

Effective Rust

35 Specific Ways to Improve Your Rust Code

David Drysdale

Introduction

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

Scott Meyers' original *Effective C++* book was phenomenally successful because it introduced a new style of programming book, focused on a collection of guidelines that had been learned from real world experience of creating software in C++. Significantly, those guidelines were explained in the context of the reasons why they were necessary – allowing the reader to decide for themselves whether their particular scenario warranted breaking the rules.

The first edition of *Effective C++* was published in 1992, and at that time C++, although young, was already a subtle language that included many footguns; having a guide to the interactions of its different features was essential.

Rust is also a young language, but in contrast to C++ it is remarkably free of footguns. The strength and consistency of its type system means that if a Rust program compiles, there is already a decent chance that it will work – a phenomenon previously only observed with more academic, less accessible languages such as Haskell.

This safety – both type safety and memory safety – does come with a cost, though. 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 just working, but the struggle to get it to compile is real – even with the Rust compiler's remarkably helpful error diagnostics.

As a result, this book is aimed at a slightly different level than other *Effective <Language>* books; there are more Items that cover the concepts that are new with Rust, even though the official documentation already includes good introductions of these topics. These Items have titles like "*Understand...*" and "*Familiarize yourself with...*".

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.

That said, the text still assumes an understanding of the basics of the language. It also assumes the 2018 edition of Rust, using the stable toolchain.

The specific `rustc` version used for code fragments and error messages is 1.60. Rust is now stable enough (and has sufficient back-compatibility guarantees) that the code fragments are unlikely to need changes for later versions, but the error messages may vary with your particular compiler version.

The text also has a number of references to and comparisons with C++, as this is





probably the closest equivalent language (particularly with C++11's move semantics), and the most likely previous language that newcomers to Rust will have encountered.

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

- **Types:** Suggestions that revolve around Rust's core type system.
- **Concepts:** Core ideas that form the design of Rust.
- **Dependencies:** Advice for working with Rust's package ecosystem.
- **Tooling:** Suggestions on how to improve your codebase by going beyond just the Rust compiler.
- **Asynchronous Rust:** Advice for working with Rust's `async` mechanisms.
- **Beyond Standard Rust:** Suggestions for when you have to work beyond Rust's standard, safe environment.

Although the "Concepts" section is arguably more fundamental than the "Types" section, it is deliberately placed second so that readers who are reading from beginning to end can build up some confidence first.

The following markers, borrowing Ferris from the [Rust Book](#), are used to identify code that isn't right in some way.

Ferris	Meaning
	This code does not compile!
	This code panics!
	This code block contains unsafe code.
	This code does not produce the desired behaviour.

Acknowledgments

My thanks go to:

- Tiziano Santoro, from whom I originally learned many things about Rust.
- 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 ProtonMail user, who pointed out mistakes in the text.

Types

The first section 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 other core pillar of Rust's type system is the `trait` type. Traits are roughly equivalent to interface types in other languages, but they are also tied to Rust's *generics* ([Item 12](#)), to allow interface re-use without runtime overhead.

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

"who called them programmers and not type writers" – [@thingskatedid](#)

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's also signed (`isize`) and unsigned (`usize`) integers whose size matches the pointer size on the target system. Rust isn't a language where you're going to be doing much in the way of converting between pointers and integers, so that characterization 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++ – attempting to put a quart (`i32`) into a pint pot (`i16`) generates a compile-time error.

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

```
error[E0308]: mismatched types
  --> use-types/src/main.rs:14:22
14 |         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
14 |         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. It 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 alter the value, 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":

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

```
error[E0308]: mismatched types
  --> use-types/src/main.rs:23:22
   |
23 |         let y: i64 = x;
   |                   ^ expected `i64`, found `i32`
   |                   |
   |                   expected due to this
help: you can convert an `i32` to an `i64`
23 |         let y: i64 = x.into();
   |                        +++++++
```

Here, the suggested solution doesn't raise the spectre of error handling, but the conversion does still need to be explicit. We'll discuss type conversions in more detail later ([Item 6](#)).

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 4 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 than a `char` , which in turn is different than a sequence of UTF-8 bytes, which in turn is different than a sequence of arbitrary bytes, and it's up to you to specify exactly which you mean¹. 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 point² can always be represented in 32 bits, so `'a'` as `u32` is allowed, but the other direction is trickier (as there are `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 is marked `unsafe` as a result, forcing the caller to use `unsafe` too ([Item 16](#)).

Aggregate Types

Moving on to aggregate types, Rust has:

- **Arrays**, which 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**, which hold instances of multiple heterogeneous types, where the number of elements and their types are known at compile time, for example `(WidgetOffset, WidgetSize, WidgetColour)`. 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...
- **Structs**, which also hold instances of heterogeneous types known at compile time, but which allows both the overall type and the individual fields to be referred to by name.

The *tuple struct* is a cross-breed of a `struct` with 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`, etc.

```
struct TextMatch(usize, String);
let m = TextMatch(12, "needle".to_owned());
assert_eq!(m.0, 12);
```

This brings us to the jewel in the crown of Rust's type system, the `enum`.

In its basic form, 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 or string value attached.

```
enum HttpStatusCode {
    Ok = 200,
    NotFound = 404,
    Teapot = 418,
}
let code = HttpStatusCode::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, /* colour= */ false);
```

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


```

pub enum Sides {
    Both,
    Single,
}

pub enum Output {
    BlackAndWhite,
    Colour,
}

pub fn print_page(sides: Sides, colour: 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]: mismatched types
  --> use-types/src/main.rs:89:20
   |
89 |         print_page(Output::BlackAndWhite, Sides::Single);
   |                                ^^^^^^^^^^^^^^^^^^^^^^^^^ expected enum
`enums::Sides`, found enum `enums::Output`
error[E0308]: mismatched types
  --> use-types/src/main.rs:89:43
   |
89 |         print_page(Output::BlackAndWhite, Sides::Single);
   |                                ^^^^^^^^^^^^^^^^^^^^^^^^^ expected enum `enums::Output`, found enum `enums::Sides`

```

(Using the newtype pattern ([Item 7](#)) to wrap a `bool` also achieves type safety and maintainability; it's generally best to use that 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 {
    HttpStatusCode::Ok => "Ok",
    HttpStatusCode::NotFound => "Not found",
    // forgot to deal with the all-important "I'm a teapot"
code
};

```



```

error[E0004]: non-exhaustive patterns: `Teapot` not covered
--> use-types/src/main.rs:65:25
51 | /      enum HttpStatusCode {
52 | |          Ok = 200,
53 | |          NotFound = 404,
54 | |          Teapot = 418,
55 | |          ----- not covered
    | |          }
    | |_____- `HttpStatusCode` defined here
...
65 |         let msg = match code {
    |                        ^^^^ pattern `Teapot` not covered
    = help: ensure that all possible cases are being handled,
    possibly by adding wildcards or more match arms
    = note: the matched value is of type `HttpStatusCode`

```

The compiler forces the programmer to consider *all* of the possibilities³ that are represented by the `enum`, 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.)

enums 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 into 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:

```

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:

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

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

```
#[derive(Debug)]
enum Colour {
    Monochrome,
    Foreground(RgbColour),
}

struct DisplayProperties {
    x: u32,
    y: u32,
    colour: Colour,
}
```

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

Options and Errors

Returning to the power of the `enum`, there are two concepts that are so common that Rust includes built-in `enum` types to express them.

The first is the concept 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 `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](#) columns in SQL.)

One common edge case that's in the middle is a `String` which 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.

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, **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, and allows the use of standard transformations ([Item 3](#)) and error processing ([Item 4](#)); it also makes it possible to streamline error processing with the `?` operator.

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: This also means that adding a new variant to an existing `enum` in a library is a *breaking change* ([Item 21](#)): clients of the library will need to change their code to cope with the new variant. If an `enum` is really just an old-style list of values, this behaviour can be avoided by marking it as a `non_exhaustive` `enum`; see [Item 21](#).

Item 2: Use the type system to express common behaviour

[Item 1](#) discussed how to express data structures in the type system; this Item moves on to discuss the encoding of *behaviour* in Rust's type system.

Methods

The first place where behaviour is visible in Rust's type system is the addition of *methods* to data structures: functions that act on an item of that type, identified by `self`. This encapsulates related data and code together in a 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 gives a label for the behaviour that 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.

Abstracting Behaviour

Invoking a method always results in the same code being executed; all that changes from invocation to invocation is the data that the method operates on. That covers a lot of possible scenarios, but what if the *code* needs to vary at runtime?

Rust includes several features in its type system to accomodate this, which this section explores.

Function Pointers

The simplest behavioural abstraction is the *function pointer*: a pointer to (just) some code, with a type that reflects the signature of the function. The type is checked at compile time, so by the time the program runs the value is just the size of a pointer.

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

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 na
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}`
  --> use-types-behaviour/src/main.rs:117:21
117 |         assert!(op1 == op2);
    |                   ^^^   ^^^ fn(i32, i32) -> i32
{main::sum}
    |                   |
    |                   fn(i32, i32) -> i32 {main::sum}
help: you might have forgotten to call this function
117 |         assert!(op1( /* arguments */ ) == op2);
    |                   ++++++
help: you might have forgotten to call this function
117 |         assert!(op1 == op2( /* arguments */ ));
    |                   ++++++

```

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 which 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 6) to a `fn` type.

Closures

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 modified 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
  --> use-types-behaviour/src/main.rs:142:17
142 |         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 that:

- it can be built as part of an expression, and so 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 = || {
    // 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 9](#)) 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
  --> use-types-behaviour/src/main.rs:165:31
   |
165 |         modify_all(&mut data, |y| y + amount_to_add);
   |                                ^^^^^^^^^^^^^^^^^^^^^ expected
fn pointer, found closure
   |
   = note: expected fn pointer `fn(u32) -> u32`
           found closure `[closure@use-types-behaviour/src
/main.rs:165:31: 165:52]`
note: closures can only be coerced to `fn` types if they do not
capture any variables
  --> use-types-behaviour/src/main.rs:165:39
   |
165 |         modify_all(&mut data, |y| y + amount_to_add);
   |                                ^^^^^^^^^^^^^^^^^^^^^
`amount_to_add` captured here
```

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 behaviour.

- `FnOnce` describes a closure that can only be called *once*. If some part of its environment is moved into the closure, then that move can only happen once – there's no other copy of the source item to move from – and so the closure can only be invoked once.
- `FnMut` describes a closure that can be called repeatedly, and which can make changes to its environment because it *mutably* borrows from the environment.
- `Fn` describes a closure that can be called repeatedly, and which 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 traits¹ (unlike C++'s `operator()` overload).

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

- `FnOnce` : any moved values
- `FnMut` : any mutable references to values (`&mut T`)
- `Fn` : only normal references to values (`&T`).

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

- If something only expects to call a closure 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 only used once. The same reasoning also leads to advice to **prefer `Fn*` trait bounds to bare function pointers (`fn`)**.

Traits

The `Fn*` traits are more flexible than a bare function pointer, but they can still only describe the behaviour 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 behaviour in Rust's type system, the *trait*. A trait defines a set of related methods that some underlying item makes publicly available. Each method in a trait also has a *name*, providing a label which allows the compiler to disambiguate methods with the same signature, and more importantly which allows programmers to deduce the intent of the method.

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 methods (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 manner.

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

This leads to the same kind of advice that turns up for other OO-influenced languages²: **prefer accepting trait types to concrete types** if future flexibility is anticipated.

Sometimes, there is some behaviour that you want to distinguish in the type system, but which cannot be expressed as some specific method signature in a trait definition. For example, consider a 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 {
    /// Re-arrange contents into sorted order.
    fn sort(&mut self);
}

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

A marker trait has no methods, 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 honour system to preserve its invariants. **Use marker traits to distinguish behaviours that cannot be expressed in the trait method signatures.**

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

- as a *trait bound*, which constrains what types are acceptable for a generic data type or method at compile-time, or
- as a *trait object*, which constrains what types can be stored or passed to a method at run-time.

[Item 12](#) discusses the trade-offs between these in more detail.

A *trait bound* indicates that generic code which is parameterized by some type `T` can only be used when that type `T` implements some specific trait. The presence of the trait bound means that the implementation of the generic can use the methods from that trait, secure in the knowledge that the compiler will ensure that any `T` that compiles does indeed have those methods. This check happens at compile-time, when the generic is *monomorphized* (Rust's term for what C++ would call "template instantiation").

This restriction on the target type `T` is *explicit*, encoded in the trait bounds: the trait can only be implemented by types that satisfy the trait bounds. This is in contrast to 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 only compiles if all of the referenced methods are available at compile-time, but the checks are purely based on method and signature. (This "duck typing" leads to the chance of 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 behaviour.)

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 *no* trait bounds on `T`. Without a trait bound, the `Thing` can only perform operations that apply to *any* type `T`; this allows for 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: 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.

A *trait object* is the other way of making use of the encapsulation defined by a trait, but here different possible implementations of the trait are chosen at run-time rather than compile-time. This *dynamic dispatch* is analogous to the use of 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 (`&dyn Trait`) or a pointer (`Box<dyn Trait>`). This is because 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.

A similar concern means that traits used as trait objects cannot have methods that return the `Self` type, because 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 method `fn method<T>(t:T)` allows for the possibility of an infinite number of implemented methods, for all 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 methods becomes a finite set of *actually* invoked generic methods 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 run-time.

These two restrictions – no returning `Self` and no generic methods – are combined into 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 future.

2: For example, *Effective Java* 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 only performed when the template is instantiated, not when it is declared.

Item 3: Avoid matching Option and Result

Item 1 expounded the virtues of `enum` and showed how `match` expressions force the programmer to take all possibilities into account; this Item explores situations where you should prefer to avoid `match` expressions – explicitly at least.

Item 1 also introduced the two ubiquitous `enum`s that are provided by the Rust standard library:

- `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`).

For these particular `enum`s, *explicitly* using `match` often leads to code that is less compact than it needs to be, and which isn't idiomatic Rust.

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 absence of a value, and an associated error, is going to be something that the programmer has 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 manoeuvre and explicitly not cope with failure. 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!"),
};
```

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`**, even though this may involve conversions between different error types ([Item 4](#)).

`Result` has also 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
--> transform/src/main.rs:32:5
32 |         f.set_len(0); // Truncate the file
    |         ^^^^^^^^^^^
= note: `#[warn(unused_must_use)]` on by default
= note: this `Result` may be an `Err` variant, which should be handled
```

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(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 and the `return Err(...)` expression 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 to 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 sub-libraries, each of which uses different error types.

Error mapping in general is discussed in [Item 4](#); 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))
        }
    };
    // ...
}
```

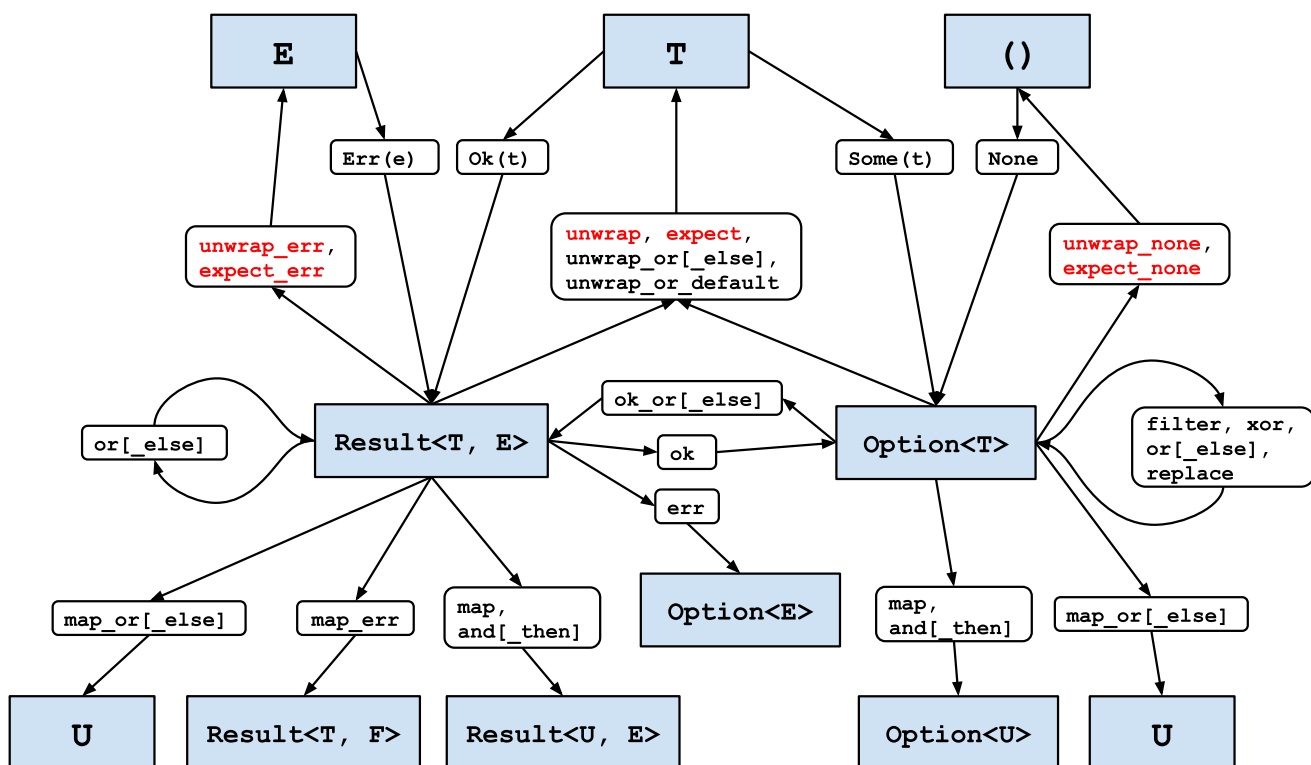
can be more succinctly and idiomatically expressed with the `.map_err()` transformation:

```
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 5](#)), 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 manoeuvre 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, as shown in the following map. In line with [Item 18](#), methods that can panic! are highlighted in red.



(The [online version](#) of this diagram is clickable: each box links to the relevant documentation.)

One common situation which isn't covered by the diagram is dealing with references. For example, consider a structure that optionally holds some data.

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

A method on this struct which 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

--> transform/src/main.rs:62:22

```
62 |         encrypt(&self.payload.unwrap_or(vec![]))
    |               ^^^^^^^^^^^^^^^^^ move occurs because
`self.payload` has type `Option<Vec<u8>>`, which does not implement
the `Copy` trait
```

help: consider borrowing the `Option`'s content

```
62 |         encrypt(&self.payload.as_ref().unwrap_or(vec![]))
    |                               ++++++
```

The error message describes exactly what's needed to make the code work, the `as_ref()` method¹ on `Option`. This method converts a reference-to-an- `Option` to be an `Option` - of-a-reference:

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

To sum up:

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

¹: Note that this method is separate from the `AsRef` trait, even though the method name is the same.

Item 4: Prefer idiomatic Error variants

[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 only really relevant 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 be made about whether the sub-error type information should be preserved.

The Error Trait

It's always good to understand what the standard traits ([Item 5](#)) 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**. However, if you're writing code for a `no_std` environment ([Item 33](#)), this recommendation is more awkward to apply – the `Error` trait is currently¹ implemented in `std`, not `core`, and so is not available.

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 both:

- the `Display` trait, meaning that it can be formatted with `{}`, and
- the `Debug` trait, meaning that it can be formatted with `{:?}`.

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

The only² method in the trait is `source()`, 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.

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 in `Errors`. 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 arbitrary types
```

```
--> errors/src/main.rs:20:5
```

```
20 |         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 arbitrary types
--> errors/src/main.rs:43:5
43 |         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 of how to solve the problem. Defining a tuple struct that wraps the `String` type (the "newtype pattern", [Item 7](#)) 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 6](#)):

```

impl std::convert::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));
        // ...
    }

```

For the error path here:

- `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,
        }
    }
}
```

This allows the error handling to be concise while still preserving all of the type information across different classes of error:

```
/// 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 sub-error types ([Item 6](#)):

```
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:

```
/// Return the first line of the given file.
pub fn first_line(filename: &str) -> Result<String, MyError> {
    let file = std::fs::File::open(filename)?; // via
    `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)?; // via
    `From<std::io::Error>`
    let result = String::from_utf8(buf)?; // via
    `From<std::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; **consider using the [thiserror](#) crate** to help with this, as it reduces the effort involved without adding an extra runtime dependency.

Trait Objects

The first approach to nested errors threw away all of the sub-error detail, just preserving some string output (`format!("{:?}", err)`). The second approach preserved the full type information for all possible sub-errors, but required a full enumeration of all possible

types of sub-error.

This raises the question: is there a half-way house between these two approaches, preserving sub-error information without needing to manually include every possible error type?

Encoding the sub-error 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 – `display()`, `debug()` and `source()` in turn – but wouldn't know the original static type of the sub-error.

```
#[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` would naively be expected to both implement the `Error` trait, and also to implement the `From<Error>` trait to allow sub-errors 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`:

```
error[E0119]: conflicting implementations of trait
`std::convert::From<WrappedError>` for type `WrappedError`
--> errors/src/main.rs:253:1
253 | 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](#), and which 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 of 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 re-use in a library, and code that forms a top-level application³.

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 towards the `enum`-style nested errors described previously (and also avoids a dependency on `anyhow` in the public API of the library, cf. [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.

Summary

This item has covered a lot of ground, so a summary is in order:

- 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 preserving those types is needed.
 - If not, consider using `anyhow` to wrap sub-errors in application code.
 - If they are needed, 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.

It's your decision, but whatever you decide, encode it in the type system ([Item 1](#)).

¹: At the time of writing, `Error` has been [moved to core](#) but is not yet available in stable Rust.

2: Or at least the only non-deprecated, stable method.

3: This section is inspired by [Nick Groenen's "Rust: Structuring and handling errors in 2020" article](#).

Item 5: Familiarize yourself with standard traits

Rust encodes key behavioural aspects of its type system in the type system itself, through a collection of fine-grained standard traits that describe those behaviours (cf. [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, through use of [derive macros](#). This leads to type definitions like:

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

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

A rough one-sentence summary of each of the standard traits that this Item covers is:

- [Clone](#) : Items of this type can make a copy of themselves when asked.
- [Copy](#) : If the compiler makes a bit-for-bit copy of this item's memory representation, 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.

Rust also allows various built-in unary and binary operators to be overloaded for user-defined types, by implementing various traits from the [std::ops module](#). These traits are not derivable, and are typically only needed for types that represent "algebraic" objects.

Other (non- `derive able`) standard traits are covered in other Items, and so are not included here. These include:

- [Fn](#) , [FnOnce](#) and [FnMut](#) : Items implementing this trait 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 which *may* hold nested sub-error 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 this trait can be automatically created from items of some other type, but with a possibility of failure in the latter case. See [Item 6](#).
- [Deref](#) and [DerefMut](#) : Items implementing this trait are pointer-like objects that can be dereferenced to get access to an inner item. See [Item 9](#).
- [Iterator](#) and friends: Items implementing this trait represent collections that can be iterated over. See [Item 10](#).
- [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](#).

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 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`; the macro implementation clones an aggregate type by cloning each of its members in turn, again, 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 behaviour 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 `RAll` 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:
 - Fields that are mutable references (`&mut T`), because the borrow checker ([Item 15](#)) only allows 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 book-keeping 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 not only can items of this type be copied (hence the `Clone` trait bound), but also 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.

In contrast to user-defined marker traits ([Item 1](#)), `Copy` has a special significance to the compiler¹ 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 in to k2
println!("k={:?}", k);
```



```
error[E0382]: borrow of moved value: `k`
  --> std-traits/src/main.rs:52:28
   |
50 |         let k = KeyId(42);
   |         - move occurs because `k` has type `main::KeyId`,
   |         which does not implement the `Copy` trait
51 |         let k2 = k; // value moves out of k in to k2
   |         - value moved here
52 |         println!("k={:?}", k);
   |         ^ value borrowed here after move

= note: this error originates in the macro
`$crate::format_args_nl` (in Nightly builds, run with -Z macro-
backtrace for more info)
```

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 behaviour of assignments – and this includes 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).

To finish this section, observe that 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 `main::KeyId` which implements the
`Copy` trait
--> std-traits/src/main.rs:68:18
68 |         let k3 = k.clone();
   |                   ^^^^^^^^^ help: try removing the `clone`
call: `k`
```

Default

The `Default` trait defines a *default constructor*, via a `default()` method. This trait can be derived for user-defined types, provided that all of the sub-types 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 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 Colour {
    red: u8,
    green: u8,
    blue: u8,
    alpha: u8,
}

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

This makes it much easier to initialize structures with lots of fields, only some of which have non-default values. (The builder pattern, [Item 8](#), 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` which 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](#)², 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.

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 re-order 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 relationship³ among collections: `{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 behaviour of your type, **be wary of implementing just `PartialOrd`** (a rare occasion that contradicts the advice of [Item 2](#) to encode behaviour in the type system) – it can lead to surprising results:

```

let x = Oddity(1);
let x2 = Oddity(1);
if x <= x2 {
    println!("It's possible to not hit this!");
}

let x = Oddity(1);
let y = Oddity(2);
// Programmers are unlikely to cover all possible arms...
if x <= y {
    println!("y is bigger"); // Not hit
} else if y <= x {
    println!("x is bigger"); // Not hit
} else {
    println!("Neither is bigger"); // This one
}

```



Hash

The `Hash` trait is used to produce a single value that has a high probability of being different for different items; this value is used as the basis for hash-bucket based data structures like `HashMap` and `HashSet`.

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. This is related to...
- 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.). A manual implementation of `Debug` can be appropriate when the automatically generated version would emit voluminous amounts of detail.

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

Operator Overloads

Similarly to C++, Rust allows various arithmetic and bitwise operators to be overloaded for user-defined types. This is useful for "algebraic" or bit-manipulation types (respectively) 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).

Continuing with the principle of least surprise, 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`), 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 $4 = 2 \times 2$ 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, the traits of this Item, all of which can be automatically derived **except `Display`**.

Trait	Compiler Use	Bound	Methods
<code>Clone</code>			<code>clone</code>
<code>Copy</code>	<code>let y = x;</code>	<code>Clone</code>	Marker trait

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

The operator overloads are in the next table. None of these can be derived.

Trait	Compiler Use	Bound	Methods
Add	<code>x + y</code>		add
AddAssign	<code>x += y</code>		add_assign
BitAnd	<code>x & y</code>		bitand
BitAndAssign	<code>x &= y</code>		bitand_assign
BitOr	<code>x y</code>		bitor
BitOrAssign	<code>x = y</code>		bitor_assign
BitXor	<code>x ^ y</code>		bitxor
BitXorAssign	<code>x ^= y</code>		bitxor_assign
Div	<code>x / y</code>		div
DivAssign	<code>x /= y</code>		div_assign
Mul	<code>x * y</code>		mul
MulAssign	<code>x *= y</code>		mul_assign
Neg	<code>-x</code>		neg
Not	<code>!x</code>		not
Rem	<code>x % y</code>		rem
RemAssign	<code>x %= y</code>		rem_assign
Shl	<code>x << y</code>		shl
ShlAssign	<code>x <<= y</code>		shl_assign
Shr	<code>x >> y</code>		shr
ShrAssign	<code>x >>= y</code>		shr_assign

Trait	Compiler Use	Bound	Methods
Sub	<code>x - y</code>		sub
SubAssign	<code>x -= y</code>		sub_assign

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

Trait	Item	Compiler Use	Bound	Methods
Fn	Item 2	<code>x(a)</code>	<code>FnMut</code>	call
FnMut	Item 2	<code>x(a)</code>	<code>FnOnce</code>	call_mut
FnOnce	Item 2	<code>x(a)</code>		call_once
Error	Item 4		<code>Display</code> <code>+ Debug</code>	[source]
From	Item 6			from
TryFrom	Item 6			try_from
Into	Item 6			into
TryInto	Item 6			try_into
AsRef	Item 9			as_ref
AsMut	Item 9			as_mut
Borrow	Item 9			borrow
BorrowMut	Item 9		<code>Borrow</code>	borrow_mut
ToOwned	Item 9			to_owned
Deref	Item 9	<code>*x</code> , <code>&x</code>		deref
DerefMut	Item 9	<code>*x</code> , <code>&mut x</code>	<code>Deref</code>	deref_mut

Trait	Item	Compiler Use	Bound	Methods
Index	Item 9	<code>x[idx]</code>		index
IndexMut	Item 9	<code>x[idx] = ...</code>	Index	index_mut
Pointer	Item 9	<code>format!("{:p}", x)</code>		fmt
Iterator	Item 10			next
Intolterator	Item 10	<code>for y in x</code>		into_iter
FromIterator	Item 10			from_iter
ExactSizeIterator	Item 10		Iterator	(size_hint)
DoubleEndedIterator	Item 10		Iterator	next_back
Drop	Item 11	<code>}</code> (end of scope)		drop
Sized	Item 15			Marker trait
Send	Item 17	cross-thread transfer		Marker trait
Sync	Item 17	cross-thread use		Marker trait

1: As do several of the other marker traits in `std::marker`.

2: 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.

3: More generally, any [lattice structure](#) also has a partial order.

Item 6: Understand type conversions

In general, Rust does not perform automatic conversion between types. This includes integral types, even when the transformation is "safe":

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

```
error[E0308]: mismatched types
  --> casts/src/main.rs:69:22
69 |         let y: u64 = x;
    |                   ^ expected `u64`, found `u32`
    |                   |
    |                   expected due to this
help: you can convert a `u32` to a `u64`
69 |         let y: u64 = x.into();
    |                        +++++++
```

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 latter two don't apply to conversions of user defined types (with a couple of exceptions), so the majority of this Item will focus on manual conversion – which the compiler error message also pointed towards.

However, sections at the end of the Item discuss casting and coercion – including the exceptions where they can apply to a user-defined type.

User-Defined Type Conversions

As with other features of the language ([Item 5](#)) 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:

- `From<T>`: Items of this type can be built from items of type `T`.

- `TryFrom<T>` : Items of this type can *sometimes* be built from items of type `T` .
- `Into<T>` : Items of this type can converted into items of type `T` .
- `TryInto<T>` : Items of this type can *sometimes* be converted into items of type `T` .

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 transmuted into a type `U` , isn't that the same as it being possible to create an item of type `U` by transmutation from an item of type `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 spiralling around in dizzy circles¹, 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 safe integral conversions (`From<u32>` for `u64`) and `TryFrom` implementations when the conversion isn't safe (`TryFrom<u64>` from `u32`).

There are also various other blanket trait implementations in addition to the `Into` version shown above, which you can find by searching for `impl<T> From<T> for ...` . These are almost all for *smart pointer* types, allowing the smart pointer to be automatically constructed from an instance of the type that it holds, so that methods that accept smart pointer parameters can also be called with plain old items; more on this below and in [Item 9](#).

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 above) 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 ² the helpfully named `Infallible` .

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
    }
}
```

Translating 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 7](#)) 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

```
error[E0308]: mismatched types
  --> casts/src/main.rs:82:29
    |
82 |         if is_iana_reserved(42) {
    |                             ^^ expected struct
    |                             `IanaAllocated`, found integer
```

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 which 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 which 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 9](#).

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 is a fairly limited set, and the only user-defined types it includes are "C-like" `enums` (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 conversions³:

```
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
  --> casts/src/main.rs:124:20
   |
124 |     let y: u16 = x.into();
   |                        ^^^^ the trait `From<u32>` is not
   | implemented for `u16`
   |
   = help: the following implementations were found:
           <u16 as From<NonZeroU16>>
           <u16 as From<bool>>
           <u16 as From<u8>>
           <f32 as From<i16>>
           and 71 others
   = note: required because of the requirements on the impl of
           `Into<u16>` for `u32`
```

For consistency and safety you should **prefer `from / into` conversions to `as` casts**, unless you understand and need the precise [casting semantics](#) (e.g for C interoperability).

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

- converting a mutable reference to a non-mutable references (so you can use a `&mut T` as the argument to a function that takes a `&T`)
- converting a reference to a raw pointer (this isn't `unsafe` – the unsafety happens at the point where you're foolish enough to *use* a raw pointer)
- converting a closure that happens not to capture any variables into a bare function pointer ([Item 2](#))
- converting an [array](#) to a [slice](#)
- converting a concrete item to a [trait object](#), for a trait that the concrete item implements
- converting⁴ an item lifetime to a "shorter" one ([Item 14](#)).

There are only two coercions whose behaviour can be affected by user-defined types. The first of these is 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 9](#)), 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, together with a pointer to the *vtable* for the concrete type's implementation of the trait – see [Item 9](#).

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 behaviour.

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

Item 7: 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, which is a pattern that's 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 behaviour. To illustrate this, imagine a project that's going to send a satellite to Mars¹. 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 force in Newton
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 pound
force seconds.
pub fn update_trajectory(force: f64) {
    // ...
}
```

Eventually these different parts eventually need to be joined together:

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

Ruh-roh.

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

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

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

```

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

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

```

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 `NewtonSeconds` value being used where a `PoundForceSeconds` value is expected:

```

    let thruster_force: NewtonSeconds =
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 NewtonSeconds(pub f64);

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

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

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

```

As the name implies, a newtype is a new type, and as such the compiler objects to type conversions (cf. [Item 6](#)):

```

    let thruster_force: NewtonSeconds =
thruster_impulse(direction);
    let new_direction = update_trajectory(thruster_force);

```

```
error[E0308]: mismatched types
  --> newtype/src/main.rs:76:43
76 |         let new_direction = update_trajectory(thruster_force);
   |                                           ^^^^^^^^^^^^^^^^^
expected struct `PoundForceSeconds`, found struct `NewtonSeconds`
```

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

```
struct DoubleSided(pub bool);

struct ColourOutput(pub bool);

fn print_page(sides: DoubleSided, colour: ColourOutput) {
    // ...
}

print_page(DoubleSided(true), ColourOutput(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 you can only implement a trait for a type if:

- you own the trait, or
- you own the type.

Attempting to implement a foreign trait for a foreign type:


```
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 arbitrary types
--> newtype/src/main.rs:125:1
125 | 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 prevents² you from writing `impl serde::Serialize for somecrate::SomeType`.

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

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

use std::fmt;
impl fmt::Display for MyRng {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> Result<(),
    fmt::Error> {
        write!(f, "<Rng 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: This is a sufficiently common problem for `serde` that it includes a [mechanism to help](#).

Item 8: Use builders for complex types

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, but does lead to more verbose boilerplate code than is ideal.

```
#[derive(Debug, Default)]
struct BaseDetails {
    given_name: String,
    preferred_name: Option<String>,
    middle_name: Option<String>,
    family_name: String,
    mobile_phone_e164: Option<String>,
}

// ...

let dizzy = BaseDetails {
    given_name: "Dizzy".to_owned(),
    preferred_name: None,
    middle_name: None,
    family_name: "Mixer".to_owned(),
    mobile_phone_e164: 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 5](#):

```
let dizzy = BaseDetails {
    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` only works 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)]
struct Details {
    given_name: String,
    preferred_name: Option<String>,
    middle_name: Option<String>,
    family_name: String,
    mobile_phone_e164: Option<String>,
    dob: chrono::Date<chrono::Utc>,
    last_seen: Option<chrono::DateTime<chrono::Utc>>,
}
```



```
error[E0277]: the trait bound `Date<Utc>: Default` is not satisfied
--> builders/src/main.rs:176:9
   |
169 |     #[derive(Debug, Default)]
   |               ----- in this derive macro expansion
...
176 |         dob: chrono::Date<chrono::Utc>,
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `Default` is
not implemented for `Date<Utc>`
   |
   = note: this error originates in the derive macro `Default` (in
Nightly builds, run with -Z macro-backtrace for more info)
```

The code can't implement `Default` for `chrono::Utc` because of the orphan rule, so this means that all of the fields have to be filled out manually:

```
use chrono::TimeZone;

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_e164: None,
    dob: chrono::Utc.ymd(1998, 11, 28),
    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.

```

struct DetailsBuilder(Details);

impl DetailsBuilder {
    /// Start building a new [`Details`] object.
    fn new(
        given_name: &str,
        family_name: &str,
        dob: chrono::Date<chrono::Utc>,
    ) -> Self {
        DetailsBuilder(Details {
            given_name: given_name.to_owned(),
            preferred_name: None,
            middle_name: None,
            family_name: family_name.to_owned(),
            mobile_phone_e164: None,
            dob,
            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.
fn preferred_name(mut self, preferred_name: &str) -> Self {
    self.0.preferred_name = Some(preferred_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.
fn just_seen(mut self) -> Self {
    self.0.last_seen = Some(chrono::Utc::now());
    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.
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",
        chrono::Utc.ymd(1998, 11, 28))
        .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",
    chrono::Utc.ymd(1998, 11, 28),
);
if informal {
    builder.preferred_name("Bob");
}
let bob = builder.build();
```



```
error[E0382]: use of moved value: `builder`
--> builders/src/main.rs:249:19
241 |         let builder = DetailsBuilder::new(
    |         ----- move occurs because `builder` has type
    |         `DetailsBuilder`, which does not implement the `Copy` trait
...
247 |         builder.preferred_name("Bob");
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `builder` moved due
to this method call
248 |     }
249 |     let bob = builder.build();
    |               ^^^^^^^^ value used here after move

note: this function takes ownership of the receiver `self`, which
moves `builder`
--> builders/src/main.rs:49:27
49 |     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",
chrono::Utc.ymd(1998, 11, 28));
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 repeatedly `build()` copies:

```

let smithy =
    DetailsBuilder::new("Agent", "Smith",
chrono::Utc.ymd(1999, 6, 11));
let clones = vec![smithy.build(), smithy.build(),
smithy.build()];

```

falls foul of the borrow checker, as you'd expect:

```

error[E0382]: use of moved value: `smithy`
  --> builders/src/main.rs:269:43
   |
267 |         let smithy =
   |         ----- move occurs because `smithy` has type
   |         `DetailsBuilder`, which does not implement the `Copy` trait
268 |         DetailsBuilder::new("Agent", "Smith",
   |         chrono::Utc.ymd(1999, 6, 11));
269 |         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.
fn just_seen(&mut self) -> &mut Self {
    self.0.last_seen = Some(chrono::Utc::now());
    self
}

```

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

```

let mut builder = DetailsRefBuilder::new(
    "Robert",
    "Builder",
    chrono::Utc.ymd(1998, 11, 28),
);
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 = DetailsRefBuilder::new(
    "Robert",
    "Builder",
    chrono::Utc.ymd(1998, 11, 28),
)
    .middle_name("the")
    .just_seen();
let bob = builder.build();

```



error[E0716]: temporary value dropped while borrowed

--> builders/src/main.rs:289:23

```

289 |         let builder = DetailsRefBuilder::new(
290 |             "Robert",
291 |             "Builder",
292 |             chrono::Utc.ymd(1998, 11, 28),
293 |         )
    |         ^ creates a temporary which is freed while still in
use
294 |         .middle_name("the")
295 |         .just_seen();
    |         - temporary value is freed at the end
of this statement
296 |         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, this can be worked around by letting the builder item have a name:


```
let mut builder = DetailsRefBuilder::new(
    "Robert",
    "Builder",
    chrono::Utc.ymd(1998, 11, 28),
);
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:

```
/// Construct a fully built [Details`] object.
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 9: 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 only be accessible 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, "normal" 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*, whose type 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 which satisfies 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 constraints that a Rust reference must point to a valid, correctly-aligned item are shared by C++'s reference types. However, C++ has no concept of lifetimes and so allows footguns¹ with dangling references:



```
// 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 make the equivalent code broken at compile time:

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

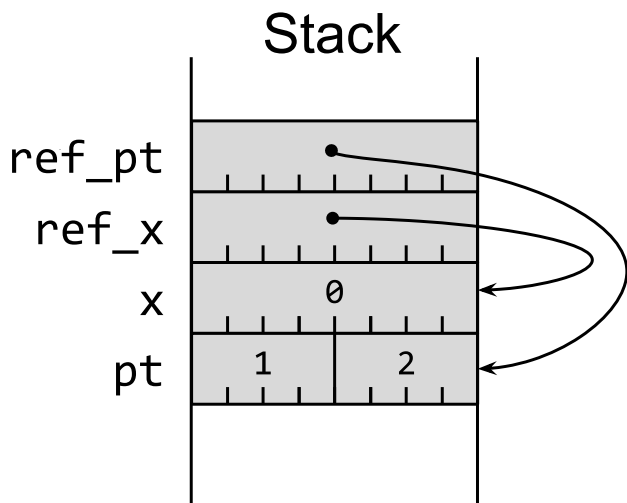
```
error[E0515]: cannot return reference to local variable `x`
  --> references/src/main.rs:399:5
   |
399 |     &x
   |     ^^ returns a reference to data owned by the current
function
```

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`).

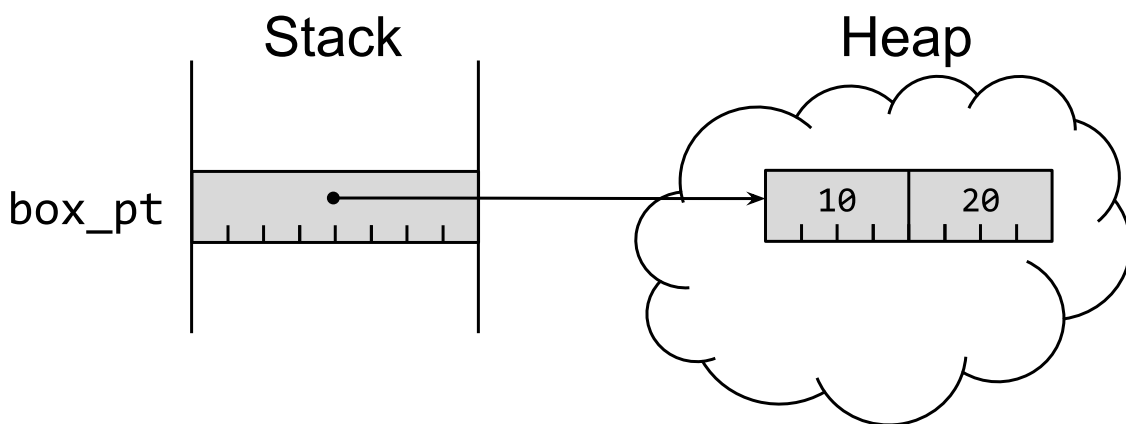
In the generated code, a Rust reference is a simple pointer, 8 bytes in size on a 64-bit platform (which this Item assumes throughout):

```
struct Point {
    x: u32,
    y: u32,
}
let pt = Point { x: 1, y: 2 };
let x = 0u64;
let ref_x = &x;
let ref_pt = &pt;
```



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 8 byte pointer value.

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



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 behaviour 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 6](#)).

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 behaviour in the compiler, but also 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:

- `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`).

We saw above that a function that takes a reference can automatically take any type that implements `Deref`, via the `Deref` coercion that the compiler performs. Such a function

can be made even more general, by making it *generic* over one of the `AsRef` / `AsMut` traits, and changing it to use `.as_ref()` on the input. This means it accepts the widest range of reference-like types:

```
fn show_as_ref<T: AsRef<Point>>(pt: T) {  
    let pt: &Point = pt.as_ref();  
    println!("{}", pt.x, pt.y);  
}
```

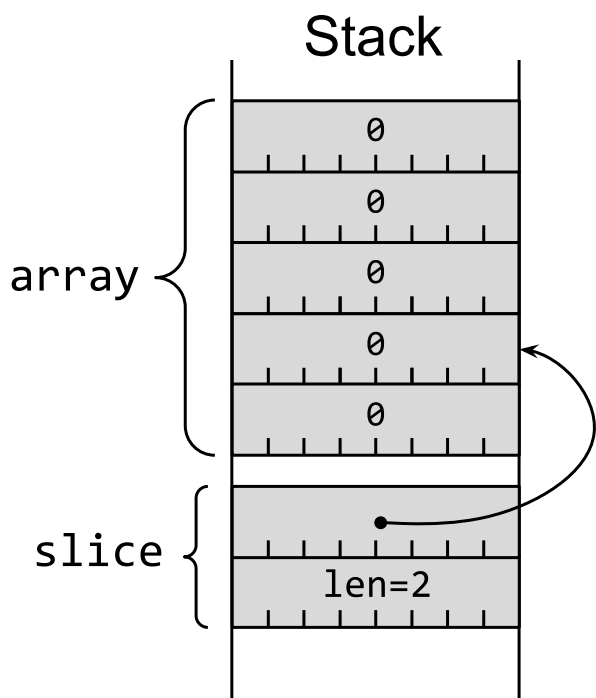
Fat Pointer Types

Rust has two built-in *fat pointer* types: types that act as pointers, but which hold additional information about the thing they are pointing to.

The first such 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`.

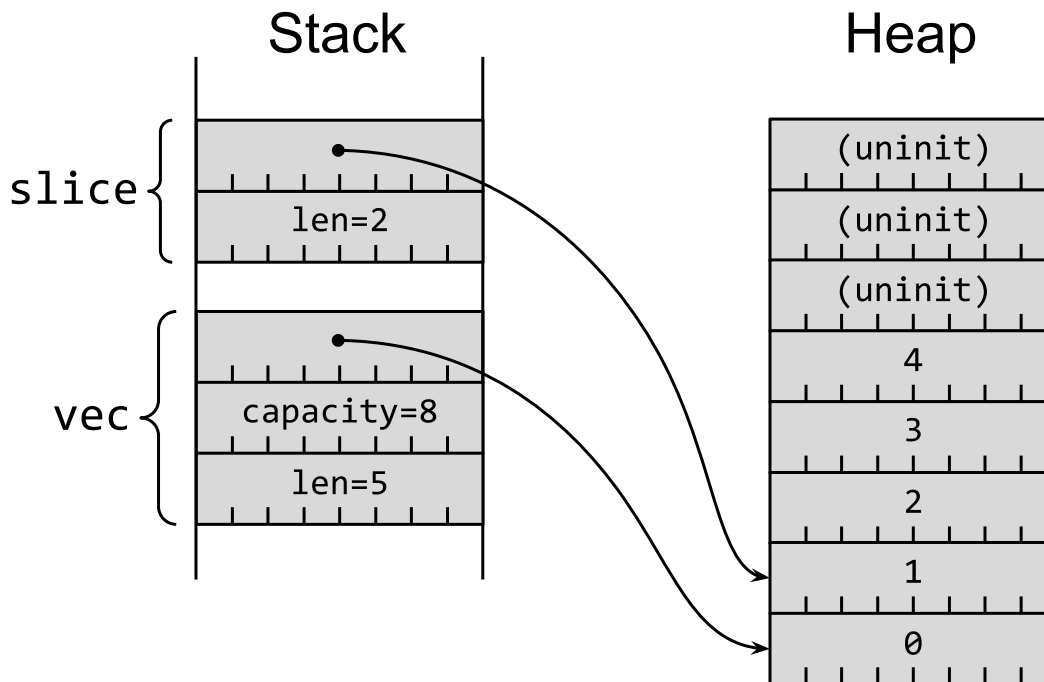
The notional type `[T]` can't be instantiated, but there are two common containers that embody it. The first is the *array*: a contiguous collection of values whose size is known at compile time. A slice can therefore refer to a subset of an array:

```
let array = [0u64; 5];  
let slice = &array[1..3];
```



The other common container for contiguous values is a `Vec<T>`. This holds a contiguous collection of values whose size can vary, and whose contents are held on the heap. A slice can therefore refer to a subset of a vector:

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



There's quite a lot going on under the covers for the expression `&vec[1..3]`, so it's worth breaking 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 `vec[]` part is an [indexing expression](#); the compiler converts this into an invocation of the `Index` trait's `index` method on `vec`, together with a dereference (i.e. `*vec.index()`).
 - The equivalent trait for mutable expressions is `IndexMut`.
- `vec[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`.
- `&vec[1..3]` un-does the dereference, resulting in a final expression type of `&[u64]`.

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](#))².

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);

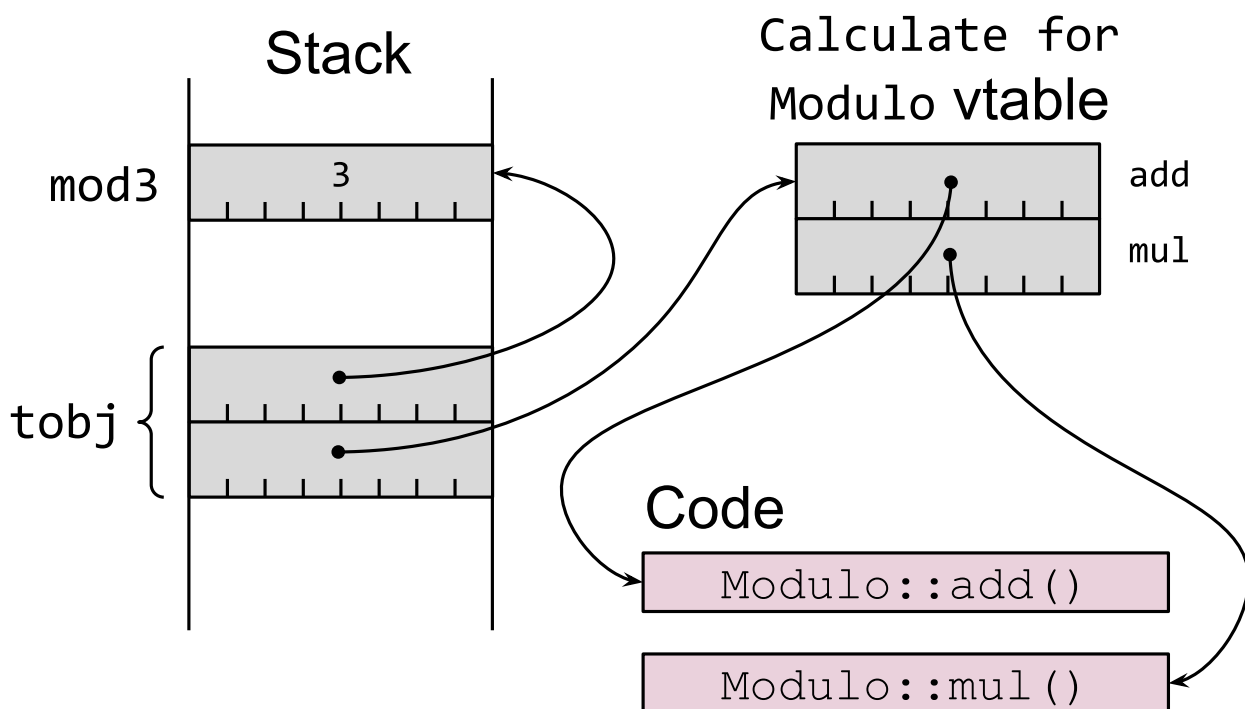
```

can be converted to a trait object of type `&dyn Trait` (where 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);

```



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 means that:

- 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, it's worth mentioning the `Cow` type 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 stay as borrowed data right up to the point where it needs to be modified, but 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 described above. 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 behaviour, 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, so acts like a `Box<T>` in many ways.

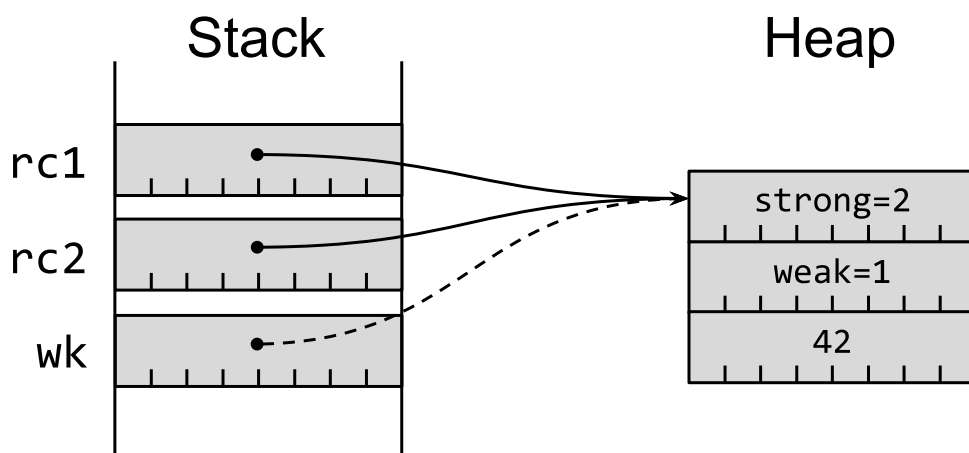
This is useful for data structures where the same item can be reached in different ways, but it removes one of Rust's core rules around ownership – that each item has only one owner. Relaxing this rule means that it is now possible to leak data: if item A has an `Rc` pointer to item B, and item B has an `Rc` pointer to A, then the pair will never be dropped. 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 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 pair of reference counts together with

the referenced items, all stored on the heap.

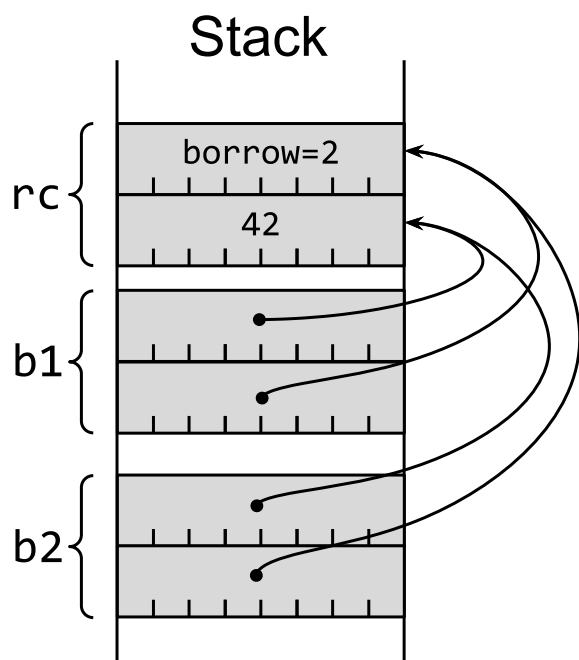
```
use std::rc::Rc;
let rc1: Rc<u64> = Rc::new(42);
let rc2 = rc1.clone();
let wk = Rc::downgrade(&rc1);
```



An `Rc` on its own gives you the ability to reach an item in different ways, but when you reach that item you can only modify it (via `get_mut`) 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 another smart pointer type...

The next smart pointer type `RefCell<T>` relaxes the rule (Item 15) that an item can only be mutated 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 only allows `&self`. However, it also incurs costs: as well as the extra storage overhead (an extra `isize` to track current borrows), the normal borrow checks are moved from compile-time to run-time.

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



The run-time 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 run-time 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 5](#)), 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 only suitable for single threaded use; their implementations assume that there is no concurrent access to their internals. If this is not the case, then different smart pointers are needed, which include the additional synchronization overhead.

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 multi-threaded code (but this only guards against *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 behaviour of the different smart pointers implies. To borrow³ [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 behaviours.

1: Albeit with a warning from modern compilers.

2: 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.

3: Pun intended

Item 10: 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) only had `while (condition) { ... }`, but with the arrival of C the common scenario of iterating through indexes of an array was made 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++ improved convenience and scoping further by allowing the loop variable declaration to be embedded in the `for` statement (and 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, and 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 container¹, until exhausted.
- **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*. 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 to give guidance as to when it's a good idea.

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 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 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 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`).

Collections that allow iteration over their contents – called *iterables* – 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`
  --> iterators/src/main.rs:156:35
   |
148 |         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
149 |         for item in collection {
   |         |
   |         | -----
   |         | `collection` moved due to this implicit call
   |         | to `.into_iter()`
   |         | help: consider borrowing to avoid moving into
   |         | the for loop: `&collection`
   |         |
   |         |
156 |         println!("Collection = {:?}" , collection);
   |         |                                     ^^^^^^^^^^ value borrowed
   |         |                                     here after move
   |         |
   |         | note: this function takes ownership of the receiver `self`, which
   |         | moves `collection`

```

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

To ensure that behaviour is clear, the examples here use an `Item` type that is *not* `Copy` ([Item 5](#)), 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 `Intolterator` 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:

```

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

```

If it makes sense to provide iteration over mutable references², 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 equivalent an `iter_mut()` method if appropriate, with the same behaviour 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 only emits every `n`-th 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| {...})` is the most general version, repeatedly applying a closure to transform each item in turn. 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`).
- `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| {...})` is the most general version, applying a `bool`-returning closure to each item reference to determine whether it should be passed through.
- `take_while()` and `skip_while()` are mirror images of each other, emitting either an initial subrange or a final subrange of the iterator, based on a predicate.

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 flattening 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 transmute it into exactly the right form 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 could do (the exceptions are described below), 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()`, for summing a collection of numeric values (integers or floats).
- `product()`, for multiplying together a collection of numeric values.
- `min()` and `max()`, for finding the extreme values of a collection, relative to the `Item`'s `Ord` implementation (see [Item 5](#)).
- `min_by(f)` and `max_by(f)`, for finding the extreme values of a collection, relative to a user-specified comparison function `f`.
- `reduce(f)` is a more general operation that encompasses the previous methods, building 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.
- `fold(f)` is a generalization of `reduce`, allowing the "accumulated value" to be of an arbitrary type (not just the `Iterator::Item` type).
- `scan(f)` generalizes in a slightly different way, giving the closure a mutable reference to some internal state at each step.

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 counter-example 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)` is like `for_each`, but the closure can fail.
- `try_fold(f)` is like `fold`, but the closure can fail.
- `try_find(f)` is 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();
```

(As an aside, 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:

- `unzip()`, which divides an iterator of pairs into two collections.
- `partition(p)`, which 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), which has extensive coverage of the possibilities.

This rich collection of iterator transformations is meant to be used – to produce code that is more idiomatic, more compact, and with clearer intent.

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:

```
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 run-time failure:

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err`
value: TryFromIntError(() )', iterators/src/main.rs:249:36
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

Following the advice of [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::()?;
```

Combining this with the question mark operator gives useful behaviour:

- 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 5 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;
}

```

The value of the item is never used directly, only 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 have resulting 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 multi-functional, 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 only help 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 over-optimistic 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 done 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`.

Item 11: Implement the Drop trait for RAI patterns

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

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

With an RAI type,

- the type's *constructor* acquires access to some resource, and
- the type's *destructor* releases access to that resource.

The result of this is that the RAI 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 exit¹.

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; 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 behaviour into an RAII class:

```
// C++ code
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 which allows a pointer to the memory to be "borrowed" for ephemeral use (`ptr.get()`).

In Rust, this behaviour for memory pointers is built into the language ([Item 15](#)), but the general principle of RAII is still useful for other kinds of resources². **Implement Drop for**

any types that hold resources that must be released, such as:

- 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 on to 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, ...) may need similar encapsulation.
- Access to raw memory, for `unsafe` types that deal with low-level memory management (e.g. for FFI).

The most obvious instance of RAI 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 above, but in Rust the `MutexGuard` item acts as a proxy to the mutex-protected data in addition to being an RAI 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 RAI items**. This leads to slightly odd indentation, but it's worth it for the added safety and lifetime precision.

```
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 RAI pattern, an explanation of how to implement it is in order. The `Drop` trait allows you to add user-defined behaviour 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);
    }
}
```

The `drop` method is specially reserved to the compiler and can't be manually invoked, because the item would be left in a potentially messed-up state afterwards:

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

```
error[E0040]: explicit use of destructor method
--> raii/src/main.rs:65:7
65 |     x.drop();
    |     ^^^^^
    |
    | explicit destructor calls not allowed
    | help: consider using `drop` function: `drop(x)`
```

(As suggested by the compiler, just call `drop(obj)` instead to manually drop an item, or enclose it in a scope as suggested above.)

The `drop` method is therefore the key place for implementing RAI patterns; its implementation is the ideal place to release associated resources.

1: This also means that RAI as a technique is mostly only available 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: RAI is also still useful for memory management in low-level `unsafe` code, but that is (mostly) beyond the scope of this book

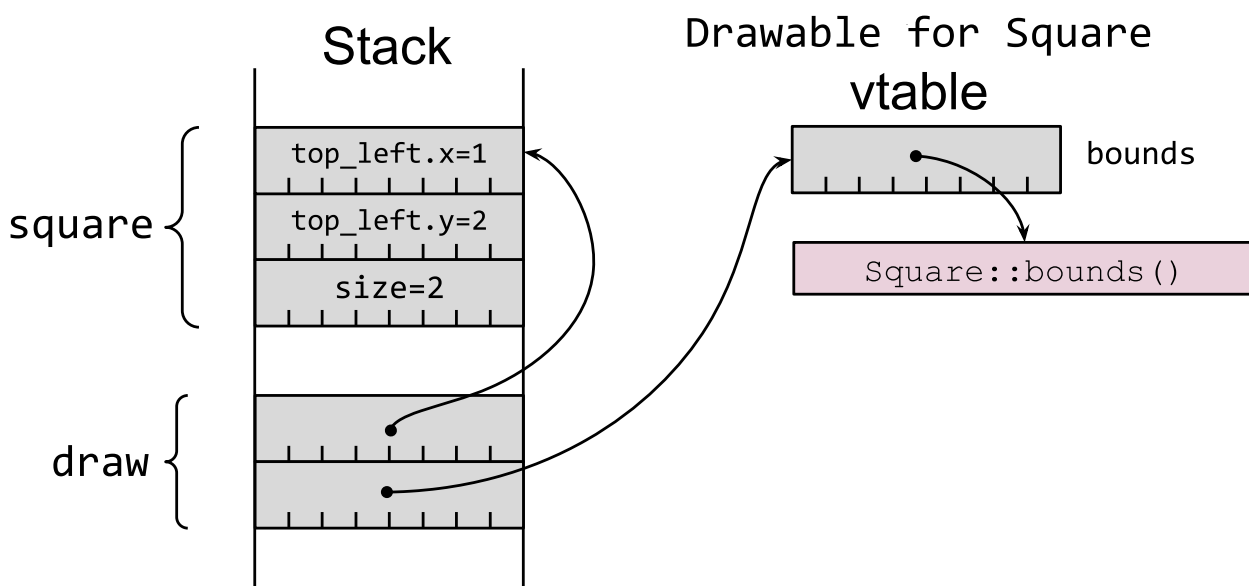
Item 12: Prefer generics to trait objects

Item 2 described the use of traits to encapsulate behaviour 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.

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.

In comparison, trait objects are fat pointers (Item 9) 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.

```
let square = Square::new(1, 2, 2);
let draw: &dyn Drawable = &square;
```



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 of the code `generic::<T>(t: &T)` for every type `T` that gets used; a `traitobj(t: &dyn T)` method only needs a single instance.
- Invoking a trait method from a generic will generally be ever-so-slightly faster than 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 only use optimization-related concerns as a primary decision driver 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 methods available, depending on whether the type parameter implements *multiple* traits.

```
trait Drawable {
    fn bounds(&self) -> Bounds;
}

struct Container<T>(T);

impl<T: Drawable> Container<T> {
    // The `area` method is available for all `Drawable`
containers.
    fn area(&self) -> i64 {
        let bounds = self.0.bounds();
        (bounds.bottom_right.x - bounds.top_left.x)
            * (bounds.bottom_right.y - bounds.top_left.y)
    }
}

impl<T: Drawable + Debug> Container<T> {
    // The `show` method is only available if `Debug` is also
implemented.
    fn show(&self) {
        println!("{:?} has bounds {:?}", self.0,
self.0.bounds());
    }
}

let square = Container(Square::new(1, 2, 2)); // Square is not
Debug
let circle = Container(Circle::new(3, 4, 1)); // Circle is
Debug

println!("area(square) = {}", square.area());
println!("area(circle) = {}", circle.area());
circle.show();
// The following line would not compile.
// square.show();
```

A trait object only encodes the implementation vtable for a single trait, so doing

something equivalent is much more awkward. For example, a combination `DebugDrawable` trait could be defined for the `show()` case, together with some conversion operations ([Item 6](#)) to make life easier. However, if there are multiple different combinations of distinct traits, it's clear that the combinatorics of this approach rapidly become unwieldy.

[Item 2](#) described the use of trait bounds to restrict what type parameters are acceptable for a generic function. Trait bounds can also be applied to trait definitions themselves:

```
trait Shape: Drawable {  
    fn render_in(&self, bounds: Bounds);  
    fn render(&self) {  
        self.render_in(overlap(SCREEN_BOUNDS, self.bounds()));  
    }  
}
```

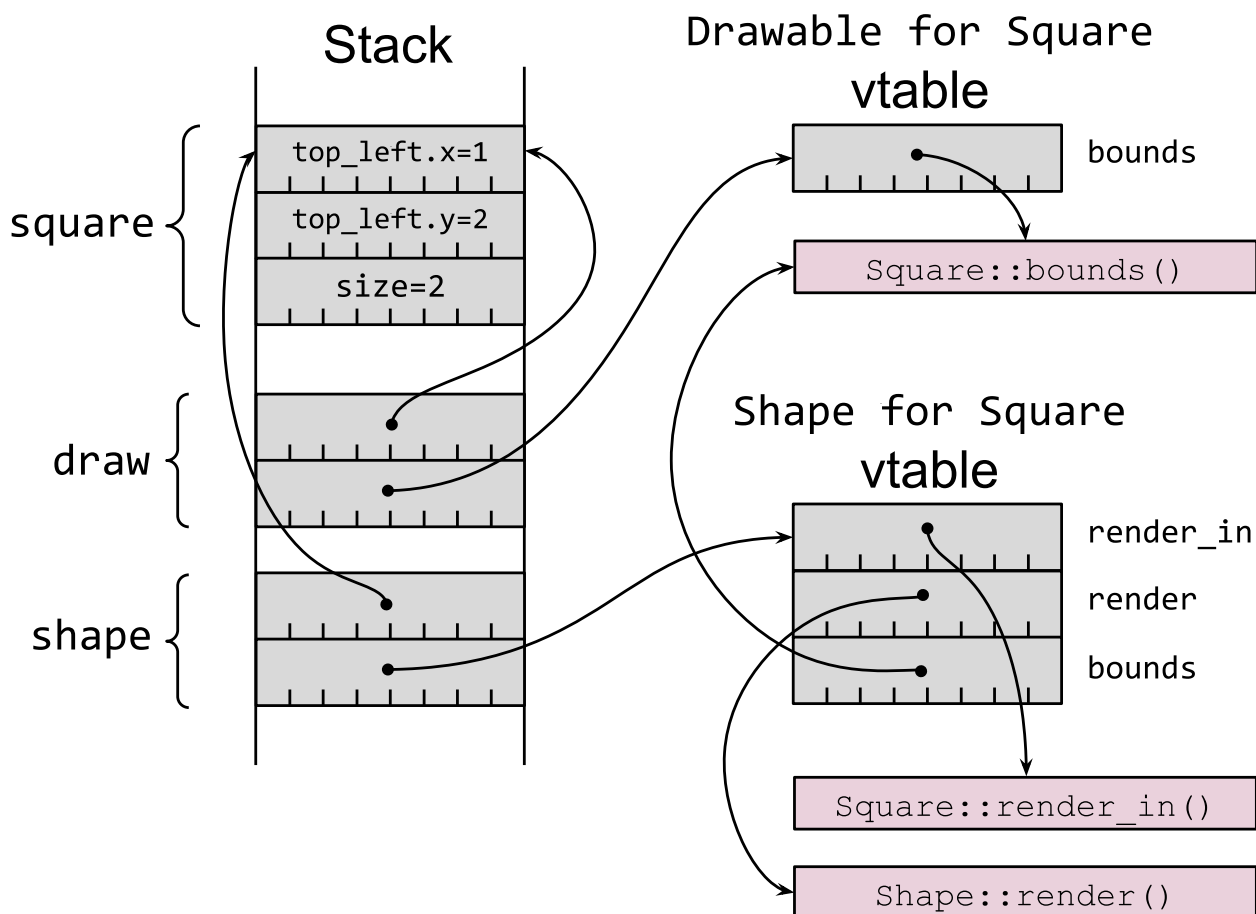
In this example, the `render()` method's default implementation ([Item 13](#)) makes use of the trait bound, relying on the availability of the `bounds()` method from `Drawable`.

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** `Drawable`. That's not the case: the relationship between the two types is better expressed as `Shape` **also-implements** `Drawable`.

Under the covers, trait objects for traits that have trait bounds

```
let square = Square::new(1, 2, 2);  
let draw: &dyn Drawable = &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 means that there is no way to "upcast" from `Shape` to `Drawable`, because the (pure) `Drawable` vtable can't be recovered at runtime (see [Item 19](#) for more on this). There is no way to convert between related trait objects, which in turn means there is no [Liskov substitution](#).

Repeating the same point in different words, a method that accepts a `Shape` trait object

- *can* make use of methods from `Drawable` (because `Shape` also-implements `Drawable`, and because the relevant function pointers are present in the `Shape` vtable)
- *cannot* pass the trait object on to another method that expects a `Drawable` trait object (because `Shape` is-not `Drawable`, and because the `Drawable` vtable isn't available).

In contrast, a generic method that accepts items that implement `Shape`

- *can* use methods from `Drawable`
- *can* pass the item on to another generic method that has a `Drawable` trait bound, because the trait bound is monomorphized at compile time to use the `Drawable` methods of the concrete type.

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 return a type that includes `Self`.

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>`, ... The trait object's vtable, on the other, is very much a finite collection of function pointers, and so it's not possible to fit an infinite quart into a finite pint pot.

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 5](#)) immediately fall under this rule. 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 mention¹ `Self` lead to a trait that is not object safe.

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: `Self: Sized`. This trait bound means that the method can't be used with trait objects anyway, because trait objects are explicitly of unknown size (`!Sized`), and so the method is irrelevant for object safety.

The balance of factors so far leads to the advice to **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 at the start of this Item).

A more theoretical aspect that leads towards 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 differently (concretely) typed items.

The traditional OO example of rendering a list of shapes would be 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 run-time (e.g via

`dlopen(3)`), then items that implement traits in the new code can only be invoked via a trait object, because there's no source code to monomorphize over.

1: At present, 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 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`; it 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 more optimal 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 only has to implement 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 which includes a panoply of pre-provided methods ([Item 10](#)), 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: 'a>(self) -> Cloned<Self>
where
    Self: Sized + Iterator<Item = &'a T>,
    T: Clone,
{
    Cloned::new(self)
}
```

In other words, the `cloned()` method is only available 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 be safely added to a trait even after an initial version of the trait is released. An addition like this preserves backwards compatibility (see [Item 21](#)) for both users and implementors¹ of the trait.

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: This is true even if an implementor was unlucky enough to have already added a method of the same name in the concrete type, as the concrete method – known as an *inherent implementation* – will be used ahead of trait method. The trait method can be explicitly selected instead by casting: `<Concrete as Trait>::method()`.

Concepts

The first section of this book covered Rust's type system, which helps provide the vocabulary needed to work with some of the *concepts* involved in writing Rust code.

The borrow checker and lifetime checks are central to what makes Rust unique; they are also a common stumbling block for newcomers to Rust.

However, 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 behaviour of C/C++ in Rust, but why bother to use Rust if you do?

Item 14: Understand lifetimes

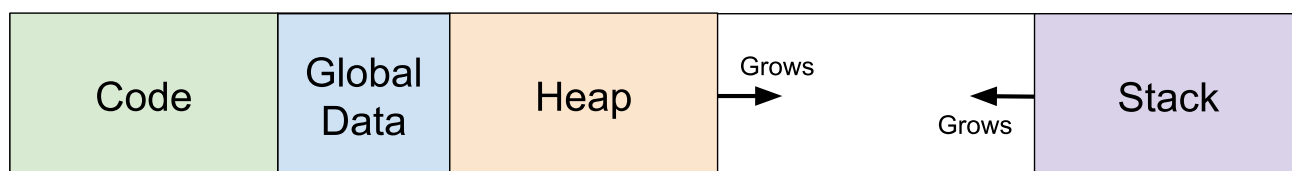
"Eppur si muove" – Galileo Galilei

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 downwards and the other can grow upwards (at least until your program runs out of memory and crashes).



Of these two dynamically sized chunks, the stack is used to hold state related to the currently executing function, specifically its parameters, local variables and temporary values, held in a *stack frame*. 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.

When the caller invokes a different function `g()`, the process happens again, which means that the stack frame for `g()` will re-use the same area of memory that `f()` previously used.

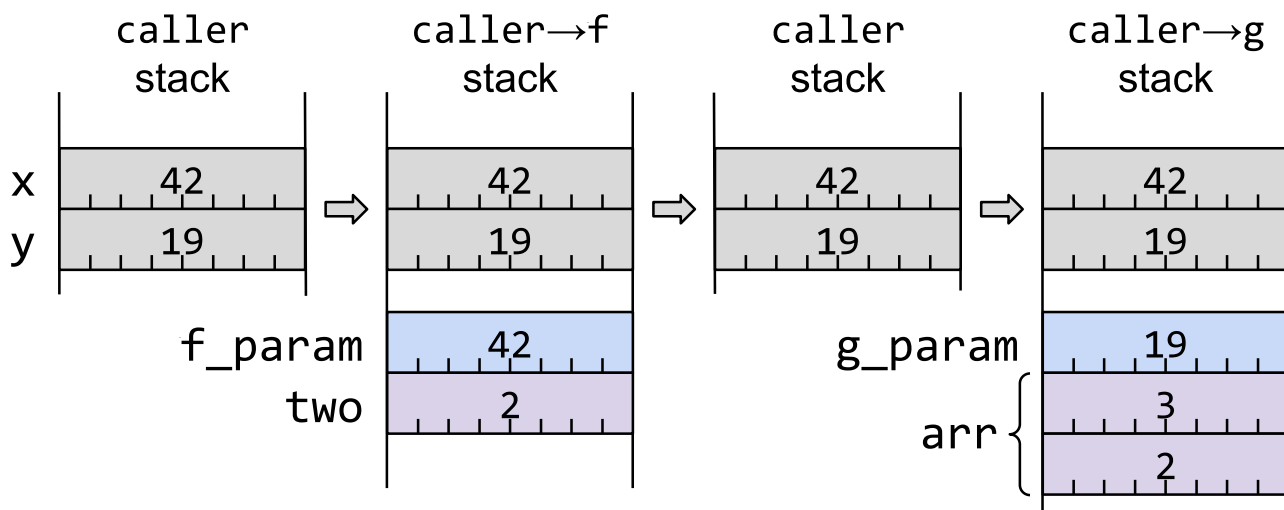

```

fn caller() -> u64 {
    let x = 42u64;
    let y = 19u64;
    f(x) + g(y)
}

fn f(f_param: u64) -> u64 {
    let two = 2;
    f_param + two
}

fn g(g_param: u64) -> u64 {
    let arr = [2, 3];
    g_param + arr[1]
}

```



Of course, this is a dramatically simplified version of what really goes on; putting things on and off the stack takes time and so there are many optimizations for real processors. 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, but only ephemeraly. 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* 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 0x7ff7b5ca9408 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 re-used 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 0x7ff7b5ca9408 has fd=1842872565
```

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 forwards in time to C++, this problem of losing access to resources was solved by the inclusion of *destructors*, enabling RAII (cf. [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() {
    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 0x7ff7b57ef408 has fd=-1
```

However, C++ did nothing to help with the problem of dangling pointers: it's still possible to hold on to a pointer to an object that's gone (and whose destructor 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 0x7ff7b57ef408 has fd=1711145032
```

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 fraction¹ of security problems.

Doing so requires that the concept of lifetimes move from the background (where C/C++ programmers have to just 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 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:

```

    {
        let item1 = Item { contents: 1 }; // `item1` created
here
        let item2 = Item { contents: 2 }; // `item2` created
here
        println!("item1 = {:?}, item2 = {:?}", item1, 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 over-simplified but helpful way to think about this is to imagine that each branch of the expression's syntax tree 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 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 beyond the lifetime's scope:

```
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
  --> lifetimes/src/main.rs:206:17
   |
206 |         r = &item;
   |             ^^^^^ borrowed value does not live long
   |             enough
207 |     }
   |     - `item` dropped here while still borrowed
208 |     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
  --> lifetimes/src/main.rs:236:46
    |
236 |         let r: &Item = fn_returning_ref(&mut Item { contents:
42  | });
    |
~~~~~ - temporary value is freed at the end of this
statement
    |
    |                                     |
    |                                     | creates a
    | temporary which is freed while still in use
237 |         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 the end of the enclosing scope. This feature (known as [non-lexical lifetimes](#)) allows the borrow checker to be a little bit more generous:

```

{
    let mut s: String = "Hello, world".to_string(); // `s` owns
the `String`

    let greeting = &mut s[..5]; // mutable reference to
`String`
    greeting.make_ascii_uppercase();
    // .. no use of `greeting` after this point

    let r: &str = &s; // immutable reference to `String`
    println!("s = '{}'", r); // s = 'HELLO, world'
} // where the mutable reference `greeting` would naively be
dropped

```

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`". (There's one partial exception to this, covered below: `'static`.)

Instead, your code refers to lifetimes with arbitrary labels, conventionally `'a`, `'b`, `'c`, ...,

and the compiler has its own internal, inaccessible representation of what that equates to in the source code.

You don't get to do much with these lifetime labels; the main thing that's possible is to compare one label with another, repeating a label to indicate that two lifetimes are the "same". (Later, we'll see that it's also possible to specify that one lifetime must be bigger than another, when expressed as the lifetime bounds for a generic.)

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 to choose from: the lifetime of the input, which means that they both share the same label, say 'a :

```
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 lifetimes for this case. A more idiomatic version of the same function signature would be:

```
pub fn first(data: &[Item]) -> Option<&Item> {
```

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
  --> lifetimes/src/main.rs:399:59
    |
399 |     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
399 |     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, a function where this combination of lifetimes might make sense is:

```

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 above) 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 out-live 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 {:?} and {:?} is {:?}", inner,
outer, 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` use within 'a',
outside of 'b'
} // end of lifetime 'a'

```

Lifetime Elision Rules

In addition to the "one in, one out" elision rule described above, there are two other elision rules that mean that lifetimes 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.

Summarizing 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
```

The 'static Lifetime

The previous section described various different 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
  --> lifetimes/src/main.rs:411:28
   |
411 |     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
411 |     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 a placeholder label.

```
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.

Note that a `const` global variable does *not* have the same guarantee: only the *value* is guaranteed to be the same everywhere, and the compiler is allowed to make as many copies as it likes, wherever the variable is used. These potential copies may be ephemeral, and so won't satisfy the `'static` requirements:

```
const ANSWER: Item = Item { contents: 42 };
pub fn the_answer() -> &'static Item {
    &ANSWER
}
```

```
error[E0515]: cannot return reference to temporary value
```

```
--> lifetimes/src/main.rs:424:9
```

```
424 |         &ANSWER
    |         ^-----
    |         ||
    |         |temporary value created here
    |         |returns a reference to data owned by the current
function
```

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
  --> lifetimes/src/main.rs:318:32
   |
318 |         let r: &'static Item = &boxed;
   |                                ^^^^^^^ borrowed value does not
live long enough
   |                                |
   |                                type annotation requires that `boxed` is
borrowed for `'_static`
319 |         println!("'static item is {:?}", r);
320 |     }
   |     - `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, because it was already moved

```

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. Recovering the memory requires `unsafe` code, which makes this a technique to reserve for special circumstances.

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
} // `b` 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
  --> lifetimes/src/main.rs:452:19
   |
452 |         pub item: &Item,
   |                   ^ expected named lifetime parameter
help: consider introducing a named lifetime parameter
   |
450 ~     pub struct ReferenceHolder<'a> {
451 |         pub index: usize,
452 ~         pub item: &'a Item,
   |

```

As usual, the compiler error message tells what to do. The first part is simple enough: give the reference type an explicit lifetime `'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 annotation `<'a>` that matches the lifetime of the reference contained within it:

```

pub struct ReferenceHolder<'a> {
    pub index: usize,
    pub item: &'a Item,
}

```

The lifetime annotation for the data structure is infectious: any containing data structure that uses the type also has to acquire a lifetime annotation:

```

// Annotation includes lifetimes of all fields
pub struct MultiRefHolder<'a, 'b> {
    pub left: ReferenceHolder<'a>,
    pub right: ReferenceHolder<'b>, // Could choose 'a instead
here
}

```

This makes sense: anything that contains a reference, no matter how deeply nested, is only valid 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 does also mean 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 9](#) can help untangle the lifetime constraints.

Lifetime Bounds

Generic code ([Item 12](#)) involves some unknown type `T`, often constrained by a trait bound `T: SomeTrait`. But what happens if the code that's built around `T` is a reference type?

There's various different ways that references can make their way into the generics mix:

- A generic might take a `<T>`, but include code that deals with references-to- `T`, for example `&T`.
- The type `T` might itself be a reference type, for example `&Thing`, or some bigger data structure `MultiRefHolder<'a, 'b>` that includes references.

Regardless of the route by which references arise, any generic data structure needs to propagate their associated lifetimes, as in the previous section.

The main way to allow for this is to specify [lifetime bounds](#), which indicate that either a type (`T: 'b`) or specific lifetime `'a: 'b` has to outlive some other lifetime `'b`.

For example, consider a type that wraps a reference (somewhat akin to the `Ref` type returned by `RefCell::borrow()`):

```
pub struct Ref<'a, T: 'a>(&'a T);
```

This generic data structure holds an explicit reference `&'a T`, as per the first bullet above. But the type `T` might itself contain references with some lifetime `'b`, as per the second bullet above. If `T`'s inherent lifetime `'b` were smaller than the exterior lifetime `'a` we'd have a potential disaster: the `Ref` would be holding a reference to a data structure whose own references have gone bad.

To prevent this, we need `'b` to be larger than `'a`; for named lifetimes this would be written as `'b: 'a`, but we need to say that slightly differently, as `T: 'a`. Roughly translated into words, that says "any references in the type `T` must have a lifetime that outlives `'a`", and that makes `Ref` safe: if its own lifetime (`'a`) is still valid, then so are any references hidden inside `T`.

Translating lifetime bounds into words also helps with a 'static lifetime bound like `T: 'static`. This says that "any references in the type `T` must have a 'static lifetime", which means that `T` can't have any dynamic references. Any non-reference type `T` that owns its contents – `String`, `Vec` etc. – satisfies this bound, but any type that has an `<'a>` creeping in does not.

One common place this shows up is when you try to move values between threads with [std::thread::spawn](#). The moved values need to be of types that implement `Send` (see [Item 17](#)), indicating that they're safe to move between threads, but they also need to not contain any dynamic references (the 'static lifetime bound). This makes sense when you

realize that a reference to something on the stack now raises the question: which stack? Each thread's stack is independent, and so lifetimes can't be tracked between them.

1: For example, the Chromium project estimates that [70% of security bugs are due to memory safety](#).

Item 15: Understand the borrow checker

"The power to destroy a thing is the absolute control over it." – Frank Herbert

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*.

Under the covers this uses the same kind of *pointer* values ([Item 9](#)) that are so prevalent in C or C++ code, but 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 value¹.
- 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 expressions² like `f(&mut value)` have the possibility of changing the contents of `value` .

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 have to give it back*. In particular, you have to give it back *before* the lifetime of the underlying item expires, as tracked by the compiler and explored in [Item 14](#).

These restrictions on the use of references are at the heart of Rust's memory safety guarantees, but they do mean you have to accept the cognitive costs of the borrow rules – accept that it will change how you design your software, particularly its data structures.

Access Control

There are three different ways that a Rust item can be accessed: via the item's *owner* (`item`), via a *reference* (`&item`), or via a *mutable reference* (`&mut item`).

Each of these different ways of accessing the item comes with different powers over the item. Putting things in [CRUDE](#) terms:

- The owner of an item gets to **c**reate it, **r**ead from it, **u**ppdate it, and **d**rop it (CRUD).
- A mutable reference can be used to **r**ead from the underlying item, and to **u**ppdate it (`_RU_`).
- A (normal) reference can only be used to **r**ead from the underlying item (`_R_`).

There's an important Rust-specific extension 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 **c**reating (in the new location) and **d**ropping 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` :

```
fn overwrite(item: &mut Option<Item>, val: Item) {
    *item = Some(val);
}
```

but a modification to return the previous value falls foul of the move restriction:

```
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
```

```
--> borrows/src/main.rs:27:24
```

```
27 |         let previous = *item; // move out
    |                        ^^^^^ move occurs because `*item` has
type `Option<Item>`, which does not implement the `Copy` trait
```

```
help: consider borrowing the `Option`'s content
```

```
27 |         let previous = *item.as_ref(); // move out
    |                        ++++++
```

```
help: consider borrowing here
```

```
27 |         let previous = &*item; // move out
    |                        ~~~~~
```

It's valid to read from a mutable reference, and it's valid to write to a mutable reference, and so the ability to do both at once is provided by the `std::mem::replace` function in the standard library. This uses `unsafe` code under the covers (as per [Item 16](#)) to perform the swap in one go:

```

    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:

```

    pub fn replace(item: &mut Option<Item>, val: Item) ->
Option<Item> {
    item.replace(val)
}

```

Borrow Rules

The first rule for borrowing references in Rust is that the scope of any reference must be smaller than the lifetime of the item that it refers to. However, the compiler is smarter than just assuming that a reference lasts until it is dropped – 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 (as described in [Item 14](#)).

The second rule for borrowing references is that, in addition to the owner of an item, there can be

- any number of immutable references to the item, *or*
- a single mutable reference to the item

but not both.

So a method that takes multiple immutable references can be fed references to the same item:

```

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:

```
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

```
--> borrows/src/main.rs:115:26
|
115 |     zero_both(&mut item, &mut item);
|           ----- ^^^^^^^^^ second mutable borrow
occurs here
|           |           |
|           |           first mutable borrow occurs here
|           first borrow later used by call
```

and similarly for a mixture of the two:

```
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

```
--> borrows/src/main.rs:140:30
|
140 |     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 is:

- better optimized: values can be (e.g.) cached in registers, secure in the knowledge that the underlying memory contents will not change in the meanwhile
- 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. To help understand this, consider operations involving the owner as though they make use of references along the way.

For example, an attempt to update the item via its owner while a reference exists fails, because of this transient second mutable reference:

```
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
```

```
--> borrows/src/main.rs:164:5
```

```
163 |     let r = &item;
    |           ----- borrow of `item.contents` occurs here
164 |     item.contents = 0;
    |     ^^^^^^^^^^^^^^^^^ assignment to borrowed `item.contents`
occurs here
...
167 |     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 is `Copy`
r.contents = 0;
```

error[E0503]: cannot use `item.contents` because it was mutably borrowed

--> borrows/src/main.rs:194:20

```
193 |     let r = &mut item;
    |           ----- borrow of `item` occurs here
194 |     let contents = item.contents; // i64 is `Copy`
    |                   ^^^^^^^^^^^^^ use of borrowed `item`
195 |     r.contents = 0;
    |     ----- borrow later used here
```

Finally, the existence of any sort of reference prevents the owner of the item from moving or dropping the item, exactly because this 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

--> borrows/src/main.rs:151:20

```
150 |     let r = &item;
    |           ----- borrow of `item` occurs here
151 |     let new_item = item; // move
    |                   ^^^^ move out of `item` occurs here
152 |     println!("reference to item is {:?}", r);
    |                                           - borrow later used here
```


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>(a: &'a str, b: &'b str) -> Option<&'a str> {
    a.find(b).map(|i| &a[i..i + b.len()])
}
```



```
// ...
```

```
let found = find(&format!("{}", to search", "Text"), "ex");
if let Some(text) = found {
    println!("Found '{}'", text);
}
```

```
error[E0716]: temporary value dropped while borrowed
```

```
--> borrows/src/main.rs:312:23
```

```
312 |         let found = find(&format!("{}", to search", "Text"), "ex");
```

```
temporary value is freed at the end of this statement
```

```
while still in use
313 |         if let Some(text) = found {
            ----- borrow later used here
```

```
= note: consider using a `let` binding to create a longer lived
value
```

```
= note: this error originates in the macro `format` (in Nightly
builds, run with -Z macro-backtrace for more info)
```

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 rules above.

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 out-lives 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 only extends to the end of the expression) to be 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

--> borrows/src/main.rs:257:35

```
257 |         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`
--> borrows/src/main.rs:269:40
269 |         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 re-coalesce the local variables back together, so that you can pretend that 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 if your data structures can own 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 inter-connected than a tree pattern (a `Root` that owns multiple `Branch`s, 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.

```
#[derive(Clone)]
struct Guest {
    name: String,
    phone: PhoneNumber,
    address: String,
    // ... many other fields
}

#[derive(Default, Debug)]
struct GuestRegister(Vec<Guest>);

impl GuestRegister {
    fn register(&mut self, guest: Guest) {
        self.0.push(guest)
    }
    fn nth(&self, idx: usize) -> Option<&Guest> {
        if idx < self.0.len() {
            Some(&self.0[idx])
        } else {
            None
        }
    }
}
```

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 taking a copy can give a quick solution.

```
#[derive(Default, Debug)]
struct ClonedGuestRegister {
    by_arrival: Vec<Guest>,
    by_name: BTreeMap<String, Guest>,
}

impl ClonedGuestRegister {
    fn register(&mut self, guest: Guest) {
        self.by_arrival.push(guest.clone()); // requires `Guest` to
        be `Clone`
        self.by_name.insert(guest.name.clone(), guest);
    }
    fn named(&self, name: &str) -> Option<&Guest> {
        self.by_name.get(name)
    }
    fn nth(&self, idx: usize) -> Option<&Guest> {
        // snip
    }
}
```

This approach of taking copies copes poorly if the data can be modified – if the telephone number 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.

This approach copes fine with a changing phone number – the (single) `Guest` is owned by the `Vec`, and will always be reached that way under the covers:

```
let new_number = PhoneNumber::new(123456);
ledger.named_mut("Bob").unwrap().phone = new_number;
assert_eq!(ledger.named("Bob").unwrap().phone, new_number);
```

However, it copes less well with a different kind of modification: what happens if guests can de-register:

```
// Deregister the `Guest` at position `idx`, moving up all
subsequent guests.
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, 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 re-introduced a world where those "pointers" can point to nothing (beyond the `Vec` bounds) or can 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);
```



```
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({n: 'Bob', ...})
Bob is Some({n: 'Charlie', ...})
Charlie is None
```

Regardless of approach, the code needs to be fixed to ensure the data structures stay in sync. However, a better approach to the underlying data structure would be to use Rust's smart pointers instead ([Item 9](#)). Shifting to a combination of [Rc](#) and [RefCell](#) avoids the invalidation problems of using indices as pseudo-pointers:

```

#[derive(Default)]
struct RcGuestRegister {
    by_arrival: Vec<Rc<RefCell<Guest>>>,
    by_name: BTreeMap<String, Rc<RefCell<Guest>>>,
}

impl RcGuestRegister {
    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);
    }
    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(())
    }
    // snip
}

```

```

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: {n: 'Alice', ...} })
Bob is Some(RefCell { value: {n: 'Bob', ...} })
Charlie is Some(RefCell { value: {n: 'Charlie', ...} })

```

The output is now valid, but there's a lingering entry for Alice that remains until we ensure that the two collections stay in sync:

```

fn deregister_fixed(&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: {n: 'Bob', ...} })
Charlie is Some(RefCell { value: {n: 'Charlie', ...} })

```

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 9](#) 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. Often combined with...
- `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 run-time.
- `Arc` is the multi-threading equivalent to `Rc`.
- `Mutex` (and `RwLock`) allows interior mutability in a multi-threading environment, roughly equivalent to `RefCell`.
- `Cell` allows interior mutability for `Copy` types.

For programmers and designs that 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 naïve 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 run-time `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:

```
// C++ code (with lackadaisical pointer use)
struct Tree {
    std::string id() const;

    std::string tree_id_;
    std::vector<Branch*> branches_; // `Tree` owns `Branch` objects
};

struct Branch {
    std::string id() const; // hierarchical identifier for `Branch`

    std::string branch_id_;
    std::vector<Leaf*> leaves_; // `Branch` owns `Leaf` objects
    Tree* owner_; // back-reference to owning `Tree`
};

struct Leaf {
    std::string id() const; // hierarchical identifier for `Leaf`

    std::string leaf_id_;
    Branch* owner_; // back-reference to owning `Branch`
};

std::string Branch::id() const {
    if (owner_ == nullptr) {
        return "<unowned>." + branch_id_;
    } else {
        return owner_->id() + "." + branch_id_;
    }
}
```

Implementing the equivalent pattern in Rust can make use of `Rc<T>`'s more tentative partner, `Weak<T>`:

```

struct Tree {
    tree_id: String,
    branches: Vec<Rc<RefCell<Branch>>>,
}

struct Branch {
    branch_id: String,
    leaves: Vec<Rc<RefCell<Leaf>>>,
    owner: Option<Weak<RefCell<Tree>>>,
}

struct Leaf {
    leaf_id: String,
    owner: Option<Weak<RefCell<Branch>>>,
}

```

The `Weak` reference doesn't increment the 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 id(&self) -> String {
        match &self.owner {
            None => format!("<unowned>.{})", self.branch_id),
            Some(t) => {
                let tree = t.upgrade().expect("internal error:
owner gone!");
                format!("{}.{}", tree.borrow().id(),
self.branch_id)
            }
        }
    }
}

```

If Rust's smart pointer 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 already have implemented the semantics you want, inside a safe interface, and if you search the standard library and `crates.io` you might find it.

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 of the pointer traits described in [Item 9](#). But you don't have to: the standard library already includes the `std::borrow::Cow`

type³ that covers exactly this scenario once you know it exists.

Self-Referential Data Structures

One particular style of data structure 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 9](#): the reference needs a lifetime annotation, but we wouldn't want that lifetime annotation to be propagated to the containing data structure, because the intent is not to refer to anything external.

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<Range<usize>>,  
}
```

However, this indexing approach only works for simple examples. A more general version of the self-reference problem turns up when the compiler deals with `async` code⁴. Roughly speaking, the compiler bundles up a pending chunk of `async` code into a lambda, and the data for that lambda 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 (as its [official docs](#) make clear):

- The internal reference fields need to use raw pointers, or near relatives (e.g. `NonNull`) thereof.
- The type being pinned needs to *not* implement the `Unpin` marker trait. This trait is automatically implemented for almost every type, so this typically involves adding a (zero-sized) field of type `PhantomPinned` to the struct definition⁵.
- The item is only pinned once it's on the heap and held via `Pin`; in other words, only the contents of something like `Pin<Box<MyType>>` is pinned. This means that the internal reference fields can only be safely filled in after this point, but as they are raw pointers the compiler will give you no warning if you incorrectly set them *before* calling `Box::pin`.

Where possible, **avoid self-referential data structures** or try to find library crates that encapsulate the difficulties for you (e.g. [ouroborous](#)).

1: However, it may be ownership of a *copy* of the item, if the `value`'s type is `Copy`; see [Item 5](#).

2: Note that all bets are off with expressions like `m!(value)` that involve a macro ([Item 28](#)), because it can expand to arbitrary code.

3: `Cow` stands for copy-on-write; a copy of the underlying data is only made 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.

5: In [future](#) it may be possible to just declare `impl !Unpin for MyType {}`

Item 16: Avoid writing unsafe code

The memory safety guarantees 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 re-organize your code to mollify the borrow checker ([Item 15](#)), and to precisely specify the pointer types that you use ([Item 9](#)).

Unsafe Rust weakens some of those guarantees, in particular by allowing 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 used.

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 1000 uses of `unsafe` in the `alloc` library, 1500 in `core` and a further 2000 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 9](#) use `unsafe` code (often raw pointers) internally in order 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.

The standard library¹ also has other functionality covering more advanced features, implemented with `unsafe` internally:

- `std::pin::Pin` forces an item to not move in memory ([Item 14](#)). 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 only happens 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. For example:

- [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.

There are many other examples, but hopefully the general idea is clear. If you want to do something that doesn't obviously fit with the constraints of Rust (especially [Item 15](#) and [Item 14](#)) hunt through the standard library to see if there's existing functionality that does what you need. If you don't find it, try also hunting through crates.io ; after all, most of the time your problem will not be a unique one 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; see [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 of 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".

- Write even more tests ([Item 30](#)) than usual.
- Run additional diagnostic tools ([Item 31](#)) over the code. In particular, **run Miri over your `unsafe` code** – [Miri](#) interprets the intermediate level output from the compiler, which allows it to detect classes of errors that are invisible to the Rust compiler.
- Think about multi-threaded use, particularly if there's shared state ([Item 17](#)).

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.

This Item is specific to *shared-state* parallelism: where different threads of execution communicate with each other by sharing memory.

Data Races

Let's start with the good news, by exploring *data races* and Rust. The definition is roughly:

A data race is defined to occur when two distinct threads access the same memory location, where

- at least one of them is a write, and
 - there is no synchronization mechanism that enforces an ordering on the accesses.
-

The basics of this are best illustrated with an example:

```
// C++ code.
class BankAccount {
public:
    BankAccount() : balance_(0) {}

    int64_t balance() const {
        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 which will become clear shortly. However, the same general concepts apply in lots of other languages (that aren't Rust) – Java, or Go, or Python ...

This class works fine in a single-threaded setting, but in a multi-threaded setting:

```
std::thread taker(take_out, &account, 100);
std::thread taker2(take_out, &account, 100);
std::thread taker3(take_out, &account, 100);
std::thread payer(pay_in, &account, 300);
```

where several threads are repeatedly trying to withdraw from the account:

```
int64_t check_balance(const BankAccount* account) {
    int64_t bal = account->balance();
    if (bal < 0) {
        std::cerr << "** Oh no, gone overdrawn: " << bal << " **!\n";
        abort();
    }
    std::cout << "Balance now " << bal << "\n";
    return bal;
}
```

```
void take_out(BankAccount* account, int count) {
    for (int ii = 0; ii < count; ii++) {
        if (account->withdraw(100)) {
            log("Withdrew 100");
        } else {
            log("Failed to withdraw 100");
        }
        check_balance(account);
        std::this_thread::sleep_for(std::chrono::milliseconds(6));
    }
}
```

then 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 `sleep` call also artificially raises 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 class ensure that access to it is governed by some kind of external synchronization mechanism.

The class can be converted to a *thread-safe* class¹ – meaning that it is safe to use from multiple threads – by adding internal synchronization operations:

```
// C++ code.
class BankAccount {
public:
    BankAccount() : balance_(0) {}

    int64_t balance() const {
        const std::lock_guard<std::mutex> with_lock(mu_);
        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;
        }
        // What if another thread changes `balance_` at this point?
        balance_ -= amount;
        return true;
    }

private:
    mutable std::mutex mu_;
    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 `lock()` at a time. The second and subsequent callers will block until `unlock()` is called – and even then, only *one* of the blocked threads will unblock and proceed through `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 (cf. [Item 11](#)) that 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:


```

bool transfer(BankAccount* destination, uint32_t amount) {
    // oops, forgot about mu_
    if (balance_ < amount) {
        return false;
    }
    balance_ -= amount;
    destination->balance_ += amount;
    return true;
}

```

and the thread-safety has been destroyed².

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 {
        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
    }
}

```

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,
30));
// Wait for thread completion
payer.join().unwrap();


```

but a naïve attempt to use the `BankAccount` across multiple threads:

```

        let mut account = BankAccount::new();
        let taker = std::thread::spawn(move || take_out(&mut
account, 100));
        let payer = std::thread::spawn(move || pay_in(&mut account,
300));
        payer.join().unwrap();
        taker.join().unwrap();

```



immediately falls foul of the compiler:

```

error[E0382]: use of moved value: `account`
  --> deadlock/src/main.rs:76:40
   |
74 |         let mut account = BankAccount::new();
   |         ----- move occurs because `account` has type
`broken::BankAccount`, which does not implement the `Copy` trait
75 |         let taker = std::thread::spawn(move || take_out(&mut
account, 100));
   |         -----
   |         variable moved due to use in closure
   |         |
   |         |         value moved into
closure here
76 |         let payer = std::thread::spawn(move || pay_in(&mut
account, 300));
   |         ^^^^^^^
   |         use occurs due to use in closure
   |         |
   |         |         value used here after
move

```

With experience of the borrow checker ([Item 15](#)), the problem sticks out clearly: 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 an RAII behaviour, like C++'s `std::lock_guard`: the mutex is automatically released at the end of the scope when the guard is dropped. (Rust's `Mutex` has no manual lock/unlock methods, as they would expose developers to the danger of forgetting to keep `lock()` and `unlock()` calls exactly in sync.)

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 9](#)), allowing it to be used for both read operations:

```
pub fn balance(&self) -> i64 {
    *self.balance.lock().unwrap()
}
```

and write operations:

```
// Note: no longer needs `&mut self`.
pub fn deposit(&self, amount: i64) {
    *self.balance.lock().unwrap() += amount
}
```

There's an interesting detail lurking in the signature of this method: although it is modifying the balance of the `BankAccount`, this method now takes `&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 9](#): borrow checks are effectively moved from compile-time to run-time, but now with a specific concern for cross-thread synchronization.

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,
100));
    let payer = std::thread::spawn(|| pay_in(&account,
300));
}
```



++++

1/24/24, 13:21

at the end of the block, but there are two new threads which have a reference to it, and which will carry on running afterwards.

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 multi-threaded use is `std::sync::Arc` :

```
let account = std::sync::Arc::new(BankAccount::new());

let account2 = account.clone();
let taker = std::thread::spawn(move || take_out(&account2,
100));

let account3 = account.clone();
let payer = std::thread::spawn(move || pay_in(&account3,
300));
```

Each thread gets its own (moved) copy of the reference counting pointer, and the underlying `BankAccount` will only be dropped 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 multi-threaded 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 5](#)) that have no associated methods, but they have special significance to the compiler in multi-threaded 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 are also `Send` / `Sync` .

The majority of safe types are `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 non- `Send` type is one that can only be used in a single thread. The canonical example of this is the unsynchronized reference counting pointer `Rc<T>` (Item 9). The implementation of this type explicitly assumes single-threaded use (for speed); there is no attempt at synchronizing the internal refcount for multi-threaded use. As such, transferring an `Rc<T>` between threads is not allowed; use `Arc<T>` (with its additional synchronization overhead) for this case.

A non- `Sync` type 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 multi-threaded environment.

(Raw pointer types like `*const T` and `*mut T` are also neither `Send` nor `Sync`; see Item 16 and Item 34.)

Deadlocks

Now for the bad news. Multi-threaded code comes with *two* terrible problems:

- **data races**, which can lead to corrupted data, and
- **deadlocks**, which can lead to your program grinding to a halt.

Both of these problems are terrible because they can be very hard to debug in practice: the failures occur non-deterministically 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 has solved the problem of data races (as described above), but the problem of deadlocks applies to Rust as much as it does to any other language that supports multi-threading.

Consider a simplified multiple player game server, implemented as a multithreaded application in order to service many players in parallel. Two core data structures might be a collection of players, indexed by username:

```
players: Mutex<HashMap<String, Player>>,
```

and a collection of games in progress indexed by some unique identifier:

```
games: Mutex<HashMap<GameID, Game>>,
```

Both of these data structures are `Mutex`-protected and so safe from data races; however, code that manipulates *both* data structures opens up potential problems. A single interaction between the two might work fine:

```
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);
        }
    }
    None
}
```

However, a second interaction between the two independently locked data structures is where problems start:

```
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:

- Thread 1 enters `add_and_join()` and immediately acquires the `players` lock.
- Thread 2 enters `ban_player()` and immediately acquires the `games` lock.
- Thread 1 now tries to acquire the `games` lock; this is held by thread 2, so thread 1 blocks.
- Thread 2 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, and 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 the simplest example of a more general description of the problem:

- Build a [directed graph](#) where:
 - Each `Mutex` instance is a vertex.
 - Each edge indicates a situation where one `Mutex` gets acquired while another `Mutex` is already held.
- If there are any cycles in the graph, a deadlock can occur.

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:

```
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);
            }
        }
    }
    None
}

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.

- Thread 1 enters `add_and_join("Alice")` and adds Alice to the `players` data structure (then releases the `players` lock).
- Thread 2 enters `ban_player("Alice")` and removes Alice from all `games` (then releases the `games` lock).
- Thread 2 now removes Alice from the `players` data structure; thread 1 has already released the lock so this does not block.
- Thread 1 now carries on and acquires the `games` lock (already released by thread 2). With the lock held, thread 1 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 how to avoid the problems that come 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 obviously doomed to failure in the long term.

More abstractly, multi-threaded code is an ideal place to apply the 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 behaviour 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 multi-threaded 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).

The first thing to understand about Rust's panic system is that it is *not* equivalent to an exception system (like the ones in Java or C++), even though there appears to be 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!',
panic/src/main.rs:11:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```

A wrapper that uses [std::panic::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:

```
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 behaviour so that it immediately aborts the process.

```
thread 'main' panicked at 'Cowardly refusing to divide by zero!',
panic/src/main.rs:11:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
Abort trap: 6
```

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* only occur 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` / `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 of this item, there are a few things to bear in mind. The first is that `panic`s can appear in different guises; avoiding `panic!` also involves avoiding:

- `unwrap()` and `unwrap_err()`
- `expect()` and `expect_err()`
- `unreachable!()`

Harder to spot are things like:

- `slice[index]` when the index is out of range
- `x / y` when `y` is zero.

The second observation around avoiding `panic`s 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 ([Item 32](#)) system that spots new panics is much more reliable. A simple version could be a simple grep for the most common panicking entrypoints (as above); 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](#) explored just how difficult exception safety is for C++ template code, as did 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 run-time. Given an item at run-time, it covers:

- 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 – as well as determining an item's type at run-time, its contents can be explored, its fields modified and its methods invoked. Languages that have this level of reflection support *tend* to be dynamically typed languages (e.g. Python, Ruby), but there are also some notable statically typed languages that also support this, 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 explored below) and so best avoided unless no other alternatives are possible.

The first reflection-like feature *looks* magic at first – a way of determining the name of an item's type:

```
let x = 42u32;
let y = Square::new(3, 4, 2);
println!("x: {} = {}", tname(&x), x);
println!("y: {} = {:?}", tname(&y), y);
```

```
x: u32 = 42
y: reflection::Square = Square { top_left: Point { x: 3, y: 4 },
size: 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 `std::any::type_name<T>` library function only has access to *compile-time* information; nothing clever is happening at run-time.

The string returned by `type_name` is only suitable 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: 14816064564273904734 }
y has TypeId { t: 7700407161019666586 }
```

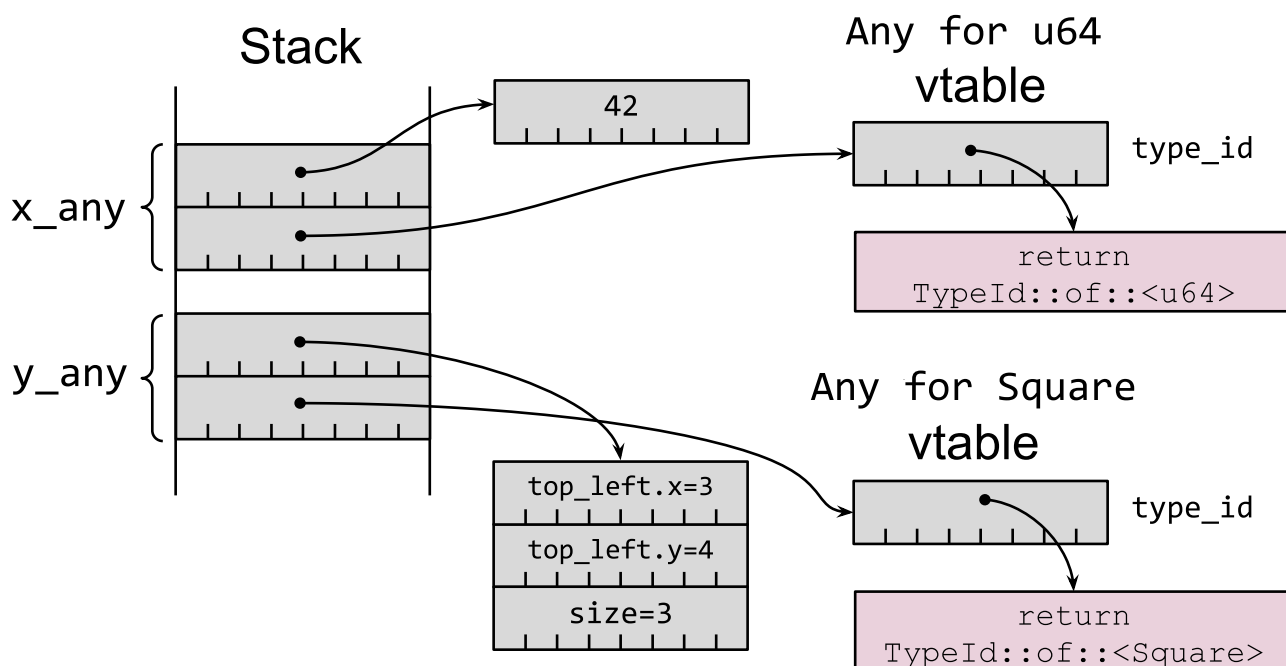
The output is less helpful for humans, but the guarantee of uniqueness means that the result can be used in code. However, it's usually best not to do so directly, but to use the [std::any::Any](#) trait¹ instead.

This 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 every type `T`:

```
impl<T: 'static + ?Sized> Any for T {
    fn type_id(&self) -> TypeId {
        TypeId::of:::<T>()
    }
}
```

Recall from [Item 9](#) 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 method that returns the item's type.

```
let x_any: Box<dyn Any> = Box::new(42u64);
let y_any: Box<dyn Any> = Box::new(Square::new(3, 4, 3));
```



Modulo a couple of indirections, a `dyn Any` trait object is effectively a combination of a raw pointer and a type identifier. This means that `Any` can offer some additional generic methods:

- `is<T>` to indicate whether the trait object's type is equal to some specific other type `T`.
- `downcast_ref<T>` which returns a reference to the concrete type `T`, provided that the type matches.
- `downcast_mut<T>` for the mutable variant of `downcast_ref`.

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 only possible if the overhead of building an `Any` trait object has happened.

There are comparatively few scenarios where Rust has different compile-time and run-time 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: Drawable` does *not* mean **is-a**, it just means **also-implements**; the vtable for `Shape` includes the entries for the methods of `Drawable`.

For some simple trait bounds:

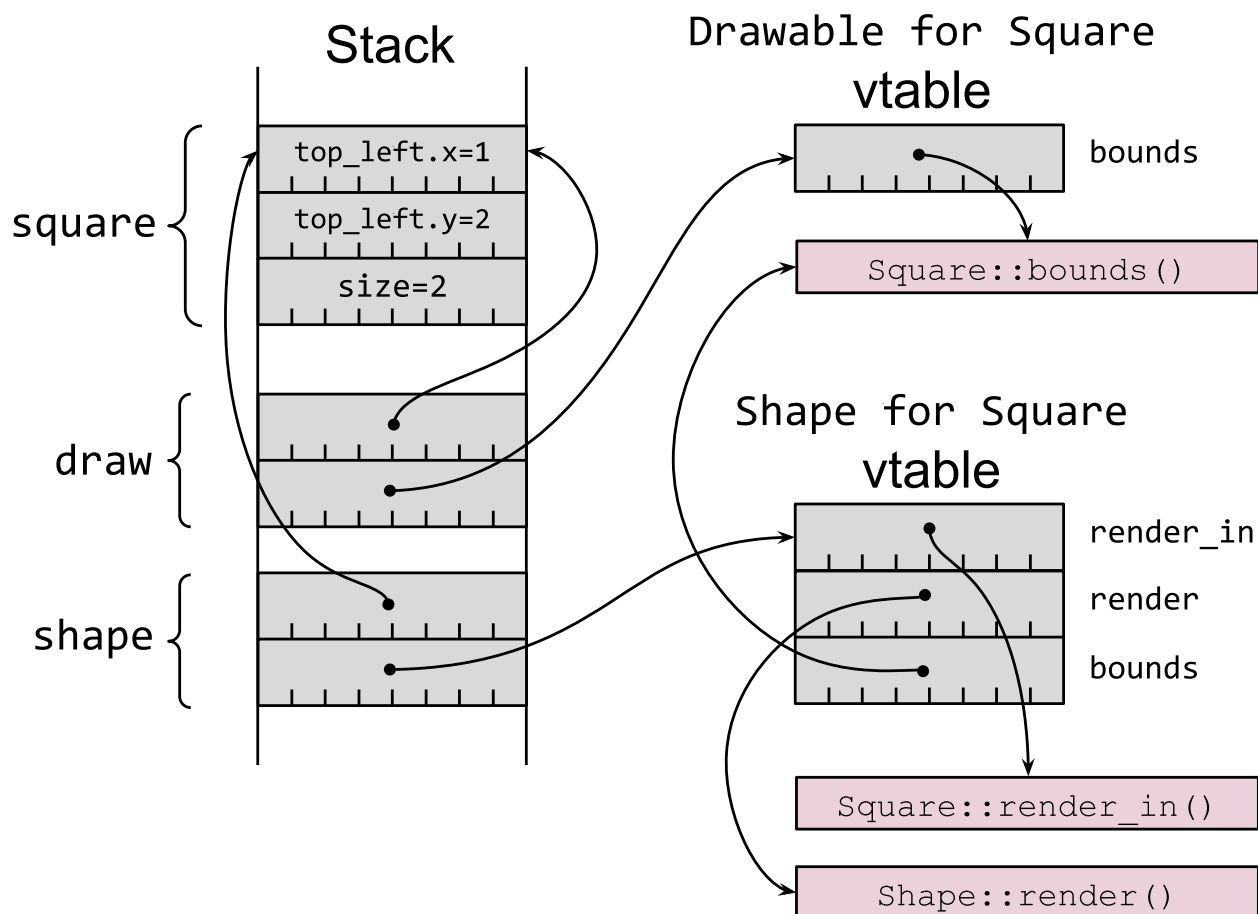
```
trait Drawable: Debug {
    fn bounds(&self) -> Bounds;
}

trait Shape: Drawable {
    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 Drawable = &square;
let shape: &dyn Shape = &square;
```

have a layout whose arrows make the problem clear: given a `dyn Shape` object, there's no way to build a `dyn Drawable` trait object, because there's no way to get back to the vtable for `impl Drawable for Square` – even though the relevant parts of its contents (the address of the `Square::bounds` method) is theoretically recoverable.



Comparing 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 run-time way to see what traits it implement, nor 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 certain behaviour being available for an item, encode that behaviour as a trait (Item 2). Even if the desired behaviour can't be expressed as a set of method signatures, use marker traits to indicate compliance with the desired behaviour – 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 whose backing code was not available at program link time, because it has been dynamically loaded at run-time (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 run-time.

1: The C++ equivalent of `Any` is `std::any`, and [advice is to avoid it too](#)

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](#)

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 re-used across many different applications was an ideal, only realized 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 re-use, was [CPAN](#): the Comprehensive Perl Archive Network, online since 1995. By the present day, almost every modern language¹ 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.

However, new problems come along with that ease, convenience and speed. It's *usually* still easier to re-use 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 part of the book will help you be aware of these.

The focus is specifically on Rust, and with it use of the [cargo](#) tool, but many of the concerns, topics and issues covered apply equally well to 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](#)"

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 = "1.0.*"
```

indicates to `cargo` what ranges of semver versions are acceptable for this dependency (see the [official docs](#) for more detail on specifying precise ranges of acceptable versions).

When choosing dependency versions, Cargo will then generally pick the most recent version that's within the combination of all of these semver ranges. However, if the `-Z minimal-versions` flag is passed to Cargo, it will instead pick the *oldest* version of each dependency that satisfies the semver ranges; **consider including a `-Z minimal-versions` build** in your CI system ([Item 32](#)) to confirm that the lower bounds of the semver ranges are accurate.

Because semantic versioning is at the heart of `cargo`'s dependency resolution process, this Item explores more details about what that means.

The essentials of semantic versioning are given by its [summary](#)

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 backwards compatible manner, and
 - PATCH version when you make backwards compatible bug fixes.
-

An important detail 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 in 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 these rules slightly, "left-shifting" the rules so that changes in the left-most 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 only be moved 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 non-compliance 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.

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 behaviour.

The difficult part for crate authors is the later 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 backwards 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 marked as `non_exhaustive`).
 - Explicitly creating an instance of a `struct` requires an initial value for all fields, so [adding a field to a structure that can be publically 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 performing explicit construction.
- Changing a trait so it is [no longer object safe](#) ([Item 2](#)) 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 MSRV increase as a [non-breaking change](#), so **consider whether the minimum supported Rust version (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 only likely to yield 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 easier¹.

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 to:

- Release a MINOR version update that includes the new version of the API, and which marks the older variant as [deprecated](#), including an indication of how to migrate.
- Subsequently 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 behaviour in a way that's actually incompatible for existing users, but which *could* re-use 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 continuous integration check ([Item 32](#)) that detects changes, using tooling (e.g. [cargo-deny](#), see [Item 31](#)) 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

As a user of a crate, the *theoretical* expectations for a new version of a dependency are:

- 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 cleaner/better ways of using the crate now. 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 re-write 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 behaviour 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.*"` or `"0.7.*"` ; **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.

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, with any security problems left unaddressed.

As a result, you either need to accept the risks of being stuck on an old version, or you need to **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 a [highly-tested monorepo that contains all the code in the world](#). As such, understanding its concepts and limitations is necessary for managing dependencies.

1: [rust-semverver](#) is a tool that attempts to do something along these lines.

Item 22: Minimize visibility

Rust's basic unit of visibility is the module; by default, a module's items (types, methods, constants) are *private* and only accessible 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. A bare `pub` is the most common version, 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.

The more-specific variants of `pub` are as follows, in descending order of usefulness:

- `pub(crate)` is accessible anywhere within the owning crate. Another way of achieving the same effect is to have a `pub` item in a non-`pub` module of the crate, but `pub(crate)` allows the item to live near the code it is relevant for.
- `pub(super)` is accessible to the parent module of the current module, which is occasionally useful for selectively increasing visibility in a crate that has a deep module structure.
- `pub(in <path>)` is accessible to code in `<path>`, which has to be a description of some ancestor module of the current module. This is even more occasionally useful for selectively increasing visibility in a crate that has an even deeper module structure.
- `pub(self)` is 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 which is not used within that module (and its submodules):

```
// Private function that's been written but which is not yet
used.
fn not_used_yet(x: i32) -> i32 {
    x + 3
}
```

Although the warning mentions code that is "never used", it's often really a warning that code *can't* be used from outside the module.

```

warning: function is never used: `not_used_yet`
--> visibility/src/main.rs:50:8
50 |     fn not_used_yet(x: i32) -> i32 {
    |         ^^^^^^^^^^^^^

```

Separately from the question of *how* to increase visibility, is the question of *when* to do so. The answer: *as little as possible*, at least for code that's intended to be re-used as a self-contained crate (i.e. not an internal or experimental project that will never be re-used).

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 only needs a minor version bump, and leaves crate users unaffected. (Read through the [API compatibility guidelines](#) and notice how many are only relevant if there are `pub` items in play).

This advice is by no means unique to this Item, nor unique to Rust:

- The Rust [API guidelines](#) include advice that
 - [Structs should have private fields](#)
- *Effective Java* (3rd edition) has:
 - Item 15: Minimize the accessibility of classes and members
 - Item 16: In public classes, use accessor methods, not public fields
- *Effective C++* (2nd edition) has:
 - 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 backwards compatible change.

The combination of these two observations raises the worry that a non-breaking 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);
```

Unfortunately, there are still cases where clashes can occur; for example, if the dependency adds a new trait and implements it for some type `T`, then if any method names from the new trait clash with existing method names in your own code for `T`

```
trait BytesLeft {
    // Method name clashes with wildcard-imported
    `bytes::Buf::remaining`...
    fn remaining(&self) -> usize;
}

impl BytesLeft for &[u8] {
    // ... and implementation for `&[u8]` clashes with `bytes::Buf`
    impl {
        fn remaining(&self) -> usize {
            self.len()
        }
    }
}

let arr = [1u8, 2u8, 3u8];
let v = &arr[1..];
assert_eq!(v.remaining(), 3);
```

then a compile-time error is the result:

```

error[E0034]: multiple applicable items in scope
--> wildcard/src/main.rs:32:22
32 |         assert_eq!(v.remaining(), 3);
    |                   ^^^^^^^^^ multiple `remaining` found
note: candidate #1 is defined in an impl of the trait `BytesLeft`
for the type `&[u8]`
--> wildcard/src/main.rs:17:5
17 |     fn remaining(&self) -> usize {
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
= note: candidate #2 is defined in an impl of the trait
`bytes::Buf` for the type `&[u8]`
help: disambiguate the associated function for candidate #1
32 |         assert_eq!(BytesLeft::remaining(&v), 3);
    |                   ~~~~~
help: disambiguate the associated function for candidate #2
32 |         assert_eq!(bytes::Buf::remaining(&v), 3);
    |                   ~~~~~

```

As a result, you should **avoid wildcard imports from crates that you don't control**. There are a couple of common exceptions to this advice, though.

Firstly, if you control the source of the wildcard import, then the concerns given above disappear. For example, it's common for a `test` module to do `import super::*`; . It's also possible for crates that use modules primarily as a way of dividing up code to have:

```

mod thing;
pub use thing::*;

```

Secondly, 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 of this Item, **consider pinning dependencies that you wildcard import to a precise version**, 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.

[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 which uses some 0.8 version of the crate:

```
# Top-level binary crate
[dependencies]
dep-lib = "0.1.0"
rand = "0.8.*"

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, and this crate internally uses¹ a 0.7 version of the `rand` crate:

```
# dep-lib library crate
[dependencies]
rand = "0.7.3"

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 non-`self` 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 non-`self` parameter `range`.

This is a non-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 out².

However, things get a lot more awkward if the library crate's API exposes a type from its dependency, making that dependency a [public dependency](#). In the example, this involves an `Rng` item – but specifically a version-0.7 `Rng` item:

```
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 high quality and widely used, and which only pulls in 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
--> re-export/src/main.rs:17:48
17 |         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
    |
    = note: required because of the requirements on the impl of
`rand::Rng` for `ThreadRng`
note: required by a bound in `pick_number_with`
--> /Users/dmd/src/effective-rust/examples/dep-lib/src
/lib.rs:17:28
17 | pub fn pick_number_with<R: Rng>(rng: &mut R, n: usize) ->
usize {
    |                                     ^^^ required by this bound in
`pick_number_with`
```


Investigating the types involved leads to confusion because the relevant traits do *appear* to be implemented – but the caller actually implements a `RngCore_v0_8_3` 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 clash³ is the underlying cause, how can you fix it? The key observation is to realize that the while the binary can't *directly* use two different versions of the same crate, it can do so *indirectly* (as in the original example above).

From the perspective of the binary author, the problem can 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:

- the types involved in the API
- the entire dependency crate.

For the example, the latter approach work 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 of the title 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 is possible because the Rust toolchain handles linking, and does not have the constraint that C++ inherits from C of needing to support separate compilation.

3: 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*. By default, Cargo will:

- download any crates named in the `[dependencies]` section of your `Cargo.toml` file from crates.io,
- finding versions that match the preferences configured in `Cargo.toml`.

There are a few subtleties lurking underneath this simple statement. The first thing to notice is that crate names form a single flat namespace (and this global namespace also overlaps with the names of *features* in a crate, see [Item 26](#)). 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 pre-registering an empty crate – is frowned upon, unless you really are going to release in the near future.)

As a minor wrinkle, there's also a slight difference between what's allowed as a crate name in this 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 aspect to be aware of is Cargo's [version selection algorithm](#). Each `Cargo.toml` dependency line specifies an acceptable range of versions, according to semver ([Item 21](#)) 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 pick the most recent version of the crate within the overlap.

However, if there is no semver-compatible overlap, then Cargo will build multiple copies of the dependency at different versions. 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 only be a single version of any function, constant or global variable. This is most commonly encountered with the [ring](#) cryptographic library, because it includes parts of the [BoringSSL library](#) (which is written in C and assembler).

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.

Once Cargo has picked acceptable versions for all dependencies, its choices are recorded in the `Cargo.lock` file. Subsequent builds will then re-use 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 that:

- 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 continuous integration ([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 only see the latter version.

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 too-specific a 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 only allows a single version of a crate to be used within a semver-compatible range).

- **Avoid too-general a version dependency:** it's *possible* to specify a version dependency (`"*"`) that allows non-semver compatible versions to be included, but it's a bad idea: do you really mean that the crate can completely change every aspect of its API and your code will still work? Thought not.

The most common Goldilocks specification 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"`)

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 latter inevitably becomes necessary for large projects, particularly for the multiple-version warning.

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 continuous integration 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; 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, including:

- `--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](#)).

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 yourself¹.

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 yanking 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](#) or by more sophisticated attacks².

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. That means that a compromised dependency could end up running a cryptocurrency miner as part of your continuous integration 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.

1: If you are targetting 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](#)).

2: Go's [module transparency](#) takes a step towards addressing this, by ensuring that all clients see the same code for a given version of a package.

Item 26: Be wary of feature creep

Rust includes support for [conditional compilation](#), which is controlled by `cfg` (and `cfg_attr`) attributes. These attributes govern the thing – function, line, block etc. – that they are attached to (in contrast to C/C++'s line-based preprocessor), based on configuration options that are either plain names (e.g. `test`) or key-value pairs (e.g. `panic = "abort"`).

The toolchain populates a variety of config values that describe the target environment, including 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 only compiled in when building for that target.

The [Cargo](#) package manager builds on this base `cfg` mechanism to provide the concept of **features**: specific named aspects of the functionality of a crate that can be enabled when building the crate. Cargo ensures that the `feature` option is populated with 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 four features:

```
[features]
default = ["featureA"]
featureA = []
featureB = []
featureAB = ["featureA", "featureB"]
schema = []

[dependencies]
rand = { version = "^0.8", optional = true }
hex = "^0.4"
```

However, the four features are *not* just the four lines in the `[features]` stanza; there are a couple of subtleties to watch out for.

Firstly, the `default` line in the `[features]` stanza is a special feature name, used to indicate 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 }
```

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. If the crate is compiled with `--features rand`, then that dependency is activated (and the crate will presumably include code that uses `rand` and which is protected by `#[cfg(feature = "rand")]`).

This also means that *crate names and feature names share a namespace*, even though one is global (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.

So you can **determine a crate's features by examining `[features]` and 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, 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, in order to satisfy everyone¹. 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.

A specific consequence of this 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;
}

```

This leaves external trait implementors in a quandary: should they implement the `cddl()` method or not?

For code that doesn't use the `schema` feature, the answer seems to obviously be "No" – the code won't compile if you do. But if something else in the dependency graph *does* use the `schema` feature, an implementation of this method suddenly becomes required. 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 independent² features, 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 continuous integration system ([Item 32](#)) covers all of these 2^N combinations, in all of the available test variants ([Item 30](#)).

Summing up the aspects of features to be wary of:

- Features should be additive.
- Feature names should be carefully chosen not to clash with potential dependency names.
- Having lots of independent features potentially leads to a combinatorial explosion of different build configurations.

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.

¹: The `cargo tree --edges features` command can help with determining which features are enabled for which crates, and why.

2: 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 section 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 only gets run twice. But for most of the things described in this section, 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, throw-away 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](#), but there are some specific details worth highlighting.

```
/// Calculate the [BoundingBox] that exactly encompasses a pair  
/// of [BoundingBox] objects.  
pub fn union(a: &BoundingBox, b: &BoundingBox) -> BoundingBox {
```

- **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.
- **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 does also mean 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`.

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 attribute¹:

```
#![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
note: the lint level is defined here
  --> docs/src/main.rs:2:9
   |
2  | #![deny(broken_intra_doc_links)]
   |           ^^^^^^^^^^^^^^^^^^^^^
   = help: to escape `[` and `]` characters, add `\[` before them
   like `\[` or `\[`
error: could not document `docs`
```

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 below.

As ever, any tooling that detects potential problems should form a part of your continuous integration 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 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.

For a published crate, any top-level `README.md` file² for the project is presented as the main page content on the `crates.io` listing for the crate (for example, see the [rand crate page](#)). However, this content is *not* easily visible on the documentation pages, and so serves a somewhat different purpose – it's aimed at people who are choosing what crate to use, rather than figuring out how to use a crate they've already included (although there's obviously considerable overlap between the two).

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 help 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:

```

    /// 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 behaviour 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 advice above 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 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 behaviour 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 is true for documentation (albeit not literally): focusing on the semantics, the *whys* and the *why nots*, gives text that is more likely to remain helpful in the long run.

1: Historically, this option used to be called `intra_doc_link_resolution_failure`.

2: The default behaviour of automatically including `README.md` can be overridden with the `readme` field in `Cargo.toml`.

3: Scott Meyers *More Effective C++* 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](#)"

Rust's macro systems allow you 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 which would otherwise need to be kept in sync manually.

The macros that programmers coming to Rust are most likely to have previously encountered are those provided by C/C++'s preprocessor. However, the Rust approach is a completely different beast – where the C preprocessor performs textual substitution on the tokens of the input text, Rust macros instead operate on the *abstract syntax tree* (AST) of the program.

This means that Rust macros can be aware of code structure and can consequently avoid entire classes of macro-related footguns. In particular, Rust macros are *hygienic* – they cannot accidentally refer to ("capture") local variables in the surrounding code.

One way to think about macros is see it 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 chunks of the AST that play the same role (type, identifier, expression, etc.); the implementation can then include any code that makes use of those chunks in the same AST role.

Macro Basics

Although this Item isn't the place to reproduce the [documentation for macros](#), a few reminders of details to watch out for are in order.

First, be aware that the scoping rules for using a 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!("double {} is {}", 2, double!(2));
}
```



```
macro_rules! double {
    { $e:expr } => { $e * $e }
}
```

```
fn after() {
    println!("double {} is {}", 2, double!(2));
}
```

error: cannot find macro `double` in this scope

--> macros/src/main.rs:4:36

```
4 |     println!("double {} is {}", 2, double!(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 defined in a module:

```
mod defn {
    #[macro_export]
    macro_rules! treble {
        { $e:expr } => { $e * $e * $e }
    }
}

mod user {
    pub fn use_macro() {
        // Note: *not* `crate::defn::treble!`
        let cubed = crate::treble!(3);
        println!("treble {} is {}", 3, cubed);
    }
}
```

Procedural macros (which are macros that get access to the program's syntax tree at compile time), also have a limitation around code location, in that they must be defined in a separate crate from where they are used.

Even though Rust's macros are safer than C preprocessor macros, there are still a couple of minor gotchas to be aware of in their use.

The first is to realize that even if a macro invocation *looks* like a function invocation, it's not. In particular, the normal intuition about whether parameters are moved or & referred to doesn't apply:

```

    let mut x = Item { contents: 42 }; // type is not `Copy`
    inc!(x); // Item is *not* moved, despite the (x) syntax, and
*can* be modified
    println!("x is {:?}", x);

```

The exclamation mark serves as a warning: the expanded code for the macro may do arbitrary things to/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 behaviour aligns with normal Rust** where possible.

For example, a macro 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"));
        }
    }
}

```

makes the control flow of the calling code somewhat obscure:

```

let rc = perform_http_operation();
check_successful!(rc); // may silently exit the function

// ...

```

An alternative version of the macro that generates code which 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 calling 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, watch out for repeated use of the argument in the macro:

```
let mut x = 1;
let y = double!({
    x += 1;
    x
});
println!("x = {}, y = {}", x, y);
// output: x = 3, y = 6
```



Assuming that this behaviour isn't intended, the solution is simply to evaluate the expression once, and assign the result to a local variable:

```
macro_rules! double_once {
    { $e:expr } => { { let x = $e; x*x } }
}
// output now: x = 2, y = 4
```

When to use Macros

The primary reason to use macros is to avoid repetitive 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 instances 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 different types, encapsulate that code into a generic and trait bound, and use the generic from all of the repeated places.
- If you repeat the same structure of code in multiple different 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 only be done by a macro:

```

enum Multi {
    Byte(u8),
    Int(i32),
    Str(String),
}

#[macro_export]
macro_rules! values_of_type {
    { $values:expr, $variant:ident } => {
        {
            let mut result = Vec::new(); // explicit use of Vec
allows type deduction
            for val in $values {
                if let Multi::$variant(v) = val {
                    result.push(v.clone());
                }
            }
            result
        }
    }
}

fn use_multi() {
    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);
    assert_eq!(ints, vec![1000]);
    let bytes = values_of_type!(&values, Byte);
    assert_eq!(bytes, vec![1u8, 2u8]);
}

```

Macros also allow for grouping all of the key information about a collection of data values together:

```

http_codes! {
    Continue => (100, Informational, "Continue"),
    SwitchingProtocols => (101, Informational, "Switching
    Protocols"),
    // ...
    Ok => (200, Successful, "Ok"),
    Created => (201, Successful, "Created"),
    // ...
}

```

Only the information that changes between different values is encoded, in a compact form that acts as a kind of domain-specific language (DSL) holding the source-of-truth for

the data. The macro definition then takes care of generating all of the code that derives from these values:

```
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 TryFrom<i32> for Status {
            type Error = ();
            fn try_from(v: i32) -> Result<Self, Self::Error> {
                match v {
                    $( $val => Ok(Self::$name), )+
                    _ => Err(())
                }
            }
        }
    }
}
```

If an extra value needs to be added later, rather than having to manually adjust four different places, all that's needed is a single additional line:

```
ImATeapot => (418, ClientError, "I'm a teapot"),
```

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 `file!()` and `line!()` macros from the standard library that emit source code information:

```
macro_rules! diags {
    { $e:expr } => {
        {
            let result = $e;
            if let Err(err) = &result {
                log::error!("{}: operation '{}' failed: {:?}",
                    file!(),
                    line!(),
                    stringify!($e),
                    err);
            }
            result
        }
    }
}
```

When failures occur, the log file then automatically includes details of what failed and where:

```
let x: Result<u8, _> = diags!(512.try_into());
let y = diags!(std::str::from_utf8(b"\xc3\x28")); // invalid
UTF-8
```

```
[2023-04-16T08:54:14Z ERROR macros] macros/src/main.rs:239:
operation '512.try_into()' failed: TryFromIntError(())
[2023-04-16T08:54:14Z ERROR macros] macros/src/main.rs:240:
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 previous section explained that macros allow you to create a domain-specific language 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.

This potential impenetrability of macro-based code extends beyond other engineers: various of the tools that analyze and interact with Rust code may treat the code as opaque, because it no longer necessarily follows the syntactical conventions of Rust code. Even the compiler itself is less helpful: its error messages don't always follow the chain of macro use and definition.

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 rarely 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 which 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* that no-one could confuse the two. In particular:

- **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 non-local 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 (cf. [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::ITryFromPrimitive`, `num_derive::FromPrimitive`, and `strum::FromRepr` cover some aspect of this problem.

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 that emits warnings about your Rust usage (`cargo clippy ...`), across a variety of categories:

- **Correctness:** warn about common programming errors.
- **Idiom:** warn about code constructs that aren't quite in standard Rust style.
- **Concision:** point out variations on the code that are more compact.
- **Performance:** suggest alternatives that avoid unnecessary processing or allocation.
- **Readability:** describe 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
--> clippy/src/main.rs:5:18
5 |         let pi = 3.14;
  |                   ^^^^
= note: `#[deny(clippy::approx_constant)]` on by default
= help: consider using the constant directly
= help: for further information visit https://rust-lang.github.io/rust-clippy/master/index.html#approx_constant
```

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 above, 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)]`). 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 continuous integration 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:
 - [unnecessarily converting `Result` to `Option`](#)
 - [opportunities to use `unwrap_or_default`](#).
- [Item 3](#) also suggest that errors should be returned to the caller where possible; Clippy [points out some missing opportunities to do that](#).
- [Item 5](#) described Rust's standard traits, and included some implementation requirements that Clippy checks:
 - [Ord must agree with PartialOrd](#)
 - [PartialEq::ne should not need a non-default implementation](#) (cf. [Item 13](#))
 - [Hash and Eq must be consistent](#)
 - [Clone for Copy types should match](#).
- [Item 6](#) suggests implementing `From` rather than `Into`, which [Clippy also suggests](#).
- [Item 6](#) also described [casts](#), and Clippy can warn on:
 - [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`* (disabled by default).
- **Item 9** describes **fat pointer types**, and various Clippy lints point out scenarios where there are unnecessary extra pointer indirection:
 - holding a heap-allocated collection in a `Box`
 - holding a heap-allocation collection of `Box` items
 - taking a reference to a `Box` .
- **Item 10** describes the panoply of different 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 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:
 - 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 pedantic or high-false-positive checks that are disabled by default. 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 which give 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 test. However, the Rust toolchain and extensions to it allow for various other types of test; 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:

```
#[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, which are...

- marked with the `#[test]` attribute, and included within...
- a `#[cfg(test)]` attribute, so the code only gets built in test configurations.

Other aspects of this example illustrate things that are optional, and may only be relevant 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.
- The test module may 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 of [Item 23](#) to avoid wildcard imports).
- A unit test has the ability to use anything from the parent module, whether it is `pub` or not. This allows for "whitebox" testing of the code, where the unit tests exercise internal features that aren't visible to normal users.
- The test code makes use of `unwrap()` for its expected results; the advice of [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, and the tests exercise that in a test that's marked with the `#[should_panic]` attribute. This might be 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 of [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 it's safe to include a new version of that dependency in practice – separately from whether the semantic version number ([Item 21](#)) indicates that the new version is safe in theory.

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 black-box tests that can only exercise 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>`.

Assuming that you regularly run tests as part of your continuous integration environment ([Item 32](#)), this 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 only have access to 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 continuous integration 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 above, 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 backwards 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 still 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 continuous integration, [Item 32](#)) allows you to detect when changes to the code or the toolchains adversely affect that performance.

The `cargo bench` command¹ runs special test cases that repeatedly perform an operation, and emits average timing information for the operation.

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 naïve benchmark for this code:

```
#[bench]
fn bench_factorial(b: &mut Bencher) {
    b.iter(|| {
        let result = factorial(15);
        assert_eq!(result, 1_307_674_368_000);
    });
}
```

gives incredibly positive results:

```
test naive::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 (experimental) `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 code under test to use this hint:

```
#![feature(bench_black_box)] // nightly-only

pub fn factorial(n: u128) -> u128 {
    match n {
        0 => 1,
        n => n * std::hint::black_box(factorial(n - 1)),
    }
}
```

gives more realistic results:

```
test bench_factorial          ... bench:          42 ns/iter
(+/- 6)
```

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::Bencher` functionality which is:

- more convenient (it runs with stable Rust)
- 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!` (cf. [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 favours 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 into the LLVM toolchain as [libFuzzer](#) .

The Rust compiler is built on LLVM, and so the [cargo-fuzz](#) sub-command exposes `libFuzzer` functionality for Rust (albeit only for a limited number of platforms).

To set up a fuzz test, first 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".
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
    }
}
```



Next, write a small driver that connects this entrypoint to the fuzzing infrastructure:

```
fuzz_target!(|data: &[u8]| {
    let _ = is_fuzz(data);
});
```

Running `cargo +nightly fuzz run target1` continuously executes the fuzz target with random data, only stopping if a crash is found. In this case, a failure is found almost immediately:

```

INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1139733386
INFO: Loaded 1 modules   (1596 inline 8-bit counters): 1596
[0x10cba9c60, 0x10cbaa29c),
INFO: Loaded 1 PC tables (1596 PCs): 1596
[0x10cbaa2a0,0x10cbb0660),
INFO:       7 files found in /Users/dmd/src/effective-
rust/examples/testing/fuzz/corpus/target1
INFO: -max_len is not provided; libFuzzer will not generate inputs
larger than 4096 bytes
INFO: seed corpus: files: 7 min: 1b max: 8b total: 34b rss: 38Mb
#8      INITED cov: 22 ft: 22 corp: 6/26b exec/s: 0 rss: 38Mb
thread '<unnamed>' panicked at 'index out of bounds: the len is 3
but the index is 3', fuzz_targets/target1.rs:11:12
stack backtrace:
  0: rust_begin_unwind
      at /rustc/f77bfb7336f21bfe6a5fb5f7358d4406e2597289
/library/std/src/panicking.rs:579:5
  1: core::panicking::panic_fmt
      at /rustc/f77bfb7336f21bfe6a5fb5f7358d4406e2597289
/library/core/src/panicking.rs:64:14
  2: core::panicking::panic_bounds_check
      at /rustc/f77bfb7336f21bfe6a5fb5f7358d4406e2597289
/library/core/src/panicking.rs:159:5
  3: _rust_fuzzer_test_input
  4: __rust_try
  5: __LLVMFuzzerTestOneInput
  6: __ZN6fuzzer6Fuzzer15ExecuteCallbackEPKhm
  7: __ZN6fuzzer6Fuzzer6RunOneEPKhmbPNS_9InputInfoEbPb
  8: __ZN6fuzzer6Fuzzer16MutateAndTestOneEv
  9:
 10: __ZN6fuzzer6Fuzzer4LoopERNSt3__16vectorINS_9SizedFileENS_16fuzzer_a
lllocatorIS3_EEEE
 11: __ZN6fuzzer12FuzzerDriverEPiPPcPFiPKhmE
 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 continuous integration. 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 time².

You can also make subsequent runs of the fuzzing infrastructure more efficient, by storing and re-using a *corpus* of previous inputs which the fuzzer found to explore new code paths; this helps subsequent runs of the fuzzer explore new ground, rather than re-testing 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 continuous integration 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 re-introduced in 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.

Summary

This Item has covered a lot of different types of test, so a summary is in order:

- Write unit tests for comprehensive testing that includes testing of internal-only code; run with `cargo test`.
- Write integration tests to exercise your public API; run with `cargo test`.
- Write doc tests that exemplify how to use individual items in your public API; run with `cargo test`.
- Write example programs that show how to use your public API as a whole; run with `cargo test --examples` or `cargo run --example <name>`.
- Write benchmarks if your code has significant performance requirements; run with `cargo bench`.
- Write fuzz tests if your code is exposed to untrusted inputs; run (continuously) with `cargo fuzz`.

That's a lot of different types of test, so it's up to you 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), which may be more than end users need. If that's the case, you can either [exclude](#) some of the files, or (for black-box tests) move the tests out of the crate and into a separate test crate.

¹: Support for benchmarks is not stable, so the command may need to be `cargo +nightly`

bench .

2: 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 above and beyond the essential task of converting Rust into machine code, many of which help with the wider task of maintaining a codebase, and improving the quality of that codebase.

The [tools that are present](#) 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 quick 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 compliant with semantic versioning ([Item 21](#)).
- `cargo tree` displays the dependency graph ([Item 25](#)).
- `cargo metadata` emits metadata about the packages 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, for example:

- `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, 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 continuous integration system** (as per [Item 32](#)).

Item 32: Set up a continuous integration (CI) system

A *continuous integration* (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 continuous integration system** is not at all Rust-specific, so this Item is a melange of general advice mixed with Rust-specific tool suggestions.

CI Steps

Starting with the specific, what kinds of steps should be included in your CI system? The obvious initial candidates are to:

- 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.

Throughout this book, various Items have suggested tools and techniques that can help improve your codebase; wherever possible, encode these as CI steps:

- [Item 30](#) described 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.
 - Other test types (e.g. example programs) may need to be explicitly triggered.
- [Item 29](#) waxed lyrical about the advantages of running Clippy over your code; **run Clippy in CI**.
- [Item 27](#) suggested 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 21](#) included 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.
- [Item 21](#) also described the `-Z minimal-versions` flag for Cargo's dependency resolution; **include a CI step that checks semver lower bounds are accurate**.
- [Item 26](#) described 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 25](#) mentioned 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](#) discussed the Rust tool ecosystem; consider which of these tools are worth regularly running over your codebase. For example, running `rustfmt / cargo fmt` in CI will detect code that doesn't comply with your project's style guidelines.
- [Item 33](#) suggests that you consider making library code `no_std` compatible where possible. You can only be confident that your code genuinely is `no_std` compatible 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`).

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.

These suggestions are a bit more complicated to set up, because the output of the CI step is more useful when it's compared to previous results – in an ideal world, the CI system will detect when a code change is not fully tested, or has an adverse affect on performance, and this typically involves integration with some external tracking system.

Some other suggestions for CI steps that may or may not be relevant for your codebase include:

- 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.
 - A CI step that uses the `-Z minimal-versions` option to detect whether the minimum matching versions of dependencies work correctly.
- If your project includes any kind of machine-generated resources that are version-controlled (for example, code generated from protocol buffer messages by `prost`), then include a CI step that re-generates 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 works on that platform.
- 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](#)).

Continuous integration 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); 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, Rust itself has an [extensive CI system](#) that includes dozens of steps; most of these steps are overkill for a smaller project, but some may be useful.

CI Principles

Moving from the specific to the general, there are some general principles that should guide the details of your continuous integration 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 engineer's time is tests that are *flaky*: sometimes they pass, sometimes they fail, even when the setup and codebase is 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 continuous integration system that takes a long time to run, and which only runs *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.

This may also require splitting the checks up ("Testing, Fast and Slow") if there are time-consuming tests that rarely find problems, but which are there as a back-stop to prevent obscure scenarios breaking.

However, 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 their 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, applied to problems with your processes and codebase as a whole, rather than specific lines of code. If you discover a process-like problem, then think about how the continuous integration could have caught the problem, and add that extra step before fixing the problem (and seeing the build turn green).

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 continuous integration system.

First is the good news: there are lots of free, reliable options for building a continuous integration 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.

Secondly, 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 re-usable 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:

- Restricting access so that continuous integration scripts only run automatically for known collaborators, and have to be triggered manually for new contributors.
- Pinning the versions of any external scripts to particular versions, or (better yet) specific known hashes.
- Closely monitoring any integration steps that need more than just read access to

the codebase.

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 foreign function interface (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 targetted.
- It even supports configurations that do not support heap allocation, but which only have a stack.

These non-standard Rust environments are less safe – they are often even `unsafe` – but give more options for getting the job done.

This part 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 multi-threading support to file I/O. For convenience, many of the items from `std` are automatically imported into your program, via the [prelude](#): a set of common `use` statements.

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, so the focus here is how to make sure that library code is available for those poor souls who 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 flavours of [Iterator](#).

The different names for these fundamental types starts 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 an easy way to tell if a `std::` item is available in `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). 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. This is because a `core`¹ 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, is grouped into Rust's `alloc` library.

As with `core`, these `alloc` variants are actually the same types under the covers; for example, following the [source link](#) from the documentation for `std::vec::Vec` leads to a `src/alloc/...` location.

A `no_std` Rust crate needs to explicitly opt-in to the use of `alloc`, by adding an `extern crate alloc;` declaration² to `src/lib.rs`:

```
//! My `no_std` compatible crate.
#![no_std]

// Requires `alloc`.
extern crate alloc;
```

Functionality that's enabled by turning on `alloc` includes 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`
- `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 – e.g. for libraries that don'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 multi-threaded 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 multi-threaded, 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:

- 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 for the work of making a crate `no_std`-compatible to be undone – 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 (e.g. `--target thumbv6m-none-eabi`) that does not support `std` – 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 re-use 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 function:

```
#[cfg(feature = "alloc")]  
extern crate alloc;
```

As ever with feature-gated code ([Item 26](#)), make sure that your CI system 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`), or 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, 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 to this question depends on the toolchain, target and [configuration](#), but is likely to involve `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 is likely to have 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 mechanism³ to catch allocation failures in the form of `std::bad_alloc` exceptions.

At the time of writing, Rust's inability to cope with failed allocation has been flagged in some high-profile contexts (such as the [Linux kernel](#), Android, and the [Curl tool](#)), and so work is on-going to fix the omission.

The first step is 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 a `Result<_, AllocError>` (although the `try_...` variant is currently only available with the nightly toolchain). For example:

- `Box::try_new` is available as an alternative to `Box::new`
- `Vec::try_reserve` is available as an alternative to `Vec::reserve`
- `BTreeMap::try_insert` is available as an alternative to `BTreeMap::insert`

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:

```
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| {
    MyError::new(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);
```

Fallible allocation is an area where work on Rust is on-going. The entrypoints described above will hopefully be stabilized and expanded, and there has also been a [proposal](#) to make *infallible* allocation operations controlled by a default-on feature – by explicitly disabling the feature, a programmer can then be sure that no use of infallible allocation has inadvertently crept into their program.

¹: Pun intended.

2: 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 that are optional in `no_std` environments.

3: 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 which can therefore only signal allocation failure via an exception.

Item 34: Control what crosses FFI boundaries

Even though Rust comes with with a comprehensive [standard library](#) and a burgeoning [crate ecosystem](#), there is still a lot more non-Rust code available 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; this 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++, ...).

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 of [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:

```
/* C function definition. */
int add(int x, int y) {
    return x + y;
}
```

To use this function in Rust, there needs to be an equivalent declaration:

```
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 actual code for the function. The `extern "C"` marker also automatically marks the function as `#[no_mangle]`, which is explored more below.

(Note that if the FFI functionality you want to use is just the standard C library, then you don't need to create these declarations – the `libc` crate already provides them.)

The build system will typically also need an indication of how/where to find the library holding the C code, either via the `link` attribute or the `links` manifest key.

But even this simplest of examples comes with some gotchas. First, use of FFI functions is automatically `unsafe`:

```
error[E0133]: call to unsafe function is unsafe and requires unsafe
function or block
  --> ffi/src/main.rs:156:13
    |
156 |     let x = add(1, 1);
    |               ^^^^^^^ call to unsafe function
    = note: consult the function's documentation for information on
    how to avoid undefined behavior
```

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 size of an `int` for the toolchain that compiled the C library, and
- the size of a `std::os::raw::c_int` for the Rust toolchain

are the same. But why take the chance? **Prefer sized types at FFI boundaries**, where possible – which for C means making use of the types 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.

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, 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 re-used in any way, but the introduction of C++ caused a problem. 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, and the `c++filt` tool helps translate this back into what would be visible in C++ code:

```
% nm ffi-cpp-lib.o | grep add # what the linker sees
0000000000000000 T __ZN3ns13addEii
0000000000000020 T __ZN3ns13addExx
0000000000000040 T __ZN3ns23addEii
% 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]`, which means that this level of type safety is lost – the linker only sees the "bare" names for functions and variables, and if there are any differences in type expectations between definition and use, this will only cause problems at runtime.

Accessing C Data from Rust

"I'm playing all the right notes, but not necessarily in the right order." – Eric Morecambe

Even though the example of the previous section passed the simplest possible data – an integer that fits in a machine register – between Rust and C, there were still things to be careful about. 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 realised 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 type (pun intended) 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 strings that need to be interoperable with C, and then use the `as_ptr()` method to pass the 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.

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" {
    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](#)), and these constraints get lost in the conversion to a raw pointer. That makes use of raw pointers 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 on to the value of the pointer, and use it at a later point when the associated memory has either been freed from the heap ("use-after-free"), or re-used on the stack.
- 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 spectre 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 re-use 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. */
FfiStruct* new_struct(uint32_t v);
void free_struct(FfiStruct* s);
```

with corresponding Rust FFI declarations:

```
extern "C" {
    pub fn new_struct(v: u32) -> *mut 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. 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()`
        // NULL pointers.
        unsafe { free_struct(self.inner) }
    }
}

```

The same principle applies to more than just heap memory; as described in [Item 11](#), **implement Drop to apply RAI to FFI-derived resources** – open files, database connections, etc.

Encapsulating the interactions with the C library into a wrapper struct also makes it possible to catch some other potential footguns, transforming an otherwise invisible failure into a `Result`:

```

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) {
        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 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");  
    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 of [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 mangled¹ by default (like C++), 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 {
    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 the examples above, 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 9](#) showed how Rust's memory safety prevents references to expired stack objects from being returned; those problems reappear if you try to hand out a raw pointer:

```
// 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!
}
```



because the owning `Box` is on the stack, so when it goes out of scope it will free the heap object.

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 of course raises the question of how the heap object now gets freed. The advice above was that allocating and freeing memory should happen 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) {
    let _b = unsafe { Box::from_raw(p) }; // assumes non-NULL
} // `_b` drops at end of scope, freeing the `FfiStruct`
```

This does still leave 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.

1: The Rust equivalent of the `c++filt` tool for translating mangled names back to

programmer-visible names is `rustfilt` .

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](#) warned that the toolchain wouldn't help with this – mismatches would be silently ignored, hiding problems for later.

Keeping two things perfectly in sync sounds like a good target for automation, and the Rust toolchain comes with 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](#):

```
/* C data structure definition. */
/* Changes here must be reflected in lib.rs. */
typedef struct {
    uint8_t byte;
    uint32_t integer;
} FfiStruct;

uint32_t add32(uint32_t x, uint32_t y);
int add(int x, int 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 \
          ../elsewhere/somelib.h
```

The generated Rust is identical to the hand-crafted declarations of [Item 34](#)

```

/* automatically generated by rust-bindgen 0.59.2 */

extern "C" {
    pub fn add32(x: u32, y: u32) -> u32;
}
extern "C" {
    pub fn add(
        x: ::std::os::raw::c_int,
        y: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct FfiStruct {
    pub byte: u8,
    pub integer: u32,
}

```

and can be pulled into the 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 hand-crafted artisanal 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 the Rust declaration no longer matches the C declaration; auto-generating the Rust declaration from the C declaration ensures that never happens.

This also means that the `bindgen` step is an ideal candidate to include in a continuous integration 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 particularly 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, therefore error-prone – and as noted above, 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 illustrated above¹).

This also allows a layered approach to exposing an existing C library in Rust; a common convention for wrapping some `xyzyz` library is to have:

- An `xyzyz-sys` crate that holds (just) the `bindgen`-erated code – use of which is necessarily `unsafe`.

- An `xyzyzy` 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 of [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 interoperability](#)**. 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.

Index

!, see exclamation mark

π , [1](#)

() , [1](#), [2](#)

.. , see struct update syntax

? , [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)

abstract class, [1](#)

abstract syntax tree, [1](#), [2](#), [3](#)

Adams, Douglas, [1](#)

Add , [1](#), [2](#)

AddAssign , [1](#)

AddressSanitizer, [1](#)

ADT, see algebraic data type

AFL, [1](#)

afraid, reasons to be, [1](#)

algebraic data type, [1](#), [2](#)

aliasing, [1](#)

alignment, [1](#), [2](#)

all , [1](#)

alloc , [1](#), [2](#)

allocation, fallible, [1](#)

allocation, infallible, [1](#)

allow , [1](#)

also-implements, [1](#)

Any , [1](#)

any , [1](#)

anyhow , [1](#)

Arc , [1](#), [2](#), [3](#), [4](#), [5](#)

array, [1](#), [2](#)

as , [1](#), [2](#), [3](#)

as_ref , [1](#)

AsMut , [1](#), [2](#), [3](#)

AsRef , [1](#), [2](#), [3](#), [4](#)

associated type, [1](#), [2](#)

AST, see abstract syntax tree

async , [1](#)

attacker, [1](#)

auto trait, [1](#)

B language, [1](#)

back-compatibility, [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#)

Barbossa, Hector, [1](#)

benchmarks, [1](#)

- [bindgen](#) , [1](#), [2](#)
- [BitAnd](#) , [1](#)
- [BitAndAssign](#) , [1](#)
- [BitOr](#) , [1](#)
- [BitOrAssign](#) , [1](#)
- [BitXor](#) , [1](#)
- [BitXorAssign](#) , [1](#)
- [black_box](#) , [1](#)
- [blanket implementation](#), [1](#), [2](#), [3](#), [4](#), [5](#)
- [bool](#) , [1](#), [2](#), [3](#)
- [BoringSSL](#), [1](#)
- [Borrow](#) , [1](#), [2](#)
- [borrow](#), [1](#)
- [borrow checker](#), [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#)
- [BorrowMut](#) , [1](#), [2](#)
- [Box](#) , [1](#), [2](#), [3](#)
 - [Box::from_raw](#) , [1](#)
 - [Box::into_raw](#) , [1](#)
- [BTreeMap](#) , [1](#)
- [BTreeSet](#) , [1](#), [2](#)
- [bug](#), [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)
- [build.rs](#) , [1](#), [2](#)
- [builder pattern](#), [1](#), [2](#)
- [byteorder](#) , [1](#)
- [C](#), [1](#), [2](#), [3](#), [4](#), [5](#)
 - [C99](#), [1](#)
- [C++](#), [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#), [21](#)
 - [C++11](#), [1](#), [2](#), [3](#)
 - [C++20](#), [1](#)
- [c++filt](#) , [1](#)
- [c_int](#) , [1](#)
- [cackle maniacally](#), [1](#)
- [Cargill, Tom](#), [1](#)
- [Cargo](#), [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#)
- [cargo bench](#) , [1](#)
- [cargo check](#) , [1](#)
- [cargo doc](#) , [1](#), [2](#), [3](#)
- [cargo fmt](#) , [1](#)
- [cargo metadata](#) , [1](#)
- [cargo update](#) , [1](#)
- [cargo-bench](#) , [1](#), [2](#)
- [cargo-deny](#) , [1](#), [2](#), [3](#), [4](#)
- [cargo-expand](#) , [1](#)
- [cargo-fmt](#) , [1](#)

- cargo-fuzz , 1
- cargo-tarpaulin , 1, 2
- cargo-tree , 1, 2
- cargo-udeps , 1, 2, 3
- Cargo.lock , 1, 2
- Cargo.toml , 1, 2, 3, 4, 5, 6
- cargo_metadata , 1
- cast, 1, 2, 3, 4
- catch_unwind , 1
- Cell , 1, 2, 3
- cfg , 1
- cfg_attr , 1
- chain , 1
- channel (Go), 1
- char , 1
- Chromium, 1
- CI, see continuous integration
- Clang, 1
- Clippy, 1, 2, 3, 4
- Clone , 1, 2, 3, 4, 5, 6
- cloned , 1, 2
- closure, 1
- coercion, see type coercion
- collect , 1
- conditional compilation, 1
- const , 1
- continuous integration, 1, 2, 3, 4, 5, 6, 7, 8, 9
- copied , 1
- Copy , 1, 2, 3, 4, 5
- copy semantics, 1
- core , 1, 2
- corpus, 1
- Cow , 1, 2, 3
- CPAN, 1
- crates.io , 1, 2, 3, 4, 5, 6
- Criterion, 1
- cross-compilation, 1
- CRUD, 1
- CString , 1
- cxx , 1, 2
- cycle , 1
- data race, 1, 2
- deadlock, 1, 2
- Debug , 1, 2, 3, 4

- Default , [1](#), [2](#), [3](#), [4](#)
- default features, [1](#), [2](#)
- default implementation, [1](#), [2](#), [3](#), [4](#), [5](#)
- defer (Go), [1](#)
- Dependabot, [1](#), [2](#)
- dependency graph, [1](#)
- Deref , [1](#), [2](#), [3](#), [4](#), [5](#)
- DerefMut , [1](#), [2](#), [3](#), [4](#), [5](#)
- derive , [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#)
- derive_builder , [1](#)
- directed graph, [1](#)
- Display , [1](#), [2](#), [3](#), [4](#), [5](#)
- Div , [1](#)
- DivAssign , [1](#)
- dlopen , [1](#)
- doc test, see test, doc
- domain-specific language, [1](#), [2](#)
- DoubleEndedIterator , [1](#), [2](#)
- downcast_mut , [1](#)
- downcast_ref , [1](#)
- Drop , [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#)
- drop , [1](#)
- DSL, see domain-specific language
- duck typing, [1](#)
- dynamic_cast (C++), [1](#)
- Effective C++*, [1](#), [2](#)
- Effective Java*, [1](#), [2](#), [3](#), [4](#)
- enum , [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#)
- enumerate , [1](#)
- Eq , [1](#), [2](#), [3](#)
- Err , [1](#), [2](#)
- Error , [1](#), [2](#), [3](#), [4](#)
- error handling, [1](#)
- ExactSizeIterator , [1](#), [2](#)
- exception, [1](#), [2](#)
- exception safety, [1](#)
- exclamation mark, [1](#)
- expect , [1](#), [2](#), [3](#)
- expect_err , [1](#)
- extern "C" , [1](#), [2](#)
- extern crate , [1](#)
- f32 , [1](#)
 - f32::NAN , [1](#)
- f64 , [1](#)

- f64::NAN , 1
- fallible, 1, 2
- fat pointer, 1, 2, 3, 4, 5
- @FearlessSon, 1
- feature, 1, 2, 3, 4, 5, 6, 7
- feature unification, 1, 2
- Ferris, 1
- FFI, see foreign function interface
- file!() , 1
- filter , 1
- find , 1
- flatten , 1
- floating point, 1, 2
- Fn , 1, 2, 3
- FnMut , 1, 2, 3
- FnOnce , 1, 2, 3
- fold , 1
- for , 1, 2
- for-each, 1
- for_each , 1
- foreign function interface, 1, 2, 3, 4, 5
- formal verification, 1
- format! , 1
- free (C), 1
- From , 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- from_utf8 , 1
- from_utf8_unchecked , 1
- FromIterator , 1
- function pointer, 1, 2
- functional style, 1
- fuzz test, see test, fuzz
- Galileo, 1
- generics, 1, 2, 3, 4, 5, 6, 7
 - generic function, 1, 2, 3, 4
 - generic method, 1, 2
 - generic trait, 1
 - generic type, 1
- get_mut , 1
- GitHub Actions, 1
- Gjengset, Jon, 1
- glob import, see wildcard import
- global variable, 1
- global variables, 1
- Go, 1, 2, 3, 4, 5, 6, 7, 8

- Godbolt compiler explorer, [1](#), [2](#)
- Graham, Paul, [1](#)
- Groenen, Nick, [1](#)
- Hash , [1](#), [2](#), [3](#), [4](#)
- HashMap , [1](#), [2](#), [3](#), [4](#)
- HashSet , [1](#), [2](#)
- Haskell, [1](#)
- heap, [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#)
- hygienic macro, [1](#)
- Hyrum's Law, [1](#), [2](#)
- i128 , [1](#)
- i16 , [1](#)
- i32 , [1](#)
- i64 , [1](#)
- i8 , [1](#)
- if let , [1](#)
- include! , [1](#)
- Index , [1](#), [2](#)
- IndexMut , [1](#), [2](#)
- infallible, [1](#), [2](#)
- inline , [1](#)
- integration test, see test, integration
- interior mutability, [1](#), [2](#)
- Into , [1](#), [2](#), [3](#), [4](#)
- Intolterator , [1](#), [2](#)
- is , [1](#)
- is_empty() , [1](#)
- isize , [1](#)
- iterable, [1](#)
- Iterator , [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#)
- iterator, [1](#)
- iterator transform, [1](#)
- Java, [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)
- lambda expression, [1](#)
- lattice, [1](#)
- layout, [1](#), [2](#)
- leak , [1](#)
- let , [1](#)
- libFuzzer , [1](#)
- license, [1](#), [2](#)
- lifetime, [1](#), [2](#), [3](#), [4](#)
- lifetime annotation, [1](#), [2](#)
- lifetime bounds, [1](#), [2](#)
- lifetime elision, [1](#), [2](#)

- line!() , 1
- linker, 1, 2, 3
- Liskov substitution, 1, 2
- Lisp, 1
- lock inversion, 1
 - lock_guard (C++), 1
- locking hierarchy, 1
- macro, 1, 2, 3, 4, 5, 6
 - macro, declarative, 1, 2
 - macro, derive, 1, 2, 3
 - macro, procedural, 1
- macro_export , 1
- main , 1, 2
- map , 1
- map_err , 1
- Markdown, 1
- marker trait, 1, 2, 3, 4, 5
- match , 1, 2, 3
- max , 1
- max_by , 1
- memory safety, 1
- metaprogramming, 1
- method signature, 1
- methods, 1
- Meyers, Scott, 1, 2
- min , 1
- min_by , 1
- minimal-versions , 1, 2
- minimum supported Rust version, 1, 2, 3
- Miri, 1
- module, 1, 2, 3
- monomorphization, 1, 2, 3, 4
- More Effective C++*, 1
- move semantics, 1, 2, 3
- mpsc , 1
- MSRV, see minimum supported Rust version
- Mul , 1
- MulAssign , 1
- must_use , 1
- mutable reference, 1
- Mutex , 1, 2, 3, 4, 5, 6, 7
- MutexGuard , 1, 2, 3, 4
- name mangling, 1
- Neg , 1, 2

- newtype pattern, [1](#), [2](#), [3](#)
- nm , [1](#)
- no_deadlocks , [1](#)
- no_mangle , [1](#)
- no_panic , [1](#)
- no_std , [1](#), [2](#), [3](#), [4](#)
- non-lexical lifetimes, [1](#), [2](#), [3](#)
- non_exhaustive , [1](#), [2](#)
- None , [1](#), [2](#), [3](#)
- NonNull , [1](#)
- Not , [1](#)
- nothrow (C++), [1](#)
- nth , [1](#)
- NULL (SQL), [1](#)
- nullptr (C++), [1](#)
- object safety, [1](#), [2](#), [3](#)
- object-oriented, [1](#), [2](#), [3](#), [4](#)
- OCaml, [1](#)
- ODR, see one definition rule
- Ok , [1](#), [2](#)
- once_cell , [1](#)
- one definition rule, [1](#), [2](#)
- Option , [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#)
- Ord , [1](#), [2](#), [3](#), [4](#)
- orphan rule, [1](#), [2](#), [3](#), [4](#)
- OSS-Fuzz, [1](#)
- ostrich manoeuvre, [1](#)
- ouroborous , [1](#)
- panic! , [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#)
- parking_lot::deadlock , [1](#)
- PartialEq , [1](#), [2](#), [3](#)
- PartialOrd , [1](#), [2](#), [3](#)
- partition , [1](#)
- PhantomPinned , [1](#)
- Pin , [1](#), [2](#)
- Pointer , [1](#), [2](#)
- pointer, [1](#), [2](#), [3](#)
- position , [1](#)
- prelude, [1](#), [2](#), [3](#), [4](#)
- preprocessor (C), [1](#), [2](#)
- principle of least astonishment, [1](#), [2](#)
- product , [1](#)
- Programming Rust*, [1](#)
- prost , [1](#)

- protocol buffer, [1](#)
- pub , [1](#), [2](#), [3](#)
 - pub(crate) , [1](#)
 - pub(super) , [1](#)
- Python, [1](#), [2](#)
- question mark, see ?
- RAII, [1](#), [2](#), [3](#), [4](#), [5](#), [6](#)
- rand , [1](#), [2](#), [3](#)
- Range , [1](#)
- range expression, [1](#)
- range expressions, [1](#)
- raw pointer, [1](#), [2](#), [3](#), [4](#)
- Rc , [1](#), [2](#), [3](#), [4](#), [5](#)
- re-export, [1](#), [2](#), [3](#)
- reduce , [1](#)
- Ref , [1](#)
- Ref<T> , [1](#)
- RefCell , [1](#), [2](#), [3](#), [4](#)
- RefCell<T> , [1](#)
- reference, [1](#), [2](#), [3](#), [4](#), [5](#)
- reflection, [1](#)
- reflexive, [1](#), [2](#)
 - RefMut<T> , [1](#)
- Reid, Alastair, [1](#)
- Rem , [1](#)
- RemAssign , [1](#)
- replace , [1](#), [2](#)
- repr(C) , [1](#)
- repr(transparent) , [1](#)
- required method, [1](#)
- Result , [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#)
- return , [1](#)
- rev , [1](#)
- rewrite it in Rust, [1](#)
- RIIR, see rewrite it in Rust
- ring , [1](#)
- RTTI, [1](#)
- Rust for Rustaceans*, [1](#)
- rust-semverver , [1](#)
- RustCrypto, [1](#)
- rustfilt , [1](#)
- rustfmt , [1](#), [2](#)
- Rustonomicon*, [1](#), [2](#)
- RwLock , [1](#), [2](#), [3](#)

- scan , 1
- scope, 1
- segment, 1
 - Self , 1, 2, 3, 4, 5
- semantic versioning, 1, 2, 3, 4, 5
- semicolon, 1
- semver, see semantic versioning
 - Send , 1, 2, 3, 4, 5
- sentinel values, 1, 2
- separate compilation, 1
 - serde , 1
- shared-state, 1, 2, 3, 4, 5, 6
- shared_ptr (C++), 1, 2
- Shl , 1
- ShlAssign , 1
- Shr , 1
- ShrAssign , 1
- Sized , 1, 2
 - !Sized , 1
- skip , 1
- skip_while , 1
- sleep , 1
- slice, 1
 - SliceIndex<T> , 1
- smart pointer, 1, 2, 3, 4
- Smith, Agent, 1
- software engineering, 1
 - Some , 1, 2, 3
- spin , 1
- SQL, 1
- stack, 1, 2
- stack frame, 1
- stack pointer, 1
- 'static , 1, 2
- static , 1
- std , 1, 2
- step_by , 1
- String , 1, 2, 3
- struct , 1, 2, 3, 4, 5
- struct, 1
- struct update syntax, 1, 2
- Sub , 1, 2
- SubAssign , 1
- sum , 1

- supply chain attacks, [1](#), [2](#)
- Sutter, Herb, [1](#)
 - swap , [1](#)
 - syn , [1](#)
 - Sync , [1](#), [2](#), [3](#), [4](#)
- synchronization, [1](#)
- syntactic sugar, [1](#)
- tag (Git), [1](#)
 - take , [1](#), [2](#)
 - take_while , [1](#)
- taste, [1](#)
- template (C++), [1](#), [2](#)
- temporaries, [1](#)
- test, [1](#)
 - test, doc, [1](#), [2](#), [3](#)
 - test, fuzz, [1](#)
 - test, integration, [1](#), [2](#), [3](#)
 - test, unit, [1](#), [2](#), [3](#)
- test, flaky, [1](#)
- @thingskatedid, [1](#)
 - thiserror , [1](#)
- thread-compatible, [1](#)
- thread-hostile, [1](#)
- thread-safe, [1](#), [2](#)
- ThreadSanitizer, [1](#), [2](#)
- Tolnay, David, [1](#)
 - ToOwned , [1](#), [2](#)
- trait, [1](#), [2](#), [3](#), [4](#)
- trait bound, [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#)
- trait coherence, [1](#)
- trait object, [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#)
 - try_find , [1](#)
 - try_fold , [1](#)
 - try_for_each , [1](#)
- TryFrom , [1](#), [2](#), [3](#)
- TryInto , [1](#), [2](#)
- tuple, [1](#)
- tuple struct, [1](#), [2](#)
- Turon, Aaron, [1](#)
- type alias, [1](#), [2](#)
- type coercion, [1](#), [2](#), [3](#), [4](#)
- type erasure, [1](#)
- type system, [1](#), [2](#), [3](#), [4](#)
 - type_name , [1](#)

- TypeId , 1
- typeid (C++), 1
- u128 , 1
- u16 , 1
- u32 , 1
- u64 , 1
- u8 , 1
- Unicode, 1, 2
 - unique_ptr (C++), 1
- unit test, see test, unit
- unit type, see ()
- Unpin , 1
- unreachable! , 1
- unsafe , 1, 2, 3, 4, 5, 6, 7
- unwrap , 1, 2
- unwrap_err , 1
- unzip , 1
- use , 1, 2, 3
- use-after-free, 1
- usize , 1
- UTF-8, 1, 2, 3
- Vec , 1, 2, 3, 4
- vigilance, constant, 1
- void (C), 1
- vtable, 1, 2, 3, 4, 5, 6, 7
- Weak , 1, 2
 - weak_ptr (C++), 1
- WebAssembly, 1
- wildcard dependency, 1, 2
- wildcard import, 1, 2, 3
- Wilde, Oscar, 1
- Winters, Titus, 1, 2
- zip , 1