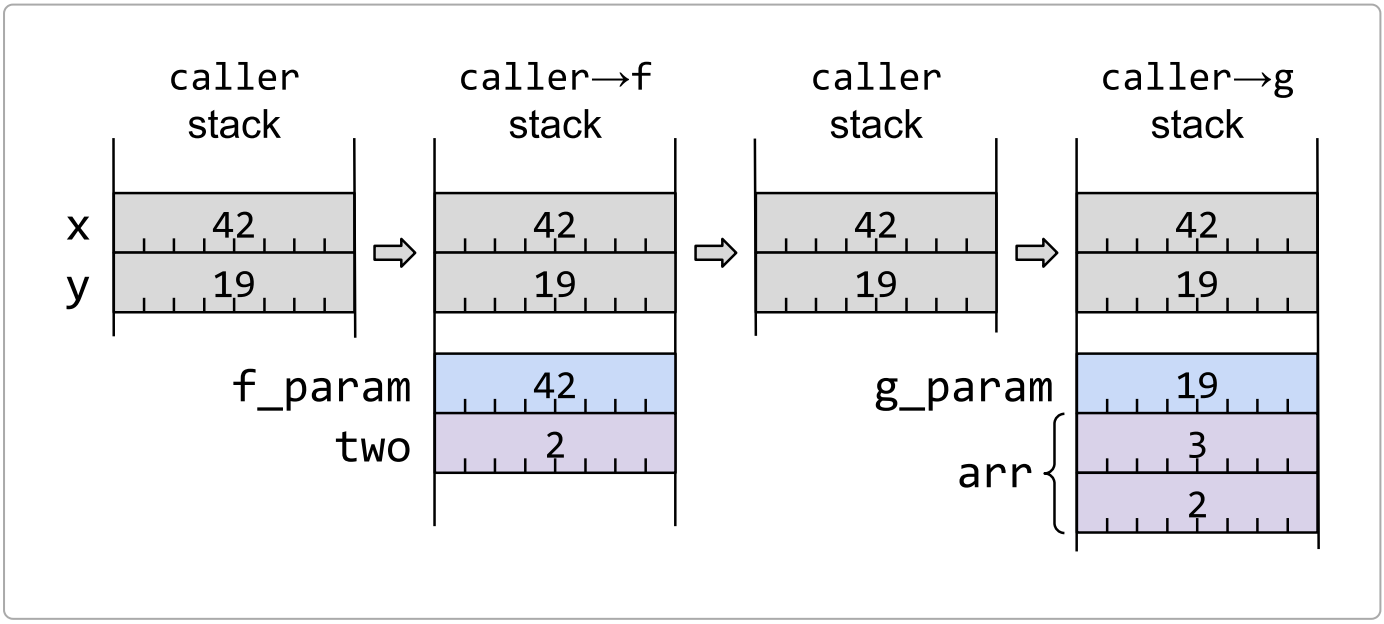


Figure 3-2. Evolution of stack usage as functions are called and returned from

Of course, this is a dramatically simplified version of what really goes on—putting things on and off the stack takes time, and so real processors will have many optimizations. However, the simplified conceptual picture is enough for understanding the subject of this Item.

Evolution of Lifetimes

The previous section explained how parameters and local variables are stored on the stack and pointed out that those values are stored only ephemerally.

Historically, this allowed for some dangerous footguns: what happens if you hold onto a pointer to one of these ephemeral stack values?

Starting back with C, it was perfectly OK to return a pointer to a local variable (although modern compilers will emit a warning for it):

```
/* C code. */
struct File {
  int fd;
};

struct File* open_bugged() {
  struct File f = { open("README.md", O_RDONLY) };
  return &f;  /* return address of stack object! */
}
```

You *might* get away with this, if you're unlucky and the calling code uses the returned value immediately:

```
struct File* f = open_bugged();
printf("in caller: file at %p has fd=%d\n", f, f->fd);
```

```
in caller: file at 0x7ff7bc019408 has fd=3
```

This is unlucky because it only *appears* to work. As soon as any other function calls happen, the stack area will be reused and the memory that used to hold the object will be overwritten:

```
investigate_file(f);
```

```
/* C code. */
void investigate_file(struct File* f) {
  long array[4] = {1, 2, 3, 4}; // put things on the stack
  printf("in function: file at %p has fd=%d\n", f, f->fd);
}
```

```
in function: file at 0x7ff7bc019408 has fd=1592262883
```

Trashing the contents of the object has an additional bad effect for this example: the file descriptor corresponding to the open file is lost, and so the program leaks the resource that was held in the data structure.

Moving forward in time to C++, this latter problem of losing access to resources was solved by the inclusion of *destructors*, enabling RAII (see Item 11). Now, the things on the stack have the ability to tidy themselves up: if the object holds some kind of resource, the destructor can tidy it up, and the C++ compiler guarantees that the destructor of an object on the stack gets called as part of tidying up the stack frame:

```
// C++ code.
File::~File() {
  std::cout << "~File(): close fd " << fd << "\n";
  close(fd);
  fd = -1;
}
```

The caller now gets an (invalid) pointer to an object that's been destroyed and its resources reclaimed:

```
File* f = open_bugged();
printf("in caller: file at %p has fd=%d\n", f, f->fd);
```

```
~File(): close fd 3
in caller: file at 0x7ff7b6a7c438 has fd=-1
```

However, C++ did nothing to help with the problem of dangling pointers: it's still possible to hold onto a pointer to an object that's gone (with a destructor that has been called):

```
// C++ code.
void investigate_file(File* f) {
  long array[4] = {1, 2, 3, 4}; // put things on the stack
  std::cout << "in function: file at " << f << " has fd=" << f->fd << "\n";
}
```

```
in function: file at 0x7ff7b6a7c438 has fd=-183042004
```

As a C/C++ programmer, it's up to you to notice this and make sure that you don't dereference a pointer that points to something that's gone. Alternatively, if you're an attacker and you find one of these dangling pointers, you're more likely to cackle maniacally and gleefully dereference the pointer on your way to an exploit.

Enter Rust. One of Rust's core attractions is that it fundamentally solves the problem of dangling pointers, immediately solving a large fraction of security problems. [1]

Doing so requires moving the concept of lifetimes from the background (where C/C++ programmers just have to know to watch out for them, without any language support) to the foreground: every type that includes an ampersand `&` has an associated lifetime (`'a`), even if the compiler lets you omit mention of it much of the time.

Scope of a Lifetime

The lifetime of an item on the stack is the period where that item is guaranteed to stay in the same place; in other words, this is exactly the period where a *reference* (pointer) to the item is guaranteed not to become invalid.

This starts at the point where the item is created, and extends to where it is either *dropped* (Rust's equivalent to object destruction in C++) or *moved*.

The ubiquity of the latter is sometimes surprising for programmers coming from C/C++: Rust moves items from one place on the stack to another, or from the stack to the heap, or from the heap to the stack, in lots of situations.

Precisely where an item gets automatically dropped depends on whether an item has a name or not.

Local variables and function parameters have names, and the corresponding item's lifetime starts when the item is created and the name is populated:

- For a local variable: at the `let var = ...` declaration
- For a function parameter: as part of setting up the execution frame for the function invocation

The lifetime for a named item ends when the item is either moved somewhere else or when the name goes out of scope:

```
#[derive(Debug, Clone)]
/// Definition of an item of some kind.
pub struct Item {
    contents: u32,
}


{
    let item1 = Item { contents: 1 }; // `item1` created here
    let item2 = Item { contents: 2 }; // `item2` created here
    println!("item1 = {item1:?}, item2 = {item2:?}");
    consuming_fn(item2); // `item2` moved here
} // `item1` dropped here
```

It's also possible to build an item "on the fly", as part of an expression that's then fed into something else. These unnamed temporary items are then dropped when they're no longer

needed. One oversimplified but helpful way to think about this is to imagine that each part of the expression gets expanded to its own block, with temporary variables being inserted by the compiler. For example, an expression like:

```
let x = f((a + b) * 2);
```

would be roughly equivalent to:

```
let x = {
    let temp1 = a + b;
    {
        let temp2 = temp1 * 2;
        f(temp2)
    } // `temp2` dropped here
}; // `temp1` dropped here
```

By the time execution reaches the semicolon at the end of the original line, the temporaries have all been dropped.

One way to see what the compiler calculates as an item's lifetime is to insert a deliberate error for the borrow checker (Item 15) to detect. For example, hold onto a reference to an item beyond the scope of the item's lifetime:

```
let r: &Item;
{
    let item = Item { contents: 42 };
    r = &item;
}
println!("r.contents = {}", r.contents);
```

The error message indicates the exact endpoint of `item`'s lifetime:

```
error[E0597]: `item` does not live long enough
   --> src/main.rs:190:13
    |
189 |         let item = Item { contents: 42 };
    |             ---- binding `item` declared here
190 |         r = &item;
    |             ^^^^^ borrowed value does not live long enough
191 |     }
    |     - `item` dropped here while still borrowed
192 |     println!("r.contents = {}", r.contents);
    |                                 ---------- borrow later used here
```

Similarly, for an unnamed temporary:

```
let r: &Item = fn_returning_ref(&mut Item { contents: 42 });
println!("r.contents = {}", r.contents);
```

the error message shows the endpoint at the end of the expression:

```
error[E0716]: temporary value dropped while borrowed
   --> src/main.rs:209:46
    |
209 | let r: &Item = fn_returning_ref(&mut Item { contents: 42 });
    |                                 ^^^^^^^^^^^^^^^^^^^^^^^^^^ - temporary
    |                                 |                          value is freed at the
    |                                 |                          end of this statement
    |                                 |
    |                                 creates a temporary value which is
    |                                 freed while still in use
210 | println!("r.contents = {}", r.contents);
    |                             ---------- borrow later used here
    |
    = note: consider using a `let` binding to create a longer lived value
```

One final point about the lifetimes of *references*: if the compiler can prove to itself that there is no use of a reference beyond a certain point in the code, then it treats the endpoint of the reference's lifetime as the last place it's used, rather than at the end of the enclosing scope. This feature, known as non-lexical lifetimes, allows the borrow checker to be a little bit more generous:

```
{
    // `s` owns the `String`.
    let mut s: String = "Hello, world".to_string();

    // Create a mutable reference to the `String`.
    let greeting = &mut s[..5];
    greeting.make_ascii_uppercase();
    // .. no use of `greeting` after this point

    // Creating an immutable reference to the `String` is allowed,
    // even though there's a mutable reference still in scope.
    let r: &str = &s;
    println!("s = '{}'", r); // s = 'HELLO, world'
} // The mutable reference `greeting` would naively be dropped here.
```

Algebra of Lifetimes

Although lifetimes are ubiquitous when dealing with references in Rust, you don't get to specify them in any detail—there's no way to say, "I'm dealing with a lifetime that extends from line 17 to line 32 of `ref.rs`". Instead, your code refers to lifetimes with arbitrary names, conventionally `'a`, `'b`, `'c`, ..., and the compiler has its own internal, inaccessible representation of what that equates to in the source code. (The one exception to this is the `'static` lifetime, which is a special case that's covered in a subsequent section.)

You don't get to do much with these lifetime names; the main thing that's possible is to compare one name with another, repeating a name to indicate that two lifetimes are the "same".

This algebra of lifetimes is easiest to illustrate with function signatures: if the inputs and outputs of a function deal with references, what's the relationship between their lifetimes?

The most common case is a function that receives a single reference as input and emits a reference as output. The output reference must have a lifetime, but what can it be? There's only one possibility (other than `'static`) to choose from: the lifetime of the input, which means that they both share the same name, say, `'a`. Adding that name as a lifetime annotation to both types gives:

```
pub fn first<'a>(data: &'a [Item]) -> Option<&'a Item> {
    // ...
}
```

Because this variant is so common, and because there's (almost) no choice about what the output lifetime can be, Rust has *lifetime elision* rules that mean you don't have to explicitly write the lifetime names for this case. A more idiomatic version of the same function signature would be the following:

```
pub fn first(data: &[Item]) -> Option<&Item> {
    // ...
}
```

The references involved still have lifetimes—the elision rule just means that you don't have to make up an arbitrary lifetime name and use it in both places.

What if there's more than one choice of input lifetimes to map to an output lifetime? In this case, the compiler can't figure out what to do:

```
pub fn find(haystack: &[u8], needle: &[u8]) -> Option<&[u8]> {
    // ...
}
```

```
error[E0106]: missing lifetime specifier
   --> src/main.rs:56:55
    |
56 | pub fn find(haystack: &[u8], needle: &[u8]) -> Option<&[u8]> {
    |                       -----          -----            ^ expected named
    |                                                     lifetime parameter
    |
    = help: this function's return type contains a borrowed value, but the
            signature does not say whether it is borrowed from `haystack` or
            `needle`
help: consider introducing a named lifetime parameter
    |
56 | pub fn find<'a>(haystack: &'a [u8], needle: &'a [u8]) -> Option<&'a [u8]>
{
    |            ++++            ++                 ++                   ++
```

A shrewd guess based on the function and parameter names is that the intended lifetime for the output here is expected to match the `haystack` input:

```
pub fn find<'a, 'b>(
    haystack: &'a [u8],
    needle: &'b [u8],
) -> Option<&'a [u8]> {
    // ...
}
```

Interestingly, the compiler suggested a different alternative: having both inputs to the function use the *same* lifetime `'a`. For example, the following is a function where this combination of lifetimes might make sense:

```
pub fn smaller<'a>(left: &'a Item, right: &'a Item) -> &'a Item {
    // ...
}
```

This *appears* to imply that the two input lifetimes are the "same", but the scare quotes (here and previously) are included to signify that that's not quite what's going on.

The raison d'être of lifetimes is to ensure that references to items don't outlive the items themselves; with this in mind, an output lifetime `'a` that's the "same" as an input lifetime `'a` just means that the input has to live longer than the output.

When there are two input lifetimes `'a` that are the "same", that just means that the output lifetime has to be contained within the lifetimes of *both* of the inputs:

```
{
    let outer = Item { contents: 7 };
    {
        let inner = Item { contents: 8 };
        {
            let min = smaller(&inner, &outer);
            println!("smaller of {inner:?} and {outer:?} is {min:?}");
        } // `min` dropped
    } // `inner` dropped
} // `outer` dropped
```

To put it another way, the output lifetime has to be subsumed within the *smaller* of the lifetimes of the two inputs.

In contrast, if the output lifetime is unrelated to the lifetime of one of the inputs, then there's no requirement for those lifetimes to nest:

```
{
    let haystack = b"123456789"; // start of  lifetime 'a
    let found = {
        let needle = b"234"; // start of lifetime 'b
        find(haystack, needle)
    }; // end of lifetime 'b
    println!("found={:?}", found); // `found` used within 'a, outside of 'b
} // end of lifetime 'a
```

Lifetime Elision Rules

In addition to the "one in, one out" elision rule previously described, there are two other elision rules that mean that lifetime names can be omitted.

The first occurs when there are no references in the outputs from a function; in this case, each of the input references automatically gets its own lifetime, different from any of the other input parameters.

The second occurs for methods that use a reference to `self` (either `&self` or `&mut self`); in this case, the compiler assumes that any output references take the lifetime of `self`, as this turns out to be (by far) the most common situation.

Here's a summary of the elision rules for functions:

- One input, one or more outputs: assume outputs have the "same" lifetime as the input:

```
fn f(x: &Item) -> (&Item, &Item)
// ... is equivalent to ...
fn f<'a>(x: &'a Item) -> (&'a Item, &'a Item)
```

- Multiple inputs, no output: assume all the inputs have different lifetimes:

```
fn f(x: &Item, y: &Item, z: &Item) -> i32
// ... is equivalent to ...
fn f<'a, 'b, 'c>(x: &'a Item, y: &'b Item, z: &'c Item) -> i32
```

- Multiple inputs including `&self`, one or more outputs: assume output lifetime(s) are the "same" as `&self`'s lifetime:

```
fn f(&self, y: &Item, z: &Item) -> &Thing
// ... is equivalent to ...
fn f(&'a self, y: &'b Item, z: &'c Item) -> &'a Thing
```

Of course, if the elided lifetime names don't match what you want, you can always explicitly write lifetime names that specify which lifetimes are related to each other. In practice, this is likely to be triggered by a compiler error that indicates that the elided lifetimes don't match how the function or its caller are using the references involved.

The `'static` Lifetime

The previous section described various possible mappings between the input and output reference lifetimes for a function, but it neglected to cover one special case. What happens if there are *no* input lifetimes, but the output return value includes a reference anyway?

```
pub fn the_answer() -> &Item {
    // ...
}
```

```
error[E0106]: missing lifetime specifier
   --> src/main.rs:471:28
    |
471 |     pub fn the_answer() -> &Item {
    |                            ^ expected named lifetime parameter
    |
    = help: this function's return type contains a borrowed value, but there
            is no value for it to be borrowed from
help: consider using the `'static` lifetime
    |
471 |     pub fn the_answer() -> &'static Item {
    |                             +++++++
```

The only allowed possibility is for the returned reference to have a lifetime that's guaranteed to never go out of scope. This is indicated by the special lifetime `'static`, which is also the only lifetime that has a specific name rather than an arbitrary placeholder name:

```
pub fn the_answer() -> &'static Item {
```

The simplest way to get something with the `'static` lifetime is to take a reference to a global variable that's been marked as `static`:

```
static ANSWER: Item = Item { contents: 42 };

pub fn the_answer() -> &'static Item {
    &ANSWER
}
```

The Rust compiler guarantees that a `static` item always has the same address for the entire duration of the program and never moves. This means that a reference to a `static` item has a `'static` lifetime, logically enough.

In many cases, a reference to a `const` item will also be promoted to have a `'static` lifetime, but there are a couple of minor complications to be aware of. The first is that this promotion doesn't happen if the type involved has a destructor or interior mutability:

```
pub struct Wrapper(pub i32);

impl Drop for Wrapper {
    fn drop(&mut self) {}
}

const ANSWER: Wrapper = Wrapper(42);

pub fn the_answer() -> &'static Wrapper {
    // `Wrapper` has a destructor, so the promotion to the `'static`
    // lifetime for a reference to a constant does not apply.
    &ANSWER
}
```

```
error[E0515]: cannot return reference to temporary value
   --> src/main.rs:520:9
    |
520 |         &ANSWER
    |         ^------
    |         ||
    |         |temporary value created here
    |         returns a reference to data owned by the current function
```

The second potential complication is that only the *value* of a `const` is guaranteed to be the same everywhere; the compiler is allowed to make as many copies as it likes, wherever the variable is used. If you're doing nefarious things that rely on the underlying pointer value behind the `'static` reference, be aware that multiple memory locations may be involved.

There's one more possible way to get something with a `'static` lifetime. The key promise of `'static` is that the lifetime should outlive any other lifetime in the program; a value that's allocated on the heap but *never freed* also satisfies this constraint.

A normal heap-allocated `Box<T>` doesn't work for this, because there's no guarantee (as described in the next section) that the item won't get dropped along the way:

```
{
    let boxed = Box::new(Item { contents: 12 });
    let r: &'static Item = &boxed;
    println!("'static item is {:?}", r);
}
```

```
error[E0597]: `boxed` does not live long enough
   --> src/main.rs:344:32
    |
343 |     let boxed = Box::new(Item { contents: 12 });
    |         ----- binding `boxed` declared here
344 |     let r: &'static Item = &boxed;
    |            -------------   ^^^^^^ borrowed value does not live long
enough
    |            |
    |            type annotation requires that `boxed` is borrowed for
`'static`
345 |     println!("'static item is {:?}", r);
346 | }
    | - `boxed` dropped here while still borrowed
```

However, the `Box::leak` function converts an owned `Box<T>` to a mutable reference to `T`. There's no longer an owner for the value, so it can never be dropped—which satisfies the requirements for the `'static` lifetime:

```
{
    let boxed = Box::new(Item { contents: 12 });

    // `leak()` consumes the `Box<T>` and returns `&mut T`.
    let r: &'static Item = Box::leak(boxed);

    println!("'static item is {:?}", r);
} // `boxed` not dropped here, as it was already moved into `Box::leak()`

// Because `r` is now out of scope, the `Item` is leaked forever.
```

The inability to drop the item also means that the memory that holds the item can never be reclaimed using safe Rust, possibly leading to a permanent memory leak. (Note that leaking memory doesn't violate Rust's memory safety guarantees—an item in memory that you can no longer access is still safe.)

Lifetimes and the Heap

The discussion so far has concentrated on the lifetimes of items on the stack, whether function parameters, local variables, or temporaries. But what about items on the heap?

The key thing to realize about heap values is that every item has an owner (excepting special cases like the deliberate leaks described in the previous section). For example, a simple `Box<T>` puts the `T` value on the heap, with the owner being the variable holding the `Box<T>`:

```
{
    let b: Box<Item> = Box::new(Item { contents: 42 });
} // `b` dropped here, so `Item` dropped too.
```

The owning `Box<Item>` drops its contents when it goes out of scope, so the lifetime of the `Item` on the heap is the same as the lifetime of the `Box<Item>` variable on the stack.

The owner of a value on the heap may itself be on the heap rather than the stack, but then who owns the owner?

```
{
    let b: Box<Item> = Box::new(Item { contents: 42 });
    let bb: Box<Box<Item>> = Box::new(b); // `b` moved onto heap here
} // `bb` dropped here, so `Box<Item>` dropped too, so `Item` dropped too.
```

The chain of ownership has to end somewhere, and there are only two possibilities:

- The chain ends at a local variable or function parameter—in which case the lifetime of everything in the chain is just the lifetime `'a` of that stack variable. When the stack variable goes out of scope, everything in the chain is dropped too.
- The chain ends at a global variable marked as `static`—in which case the lifetime of everything in the chain is `'static`. The `static` variable never goes out of scope, so nothing in the chain ever gets automatically dropped.

As a result, the lifetimes of items on the heap are fundamentally tied to stack lifetimes.

Lifetimes in Data Structures

The earlier section on the algebra of lifetimes concentrated on inputs and outputs for functions, but there are similar concerns when references are stored in data structures.

If we try to sneak a reference into a data structure without mentioning an associated lifetime, the compiler brings us up sharply:

```
pub struct ReferenceHolder {
    pub index: usize,
    pub item: &Item,
}
```

```
error[E0106]: missing lifetime specifier
   --> src/main.rs:548:19
    |
548 |         pub item: &Item,
    |                   ^ expected named lifetime parameter
    |
help: consider introducing a named lifetime parameter
    |
546 ~     pub struct ReferenceHolder<'a> {
547 |         pub index: usize,
548 ~         pub item: &'a Item,
    |
```

As usual, the compiler error message tells us what to do. The first part is simple enough: give the reference type an explicit lifetime name `'a`, because there are no lifetime elision rules when using references in data structures.

The second part is less obvious and has deeper consequences: the data structure itself has to have a lifetime parameter `<'a>` that matches the lifetime of the reference contained within it:

```
// Lifetime parameter required due to field with reference.
pub struct ReferenceHolder<'a> {
    pub index: usize,
    pub item: &'a Item,
}
```

The lifetime parameter for the data structure is infectious: any containing data structure that uses the type also has to acquire a lifetime parameter:

```
// Lifetime parameter required due to field that is of a
// type that has a lifetime parameter.
pub struct RefHolderHolder<'a> {
    pub inner: ReferenceHolder<'a>,
}
```

The need for a lifetime parameter also applies if the data structure contains slice types, as these are again references to borrowed data.

If a data structure contains multiple fields that have associated lifetimes, then you have to choose what combination of lifetimes is appropriate. An example that finds common substrings within a pair of strings is a good candidate to have independent lifetimes:

```
/// Locations of a substring that is present in
/// both of a pair of strings.
pub struct LargestCommonSubstring<'a, 'b> {
    pub left: &'a str,
    pub right: &'b str,
}

/// Find the largest substring present in both `left`
/// and `right`.
pub fn find_common<'a, 'b>(
    left: &'a str,
    right: &'b str,
) -> Option<LargestCommonSubstring<'a, 'b>> {
    // ...
}
```

whereas a data structure that references multiple places within the same string would have a common lifetime:

```
/// First two instances of a substring that is repeated
/// within a string.
pub struct RepeatedSubstring<'a> {
    pub first: &'a str,
    pub second: &'a str,
}

/// Find the first repeated substring present in `s`.
pub fn find_repeat<'a>(s: &'a str) -> Option<RepeatedSubstring<'a>> {
    // ...
}
```

The propagation of lifetime parameters makes sense: anything that contains a reference, no matter how deeply nested, is valid only for the lifetime of the item referred to. If that item is moved or dropped, then the whole chain of data structures is no longer valid.

However, this also means that data structures involving references are harder to use—the owner of the data structure has to ensure that the lifetimes all line up. As a result, **prefer data structures that own their contents** where possible, particularly if the code doesn't need to be highly optimized (Item 20). Where that's not possible, the various smart pointer types (e.g., `Rc`) described in Item 8 can help untangle the lifetime constraints.

Anonymous Lifetimes

When it's not possible to stick to data structures that own their contents, the data structure will necessarily end up with a lifetime parameter, as described in the previous section. This can create a slightly unfortunate interaction with the lifetime elision rules described earlier in the Item.

For example, consider a function that returns a data structure with a lifetime parameter. The fully explicit signature for this function makes the lifetimes involved clear:

```
pub fn find_one_item<'a>(items: &'a [Item]) -> ReferenceHolder<'a> {
    // ...
}
```

However, the same signature with lifetimes elided can be a little misleading:

```
pub fn find_one_item(items: &[Item]) -> ReferenceHolder {
    // ...
}
```

Because the lifetime parameter for the return type is elided, a human reading the code doesn't get much of a hint that lifetimes are involved.

The anonymous lifetime `'_` allows you to mark an elided lifetime as being present, without having to fully restore *all* of the lifetime names:

```
pub fn find_one_item(items: &[Item]) -> ReferenceHolder<'_> {
    // ...
}
```

Roughly speaking, the `'_` marker asks the compiler to invent a unique lifetime name for us, which we can use in situations where we never need to use the name elsewhere.

That means it's also useful for other lifetime elision scenarios. For example, the declaration for the `fmt` method of the `Debug` trait uses the anonymous lifetime to indicate that the `Formatter` instance has a different lifetime than `&self`, but it's not important what that lifetime's name is:

```
pub trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

Things to Remember

- All Rust references have an associated lifetime, indicated by a lifetime label (e.g., `'a`). The lifetime labels for function parameters and return values can be elided in some common cases (but are still present under the covers).
- Any data structure that (transitively) includes a reference has an associated lifetime parameter; as a result, it's often easier to work with data structures that own their contents.
- The `'static` lifetime is used for references to items that are guaranteed never to go out of scope, such as global data or items on the heap that have been explicitly leaked.
- Lifetime labels can be used only to indicate that lifetimes are the "same", which means that the output lifetime is contained within the input lifetime(s).
- The anonymous lifetime label `'_` can be used in places where a specific lifetime label is not needed.

---

[1] For example, the Chromium project estimates that 70% of security bugs are due to memory safety.

Item 15: Understand the borrow checker

Values in Rust have an owner, but that owner can lend the values out to other places in the code. This *borrowing* mechanism involves the creation and use of *references*, subject to rules policed by the *borrow checker*—the subject of this Item.

Under the covers, Rust's references use the same kind of *pointer* values (Item 8) that are so prevalent in C or C++ code but are girded with rules and restrictions to make sure that the sins of C/C++ are avoided. As a quick comparison:

- Like a C/C++ pointer, a Rust reference is created with an ampersand: `&value`.
- Like a C++ reference, a Rust reference can never be `nullptr`.
- Like a C/C++ pointer or reference, a Rust reference can be modified after creation to refer to something different.
- Unlike C++, producing a reference from a value always involves an explicit (`&`) conversion—if you see code like `f(value)`, you know that `f` is receiving ownership of the value. (However, it may be ownership of a *copy* of the item, if the `value`'s type implements `Copy`—see Item 10.)
- Unlike C/C++, the mutability of a newly created reference is always explicit (`&mut`). If you see code like `f(&value)`, you know that `value` won't be modified (i.e., is `const` in C/C++ terminology). Only expressions like `f(&mut value)` have the potential to change the contents of `value`.[1]

The most important difference between a C/C++ pointer and a Rust reference is indicated by the term *borrow*: you can take a reference (pointer) to an item, *but you can't keep that reference forever*. In particular, you can't keep it longer than the lifetime of the underlying item, as tracked by the compiler and explored in Item 14.

These restrictions on the use of references enable Rust to make its memory safety guarantees, but they also mean that you have to accept the cognitive costs of the borrow rules, and accept that it will change how you design your software—particularly its data structures.

This Item starts by describing what Rust references can do, and the borrow checker's rules for using them. The rest of the Item focuses on dealing with the consequences of those rules: how to refactor, rework, and redesign your code so that you can win fights against the borrow checker.

Access Control

There are three ways to access the contents of a Rust item: via the item's *owner* (`item`), a *reference* (`&item`), or a *mutable reference* (`&mut item`). Each of these ways of accessing the item comes with different powers over the item. Putting things roughly in terms of the CRUD (create/read/update/delete) model for storage (using Rust's *drop* terminology in place of *delete*):

- The owner of an item gets to *create* it, *read* from it, *update* it, and *drop* it.
- A mutable reference can be used to *read* from the underlying item and *update* it.
- A (normal) reference can be used only to *read* from the underlying item.

There's an important Rust-specific aspect to these data access rules: only the item's owner can *move* the item. This makes sense if you think of a move as being some combination of *creating* (in the new location) and *dropping* the item's memory (at the old location).

This can lead to some oddities for code that has a mutable reference to an item. For example, it's OK to overwrite an `Option` :

```
/// Some data structure used by the code.
#[derive(Debug)]
pub struct Item {
    pub contents: i64,
}

/// Replace the content of `item` with `val`.
pub fn replace(item: &mut Option<Item>, val: Item) {
    *item = Some(val);
}
```

but a modification to also return the previous value falls foul of the move restriction:[2]

```
/// Replace the content of `item` with `val`, returning the previous
/// contents.
pub fn replace(item: &mut Option<Item>, val: Item) -> Option<Item> {
    let previous = *item; // move out
    *item = Some(val); // replace
    previous
}
```

```
error[E0507]: cannot move out of `*item` which is behind a mutable reference
  --> src/main.rs:34:24
   |
34 |         let previous = *item; // move out
   |                        ^^^^^ move occurs because `*item` has type
   |                              `Option<inner::Item>`, which does not
   |                              implement the `Copy` trait
   |
help: consider removing the dereference here
   |
34 -         let previous = *item; // move out
34 +         let previous = item; // move out
   |
```

Although it's valid to *read* from a mutable reference, this code is attempting to *move* the value out, just prior to replacing the moved value with a new value—in an attempt to avoid making a copy of the original value. The borrow checker has to be conservative and notices that there's a moment between the two lines when the mutable reference isn't referring to a valid value.

As humans, we can see that this combined operation—extracting the old value and replacing it with a new value—is both safe and useful, so the standard library provides the `std::mem::replace` function to perform it. Under the covers, `replace` uses `unsafe` (as per Item 16) to perform the swap in one go:

```
/// Replace the content of `item` with `val`, returning the previous
/// contents.
pub fn replace(item: &mut Option<Item>, val: Item) -> Option<Item> {
    std::mem::replace(item, Some(val)) // returns previous value
}
```

For `Option` types in particular, this is a sufficiently common pattern that there is also a `replace` method on `Option` itself:

```
/// Replace the content of `item` with `val`, returning the previous
/// contents.
pub fn replace(item: &mut Option<Item>, val: Item) -> Option<Item> {
    item.replace(val) // returns previous value
}
```

Borrow Rules

There are two key rules to remember when borrowing references in Rust.

The first rule is that the scope of any reference must be smaller than the lifetime of the item that it refers to. Lifetimes are explored in detail in Item 14, but it's worth noting that the compiler has special behavior for reference lifetimes; the *non-lexical lifetimes* feature allows

reference lifetimes to be shrunk so they end at the point of last use, rather than the enclosing block.

The second rule for borrowing references is that, in addition to the owner of an item, there can be either of the following:

- Any number of immutable references to the item
- A single mutable reference to the item

However, there can't be both (at the same point in the code).

So a function that takes multiple immutable references can be fed references to the same item:

```
/// Indicate whether both arguments are zero.
fn both_zero(left: &Item, right: &Item) -> bool {
    left.contents == 0 && right.contents == 0
}

let item = Item { contents: 0 };
assert!(both_zero(&item, &item));
```

but one that takes *mutable* references cannot:

```
/// Zero out the contents of both arguments.
fn zero_both(left: &mut Item, right: &mut Item) {
    left.contents = 0;
    right.contents = 0;
}

let mut item = Item { contents: 42 };
zero_both(&mut item, &mut item);
```

```
error[E0499]: cannot borrow `item` as mutable more than once at a time
   --> src/main.rs:131:26
    |
131 |     zero_both(&mut item, &mut item);
    |     --------- ---------  ^^^^^^^^^ second mutable borrow occurs here
    |     |         |
    |     |         first mutable borrow occurs here
    |     first borrow later used by call
```

The same restriction is true for a function that uses a mixture of mutable and immutable references:

```
/// Set the contents of `left` to the contents of `right`.
fn copy_contents(left: &mut Item, right: &Item) {
    left.contents = right.contents;
}

let mut item = Item { contents: 42 };
copy_contents(&mut item, &item);
```

```
error[E0502]: cannot borrow `item` as immutable because it is also borrowed
              as mutable
   --> src/main.rs:159:30
    |
159 |     copy_contents(&mut item, &item);
    |     ------------- ---------  ^^^^^ immutable borrow occurs here
    |     |             |
    |     |             mutable borrow occurs here
    |     mutable borrow later used by call
```

The borrowing rules allow the compiler to make better decisions around *aliasing*: tracking when two different pointers may or may not refer to the same underlying item in memory. If the compiler can be sure (as in Rust) that the memory location pointed to by a collection of immutable references cannot be altered via an aliased *mutable* reference, then it can generate code that has the following advantages:

- *It's better optimized*: Values can be, for example, cached in registers, secure in the knowledge that the underlying memory contents will not change in the meantime.
- *It's safer*: Data races arising from unsynchronized access to memory between threads (Item 17) are not possible.

Owner Operations

One important consequence of the rules around the existence of references is that they also affect what operations can be performed by the owner of the item. One way to help understand this is to imagine that operations involving the owner are performed by creating and using references under the covers.

For example, an attempt to update the item via its owner is equivalent to making an ephemeral mutable reference and then updating the item via that reference. If another reference already exists, this notional second mutable reference can't be created:

```
let mut item = Item { contents: 42 };
let r = &item;
item.contents = 0;
// ^^^ Changing the item is roughly equivalent to:
//   (&mut item).contents = 0;
println!("reference to item is {:?}", r);
```

```
error[E0506]: cannot assign to `item.contents` because it is borrowed
   --> src/main.rs:200:5
    |
199 |     let r = &item;
    |             ----- `item.contents` is borrowed here
200 |     item.contents = 0;
    |     ^^^^^^^^^^^^^^^^^ `item.contents` is assigned to here but it was
    |                       already borrowed
...
203 |     println!("reference to item is {:?}", r);
    |                                           - borrow later used here
```

On the other hand, because multiple *immutable* references are allowed, it's OK for the owner to read from the item while there are immutable references in existence:

```
let item = Item { contents: 42 };
let r = &item;
let contents = item.contents;
// ^^^ Reading from the item is roughly equivalent to:
//   let contents = (&item).contents;
println!("reference to item is {:?}", r);
```

but not if there is a *mutable* reference:

```
let mut item = Item { contents: 42 };
let r = &mut item;
let contents = item.contents; // i64 implements `Copy`
r.contents = 0;
```

```
error[E0503]: cannot use `item.contents` because it was mutably borrowed
   --> src/main.rs:231:20
    |
230 |     let r = &mut item;
    |             --------- `item` is borrowed here
231 |     let contents = item.contents; // i64 implements `Copy`
    |                    ^^^^^^^^^^^^ use of borrowed `item`
232 |     r.contents = 0;
    |     -------------- borrow later used here
```

Finally, the existence of any sort of active reference prevents the owner of the item from moving or dropping the item, exactly because this would mean that the reference now refers to an invalid item:

```
let item = Item { contents: 42 };
let r = &item;
let new_item = item; // move
println!("reference to item is {:?}", r);
```

```
error[E0505]: cannot move out of `item` because it is borrowed
   --> src/main.rs:170:20
    |
168 |     let item = Item { contents: 42 };
    |         ---- binding `item` declared here
169 |     let r = &item;
    |             ----- borrow of `item` occurs here
170 |     let new_item = item; // move
    |                    ^^^^ move out of `item` occurs here
171 |     println!("reference to item is {:?}", r);
    |                                           - borrow later used here
```

This is a scenario where the non-lexical lifetime feature described in Item 14 is particularly helpful, because (roughly speaking) it terminates the lifetime of a reference at the point where the reference is last used, rather than at the end of the enclosing scope. Moving the final use of the reference up before the move happens means that the compilation error evaporates:

```
let item = Item { contents: 42 };
let r = &item;
println!("reference to item is {:?}", r);

// Reference `r` is still in scope but has no further use, so it's
// as if the reference has already been dropped.
let new_item = item; // move works OK
```

Winning Fights Against the Borrow Checker

Newcomers to Rust (and even more experienced folk!) can often feel that they are spending time fighting against the borrow checker. What kinds of things can help you win these battles?

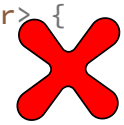
Local code refactoring

The first tactic is to pay attention to the compiler's error messages, because the Rust developers have put a lot of effort into making them as helpful as possible:

```
/// If `needle` is present in `haystack`, return a slice containing it.
pub fn find<'a, 'b>(haystack: &'a str, needle: &'b str) -> Option<&'a str> {
    haystack
        .find(needle)
        .map(|i| &haystack[i..i + needle.len()])
}

// ...

let found = find(&format!("{} to search", "Text"), "ex");
if let Some(text) = found {
    println!("Found '{text}'!");
}
```



```
error[E0716]: temporary value dropped while borrowed
   --> src/main.rs:353:23
    |
353 | let found = find(&format!("{} to search", "Text"), "ex");
    |                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^      - temporary value
    |                   |                 is freed at the end of this statement
    |                   |
    |                   creates a temporary value which is freed while still in
    |                   use
354 | if let Some(text) = found {
    |                     ----- borrow later used here
    |
    = note: consider using a `let` binding to create a longer lived value
```

The first part of the error message is the important part, because it describes what borrowing rule the compiler thinks you have broken and why. As you encounter enough of these errors—which you will—you can build up an intuition about the borrow checker that matches the more theoretical version encapsulated in the previously stated rules.

The second part of the error message includes the compiler's suggestions for how to fix the problem, which in this case is simple:

```
let haystack = format!("{} to search", "Text");
let found = find(&haystack, "ex");
if let Some(text) = found {
    println!("Found '{text}'!");
}
// `found` now references `haystack`, which outlives it
```

This is an instance of one of the two simple code tweaks that can help mollify the borrow checker:

- *Lifetime extension*: Convert a temporary (whose lifetime extends only to the end of the expression) into a new named local variable (whose lifetime extends to the end of the block) with a `let` binding.
- *Lifetime reduction*: Add an additional block `{ ... }` around the use of a reference so that its lifetime ends at the end of the new block.

The latter is less common, because of the existence of non-lexical lifetimes: the compiler can often figure out that a reference is no longer used, ahead of its official drop point at the end of the block. However, if you do find yourself repeatedly introducing an artificial block around similar small chunks of code, consider whether that code should be encapsulated into a method of its own.

The compiler's suggested fixes are helpful for simpler problems, but as you write more sophisticated code, you're likely to find that the suggestions are no longer useful and that the explanation of the broken borrowing rule is harder to follow:

```
let x = Some(Rc::new(RefCell::new(Item { contents: 42 })));

// Call function with signature: `check_item(item: Option<&Item>)`
check_item(x.as_ref().map(|r| r.borrow().deref()));


error[E0515]: cannot return reference to temporary value
   --> src/main.rs:293:35
    |
293 |     check_item(x.as_ref().map(|r| r.borrow().deref()));
    |                                   ----------^^^^^^^^
    |                                   |
    |                                   returns a reference to data owned by
the
    |                                       current function
    |                                   temporary value created here
```

In this situation, it can be helpful to temporarily introduce a sequence of local variables, one for each step of a complicated transformation, and each with an explicit type annotation:

```
let x: Option<Rc<RefCell<Item>>> =
    Some(Rc::new(RefCell::new(Item { contents: 42 })));

let x1: Option<&Rc<RefCell<Item>>> = x.as_ref();
let x2: Option<std::cell::Ref<Item>> = x1.map(|r| r.borrow());
let x3: Option<&Item> = x2.map(|r| r.deref());
check_item(x3);


error[E0515]: cannot return reference to function parameter `r`
   --> src/main.rs:305:40
    |
305 |     let x3: Option<&Item> = x2.map(|r| r.deref());
    |                                        ^^^^^^^^^ returns a reference to
    |                                      data owned by the current function
```

This narrows down the precise conversion that the compiler is complaining about, which in turn allows the code to be restructured:

```
let x: Option<Rc<RefCell<Item>>> =
    Some(Rc::new(RefCell::new(Item { contents: 42 })));

let x1: Option<&Rc<RefCell<Item>>> = x.as_ref();
let x2: Option<std::cell::Ref<Item>> = x1.map(|r| r.borrow());
match x2 {
    None => check_item(None),
    Some(r) => {
        let x3: &Item = r.deref();
        check_item(Some(x3));
    }
}
```

Once the underlying problem is clear and has been fixed, you're then free to recoalesce the local variables back together so that you can pretend you got it right all along:

```
let x = Some(Rc::new(RefCell::new(Item { contents: 42 })));

match x.as_ref().map(|r| r.borrow()) {
    None => check_item(None),
    Some(r) => check_item(Some(r.deref())),
};
```

Data structure design

The next tactic that helps for battles against the borrow checker is to design your data structures with the borrow checker in mind. The panacea is your data structures owning all of the data that they use, avoiding any use of references and the consequent propagation of lifetime annotations described in Item 14.

However, that's not always possible for real-world data structures; any time the internal connections of the data structure form a graph that's more interconnected than a tree pattern (a `Root` that owns multiple `Branch`es, each of which owns multiple `Leaf`s, etc.), then simple single-ownership isn't possible.

To take a simple example, imagine a simple register of guest details recorded in the order in which they arrive:

```rust
#[derive(Clone, Debug)]
pub struct Guest {
    name: String,
    address: String,
    // ... many other fields
}

/// Local error type, used later.
#[derive(Clone, Debug)]
pub struct Error(String);

/// Register of guests recorded in order of arrival.
#[derive(Default, Debug)]
pub struct GuestRegister(Vec<Guest>);

impl GuestRegister {
    pub fn register(&mut self, guest: Guest) {
        self.0.push(guest)
    }
    pub fn nth(&self, idx: usize) -> Option<&Guest> {
        self.0.get(idx)
    }
}
```

If this code *also* needs to be able to efficiently look up guests by arrival and alphabetically by name, then there are fundamentally two distinct data structures involved, and only one of them can own the data.

If the data involved is both small and immutable, then just cloning the data can be a quick solution:

```
mod cloned {
    use super::Guest;

    #[derive(Default, Debug)]
    pub struct GuestRegister {
        by_arrival: Vec<Guest>,
        by_name: std::collections::BTreeMap<String, Guest>,
    }

    impl GuestRegister {
        pub fn register(&mut self, guest: Guest) {
            // Requires `Guest` to be `Clone`
            self.by_arrival.push(guest.clone());
            // Not checking for duplicate names to keep this
            // example shorter.
            self.by_name.insert(guest.name.clone(), guest);
        }
        pub fn named(&self, name: &str) -> Option<&Guest> {
            self.by_name.get(name)
        }
        pub fn nth(&self, idx: usize) -> Option<&Guest> {
            self.by_arrival.get(idx)
        }
    }
}
```

However, this approach of cloning copes poorly if the data can be modified. For example, if the address for a `Guest` needs to be updated, you have to find both versions and ensure they stay in sync.

Another possible approach is to add another layer of indirection, treating the `Vec<Guest>` as the owner and using an index into that vector for the name lookups:

```
mod indexed {
    use super::Guest;

    #[derive(Default)]
    pub struct GuestRegister {
        by_arrival: Vec<Guest>,
        // Map from guest name to index into `by_arrival`.
        by_name: std::collections::BTreeMap<String, usize>,
    }

    impl GuestRegister {
        pub fn register(&mut self, guest: Guest) {
            // Not checking for duplicate names to keep this
            // example shorter.
            self.by_name
                .insert(guest.name.clone(), self.by_arrival.len());
            self.by_arrival.push(guest);
        }
        pub fn named(&self, name: &str) -> Option<&Guest> {
            let idx = *self.by_name.get(name)?;
            self.nth(idx)
        }
        pub fn named_mut(&mut self, name: &str) -> Option<&mut Guest> {
            let idx = *self.by_name.get(name)?;
            self.nth_mut(idx)
        }
        pub fn nth(&self, idx: usize) -> Option<&Guest> {
            self.by_arrival.get(idx)
        }
        pub fn nth_mut(&mut self, idx: usize) -> Option<&mut Guest> {
            self.by_arrival.get_mut(idx)
        }
    }
}
```

In this approach, each guest is represented by a single `Guest` item, which allows the `named_mut()` method to return a mutable reference to that item. That in turn means that changing a guest's address works fine—the (single) `Guest` is owned by the `Vec` and will always be reached that way under the covers:

```
let new_address = "123 Bigger House St";
// Real code wouldn't assume that "Bob" exists...
ledger.named_mut("Bob").unwrap().address = new_address.to_string();

assert_eq!(ledger.named("Bob").unwrap().address, new_address);
```

However, if guests can deregister, it's easy to inadvertently introduce a bug:

```
// Deregister the `Guest` at position `idx`, moving up all
// subsequent guests.
pub fn deregister(&mut self, idx: usize) -> Result<(), super::Error> {
    if idx >= self.by_arrival.len() {
        return Err(super::Error::new("out of bounds"));
    }
    self.by_arrival.remove(idx);

    // Oops, forgot to update `by_name`.

    Ok(())
}
```

Now that the `Vec` can be shuffled, the `by_name` indexes into it are effectively acting like pointers, and we've reintroduced a world where a bug can lead those "pointers" to point to nothing (beyond the `Vec` bounds) or to point to incorrect data:

```
ledger.register(alice);
ledger.register(bob);
ledger.register(charlie);
println!("Register starts as: {ledger:?}");

ledger.deregister(0).unwrap();
println!("Register after deregister(0): {ledger:?}");

let also_alice = ledger.named("Alice");
// Alice still has index 0, which is now Bob
println!("Alice is {also_alice:?}");

let also_bob = ledger.named("Bob");
// Bob still has index 1, which is now Charlie
println!("Bob is {also_bob:?}");

let also_charlie = ledger.named("Charlie");
// Charlie still has index 2, which is now beyond the Vec
println!("Charlie is {also_charlie:?}");
```

The code here uses a custom `Debug` implementation (not shown), in order to reduce the size of the output; this truncated output is as follows:

```
Register starts as: {
  by_arrival: [{n: 'Alice', ...}, {n: 'Bob', ...}, {n: 'Charlie', ...}]
  by_name: {"Alice": 0, "Bob": 1, "Charlie": 2}
}
Register after deregister(0): {
  by_arrival: [{n: 'Bob', ...}, {n: 'Charlie', ...}]
  by_name: {"Alice": 0, "Bob": 1, "Charlie": 2}
}
Alice is Some(Guest { name: "Bob", address: "234 Bobton" })
Bob is Some(Guest { name: "Charlie", address: "345 Charlieland" })
Charlie is None
```

The preceding example showed a bug in the `deregister` code, but even after that bug is fixed, there's nothing to prevent a caller from hanging onto an index value and using it with `nth()` —getting unexpected or invalid results.

The core problem is that the two data structures need to be kept in sync. A better approach for handling this is to use Rust's smart pointers instead (Item 8). Shifting to a combination of `Rc` and `RefCell` avoids the invalidation problems of using indices as pseudo-pointers. Updating the example—but keeping the bug in it—gives the following:

```
mod rc {
    use super::{Error, Guest};
    use std::{cell::RefCell, rc::Rc};

    #[derive(Default)]
    pub struct GuestRegister {
        by_arrival: Vec<Rc<RefCell<Guest>>>,
        by_name: std::collections::BTreeMap<String, Rc<RefCell<Guest>>>,
    }

    impl GuestRegister {
        pub fn register(&mut self, guest: Guest) {
            let name = guest.name.clone();
            let guest = Rc::new(RefCell::new(guest));
            self.by_arrival.push(guest.clone());
            self.by_name.insert(name, guest);
        }
        pub fn deregister(&mut self, idx: usize) -> Result<(), Error> {
            if idx >= self.by_arrival.len() {
                return Err(Error::new("out of bounds"));
            }
            self.by_arrival.remove(idx);

            // Oops, still forgot to update `by_name`.

            Ok(())
        }
        // ...
    }
}
```

```
Register starts as: {
  by_arrival: [{n: 'Alice', ...}, {n: 'Bob', ...}, {n: 'Charlie', ...}]
  by_name: [("Alice", {n: 'Alice', ...}), ("Bob", {n: 'Bob', ...}),
            ("Charlie", {n: 'Charlie', ...})]
}
Register after deregister(0): {
  by_arrival: [{n: 'Bob', ...}, {n: 'Charlie', ...}]
  by_name: [("Alice", {n: 'Alice', ...}), ("Bob", {n: 'Bob', ...}),
            ("Charlie", {n: 'Charlie', ...})]
}
Alice is Some(RefCell { value: Guest { name: "Alice",
                                       address: "123 Aliceville" } })
Bob is Some(RefCell { value: Guest { name: "Bob",
                                     address: "234 Bobton" } })
Charlie is Some(RefCell { value: Guest { name: "Charlie",
                                         address: "345 Charlieland" } })
```

The output no longer has mismatched names, but a lingering entry for Alice remains until we fix the bug by ensuring that the two collections stay in sync:

```
pub fn deregister(&mut self, idx: usize) -> Result<(), Error> {
    if idx >= self.by_arrival.len() {
        return Err(Error::new("out of bounds"));
    }
    let guest: Rc<RefCell<Guest>> = self.by_arrival.remove(idx);
    self.by_name.remove(&guest.borrow().name);
    Ok(())
}
```

```
Register after deregister(0): {
  by_arrival: [{n: 'Bob', ...}, {n: 'Charlie', ...}]
  by_name: [("Bob", {n: 'Bob', ...}), ("Charlie", {n: 'Charlie', ...})]
}
Alice is None
Bob is Some(RefCell { value: Guest { name: "Bob",
                                     address: "234 Bobton" } })
Charlie is Some(RefCell { value: Guest { name: "Charlie",
                                         address: "345 Charlieland" } })
```

Smart pointers

The final variation of the previous section is an example of a more general approach: **use Rust's smart pointers for interconnected data structures**.

Item 8 described the most common smart pointer types provided by Rust's standard library:

- `Rc` allows shared ownership, with multiple things referring to the same item. `Rc` is often combined with `RefCell`.
- `RefCell` allows interior mutability so that internal state can be modified without needing a mutable reference. This comes at the cost of moving borrow checks from

compile time to runtime.

- `Arc` is the multithreading equivalent to `Rc`.
- `Mutex` (and `RwLock`) allows interior mutability in a multithreading environment, roughly equivalent to `RefCell`.
- `Cell` allows interior mutability for `Copy` types.

For programmers who are adapting from C++ to Rust, the most common tool to reach for is `Rc<T>` (and its thread-safe cousin `Arc<T>`), often combined with `RefCell` (or the thread-safe alternative `Mutex`). A naive translation of shared pointers (or even `std::shared_ptr`s) to `Rc<RefCell<T>>` instances will generally give something that works in Rust without too much complaint from the borrow checker.

However, this approach means that you miss out on some of the protections that Rust gives you. In particular, situations where the same item is mutably borrowed (via `borrow_mut()`) while another reference exists result in a runtime `panic!` rather than a compile-time error.

For example, one pattern that breaks the one-way flow of ownership in tree-like data structures is when there's an "owner" pointer back from an item to the thing that owns it, as shown in Figure 3-3. These `owner` links are useful for moving around the data structure; for example, adding a new sibling to a `Leaf` needs to involve the owning `Branch`.

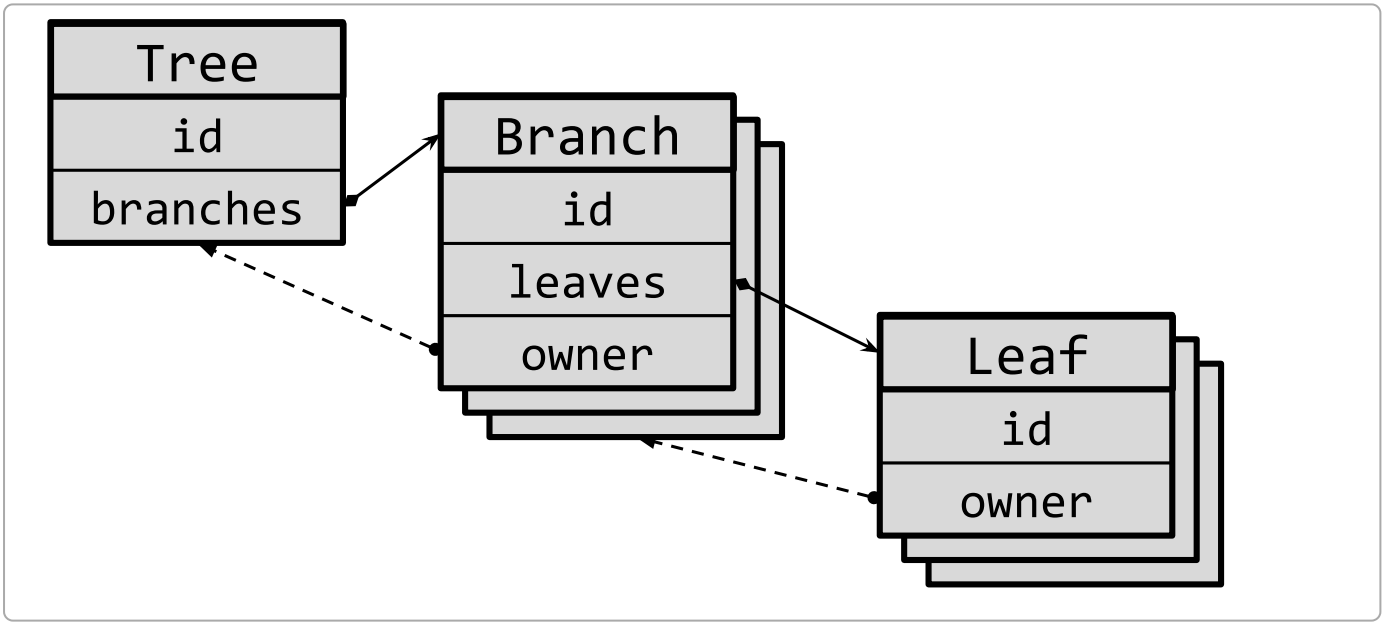


Figure 3-3. Tree data structure layout

Implementing this pattern in Rust can make use of `Rc<T>`'s more tentative partner, `Weak<T>`:

```
use std::{
    cell::RefCell,
    rc::{Rc, Weak},
};

// Use a newtype for each identifier type.
struct TreeId(String);
struct BranchId(String);
struct LeafId(String);

struct Tree {
    id: TreeId,
    branches: Vec<Rc<RefCell<Branch>>>,
}

struct Branch {
    id: BranchId,
    leaves: Vec<Rc<RefCell<Leaf>>>,
    owner: Option<Weak<RefCell<Tree>>>,
}

struct Leaf {
    id: LeafId,
    owner: Option<Weak<RefCell<Branch>>>,
}
```

The `Weak` reference doesn't increment the main refcount and so has to explicitly check whether the underlying item has gone away:

```
impl Branch {
    fn add_leaf(branch: Rc<RefCell<Branch>>, mut leaf: Leaf) {
        leaf.owner = Some(Rc::downgrade(&branch));
        branch.borrow_mut().leaves.push(Rc::new(RefCell::new(leaf)));
    }

    fn location(&self) -> String {
        match &self.owner {
            None => format!("<unowned>.{}", self.id.0),
            Some(owner) => {
                // Upgrade weak owner pointer.
                let tree = owner.upgrade().expect("owner gone!");
                format!("{}.{}", tree.borrow().id.0, self.id.0)
            }
        }
    }
}
```

If Rust's smart pointers don't seem to cover what's needed for your data structures, there's always the final fallback of writing `unsafe` code that uses raw (and decidedly un-smart) pointers. However, as per Item 16, this should very much be a last resort—someone else might have already implemented the semantics you want, inside a safe interface, and if you search the standard library and `crates.io`, you might find just the tool for the job.

For example, imagine that you have a function that sometimes returns a reference to one of its inputs but sometimes needs to return some freshly allocated data. In line with Item 1, an `enum` that encodes these two possibilities is the natural way to express this in the type system, and you could then implement various pointer traits described in Item 8. But you don't have to: the standard library already includes the `std::borrow::Cow` type that covers exactly this scenario once you know it exists.[3]

Self-referential data structures

One particular battle with the borrow checker always stymies programmers arriving at Rust from other languages: attempting to create self-referential data structures, which contain a mixture of owned data together with references to within that owned data:

```
struct SelfRef {
    text: String,
    // The slice of `text` that holds the title text.
    title: Option<&str>,
}
```

At a syntactic level, this code won't compile because it doesn't comply with the lifetime rules described in Item 14: the reference needs a lifetime annotation, and that means the containing data structure would also need a lifetime parameter. But a lifetime would be for something external to this `SelfRef` struct, which is not the intent: the data being referenced is internal to the struct.

It's worth thinking about the reason for this restriction at a more semantic level. Data structures in Rust can *move*: from the stack to the heap, from the heap to the stack, and from one place to another. If that happens, the "interior" `title` pointer would no longer be valid, and there's no way to keep it in sync.

A simple alternative for this case is to use the indexing approach explored earlier: a range of offsets into the `text` is not invalidated by a move and is invisible to the borrow checker because it doesn't involve references:

```
struct SelfRefIdx {
    text: String,
    // Indices into `text` where the title text is.
    title: Option<std::ops::Range<usize>>,
}
```

However, this indexing approach works only for simple examples and has the same drawbacks as noted previously: the index itself becomes a pseudo-pointer that can become out of sync or even refer to ranges of the `text` that no longer exist.

A more general version of the self-reference problem turns up when the compiler deals with `async` code.[4] Roughly speaking, the compiler bundles up a pending chunk of `async` code

into a closure, which holds both the code and any captured parts of the environment that the code works with (as described in Item 2). This captured environment can include both values and references to those values. That's inherently a self-referential data structure, and so `async` support was a prime motivation for the `Pin` type in the standard library. This pointer type "pins" its value in place, forcing the value to remain at the same location in memory, thus ensuring that internal self-references remain valid.

So `Pin` is available as a possibility for self-referential types, but it's tricky to use correctly—be sure to read the official docs.

Where possible, **avoid self-referential data structures**, or try to find library crates that encapsulate the difficulties for you (e.g., `ouroborous`).

Things to Remember

- Rust's references are *borrowed*, indicating that they cannot be held forever.
- The borrow checker allows multiple immutable references or a single mutable reference to an item but not both. The lifetime of a reference stops at the point of last use, rather than at the end of the enclosing scope, due to non-lexical lifetimes.
- Errors from the borrow checker can be dealt with in various ways:
  - Adding an additional `{ ... }` scope can reduce the extent of a value's lifetime.
  - Adding a named local variable for a value extends the value's lifetime to the end of the scope.
  - Temporarily adding multiple local variables can help narrow down what the borrow checker is complaining about.
- Rust's smart pointer types provide ways around the borrow checker's rules and so are useful for interconnected data structures.
- However, self-referential data structures remain awkward to deal with in Rust.

---

[1] Note that all bets are off with expressions like `m!(value)` that involve a macro (Item 28), because that can expand to arbitrary code.

[2] The compiler's suggestion doesn't help here, because `item` is needed on the subsequent line.

[3] `Cow` stands for clone-on-write; a copy of the underlying data is made only if a change (write) needs to be made to it.

[4] Dealing with `async` code is beyond the scope of this book; to understand more about its need for self-referential data structures, see Chapter 8 of *Rust for Rustaceans* by Jon Gjengset (No Starch Press).

Item 16: Avoid writing unsafe code

The memory safety guarantees—without runtime overhead—of Rust are its unique selling point; it is the Rust language feature that is not found in any other mainstream language. These guarantees come at a cost: writing Rust requires you to reorganize your code to mollify the borrow checker (Item 15) and to precisely specify the reference types that you use (Item 8).

Unsafe Rust is a superset of the Rust language that weakens some of these restrictions—and the corresponding guarantees. Prefixing a block with the `unsafe` keyword switches that block into unsafe mode, which allows things that are not supported in normal Rust. In particular, it allows the use of *raw pointers* that work more like old-style C pointers. These pointers are not subject to the borrowing rules, and the programmer is responsible for ensuring that they still point to valid memory whenever they're dereferenced.

So at a superficial level, the advice of this Item is trivial: why move to Rust if you're just going to write C code in Rust? However, there are occasions where `unsafe` code is absolutely required: for low-level library code or for when your Rust code has to interface with code in other languages (Item 34).

The wording of this Item is quite precise, though: **avoid *writing* `unsafe` code**. The emphasis is on the "writing", because much of the time, the `unsafe` code you're likely to need has already been written for you.

The Rust standard libraries contain a lot of `unsafe` code; a quick search finds around 1,000 uses of `unsafe` in the `alloc` library, 1,500 in `core`, and a further 2,000 in `std`. This code has been written by experts and is battle-hardened by use in many thousands of Rust codebases.

Some of this `unsafe` code happens under the covers in standard library features that we've already covered:

- The smart pointer types—`Rc`, `RefCell`, `Arc`, and friends—described in Item 8 use `unsafe` code (often raw pointers) internally to be able to present their particular semantics to their users.
- The synchronization primitives—`Mutex`, `RwLock`, and associated guards—from Item 17 use `unsafe` OS-specific code internally. *Rust Atomics and Locks* by Mara Bos (O'Reilly) is recommended if you want to understand the subtle details involved in these primitives.

The standard library also has other functionality covering more advanced features, implemented with `unsafe` internally:[1]

- `std::pin::Pin` forces an item to not move in memory (Item 15). This allows self-referential data structures, often a bête noire for new arrivals to Rust.
- `std::borrow::Cow` provides a clone-on-write smart pointer: the same pointer can be used for both reading and writing, and a clone of the underlying data happens only if and when a write occurs.
- Various functions (`take` , `swap` , `replace`) in `std::mem` allow items in memory to be manipulated without falling foul of the borrow checker.

These features may still need a little caution to be used correctly, but the `unsafe` code has been encapsulated in a way that removes whole classes of problems.

Moving beyond the standard library, the `crates.io` ecosystem also includes many crates that encapsulate `unsafe` code to provide a frequently used feature:

- `once_cell` : Provides a way to have something like global variables, initialized exactly once.
- `rand` : Provides random number generation, making use of the lower-level underlying features provided by the operating system and CPU.
- `byteorder` : Allows raw bytes of data to be converted to and from numbers.
- `cxx` : Allows C++ code and Rust code to interoperate (also mentioned in Item 35).

There are many other examples, but hopefully the general idea is clear. If you want to do something that doesn't obviously fit within the constraints of Rust (especially Item 14 and Item 15), hunt through the standard library to see if there's existing functionality that does what you need. If you don't find what you need, try also hunting through `crates.io` . After all, it's unusual to encounter a unique problem that no one else has ever faced before.

Of course, there will always be places where `unsafe` is forced, for example, when you need to interact with code written in other languages via a foreign function interface (FFI), as discussed in Item 34. But when it's necessary, **consider writing a wrapper layer that holds all the `unsafe` code** that's required so that other programmers can then follow the advice given in this Item. This also helps to localize problems: when something goes wrong, the `unsafe` wrapper can be the first suspect.

Also, if you're forced to write `unsafe` code, pay attention to the warning implied by the keyword itself: *Hic sunt dracones*.

- Add safety comments that document the preconditions and invariants that the `unsafe` code relies on. Clippy (Item 29) has a warning to remind you about this.
- Minimize the amount of code contained in an `unsafe` block, to limit the potential blast radius of a mistake. Consider enabling the `unsafe_op_in_unsafe_fn` lint so that explicit `unsafe` blocks are required when performing `unsafe` operations, even when those operations are performed in a function that is `unsafe` itself.
- Write even more tests (Item 30) than usual.
- Run additional diagnostic tools (Item 31) over the code. In particular, **consider running Miri over your `unsafe` code**—Miri interprets the intermediate level output from the

compiler, that allows it to detect classes of errors that are invisible to the Rust compiler.
- Think carefully about multithreaded use, particularly if there's shared state (Item 17).

Adding the `unsafe` marker doesn't mean that no rules apply—it means that *you* (the programmer) are now responsible for maintaining Rust's safety guarantees, rather than the compiler.

---

[1] In practice, most of this `std` functionality is actually provided by `core` and so is available to `no_std` code as described in Item 33.

Item 17: Be wary of shared-state parallelism

> "Even the most daring forms of sharing are guaranteed safe in Rust." – Aaron Turon

---

The official documentation describes Rust as enabling "fearless concurrency", but this Item will explore why (sadly) there are still some reasons to be afraid of concurrency, even in Rust.

This Item is specific to *shared-state* parallelism: where different threads of execution communicate with each other by sharing memory. Sharing state between threads generally comes with *two* terrible problems, regardless of the language involved:

- *Data races*: These can lead to corrupted data.
- *Deadlocks*: These can lead to your program grinding to a halt.

Both of these problems are terrible ("causing or likely to cause terror") because they can be very hard to debug in practice: the failures occur nondeterministically and are often more likely to happen under load—which means that they don't show up in unit tests, integration tests, or any other sort of test (Item 30), but they do show up in production.

Rust is a giant step forward, because it completely solves one of these two problems. However, the other still remains, as we shall see.

Data Races

Let's start with the good news, by exploring *data races* and Rust. The precise technical definition of a data race varies from language to language, but we can summarize the key components as follows:

> A data race is defined to occur when two distinct threads access the same memory location, under the following conditions:
>
> - At least one of them is a write.
> - There is no synchronization mechanism that enforces an ordering on the accesses.

Data races in C++

The basics of this are best illustrated with an example. Consider a data structure that tracks a bank account:

```
// C++ code.
class BankAccount {
 public:
  BankAccount() : balance_(0) {}

  int64_t balance() const {
    if (balance_ < 0) {
      std::cerr << "** Oh no, gone overdrawn: " << balance_ << "! **\n";
      std::abort();
    }
    return balance_;
  }
  void deposit(uint32_t amount) {
    balance_ += amount;
  }
  bool withdraw(uint32_t amount) {
    if (balance_ < amount) {
      return false;
    }
    // What if another thread changes `balance_` at this point?
    std::this_thread::sleep_for(std::chrono::milliseconds(500));

    balance_ -= amount;
    return true;
  }

 private:
  int64_t balance_;
};
```

This example is in C++, not Rust, for reasons that will become clear shortly. However, the same general concepts apply in many other (non-Rust) languages—Java, or Go, or Python, etc.

This class works fine in a single-threaded setting, but consider a multithreaded setting:

```
BankAccount account;
account.deposit(1000);

// Start a thread that watches for a low balance and tops up the account.
std::thread payer(pay_in, &account);

// Start 3 threads that each try to repeatedly withdraw money.
std::thread taker(take_out, &account);
std::thread taker2(take_out, &account);
std::thread taker3(take_out, &account);
```

Here several threads are repeatedly trying to withdraw from the account, and there's an additional thread that tops up the account when it runs low:

```
// Constantly monitor the `account` balance and top it up if low.
void pay_in(BankAccount* account) {
  while (true) {
    if (account->balance() < 200) {
      log("[A] Balance running low, deposit 400");
      account->deposit(400);
    }
    // (The infinite loop with sleeps is just for demonstration/simulation
    // purposes.)
    std::this_thread::sleep_for(std::chrono::milliseconds(5));
  }
}

// Repeatedly try to perform withdrawals from the `account`.
void take_out(BankAccount* account) {
  while (true) {
    if (account->withdraw(100)) {
      log("[B] Withdrew 100, balance now " +
          std::to_string(account->balance()));
    } else {
      log("[B] Failed to withdraw 100");
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
  }
}
```

Eventually, things will go wrong:

```
** Oh no, gone overdrawn: -100! **
```

The problem isn't hard to spot, particularly with the helpful comment in the `withdraw()` method: when multiple threads are involved, the value of the balance can change between the check and the modification. However, real-world bugs of this sort are much harder to spot—particularly if the compiler is allowed to perform all kinds of tricks and reorderings of code under the covers (as is the case for C++).

The various `sleep` calls are included in order to artificially raise the chances of this bug being hit and thus detected early; when these problems are encountered in the wild, they're likely to occur rarely and intermittently—making them very hard to debug.

The `BankAccount` class is *thread-compatible*, which means that it can be used in a multithreaded environment as long as the users of the class ensure that access to it is governed by some kind of external synchronization mechanism.

The class can be converted to a *thread-safe* class—meaning that it is safe to use from multiple threads—by adding internal synchronization operations:[1]

```
// C++ code.
class BankAccount {
 public:
  BankAccount() : balance_(0) {}

  int64_t balance() const {
    // Lock mu_ for all of this scope.
    const std::lock_guard<std::mutex> with_lock(mu_);
    if (balance_ < 0) {
      std::cerr << "** Oh no, gone overdrawn: " << balance_ << " **!\n";
      std::abort();
    }
    return balance_;
  }
  void deposit(uint32_t amount) {
    const std::lock_guard<std::mutex> with_lock(mu_);
    balance_ += amount;
  }
  bool withdraw(uint32_t amount) {
    const std::lock_guard<std::mutex> with_lock(mu_);
    if (balance_ < amount) {
      return false;
    }
    balance_ -= amount;
    return true;
  }

 private:
  mutable std::mutex mu_; // protects balance_
  int64_t balance_;
};
```

The internal `balance_` field is now protected by a *mutex* `mu_`: a synchronization object that ensures that only one thread can successfully hold the mutex at a time. A caller can acquire the mutex with a call to `std::mutex::lock()`; the second and subsequent callers of `std::mutex::lock()` will block until the original caller invokes `std::mutex::unlock()`, and then *one* of the blocked threads will unblock and proceed through `std::mutex::lock()`.

All access to the balance now takes place with the mutex held, ensuring that its value is consistent between check and modification. The `std::lock_guard` is also worth highlighting: it's an RAII class (see Item 11) that calls `lock()` on creation and `unlock()` on destruction. This ensures that the mutex is unlocked when the scope exits, reducing the chances of making a mistake around balancing manual `lock()` and `unlock()` calls.

However, the thread safety here is still fragile; all it takes is one erroneous modification to the class:

```
// Add a new C++ method...
void pay_interest(int32_t percent) {
  // ...but forgot about mu_
  int64_t interest = (balance_ * percent) / 100;
  balance_ += interest;
}
```

and the thread safety has been destroyed.[2]

Data races in Rust

For a book about Rust, this Item has covered a lot of C++, so consider a straightforward translation of this class into Rust:

```
pub struct BankAccount {
    balance: i64,
}

impl BankAccount {
    pub fn new() -> Self {
        BankAccount { balance: 0 }
    }
    pub fn balance(&self) -> i64 {
        if self.balance < 0 {
            panic!("** Oh no, gone overdrawn: {}", self.balance);
        }
        self.balance
    }
    pub fn deposit(&mut self, amount: i64) {
        self.balance += amount
    }
    pub fn withdraw(&mut self, amount: i64) -> bool {
        if self.balance < amount {
            return false;
        }
        self.balance -= amount;
        true
    }
}
```

along with the functions that try to pay into or withdraw from an account forever:

```
pub fn pay_in(account: &mut BankAccount) {
    loop {
        if account.balance() < 200 {
            println!("[A] Running low, deposit 400");
            account.deposit(400);
        }
        std::thread::sleep(std::time::Duration::from_millis(5));
    }
}

pub fn take_out(account: &mut BankAccount) {
    loop {
        if account.withdraw(100) {
            println!("[B] Withdrew 100, balance now {}", account.balance());
        } else {
            println!("[B] Failed to withdraw 100");
        }
        std::thread::sleep(std::time::Duration::from_millis(20));
    }
}
```

This works fine in a single-threaded context—even if that thread is not the main thread:

```
{
    let mut account = BankAccount::new();
    let _payer = std::thread::spawn(move || pay_in(&mut account));
    // At the end of the scope, the `_payer` thread is detached
    // and is the sole owner of the `BankAccount`.
}
```

but a naive attempt to use the `BankAccount` across multiple threads:

```
{
    let mut account = BankAccount::new();
    let _taker = std::thread::spawn(move || take_out(&mut account));
    let _payer = std::thread::spawn(move || pay_in(&mut account));
}
```

immediately falls foul of the compiler:

```
error[E0382]: use of moved value: `account`
   --> src/main.rs:102:41
    |
100 | let mut account = BankAccount::new();
    |     ----------- move occurs because `account` has type
    |                 `broken::BankAccount`, which does not implement the
    |                 `Copy` trait
101 | let _taker = std::thread::spawn(move || take_out(&mut account));
    |                                 -------               ------- variable
    |                                 |                         moved due to
    |                                 |                         use in closure
    |                                 |
    |                                 value moved into closure here
102 | let _payer = std::thread::spawn(move || pay_in(&mut account));
    |                                 ^^^^^^^             ------- use occurs
due
    |                                 |                       to use in
closure
    |                                 |
    |                                 value used here after move
```

The rules of the borrow checker (Item 15) make the problem clear: there are two mutable references to the same item, one more than is allowed. The rules of the borrow checker are that you can have a single mutable reference to an item, or multiple (immutable) references, but not both at the same time.

This has a curious resonance with the definition of a data race at the start of this Item: enforcing that there is a single writer, or multiple readers (but never both), means that there can be no data races. By enforcing memory safety, Rust gets thread safety "for free".

As with C++, some kind of synchronization is needed to make this `struct` thread-safe. The most common mechanism is also called `Mutex`, but the Rust version "wraps" the protected data rather than being a standalone object (as in C++):

```
pub struct BankAccount {
    balance: std::sync::Mutex<i64>,
}
```

The `lock()` method on this `Mutex` generic returns a `MutexGuard` object with RAII behavior, like C++'s `std::lock_guard`: the mutex is automatically released at the end of the scope when the guard is `drop`ped. (In contrast to C++, Rust's `Mutex` has no methods that manually acquire or release the mutex, as they would expose developers to the danger of forgetting to keep these calls exactly in sync.)

To be more precise, `lock()` actually returns a `Result` that holds the `MutexGuard`, to cope with the possibility that the `Mutex` has been *poisoned*. Poisoning happens if a thread fails while holding the lock, because this might mean that any mutex-protected invariants can no longer be relied on. In practice, lock poisoning is sufficiently rare (and it's sufficiently desirable that the program terminates when it happens) that it's common to just `.unwrap()` the `Result` (despite the advice in Item 18).

The `MutexGuard` object also acts as a proxy for the data that is enclosed by the `Mutex`, by implementing the `Deref` and `DerefMut` traits (Item 8), allowing it to be used both for read operations:

```
impl BankAccount {
    pub fn balance(&self) -> i64 {
        let balance = *self.balance.lock().unwrap();
        if balance < 0 {
            panic!("** Oh no, gone overdrawn: {}", balance);
        }
        balance
    }
}
```

and for write operations:

```
impl BankAccount {
    // Note: no longer needs `&mut self`.
    pub fn deposit(&self, amount: i64) {
        *self.balance.lock().unwrap() += amount
    }
    pub fn withdraw(&self, amount: i64) -> bool {
        let mut balance = self.balance.lock().unwrap();
        if *balance < amount {
            return false;
        }
        *balance -= amount;
        true
    }
}
```

There's an interesting detail lurking in the signatures of these methods: although they are modifying the balance of the `BankAccount`, the methods now take `&self` rather than `&mut self`. This is inevitable: if multiple threads are going to hold references to the same `BankAccount`, by the rules of the borrow checker, those references had better not be mutable. It's also another instance of the *interior mutability* pattern described in Item 8: borrow checks are effectively moved from compile time to runtime but now with cross-thread synchronization behavior. If a mutable reference already exists, an attempt to get a second blocks until the first reference is dropped.

Wrapping up shared state in a `Mutex` mollifies the borrow checker, but there are still lifetime issues (Item 14) to fix:

```
{
    let account = BankAccount::new();
    let taker = std::thread::spawn(|| take_out(&account));
    let payer = std::thread::spawn(|| pay_in(&account));
    // At the end of the scope, `account` is dropped but
    // the `_taker` and `_payer` threads are detached and
    // still hold (immutable) references to `account`.
}
```

```
error[E0373]: closure may outlive the current function, but it borrows
`account`
              which is owned by the current function
   --> src/main.rs:206:40
    |
206 |     let taker = std::thread::spawn(|| take_out(&account));
    |                                    ^^           ------- `account` is
    |                                    |                    borrowed here
    |                                    |
    |                                    may outlive borrowed value `account`
    |
note: function requires argument type to outlive `'static`
   --> src/main.rs:206:21
    |
206 |     let taker = std::thread::spawn(|| take_out(&account));
    |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
help: to force the closure to take ownership of `account` (and any other
      referenced variables), use the `move` keyword
    |
206 |     let taker = std::thread::spawn(move || take_out(&account));
    |                                    ++++
error[E0373]: closure may outlive the current function, but it borrows
`account`
              which is owned by the current function
   --> src/main.rs:207:40
    |
207 |     let payer = std::thread::spawn(|| pay_in(&account));
    |                                    ^^         ------- `account` is
    |                                    |                  borrowed here
    |                                    |
    |                                    may outlive borrowed value `account`
    |
note: function requires argument type to outlive `'static`
   --> src/main.rs:207:21
    |
207 |     let payer = std::thread::spawn(|| pay_in(&account));
    |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
help: to force the closure to take ownership of `account` (and any other
      referenced variables), use the `move` keyword
    |
207 |     let payer = std::thread::spawn(move || pay_in(&account));
    |                                    ++++
```

The error message makes the problem clear: the `BankAccount` is going to be `drop`ped at the end of the block, but there are two new threads that have a reference to it and that may carry on running afterward. (The compiler's suggestion for how to fix the problem is less helpful—if the `BankAccount` item is moved into the first closure, it will no longer be available for the second closure to receive a reference to it!)

The standard tool for ensuring that an object remains active until all references to it are gone is a reference-counted pointer, and Rust's variant of this for multithreaded use is `std::sync::Arc`:

```
let account = std::sync::Arc::new(BankAccount::new());
account.deposit(1000);

let account2 = account.clone();
let _taker = std::thread::spawn(move || take_out(&account2));

let account3 = account.clone();
let _payer = std::thread::spawn(move || pay_in(&account3));
```

Each thread gets its own copy of the reference-counting pointer, moved into the closure, and the underlying `BankAccount` will be `drop`ped only when the refcount drops to zero. This combination of `Arc<Mutex<T>>` is common in Rust programs that use shared-state parallelism.

Stepping back from the technical details, observe that Rust has entirely avoided the problem of data races that plagues multithreaded programming in other languages. Of course, this good news is restricted to *safe* Rust— `unsafe` code (Item 16) and FFI boundaries in particular (Item 34) may not be data-race free—but it's still a remarkable phenomenon.

Standard marker traits

There are two standard traits that affect the use of Rust objects between threads. Both of these traits are *marker traits* (Item 10) that have no associated methods but have special significance to the compiler in multithreaded scenarios:

- The `Send` trait indicates that items of a type are safe to transfer between threads; ownership of an item of this type can be passed from one thread to another.
- The `Sync` trait indicates that items of a type can be safely accessed by multiple threads, subject to the rules of the borrow checker.

Another way of saying this is to observe that `Send` means `T` can be transferred between threads, and `Sync` means that `&T` can be transferred between threads.

Both of these traits are *auto traits*: the compiler automatically derives them for new types, as long as the constituent parts of the type also implement `Send` / `Sync`.

The majority of safe types implement `Send` and `Sync`, so much so that it's clearer to understand what types *don't* implement these traits (written in the form `impl !Sync for Type`).

A type that doesn't implement `Send` is one that can be used only in a single thread. The canonical example of this is the unsynchronized reference-counting pointer `Rc<T>` (Item 8). The implementation of this type explicitly assumes single-threaded use (for speed); there is no attempt at synchronizing the internal refcount for multithreaded use. As such, transferring an `Rc<T>` between threads is not allowed; use `Arc<T>` (with its additional synchronization overhead) for this case.

A type that doesn't implement `Sync` is one that's not safe to use from multiple threads via *non-*`mut` references (as the borrow checker will ensure there are never multiple `mut` references). The canonical examples of this are the types that provide *interior mutability* in an unsynchronized way, such as `Cell<T>` and `RefCell<T>`. Use `Mutex<T>` or `RwLock<T>` to provide interior mutability in a multithreaded environment.

Raw pointer types like `*const T` and `*mut T` also implement neither `Send` nor `Sync`; see Item 16 and Item 34.

Deadlocks

Now for the bad news. Although Rust has solved the problem of data races (as previously described), it is still susceptible to the *second* terrible problem for multithreaded code with shared state: *deadlocks*.

Consider a simplified multiple-player game server, implemented as a multithreaded application to service many players in parallel. Two core data structures might be a collection of players, indexed by username, and a collection of games in progress, indexed by some unique identifier:

```
struct GameServer {
    // Map player name to player info.
    players: Mutex<HashMap<String, Player>>,
    // Current games, indexed by unique game ID.
    games: Mutex<HashMap<GameId, Game>>,
}
```

Both of these data structures are `Mutex`-protected and so are safe from data races. However, code that manipulates *both* data structures opens up potential problems. A single interaction between the two might work fine:

```
impl GameServer {
    /// Add a new player and join them into a current game.
    fn add_and_join(&self, username: &str, info: Player) -> Option<GameId> {
        // Add the new player.
        let mut players = self.players.lock().unwrap();
        players.insert(username.to_owned(), info);

        // Find a game with available space for them to join.
        let mut games = self.games.lock().unwrap();
        for (id, game) in games.iter_mut() {
            if game.add_player(username) {
                return Some(id.clone());
            }
        }
        None
    }
}
```

However, a second interaction between the two independently locked data structures is where problems start:

```
impl GameServer {
    /// Ban the player identified by `username`, removing them from
    /// any current games.
    fn ban_player(&self, username: &str) {
        // Find all games that the user is in and remove them.
        let mut games = self.games.lock().unwrap();
        games
            .iter_mut()
            .filter(|(_id, g)| g.has_player(username))
            .for_each(|(_id, g)| g.remove_player(username));

        // Wipe them from the user list.
        let mut players = self.players.lock().unwrap();
        players.remove(username);
    }
}
```

To understand the problem, imagine two separate threads using these two methods, where their execution happens in the order shown in Table 3-1.

Table 3-1. Thread deadlock sequence

Thread 1	Thread 2
Enters `add_and_join()` and immediately acquires the `players` lock.	
	Enters `ban_player()` and immediately acquires the `games` lock.
Tries to acquire the `games` lock; this is held by thread 2, so thread 1 blocks.	
	Tries to acquire the `players` lock; this is held by thread 1, so thread 2 blocks.

At this point, the program is *deadlocked*: neither thread will ever progress, nor will any other thread that does anything with either of the two `Mutex` -protected data structures.

The root cause of this is a *lock inversion*: one function acquires the locks in the order `players` then `games` , whereas the other uses the opposite order (`games` then `players`). This is a simple example of a more general problem; the same situation can arise with longer chains of nested locks (thread 1 acquires lock A, then B, then it tries to acquire C; thread 2 acquires C, then tries to acquire A) and across more threads (thread 1 locks A, then B; thread 2 locks B, then C; thread 3 locks C, then A).

A simplistic attempt to solve this problem involves reducing the scope of the locks, so there is no point where both locks are held at the same time:

```
    /// Add a new player and join them into a current game.
    fn add_and_join(&self, username: &str, info: Player) -> Option<GameId> {
        // Add the new player.
        {
            let mut players = self.players.lock().unwrap();
            players.insert(username.to_owned(), info);
        }

        // Find a game with available space for them to join.
        {
            let mut games = self.games.lock().unwrap();
            for (id, game) in games.iter_mut() {
                if game.add_player(username) {
                    return Some(id.clone());
                }
            }
        }
        None
    }
    /// Ban the player identified by `username`, removing them from
    /// any current games.
    fn ban_player(&self, username: &str) {
        // Find all games that the user is in and remove them.
        {
            let mut games = self.games.lock().unwrap();
            games
                .iter_mut()
                .filter(|(_id, g)| g.has_player(username))
                .for_each(|(_id, g)| g.remove_player(username));
        }

        // Wipe them from the user list.
        {
            let mut players = self.players.lock().unwrap();
            players.remove(username);
        }
    }
```

(A better version of this would be to encapsulate the manipulation of the `players` data structure into `add_player()` and `remove_player()` helper methods, to reduce the chances of forgetting to close out a scope.)

This solves the deadlock problem but leaves behind a data consistency problem: the `players` and `games` data structures can get out of sync with each other, given an execution sequence like the one shown in Table 3-2.

Table 3-2. State inconsistency sequence

Thread 1	Thread 2
Enters `add_and_join("Alice")` and adds Alice to the `players` data structure (then releases the `players` lock).	

Thread 1	Thread 2
	Enters `ban_player("Alice")` and removes Alice from all `games` (then releases the `games` lock).
	Removes Alice from the `players` data structure; thread 1 has already released the lock, so this does not block.
Carries on and acquires the `games` lock (already released by thread 2). With the lock held, adds "Alice" to a game in progress.	

At this point, there is a game that includes a player that doesn't exist, according to the `players` data structure!

The heart of the problem is that there are two data structures that need to be kept in sync with each other. The best way to do this is to have a single synchronization primitive that covers both of them:

```
struct GameState {
    players: HashMap<String, Player>,
    games: HashMap<GameId, Game>,
}

struct GameServer {
    state: Mutex<GameState>,
    // ...
}
```

Advice

The most obvious advice for avoiding the problems that arise with shared-state parallelism is simply to avoid shared-state parallelism. The Rust book quotes from the Go language documentation: "Do not communicate by sharing memory; instead, share memory by communicating".

The Go language has *channels* that are suitable for this built into the language; for Rust, equivalent functionality is included in the standard library in the `std::sync::mpsc` module: the `channel()` function returns a `(Sender, Receiver)` pair that allows values of a particular type to be communicated between threads.

If shared-state concurrency can't be avoided, then there are some ways to reduce the chances of writing deadlock-prone code:

- **Put data structures that must be kept consistent with each other under a single lock**.
- **Keep lock scopes small and obvious**; wherever possible, use helper methods that get and set things under the relevant lock.
- **Avoid invoking closures with locks held**; this puts the code at the mercy of whatever closure gets added to the codebase in the future.
- Similarly, **avoid returning a `MutexGuard` to a caller**: it's like handing out a loaded gun, from a deadlock perspective.
- **Include deadlock detection tools** in your CI system (Item 32), such as `no_deadlocks`, ThreadSanitizer, or `parking_lot::deadlock`.
- As a last resort: design, document, test, and police a *locking hierarchy* that describes what lock orderings are allowed/required. This should be a last resort because any strategy that relies on engineers never making a mistake is likely to be doomed to failure in the long term.

More abstractly, multithreaded code is an ideal place to apply the following general advice: prefer code that's so simple that it is obviously not wrong, rather than code that's so complex that it's not obviously wrong.

---

[1] The third category of behavior is *thread-hostile*: code that's dangerous in a multithreaded environment *even if* all access to it is externally synchronized.

[2] The Clang C++ compiler includes a `-Wthread-safety` option, sometimes known as *annotalysis*, that allows data to be annotated with information about which mutexes protect which data, and functions to be annotated with information about the locks they acquire. This gives *compile-time* errors when these invariants are broken, like Rust; however, there is nothing to enforce the use of these annotations in the first place—for example, when a thread-compatible library is used in a multithreaded environment for the first time.

Item 18: Don't panic

> "It looked insanely complicated, and this was one of the reasons why the snug plastic cover it fitted into had the words DON'T PANIC printed on it in large friendly letters." – Douglas Adams

The title of this Item would be more accurately described as **prefer returning a `Result` to using `panic!`** (but **don't panic** is much catchier).

Rust's panic mechanism is primarily designed for unrecoverable bugs in your program, and *by default* it terminates the thread that issues the `panic!`. However, there are alternatives to this default.

In particular, newcomers to Rust who have come from languages that have an exception system (such as Java or C++) sometimes pounce on `std::panic::catch_unwind` as a way to simulate exceptions, because it appears to provide a mechanism for catching panics at a point further up the call stack.

Consider a function that panics on an invalid input:

```
fn divide(a: i64, b: i64) -> i64 {
    if b == 0 {
        panic!("Cowardly refusing to divide by zero!");
    }
    a / b
}
```

Trying to invoke this with an invalid input fails as expected:

```
// Attempt to discover what 0/0 is...
let result = divide(0, 0);
```

```
thread 'main' panicked at 'Cowardly refusing to divide by zero!', main.rs:11:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

A wrapper that uses `catch_unwind` to catch the panic:

```
fn divide_recover(a: i64, b: i64, default: i64) -> i64 {
    let result = std::panic::catch_unwind(|| divide(a, b));
    match result {
        Ok(x) => x,
        Err(_) => default,
    }
}
```

appears to work and to simulate `catch`:

```
let result = divide_recover(0, 0, 42);
println!("result = {result}");
```

```
result = 42
```

Appearances can be deceptive, however. The first problem with this approach is that panics don't always unwind; there is a compiler option (which is also accessible via a *Cargo.toml* profile setting) that shifts panic behavior so that it immediately aborts the process:

```
thread 'main' panicked at 'Cowardly refusing to divide by zero!', main.rs:11:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
/bin/sh: line 1: 29100 Abort trap: 6  cargo run --release
```

This leaves any attempt to simulate exceptions entirely at the mercy of the wider project settings. It's also the case that some target platforms (for example, WebAssembly) *always* abort on panic, regardless of any compiler or project settings.

A more subtle problem that's surfaced by panic handling is *exception safety*: if a panic occurs midway through an operation on a data structure, it removes any guarantees that the data structure has been left in a self-consistent state. Preserving internal invariants in the presence of exceptions has been known to be extremely difficult since the 1990s;[1] this is one of the main reasons why Google (famously) bans the use of exceptions in its C++ code.

Finally, panic propagation also interacts poorly with FFI (foreign function interface) boundaries (Item 34); **use `catch_unwind` to prevent panics in Rust code from propagating to non-Rust calling code** across an FFI boundary.

So what's the alternative to `panic!` for dealing with error conditions? For library code, the best alternative is to make the error someone else's problem, by returning a `Result` with an appropriate error type (Item 4). This allows the library user to make their own decisions about what to do next—which may involve passing the problem on to the next caller in line, via the `?` operator.

The buck has to stop somewhere, and a useful rule of thumb is that it's OK to `panic!` (or to `unwrap()`, `expect()`, etc.) if you have control of `main`; at that point, there's no further caller that the buck could be passed to.

Another sensible use of `panic!`, even in library code, is in situations where it's very rare to encounter errors, and you don't want users to have to litter their code with `.unwrap()` calls.

If an error situation *should* occur only because (say) internal data is corrupted, rather than as a result of invalid inputs, then triggering a `panic!` is legitimate.

It can even be occasionally useful to allow panics that can be triggered by invalid input but where such invalid inputs are out of the ordinary. This works best when the relevant

entrypoints come in pairs:

- An "infallible" version whose signature implies it always succeeds (and which panics if it can't succeed)
- A "fallible" version that returns a `Result`

For the former, Rust's API guidelines suggest that the `panic!` should be documented in a specific section of the inline documentation (Item 27).

The `String::from_utf8_unchecked` and `String::from_utf8` entrypoints in the standard library are an example of the latter (although in this case, the panics are actually deferred to the point where a `String` constructed from invalid input gets used).

Assuming that you are trying to comply with the advice in this Item, there are a few things to bear in mind. The first is that panics can appear in different guises; avoiding `panic!` also involves avoiding the following:

- `unwrap()` and `unwrap_err()`
- `expect()` and `expect_err()`
- `unreachable!()`

Harder to spot are things like these:

- `slice[index]` when the index is out of range
- `x / y` when `y` is zero

The second observation around avoiding panics is that a plan that involves constant vigilance of humans is never a good idea.

However, constant vigilance of machines is another matter: adding a check to your continuous integration (see Item 32) system that spots new, potentially panicking code is much more reliable. A simple version could be a simple grep for the most common panicking entrypoints (as shown previously); a more thorough check could involve additional tooling from the Rust ecosystem (Item 31), such as setting up a build variant that pulls in the `no_panic` crate.

---

[1] Tom Cargill's 1994 article in the *C++ Report* explores just how difficult exception safety is for C++ template code, as does Herb Sutter's Guru of the Week #8 column.

Item 19: Avoid reflection

Programmers coming to Rust from other languages are often used to reaching for reflection as a tool in their toolbox. They can waste a lot of time trying to implement reflection-based designs in Rust, only to discover that what they're attempting can only be done poorly, if at all. This Item hopes to save that time wasted exploring dead ends, by describing what Rust does and doesn't have in the way of reflection, and what can be used instead.

Reflection is the ability of a program to examine itself at runtime. Given an item at runtime, it covers these questions:

- What information can be determined about the item's type?
- What can be done with that information?

Programming languages with full reflection support have extensive answers to these questions. Languages with reflection typically support some or all of the following at runtime, based on the reflection information:

- Determining an item's type
- Exploring its contents
- Modifying its fields
- Invoking its methods

Languages that have this level of reflection support also *tend* to be dynamically typed languages (e.g., Python, Ruby), but there are also some notable statically typed languages that also support reflection, particularly Java and Go.

Rust does not support this type of reflection, which makes the advice to **avoid reflection** easy to follow at this level—it's just not possible. For programmers coming from languages with support for full reflection, this absence may seem like a significant gap at first, but Rust's other features provide alternative ways of solving many of the same problems.

C++ has a more limited form of reflection, known as *run-time type identification* (RTTI). The `typeid` operator returns a unique identifier for every type, for objects of *polymorphic type* (roughly: classes with virtual functions):

- `typeid` : Can recover the concrete class of an object referred to via a base class reference
- `dynamic_cast<T>` : Allows base class references to be converted to derived classes, when it is safe and correct to do so

Rust does not support this RTTI style of reflection either, continuing the theme that the advice of this Item is easy to follow.

Rust does support some features that provide *similar* functionality in the `std::any` module, but they're limited (in ways we will explore) and so best avoided unless no other alternatives are possible.

The first reflection-like feature from `std::any` *looks* like magic at first—a way of determining the name of an item's type. The following example uses a user-defined `tname()` function:

```
let x = 42u32;
let y = vec![3, 4, 2];
println!("x: {} = {}", tname(&x), x);
println!("y: {} = {:?}", tname(&y), y);
```

to show types alongside values:

```
x: u32 = 42
y: alloc::vec::Vec<i32> = [3, 4, 2]
```

The implementation of `tname()` reveals what's up the compiler's sleeve: the function is generic (as per Item 12), and so each invocation of it is actually a different function (`tname::<u32>` or `tname::<Square>`):

```
fn tname<T: ?Sized>(_v: &T) -> &'static str {
    std::any::type_name::<T>()
}
```

The implementation is provided by the `std::any::type_name<T>` library function, which is also generic. This function has access only to *compile-time* information; there is no code run that determines the type at runtime. Returning to the trait object types used in Item 12 demonstrates this:

```
let square = Square::new(1, 2, 2);
let draw: &dyn Draw = &square;
let shape: &dyn Shape = &square;

println!("square: {}", tname(&square));
println!("shape: {}", tname(&shape));
println!("draw: {}", tname(&draw));
```

Only the types of the trait objects are available, not the type (`Square`) of the concrete underlying item:

```
square: reflection::Square
shape: &dyn reflection::Shape
draw: &dyn reflection::Draw
```

The string returned by `type_name` is suitable only for diagnostics—it's explicitly a "best-effort" helper whose contents may change and may not be unique—so **don't attempt to**

parse `type_name` results. If you need a globally unique type identifier, use `TypeId` instead:

```
use std::any::TypeId;

fn type_id<T: 'static + ?Sized>(_v: &T) -> TypeId {
    TypeId::of::<T>()
}


println!("x has {:?}", type_id(&x));
println!("y has {:?}", type_id(&y));


x has TypeId { t: 18349839772473174998 }
y has TypeId { t: 2366424454607613595 }
```

The output is less helpful for humans, but the guarantee of uniqueness means that the result can be used in code. However, it's usually best not to use `TypeId` directly but to use the `std::any::Any` trait instead, because the standard library has additional functionality for working with `Any` instances (described below).

The `Any` trait has a single method `type_id()`, which returns the `TypeId` value for the type that implements the trait. You can't implement this trait yourself, though, because `Any` already comes with a blanket implementation for most arbitrary types `T`:

```
impl<T: 'static + ?Sized> Any for T {
    fn type_id(&self) -> TypeId {
        TypeId::of::<T>()
    }
}
```

The blanket implementation doesn't cover *every* type `T`: the `T: 'static` *lifetime bound* means that if `T` includes any references that have a non-`'static` lifetime, then `TypeId` is not implemented for `T`. This is a deliberate restriction that's imposed because lifetimes aren't fully part of the type: `TypeId::of::<&'a T>` would be the same as `TypeId::of::<&'b T>`, despite the differing lifetimes, increasing the likelihood of confusion and unsound code.

Recall from Item 8 that a trait object is a fat pointer that holds a pointer to the underlying item, together with a pointer to the trait implementation's vtable. For `Any`, the vtable has a single entry, for a `type_id()` method that returns the item's type, as shown in Figure 3-4:

```
let x_any: Box<dyn Any> = Box::new(42u64);
let y_any: Box<dyn Any> = Box::new(Square::new(3, 4, 3));
```

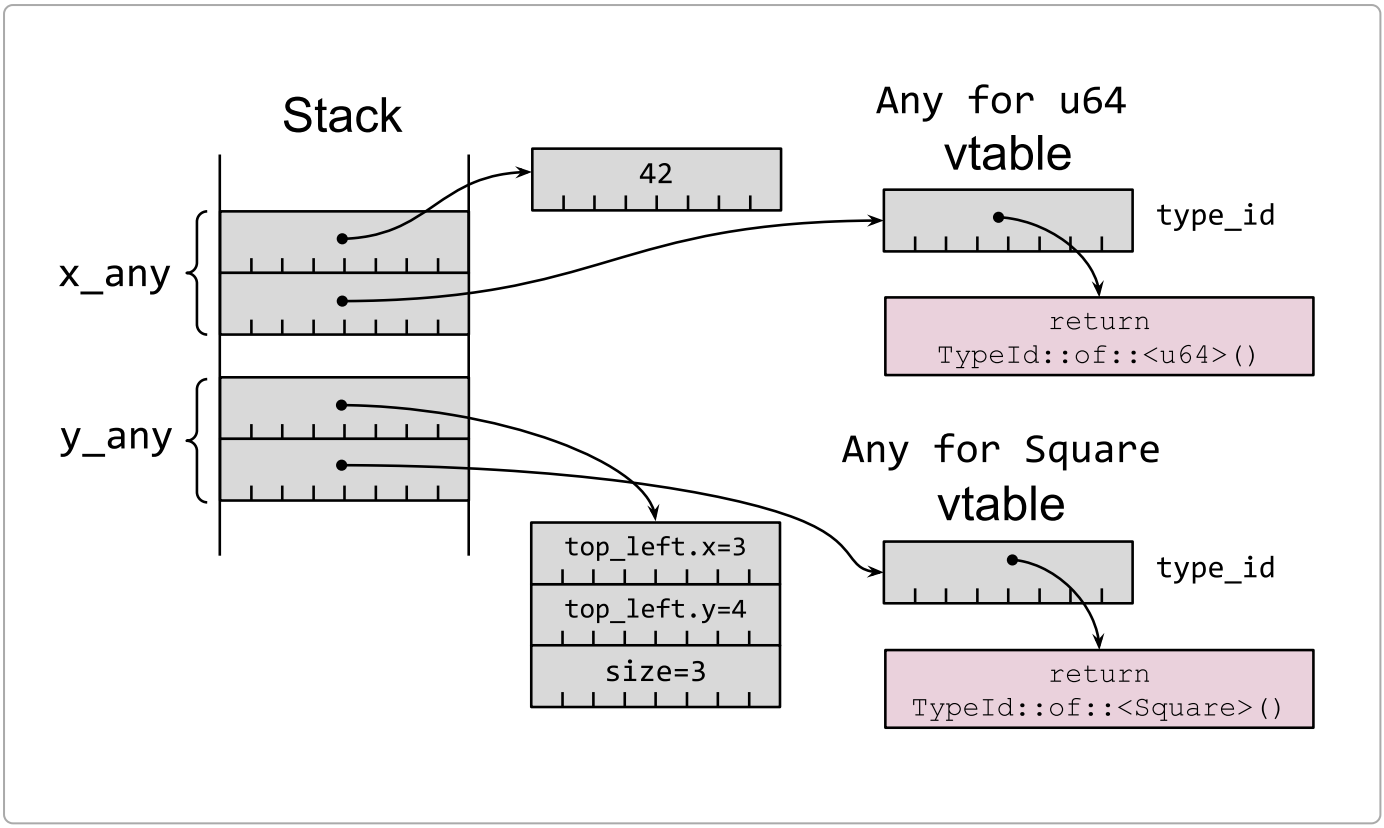


Figure 3-4. `Any` trait objects, each with pointers to concrete items and vtables

Aside from a couple of indirections, a `dyn Any` trait object is effectively a combination of a raw pointer and a type identifier. This means that the standard library can offer some additional generic methods that are defined for a `dyn Any` trait object; these methods are generic over some additional type `T`:

- `is::<T>()`: Indicates whether the trait object's type is equal to some specific other type `T`
- `downcast_ref::<T>()`: Returns a reference to the concrete type `T`, provided that the trait object's type matches `T`
- `downcast_mut::<T>()`: Returns a mutable reference to the concrete type `T`, provided that the trait object's type matches `T`

Observe that the `Any` trait is only approximating reflection functionality: the programmer chooses (at compile time) to explicitly build something (`&dyn Any`) that keeps track of an item's compile-time type as well as its location. The ability to (say) downcast back to the original type is possible only if the overhead of building an `Any` trait object has already happened.

There are comparatively few scenarios where Rust has different compile-time and runtime types associated with an item. Chief among these is *trait objects*: an item of a concrete type `Square` can be coerced into a trait object `dyn Shape` for a trait that the type implements. This coercion builds a fat pointer (object + vtable) from a simple pointer (object/item).

Recall also from Item 12 that Rust's trait objects are not really object-oriented. It's not the case that a `Square` *is-a* `Shape`; it's just that a `Square` implements `Shape`'s interface. The

same is true for trait bounds: a trait bound `Shape: Draw` does *not* mean *is-a*; it just means *also-implements* because the vtable for `Shape` includes the entries for the methods of `Draw`.

For some simple trait bounds:

```
trait Draw: Debug {
    fn bounds(&self) -> Bounds;
}

trait Shape: Draw {
    fn render_in(&self, bounds: Bounds);
    fn render(&self) {
        self.render_in(overlap(SCREEN_BOUNDS, self.bounds()));
    }
}
```

the equivalent trait objects:

```
let square = Square::new(1, 2, 2);
let draw: &dyn Draw = &square;
let shape: &dyn Shape = &square;
```

have a layout with arrows (shown in Figure 3-5; repeated from Item 12) that make the problem clear: given a `dyn Shape` object, there's no immediate way to build a `dyn Draw` trait object, because there's no way to get back to the vtable for `impl Draw for Square` — even though the relevant part of its contents (the address of the `Square::bounds()` method) *is* theoretically recoverable. (This is likely to change in later versions of Rust; see the final section of this Item.)

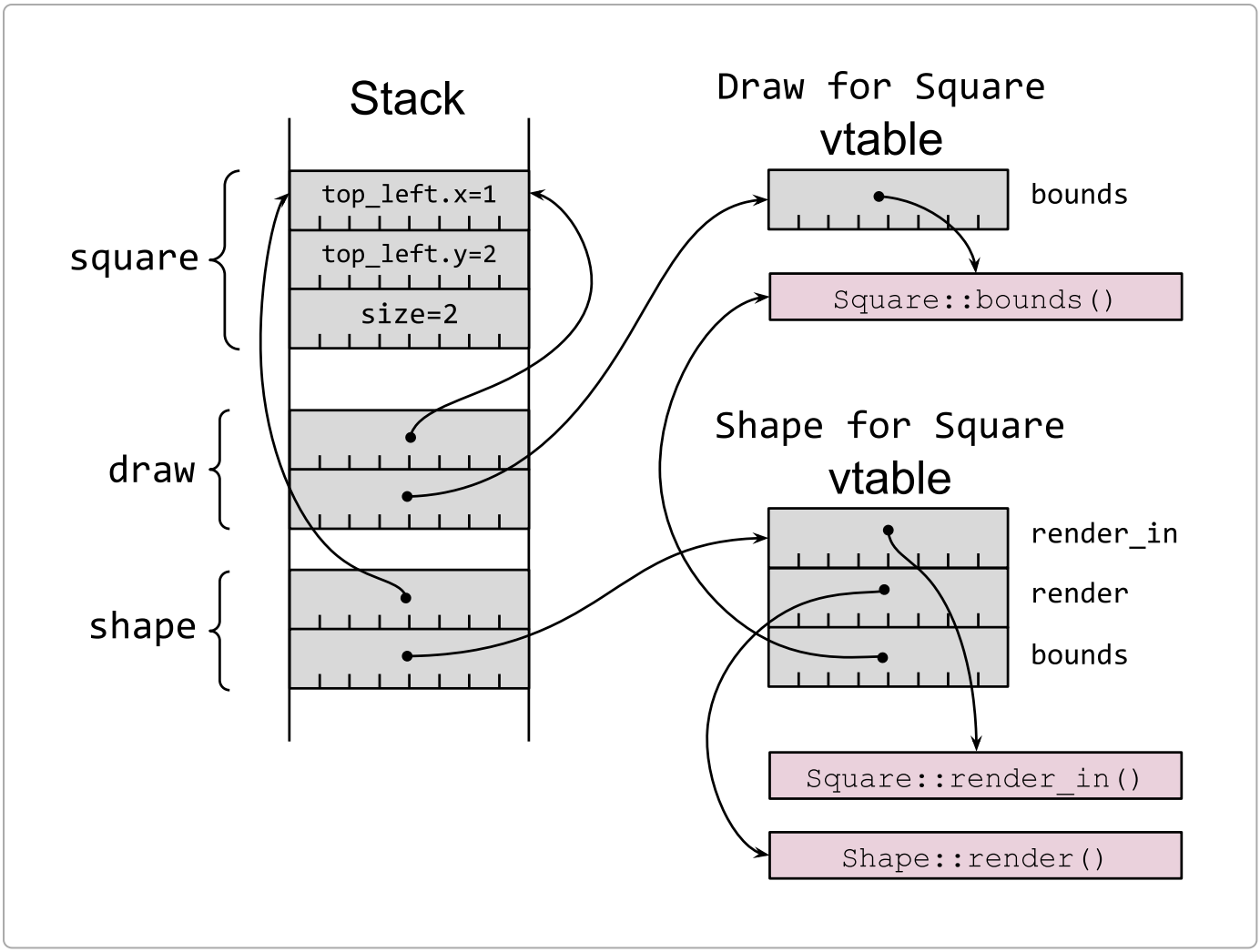


Figure 3-5. Trait objects for trait bounds, with distinct vtables for `Draw` and `Shape`

Comparing this with the previous diagram, it's also clear that an explicitly constructed `&dyn Any` trait object doesn't help. `Any` allows recovery of the original concrete type of the underlying item, but there is no runtime way to see what traits it implements, or to get access to the relevant vtable that might allow creation of a trait object.

So what's available instead?

The primary tool to reach for is trait definitions, and this is in line with advice for other languages—*Effective Java* Item 65 recommends, "Prefer interfaces to reflection". If code needs to rely on the availability of certain behavior for an item, encode that behavior as a trait (Item 2). Even if the desired behavior can't be expressed as a set of method signatures, use marker traits to indicate compliance with the desired behavior—it's safer and more efficient than (say) introspecting the name of a class to check for a particular prefix.

Code that expects trait objects can also be used with objects having backing code that was not available at program link time, because it has been dynamically loaded at runtime (via `dlopen(3)` or equivalent)—which means that monomorphization of a generic (Item 12) isn't possible.

Relatedly, reflection is sometimes also used in other languages to allow multiple incompatible versions of the same dependency library to be loaded into the program at

once, bypassing linkage constraints that There Can Be Only One. This is not needed in Rust, where Cargo already copes with multiple versions of the same library (Item 25).

Finally, macros—especially `derive` macros—can be used to auto-generate ancillary code that understands an item's type at compile time, as a more efficient and more type-safe equivalent to code that parses an item's contents at runtime. Item 28 discusses Rust's macro system.

Upcasting in Future Versions of Rust

The text of this Item was first written in 2021, and remained accurate all the way until the book was being prepared for publication in 2024—at which point a new feature is due to be added to Rust that changes some of the details.

This new "trait upcasting" feature enables upcasts that convert a trait object `dyn T` to a trait object `dyn U`, when `U` is one of `T`'s supertraits (`trait T: U {...}`). The feature is gated on `#![feature(trait_upcasting)]` in advance of its official release, expected to be Rust version 1.76.

For the preceding example, that means a `&dyn Shape` trait object *can* now be converted to a `&dyn Draw` trait object, edging closer to the *is-a* relationship of Liskov substitution. Allowing this conversion has a knock-on effect on the internal details of the vtable implementation, which are likely to become more complex than the versions shown in the preceding diagrams.

However, the central points of this Item are not affected—the `Any` trait has no supertraits, so the ability to upcast adds nothing to its functionality.

Item 20: Avoid the temptation to over-optimize

> "Just because Rust *allows* you to write super cool non-allocating zero-copy algorithms safely, doesn't mean *every* algorithm you write should be super cool, zero-copy and non-allocating." – trentj

Most of the Items in this book are designed to help existing programmers become familiar with Rust and its idioms. This Item, however, is all about a problem that can arise when programmers stray too far in the other direction and become obsessed with exploiting Rust's potential for efficiency—at the expense of usability and maintainability.

Data Structures and Allocation

Like pointers in other languages, Rust's references allow you to reuse data without making copies. Unlike other languages, Rust's rules around reference lifetimes and borrows allow you to reuse data *safely*. However, complying with the borrow checking rules (Item 15) that make this possible can lead to code that's harder to use.

This is particularly relevant for data structures, where you can choose between allocating a fresh copy of something that's stored in the data structure or including a reference to an existing copy of it.

As an example, consider some code that parses a data stream of bytes, extracting data encoded as type-length-value (TLV) structures where data is transferred in the following format:

- One byte describing the type of the value (stored in the `type_code` field here)[1]
- One byte describing the length of the value in bytes (used here to create a slice of the specified length)
- Followed by the specified number of bytes for the value (stored in the `value` field):

```
/// A type-length-value (TLV) from a data stream.
#[derive(Clone, Debug)]
pub struct Tlv<'a> {
    pub type_code: u8,
    pub value: &'a [u8],
}

pub type Error = &'static str; // Some local error type.

/// Extract the next TLV from the `input`, also returning the remaining
/// unprocessed data.
pub fn get_next_tlv(input: &[u8]) -> Result<(Tlv, &[u8]), Error> {
    if input.len() < 2 {
        return Err("too short for a TLV");
    }
    // The TL parts of the TLV are one byte each.
    let type_code = input[0];
    let len = input[1] as usize;
    if 2 + len > input.len() {
        return Err("TLV longer than remaining data");
    }
    let tlv = Tlv {
        type_code,
        // Reference the relevant chunk of input data
        value: &input[2..2 + len],
    };
    Ok((tlv, &input[2 + len..]))
}
```

This `Tlv` data structure is efficient because it holds a reference to the relevant chunk of the input data, without copying any of the data, and Rust's memory safety ensures that the reference is always valid. That's perfect for some scenarios, but things become more awkward if something needs to hang onto an instance of the data structure (as discussed in Item 15).

For example, consider a network server that is receiving messages in the form of TLVs. The received data can be parsed into `Tlv` instances, but the lifetime of those instances will match that of the incoming message—which might be a transient `Vec<u8>` on the heap or might be a buffer somewhere that gets reused for multiple messages.

That induces a problem if the server code ever wants to store an incoming message so that it can be consulted later:

```
pub struct NetworkServer<'a> {
    // ...
    /// Most recent max-size message.
    max_size: Option<Tlv<'a>>,
}

/// Message type code for a set-maximum-size message.
const SET_MAX_SIZE: u8 = 0x01;

impl<'a> NetworkServer<'a> {
    pub fn process(&mut self, mut data: &'a [u8]) -> Result<(), Error> {
        while !data.is_empty() {
            let (tlv, rest) = get_next_tlv(data)?;
            match tlv.type_code {
                SET_MAX_SIZE => {
                    // Save off the most recent `SET_MAX_SIZE` message.
                    self.max_size = Some(tlv);
                }
                // (Deal with other message types)
                // ...
                _ => return Err("unknown message type"),
            }
            data = rest; // Process remaining data on next iteration.
        }
        Ok(())
    }
}
```

This code compiles as is but is effectively impossible to use: the lifetime of the `NetworkServer` has to be smaller than the lifetime of any data that gets fed into its `process()` method. That means that a straightforward processing loop:

```
let mut server = NetworkServer::default();
while !server.done() {
    // Read data into a fresh vector.
    let data: Vec<u8> = read_data_from_socket();
    if let Err(e) = server.process(&data) {
        log::error!("Failed to process data: {:?}", e);
    }
}
```

fails to compile because the lifetime of the ephemeral data gets attached to the longer-lived server:

```
error[E0597]: `data` does not live long enough
   --> src/main.rs:375:40
    |
372 |     while !server.done() {
    |            ------------- borrow later used here
373 |         // Read data into a fresh vector.
374 |         let data: Vec<u8> = read_data_from_socket();
    |             ---- binding `data` declared here
375 |         if let Err(e) = server.process(&data) {
    |                                        ^^^^^ borrowed value does not live
    |                                              long enough
...
378 |     }
    |     - `data` dropped here while still borrowed
```

Switching the code so it reuses a longer-lived buffer doesn't help either:

```
let mut perma_buffer = [0u8; 256];
let mut server = NetworkServer::default(); // lifetime within `perma_buffer`

while !server.done() {
    // Reuse the same buffer for the next load of data.
    read_data_into_buffer(&mut perma_buffer);
    if let Err(e) = server.process(&perma_buffer) {
        log::error!("Failed to process data: {:?}", e);
    }
}
```

This time, the compiler complains that the code is trying to hang on to a reference while also handing out a mutable reference to the same buffer:

```
error[E0502]: cannot borrow `perma_buffer` as mutable because it is also
              borrowed as immutable
   --> src/main.rs:353:31
    |
353 |         read_data_into_buffer(&mut perma_buffer);
    |                               ^^^^^^^^^^^^^^^^^ mutable borrow occurs
here
354 |         if let Err(e) = server.process(&perma_buffer) {
    |                         ------------------------------
    |                         |              |
    |                         |              immutable borrow occurs here
    |                         immutable borrow later used here
```

The core problem is that the `Tlv` structure references transient data—which is fine for transient processing but is fundamentally incompatible with storing state for later. However, if the `Tlv` data structure is converted to own its contents:

```
#[derive(Clone, Debug)]
pub struct Tlv {
    pub type_code: u8,
    pub value: Vec<u8>, // owned heap data
}
```

and the `get_next_tlv()` code is correspondingly tweaked to include an additional call to `.to_vec()`:

```
// ...
let tlv = Tlv {
    type_code,
    // Copy the relevant chunk of data to the heap.
    // The length field in the TLV is a single `u8`,
    // so this copies at most 256 bytes.
    value: input[2..2 + len].to_vec(),
};
```

then the server code has a much easier job. The data-owning `Tlv` structure has no lifetime parameter, so the server data structure doesn't need one either, and both variants of the processing loop work fine.

Who's Afraid of the Big Bad Copy?

One reason why programmers can become overly obsessed with reducing copies is that Rust generally makes copies and allocations explicit. A visible call to a method like `.to_vec()` or `.clone()`, or to a function like `Box::new()`, makes it clear that copying and allocation are occurring. This is in contrast to C++, where it's easy to inadvertently write code that blithely performs allocation under the covers, particularly in a copy-constructor or assignment operator.

Making an allocation or copy operation visible rather than hidden isn't a good reason to optimize it away, especially if that happens at the expense of usability. In many situations, it makes more sense to focus on usability first, and **fine-tune for optimal efficiency only if performance is genuinely a concern**—and if benchmarking (see Item 30) indicates that reducing copies will have a significant impact.

Also, the efficiency of your code is usually important only if it needs to scale up for extensive use. If it turns out that the trade-offs in the code are wrong, and it doesn't cope well when millions of users start to use it—well, that's a nice problem to have.

However, there are a couple of specific points to remember. The first was hidden behind the weasel word *generally* when pointing out that copies are generally visible. The big exception to this is `Copy` types, where the compiler silently makes copies willy-nilly, shifting from move semantics to copy semantics. As such, the advice in Item 10 bears repeating here: don't implement `Copy` unless a bitwise copy is valid and fast. But the converse is true too: do

consider implementing `Copy` if a bitwise copy is valid and fast. For example, `enum` types that don't carry additional data are usually easier to use if they derive `Copy`.

The second point that might be relevant is the potential trade-off with `no_std` use. Item 33 suggests that it's often possible to write code that's `no_std`-compatible with only minor modifications, and code that avoids allocation altogether makes this more straightforward. However, targeting a `no_std` environment that supports heap allocation (via the `alloc` library, also described in Item 33) may give the best balance of usability and `no_std` support.

References and Smart Pointers

> "So very recently, I've consciously tried the experiment of not worrying about the hypothetical perfect code. Instead, I call .clone() when I need to, and use Arc to get local objects into threads and futures more smoothly.
>
> And it feels glorious." – Josh Triplett

Designing a data structure so that it owns its contents can certainly make for better ergonomics, but there are still potential problems if multiple data structures need to make use of the same information. If the data is immutable, then each place having its own copy works fine, but if the information might change (which is very commonly the case), then multiple copies means multiple places that need to be updated, in sync with each other.

Using Rust's smart pointer types helps solve this problem, by allowing the design to shift from a single-owner model to a shared-owner model. The `Rc` (for single-threaded code) and `Arc` (for multithreaded code) smart pointers provide reference counting that supports this shared-ownership model. Continuing with the assumption that mutability is needed, they are typically paired with an inner type that allows interior mutability, independently of Rust's borrow checking rules:

- `RefCell`: For interior mutability in single-threaded code, giving the common `Rc<RefCell<T>>` combination
- `Mutex`: For interior mutability in multithreaded code (as per Item 17), giving the common `Arc<Mutex<T>>` combination

This transition is covered in more detail in the `GuestRegister` example in Item 15, but the point here is that you don't have to treat Rust's smart pointers as a last resort. It's not an admission of defeat if your design uses smart pointers instead of a complex web of interconnected reference lifetimes—**smart pointers can lead to a simpler, more maintainable, and more usable design**.

---

[1] The field can't be named `type` because that's a reserved keyword in Rust. It's possible to work around this restriction by using the raw identifier prefix `r#` (giving a field `r#type: u8`), but it's normally easier just to rename the field.

Dependencies

> "When the Gods wish to punish us, they answer our prayers." – Oscar Wilde

For decades, the idea of code reuse was merely a dream. The idea that code could be written once, packaged into a library, and reused across many different applications was an ideal, realized only for a few standard libraries and for corporate in-house tools.

The growth of the internet and the rise of open source software finally changed that. The first openly accessible repository that held a wide collection of useful libraries, tools, and helpers, all packaged up for easy reuse, was CPAN: the Comprehensive Perl Archive Network, online since 1995. Today, almost every modern language has a comprehensive collection of open source libraries available, housed in a package repository that makes the process of adding a new dependency easy and quick.[1]

However, new problems come along with that ease, convenience, and speed. It's *usually* still easier to reuse existing code than to write it yourself, but there are potential pitfalls and risks that come along with dependencies on someone else's code. This chapter of the book will help you be aware of these.

The focus is specifically on Rust, and with it the use of the `cargo` tool, but many of the concerns, topics, and issues covered apply equally well to other toolchains (and other languages).

---

[1] With the notable exception of C and C++, where package management remains somewhat fragmented.

Item 21: Understand what semantic versioning promises

> "If we acknowledge that SemVer is a lossy estimate and represents only a subset of the possible scope of changes, we can begin to see it as a blunt instrument." – Titus Winters, "Software Engineering at Google (O'Reilly)"

Cargo, Rust's package manager, allows automatic selection of dependencies (Item 25) for Rust code according to *semantic versioning* (semver). A *Cargo.toml* stanza like:

```
[dependencies]
serde = "1.4"
```

indicates to `cargo` what ranges of semver versions are acceptable for this dependency. The official documentation provides the details on specifying precise ranges of acceptable versions, but the following are the most commonly used variants:

- `"1.2.3"` : Specifies that any version that's semver-compatible with 1.2.3 is acceptable
- `"^1.2.3"` : Is another way of specifying the same thing more explicitly
- `"=1.2.3"` : Pins to one particular version, with no substitutes accepted
- `"~1.2.3"` : Allows versions that are semver-compatible with 1.2.3 but only where the last specified component changes (so 1.2.4 is acceptable but 1.3.0 is not)
- `"1.2.*"` : Accepts any version that matches the wildcard

Examples of what these specifications allow are shown in Table 4-1.

Table 4-1. Cargo dependency version specification

Specification	1.2.2	1.2.3	1.2.4	1.3.0	2.0.0
`"1.2.3"`	No	Yes	Yes	Yes	No
`"^1.2.3"`	No	Yes	Yes	Yes	No
`"=1.2.3"`	No	Yes	No	No	No
`"~1.2.3"`	No	Yes	Yes	No	No
`"1.2.*"`	Yes	Yes	Yes	No	No
`"1.*"`	Yes	Yes	Yes	Yes	No
`"*"`	Yes	Yes	Yes	Yes	Yes

When choosing dependency versions, Cargo will generally pick the largest version that's within the combination of all of these semver ranges.

Because semantic versioning is at the heart of `cargo`'s dependency resolution process, this Item explores more details about what semver means.

Semver Essentials

The essentials of semantic versioning are listed in the summary in the semver documentation, reproduced here:

> Given a version number MAJOR.MINOR.PATCH, increment the:
>
> - MAJOR version when you make incompatible API changes
> - MINOR version when you add functionality in a backward compatible manner
> - PATCH version when you make backward compatible bug fixes

---

An important point lurks in the details:

> 3. Once a versioned package has been released, the contents of that version MUST NOT be modified. Any modifications MUST be released as a new version.

Putting this into different words:

- Changing *anything* requires a new patch version.
- *Adding* things to the API in a way that means existing users of the crate still compile and work requires a minor version upgrade.
- *Removing* or *changing* things in the API requires a major version upgrade.

There is one more important codicil to the semver rules:

> 4. Major version zero (0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.

Cargo adapts this last rule slightly, "left-shifting" the earlier rules so that changes in the leftmost non-zero component indicate incompatible changes. This means that 0.2.3 to 0.3.0 can include an incompatible API change, as can 0.0.4 to 0.0.5.

Semver for Crate Authors

> "In theory, theory is the same as practice. In practice, it's not."

As a crate author, the first of these rules is easy to comply with, in theory: if you touch anything, you need a new release. Using Git *tags* to match releases can help with this—by default, a tag is fixed to a particular commit and can be moved only with a manual `--force` option. Crates published to `crates.io` also get automatic policing of this, as the registry will reject a second attempt to publish the same crate version. The main danger for noncompliance is when you notice a mistake *just after* a release has gone out, and you have to resist the temptation to just nip in a fix.

The semver specification covers API compatibility, so if you make a minor change to behavior that doesn't alter the API, then a patch version update should be all that's needed. (However, if your crate is widely depended on, then in practice you may need to be aware of Hyrum's Law: regardless of how minor a change you make to the code, someone out there is likely to depend on the old behavior—even if the API is unchanged.)

The difficult part for crate authors is the latter rules, which require an accurate determination of whether a change is back compatible or not. Some changes are obviously incompatible—removing public entrypoints or types, changing method signatures—and some changes are obviously backward compatible (e.g., adding a new method to a `struct`, or adding a new constant), but there's a lot of gray area left in between.

To help with this, the Cargo book goes into considerable detail as to what is and is not back compatible. Most of these details are unsurprising, but there are a few areas worth highlighting:

- Adding new items is *usually* safe—but may cause clashes if code using the crate already makes use of something that happens to have the same name as the new item.
  - This is a particular danger if the user does a wildcard import from the crate, because all of the crate's items are then automatically in the user's main namespace. Item 23 advises against doing this.
  - Even without a wildcard import, a new trait method (with a default implementation; Item 13) or a new inherent method has a chance of clashing with an existing name.
- Rust's insistence on covering all possibilities means that changing the set of available possibilities can be a breaking change.
  - Performing a `match` on an `enum` must cover all possibilities, so if a crate adds a new `enum` variant, that's a breaking change (unless the `enum` is already marked as `non_exhaustive` —*adding* `non_exhaustive` is also a breaking change).
  - Explicitly creating an instance of a `struct` requires an initial value for all fields, so adding a field to a structure that can be publicly instantiated is a breaking change. Structures that have private fields are OK, because crate users can't explicitly

construct them anyway; a `struct` can also be marked as `non_exhaustive` to prevent external users from performing explicit construction.

- Changing a trait so it is no longer *object safe* (Item 12) is a breaking change; any users that build trait objects for the trait will stop being able to compile their code.
- Adding a new blanket implementation for a trait is a breaking change; any users that already implement the trait will now have two conflicting implementations.
- Changing the *license* of an open source crate is an incompatible change: users of your crate who have strict restrictions on what licenses are acceptable may be broken by the change. **Consider the license to be part of your API**.
- Changing the default features (Item 26) of a crate is potentially a breaking change. Removing a default feature is almost certain to break things (unless the feature was already a no-op); adding a default feature may break things depending on what it enables. **Consider the default feature set to be part of your API**.
- Changing library code so that it uses a new feature of Rust *might* be an incompatible change, because users of your crate who have not yet upgraded their compiler to a version that includes the feature will be broken by the change. However, most Rust crates treat a minimum supported Rust version (MSRV) increase as a *non*-breaking change, so **consider whether the MSRV forms part of your API**.

An obvious corollary of the rules is this: the fewer public items a crate has, the fewer things there are that can induce an incompatible change (Item 22).

However, there's no escaping the fact that comparing all public API items for compatibility from one release to the next is a time-consuming process that is likely to yield only an *approximate* (major/minor/patch) assessment of the level of change, at best. Given that this comparison is a somewhat mechanical process, hopefully tooling (Item 31) will arrive to make the process easier.[1]

If you do need to make an incompatible major version change, it's nice to make life easier for your users by ensuring that the same overall functionality is available after the change, even if the API has radically changed. If possible, the most helpful sequence for your crate users is as follows:

1. Release a minor version update that includes the new version of the API and that marks the older variant as `deprecated`, including an indication of how to migrate.
2. Release a major version update that removes the deprecated parts of the API.

A more subtle point is **make breaking changes breaking**. If your crate is changing its behavior in a way that's actually incompatible for existing users but that *could* reuse the same API: don't. Force a change in types (and a major version bump) to ensure that users can't inadvertently use the new version incorrectly.

For the less tangible parts of your API—such as the MSRV or the license—consider setting up a CI check (Item 32) that detects changes, using tooling (e.g., `cargo-deny`; see Item 25) as needed.

Finally, don't be afraid of version 1.0.0 because it's a commitment that your API is now fixed. Lots of crates fall into the trap of staying at version 0.x forever, but that reduces the already-limited expressivity of semver from three categories (major/minor/patch) to two (effective-major/effective-minor).

Semver for Crate Users

For the user of a crate, the *theoretical* expectations for a new version of a dependency are as follows:

- A new patch version of a dependency crate Should Just Work™.
- A new minor version of a dependency crate Should Just Work™, but the new parts of the API might be worth exploring to see if there are now cleaner or better ways of using the crate. However, if you do use the new parts, you won't be able to revert the dependency back to the old version.
- All bets are off for a new major version of a dependency; chances are that your code will no longer compile, and you'll need to rewrite parts of your code to comply with the new API. Even if your code does still compile, you should **check that your use of the API is still valid after a major version change**, because the constraints and preconditions of the library may have changed.

In practice, even the first two types of change *may* cause unexpected behavior changes, even in code that still compiles fine, due to Hyrum's Law.

As a consequence of these expectations, your dependency specifications will commonly take a form like `"1.4.3"` or `"0.7"`, which includes subsequent compatible versions; **avoid specifying a completely wildcard dependency** like `"*"` or `"0.*"`. A completely wildcard dependency says that *any* version of the dependency, with *any* API, can be used by your crate—which is unlikely to be what you really want. Avoiding wildcards is also a requirement for publishing to `crates.io`; submissions with `"*"` wildcards will be rejected.

However, in the longer term, it's not safe to just ignore major version changes in dependencies. Once a library has had a major version change, the chances are that no further bug fixes—and more importantly, security updates—will be made to the previous major version. A version specification like `"1.4"` will then fall further and further behind as new 2.x releases arrive, with any security problems left unaddressed.

As a result, you need to either accept the risks of being stuck on an old version or **eventually follow major version upgrades to your dependencies**. Tools such as `cargo update` or Dependabot (Item 31) can let you know when updates are available; you can then schedule the upgrade for a time that's convenient for you.

Discussion

Semantic versioning has a cost: every change to a crate has to be assessed against its criteria, to decide the appropriate type of version bump. Semantic versioning is also a blunt tool: at best, it reflects a crate owner's guess as to which of three categories the current release falls into. Not everyone gets it right, not everything is clear-cut about exactly what "right" means, and even if you get it right, there's always a chance you may fall foul of Hyrum's Law.

However, semver is the only game in town for anyone who doesn't have the luxury of working in an environment like Google's highly tested gigantic internal monorepo. As such, understanding its concepts and limitations is necessary for managing dependencies.

---

[1] For example, `cargo-semver-checks` is a tool that attempts to do something along these lines.

Item 22: Minimize visibility

Rust allows elements of the code to either be hidden from or exposed to other parts of the codebase. This Item explores the mechanisms provided for this and suggests advice for where and when they should be used.

Visibility Syntax

Rust's basic unit of visibility is the module. By default, a module's items (types, methods, constants) are *private* and accessible only to code in the same module and its submodules.

Code that needs to be more widely available is marked with the `pub` keyword, making it public to some other scope. For most Rust syntactic features, making the feature `pub` does not automatically expose the contents—the types and functions in a `pub mod` are not public, nor are the fields in a `pub struct`. However, there are a couple of exceptions where applying the visibility to the contents makes sense:

- Making an `enum` public automatically makes the type's variants public too (together with any fields that might be present in those variants).
- Making a `trait` public automatically makes the trait's methods public too.

So a collection of types in a module:

```
pub mod somemodule {
    // Making a `struct` public does not make its fields public.
    #[derive(Debug, Default)]
    pub struct AStruct {
        // By default fields are inaccessible.
        count: i32,
        // Fields have to be explicitly marked `pub` to be visible.
        pub name: String,
    }

    // Likewise, methods on the struct need individual `pub` markers.
    impl AStruct {
        // By default methods are inaccessible.
        fn canonical_name(&self) -> String {
            self.name.to_lowercase()
        }
        // Methods have to be explicitly marked `pub` to be visible.
        pub fn id(&self) -> String {
            format!("{}-{}", self.canonical_name(), self.count)
        }
    }

    // Making an `enum` public also makes all of its variants public.
    #[derive(Debug)]
    pub enum AnEnum {
        VariantOne,
        // Fields in variants are also made public.
        VariantTwo(u32),
        VariantThree { name: String, value: String },
    }

    // Making a `trait` public also makes all of its methods public.
    pub trait DoSomething {
        fn do_something(&self, arg: i32);
    }
}
```

allows access to `pub` things and the exceptions previously mentioned:

```
use somemodule::*;

let mut s = AStruct::default();
s.name = "Miles".to_string();
println!("s = {:?}, name='{}', id={}", s, s.name, s.id());

let e = AnEnum::VariantTwo(42);
println!("e = {e:?}");

#[derive(Default)]
pub struct DoesSomething;
impl DoSomething for DoesSomething {
    fn do_something(&self, _arg: i32) {}
}

let d = DoesSomething::default();
d.do_something(42);
```

but non-`pub` things are generally inaccessible:

```
let mut s = AStruct::default();
s.name = "Miles".to_string();
println!("(inaccessible) s.count={}", s.count);
println!("(inaccessible) s.canonical_name()={}", s.canonical_name());


error[E0616]: field `count` of struct `somemodule::AStruct` is private
   --> src/main.rs:230:45
    |
230 |     println!("(inaccessible) s.count={}", s.count);
    |                                             ^^^^^ private field
error[E0624]: method `canonical_name` is private
   --> src/main.rs:231:56
    |
86  |         fn canonical_name(&self) -> String {
    |         ---------------------------------- private method defined here
...
231 |     println!("(inaccessible) s.canonical_name()={}", s.canonical_name());
    |                                                       private method ^^^^^^^^^^^^^^
Some errors have detailed explanations: E0616, E0624.
For more information about an error, try `rustc --explain E0616`.
```

The most common visibility marker is the bare `pub` keyword, which makes the item visible to anything that's able to see the module it's in. That last detail is important: if a `somecrate::somemodule` module isn't visible to other code in the first place, anything that's `pub` inside it is still not visible.

However, there are also some more-specific variants of `pub` that allow the scope of the visibility to be constrained. In descending order of usefulness, these are as follows:

- `pub(crate)`: Accessible anywhere within the owning crate. This is particularly useful for crate-wide internal helper functions that should not be exposed to external crate users.

- `pub(super)`: Accessible to the parent module of the current module and its submodules. This is occasionally useful for selectively increasing visibility in a crate that has a deep module structure. It's also the effective visibility level for modules: a plain `mod mymodule` is visible to its parent module or crate and the corresponding submodules.
- `pub(in <path>)`: Accessible to code in `<path>`, which has to be a description of some ancestor module of the current module. This can occasionally be useful for organizing source code, because it allows subsets of functionality to be moved into submodules that aren't necessarily visible in the public API. For example, the Rust standard library consolidates all of the iterator adapters into an internal `std::iter::adapters` submodule and has the following:
  - A `pub(in crate::iter)` visibility marker on all of the required adapter methods in submodules, such as `std::iter::adapters::map::Map::new`.
  - A `pub use` of all of the `adapters::` types in the outer `std::iter` module.
- `pub(self)`: Equivalent to `pub(in self)`, which is equivalent to not being `pub`. Uses for this are very obscure, such as reducing the number of special cases needed in code-generation macros.

The Rust compiler will warn you if you have a code item that is private to the module but not used within that module (and its submodules):

```
pub mod anothermodule {
    // Private function that is not used within its module.
    fn inaccessible_fn(x: i32) -> i32 {
        x + 3
    }
}
```

Although the warning indicates that the code is "never used" in its owning module, in practice this warning often indicates that code *can't* be used from outside the module, because the visibility restrictions don't allow it:

```
warning: function `inaccessible_fn` is never used
  --> src/main.rs:56:8
   |
56 |     fn inaccessible_fn(x: i32) -> i32 {
   |        ^^^^^^^^^^^^^^^
   |
   = note: `#[warn(dead_code)]` on by default
```

Visibility Semantics

Separate from the question of *how* to increase visibility is the question of *when* to do so. The generally accepted answer to this is *as little as possible*, at least for any code that may possibly get used and reused in the future.

The first reason for this advice is that visibility changes can be hard to undo. Once a crate item is public, it can't be made private again without breaking any code that uses the crate, thus necessitating a major version bump (Item 21). The converse is not true: moving a private item to be public generally needs only a minor version bump and leaves crate users unaffected—read through Rust's API compatibility guidelines and notice how many are relevant only if there are `pub` items in play.

A more important—but more subtle—reason to prefer privacy is that it keeps your options open. The more things that are exposed, the more things there are that need to stay fixed for the future (absent an incompatible change). If you expose the internal implementation details of a data structure, a putative future change to use a more efficient algorithm becomes a breaking change. If you expose internal helper functions, it's inevitable that some external code will come to depend on the exact details of those functions.

Of course, this is a concern only for library code that potentially has multiple users and a long lifespan. But nothing is as permanent as a temporary solution, and so it's a good habit to fall into.

It's also worth observing that this advice to restrict visibility is by no means unique to this Item or to Rust:

- The Rust API guidelines include this advice:
  - Structs should have private fields.
- *Effective Java*, 3rd edition, (Addison-Wesley Professional) has the following:
  - Item 15: Minimize the accessibility of classes and members.
  - Item 16: In public classes, use accessor methods, not public fields.
- *Effective C++* by Scott Meyers (Addison-Wesley Professional) has the following in its second edition:
  - Item 18: Strive for class interfaces that are complete and *minimal* (my italics).
  - Item 20: Avoid data members in the public interface.

Item 23: Avoid wildcard imports

Rust's `use` statement pulls in a named item from another crate or module and makes that name available for use in the local module's code without qualification. A *wildcard import* (or *glob import*) of the form `use somecrate::module::*` says that *every* public symbol from that module should be added to the local namespace.

As described in Item 21, an external crate may add new items to its API as part of a minor version upgrade; this is considered a backward-compatible change.

The combination of these two observations raises the worry that a nonbreaking change to a dependency might break your code: what happens if the dependency adds a new symbol that clashes with a name you're already using?

At the simplest level, this turns out not to be a problem: the names in a wildcard import are treated as being lower priority, so any matching names that are in your code take precedence:

```
use bytes::*;

// Local `Bytes` type does not clash with `bytes::Bytes`.
struct Bytes(Vec<u8>);
```

Unfortunately, there are still cases where clashes can occur. For example, consider the case when the dependency adds a new trait and implements it for some type:

```
trait BytesLeft {
    // Name clashes with the `remaining` method on the wildcard-imported
    // `bytes::Buf` trait.
    fn remaining(&self) -> usize;
}

impl BytesLeft for &[u8] {
    // Implementation clashes with `impl bytes::Buf for &[u8]`.
    fn remaining(&self) -> usize {
        self.len()
    }
}
```

If any method names from the new trait clash with existing method names that apply to the type, then the compiler can no longer unambiguously figure out which method is intended:

```
let arr = [1u8, 2u8, 3u8];
let v = &arr[1..];

assert_eq!(v.remaining(), 2);
```

as indicated by the compile-time error:

```
error[E0034]: multiple applicable items in scope
  --> src/main.rs:40:18
   |
40 |     assert_eq!(v.remaining(), 2);
   |                  ^^^^^^^^^ multiple `remaining` found
   |
note: candidate #1 is defined in an impl of the trait `BytesLeft` for the
      type `&[u8]`
  --> src/main.rs:18:5
   |
18 |     fn remaining(&self) -> usize {
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   = note: candidate #2 is defined in an impl of the trait `bytes::Buf` for the
           type `&[u8]`
help: disambiguate the method for candidate #1
   |
40 |     assert_eq!(BytesLeft::remaining(&v), 2);
   |                ~~~~~~~~~~~~~~~~~~~~~
help: disambiguate the method for candidate #2
   |
40 |     assert_eq!(bytes::Buf::remaining(&v), 2);
   |                ~~~~~~~~~~~~~~~~~~~~~~
```

As a result, you should **avoid wildcard imports from crates that you don't control**.

If you do control the source of the wildcard import, then the previously mentioned concerns disappear. For example, it's common for a `test` module to do `use super::*;`. It's also possible for crates that use modules primarily as a way of dividing up code to have a wildcard import from an internal module:

```
mod thing;
pub use thing::*;
```

However, there's another common exception where wildcard imports make sense. Some crates have a convention that common items for the crate are re-exported from a *prelude* module, which is explicitly intended to be wildcard imported:

```
use thing::prelude::*;
```

Although in theory the same concerns apply in this case, in practice such a prelude module is likely to be carefully curated, and higher convenience may outweigh a small risk of future problems.

Finally, if you don't follow the advice in this Item, **consider pinning dependencies that you wildcard import to a precise version** (see Item 21) so that minor version upgrades of the dependency aren't automatically allowed.

Item 24: Re-export dependencies whose types appear in your API

The title of this Item is a little convoluted, but working through an example will make things clearer.[1]

Item 25 describes how `cargo` supports different versions of the same library crate being linked into a single binary, in a transparent manner. Consider a binary that uses the `rand` crate—more specifically, one that uses some 0.8 version of the crate:

```
# Cargo.toml file for a top-level binary crate.

[dependencies]
# The binary depends on the `rand` crate from crates.io
rand = "=0.8.5"

# It also depends on some other crate (`dep-lib`).
dep-lib = "0.1.0"
```

```
let mut rng = rand::thread_rng(); // rand 0.8
let max: usize = rng.gen_range(5..10);
let choice = dep_lib::pick_number(max);
```

The final line of code also uses a notional `dep-lib` crate as another dependency. This crate might be another crate from `crates.io`, or it could be a local crate that is located via Cargo's `path` mechanism.

This `dep-lib` crate internally uses a 0.7 version of the `rand` crate:

```
# Cargo.toml file for the `dep-lib` library crate.

[dependencies]
# The library depends on the `rand` crate from crates.io
rand = "=0.7.3"
```

```
//! The `dep-lib` crate provides number picking functionality.
use rand::Rng;

/// Pick a number between 0 and n (exclusive).
pub fn pick_number(n: usize) -> usize {
    rand::thread_rng().gen_range(0, n)
}
```

An eagle-eyed reader might notice a difference between the two code examples:

- In version 0.7.x of `rand` (as used by the `dep-lib` library crate), the `rand::gen_range()` method takes two parameters, `low` and `high`.
- In version 0.8.x of `rand` (as used by the binary crate), the `rand::gen_range()` method takes a single parameter `range`.

This is not a back-compatible change, and so `rand` has increased its leftmost version component accordingly, as required by semantic versioning (Item 21). Nevertheless, the binary that combines the two incompatible versions works just fine— `cargo` sorts everything out.

However, things get a lot more awkward if the `dep-lib` library crate's API exposes a type from its dependency, making that dependency a *public dependency*.

For example, suppose that the `dep-lib` entrypoint involves an `Rng` item—but specifically a version-0.7 `Rng` item:

```
/// Pick a number between 0 and n (exclusive) using
/// the provided `Rng` instance.
pub fn pick_number_with<R: Rng>(rng: &mut R, n: usize) -> usize {
    rng.gen_range(0, n) // Method from the 0.7.x version of Rng
}
```

As an aside, **think carefully before using another crate's types in your API**: it intimately ties your crate to that of the dependency. For example, a major version bump for the dependency (Item 21) will automatically require a major version bump for your crate too.

In this case, `rand` is a semi-standard crate that is widely used and pulls in only a small number of dependencies of its own (Item 25), so including its types in the crate API is probably fine on balance.

Returning to the example, an attempt to use this entrypoint from the top-level binary fails:

```
let mut rng = rand::thread_rng();
let max: usize = rng.gen_range(5..10);
let choice = dep_lib::pick_number_with(&mut rng, max);
```

Unusually for Rust, the compiler error message isn't very helpful:

```
error[E0277]: the trait bound `ThreadRng: rand_core::RngCore` is not satisfied
  --> src/main.rs:22:44
   |
22 |     let choice = dep_lib::pick_number_with(&mut rng, max);
   |                  ------------------------- ^^^^^^^^ the trait
   |                  |                          `rand_core::RngCore` is not
   |                  |                           implemented for `ThreadRng`
   |                  |
   |                  required by a bound introduced by this call
   |
   = help: the following other types implement trait `rand_core::RngCore`:
             &'a mut R
```

Investigating the types involved leads to confusion because the relevant traits do *appear* to be implemented—but the caller actually implements a (notional) `RngCore_v0_8_5` and the library is expecting an implementation of `RngCore_v0_7_3`.

Once you've finally deciphered the error message and realized that the version clash is the underlying cause, how can you fix it?[2] The key observation is to realize that while the binary can't *directly* use two different versions of the same crate, it can do so *indirectly* (as in the original example shown previously).

From the perspective of the binary author, the problem could be worked around by adding an intermediate wrapper crate that hides the naked use of `rand` v0.7 types. A wrapper crate is distinct from the binary crate and so is allowed to depend on `rand` v0.7 separately from the binary crate's dependency on `rand` v0.8.

This is awkward, and a much better approach is available to the author of the library crate. It can make life easier for its users by explicitly re-exporting either of the following:

- The types involved in the API
- The entire dependency crate

For this example, the latter approach works best: as well as making the version 0.7 `Rng` and `RngCore` types available, it also makes available the methods (like `thread_rng()`) that construct instances of the type:

```
// Re-export the version of `rand` used in this crate's API.
pub use rand;
```

The calling code now has a different way to directly refer to version 0.7 of `rand`, as `dep_lib::rand`:

```
let mut prev_rng = dep_lib::rand::thread_rng(); // v0.7 Rng instance
let choice = dep_lib::pick_number_with(&mut prev_rng, max);
```

With this example in mind, the advice given in the title of the Item should now be a little less obscure: **re-export dependencies whose types appear in your API**.

[1] This example (and indeed Item) is inspired by the approach used in the RustCrypto crates.

[2] This kind of error can even appear when the dependency graph includes two alternatives for a crate with the *same version*, when something in the build graph uses the `path` field to specify a local directory instead of a `crates.io` location.

Item 25: Manage your dependency graph

Like most modern programming languages, Rust makes it easy to pull in external libraries, in the form of *crates*. Most nontrivial Rust programs use external crates, and those crates may themselves have additional dependencies, forming a *dependency graph* for the program as a whole.

By default, Cargo will download any crates named in the `[dependencies]` section of your *Cargo.toml* file from `crates.io` and find versions of those crates that match the requirements configured in *Cargo.toml*.

A few subtleties lurk underneath this simple statement. The first thing to notice is that crate names from `crates.io` form a single flat namespace—and this global namespace also overlaps with the names of *features* in a crate (see Item 26).[1]

If you're planning on *publishing* a crate on `crates.io`, be aware that names are generally allocated on a first-come, first-served basis; so you may find that your preferred name for a public crate is already taken. However, name-squatting—reserving a crate name by preregistering an empty crate—is frowned upon, unless you really are going to release code in the near future.

As a minor wrinkle, there's also a slight difference between what's allowed as a crate name in the crates namespace and what's allowed as an identifier in code: a crate can be named `some-crate`, but it will appear in code as `some_crate` (with an underscore). To put it another way: if you see `some_crate` in code, the corresponding crate name may be either `some-crate` or `some_crate`.

The second subtlety to understand is that Cargo allows multiple semver-incompatible versions of the same crate to be present in the build. This can seem surprising to begin with, because each *Cargo.toml* file can have only a single version of any given dependency, but the situation frequently arises with indirect dependencies: your crate depends on `some-crate` version 3.x but also depends on `older-crate`, which in turn depends on `some-crate` version 1.x.

This can lead to confusion if the dependency is exposed in some way rather than just being used internally (Item 24)—the compiler will treat the two versions as being distinct crates, but its error messages won't necessarily make that clear.

Allowing multiple versions of a crate can also go wrong if the crate includes C/C++ code accessed via Rust's FFI mechanisms (Item 34). The Rust toolchain can internally disambiguate distinct versions of Rust code, but any included C/C++ code is subject to the *one definition rule*: there can be only a single version of any function, constant, or global variable.

There are restrictions on Cargo's multiple-version support. Cargo does *not* allow multiple versions of the same crate within a semver-compatible range (Item 21):

- `some-crate` 1.2 and `some-crate` 3.1 can coexist
- `some-crate` 1.2 and `some-crate` 1.3 cannot

Cargo also extends the semantic versioning rules for pre-1.0 crates so that the first non-zero subversion counts like a major version, so a similar constraint applies:

- `other-crate` 0.1.2 and `other-crate` 0.2.0 can coexist
- `other-crate` 0.1.2 and `other-crate` 0.1.4 cannot

Cargo's version selection algorithm does the job of figuring out what versions to include. Each *Cargo.toml* dependency line specifies an acceptable range of versions, according to semantic versioning rules, and Cargo takes this into account when the same crate appears in multiple places in the dependency graph. If the acceptable ranges overlap and are semver-compatible, then Cargo will (by default) pick the most recent version of the crate within the overlap. If there is no semver-compatible overlap, then Cargo will build multiple copies of the dependency at different versions.

Once Cargo has picked acceptable versions for all dependencies, its choices are recorded in the *Cargo.lock* file. Subsequent builds will then reuse the choices encoded in *Cargo.lock* so that the build is stable and no new downloads are needed.

This leaves you with a choice: should you commit your *Cargo.lock* files into version control or not? The advice from the Cargo developers is as follows:

- Things that produce a final product, namely applications and binaries, should commit *Cargo.lock* to ensure a deterministic build.
- Library crates should *not* commit a *Cargo.lock* file, because it's irrelevant to any downstream consumers of the library—they will have their own *Cargo.lock* file; **be aware that the Cargo.lock file for a library crate is ignored by library users**.

Even for a library crate, it can be helpful to have a checked-in *Cargo.lock* file to ensure that regular builds and CI (Item 32) don't have a moving target. Although the promises of semantic versioning (Item 21) should prevent failures in theory, mistakes happen in practice, and it's frustrating to have builds that fail because someone somewhere recently changed a dependency of a dependency.

However, **if you version-control Cargo.lock, set up a process to handle upgrades** (such as GitHub's Dependabot). If you don't, your dependencies will stay pinned to versions that get older, outdated, and potentially insecure.

Pinning versions with a checked-in *Cargo.lock* file doesn't avoid the pain of handling dependency upgrades, but it does mean that you can handle them at a time of your own choosing, rather than immediately when the upstream crate changes. There's also some fraction of dependency-upgrade problems that go away on their own: a crate that's released

with a problem often gets a second, fixed, version released in a short space of time, and a batched upgrade process might see only the latter version.

The third subtlety of Cargo's resolution process to be aware of is *feature unification*: the features that get activated for a dependent crate are the *union* of the features selected by different places in the dependency graph; see Item 26 for more details.

Version Specification

The version specification clause for a dependency defines a range of allowed versions, according to the rules explained in the Cargo book:

- *Avoid a too-specific version dependency*: Pinning to a specific version (`"=1.2.3"`) is *usually* a bad idea: you don't see newer versions (potentially including security fixes), and you dramatically narrow the potential overlap range with other crates in the graph that rely on the same dependency (recall that Cargo allows only a single version of a crate to be used within a semver-compatible range). If you want to ensure that your builds use a consistent set of dependencies, the *Cargo.lock* file is the tool for the job.
- *Avoid a too-general version dependency*: It's *possible* to specify a version dependency (`"*"`) that allows *any* version of the dependency to be used, but it's a bad idea. If the dependency releases a new major version of the crate that completely changes every aspect of its API, it's unlikely that your code will still work after a `cargo update` pulls in the new version.

The most common Goldilocks specification—not too precise, not too vague—is to allow semver-compatible versions (`"1"`) of a crate, possibly with a specific minimum version that includes a feature or fix that you require (`"1.4.23"`).

Both of these version specifications make use of Cargo's default behavior, which is to allow versions that are semver-compatible with the specified version. You can make this more explicit by adding a caret:

- A version of `"1"` is equivalent to `"^1"` , which allows all 1.x versions (and so is also equivalent to `"1.*"`).
- A version of `"1.4.23"` is equivalent to `"^1.4.23"` , which allows any 1.x versions that are larger than 1.4.23.

Solving Problems with Tooling

Item 31 recommends that you take advantage of the range of tools that are available within the Rust ecosystem. This section describes some dependency graph problems where tools can help.

The compiler will tell you pretty quickly if you use a dependency in your code but don't include that dependency in *Cargo.toml*. But what about the other way around? If there's a dependency in *Cargo.toml* that you *don't* use in your code—or more likely, *no longer* use in your code—then Cargo will go on with its business. The `cargo-udeps` tool is designed to solve exactly this problem: it warns you when your *Cargo.toml* includes an unused dependency ("udep").

A more versatile tool is `cargo-deny`, which analyzes your dependency graph to detect a variety of potential problems across the full set of transitive dependencies:

- Dependencies that have known security problems in the included version
- Dependencies that are covered by an unacceptable license
- Dependencies that are just unacceptable
- Dependencies that are included in multiple different versions across the dependency tree

Each of these features can be configured and can have exceptions specified. The exception mechanism is usually needed for larger projects, particularly the multiple-version warning: as the dependency graph grows, so does the chance of transitively depending on different versions of the same crate. It's worth trying to reduce these duplicates where possible—for binary-size and compilation-time reasons if nothing else—but sometimes there is no possible combination of dependency versions that can avoid a duplicate.

These tools can be run as a one-off, but it's better to ensure they're executed regularly and reliably by including them in your CI system (Item 32). This helps to catch newly introduced problems—including problems that may have been introduced outside of your code, in an upstream dependency (for example, a newly reported vulnerability).

If one of these tools does report a problem, it can be difficult to figure out exactly where in the dependency graph the problem arises. The `cargo tree` command that's included with `cargo` helps here, as it shows the dependency graph as a tree structure:

```
dep-graph v0.1.0
├── dep-lib v0.1.0
│   └── rand v0.7.3
│       ├── getrandom v0.1.16
│       │   ├── cfg-if v1.0.0
│       │   └── libc v0.2.94
│       ├── libc v0.2.94
│       ├── rand_chacha v0.2.2
│       │   ├── ppv-lite86 v0.2.10
│       │   └── rand_core v0.5.1
│       │       └── getrandom v0.1.16 (*)
│       └── rand_core v0.5.1 (*)
└── rand v0.8.3
    ├── libc v0.2.94
    ├── rand_chacha v0.3.0
    │   ├── ppv-lite86 v0.2.10
    │   └── rand_core v0.6.2
    │       └── getrandom v0.2.3
    │           ├── cfg-if v1.0.0
    │           └── libc v0.2.94
    └── rand_core v0.6.2 (*)
```

`cargo tree` includes a variety of options that can help to solve specific problems, such as these:

- `--invert` : Shows what depends *on* a specific package, helping you to focus on a particular problematic dependency
- `--edges features` : Shows what crate features are activated by a dependency link, which helps you figure out what's going on with feature unification (Item 26)
- `--duplicates` : Shows crates that have multiple versions present in the dependency graph

What to Depend On

The previous sections have covered the more mechanical aspect of working with dependencies, but there's a more philosophical (and therefore harder-to-answer) question: when should you take on a dependency?

Most of the time, there's not much of a decision involved: if you need the functionality of a crate, you need that function, and the only alternative would be to write it yourself.[2]

But every new dependency has a cost, partly in terms of longer builds and bigger binaries but mostly in terms of the developer effort involved in fixing problems with dependencies when they arise.

The bigger your dependency graph, the more likely you are to be exposed to these kinds of problems. The Rust crate ecosystem is just as vulnerable to accidental dependency problems as other package ecosystems, where history has shown that one developer

removing a package, or a team fixing the licensing for their package can have widespread knock-on effects.

More worrying still are supply chain attacks, where a malicious actor deliberately tries to subvert commonly used dependencies, whether by typo-squatting, hijacking a maintainer's account, or other more sophisticated attacks.

This kind of attack doesn't just affect your compiled code—be aware that a dependency can run arbitrary code at *build* time, via `build.rs` scripts or procedural macros (Item 28). That means that a compromised dependency could end up running a cryptocurrency miner as part of your CI system!

So for dependencies that are more "cosmetic", it's sometimes worth considering whether adding the dependency is worth the cost.

The answer is usually "yes", though; in the end, the amount of time spent dealing with dependency problems ends up being much less than the time it would take to write equivalent functionality from scratch.

Things to Remember

- Crate names on `crates.io` form a single flat namespace (which is shared with feature names).
- Crate names can include a hyphen, but it will appear as an underscore in code.
- Cargo supports multiple versions of the same crate in the dependency graph, but only if they are of different semver-incompatible versions. This can go wrong for crates that include FFI code.
- Prefer to allow semver-compatible versions of dependencies (`"1"` , or `"1.4.23"` to include a minimum version).
- Use *Cargo.lock* files to ensure your builds are repeatable, but remember that the *Cargo.lock* file does not ship with a published crate.
- Use tooling (`cargo tree` , `cargo deny` , `cargo udep` , ...) to help find and fix dependency problems.
- Understand that pulling in dependencies saves you writing code but doesn't come for free.

---

[1] It's also possible to configure an alternate registry of crates (for example, an internal corporate registry). Each dependency entry in *Cargo.toml* can then use the `registry` key to indicate which registry a dependency should be sourced from.

[2] If you are targeting a `no_std` environment, this choice may be made for you: many crates are not compatible with `no_std` , particularly if `alloc` is also unavailable (Item 33).

Item 26: Be wary of `feature` creep

Rust allows the same codebase to support a variety of different configurations via Cargo's *feature* mechanism, which is built on top of a lower-level mechanism for conditional compilation. However, the feature mechanism has a few subtleties to be aware of, which this Item explores.

Conditional Compilation

Rust includes support for conditional compilation, which is controlled by `cfg` (and `cfg_attr`) attributes. These attributes govern whether the thing—function, line, block, etc.—that they are attached to is included in the compiled source code or not (which is in contrast to C/C++'s line-based preprocessor). The conditional inclusion is controlled by configuration options that are either plain names (e.g., `test`) or pairs of names and values (e.g., `panic = "abort"`).

Note that the name/value variants of config options are multivalued—it's possible to set more than one value for the same name:

```
// Build with `RUSTFLAGS` set to:
//   '--cfg myname="a" --cfg myname="b"'
#[cfg(myname = "a")]
println!("cfg(myname = 'a') is set");
#[cfg(myname = "b")]
println!("cfg(myname = 'b') is set");
```

```
cfg(myname = 'a') is set
cfg(myname = 'b') is set
```

Other than the `feature` values described in this section, the most commonly used config values are those that the toolchain populates automatically, with values that describe the target environment for the build. These include the OS (`target_os`), CPU architecture (`target_arch`), pointer width (`target_pointer_width`), and endianness (`target_endian`). This allows for code portability, where features that are specific to some particular target are compiled in only when building for that target.

The standard `target_has_atomic` option also provides an example of the multi-valued nature of config values: both `[cfg(target_has_atomic = "32")]` and `[cfg(target_has_atomic = "64")]` will be set for targets that support both 32-bit and 64-bit atomic operations. (For more information on atomics, see Chapter 2 of Mara Bos's *Rust Atomics and Locks* [O'Reilly].)

Features

The Cargo package manager builds on this base `cfg` name/value mechanism to provide the concept of *features*: named selective aspects of the functionality of a crate that can be enabled when building the crate. Cargo ensures that the `feature` option is populated with each of the configured values for each crate that it compiles, and the values are crate-specific.

This is Cargo-specific functionality: to the Rust compiler, `feature` is just another configuration option.

At the time of writing, the most reliable way to determine what features are available for a crate is to examine the crate's *Cargo.toml* manifest file. For example, the following chunk of a manifest file includes *six* features:

```
[features]
default = ["featureA"]
featureA = []
featureB = []
# Enabling `featureAB` also enables `featureA` and `featureB`.
featureAB = ["featureA", "featureB"]
schema = []

[dependencies]
rand = { version = "^0.8", optional = true }
hex = "^0.4"
```

Given that there are only five entries in the `[features]` stanza; there are clearly a couple of subtleties to watch out for.

The first is that the `default` line in the `[features]` stanza is a special feature name, used to indicate to `cargo` which of the features should be enabled by default. These features can still be disabled by passing the `--no-default-features` flag to the build command, and a consumer of the crate can encode this in their *Cargo.toml* file like so:

```
[dependencies]
somecrate = { version = "^0.3", default-features = false }
```

However, `default` still counts as a feature name, which can be tested in code:

```
#[cfg(feature = "default")]
println!("This crate was built with the \"default\" feature enabled.");
#[cfg(not(feature = "default"))]
println!("This crate was built with the \"default\" feature disabled.");
```

The second subtlety of feature definitions is hidden in the `[dependencies]` section of the original *Cargo.toml* example: the `rand` crate is a dependency that is marked as `optional =`

`true`, and that effectively makes `"rand"` into the name of a feature.[1] If the crate is compiled with `--features rand`, then that dependency is activated:

```
#[cfg(feature = "rand")]
pub fn pick_a_number() -> u8 {
    rand::random::<u8>()
}

#[cfg(not(feature = "rand"))]
pub fn pick_a_number() -> u8 {
    4 // chosen by fair dice roll.
}
```

This also means that *crate names and feature names share a namespace*, even though one is typically global (and usually governed by `crates.io`), and one is local to the crate in question. Consequently, **choose feature names carefully** to avoid clashes with the names of any crates that might be relevant as potential dependencies. It is possible to work around a clash, because Cargo includes a mechanism that allows imported crates to be renamed (the `package` key), but it's easier not to have to.

So you can **determine a crate's features by examining `[features]` *as well as* `optional` `[dependencies]`** in the crate's *Cargo.toml* file. To turn on a feature of a dependency, add the `features` option to the relevant line in the `[dependencies]` stanza of your own manifest file:

```
[dependencies]
somecrate = { version = "^0.3", features = ["featureA", "rand" ] }
```

This line ensures that `somecrate` will be built with both the `featureA` and the `rand` feature enabled. However, that might not be the only features that are enabled; other features may also be enabled due to a phenomenon known as *feature unification*. This means that a crate will get built with the *union* of all of the features that are requested by anything in the build graph. In other words, if some other dependency in the build graph also relies on `somecrate`, but with just `featureB` enabled, then the crate will be built with all of `featureA`, `featureB`, and `rand` enabled, to satisfy everyone.[2] The same consideration applies to default features: if your crate sets `default-features = false` for a dependency but some other place in the build graph leaves the default features enabled, then enabled they will be.

Feature unification means that **features should be additive**; it's a bad idea to have mutually incompatible features because there's nothing to prevent the incompatible features being simultaneously enabled by different users.

For example, if a crate exposes a `struct` and its fields publicly, it's a bad idea to make the fields feature-dependent:

```
/// A structure whose contents are public, so external users can construct
/// instances of it.
#[derive(Debug)]
pub struct ExposedStruct {
    pub data: Vec<u8>,

    /// Additional data that is required only when the `schema` feature
    /// is enabled.
    #[cfg(feature = "schema")]
    pub schema: String,
}
```

A user of the crate that tries to build an instance of the `struct` has a quandary: should they fill in the `schema` field or not? One way to *try* to solve this is to add a corresponding feature in the user's *Cargo.toml*:

```
[features]
# The `use-schema` feature here turns on the `schema` feature of `somecrate`.
# (This example uses different feature names for clarity; real code is more
# likely to reuse the feature names across both places.)
use-schema = ["somecrate/schema"]
```

and to make the `struct` construction depend on this feature:

```
let s = somecrate::ExposedStruct {
    data: vec![0x82, 0x01, 0x01],

    // Only populate the field if we've requested
    // activation of `somecrate/schema`.
    #[cfg(feature = "use_schema")]
    schema: "[int int]",
};
```

However, this doesn't cover all eventualities: the code will fail to compile if this code doesn't activate `somecrate/schema` but some other transitive dependency does. The core of the problem is that only the crate that has the feature can check the feature; there's no way for the user of the crate to determine whether Cargo has turned on `somecrate/schema` or not. As a result, you should **avoid feature-gating public fields** in structures.

A similar consideration applies to public traits, intended to be used outside the crate they're defined in. Consider a trait that includes a feature gate on one of its methods:

```
/// Trait for items that support CBOR serialization.
pub trait AsCbor: Sized {
    /// Convert the item into CBOR-serialized data.
    fn serialize(&self) -> Result<Vec<u8>, Error>;

    /// Create an instance of the item from CBOR-serialized data.
    fn deserialize(data: &[u8]) -> Result<Self, Error>;

    /// Return the schema corresponding to this item.
    #[cfg(feature = "schema")]
    fn cddl(&self) -> String;
}
```

External trait implementors again have a quandary: should they implement the `cddl(&self)` method or not? The external code that tries to implement the trait doesn't know—and can't tell—whether to implement the feature-gated method or not.

So the net is that you should **avoid feature-gating methods on public traits**. A trait method with a default implementation (Item 13) might be a partial exception to this—but only if it never makes sense for external code to override the default.

Feature unification also means that if your crate has *N* independent features,[3] then all of the 2^N possible build combinations can occur in practice. To avoid unpleasant surprises, it's a good idea to ensure that your CI system (Item 32) covers all of these 2^N combinations, in all of the available test variants (Item 30).

However, the use of optional features is very helpful in controlling exposure to an expanded dependency graph (Item 25). This is particularly useful in low-level crates that are capable of being used in a `no_std` environment (Item 33)—it's common to have a `std` or `alloc` feature that turns on functionality that relies on those libraries.

Things to Remember

- Feature names overlap with dependency names.
- Feature names should be carefully chosen so they don't clash with potential dependency names.
- Features should be additive.
- Avoid feature gates on public `struct` fields or trait methods.
- Having lots of independent features potentially leads to a combinatorial explosion of different build configurations.

---

[1] This default behavior can be disabled by using a `"dep:<crate>"` reference elsewhere in the `features` stanza; see the docs for details.

[2] The `cargo tree --edges features` command can help with determining which features are enabled for which crates, and why.

[3] Features can force other features to be enabled; in the original example, the `featureAB` feature forces both `featureA` and `featureB` to be enabled.

Tooling

Titus Winters (Google's C++ library lead) describes software engineering as programming integrated over time, or sometimes as programming integrated over time and people. Over longer timescales, and a wider team, there's more to a codebase than just the code held within it.

Modern languages, including Rust, are aware of this and come with an ecosystem of tooling that goes way beyond just converting the program into executable binary code (the compiler).

This chapter explores the Rust tooling ecosystem, with a general recommendation to make use of all of this infrastructure. Obviously, doing so needs to be proportionate—setting up CI, documentation builds, and six types of test would be overkill for a throwaway program that is run only twice. But for most of the things described in this chapter, there's lots of "bang for the buck": a little bit of investment into tooling integration will yield worthwhile benefits.

Item 27: Document public interfaces

If your crate is going to be used by other programmers, then it's a good idea to add documentation for its contents, particularly its public API. If your crate is more than just ephemeral, throwaway code, then that "other programmer" includes the you-of-the-future, when you have forgotten the details of your current code.

This is not advice that's specific to Rust, nor is it new advice—for example, *Effective Java* 2nd edition (from 2008) has Item 44: "Write doc comments for all exposed API elements".

The particulars of Rust's documentation comment format—Markdown-based, delimited with `///` or `//!`—are covered in the Rust book, for example:

```
/// Calculate the [`BoundingBox`] that exactly encompasses a pair
/// of [`BoundingBox`] objects.
pub fn union(a: &BoundingBox, b: &BoundingBox) -> BoundingBox {
    // ...
}
```

However, there are some specific details about the format that are worth highlighting:

- *Use a code font for code*: For anything that would be typed into source code as is, surround it with back-quotes to ensure that the resulting documentation is in a fixed-width font, making the distinction between `code` and text clear.
- *Add copious cross-references*: Add a Markdown link for anything that might provide context for someone reading the documentation. In particular, **cross-reference identifiers** with the convenient `[`SomeThing`]` syntax—if `SomeThing` is in scope, then the resulting documentation will hyperlink to the right place.
- *Consider including example code*: If it's not trivially obvious how to use an entrypoint, adding an `# Examples` section with sample code can be helpful. Note that sample code in doc comments gets compiled and executed when you run `cargo test` (see Item 30), which helps it stay in sync with the code it's demonstrating.
- *Document panics and `unsafe` constraints*: If there are inputs that cause a function to panic, document (in a `# Panics` section) the preconditions that are required to avoid the `panic!`. Similarly, document (in a `# Safety` section) any requirements for `unsafe` code.

The documentation for Rust's standard library provides an excellent example to emulate for all of these details.

Tooling

The Markdown format that's used for documentation comments results in elegant output, but this also means that there is an explicit conversion step (`cargo doc`). This in turn raises the possibility that something goes wrong along the way.

The simplest advice for this is just to **read the rendered documentation** after writing it, by running `cargo doc --open` (or `cargo doc --no-deps --open` to restrict the generated documentation to just the current crate).

You could also check that all the generated hyperlinks are valid, but that's a job more suited to a machine—via the `broken_intra_doc_links` crate attribute:[1]

```
#![deny(broken_intra_doc_links)]

/// The bounding box for a [`Polygone`].
#[derive(Clone, Debug)]
pub struct BoundingBox {
    // ...
}
```

With this attribute enabled, `cargo doc` will detect invalid links:

```
error: unresolved link to `Polygone`
 --> docs/src/main.rs:4:30
  |
4 | /// The bounding box for a [`Polygone`].
  |                              ^^^^^^^^ no item named `Polygone` in scope
  |
```

You can also *require* documentation, by enabling the `#![warn(missing_docs)]` attribute for the crate. When this is enabled, the compiler will emit a warning for every undocumented public item. However, there's a risk that enabling this option will lead to poor-quality documentation comments that are rushed out just to get the compiler to shut up—more on this to come.

As ever, any tooling that detects potential problems should form a part of your CI system (Item 32), to catch any regressions that creep in.

Additional Documentation Locations

The output from `cargo doc` is the primary place where your crate is documented, but it's not the only place—other parts of a Cargo project can help users figure out how to use your code.

The `examples/` subdirectory of a Cargo project can hold the code for standalone binaries that make use of your crate. These programs are built and run very similarly to integration tests (Item 30) but are specifically intended to hold example code that illustrates the correct use of your crate's interface.

On a related note, bear in mind that the integration tests under the `tests/` subdirectory can also serve as examples for the confused user, even though their primary purpose is to test the crate's external interface.

Published Crate Documentation

If you publish your crate to `crates.io`, the documentation for your project will be visible at docs.rs, which is an official Rust project that builds and hosts documentation for published crates.

Note that `crates.io` and `docs.rs` are intended for slightly different audiences: `crates.io` is aimed at people who are choosing what crate to use, whereas `docs.rs` is intended for people figuring out how to use a crate they've already included (although there's obviously considerable overlap between the two).

As a result, the home page for a crate shows different content in each location:

- `docs.rs`: Shows the top-level page from the output of `cargo doc`, as generated from `//!` comments in the top-level *src/lib.rs* file.
- `crates.io`: Shows the content of any top-level *README.md* file[2] that's included in the project's repo.

What *Not* to Document

When a project *requires* that documentation be included for all public items (as mentioned in the first section), it's very easy to fall into the trap of having documentation that's a pointless waste of valuable pixels. Having the compiler warn about missing doc comments is only a proxy for what you really want—useful documentation—and is likely to incentivize programmers to do the minimum needed to silence the warning.

Good doc comments are a boon that helps users understand the code they're using; bad doc comments impose a maintenance burden and increase the chance of user confusion when they get out of sync with the code. So how to distinguish between the two?

The primary advice is to **avoid repeating in text something that's clear from the code**. Item 1 exhorted you to encode as much semantics as possible into Rust's type system; once you've done that, allow the type system to document those semantics. Assume that the

reader is familiar with Rust—possibly because they've read a helpful collection of Items describing effective use of the language—and don't repeat things that are clear from the signatures and types involved.

Returning to the previous example, an overly verbose documentation comment might be as follows:

```
/// Return a new [`BoundingBox`] object that exactly encompasses a pair
/// of [`BoundingBox`] objects.
///
/// Parameters:
///  - `a`: an immutable reference to a `BoundingBox`
///  - `b`: an immutable reference to a `BoundingBox`
/// Returns: new `BoundingBox` object.
pub fn union(a: &BoundingBox, b: &BoundingBox) -> BoundingBox {
```

This comment repeats many details that are clear from the function signature, to no benefit.

Worse, consider what's likely to happen if the code gets refactored to store the result in one of the original arguments (which would be a breaking change; see Item 21). No compiler or tool complains that the comment isn't updated to match, so it's easy to end up with an out-of-sync comment:

```
/// Return a new [`BoundingBox`] object that exactly encompasses a pair
/// of [`BoundingBox`] objects.
///
/// Parameters:
///  - `a`: an immutable reference to a `BoundingBox`
///  - `b`: an immutable reference to a `BoundingBox`
/// Returns: new `BoundingBox` object.
pub fn union(a: &mut BoundingBox, b: &BoundingBox) {
```

In contrast, the original comment survives the refactoring unscathed, because its text describes behavior, not syntactic details:

```
/// Calculate the [`BoundingBox`] that exactly encompasses a pair
/// of [`BoundingBox`] objects.
pub fn union(a: &mut BoundingBox, b: &BoundingBox) {
```

The mirror image of the preceding advice also helps improve documentation: **include in text anything that's *not* clear from the code**. This includes preconditions, invariants, panics, error conditions, and anything else that might surprise a user; if your code can't comply with the principle of least astonishment, make sure that the surprises are documented so you can at least say, "I told you so".

Another common failure mode is when doc comments describe how some other code uses a method, rather than what the method does:

```
/// Return the intersection of two [`BoundingBox`] objects, returning `None`
/// if there is no intersection. The collision detection code in `hits.rs`
/// uses this to do an initial check to see whether two objects might overlap,
/// before performing the more expensive pixel-by-pixel check in
/// `objects_overlap`.
pub fn intersection(
    a: &BoundingBox,
    b: &BoundingBox,
) -> Option<BoundingBox> {
```

Comments like this are almost guaranteed to get out of sync: when the using code (here, `hits.rs`) changes, the comment that describes the behavior is nowhere nearby.

Rewording the comment to focus more on the *why* makes it more robust to future changes:

```
/// Return the intersection of two [`BoundingBox`] objects, returning `None`
/// if there is no intersection.  Note that intersection of bounding boxes
/// is necessary but not sufficient for object collision -- pixel-by-pixel
/// checks are still required on overlap.
pub fn intersection(
    a: &BoundingBox,
    b: &BoundingBox,
) -> Option<BoundingBox> {
```

When writing software, it's good advice to "program in the future tense":[3] structure the code to accommodate future changes. The same principle is true for documentation: focusing on the semantics, the *whys* and the *why nots*, gives text that is more likely to remain helpful in the long run.

Things to Remember

- Add doc comments for public API items.
- Describe aspects of the code—such as panics and safety criteria—that aren't obvious from the code itself.
- *Don't* describe things that are obvious from the code itself.
- Make navigation clearer by providing cross-references and by making identifiers stand out.

---

[1] Historically, this option used to be called `intra_doc_link_resolution_failure`.

[2] The default behavior of automatically including *README.md* can be overridden with the `readme` field in *Cargo.toml*.

[3] Scott Meyers, *More Effective C++* (Addison-Wesley), Item 32.

Item 28: Use macros judiciously

> "In some cases it's easy to decide to write a macro instead of a function, because only a macro can do what's needed." – Paul Graham, "On Lisp (Prentice Hall)"

Rust's macro systems allow you to perform *metaprogramming*: to write code that emits code into your project. This is most valuable when there are chunks of "boilerplate" code that are deterministic and repetitive and that would otherwise need to be kept in sync manually.

Programmers coming to Rust may have previously encountered the macros provided by C/C++'s preprocessor, which perform textual substitution on the tokens of the input text. Rust's macros are a different beast, because they work on either the parsed tokens of the program or on the *abstract syntax tree* (AST) of the program, rather than just its textual content.

This means Rust macros can be aware of code structure and can consequently avoid entire classes of macro-related footguns. In particular, we see in the following section that Rust's declarative macros are *hygienic*—they cannot accidentally refer to ("capture") local variables in the surrounding code.

One way to think about macros is to see them as a different level of abstraction in the code. A simple form of abstraction is a function: it abstracts away the differences between different values of the same *type*, with implementation code that can use any of the features and methods of that type, regardless of the current value being operated on. A generic is a different level of abstraction: it abstracts away the difference between different *types* that satisfy a trait bound, with implementation code that can use any of the methods provided by the trait bounds, regardless of the current type being operated on.

A macro abstracts away the difference between different fragments of the program that play the same role (type, identifier, expression, etc.); the implementation can then include any code that makes use of those fragments in the same role.

Rust provides two ways to define macros:

- Declarative macros, also known as "macros by example", allow the insertion of arbitrary Rust code into the program, based on the input parameters to the macro (which are categorized according to their role in the AST).
- Procedural macros allow the insertion of arbitrary Rust code into the program, based on the parsed tokens of the source code. This is most commonly used for `derive` macros, which can generate code based on the contents of data structure definitions.

Declarative Macros

Although this Item isn't the place to reproduce the documentation for declarative macros, a few reminders of details to watch out for are in order.

First, be aware that the scoping rules for using a declarative macro are different than for other Rust items. If a declarative macro is defined in a source code file, only the code *after* the macro definition can make use of it:

```
fn before() {
    println!("[before] square {} is {}", 2, square!(2));
}

/// Macro that squares its argument.
macro_rules! square {
    { $e:expr } => { $e * $e }
}

fn after() {
    println!("[after] square {} is {}", 2, square!(2));
}
```

```
error: cannot find macro `square` in this scope
 --> src/main.rs:4:45
  |
4 |     println!("[before] square {} is {}", 2, square!(2));
  |                                             ^^^^^^
  |
  = help: have you added the `#[macro_use]` on the module/import?
```

The `#[macro_export]` attribute makes a macro more widely visible, but this also has an oddity: a macro appears at the top level of a crate, even if it's defined in a module:

```
mod submod {
    #[macro_export]
    macro_rules! cube {
        { $e:expr } => { $e * $e * $e }
    }
}

mod user {
    pub fn use_macro() {
        // Note: *not* `crate::submod::cube!`
        let cubed = crate::cube!(3);
        println!("cube {} is {}", 3, cubed);
    }
}
```

Rust's declarative macros are what's known as *hygienic*: the expanded code in the body of the macro is not allowed to make use of local variable bindings. For example, a macro that assumes that some variable `x` exists:

```
// Create a macro that assumes the existence of a local `x`.
macro_rules! increment_x {
    {} => { x += 1; };
}
```

will trigger a compilation failure when it is used:

```
let mut x = 2;
increment_x!();
println!("x = {}", x);
```

```
error[E0425]: cannot find value `x` in this scope
   --> src/main.rs:55:13
    |
55  |     {} => { x += 1; };
    |             ^ not found in this scope
...
314 |     increment_x!();
    |     -------------- in this macro invocation
    |
    = note: this error originates in the macro `increment_x`
```

This hygienic property means that Rust's macros are safer than C preprocessor macros. However, there are still a couple of minor gotchas to be aware of when using them.

The first is to realize that even if a macro invocation *looks* like a function invocation, it's not. A macro generates code at the point of invocation, and that generated code can perform manipulations of its arguments:

```
macro_rules! inc_item {
    { $x:ident } => { $x.contents += 1; }
}
```

This means that the normal intuition about whether parameters are moved or `&`-referred-to doesn't apply:

```
let mut x = Item { contents: 42 }; // type is not `Copy`

// Item is *not* moved, despite the (x) syntax,
// but the body of the macro *can* modify `x`.
inc_item!(x);

println!("x is {x:?}");
```

```
x is Item { contents: 43 }
```

This becomes clear if we remember that the macro inserts code at the point of invocation—in this case, adding a line of code that increments `x.contents`. The `cargo-expand` tool shows the code that the compiler sees, after macro expansion:

```
let mut x = Item { contents: 42 };
x.contents += 1;
{
    ::std::io::_print(format_args!("x is {0:?}\n", x));
};
```

The expanded code includes the modification in place, via the owner of the item, not a reference. (It's also interesting to see the expanded version of `println!`, which relies on the `format_args!` macro, to be discussed shortly.)[1]

So the exclamation mark serves as a warning: the expanded code for the macro may do arbitrary things to or with its arguments.

The expanded code can also include control flow operations that aren't visible in the calling code, whether they be loops, conditionals, `return` statements, or use of the `?` operator. Obviously, this is likely to violate the principle of least astonishment, so **prefer macros whose behavior aligns with normal Rust** where possible and appropriate. (On the other hand, if the *purpose* of the macro is to allow weird control flow, go for it! But help out your users by making sure the control flow behavior is clearly documented.)

For example, consider a macro (for checking HTTP status codes) that silently includes a `return` in its body:

```
/// Check that an HTTP status is successful; exit function if not.
macro_rules! check_successful {
    { $e:expr } => {
        if $e.group() != Group::Successful {
            return Err(MyError("HTTP operation failed"));
        }
    }
}
```

Code that uses this macro to check the result of some kind of HTTP operation can end up with control flow that's somewhat obscure:

```
let rc = perform_http_operation();
check_successful!(rc); // may silently exit the function

// ...
```

An alternative version of the macro that generates code that emits a `Result`:

```
/// Convert an HTTP status into a `Result<(), MyError>` indicating success.
macro_rules! check_success {
    { $e:expr } => {
        match $e.group() {
            Group::Successful => Ok(()),
            _ => Err(MyError("HTTP operation failed")),
        }
    }
}
```

gives code that's easier to follow:

```
let rc = perform_http_operation();
check_success!(rc)?; // error flow is visible via `?`

// ...
```

The second thing to watch out for with declarative macros is a problem shared with the C preprocessor: if the argument to a macro is an expression with side effects, beware of repeated use of the argument in the macro. The `square!` macro defined earlier takes an arbitrary expression as an argument and then uses that argument twice, which can lead to surprises:

```
let mut x = 1;
let y = square!({
    x += 1;
    x
});
println!("x = {x}, y = {y}");
// output: x = 3, y = 6
```

Assuming that this behavior isn't intended, one way to fix it is simply to evaluate the expression once and assign the result to a local variable:

```
macro_rules! square_once {
    { $e:expr } => {
        {
            let x = $e;
            x*x // Note: there's a detail here to be explained later...
        }
    }
}
// output now: x = 2, y = 4
```

The other alternative is not to allow an arbitrary expression as input to the macro. If the `expr` syntax fragment specifier is replaced with an `ident` fragment specifier, then the macro will only accept identifiers as inputs, and the attempt to feed it an arbitrary expression will no longer compile.

Formatting Values

One common style of declarative macro involves assembling a message that includes various values from the current state of the code. For example, the standard library includes `format!` for assembling a `String`, `println!` for printing to standard output, `eprintln!` for printing to standard error, and so on. The documentation describes the syntax of the formatting directives, which are roughly equivalent to C's `printf` statement. However, the format arguments are type safe and checked at compile time, and the implementations of the macro use the `Display` and `Debug` traits described in Item 10 to format individual values.[2]

You can (and should) use the same formatting syntax for any macros of your own that perform a similar function. For example, the logging macros provided by the `log` crate use the same syntax as `format!`. To do this, **use `format_args!` for macros that perform argument formatting** rather than attempting to reinvent the wheel:

```
/// Log an error including code location, with `format!`-like arguments.
/// Real code would probably use the `log` crate.
macro_rules! my_log {
    { $($arg:tt)+ } => {
        eprintln!("{}:{}: {}", file!(), line!(), format_args!($($arg)+));
    }
}


let x = 10u8;
// Format specifiers:
// - `x` says print as hex
// - `#` says prefix with '0x'
// - `04` says add leading zeroes so width is at least 4
//   (this includes the '0x' prefix).
my_log!("x = {:#04x}", x);


src/main.rs:331: x = 0x0a
```

Procedural Macros

Rust also supports *procedural macros*, often known as *proc macros*. Like a declarative macro, a procedural macro has the ability to insert arbitrary Rust code into the program's source code. However, the inputs to the macro are no longer just the specific arguments passed to it; instead, a procedural macro has access to the parsed tokens corresponding to some chunk of the original source code. This gives a level of expressive power that approaches the flexibility of dynamic languages such as Lisp—but still with compile-time guarantees. It also helps mitigate the limitations of reflection in Rust, as discussed in Item 19.

Procedural macros must be defined in a separate crate (of crate type `proc-macro`) from where they are used, and that crate will almost certainly need to depend on either `proc-macro` (provided by the standard toolchain) or `proc-macro2` (provided by David Tolnay) as a support library, to make it possible to work with the input tokens.

There are three distinct types of procedural macro:

- *Function-like macros*: Invoked with an argument
- *Attribute macros*: Attached to some chunk of syntax in the program
- *Derive macros*: Attached to the definition of a data structure

Function-like macros

Function-like procedural macros are invoked with an argument, and the macro definition has access to the parsed tokens that make up the argument, and emits arbitrary tokens as a result. Note that the previous sentence says "argument", singular—even if a function-like macro is invoked with what looks like multiple arguments:

```
my_func_macro!(15, x + y, f32::consts::PI);
```

the macro itself receives a single argument, which is a stream of parsed tokens. A macro implementation that just prints (at compile time) the contents of the stream:

```
use proc_macro::TokenStream;

// Function-like macro that just prints (at compile time) its input stream.
#[proc_macro]
pub fn my_func_macro(args: TokenStream) -> TokenStream {
    println!("Input TokenStream is:");
    for tt in args {
        println!("  {tt:?}");
    }
    // Return an empty token stream to replace the macro invocation with.
    TokenStream::new()
}
```

shows the stream corresponding to the input:

```
Input TokenStream is:
  Literal { kind: Integer, symbol: "15", suffix: None,
            span: #0 bytes(10976..10978) }
  Punct { ch: ',', spacing: Alone, span: #0 bytes(10978..10979) }
  Ident { ident: "x", span: #0 bytes(10980..10981) }
  Punct { ch: '+', spacing: Alone, span: #0 bytes(10982..10983) }
  Ident { ident: "y", span: #0 bytes(10984..10985) }
  Punct { ch: ',', spacing: Alone, span: #0 bytes(10985..10986) }
  Ident { ident: "f32", span: #0 bytes(10987..10990) }
  Punct { ch: ':', spacing: Joint, span: #0 bytes(10990..10991) }
  Punct { ch: ':', spacing: Alone, span: #0 bytes(10991..10992) }
  Ident { ident: "consts", span: #0 bytes(10992..10998) }
  Punct { ch: ':', spacing: Joint, span: #0 bytes(10998..10999) }
  Punct { ch: ':', spacing: Alone, span: #0 bytes(10999..11000) }
  Ident { ident: "PI", span: #0 bytes(11000..11002) }
```

The low-level nature of this input stream means that the macro implementation has to do its own parsing. For example, separating out what appear to be separate arguments to the macro involves looking for `TokenTree::Punct` tokens that hold the commas dividing the arguments. The `syn` crate (from David Tolnay) provides a parsing library that can help with this, as described in a section that follows.

Because of this, it's usually easier to use a declarative macro than a function-like procedural macro, because the expected structure of the macro's inputs can be expressed in the matching pattern.

The flip side of this need for manual processing is that function-like proc macros have the flexibility to accept inputs that *don't* parse as normal Rust code. That's not often needed (or sensible), so function-like macros are comparatively rare as a result.

Attribute macros

Attribute macros are invoked by placing them before some item in the program, and the parsed tokens for that item are the input to the macro. The macro can again emit arbitrary tokens as output, but the output is typically some transformation of the input.

For example, an attribute macro can be used to wrap the body of a function:

```
#[log_invocation]
fn add_three(x: u32) -> u32 {
    x + 3
}
```

so that invocations of the function are logged:

```
let x = 2;
let y = add_three(x);
println!("add_three({x}) = {y}");
```

```
log: calling function 'add_three'
log: called function 'add_three' => 5
add_three(2) = 5
```

The implementation of this macro is too large to include here, because the code needs to check the structure of the input tokens and to build up the new output tokens, but the `syn` crate can again help with this processing.

Derive macros

The final type of procedural macro is the derive macro, which allows generated code to be automatically attached to a data structure definition (a `struct`, `enum`, or `union`). This is similar to an attribute macro but there are a few `derive`-specific aspects to be aware of.

The first is that `derive` macros *add* to the input tokens, instead of replacing them altogether. This means that the data structure definition is left intact but the macro has the opportunity to append related code.

The second is that a `derive` macro can declare associated helper attributes, which can then be used to mark parts of the data structure that need special processing. For example, `serde`'s `Deserialize` derive macro has a `serde` helper attribute that can provide metadata to guide the deserialization process:

```
fn generate_value() -> String {
    "unknown".to_string()
}

#[derive(Debug, Deserialize)]
struct MyData {
    // If `value` is missing when deserializing, invoke
    // `generate_value()` to populate the field instead.
    #[serde(default = "generate_value")]
    value: String,
}
```

The final aspect of `derive` macros to be aware of is that the `syn` crate can take care of much of the heavy lifting involved in parsing the input tokens into the equivalent nodes in the AST. The `syn::parse_macro_input!` macro converts the tokens into a `syn::DeriveInput` data structure that describes the content of the item, and `DeriveInput` is much easier to deal with than a raw stream of tokens.

In practice, `derive` macros are the most commonly encountered type of procedural macro—the ability to generate field-by-field (for `struct`s) or variant-by-variant (for `enum`s) implementations allows for a lot of functionality to be provided with little effort from the programmer—for example, by adding a single line like `#[derive(Debug, Clone, PartialEq, Eq)]`.

Because the `derive`d implementations are auto-generated, it also means that the implementations automatically stay in sync with the data structure definition. For example, if you were to add a new field to a `struct`, a manual implementation of `Debug` would need to be manually updated, whereas an automatically `derive`d version would display the new field with no additional effort (or would fail to compile if that wasn't possible).

When to Use Macros

The primary reason to use macros is to avoid repetitive code—especially repetitive code that would otherwise have to be manually kept in sync with other parts of the code. In this respect, writing a macro is just an extension of the same kind of generalization process that normally forms part of programming:

- If you repeat exactly the same code for multiple values of a specific type, encapsulate that code into a common function and call the function from all of the repeated places.
- If you repeat exactly the same code for multiple types, encapsulate that code into a generic with a trait bound and use the generic from all of the repeated places.
- If you repeat the same structure of code in multiple places, encapsulate that code into a macro and use the macro from all of the repeated places.

For example, avoiding repetition for code that works on different `enum` variants can be done only by a macro:

```
enum Multi {
    Byte(u8),
    Int(i32),
    Str(String),
}

/// Extract copies of all the values of a specific enum variant.
#[macro_export]
macro_rules! values_of_type {
    { $values:expr, $variant:ident } => {
        {
            let mut result = Vec::new();
            for val in $values {
                if let Multi::$variant(v) = val {
                    result.push(v.clone());
                }
            }
            result
        }
    }
}

fn main() {
    let values = vec![
        Multi::Byte(1),
        Multi::Int(1000),
        Multi::Str("a string".to_string()),
        Multi::Byte(2),
    ];

    let ints = values_of_type!(&values, Int);
    println!("Integer values: {ints:?}");

    let bytes = values_of_type!(&values, Byte);
    println!("Byte values: {bytes:?}");

    // Output:
    //   Integer values: [1000]
    //   Byte values: [1, 2]
}
```

Another scenario where macros help avoid manual repetition is when information about a collection of data values would otherwise be spread out across different areas of the code.

For example, consider a data structure that encodes information about HTTP status codes; a macro can help keep all of the related information together:

```
// http.rs module

#[derive(Debug, PartialEq, Eq, Clone, Copy)]
pub enum Group {
    Informational, // 1xx
    Successful,    // 2xx
    Redirection,   // 3xx
    ClientError,   // 4xx
    ServerError,   // 5xx
}

// Information about HTTP response codes.
http_codes! {
    Continue           => (100, Informational, "Continue"),
    SwitchingProtocols => (101, Informational, "Switching Protocols"),
    // ...
    Ok                 => (200, Successful, "Ok"),
    Created            => (201, Successful, "Created"),
    // ...
}
```

The macro invocation holds all the related information—numeric value, group, description—for each HTTP status code, acting as a kind of domain-specific language (DSL) holding the source of truth for the data.

The macro definition then describes the generated code; each line of the form `$( ... )+` expands to multiple lines in the generated code, one per argument to the macro:

```
macro_rules! http_codes {
    { $( $name:ident => ($val:literal, $group:ident, $text:literal), )+ } => {
        #[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
        #[repr(i32)]
        enum Status {
            $( $name = $val, )+
        }
        impl Status {
            fn group(&self) -> Group {
                match self {
                    $( Self::$name => Group::$group, )+
                }
            }
            fn text(&self) -> &'static str {
                match self {
                    $( Self::$name => $text, )+
                }
            }
        }
        impl core::convert::TryFrom<i32> for Status {
            type Error = ();
            fn try_from(v: i32) -> Result<Self, Self::Error> {
                match v {
                    $( $val => Ok(Self::$name), )+
                    _ => Err(())
                }
            }
        }
    }
}
```

As a result, the overall output from the macro takes care of generating all of the code that derives from the source-of-truth values:

- The definition of an `enum` holding all the variants
- The definition of a `group()` method, which indicates which group an HTTP status belongs to
- The definition of a `text()` method, which maps a status to a text description
- An implementation of `TryFrom<i32>` to convert numbers to status `enum` values

If an extra value needs to be added later, all that's needed is a single additional line:

```
ImATeapot => (418, ClientError, "I'm a teapot"),
```

Without the macro, four different places would have to be manually updated. The compiler would point out some of them (because `match` expressions need to cover all cases) but not all— `TryFrom<i32>` could easily be forgotten.

Because macros are expanded in place in the invoking code, they can also be used to automatically emit additional diagnostic information—in particular, by using the standard library's `file!()` and `line!()` macros, which emit source code location information:

```
macro_rules! log_failure {
    { $e:expr } => {
        {
            let result = $e;
            if let Err(err) = &result {
                eprintln!("{}:{}: operation '{}' failed: {:?}",
                          file!(),
                          line!(),
                          stringify!($e),
                          err);
            }
            result
        }
    }
}
```

When failures occur, the log file then automatically includes details of what failed and where:

```
use std::convert::TryInto;

let x: Result<u8, _> = log_failure!(512.try_into()); // too big for `u8`
let y = log_failure!(std::str::from_utf8(b"\xc3\x28")); // invalid UTF-8
```

```
src/main.rs:340: operation '512.try_into()' failed: TryFromIntError(())
src/main.rs:341: operation 'std::str::from_utf8(b"\xc3\x28")' failed:
                 Utf8Error { valid_up_to: 0, error_len: Some(1) }
```

Disadvantages of Macros

The primary disadvantage of using a macro is the impact that it has on code readability and maintainability. The preceding "Declarative Macros" section explained that macros allow you to create a DSL to concisely express key features of your code and data. However, this means that anyone reading or maintaining the code now has to understand this DSL—and its implementation in macro definitions—in addition to understanding Rust. For example, the `http_codes!` example in the previous section creates a Rust `enum` named `Status`, but it's not visible in the DSL used for the macro invocation.

This potential impenetrability of macro-based code extends beyond other engineers: various tools that analyze and interact with Rust code may treat the code as opaque, because it no longer follows the syntactical conventions of Rust code. The `square_once!` macro shown earlier provided one trivial example of this: the body of the macro has not been formatted according to the normal `rustfmt` rules:

```
    {
        let x = $e;
        // The `rustfmt` tool doesn't really cope with code in
        // macros, so this has not been reformatted to `x * x`.
        x*x
    }
```

Another example is the earlier `http_codes!` macro, where the DSL uses `Group` enum variant names like `Informational` with neither a `Group::` prefix nor a `use` statement, which may confuse some code navigation tools.

Even the compiler itself is less helpful: its error messages don't always follow the chain of macro use and definition. (However, there are parts of the tooling ecosystem [see Item 31] that can help with this, such as David Tolnay's `cargo-expand`, used earlier.)

Another possible downside for macro use is the possibility of code bloat—a single line of macro invocation can result in hundreds of lines of generated code, which will be invisible to a cursory survey of the code. This is less likely to be a problem when the code is first written, because at that point the code is needed and saves the humans involved from having to write it themselves. However, if the code subsequently stops being necessary, it's not so obvious that there are large amounts of code that could be deleted.

Advice

Although the previous section listed some downsides of macros, they are still fundamentally the right tool for the job when there are different chunks of code that need to be kept consistent but that cannot be coalesced any other way: **use a macro whenever it's the only way to ensure that disparate code stays in sync**.

Macros are also the tool to reach for when there's boilerplate code to be squashed: **use a macro for repeated boilerplate code** that can't be coalesced into a function or a generic.

To reduce the impact on readability, try to avoid syntax in your macros that clashes with Rust's normal syntax rules; either make the macro invocation look like normal code or make it look sufficiently *different* so that no one could confuse the two. In particular, follow these guidelines:

- **Avoid macro expansions that insert references** where possible—a macro invocation like `my_macro!(&list)` aligns better with normal Rust code than `my_macro!(list)` would.
- **Prefer to avoid nonlocal control flow operations in macros** so that anyone reading the code is able to follow the flow without needing to know the details of the macro.

This preference for Rust-like readability sometimes affects the choice between declarative macros and procedural macros. If you need to emit code for each field of a structure, or

each variant of an enum, **prefer a derive macro to a procedural macro that emits a type** (despite the example shown in an earlier section)—it's more idiomatic and makes the code easier to read.

However, if you're adding a derive macro with functionality that's not specific to your project, check whether an external crate already provides what you need (see Item 25). For example, the problem of converting integer values into the appropriate variant of a C-like `enum` is well-covered: all of `enumn::N`, `num_enum::TryFromPrimitive`, `num_derive::FromPrimitive`, and `strum::FromRepr` cover some aspect of this problem.

---

[1] An eagle-eyed reader might notice that `format_args!` still looks like a macro invocation, even after macros have been expanded. That's because it's a special macro that's built into the compiler.

[2] The `std::fmt` module also includes various other traits that are used when displaying data in particular formats. For example, `LowerHex` is used when an `x` format specifier indicates that lower-case hexadecimal output is required.

Item 29: Listen to Clippy

> "It looks like you're writing a letter. Would you like help?" – Microsoft Clippit

Item 31 describes the ecosystem of helpful tools available in the Rust toolbox, but one tool is sufficiently helpful and important to get promoted to an Item of its very own: Clippy.

Clippy is an additional component for Cargo (`cargo clippy`) that emits warnings about your Rust usage, across a variety of categories:

- *Correctness*: Warns about common programming errors
- *Idiom*: Warns about code constructs that aren't quite in standard Rust style
- *Concision*: Points out variations on the code that are more compact
- *Performance*: Suggests alternatives that avoid unnecessary processing or allocation
- *Readability*: Describes alterations to the code that would make it easier for humans to read and understand

For example, the following code builds fine:

```
pub fn circle_area(radius: f64) -> f64 {
    let pi = 3.14;
    pi * radius * radius
}
```

but Clippy points out that the local approximation to π is unnecessary and inaccurate:

```
error: approximate value of `f{32, 64}::consts::PI` found
 --> src/main.rs:5:18
  |
5 |         let pi = 3.14;
  |                  ^^^^
  |
  = help: consider using the constant directly
  = help: for further information visit
    https://rust-lang.github.io/rust-clippy/master/index.html#approx_constant
  = note: `#[deny(clippy::approx_constant)]` on by default
```

The linked webpage explains the problem and points the way to a suitable modification of the code:

```
pub fn circle_area(radius: f64) -> f64 {
    std::f64::consts::PI * radius * radius
}
```

As shown previously, each Clippy warning comes with a link to a webpage describing the error, which explains *why* the code is considered bad. This is vital, because it allows you to decide whether those reasons apply to your code or whether there is some particular reason why the lint check isn't relevant. In some cases, the text also describes known problems with the lint, which might explain an otherwise confusing false positive.

If you decide that a lint warning isn't relevant for your code, you can disable it either for that particular item (`#[allow(clippy::some_lint)]`) or for the entire crate (`#![allow(clippy::some_lint)]` , with an extra `!` , at the top level). However, it's usually better to take the cost of a minor refactoring of the code than to waste time and energy arguing about whether the warning is a genuine false positive.

Whether you choose to fix or disable the warnings, you should **make your code Clippy-warning free**.

That way, when new warnings appear—whether because the code has been changed or because Clippy has been upgraded to include new checks—they will be obvious. Clippy should also be enabled in your CI system (Item 32).

Clippy's warnings are particularly helpful when you're learning Rust, because they reveal gotchas you might not have noticed and help you become familiar with Rust idiom.

Many of the Items in this book also have corresponding Clippy warnings, when it's possible to mechanically check the relevant concern:

- Item 1 suggests using more expressive types than plain `bool` s, and Clippy will also point out the use of multiple `bool` s in function parameters and structures.
- Item 3 covers manipulations of `Option` and `Result` types, and Clippy points out a few possible redundancies, such as the following:
  - Unnecessarily converting `Result` to `Option`
  - Opportunities to use `unwrap_or_default`
- Item 3 also suggests that errors should be returned to the caller where possible; Clippy points out some missing opportunities to do that.
- Item 5 suggests implementing `From` rather than `Into` , which Clippy also suggests.
- Item 5 also describes casts, and Clippy has (disabled by default) warnings for the following:
  - `as` casts that could be `from` instead
  - `as` casts that might truncate
  - `as` casts that might wrap
  - `as` casts that lose precision
  - `as` casts that might convert signed negative numbers to large positive numbers
  - *any* use of `as`
- Item 8 describes fat pointer types, and various Clippy lints point out scenarios where there are unnecessary extra pointer indirections:
  - Holding a heap-allocated collection in a `Box`
  - Holding a heap-allocated collection of `Box` items

    - Taking a reference to a `Box`
- Item 9 describes the myriad ways to manipulate `Iterator` instances; Clippy includes a truly astonishing number of lints that point out combinations of iterator methods that could be simplified.
- Item 10 describes Rust's standard traits and included some implementation requirements that Clippy checks:
    - `Ord` must agree with `PartialOrd`.
    - `PartialEq::ne` should not need a nondefault implementation (see Item 13).
    - `Hash` and `Eq` must be consistent.
    - `Clone` for `Copy` types should match.
- Item 18 suggests limiting the use of `panic!` or related methods like `expect`, which Clippy also detects.
- Item 21 observes that importing a wildcard version of a crate isn't sensible; Clippy agrees.
- Item 23 suggests avoiding wildcard imports, as does Clippy.
- Item 24 and Item 25 touch on the fact that multiple versions of the same crate can appear in your dependency graph; Clippy can be configured to complain when this happens.
- Item 26 explains the additive nature of Cargo features, and Clippy includes a warning about "negative" feature names (e.g., `"no_std"`) that are likely to indicate a feature that falls foul of this.
- Item 26 also explains that a crate's optional dependencies form part of its feature set, and Clippy warns if there are explicit feature names (e.g., `"use-crate-x"`) that could just make use of this instead.
- Item 27 describes conventions for documentation comments, and Clippy will also point out the following:
    - Missing descriptions of `panic!`s
    - Missing descriptions of `unsafe` concerns

As the size of this list should make clear, it can be a valuable learning experience to **read the list of Clippy lint warnings**—including the checks that are disabled by default because they are overly pedantic or because they have a high rate of false positives. Even though you're unlikely to want to enable these warnings for your code, understanding the reasons why they were written in the first place will improve your understanding of Rust and its idiom.

Item 30: Write more than unit tests

> "All companies have test environments.
>
> The lucky ones have production environments separate from the test environment." – @FearlessSon

---

Like most other modern languages, Rust includes features that make it easy to write tests that live alongside your code and that give you confidence that the code is working correctly.

This isn't the place to expound on the importance of tests; suffice it to say that if code isn't tested, it probably doesn't work the way you think it does. So this Item assumes that you're already signed up to **write tests for your code**.

Unit tests and integration tests, described in the next two sections, are the key forms of tests. However, the Rust toolchain, and extensions to the toolchain, allow for various other types of tests. This Item describes their distinct logistics and rationales.

Unit Tests

The most common form of test for Rust code is a unit test, which might look something like this:

```
// ... (code defining `nat_subtract*` functions for natural
//      number subtraction)

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_nat_subtract() {
        assert_eq!(nat_subtract(4, 3).unwrap(), 1);
        assert_eq!(nat_subtract(4, 5), None);
    }

    #[should_panic]
    #[test]
    fn test_something_that_panics() {
        nat_subtract_unchecked(4, 5);
    }
}
```

Some aspects of this example will appear in every unit test:

- A collection of unit test functions.

- Each test function is marked with the `#[test]` attribute.
- The module holding the test functions is annotated with a `#[cfg(test)]` attribute, so the code gets built only in test configurations.

Other aspects of this example illustrate things that are optional and may be relevant only for particular tests:

- The test code here is held in a separate module, conventionally called `tests` or `test`. This module may be inline (as here) or held in a separate *tests.rs* file. Using a separate file for the test module has the advantage that it's easier to spot whether code that uses a function is test code or "real" code.
- The test module might have a wildcard `use super::*` to pull in everything from the parent module under test. This makes it more convenient to add tests (and is an exception to the general advice in Item 23 to avoid wildcard imports).
- The normal visibility rules for modules mean that a unit test has the ability to use anything from the parent module, whether it is `pub` or not. This allows for "open-box" testing of the code, where the unit tests exercise internal features that aren't visible to normal users.
- The test code makes use of `expect()` or `unwrap()` for its expected results. The advice in Item 18 isn't really relevant for test-only code, where `panic!` is used to signal a failing test. Similarly, the test code also checks expected results with `assert_eq!`, which will panic on failure.
- The code under test includes a function that panics on some kinds of invalid input; to exercise that, there's a unit test function that's marked with the `#[should_panic]` attribute. This might be needed when testing an internal function that normally expects the rest of the code to respect its invariants and preconditions, or it might be a public function that has some reason to ignore the advice in Item 18. (Such a function should have a "Panics" section in its doc comment, as described in Item 27.)

Item 27 suggests *not* documenting things that are already expressed by the type system. Similarly, there's no need to test things that are guaranteed by the type system. If your `enum` types start holding values that aren't in the list of allowed variants, you've got bigger problems than a failing unit test!

However, if your code relies on specific functionality from your dependencies, it can be helpful to include basic tests of that functionality. The aim here is not to repeat testing that's already done by the dependency itself but instead to have an early warning system that indicates whether the behavior that you need from the dependency has changed—separately from whether the public API signature has changed, as indicated by the semantic version number (Item 21).

Integration Tests

The other common form of test included with a Rust project is *integration tests*, held under `tests/`. Each file in that directory is run as a separate test program that executes all of the functions marked with `#[test]`.

Integration tests do *not* have access to crate internals and so act as behavior tests that can exercise only the public API of the crate.

Doc Tests

Item 27 described the inclusion of short code samples in documentation comments, to illustrate the use of a particular public API item. Each such chunk of code is enclosed in an implicit `fn main() { ... }` and run as part of `cargo test`, effectively making it an additional test case for your code, known as a *doc test*. Individual tests can also be executed selectively by running `cargo test --doc <item-name>`.

Regularly running tests as part of your CI environment (Item 32) ensures that your code samples don't drift too far from the current reality of your API.

Examples

Item 27 also described the ability to provide example programs that exercise your public API. Each Rust file under `examples/` (or each subdirectory under `examples/` that includes a `main.rs`) can be run as a standalone binary with `cargo run --example <name>` or `cargo test --example <name>`.

These programs have access to only the public API of your crate and are intended to illustrate the use of your API as a whole. Examples are not specifically designated as test code (no `#[test]`, no `#[cfg(test)]`), and they're a poor place to put code that exercises obscure nooks and crannies of your crate—particularly as examples are *not* run by `cargo test` by default.

Nevertheless, it's a good idea to ensure that your CI system (Item 32) builds and runs all the associated examples for a crate (with `cargo test --examples`), because it can act as a good early warning system for regressions that are likely to affect lots of users. As noted, if your examples demonstrate mainline use of your API, then a failure in the examples implies that something significant is wrong:

- If it's a genuine bug, then it's likely to affect lots of users—the very nature of example code means that users are likely to have copied, pasted, and adapted the example.

- If it's an intended change to the API, then the examples need to be updated to match. A change to the API also implies a backward incompatibility, so if the crate is published, then the semantic version number needs a corresponding update to indicate this (Item 21).

The likelihood of users copying and pasting example code means that it should have a different style than test code. In line with Item 18, you should set a good example for your users by avoiding `unwrap()` calls for `Result`s. Instead, make each example's `main()` function return something like `Result<(), Box<dyn Error>>`, and then use the question mark operator throughout (Item 3).

Benchmarks

Item 20 attempts to persuade you that fully optimizing the performance of your code isn't always necessary. Nevertheless, there are definitely times when performance is critical, and if that's the case, then it's a good idea to measure and track that performance. Having *benchmarks* that are run regularly (e.g., as part of CI; Item 32) allows you to detect when changes to the code or the toolchains adversely affect that performance.

The `cargo bench` command runs special test cases that repeatedly perform an operation, and emits average timing information for the operation. At the time of writing, support for benchmarks is not stable, so the precise command may need to be `cargo +nightly bench`. (Rust's unstable features, including the `test` feature used here, are described in the Rust *Unstable Book*.)

However, there's a danger that compiler optimizations may give misleading results, particularly if you restrict the operation that's being performed to a small subset of the real code. Consider a simple arithmetic function:

```
pub fn factorial(n: u128) -> u128 {
    match n {
        0 => 1,
        n => n * factorial(n - 1),
    }
}
```

A naive benchmark for this code:

```
#![feature(test)]
extern crate test;

#[bench]
fn bench_factorial(b: &mut test::Bencher) {
    b.iter(|| {
        let result = factorial(15);
        assert_eq!(result, 1_307_674_368_000);
    });
}
```

gives incredibly positive results:

```
test bench_factorial             ... bench:           0 ns/iter (+/- 0)
```

With fixed inputs and a small amount of code under test, the compiler is able to optimize away the iteration and directly emit the result, leading to an unrealistically optimistic result.

The `std::hint::black_box` function can help with this; it's an identity function whose implementation the compiler is "encouraged, *but not required*" (their italics) to pessimize.

Moving the benchmark code to use this hint:

```
#[bench]
fn bench_factorial(b: &mut test::Bencher) {
    b.iter(|| {
        let result = factorial(std::hint::black_box(15));
        assert_eq!(result, 1_307_674_368_000);
    });
}
```

gives more realistic results:

```
test blackboxed::bench_factorial ... bench:          16 ns/iter (+/- 3)
```

The Godbolt compiler explorer can also help by showing the actual machine code emitted by the compiler, which may make it obvious when the compiler has performed optimizations that would be unrealistic for code running a real scenario.

Finally, if you are including benchmarks for your Rust code, the `criterion` crate may provide an alternative to the standard `test::bench::Bencher` functionality that is more convenient (it runs with stable Rust) and more fully featured (it has support for statistics and graphs).

Fuzz Testing

Fuzz testing is the process of exposing code to randomized inputs in the hope of finding bugs, particularly crashes that result from those inputs. Although this can be a useful technique in general, it becomes much more important when your code is exposed to inputs that may be controlled by someone who is deliberately trying to attack the code—so you should **run fuzz tests if your code is exposed to potential attackers**.

Historically, the majority of defects in C/C++ code that have been exposed by fuzzers have been memory safety problems, typically found by combining fuzz testing with runtime instrumentation (e.g., AddressSanitizer or ThreadSanitizer) of memory access patterns.

Rust is immune to some (but not all) of these memory safety problems, particularly when there is no `unsafe` code involved (Item 16). However, Rust does not prevent bugs in general, and a code path that triggers a `panic!` (see Item 18) can still result in a denial-of-service (DoS) attack on the codebase as a whole.

The most effective forms of fuzz testing are *coverage-guided*: the test infrastructure monitors which parts of the code are executed and favors random mutations of the inputs that explore new code paths. "American fuzzy lop" (AFL) was the original heavyweight champion of this technique, but in more recent years equivalent functionality has been included in the LLVM toolchain as `libFuzzer`.

The Rust compiler is built on LLVM, and so the `cargo-fuzz` subcommand exposes `libFuzzer` functionality for Rust (albeit for only a limited number of platforms).

The primary requirement for a fuzz test is to identify an entrypoint of your code that takes (or can be adapted to take) arbitrary bytes of data as input:

```
/// Determine if the input starts with "FUZZ".
pub fn is_fuzz(data: &[u8]) -> bool {
    if data.len() >= 3 /* oops */
    && data[0] == b'F'
    && data[1] == b'U'
    && data[2] == b'Z'
    && data[3] == b'Z'
    {
        true
    } else {
        false
    }
}
```

With a target entrypoint identified, the Rust *Fuzz Book* gives instructions on how to arrange the fuzzing subproject. At its core is a small driver that connects the target entrypoint to the fuzzing infrastructure:

```
// fuzz/fuzz_targets/target1.rs file
#![no_main]
use libfuzzer_sys::fuzz_target;

fuzz_target!(|data: &[u8]| {
    let _ = somecrate::is_fuzz(data);
});
```

Running `cargo +nightly fuzz run target1` continuously executes the fuzz target with random data, stopping only if a crash is found. In this case, a failure is found almost immediately:

```
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1607525774
INFO: Loaded 1 modules: 1624 [0x108219fa0, 0x10821a5f8),
INFO: Loaded 1 PC tables (1624 PCs): 1624 [0x10821a5f8,0x108220b78),
INFO:        9 files found in fuzz/corpus/target1
INFO: seed corpus: files: 9 min: 1b max: 8b total: 46b rss: 38Mb
#10     INITED cov: 26 ft: 26 corp: 6/22b exec/s: 0 rss: 39Mb
thread panicked at 'index out of bounds: the len is 3 but the index is 3',
     testing/src/lib.rs:77:12
stack backtrace:
   0: rust_begin_unwind
             at /rustc/f77bfb7336f2/library/std/src/panicking.rs:579:5
   1: core::panicking::panic_fmt
             at /rustc/f77bfb7336f2/library/core/src/panicking.rs:64:14
   2: core::panicking::panic_bounds_check
             at /rustc/f77bfb7336f2/library/core/src/panicking.rs:159:5
   3: somecrate::is_fuzz
   4: _rust_fuzzer_test_input
   5: ___rust_try
   6: _LLVMFuzzerTestOneInput
   7: __ZN6fuzzer6Fuzzer15ExecuteCallbackEPKhm
   8: __ZN6fuzzer6Fuzzer6RunOneEPKhmbPNS_9InputInfoEbPb
   9: __ZN6fuzzer6Fuzzer16MutateAndTestOneEv
  10: __ZN6fuzzer6Fuzzer4LoopERNSt3__16vectorINS_9SizedFileENS_
      16fuzzer_allocatorIS3_EEEE
  11: __ZN6fuzzer12FuzzerDriverEPiPPPcPFiPKhmE
  12: _main
```

and the input that triggered the failure is emitted.

Normally, fuzz testing does not find failures so quickly, and so it does *not* make sense to run fuzz tests as part of your CI. The open-ended nature of the testing, and the consequent compute costs, mean that you need to consider how and when to run fuzz tests—perhaps only for new releases or major changes, or perhaps for a limited period of time.[1]

You can also make subsequent runs of the fuzzing infrastructure more efficient, by storing and reusing a *corpus* of previous inputs that the fuzzer found to explore new code paths; this helps subsequent runs of the fuzzer explore new ground, rather than retesting code paths previously visited.

Testing Advice

An Item about testing wouldn't be complete without repeating some common advice (which is mostly not Rust-specific):

- As this Item has endlessly repeated, **run all your tests in CI on every change** (with the exception of fuzz tests).
- When you're fixing a bug, **write a test that exhibits the bug before fixing the bug**. That way you can be sure that the bug is fixed and that it won't be accidentally reintroduced in the future.
- If your crate has features (Item 26), **run tests over every possible combination of available features**.
- More generally, if your crate includes any config-specific code (e.g., `#[cfg(target_os = "windows")]`), **run tests for every platform** that has distinct code.

This Item has covered a lot of different types of tests, so it's up to you to decide how much each of them is relevant and worthwhile for your project.

If you have a lot of test code and you are publishing your crate to `crates.io`, then you might need to consider which of the tests make sense to include in the published crate. By default, `cargo` will include unit tests, integration tests, benchmarks, and examples (but not fuzz tests, because the `cargo-fuzz` tools store these as a separate crate in a subdirectory), which may be more than end users need. If that's the case, you can either `exclude` some of the files or (for behavior tests) move the tests out of the crate and into a separate test crate.

Things to Remember

- Write unit tests for comprehensive testing that includes testing of internal-only code. Run them with `cargo test`.
- Write integration tests to exercise your public API. Run them with `cargo test`.
- Write doc tests that exemplify how to use individual items in your public API. Run them with `cargo test`.
- Write example programs that show how to use your public API as a whole. Run them with `cargo test --examples` or `cargo run --example <name>`.
- Write benchmarks if your code has significant performance requirements. Run them with `cargo bench`.
- Write fuzz tests if your code is exposed to untrusted inputs. Run them (continuously) with `cargo fuzz`.

---

[1] If your code is a widely used open source crate, the Google OSS-Fuzz program may be willing to run fuzzing on your behalf.

Item 31: Take advantage of the tooling ecosystem

The Rust ecosystem has a rich collection of additional tools, which provide functionality above and beyond the essential task of converting Rust into machine code.

When setting up a Rust development environment, you're likely to want most of the following basic tools:[1]

- The `cargo` tool for organizing dependencies (Item 25) and driving the compiler
- The `rustup` tool, which manages the installed Rust toolchains
- An IDE with Rust support, or an IDE/editor plug-in like `rust-analyzer`, that allows you to quickly navigate around a Rust codebase, and provides autocompletion support for writing Rust code
- The Rust playground, for standalone explorations of Rust's syntax and for sharing the results with colleagues
- A bookmarked link to the documentation for the Rust standard library

Beyond these basics, Rust includes many tools that help with the wider task of maintaining a codebase and improving the quality of that codebase. The tools that are included in the official Cargo toolchain cover various essential tasks beyond the basics of `cargo build`, `cargo test`, and `cargo run`, for example:

- `cargo fmt`: Reformats Rust code according to standard conventions.
- `cargo check`: Performs compilation checks without generating machine code, which can be useful to get a fast syntax check.
- `cargo clippy`: Performs lint checks, detecting inefficient or unidiomatic code (Item 29).
- `cargo doc`: Generates documentation (Item 27).
- `cargo bench`: Runs benchmarking tests (Item 30).
- `cargo update`: Upgrades dependencies to the latest versions, selecting versions that are compliant with semantic versioning (Item 21) by default.
- `cargo tree`: Displays the dependency graph (Item 25).
- `cargo metadata`: Emits metadata about the packages that are present in the workspace and their dependencies.

The last of these is particularly useful, albeit indirectly: because there's a tool that emits information about crates in a well-defined format, it's much easier for people to produce other tools that make use of that information (typically via the `cargo_metadata` crate, which provides a set of Rust types to hold the metadata information).

Item 25 described some of the tools that are enabled by this metadata availability, such as `cargo-udeps` (which allows detection of unused dependencies) or `cargo-deny` (which

allows checks for many things, including duplicate dependencies, allowed licenses, and security advisories).

The extensibility of the Rust toolchain is not just limited to package metadata; the compiler's abstract syntax tree can also be built upon, often via the `syn` crate. This information is what makes procedural macros (Item 28) so potent but also powers a variety of other tools:

- `cargo-expand`: Shows the complete source code produced by macro expansion, which can be essential for debugging tricky macro definitions.
- `cargo-tarpaulin`: Supports the generation and tracking of code coverage information.

Any list of specific tools will always be subjective, out of date, and incomplete; the more general point is to **explore the available tools**.

For example, a search for `cargo-<something>` tools gives dozens of results; some will be inappropriate and some will be abandoned, but some might just do exactly what you want.

There are also various efforts to apply formal verification to Rust code, which may be helpful if your code needs higher levels of assurance about its correctness.

Finally, a reminder: if a tool is useful on more than a one-off basis, you should **integrate the tool into your CI system** (as per Item 32). If the tool is fast and false-positive free, it may also make sense to **integrate the tool into your editor or IDE**; the Rust Tools page provides links to relevant documentation for this.

Tools to Remember

In addition to the tools that should be configured to run over your codebase regularly and automatically (Item 32), there are various other tools that have been mentioned elsewhere in the book. For reference, these are collated here—but remember that there are many more tools out there:

- Item 16 recommends the use of Miri when writing subtle `unsafe` code.
- Item 21 and Item 25 include mention of Dependabot, for managing dependency updates.
- Item 21 also mentions `cargo-semver-checks` as a possible option for checking that semantic versioning has been done correctly.
- Item 28 explains that `cargo-expand` can help when debugging macro problems.
- Item 29 is entirely dedicated to the use of Clippy.
- The Godbolt compiler explorer allows you to explore the machine code corresponding to your source code, as described in Item 30.
- Item 30 also mentions additional testing tools, such as `cargo-fuzz` for fuzz testing and `criterion` for benchmarking.
- Item 35 covers the use of `bindgen` for auto-generating Rust FFI wrappers from C code.

[1] This list may be reduced in some environments. For example, Rust development in Android has a centrally controlled toolchain (so no `rustup`) and integrates with Android's Soong build system (so no `cargo`).

Item 32: Set up a continuous integration (CI) system

A CI system is a mechanism for automatically running tools over your codebase, which is triggered whenever there's a change to the codebase—or a proposed change to the codebase.

The recommendation to **set up a CI system** is not at all Rust-specific, so this Item is a mélange of general advice mixed with Rust-specific tool suggestions.

CI Steps

Moving to specifics, what kinds of steps should be included in your CI system? The obvious initial candidates are the following:

- Build the code.
- Run the tests for the code.

In each case, a CI step should run cleanly, quickly, deterministically, and with a zero false positive rate; more on this in the next section.

The "deterministic" requirement also leads to advice for the build step: **use rust-toolchain.toml to specify a fixed version of the toolchain in your CI build**.

The *rust-toolchain.toml* file indicates which version of Rust should be used to build the code—either a specific version (e.g., `1.70`), or a channel (`stable`, `beta`, or `nightly`) possibly with an optional date (e.g., `nightly-2023-09-19`).[1] Choosing a floating channel value here would make the CI results vary as new toolchain versions are released; a fixed value is more deterministic and allows you to deal with toolchain upgrades separately.

Throughout this book, various Items have suggested tools and techniques that can help improve your codebase; wherever possible, these should be included with the CI system. For example, the two fundamental parts of a CI system previously mentioned can be enhanced:

- Build the code.
  - Item 26 describes the use of *features* to conditionally include different chunks of code. If your crate has features, **build every valid combination of features in CI** (and realize that this may involve 2^N different variants—hence the advice to avoid feature creep).
  - Item 33 suggests that you consider making library code `no_std` compatible where possible. You can be confident that your code is genuinely `no_std`

compatible only if you **test `no_std` compatibility in CI**. One option is to make use of the Rust compiler's cross-compilation abilities and build for an explicitly `no_std` target (e.g., `thumbv6m-none-eabi`).
    - Item 21 includes a discussion around declaring a minimum supported Rust version (MSRV) for your code. If you have this, **check your MSRV in CI** by including a step that tests with that specific Rust version.
- Run the tests for the code.
    - Item 30 describes the various different styles of test; **run all test types in CI**. Some test types are automatically included in `cargo test` (unit tests, integration tests, and doc tests), but other test types (e.g., example programs) may need to be explicitly triggered.

However, there are other tools and suggestions that can help improve the quality of your codebase:

- Item 29 waxes lyrical about the advantages of running Clippy over your code; **run Clippy in CI**. To ensure that failures are flagged, set the `-Dwarnings` option (for example, via `cargo clippy -- -Dwarnings`).
- Item 27 suggests documenting your public API; use the `cargo doc` tool to check that the documentation generates correctly and that any hyperlinks in it resolve correctly.
- Item 25 mentions tools such as `cargo-udeps` and `cargo-deny` that can help manage your dependency graph; running these as a CI step prevents regressions.
- Item 31 discusses the Rust tool ecosystem; consider which of these tools are worth regularly running over your codebase. For example, running `rustfmt` / `cargo fmt` in CI allows detection of code that doesn't comply with your project's style guidelines. To ensure that failures are flagged, set the `--check` option.

You can also include CI steps that measure particular aspects of your code:

- Generate code coverage statistics (e.g., with `cargo-tarpaulin`) to show what proportion of your codebase is exercised by your tests.
- Run benchmarks (e.g., with `cargo bench`; Item 30) to measure the performance of your code on key scenarios. However, note that most CI systems run in shared environments where external factors can affect the results; getting more reliable benchmark data is likely to require a more dedicated environment.

These measurement suggestions are a bit more complicated to set up, because the output of a measurement step is more useful when it's compared to previous results. In an ideal world, the CI system would detect when a code change is not fully tested or has an adverse effect on performance; this typically involves integration with some external tracking system.

Here are other suggestions for CI steps that may or may not be relevant for your codebase:

- If your project is a library, recall (from Item 25) that any checked-in *Cargo.lock* file will be ignored by the users of your library. In theory, the semantic version constraints

(Item 21) in *Cargo.toml* should mean that everything works correctly anyway; in practice, consider including a CI step that builds without any local *Cargo.lock*, to detect whether the current versions of dependencies still work correctly.
- If your project includes any kind of machine-generated resources that are version-controlled (e.g., code generated from protocol buffer messages by `prost`), then include a CI step that regenerates the resources and checks that there are no differences compared to the checked-in version.
- If your codebase includes platform-specific (e.g., `#[cfg(target_arch = "arm")]`) code, run CI steps that confirm that the code builds and (ideally) works on that platform. (The former is easier than the latter because the Rust toolchain includes support for cross-compilation.)
- If your project manipulates secret values such as access tokens or cryptographic keys, consider including a CI step that searches the codebase for secrets that have been inadvertently checked in. This is particularly important if your project is public (in which case it may be worth moving the check from CI to a version-control presubmit check).

CI checks don't always need to be integrated with Cargo and the Rust toolchains; sometimes a simple shell script can give more bang for the buck, particularly when a codebase has a local convention that's not universally followed. For example, a codebase might include a convention that any panic-inducing method invocation (Item 18) has a special marker comment or that every `TODO:` comment has an owner (a person or a tracking ID), and a shell script is ideal for checking this.

Finally, consider examining the CI systems of public Rust projects to get ideas for additional CI steps that might be useful for your project. For example, Cargo has a CI system that includes many steps that may provide inspiration.

CI Principles

Moving from the specific to the general, there are some overall principles that should guide the details of your CI system.

The most fundamental principle is **don't waste the time of humans**. If a CI system unnecessarily wastes people's time, they will start looking for ways to avoid it.

The most annoying waste of an engineer's time is a *flaky* test: sometimes it passes and sometimes it fails, even when the setup and codebase are identical. Whenever possible, be ruthless with flaky tests: hunt them down, and put in the time up front to investigate and fix the cause of the flakiness—it will pay for itself in the long run.

Another common waste of engineering time is a CI system that takes a long time to run and that runs only *after* a request for a code review has been triggered. In this situation, there's the potential to waste two people's time: both the author and also the code reviewer, who

may spend time spotting and pointing out issues with the code that the CI bots could have flagged.

To help with this, try to make it easy to run the CI checks manually, independent from the automated system. This allows engineers to get into the habit of triggering them regularly so that code reviewers never even see problems that the CI would have flagged. Better still, make the integration even more continuous by incorporating some of the tools into your editor or IDE setup so that (for example) poorly formatted code never even makes it to disk.

This may also require splitting the checks up if there are time-consuming tests that rarely find problems but are there as a backstop to prevent obscure scenarios breaking.

More generally, a large project may need to divide up its CI checks according to the cadence at which they are run:

- Checks that are integrated into each engineer's development environment (e.g., `rustfmt`)
- Checks that run on every code review request (e.g., `cargo build`, `cargo clippy`) and are easy to run manually
- Checks that run on every change that makes it to the main branch of the project (e.g., full `cargo test` in all supported environments)
- Checks that run at scheduled intervals (e.g., daily or weekly), which can catch rare regressions after the fact (e.g., long-running integration tests and benchmark comparison tests)
- Checks that run on the current code at all times (e.g., fuzz tests)

It's important that the CI system be integrated with whatever code review system is used for your project so that a code review can clearly see a green set of checks and be confident that its code review can focus on the important meaning of the code, not on trivial details.

This need for a green build also means that there can be no exceptions to whatever checks your CI system has put in place. This is worthwhile even if you have to work around an occasional false positive from a tool; once your CI system has an accepted failure ("Oh, everyone knows that test never passes"), then it's vastly harder to spot new regressions.

Item 30 included the common advice of adding a test to reproduce a bug, before fixing the bug. The same principle applies to your CI system: when you discover process problems **add a CI step that detects a process issue, before fixing the issue**. For example, if you discover that some auto-generated code has gotten out of sync with its source, add a check for this to the CI system. This check will initially fail but then turn green once the problem is solved—giving you confidence that this category of process error will not occur again in the future.

Public CI Systems

If your codebase is open source and visible to the public, there are a few extra things to think about with your CI system.

First is the good news: there are lots of free, reliable options for building a CI system for open source code. At the time of writing, GitHub Actions are probably the best choice, but it's far from the only choice, and more systems appear all the time.

Second, for open source code it's worth bearing in mind that your CI system can act as a guide for how to set up any prerequisites needed for the codebase. This isn't a concern for pure Rust crates, but if your codebase requires additional dependencies—databases, alternative toolchains for FFI code, configuration, etc.—then your CI scripts will be an existence proof of how to get all of that working on a fresh system. Encoding these setup steps in reusable scripts allows both the humans and the bots to get a working system in a straightforward way.

Finally, there's bad news for publicly visible crates: the possibility of abuse and attacks. This can range from attempts to perform cryptocurrency mining in your CI system to theft of codebase access tokens, supply chain attacks, and worse. To mitigate these risks, consider these guidelines:

- Restrict access so that CI scripts run automatically only for known collaborators and have to be triggered manually for new contributors.
- Pin the versions of any external scripts to particular versions, or (better yet) specific known hashes.
- Closely monitor any integration steps that need more than just read access to the codebase.

---

[1] If your code relies on particular features that are available only in the nightly compiler, a *rust-toolchain.toml* file also makes that toolchain dependency clear.

Beyond Standard Rust

The Rust toolchain includes support for a much wider variety of environments than just pure Rust application code, running in userspace:

- It supports *cross-compilation*, where the system running the toolchain (the *host*) is not the same as the system that the compiled code will run on (the *target*), which makes it easy to target embedded systems.
- It supports linking with code compiled from languages other than Rust, via built-in FFI capabilities.
- It supports configurations without the full standard library `std`, allowing systems that do not have a full operating system (e.g., no filesystem, no networking) to be targeted.
- It even supports configurations that do not support heap allocation but only have a stack (by omitting use of the standard `alloc` library).

These nonstandard Rust environments can be harder to work in and may be less safe—they can even be `unsafe`—but they give more options for getting the job done.

This chapter of the book discusses just a few of the basics for working in these environments. Beyond these basics, you'll need to consult more environment-specific documentation (such as the *Rustonomicon*).

Item 33: Consider making library code `no_std` compatible

Rust comes with a standard library called `std`, which includes code for a wide variety of common tasks, from standard data structures to networking, from multithreading support to file I/O. For convenience, several of the items from `std` are automatically imported into your program, via the *prelude*: a set of common `use` statements that make common types available without needing to use their full names (e.g., `Vec` rather than `std::vec::Vec`).

Rust also supports building code for environments where it's not possible to provide this full standard library, such as bootloaders, firmware, or embedded platforms in general. Crates indicate that they should be built in this way by including the `#![no_std]` crate-level attribute at the top of *src/lib.rs*.

This Item explores what's lost when building for `no_std` and what library functions you can still rely on—which turns out to be quite a lot.

However, this Item is specifically about `no_std` support in *library* code. The difficulties of making a `no_std` *binary* are beyond this text,[1] so the focus here is how to make sure that library code is available for those poor souls who do have to work in such a minimal environment.

`core`

Even when building for the most restricted of platforms, many of the fundamental types from the standard library are still available. For example, `Option` and `Result` are still available, albeit under a different name, as are various flavors of `Iterator`.

The different names for these fundamental types start with `core::`, indicating that they come from the `core` library, a standard library that's available even in the most `no_std` of environments. These `core::` types behave exactly the same as the equivalent `std::` types, because they're actually the same types—in each case, the `std::` version is just a re-export of the underlying `core::` type.

This means that there's a quick and dirty way to tell if a `std::` item is available in a `no_std` environment: visit the `doc.rust-lang.org` page for the `std` item you're interested in and follow the "source" link (at the top right).[2] If that takes you to a `src/core/...` location, then the item is available under `no_std` via `core::`.

The types from `core` are available for all Rust programs automatically. However, they typically need to be explicitly `use` d in a `no_std` environment, because the `std` prelude is

absent.

In practice, relying purely on `core` is too limiting for many environments, even `no_std` ones. A core (pun intended) constraint of `core` is that it performs *no heap allocation*.

Although Rust excels at putting items on the stack and safely tracking the corresponding lifetimes (Item 14), this restriction still means that standard data structures—vectors, maps, sets—can't be provided, because they need to allocate heap space for their contents. In turn, this also drastically reduces the number of available crates that work in this environment.

`alloc`

However, if a `no_std` environment *does* support heap allocation, then many of the standard data structures from `std` can still be supported. These data structures, along with other allocation-using functionality, are grouped into Rust's `alloc` library.

As with `core`, these `alloc` variants are actually the same types under the covers. For example, the real name of `std::vec::Vec` is actually `alloc::vec::Vec`.

A `no_std` Rust crate needs to explicitly opt in to the use of `alloc`, by adding an `extern crate alloc;` declaration to *src/lib.rs*:[3]

```
//! My `no_std` compatible crate.
#![no_std]

// Requires `alloc`.
extern crate alloc;
```

Pulling in the `alloc` crate enables many familiar friends, now addressed by their true names:

- `alloc::boxed::Box<T>`
- `alloc::rc::Rc<T>`
- `alloc::sync::Arc<T>`
- `alloc::vec::Vec<T>`
- `alloc::string::String`
- `alloc::format!`
- `alloc::collections::BTreeMap<K, V>`
- `alloc::collections::BTreeSet<T>`

With these things available, it becomes possible for many library crates to be `no_std` compatible—for example, if a library doesn't involve I/O or networking.

There's a notable absence from the data structures that `alloc` makes available, though—the collections `HashMap` and `HashSet` are specific to `std`, not `alloc`. That's because these hash-based containers rely on random seeds to protect against hash collision attacks, but safe random number generation requires assistance from the operating system—which `alloc` can't assume exists.

Another notable absence is synchronization functionality like `std::sync::Mutex`, which is required for multithreaded code (Item 17). These types are specific to `std` because they rely on OS-specific synchronization primitives, which aren't available without an OS. If you need to write code that is both `no_std` and multithreaded, third-party crates such as `spin` are probably your only option.

Writing Code for `no_std`

The previous sections made it clear that for *some* library crates, making the code `no_std` compatible just involves the following:

- Replacing `std::` types with identical `core::` or `alloc::` crates (which requires `use` of the full type name, due to the absence of the `std` prelude)
- Shifting from `HashMap` / `HashSet` to `BTreeMap` / `BTreeSet`

However, this only makes sense if all of the crates that you depend on (Item 25) are also `no_std` compatible—there's no point in becoming `no_std` compatible if any user of your crate is forced to link in `std` anyway.

There's also a catch here: the Rust compiler will not tell you if your `no_std` crate depends on a `std`-using dependency. This means that it's easy to undo the work of making a crate `no_std` compatible—all it takes is an added or updated dependency that pulls in `std`.

To protect against this, **add a CI check for a `no_std` build** so that your CI system (Item 32) will warn you if this happens. The Rust toolchain supports cross-compilation out of the box, so this can be as simple as performing a cross-compile for a target system that does not support `std` (e.g., `--target thumbv6m-none-eabi`); any code that inadvertently requires `std` will then fail to compile for this target.

So: if your dependencies support it, and the simple transformations above are all that's needed, then **consider making library code `no_std` compatible**. When it is possible, it's not much additional work, and it allows for the widest reuse of the library.

If those transformations *don't* cover all of the code in your crate but the parts that aren't covered are only a small or well-contained fraction of the code, then consider adding a feature (Item 26) to your crate that turns on just those parts.

Such a feature is conventionally named either `std`, if it enables use of `std`-specific functionality:

```
#![cfg_attr(not(feature = "std"), no_std)]
```

or `alloc`, if it turns on use of `alloc`-derived functionality:

```
#[cfg(feature = "alloc")]
extern crate alloc;
```

Note that there's a trap for the unwary here: don't have a `no_std` feature that *disables* functionality requiring `std` (or a `no_alloc` feature similarly). As explained in Item 26, features need to be additive, and there's no way to combine two users of the crate where one configures `no_std` and one doesn't—the former will trigger the removal of code that the latter relies on.

As ever with feature-gated code, make sure that your CI system (Item 32) builds all the relevant combinations—including a build with the `std` feature disabled on an explicitly `no_std` platform.

Fallible Allocation

The earlier sections of this Item considered two different `no_std` environments: a fully embedded environment with no heap allocation whatsoever (`core`) and a more generous environment where heap allocation is allowed (`core + alloc`).

However, there are some important environments that fall between these two camps—in particular, those where heap allocation is possible but may fail because there's a limited amount of heap.

Unfortunately, Rust's standard `alloc` library includes a pervasive assumption that heap allocations cannot fail, and that's not always a valid assumption.

Even a simple use of `alloc::vec::Vec` could potentially allocate on every line:

```
let mut v = Vec::new();
v.push(1); // might allocate
v.push(2); // might allocate
v.push(3); // might allocate
v.push(4); // might allocate
```

None of these operations returns a `Result`, so what happens if those allocations fail?

The answer depends on the toolchain, target, and configuration but is likely to descend into `panic!` and program termination. There is certainly no answer that allows an allocation

failure on line 3 to be handled in a way that allows the program to move on to line 4.

This assumption of *infallible allocation* gives good ergonomics for code that runs in a "normal" userspace, where there's effectively infinite memory—or at least where running out of memory indicates that the computer as a whole has bigger problems elsewhere.

However, infallible allocation is utterly unsuitable for code that needs to run in environments where memory is limited and programs are required to cope. This is a (rare) area where there's better support in older, less memory-safe, languages:

- C is sufficiently low-level that allocations are manual, and so the return value from `malloc` can be checked for `NULL`.
- C++ can use its exception mechanism to catch allocation failures in the form of `std::bad_alloc` exceptions.[4]

Historically, the inability of Rust's standard library to cope with failed allocation was flagged in some high-profile contexts (such as the Linux kernel, Android, and the Curl tool), and so work to fix the omission is ongoing.

The first step was the "fallible collection allocation" changes, which added fallible alternatives to many of the collection APIs that involve allocation. This generally adds a `try_<operation>` variant that results in a `Result<_, AllocError>`; for example:

- `Vec::try_reserve` is available as an alternative to `Vec::reserve`.
- `Box::try_new` is available (with the nightly toolchain) as an alternative to `Box::new`.

These fallible APIs only go so far; for example, there is (as yet) no fallible equivalent to `Vec::push`, so code that assembles a vector may need to do careful calculations to ensure that allocation errors can't happen:

```
fn try_build_a_vec() -> Result<Vec<u8>, String> {
    let mut v = Vec::new();

    // Perform a careful calculation to figure out how much space is needed,
    // here simplified to...
    let required_size = 4;

    v.try_reserve(required_size)
        .map_err(|_e| format!("Failed to allocate {} items!", required_size))?;

    // We now know that it's safe to do:
    v.push(1);
    v.push(2);
    v.push(3);
    v.push(4);

    Ok(v)
}
```

As well as adding fallible allocation entrypoints, it's also possible to disable *infallible* allocation operations, by turning off the `no_global_oom_handling` config flag (which is on by default). Environments with limited heap (such as the Linux kernel) can explicitly disable this flag, ensuring that no use of infallible allocation can inadvertently creep into the code.

Things to Remember

- Many items in the `std` crate actually come from `core` or `alloc`.
- As a result, making library code `no_std` compatible may be more straightforward than you might think.
- Confirm that `no_std` code remains `no_std` compatible by checking it in CI.
- Be aware that working in a limited-heap environment currently has limited library support.

---

[1] See *The Embedonomicon* or Philipp Oppermann's older blog post for information about what's involved in creating a `no_std` binary.

[2] Be aware that this can occasionally go wrong. For example, at the time of writing, the `Error` trait is defined in `core::` but is marked as unstable there; only the `std::` version is stable.

[3] Prior to Rust 2018, `extern crate` declarations were used to pull in dependencies. This is now entirely handled by *Cargo.toml*, but the `extern crate` mechanism is still used to pull in those parts of the Rust standard library (the *sysroot crates*) that are optional in `no_std` environments.

[4] It's also possible to add the `std::nothrow` overload to calls to `new` and check for `nullptr` return values. However, there are still container methods like `vector<T>::push_back` that allocate under the covers and that can therefore signal allocation failure only via an exception.

Item 34: Control what crosses FFI boundaries

Even though Rust comes with a comprehensive standard library and a burgeoning crate ecosystem, there is still a lot more non-Rust code in the world than there is Rust code.

As with other recent languages, Rust helps with this problem by offering a *foreign function interface* (FFI) mechanism, which allows interoperation with code and data structures written in different languages—despite the name, FFI is not restricted to just functions. This opens up the use of existing libraries in different languages, not just those that have succumbed to the Rust community's efforts to "rewrite it in Rust" (RiiR).

The default target for Rust's interoperability is the C programming language, which is the same interop target that other languages aim at. This is partly driven by the ubiquity of C libraries but is also driven by simplicity: C acts as a "least common denominator" of interoperability, because it doesn't need toolchain support of any of the more advanced features that would be necessary for compatibility with other languages (e.g., garbage collection for Java or Go, exceptions and templates for C++, function overrides for Java and C++, etc.).

However, that's not to say that interoperability with plain C is simple. By including code written in a different language, all of the guarantees and protections that Rust offers are up for grabs, particularly those involving memory safety.

As a result, FFI code in Rust is automatically `unsafe`, and the advice in Item 16 has to be bypassed. This Item explores some replacement advice, and Item 35 will explore some tooling that helps to avoid some (but not all) of the footguns involved in working with FFI. (The FFI chapter of the *Rustonomicon* also contains helpful advice and information.)

Invoking C Functions from Rust

The simplest FFI interaction is for Rust code to invoke a C function, taking "immediate" arguments that don't involve pointers, references, or memory addresses:

```
/* File lib.c */
#include "lib.h"

/* C function definition. */
int add(int x, int y) {
  return x + y;
}
```

This C code provides a *definition* of the function and is typically accompanied by a header file that provides a *declaration* of the function, which allows other C code to use it:

```
/* File lib.h */
#ifndef LIB_H
#define LIB_H

/* C function declaration. */
int add(int x, int y);

#endif  /* LIB_H */
```

The declaration roughly says: somewhere out there is a function called `add`, which takes two integers as input and returns another integer as output. This allows C code to use the `add` function, subject to a promise that the actual code for `add` will be provided at a later date—specifically, at link time.

Rust code that wants to use `add` needs to have a similar declaration, with a similar purpose: to describe the signature of the function and to indicate that the corresponding code will be available later:

```
use std::os::raw::c_int;
extern "C" {
    pub fn add(x: c_int, y: c_int) -> c_int;
}
```

The declaration is marked as `extern "C"` to indicate that an external C library will provide the code for the function.[1] The `extern "C"` marker also automatically marks the function as `#[no_mangle]`, which we explore in a following section.

Linking logistics

The details of how the C toolchain generates an external C library—and its format—are environment-specific and beyond the scope of a Rust book like this. However, one simple variant that's common on Unix-like systems is a *static library* file, which will normally have the form *lib<something>.a* (e.g., *libcffi.a*) and which can be generated using the `ar` tool.

The Rust build system then needs an indication of which library holds the relevant C code. This can be specified either via the `link` attribute in the code:

```
#[link(name = "cffi")] // An external library like `libcffi.a` is needed
extern "C" {
    // ...
}
```

or via a build script that emits a `cargo:rustc-link-lib` instruction to `cargo`:[2]

```
// File build.rs
fn main() {
    // An external library like `libcffi.a` is needed
    println!("cargo:rustc-link-lib=cffi");
}
```

The latter option is more flexible, because the build script can examine its environment and behave differently depending on what it finds.

In either case, the Rust build system is also likely to need information about how to find the C library, if it's not in a standard system location. This can be specified by having a build script that emits a `cargo:rustc-link-search` instruction to `cargo`, containing the library location:

```
// File build.rs
fn main() {
    // ...

    // Retrieve the location of `Cargo.toml`.
    let dir = std::env::var("CARGO_MANIFEST_DIR").unwrap();
    // Look for native libraries one directory higher up.
    println!(
        "cargo:rustc-link-search=native={}",
        std::path::Path::new(&dir).join("..").display()
    );
}
```

Code concerns

Returning to the source code, even this simplest of examples comes with some gotchas. First, use of FFI functions is automatically `unsafe`:

```
let x = add(1, 1);
```

```
error[E0133]: call to unsafe function is unsafe and requires unsafe function
              or block
   --> src/main.rs:176:13
    |
176 |     let x = add(1, 1);
    |             ^^^^^^^^^ call to unsafe function
    |
    = note: consult the function's documentation for information on how to
            avoid undefined behavior
```

and so needs to be wrapped in `unsafe { }`.

The next thing to watch out for is the use of C's `int` type, represented as `std::os::raw::c_int`. How big is an `int`? It's *probably* true that the following two things

are the same:

- The size of an `int` for the toolchain that compiled the C library
- The size of a `std::os::raw::c_int` for the Rust toolchain

But why take the chance? **Prefer sized types at FFI boundaries**, where possible—which for C means making use of the types (e.g., `uint32_t`) defined in `<stdint.h>`. However, if you're dealing with an existing codebase that already uses `int` / `long` / `size_t`, this may be a luxury you don't have.

The final practical concern is that the C code and the equivalent Rust declaration need to exactly match. Worse still, if there's a mismatch, the build tools will not emit a warning—they will just silently emit incorrect code.

Item 35 discusses the use of the `bindgen` tool to prevent this problem, but it's worth understanding the basics of what's going on under the covers to understand *why* the build tools can't detect the problem on their own. In particular, it's worth understanding the basics of *name mangling*.

Name mangling

Compiled languages generally support *separate compilation*, where different parts of the program are converted into machine code as separate chunks (object files), which can then be combined into a complete program by the *linker*. This means that if only one small part of the program's source code changes, only the corresponding object file needs to be regenerated; the link step then rebuilds the program, combining both the changed object and all the other unmodified objects.

The link step is (roughly speaking) a "join-the-dots" operation: some object files provide definitions of functions and variables, and other object files have placeholder markers indicating that they expect to use a definition from some other object, but it wasn't available at compile time. The linker combines the two: it ensures that any placeholder in the compiled code is replaced with a reference to the corresponding concrete definition.

The linker performs this correlation between the placeholders and the definitions by simply checking for a matching name, meaning that there is a single global namespace for all of these correlations.

Historically, this was fine for linking C language programs, where a single name could not be reused in any way—the name of a function is exactly what appears in the object file. (As a result, a common convention for C libraries is to manually add a prefix to all symbols so that `lib1_process` doesn't clash with `lib2_process`.)

However, the introduction of C++ caused a problem because C++ allows overridden definitions with the same name:

```
// C++ code
namespace ns1 {
int32_t add(int32_t a, int32_t b) { return a+b; }
int64_t add(int64_t a, int64_t b) { return a+b; }
}
namespace ns2 {
int32_t add(int32_t a, int32_t b) { return a+b; }
}
```

The solution for this is *name mangling*: the compiler encodes the signature and type information for the overridden functions into the name that's emitted in the object file, and the linker continues to perform its simple-minded 1:1 correlation between placeholders and definitions.

On Unix-like systems, the `nm` tool can help show what the linker works with:

```
% nm ffi-lib.o | grep add  # what the linker sees for C
0000000000000000 T _add

% nm ffi-cpp-lib.o | grep add  # what the linker sees for C++
0000000000000000 T __ZN3ns13addEii
0000000000000020 T __ZN3ns13addExx
0000000000000040 T __ZN3ns23addEii
```

In this case, it shows three mangled symbols, all of which refer to code (the `T` indicates the *text section* of the binary, which is the traditional name for where code lives).

The `c++filt` tool helps translate this back into what would be visible in C++ code:

```
% nm ffi-cpp-lib.o | grep add | c++filt  # what the programmer sees
0000000000000000 T ns1::add(int, int)
0000000000000020 T ns1::add(long long, long long)
0000000000000040 T ns2::add(int, int)
```

Because the mangled name includes type information, the linker can and will complain about any mismatch in the type information between placeholder and definition. This gives some measure of type safety: if the definition changes but the place using it is not updated, the toolchain will complain.

Returning to Rust, `extern "C"` foreign functions are implicitly marked as `#[no_mangle]`, and the symbol in the object file is the bare name, exactly as it would be for a C program. This means that the type safety of function signatures is lost: because the linker sees only the bare names for functions, if there are any differences in type expectations between definition and use, the linker will carry on regardless and problems will arise only at runtime.

Accessing C Data from Rust

The C `add` example in the previous section passed the simplest possible type of data back and forth between Rust and C: an integer that fits in a machine register. Even so, there were still things to be careful about, so it's no surprise then that dealing with more complex data structures also has wrinkles to watch out for.

Both C and Rust use the `struct` to combine related data into a single data structure. However, when a `struct` is realized in memory, the two languages may well choose to put different fields in different places or even in different orders (the *layout*). To prevent mismatches, **use `#[repr(C)]` for Rust types used in FFI**; this representation is designed for the purpose of allowing C interoperability:

```
/* C data structure definition. */
/* Changes here must be reflected in lib.rs. */
typedef struct {
    uint8_t byte;
    uint32_t integer;
} FfiStruct;
```

```
// Equivalent Rust data structure.
// Changes here must be reflected in lib.h / lib.c.
#[repr(C)]
pub struct FfiStruct {
    pub byte: u8,
    pub integer: u32,
}
```

The structure definitions have a comment to remind the humans involved that the two places need to be kept in sync. Relying on the constant vigilance of humans is likely to go wrong in the long term; as for function signatures, it's better to automate this synchronization between the two languages via a tool like `bindgen` (Item 35).

One particular type of data that's worth thinking about carefully for FFI interactions is strings. The default definitions of what makes up a string are somewhat different between C and Rust:

- A Rust `String` holds UTF-8 encoded data, possibly including zero bytes, with an explicitly known length.
- A C string (`char *`) holds byte values (which may or may not be signed), with its length implicitly determined by the first zero byte (`\0`) found in the data.

Fortunately, dealing with C-style strings in Rust is comparatively straightforward, because the Rust library designers have already done the heavy lifting by providing a pair of types to encode them. **Use the `CString` type** to hold (owned) strings that need to be interoperable with C, and use the corresponding `CStr` type when dealing with borrowed string values. The latter type includes the `as_ptr()` method, which can be used to pass the string's

contents to any FFI function that's expecting a `const char*` C string. Note that the `const` is important: this can't be used for an FFI function that needs to modify the contents (`char *`) of the string that's passed to it.

Lifetimes

Most data structures are too big to fit in a register and so have to be held in memory instead. That in turn means that access to the data is performed via the location of that memory. In C terms, this means a *pointer*: a number that encodes a memory address—with no other semantics attached (Item 8).

In Rust, a location in memory is generally represented as a *reference*, and its numeric value can be extracted as a *raw pointer*, ready to feed into an FFI boundary:

```
extern "C" {
    // C function that does some operation on the contents
    // of an `FfiStruct`.
    pub fn use_struct(v: *const FfiStruct) -> u32;
}


let v = FfiStruct {
    byte: 1,
    integer: 42,
};
let x = unsafe { use_struct(&v as *const FfiStruct) };
```

However, a Rust reference comes with additional constraints around the *lifetime* of the associated chunk of memory, as described in Item 14; these constraints get lost in the conversion to a raw pointer.

As a result, the use of raw pointers is inherently `unsafe` , as a marker that Here Be Dragons: the C code on the other side of the FFI boundary could do any number of things that will destroy Rust's memory safety:

- The C code could hang onto the value of the pointer and use it at a later point when the associated memory has either been freed from the heap or reused on the stack (*use-after-free*).
- The C code could decide to cast away the `const` -ness of a pointer that's passed to it and modify data that Rust expects to be immutable.
- The C code is not subject to Rust's `Mutex` protections, so the specter of data races (Item 17) rears its ugly head.
- The C code could mistakenly return associated heap memory to the allocator (by calling C's `free()` library function), meaning that the *Rust* code might now be performing use-after-free operations.

All of these dangers form part of the cost-benefit analysis of using an existing library via FFI. On the plus side, you get to reuse existing code that's (presumably) in good working order, with only the need to write (or auto-generate) corresponding declarations. On the minus side, you lose the memory protections that are a big reason to use Rust in the first place.

As a first step to reduce the chances of memory-related problems, **allocate and free memory on the same side of the FFI boundary**. For example, this might appear as a symmetric pair of functions:

```
/* C functions. */

/* Allocate an `FfiStruct` */
FfiStruct* new_struct(uint32_t v);
/* Free a previously allocated `FfiStruct` */
void free_struct(FfiStruct* s);
```

with corresponding Rust FFI declarations:

```
extern "C" {
    // C code to allocate an `FfiStruct`.
    pub fn new_struct(v: u32) -> *mut FfiStruct;
    // C code to free a previously allocated `FfiStruct`.
    pub fn free_struct(s: *mut FfiStruct);
}
```

To make sure that allocation and freeing are kept in sync, it can be a good idea to implement an RAII wrapper that automatically prevents C-allocated memory from being leaked (Item 11). The wrapper structure owns the C-allocated memory:

```
/// Wrapper structure that owns memory allocated by the C library.
struct FfiWrapper {
    // Invariant: inner is non-NULL.
    inner: *mut FfiStruct,
}
```

and the `Drop` implementation returns that memory to the C library to avoid the potential for leaks:

```
/// Manual implementation of [`Drop`], which ensures that memory allocated
/// by the C library is freed by it.
impl Drop for FfiWrapper {
    fn drop(&mut self) {
        // Safety: `inner` is non-NULL, and besides `free_struct()` copes
        // with NULL pointers.
        unsafe { free_struct(self.inner) }
    }
}
```

The same principle applies to more than just heap memory: **implement `Drop` to apply RAII to FFI-derived resources**—open files, database connections, etc. (see Item 11).

Encapsulating the interactions with the C library into a wrapper `struct` also makes it possible to catch some other potential footguns, for example, by transforming an otherwise invisible failure into a `Result`:

```
type Error = String;

impl FfiWrapper {
    pub fn new(val: u32) -> Result<Self, Error> {
        let p: *mut FfiStruct = unsafe { new_struct(val) };
        // Raw pointers are not guaranteed to be non-NULL.
        if p.is_null() {
            Err("Failed to get inner struct!".into())
        } else {
            Ok(Self { inner: p })
        }
    }
}
```

The wrapper structure can then offer safe methods that allow use of the C library's functionality:

```
impl FfiWrapper {
    pub fn set_byte(&mut self, b: u8) {
        // Safety: relies on invariant that `inner` is non-NULL.
        let r: &mut FfiStruct = unsafe { &mut *self.inner };
        r.byte = b;
    }
}
```

Alternatively, if the underlying C data structure has an equivalent Rust mapping, and if it's safe to directly manipulate that data structure, then implementations of the `AsRef` and `AsMut` traits (described in Item 8) allow more direct use:

```
impl AsMut<FfiStruct> for FfiWrapper {
    fn as_mut(&mut self) -> &mut FfiStruct {
        // Safety: `inner` is non-NULL.
        unsafe { &mut *self.inner }
    }
}


let mut wrapper = FfiWrapper::new(42).expect("real code would check");
// Directly modify the contents of the C-allocated data structure.
wrapper.as_mut().byte = 12;
```

This example illustrates a useful principle for dealing with FFI: **encapsulate access to an `unsafe` FFI library inside safe Rust code**. This allows the rest of the application to follow the advice in Item 16 and avoid writing `unsafe` code. It also concentrates all of the dangerous code in one place, which you can then study (and test) carefully to uncover problems—and treat as the most likely suspect when something does go wrong.

Invoking Rust from C

What counts as "foreign" depends on where you're standing: if you're writing an application in C, then it may be a *Rust* library that's accessed via a foreign function interface.

The basics of exposing a Rust library to C code are similar to the opposite direction:

- Rust functions that are exposed to C need an `extern "C"` marker to ensure they're C-compatible.
- Rust symbols are name mangled by default (like C++),[3] so function definitions also need a `#[no_mangle]` attribute to ensure that they're accessible via a simple name. This in turn means that the function name is part of a single global namespace that can clash with any other symbol defined in the program. As such, **consider using a prefix for exposed names** to avoid ambiguities (`mylib_...`).
- Data structure definitions need the `#[repr(C)]` attribute to ensure that the layout of the contents is compatible with an equivalent C data structure.

Also like the opposite direction, more subtle problems arise when dealing with pointers, references, and lifetimes. A C pointer is different from a Rust reference, and you forget that at your peril:

```
#[no_mangle]
pub extern "C" fn add_contents(p: *const FfiStruct) -> u32 {
    // Convert the raw pointer provided by the caller into
    // a Rust reference.
    let s: &FfiStruct = unsafe { &*p }; // Ruh-roh
    s.integer + s.byte as u32
}
```

```
/* C code invoking Rust. */
uint32_t result = add_contents(NULL); // Boom!
```

When you're dealing with raw pointers, it's your responsibility to ensure that any use of them complies with Rust's assumptions and guarantees around references:

```
#[no_mangle]
pub extern "C" fn add_contents_safer(p: *const FfiStruct) -> u32 {
    let s = match unsafe { p.as_ref() } {
        Some(r) => r,
        None => return 0, // Pesky C code gave us a NULL.
    };
    s.integer + s.byte as u32
}
```

In these examples, the C code provides a raw pointer to the Rust code, and the Rust code converts it to a reference in order to operate on the structure. But where did that pointer come from? What does the Rust reference refer to?

The very first example in Item 8 showed how Rust's memory safety prevents references to expired stack objects from being returned; those problems reappear if you hand out a raw pointer:

```
impl FfiStruct {
    pub fn new(v: u32) -> Self {
        Self {
            byte: 0,
            integer: v,
        }
    }
}

// No compilation errors here.
#[no_mangle]
pub extern "C" fn new_struct(v: u32) -> *mut FfiStruct {
    let mut s = FfiStruct::new(v);
    &mut s // return raw pointer to a stack object that's about to expire!
}
```

Any pointers passed back from Rust to C should generally refer to heap memory, not stack memory. But naively trying to put the object on the heap via a `Box` doesn't help:

```
// No compilation errors here either.
#[no_mangle]
pub extern "C" fn new_struct_heap(v: u32) -> *mut FfiStruct {
    let s = FfiStruct::new(v); // create `FfiStruct` on stack
    let mut b = Box::new(s); // move `FfiStruct` to heap
    &mut *b // return raw pointer to a heap object that's about to expire!
}
```

The owning `Box` is on the stack, so when it goes out of scope, it will free the heap object and the returned raw pointer will again be invalid.

The tool for the job here is `Box::into_raw`, which abnegates responsibility for the heap object, effectively "forgetting" about it:

```
#[no_mangle]
pub extern "C" fn new_struct_raw(v: u32) -> *mut FfiStruct {
    let s = FfiStruct::new(v); // create `FfiStruct` on stack
    let b = Box::new(s); // move `FfiStruct` to heap

    // Consume the `Box` and take responsibility for the heap memory.
    Box::into_raw(b)
}
```

This raises the question of how the heap object now gets freed. The previous advice was to perform allocation and freeing of memory on the same side of the FFI boundary, which means that we need to persuade the Rust side of things to do the freeing. The corresponding tool for the job is `Box::from_raw`, which builds a `Box` from a raw pointer:

```
#[no_mangle]
pub extern "C" fn free_struct_raw(p: *mut FfiStruct) {
    if p.is_null() {
        return; // Pesky C code gave us a NULL
    }
    let _b = unsafe {
        // Safety: p is known to be non-NULL
        Box::from_raw(p)
    };
} // `_b` drops at end of scope, freeing the `FfiStruct`
```

This still leaves the Rust code at the mercy of the C code; if the C code gets confused and asks Rust to free the same pointer twice, Rust's allocator is likely to become terminally confused.

That illustrates the general theme of this Item: using FFI exposes you to risks that aren't present in standard Rust. That may well be worthwhile, as long as you're aware of the dangers and costs involved. Controlling the details of what passes across the FFI boundary helps to reduce that risk but by no means eliminates it.

Controlling the FFI boundary for C code invoking Rust also involves one final concern: if your Rust code ignores the advice in Item 18, you should **prevent `panic!`s from crossing the FFI boundary**, as this always results in undefined behavior—undefined but bad![4]

Things to Remember

- Interfacing with code in other languages uses C as a least common denominator, which means that symbols all live in a single global namespace.
- Minimize the chances of problems at the FFI boundary by doing the following:
  - Encapsulating `unsafe` FFI code in safe wrappers
  - Allocating and freeing memory consistently on one side of the boundary or the other
  - Making data structures use C-compatible layouts
  - Using sized integer types
  - Using FFI-related helpers from the standard library
  - Preventing `panic!`s from escaping from Rust

---

[1] If the FFI functionality you want to use is part of the standard C library, then you don't need to create these declarations—the `libc` crate already provides them.

[2] A corresponding `links` key in the *Cargo.toml* manifest can help to make this dependency visible to Cargo.

[3] A Rust equivalent of the `c++filt` tool for translating mangled names back to programmer-visible names is `rustfilt`, which builds on the `rustc-demangle` command.

[4] Note that Rust version 1.71 includes the C-unwind ABI, which makes some cross-language unwinding functionality possible.

Item 35: Prefer `bindgen` to manual FFI mappings

Item 34 discussed the mechanics of invoking C code from a Rust program, describing how declarations of C structures and functions need to have an equivalent Rust declaration to allow them to be used over FFI. The C and Rust declarations need to be kept in sync, and Item 34 also warned that the toolchain wouldn't help with this—mismatches would be silently ignored, hiding problems that would arise later.

Keeping two things perfectly in sync sounds like a good target for automation, and the Rust project provides the right tool for the job: `bindgen`. The primary function of `bindgen` is to parse a C header file and emit the corresponding Rust declarations.

Taking some of the example C declarations from Item 34:

```
/* File lib.h */
#include <stdint.h>

typedef struct {
    uint8_t byte;
    uint32_t integer;
} FfiStruct;

int add(int x, int y);
uint32_t add32(uint32_t x, uint32_t y);
```

the `bindgen` tool can be manually invoked (or invoked by a `build.rs` build script) to create a corresponding Rust file:

```
% bindgen --no-layout-tests \
          --allowlist-function="add.*" \
          --allowlist-type=FfiStruct \
          -o src/generated.rs \
          lib.h
```

The generated Rust is identical to the handcrafted declarations in Item 34:

```
/* automatically generated by rust-bindgen 0.59.2 */

#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct FfiStruct {
    pub byte: u8,
    pub integer: u32,
}
extern "C" {
    pub fn add(
        x: ::std::os::raw::c_int,
        y: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn add32(x: u32, y: u32) -> u32;
}
```

and can be pulled into Rust code with the source-level `include!` macro:

```
// Include the auto-generated Rust declarations.
include!("generated.rs");
```

For anything but the most trivial FFI declarations, **use `bindgen` to generate Rust bindings for C code**—this is an area where machine-made, mass-produced code is definitely preferable to artisanal handcrafted declarations. If a C function definition changes, the C compiler will complain if the C declaration no longer matches the C definition, but nothing will complain that a handcrafted Rust declaration no longer matches the C declaration; auto-generating the Rust declaration from the C declaration ensures that the two stay in sync

This also means that the `bindgen` step is an ideal candidate to include in a CI system (Item 32); if the generated code is included in source control, the CI system can error out if a freshly generated file doesn't match the checked-in version.

The `bindgen` tool comes into its own when you're dealing with an existing C codebase that has a large API. Creating Rust equivalents to a big *lib_api.h* header file is manual and tedious, and therefore error-prone—and as noted, many categories of mismatch error will not be detected by the toolchain. `bindgen` also has a panoply of options that allow specific subsets of an API to be targeted (such as the `--allowlist-function` and `--allowlist-type` options previously illustrated).[1]

This also allows a layered approach for exposing an existing C library in Rust; a common convention for wrapping some `xyzzy` library is to have the following:

- An `xyzzy-sys` crate that holds (just) the `bindgen`-erated code—use of which is necessarily `unsafe`
- An `xyzzy` crate that encapsulates the `unsafe` code and provides safe Rust access to the underlying functionality

This concentrates the `unsafe` code in one layer and allows the rest of the program to follow the advice in Item 16.

Beyond C

The `bindgen` tool has the ability to handle some C++ constructs but only a subset and in a limited fashion. For better (but still somewhat limited) integration, **consider using the `cxx` crate for C++/Rust interoperation**. Instead of generating Rust code from C++ declarations, `cxx` takes the approach of auto-generating *both* Rust and C++ code from a common schema, allowing for tighter integration.

---

[1] The example also used the `--no-layout-tests` option to keep the output simple; by default, the generated code will include `#[test]` code to check that structures are indeed laid out correctly.

Afterword

Hopefully the advice, suggestions, and information in this book will help you become a fluent, productive Rust programmer. As the Preface describes, this book is intended to cover the second step in this process, after you've learned the basics from a core Rust reference book. But there are more steps you can take and directions to explore:

- *Async Rust* is not covered in this book but is likely to be needed for efficient, concurrent server-side applications. The online documentation provides an introduction to `async`, and the forthcoming *Async Rust* by Maxwell Flitton and Caroline Morton (O'Reilly, 2024) may also help.
- Moving in the other direction, *bare-metal Rust* might align with your interests and requirements. This goes beyond the introduction to `no_std` in Item 33 to a world where there's no operating system and no allocation. The bare-metal Rust section of the Comprehensive Rust online course provides a good introduction here.
- Regardless of whether your interests are low-level or high-level, the `crates.io` ecosystem of third-party, open source crates is worth exploring—and contributing to. Curated summaries like *blessed.rs* or *lib.rs* can help navigate the huge number of possibilities.
- Rust discussion forums such as the Rust language forum or Reddit's r/rust can provide help—and include a searchable index of questions that have been asked (and answered!) previously.
- If you find yourself relying on an existing library that's not written in Rust (as per Item 34), you could *rewrite it in Rust* (RiiR). But don't underestimate the effort required to reproduce a battle-tested, mature codebase.
- As you become more skilled in Rust, Jon Gjengset's *Rust for Rustaceans* (No Starch, 2022) is an essential reference for more advanced aspects of Rust.

Good luck!

Index

Effective Rust by David Drysdale