# Rust for the Polyglot Programmer

*Ian Jackson and contributors*

***20th December 2022***

Rust for the experienced programmer, a guide.

See the Introduction for the rubric, goals, and non-goals. See the Colophon for other formats available, authorship and acknowledgements, making contributions and corrections, and document source code,

# Contents

# 1 Introduction and overview

There are many guides and introductions to Rust.

This one is something different: it is intended for the experienced programmer who already knows many other programming languages.

I try to be comprehensive enough to be a starting point for any area of Rust, but to avoid going into too much detail except where things are not as you might expect.

Also this guide is not entirely free of opinion, including recommendations of libraries (crates), tooling, etc.

## 1.1 Alternatives or supplements to this guide

- Ralf Biedert's "Rust Language Cheat Sheet", A comprehensive summary reference, with diagrams, running to 68 pages in PDF form.
- The Rust Book. Much more accessible, and less dense.
- Official docs: the Standard Library reference (excellent) and the Reference (woefully incomplete).

## 1.2 Language

Rust is a compiled language.

Rust's execution model is imperative, with strict evaluation (except that there are types that embody lazy evaluation).

Rust is statically typed, with an algebraic type system. It supports generic types (parameterised types) and generic functions. Monomorphisation and dynamic despatch are both supported (chosen at the point where a generic type is referred to). Type inference is supported in some contexts, especially local variables.

Rust is memory-safe and thread-safe, but with a clearly-defined and well-used `unsafe` escape hatch.

Concurrency is supported by multithreading, and alternatively via a green-threads-based `async` system. Concurrent Rust programs are still memory-safe.

There is no garbage collector. Stack objects are explicitly defined and automatically deallocated. Heap objects are explicitly allocated, and automatically deallocated when their references go out of scope. Rust has a novel memory and object lifetime management approach with lifetime-based aliasing/mutability rules.

There are two macro systems for metaprogramming: a pattern matcher (`macro_rules!`) and a very powerful system of arbitrary code transformation (`proc_macro`).

There are fully-supported stripped-down profiles of the Rust standard library without OS functions (`alloc`), and without even a memory allocator (`core`), for use in embedded situations.

The concrete syntax has many influences. The basic function and expression syntax resembles "bracey" languages, but with some wrinkles. Notably: `( )` are not required around the control expression for `if` etc. but `{ }` *are* required around the controlled statement block; and, presence vs absence of `;` at the end of a block is highly significant.

There is little meaningful separate compilation. The usual aggregation of the Rust libraries making up a single Rust program involves obtaining all of the source code to all the libraries and building them into a single executable with static linking.

There is a good FFI system to talk to C (and libraries for convenient interfacing to C++, WASM, Python,...) Generally, dynamic linking is still used for FFI libraries.

The unit of compilation is large: the "crate", not file or module.

## 1.3 Implementation, docs, tooling, etc.

There is one principal implementation, `rustc` which is maintained by the Rust project itself, alongside the specifications and documentation.

Compilation is slow by comparison with many other modern languages, but the runtime speed of idiomatic Rust code is extremely good.

Code generation (to native code or WASM) is currently done via LLVM but projects to allow use of GCC and Cranelift are both at an advanced stage. There is also an IR interpreter used mostly for validation.

There is no formal language specification. The Rust Reference has most of the syntax but usually lacks important information about semantics and details.

The standard library documentation is excellent and comprehensive.

For unsafe code, which plays with raw pointers etc., the semantics are formally but unofficially described in Stacked Borrows and programs can be checked by Miri, the interpreter for the Rust Mid-Intermediate Representation.

Rust is available in "stable", "beta" and "nightly" flavours. Rust intends to avoid (and in practice, generally does avoid) breaking existing code which was using stable interfaces.

There is excellent support for cross-compilation.

The project provides an online playground for playing with and sharing small experiments. This is heavily used as a stable way to share snippets, repros, etc., including in bug reports.

Obtaining Rust is canonically done with rustup, a pre-packaged installer/updater tool. rustup's rather alarming `curl|bash` install rune is mitigated by the care taken by the rustup maintainers; however, you will also end up using cargo which is more of a problem.

## 1.4 Library ecosystem

Rust relies heavily on its ecosystem of libraries (aka "crates"), and its convenient but securitywise-troubling language-specific package manager `cargo`. It is not practical to write any but the smallest programs without using external libraries.

Conversely, the library ecosystem is rich and generally of high quality although slightly lacking in certain areas (especially "webby" areas when compared with more "webby" languages).

The Rust ecosystem contains some exceptional and unique libraries, which can conveniently provide advanced capabilities found elsewhere only in special-purpose or research languages (if at all).

The combination of static linking of Rust libraries, with heavy use of monomorphised generic code, can lead to very large binaries.

## 1.5 The Rust Project

The Rust Project has robust and mature governance and review processes. The compiler implementation quality is high and the project is exceptionally welcoming.

Notable ideological features of the Rust community are:

- A strong desire to help the programmer write correct code, including a desire for the compiler to take responsibility for preventing programmer error.
- Pride in helping users write performant code.
- Effective collaboration between practising developers and academic programming language and formal methods experts. (Comprehensive survey.)

There is also a strong desire to help the programmer with accessible documentation and useful error messages, but generally ease of programming is traded off in favour of correctness, and sometimes performance.

# 2 Syntax

Rust distinguishes items, statements, and expressions. Control flow statements tend to require block expressions (`{ }`).

Also very important are patterns, which are used for variable binding and sum type matching.

Comments are usually `//` but `/*..*/` is also supported.

The top level of a module may contain only items. In particular, `let` bindings are not permitted outside code blocks.

Generally, a redundant trailing `,` is allowed at the end of lists (of values, arguments, etc.). But `;` is *very significant* and is usually either required, or forbidden.

## 2.1 Attributes

Rust code is frequently littered with `#[attributes]`. These are placed before the item or expression they apply to. The semantics are very varied. New attributes can be defined as procedural macros in libraries.

Notable is `#[derive(...)]` which invokes a macro to autogenerate code based on a data structure type. Many Rust libraries provide extremely useful derive macros for structs and enums.

The syntax `#![attribute]` applies the attribute to the thing the attribute is placed inside. Typically it is found only at the top of a whole module or crate.

Attributes are used for many important purposes:

- Conditional compilation `#[cfg(..)]`;
- Denoting functions whose value should be checked (and types which should not be simply discarded): `#[must_use]`;
- Suppressing warnings locally `#[allow(dead_code)]` or for a whole crate (at the toplevel) `#![allow(dead_code)]`;
- Enabling unstable features on Nightly `#![feature(exit_status_error)]`;
- Marking functions as tests `#[test]`;
- Request (hint) inlining `#[inline]`.
- Control a type's memory layout `#[repr(...)]`.
- Specify where to find the source for a module `#[path="foo.rs"] mod bar;`.

## 2.2 Items

```
fn function(arg0: T, arg1: U) -> ReturnValue { ... }
type TypeAlias = OtherType;                          Type alias, structural equality
pub struct Counter { counter: u64 }                  Nominal type equality
trait Trait { fn trait_method(self); }
const FORTY_TWO: u32 = 42;
static TABLE: [u8; 256] = { 0x32, 0x26, 0o11, entries... };
impl Type { ... }
impl Trait for Type { ... }
mod some_module;                                     Causes some_module.rs to be read
mod some_module { ... }                              Module source is right here
```

Rust really hates mutable globals. See under Safety.

## 2.3 Expressions

Most of the usual infix and assignment operators are available. Control flow "statements" are generally expressions:

```
{ stmt0; stmt1; }                                    With semicolon, has type ()
{ stmt0; stmt1 }                                     No semicolon, has type of stmt1

if condition { statements... }                       Can only have type ()
if condition { value } else { other_value }          No ? :, use this
if let pattern = value { ... } [else ...]            Pattern binding condition
match value { pat0 [ if cond0 ] => expr0, ... }      See Types and Patterns

'label: loop { ... }
'label: while condition { }
'label: while let pattern = expr { }                 'label: is optional of course
'label: for loopvar in something_iterable { ... }

return v                                             At end of function, it is idiomatic to just write v
break value;                                         loop only; specifies value of loop expr
break 'label value;                                  break value with named loop; Rust 1.65, Nov 2022
continue; continue 'label; break; break 'label;

function(arg0,arg1)
receiver.method(arg0,arg1,arg2)                      See on Methods
|arg0, arg1, ...| expression
|arg0: Type0, arg1: Type1, ...| -> ReturnType expression    Closures

fallible?                                            See in Error handling
*value                                               Deref, see in Traits, methods
value as OtherType                                   Type conversion (safe but maybe lossy, see in Safety)
Counter { counter: 42 }                              Constructor ("struct literal")
```

| | |
|---|---|
| `collection[index]` | Usually panics if not found, eg array bounds check |
| `thing.field` | Field of a struct with named fields |
| `tuple.0; tuple.1;` | Fields of tuple or tuple struct |
| `start..end; start..=end` | End-exclusive and -inclusive `Range` |

Note the odd semicolon rule, which determines the type of block expressions.

Missing return type on a `fn` item means `()`; missing return type on a closure means _;

## 2.4 Other statements

`let` introduces a binding.

| | |
|---|---|
| `let pattern = value;` | Irrefutable patterns |
| `let pattern = value else { diverges... };` | Refutable (Rust 1.65, Nov 2022) |
| `place = value;` | Assignment to a mutable variable or location |
| `pattern = value;` | Destructuring assignment (Rust 1.59, Feb 2022) |

Variable names may be reused by rebinding; this is often considered idiomatic.

In a block, you can define any other kind of item, which will have local scope.

## 2.5 Identifiers and scopes

Names are paths like `scope::scope::ident`.

Here `scope` can be a module, type or trait, or an external library ("crate"), or special values like `crate`, `self`, `super`.

Each Rust module (file, or `mod { }` within a file) has its own namespace. Other names are imported using `use path::to::thing;`. `use` can also refer to other crates (i.e. your `Cargo.toml` dependencies). Items can be renamed during import using `as`.

Rust has strong conventions about identifier case and spelling, which the compiler will warn you about violating:

- `snake_case`: Variables, functions and modules.
- `StudlyCaps`: Types (including enum variant names and traits).
- `SCREAMING_SNAKE_CASE`: Constants and global variables.

- is not valid in identifier names in Rust source code. In other places in the Rust world, you may see names in `kebab-case`.

Many items (including functions, types, fields of product types, etc.) can be public (`pub`) or private to the module (the default), or have more subtle visibility.

_ can often be written when an identifier is expected. For a type or lifetime, it asks the compiler to infer. For a binding, it ignores the value.

# 3 Types and patterns

Rust's type system is based on Hindley-Milner-style algebraic types, as seen in languages like ML and Haskell.

The compiler will often infer the types of variables (including closures) and also usually infer the correct types for a generic function call. Type elision is not supported everywhere, notably in function signatures and public interfaces.

## 3.1 Generics

Types, functions, and traits can be generic over other types (and over lifetimes and some types of constant). This is done with a C++-like `< >` syntax.

Generic code will be monomorphised automatically by the compiler, for all of the concrete types that are actually required.

When it is necessary to explicitly specify generic parameters, for example in a function call, one uses the turbofish syntax (so named because `::<>` looks a bit like a speedy fish):

```
let r = function::<Generic,Args>(...);
```

Generic parameters can be constrained with bounds written where they are introduced `fn foo<T: Default + Clone>() -> T;` or with where clauses `fn foo<T>() -> T where T: Default + Clone;`. Lifetimes are constrained thus: `'longer: 'shorter`, reading `:` as "outlives".

## 3.2 Types

Nominal types can be defined in terms of (combinations of) other types:

| Semantics | Syntax (definition of nominal type) |
|---|---|
| Product, named fields | `struct S { f: u64, g: &'static str }` |
| Product, tuple-like | `struct ST(u64, ());` |
| Empty products (units) | `struct Z0; struct Z1(); struct Z2{}` |
| Sum type | `enum E { V0, V1(usize), V2{ f: String, } }` |
| Uninhabited type | `enum Void { }` - see Uninhabited types |
| Generic type | e.g. `struct SG<F>{ f: F, g: &'static str }` |
| Untagged union (unsafe) | `union U {...}` |

Otherwise, types have structural equivalence.

| Semantics | Syntax (referring to a type) |
|---|---|
| Named type (see above) | `S`, `ST`, `Z0`, `E`, `Void`, `SG<u8>`, `U` |
| Empty tuple (primitive unit type) | `()` |
| Product type, tuple | `(T,)`, `(T,U)`, `(T,U,V)` etc. |
| Primitive integers | `usize`, `isize`, `u8`, `u16` .. `u128`, `i8` .. `i128` |
| Floating point (IEEE-754) | `f32`, `f64` |
| Other Primitives | `bool`, `char`, `str` |
| Array | `[T; N]` |
| Slice | `[T]` |
| References (always valid, never null) | `&T`, `&mut T`, `&'lifetime T`, `&'l mut T` |
| Raw pointers | `*const T`, `*mut T` |
| Runtime trait despatch (vtable) | `dyn Trait` |
| Existential type | `impl Trait` |
| Type to be inferred | `_`, `&'_ T`, `&'_ mut T` |

Most of these are straightforward.

**char** is a Unicode Scalar Value.

In Rust an **array** has a size fixed at compile time. (Generic types can be parameterised by constant integers, not only types, so the same code can compile with a variety of different array sizes, resulting in monomorphisation.) Often a slice is better.

A **slice** is a contiguous sequence of objects of the same type, with size known at run-time. The slice itself (`[T]`) means the actual data, not a pointer to it - rather an abstract concept. Normally one works with `&[T]`, which is a reference to a slice. This consists of a pointer to the start, and a length.

A slice is just an example of an **unsized** type (a.k.a. dynamically sized type, **DST**): a type whose size is not known at compile time. References (and heap and raw pointers) to unsized types are "fat pointers": they are two words wide - one for the data pointer, and one for the metadata.

Unsized *values* cannot be stack allocated, nor passed as parameters or returned from functions. But they can be heap allocated, and passed as references. Often, unsized references are type-erased references to sized values (see also Coercion).

**str** is identical to `[u8]` (ie, a slice of bytes), except with the guarantee that it consists entirely of valid UTF-8. As with `[u8]`, usually one works with `&str`. Making a `str` containing invalid UTF-8 is UB (and, therefore, not possible in Safe Rust). C.f. `String`, `Box<str>`

**dyn Trait** is a **trait object**: an object which implements `Trait`, with despatch done at run-time via a vtable. (Not to be confused with `impl Trait`.) `&dyn Trait` is a pointer to the object, plus a pointer to its vtable; `dyn Trait` itself is unsized.

**usize** is the type of array and slice indices. It corresponds to C `size_t`. Rust avoids the existence of objects bigger than fits into an `isize`.

The empty tuple `()`, aka "unit", is the type of blocks (incl. functions) that do not evaluate to (return) an actual value.

Unit structs are used extensively in idiomatic Rust, e.g. as things to `impl Trait` for, parameters indicating a type rather than value, and as objects to hang methods on.

### 3.2.1 Some very important nominal types from the standard library

| Purpose | Type |
|---|---|
| Heap allocation | `Box<T>` |
| Expanding vector (ptr, len, capacity) | `Vec<T>` |
| Expanding string (ptr, len, capacity) | `String` |
| Hash table / ordered B-Tree | `HashMap` / `BTreeMap` |
| Reference-counted heap allocation (no GC, can leak cycles) | `Arc<T>`, `Rc<T>` |
| Optional (aka Haskell `Maybe`) | `Option<T>` |
| Fallible (commonly a function return type) | `Result<T,E>` |
| Mutex (for multithreaded programs) | `Mutex<T>`, `RwLock<T>` |

See also our table comparing `Box`, `Rc`, `Arc`, `RefCell`, `Mutex` etc.

Other important types include: `BufReader`/`BufWriter`, `VecDeque`, `Cow`.

`Path` and `PathBuf` appear in many standard library APIs. They must be used if your program should support accessing arbitrarily named files (i.e. even files whose names are not valid Unicode). But they are very awkward. They are a veneer over `OsString`/`OsStr` which have a very very limited API, primarily because Windows is terrible. If you need to do anything nontrivial with filenames, you may need `os_str_bytes`. If you can limit yourself to valid Unicode, you can just pass `str` to the standard library file APIs.

## 3.3 Literals

| Type | Examples |
|---|---|
| integer (inferred) | `0`, `1`, `23_000`, `0x7f`, `0b010`, `0o27775` |
| integer (specified) | `0usize`, `1i8`, `0x7fu8` |
| floating point | `0.`, `1e23f64` |
| `&'static str` | `"string"`, `"\n\b\u{007d}\""`, `r#"^raw:"\.\s"#` etc. |
| `char` | `'c'`, `'\n'`, etc. |
| `&'static [u8]` | `b"byte string"` etc., `&[b'c', 42]` (actually `&[u8;2]`) |
| `[T; N]` | `["hi","there"]` (`[&str; 2]`), `[0u32;14]` (`[u32; 14]`) |
| `(), (T,), (T,U)` | `(), (None,) (42,"forty-two")` |

Numeric literals default to `i32` or `f64` as applicable, if the type is not specified and cannot be inferred.

### 3.3.1 Nominal types (e.g., structs)

Literals of nominal types use a straightforward literal display syntax. Enum variants, qualified by their enum type, are constructors (although they are not types in their own right).

Named fields can be provided in any order; the provided field values are computed in the order you provide. Aggregates can be rest-initialised with `..`, naming another value of the same type (often `Default::default()`).

Instead of `field: field`, you can just write `field`, implicitly referencing a local variable with the same name.

Using the examples from above:

```
let _ = S { f: 42, g: "forty-two" };
let _ = ST(42, ());
let _ = Z0;
let _ = Z1();
```

```
let _ = Z2{};
let _ = E::V0;
let _ = E::V1(42);
let _ = E::V2 { f: format!("hi") };
let _ = SG       { f: 0u8,                g: "type of F is inferred"  };
let _ = SG::<u8> { f: Default::default(), g: "type of F is specified" };
let f = 0u8; let _ = SG { g: "f is abbreviated", f };
```

If a nominal type has fields you cannot name because they're not `pub`, you cannot construct it.

## 3.4 Coercion

Rust does have implicit type conversions ("coercions") but only to change the type, not (in general) the value. The effect is to make many things Just Work, e.g. passing `&[T;N]` as a slice `&[T]`, or `&Struct` as `&dyn Trait` where `Struct` implements `Trait`.

Sometimes, especially with these conversions to unsized, writing *`expression`* `as _` can help, to introduce an explicit conversion to an inferred type. If the type is numeric, this can be lossy - see under Safety

## 3.5 Patterns

Rust uses functional-programming-style pattern-matching for variable binding, and for handling sum types.

The pattern syntax is made out of constructor syntax, with some additional features:

- `pat1 | pat2` for alternation (both branches must bind the same names).
- `name @ pattern` which binds `name` to the whole of whatever matched `pattern`.
- `ref name` avoids moving out of the matched value; instead, it makes the binding a reference to the value.
- `mut name` makes the binding mutable.

There is a special affordance when a reference is matched against a pattern: if the pattern does not itself start with `&` the individual bindings themselves bind references to the contents of the referred-to value (as if they had been `ref binding`).

Writing just the field name in a struct pattern binds a local variable of the same name as the field.

Unneeded (parts of) values can be ignored (not even bound) by writing _ or `...` The usual idiom to suppress the `#[must_use]` warning is `let _ = ...`.

Irrefutable patterns appear in ordinary `let` bindings and function parameters. (It is not possible to define the different pattern matches for a single function name separately like in Haskell or Ocaml; use `match`.)

Refutable patterns appear in `if let`, `let...else`, `match` and `matches!`.

`match` is the most basic way to handle a value of a sum type.

```
match variable { pat1 => ..., pat2 if cond =>, ... }
```

Here `cond` may refer to the bindings established by `pat2`.

## 3.6 Uninhabited types

You can write `!` for a function return type to indicate that it won't return. But `!` is not a first-class type in Stable Rust; you can't generally use it as a generic type parameter, etc.

You can define an enum with no variants. The standard library has `Infallible` which is an uninhabited error type, but its ergonomics are not always great. The crate `void` can help fill this gap. It provides not only a trivial uninhabited type (`Void`) but also helpful trait impls, functions and macros.

## 3.7 Other features

`#[non_exhaustive]` for reserving space to non-breakingly extend types in your published API.

`#[derive]`, often `#[derive(Trait)]`, for many `Trait`. In particular, see:

- `#[derive(Default)]`
- `#[derive(Debug)]`
- `#[derive(Clone,Copy)]`
- `#[derive(Eq,PartialEq,Ord,PartialOrd)]`
- `#[derive(Hash)]`

It is conventional for libraries to promiscuously implement these for their public types, whenever it would make sense.

If you derive `Hash`, but manually implement `Eq`, see the note in the docs for `Hash`.

# 4 Ownership, memory model

Rust has a novel ownership-based safety/memory system.

The best way to think of it is as a formalisation of the object and memory ownership rules found in C programs, which are typically documented in comments.

## 4.1 Ownership

Every object (value) in Rust has a single owner. Ownership can be lent (therefore, borrowed by the recipient), and also transferred ("moved" in Rust terms). (The reference resulting from a borrow is a machine pointer, but this is hidden from the programmer and of course might be elided by the compiler if it can.)

Objects inside other values are typically owned by that other object; but objects can also contain references to (borrows of) values held elsewhere.

Borrowing is done explicitly with the `&` reference operator. Borrows can be mutable (`&mut T`) or immutable (`&T`). The same object can be borrowed immutably any number of times, but only borrowed mutably once.

During the lifetime of a borrow, incompatible uses of the object are forbidden. In Safe Rust, incompatible uses are prevented by the borrow checker.

The lifetimes of borrows are often part of the types of objects; so types can be generic over lifetimes. For example:

```
struct CounterRef<'r>(&'r u64);
```

Even simple functions such as this

```
fn not_bonkers(s: &str) -> Option<&str> {
    if s == "bonkers" { None } else { Some(s) }
}
```

are generic over elided lifetime arguments:

```
fn not_bonkers<'s>(s: &'s str) -> Option<&'s str> {
```

Although lifetimes are part of types, there are many places where type inference is *not* supported, but lifetime inference *is* permitted (and usual). Usually, one requests lifetime inference by simply omitting the lifetime, but it can be requested explicitly with `'_`.

The special lifetime `'static` is for objects that will never go away.

## 4.2 Movement, `Copy`, `Clone`, `Drop`

Objects in Rust can be moved, without special formalities.

When you pass an owned value to a function, or it returns one to you, the value is **moved**. You can only do this with a value you own. It must not be borrowed by anyone, since moving it would invalidate any references.

This also means that Rust values do not contain addresses pointing within themselves. (Exception: see `Pin`.)

Moving in semantic terms might or might not mean that the object's memory address actually changes (perhaps the compiler can optimise away the memory copy). If it does change, the compiler will generate the necessary memcpy calls.

Usually, when you assign a value to a variable, or pass or return it, the value is moved.

Some types are "plain data": They can simply be duplicated without problem with memcpy. These types are **Copy**. `Copy` is usually implemented via `#[derive(Copy)]`. Types that are `Copy` are (semantically) copied rather than being moved out of (by assignments, parameter passing, etc.)

For other types, **Clone** is a trait with a single method `clone()` which supports getting a "new object like the original" whatever that means. You might think of it as a copy (although in the Rust world "copy" often means strictly `Copy`). For example, while `String::clone()` copies the data into a new heap allocation, `Arc::clone()` increments the reference count, rather than copying the underlying object. Obviously not every type is `Clone`.

Values are destroyed when the variable containing them goes out of scope, or (rarely) by explicit calls to `std::mem::drop` or the like. When a value is destroyed, all of its fields are automatically destroyed too. If this is nontrivial the type is said to have "drop glue" (and, obviously, it is not `Copy`).

If a type's destruction needs something more than simply destroying each of its fields, it can `impl Drop`. You provide a function **drop** which is called automatically precisely once just before the fields are themselves destroyed.

There are no special "constructors" in Rust. It is conventional to provide a function **Type::new()** for use as a constructor, but it is not special in any way. It typically does whatever setup is needed and finishes with a struct literal for the type. Conventionally, types that have a zero-argument `new()` usually implement `Default`. Constructors that take arguments are often named like `Type::with_wombat()`.

It is very common to construct from a value of another relevant type, for example via the `From` and `Into` traits, or specific methods (for purposes like complex construction or conversion, typestate arrangements, and so on).

There is no equivalent to C++'s "placement new". It is up to the caller whether the created object will go on the heap. Indeed, an object from `Type::new` might never be on the heap. Or it might be on the stack for a bit and then later be moved to the heap for example using `Box::new()`.

## 4.3 Interior mutability and runtime lifetime management

When it is necessary to share references more promiscuously, container types are provided to let you modify shared data, and manage its lifetime or mutability at runtime.

| Type | Storage: where is `T` | Lifetime mgmt | Interior mutability: `&mut T` from `&Foo<T>` | Threads `Send`/`Sync` |
|---|---|---|---|---|
| `T` [1] | itself | owner | No | Yes |
| `Box<T>` [1] | heap | owner | No | Yes |
| `Rc<T>` | heap | refcount[2] | No; `T` now immutable | No |
| `Arc<T>` | heap | refcount[2] | No; `T` now immutable | Yes |
| `RefCell<T>` | within | owner | Yes, runtime checks | `Send` |
| `Mutex<T>` | within | owner | Yes, runtime locking | Yes |
| `RwLock<T>` | within | owner | Yes, runtime locking | Yes |
| `Cell<T>` | within | owner | Only move/copy | `Send` |
| `UnsafeCell<T>` | within | owner | Up to you, `unsafe` | Maybe |
| `atomic` [3] | within | owner | Only some operations | Yes |
| `Rc<RefCell<T>>` | heap | refcount[2] | Yes, runtime checks | No |
| `Arc<Mutex<T>>` | heap | refcount[2] | Yes, runtime locking | Yes |
| `Arc<RwLock<T>>` | heap | refcount[2] | Yes, runtime locking | Yes |

1. Plain `T` and `Box` are included in this list for completeness/comparison.
2. There is no garbage collector. If you make cycles, you can leak.
3. Only types that the platform can do atomic operations on.

Here `T` can be any type, even a smart pointer (as illustrated) or reference. The use of `RefCell<&mut T>` is not uncommon. Whether `Wrapper<T>` is actually `Send` or `Sync` depends on `T` of course.

## 4.4 Borrow checker

Correctness is enforced by a proof checker in the compiler, known as the **borrow checker**.

The borrow checker is (supposed to be) sound, but not complete. The scope of its (in)completeness is not documented (and is probably not possible to document in a reasonable way). This incompleteness is often encountered in practice.

When you find your program is rejected by the borrow checker, firstly try the compiler's suggestions, which are generally very good (especially if the programmer is new to Rust).

If that fails, the right approach is to flail semi-randomly applying the various tactics you're aware of. When the program compiles, it is correct.

If the program cannot be made to compile, then one of the following is the case:

- You haven't flailed hard enough `:-)`.
- There is a mistake in the ownership model implied by the program design, or a bug. I.e. the algorithm could indeed generate or try to use incompatible references, even though you mistakenly think it can't.
- The ownership model implied by the program design is too complicated for the borrow checker. This often arises with self-referential data structures.

  Another classic example is that soundness of an implementation of `Iterator<Item=&mut T>` often depends on the *correctness* of the underlying iteration algorithm; since soundness depends on it not returning the same item twice. The borrow checker is not typically able to check the correctness of a from-scratch `impl Iterator for ...::IterMut`.

There are also a few commonly-arising particular limitations, for example one surrounding borrowing and early exits.

### 4.4.1 Tactics for fighting the borrow checker

- Copy rather than borrowing: Sprinkle `.clone()`, `.to_owned()`, etc., and/or change types to owned variants (or `Cow`).
- Introduce `let` bindings to prolong the lifetime of temporaries. (Normally if this will help the compiler will suggest it.)
- Introduce a `match`. Within the body of the `match`, all the values computed in the match expression remain live. This is often used in macros.
- Add lifetime annotations. Typically, as you add lifetime annotations, the compiler messages will become more detailed and precise. However, they will also become harder to read `:-)`. One can add lifetime annotations until the code compiles, and then commit, and start removing them again to try to trim the redundant ones.

  When applying this strategy, try to avoid reusing the same lifetime name in multiple places: keeping them separate can help identify actually-different lifetimes.
- Add redundant type and lifetime annotations to closures (`'_`, `_`, `&'_ _`, `-> &'_ _` etc.) The type and lifetime elision rules can interact badly with closures. Sometimes writing out explicit types and lifetimes (even with the `_` and `'_` inference placeholders) can make it work.
- Turn a closure into a function, and pass in the closed-over variables. Closures have complications surrounding lifetimes. There are cases where the compiler doesn't infer the correct lifetime bounds and there is no syntax to spell them. It can help to turn the closure into a `fn` (writing out all the types, sorry).

### 4.4.2 Strategies for evading the borrow checker

If you have a correct program, but the borrow checker can't see it, and you can't persuade it, you have these options:

- Use runtime ownership checking instead of compile-time checking. I.e., switch to `Arc`, `Mutex` (maybe `parking_lot`'s), `Rc`, `RefCell` etc.

  This may be not as slow as you think. `Arc` in particular is less slow than reference counting in many other languages, since you usually end up passing `&Arc<T>` or `&T` around, borrowing a reference rather than manipulating the refcount.
- Use a crate like `generational_arena`, `slotmap`, `slab` where the data structure owns the values, and your "references" are actually indices.

  These often perform very well, and are ergonomic to use.
- Completely change the algorithm and data structures (for example to make things less self-referential).
- Use `unsafe` and take on a proof obligation. How onerous that is depends very much on the situation. See Unsafe Rust.

# 5 Traits, methods

## 5.1 Methods

The `receiver.method(...)` syntax is used to call a "method": a function whose first parameter is a variant on `self`; often `&self` or `&mut self`. (`self` is a keyword; you can't choose your own name for it.)

Methods are defined in a block `impl StructName { }` (and can also be part of traits).

There is no inheritance. Some of the same effects can be achieved with traits, particularly default trait methods, and/or macro crates like `delegate` or `ambassador`.

It follows from the ownership model that a method defined `fn foo(self,...)` consumes its argument (unless it's `Copy`) so that it can no longer be used. This can be used to good effect in typestate-like APIs.

## 5.2 Traits

Rust leans very heavily on its trait system.

Rust Traits are very like Haskell Typeclasses, or C++ Concepts.

A trait defines a set of interfaces (usually, methods), including possibly default implementations. A trait must be explicitly implemented for a type with `impl Trait for Type { ... }`, giving definitions of all the items (perhaps except items with defaults).

Trait items (eg methods) and "inherent" items (belonging to a particular type) with the same name are different items. In this case when implementing a trait it can be necessary to explicitly write out the implementation of a trait method in terms of the inherent method. However, it is often idiomatic to provide functionality only through trait implementations.

When a trait has (roughly speaking) only methods, pointers to objects which implement the trait can be made into pointers to type-erased trait objects `dyn Trait`. These "fat pointers" have a vtable as well as the actual object pointer. Trait objects are often seen in the form `Box<dyn Trait>`. Ability of a trait to be used this way is called "object safety" (confusingly; it's not related to safety). The rules are a bit complicated but often a trait can be made object-safe by adding `where Self: Sized` to troublesome methods.

Rust has a strict trait coherence system. There can be only one implementation of a trait for any one concrete type, in the whole program. (Source-code level specialisation is not available in Stable Rust.) To ensure this, it is forbidden (in summary) to implement a foreign trait on a foreign type (where "foreign" means outside your crate, not outside your module).

### 5.2.1 Iterators: `Iterator`, `IntoIterator`, `FromIterator`

The `Iterator` and `IntoIterator` traits are very important in idiomatic (and performant) Rust.

Most collections and many other key types (eg, `Option`) implement `Iterator` and/or `IntoIterator`, so that they can be iterated over; this is how `for x in y` loops work: `y` must `impl IntoIterator`.

The standard library provides a large set of combinator methods on `Iterator`, for mapping, folding, filtering, and so on. These typically take closures as arguments. See also the excellent `itertools` crate.

Idiomatic coding style for iteration in Rust involves chaining iterator combinators. Effectively, Rust contains an iterator monad sublanguage with a funky syntax. (More in this essay: The problem of effects in Rust by withoutboats.)

The `.collect()` method in `Iterator` reassembles the result of an iteration back into a collection (or something which could be a collection if you squint; note for example the `FromIterator` impl for `Result`). Often one has to write the type of the desired result, perhaps like this:

```
let processed = things
    .filter_map(|t| ...)
    .map(|t| ...?; ...; Ok(u))
    .take(42)
    .collect::<Result<Vec<_>,io::Error>()?;
```

`collect` is more idiomatic than open-coding additions to a mutable collection variable: use of iterators is often faster than a `for` loop, and aggressively-Rustic style tries to minimise the use of `mut` variables.

### 5.2.2 Existential types

Rust has some very limited support for existential types. This is written `impl Trait`, and means "there is some concrete type here which implements this trait but I'm not telling you what it is". This is commonly used for functions returning iterators, and for futures (see Async Rust).

Currently this is only allowed in function signatures, typically as the return type. e.g.

```
fn get_strings() -> Result<impl Iterator<Item=String>, io::Error>;
```

It is not currently possible (on stable) to make an alias for the existential type, so you still can't name it properly, put it into variables, etc. This can be inconvenient and work is ongoing. In the meantime, the usual workaround is to use `Box<dyn Trait>` instead of `impl Trait`.

### 5.2.3 Closures and the fn pointer type

Each closure is a value of a unique opaque unnameable type implementing one or more of the special closure traits `Fn`, `FnMut` and `FnOnce`.

The different traits are because closures can borrow or own variables. If the closure modifies closed-over variables, it is `FnMut`; if it consumes them, it is `FnOnce`.

Each closure has its own separate unnameable type, so closures can only be used with polymorphism (whether monomorphised `<F: Fn()>`, or type-erased `&dyn Fn()`).

Closures borrow their captures during their whole existence, not just while they're running. This can impede their use to avoid repetition.

### 5.2.4 dyn closures

An `&dyn Fn` closure pointer is a fat pointer: closed-over data, and code pointer.

A `dyn` closure trait object cannot be passed by value because it's unsized. This can make `FnOnce` closures awkward. Use monomorphisation, `Box<dyn FnOnce>` or somehow make the closure be `FnMut`.

**5.2.5 Monomorphised closures `f: F where F: Fn()`** Monomorphisation of generic closure arguments specialises the generic function taking the closure to one which calls the specific closure.

The concrete representation of a particular closure type is an unnameable struct containing the closed-over variables.

The code is known at compile time – it is identified by the precise (unnameable) closure type – so a pointer to it not part of the closure representation at runtime. Likewise, the nature of the closed-over variables, and their uses, are known at compile-time.

The monomorphised caller of a closure calls it directly (statically known branch target). The closure can even be inlined and its code and closed-over variables intermixed with its caller's, and the outer caller's, to produce more optimal code.

**5.2.6 fn pointers** There is also a pointer type `fn(args..) -> T` but this is just a code pointer, so only actual functions, and closures with no captured variables, count.

**5.2.7 Some other key traits**

- `Deref` and `DerefMut`: method despatch (see below)
- `std::ops::*`: expression operators (overloading), incl. `Index` (`[ ]`)
- `Eq` et al for comparison, and `Hash` for putting objects in many kinds of collections.
- `From`, `Into`, `TryFrom` and `TryInto`. Prefer to `impl From` rather than `Into` if you can; that will get you `Into` automatically.
- `Debug` and `Display` for printing with `format!`, `println!` etc. and `x.to_string()`
- `io::Read`, `io::Write` (not to be confused with `fmt::Write`); `BufRead`.
- `Copy`, `Clone`, `AsRef`, `Borrow`/`ToOwned`.
- `Send`, `Sync` for thread-safety (permitting use in multithreaded programs).
- `Default` (implemented promiscuously)

## 5.3 `Deref` and method resolution

The magic traits `Deref` and `DerefMut` allow a type to "dereference to" another type. This is typically used for types like `Arc`, `Box` and `MutexGuard` which are "smart" pointers to some other type (ie, somehow a pointer, but with additional behaviour).

During method resolution, `Deref` is applied repeatedly to try to find a type with the appropriately-named method. The signature of the method is not considered during resolution, so there is no signature-based method overloading/despatch.

If it is necessary to specify a particular method, `Type::method(receiver,...)` or `Trait::method` can be used, or even `<T as Trait>::method`.

This is also required for associated functions (whether inherent or in traits) which are not methods (do not take a `self` parameter). Idiomatically this includes constructors like `T::new()` and can also include other functions that the struct's author has decided ought not to be methods. For example `Arc::downgrade` is not a method to avoid interfering with any `downgrade` method on `T`.

`Deref` effectively imports the dereference target type's methods into the method namespace of the dereferencable object. This could be used for a kind of method inheritance, but this is considered bad style (and it wouldn't work for multiple inheritance, since there can be only one deref target).

Auto-dereferencing also occurs when a reference is assigned (to a variable, or as part of parameter passing): if the type does not match, an attempt is made to see if dereferencing (perhaps multiple times) will help.

The `Deref[Mut]` implementation can be invoked explicitly with the `*` operator. Sometimes when this is necessary, one wants a reference again, so constructions like `&mut **x` are not unheard-of.

# 6 Safety, threadsafety

Most Rust code is written in Safe Rust, the memory-safe subset of Rust. Generally when people speak of Rust, they mean Safe Rust unless the context indicates otherwise.

Both optimised and unoptimised binaries are memory-safe. There is no compile-time option for reducing memory safety.

## 6.1 Safety

Safety means the lack of undefined behaviour (UB) as found in C, and generally that the program does what the programmer wrote, or crashes.

Safety in Rust does not mean the absence of errors detected at runtime. (See the chapter on error handling.)

Nor does Safe Rust guarantee the absence of memory leaks. However, in general, leaks are not very common in practice: for example, given the facilities in the standard library, leaks are only possible by making circularly referential refcounted data structures, or when using certain esoteric or explicitly-leaking functions.

## 6.2 Integers, conversion, checking

Arithmetic and type conversions are always safe, but overflow handling may need care for correct results.

The basic arithmetic operations panic on overflow in debug builds, and silently truncate (bitwise) in release builds. The `as` type conversion operator silently truncates on overflow.

The stdlib provides `checked_*` and `wrapping_*` methods, but they are not always convenient; the `Wrapping` wrapper type can be helpful.

For conversions expected to be fallible, use the `TryFrom` implementations via `TryInto::try_into()`. For conversions expected to be infallible, using `From` or `num::cast` will avoid accidentally writing a lossy raw `as` operation:

| to | i | | | | | | u | | | | | | f | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| from | 8 | 16 | 32 | 64 | 128 | size | 8 | 16 | 32 | 64 | 128 | size | 32 | 64 |
| i8 | . | . | . | . | . | T◂ | T▵ | T▵ | T▵ | T▵ | T▵ | T◂ | . | . |
| i16 | T◂ | . | . | . | . | T◂ | T◂ | T▵ | T▵ | T▵ | T▵ | T◂ | . | . |
| i32 | T◂ | T◂ | . | . | . | T◂ | T◂ | T◂ | T▵ | T▵ | T▵ | T◂ | n≈ | . |
| i64 | T◂ | T◂ | T◂ | . | . | T◂ | T◂ | T◂ | T◂ | T▵ | T▵ | T◂ | n≈ | n≈ |
| i128 | T◂ | T◂ | T◂ | T◂ | . | T◂ | T◂ | T◂ | T◂ | T◂ | T▵ | T◂ | n≈ | n≈ |
| isize | T◂ | T◂ | T◂ | T◂ | T◂ | . | T◂ | T◂ | T◂ | T◂ | T◂ | T▵ | n≈̂ | n≈ |
| u8 | T▿ | . | . | . | . | T◂ | . | . | . | . | . | T◂ | . | . |
| u16 | T◂ | T▿ | . | . | . | T◂ | T◂ | . | . | . | . | T◂ | . | . |
| u32 | T◂ | T◂ | T▿ | . | . | T◂ | T◂ | T◂ | . | . | . | T◂ | n≈ | . |
| u64 | T◂ | T◂ | T◂ | T▿ | . | T◂ | T◂ | T◂ | T◂ | . | . | T◂ | n≈ | n≈ |
| u128 | T◂ | T◂ | T◂ | T◂ | T▿ | T◂ | T◂ | T◂ | T◂ | T◂ | . | T◂ | n≈̂ | n≈ |
| usize | T◂ | T◂ | T◂ | T◂ | T◂ | T▿ | T◂ | T◂ | T◂ | T◂ | T◂ | . | n≈̂ | n≈ |
| f32 | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | . | . |
| f64 | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≠ | n≈̂ | . |

For conversion, use: **.** `From`; **T** `TryFrom`; **n** `num_traits`.
Integer `as` might: ◂ truncate; ▵ add or ▿ subtract $2^n$ (two's complement).
FP `as` might: ≠ convert NaN to zero or produce saturated integer values; ≈̂ round or give Inf; ≈ round.

## 6.3 Thread safety

Safe Rust is threadsafe. You can freely start new threads and parallelise things. The aliasing rules implied by the ownership model guarantee an absence of data races.

Of course this does not necessarily protect you from concurrency bugs (including lock deadlocks and other algorithmic bugs).

A reasonable collection of threading primitives and tools is in `std::thread` and `std::sync`. Many projects prefer the locking primitives from the `parking_lot` crate. `crossbeam` has scoped threads, which avoids everything having to be `'static`.

Multithreading in Rust can be an adjunct to, or replacement for, Async Rust.

## 6.4 Global variables

Mutable global variables `static mut` are completely unsynchronised and there is no control of reentrancy hazards. Accessing a `static mut` twice at once (including just separately making two `&mut`) is UB. Even in a single-threaded program, the reentrancy hazards remain. So *any* access to a `static mut` is `unsafe`.

Instead, either pass mutable access down your call stack, or use interior mutability.

Annoyingly, `std::sync::Mutex` is only const-initialisable in very recent Rust (1.63, August 2022). Use `parking_lot`, or `lazy_static`.

## 6.5 Unsafe Rust

If you want full manual control, `unsafe { }` exists. Many of the standard library facilities, and some important and widely used crates, are implemented using `unsafe`.

All `unsafe { }` does by itself is allow you to use unsafe facilities. When you use an unsafe facility you take on a proof obligation. How difficult a proof obligation you have depends very much on what you are doing. Sometimes it is easy.

The documentation for each facility explains what the rules are. The Reference has rules for type layout etc.

The Rust community generally tries very hard to make sound APIs for libraries which use unsafe internally. (Soundness being the property that no programs using your library, and which do not themselves use `unsafe`, have UB.) You should ensure your library APIs are sound.

Most facilities marked unsafe are unsafe because they can allow memory misuse and/or violation of the ownership and aliasing rules.

One difficulty is the lack of formal specifications. The Reference and the Nomicon have some information. It is sometimes necessary to rely on the reasonableness of the implementation. This is less bad than it sounds because the Rust community try quite hard to make things reasonable.

Aliasing rules are provenance-based. (There is no type-based alias analysis.) This has been formalised in Stacked Borrows. This was Ralf Jung's PhD thesis and has been now adopted by the Rust Project. It's not yet officially ratified as the spec but in practice it is what you must write to.

The Rust interpreter Miri (eg `cargo miri test`) will validate an execution of your program against Stacked Borrows and other aspects of a Rust Abstract Machine. With a suitable test suite, this can help give you confidence in the correctness of your code. If you are making a library with a semantically nontrivial API, soundness is something you'll have to wrestle with largely unaided. A common technique is to try to have a small internal module which uses `unsafe` but is sound, surrounded by a convenience API written entirely in Safe Rust.

Particular beartraps in Unsafe Rust are:

- Creating references which violate the aliasing rules is UB *even if the wrong aliases are never used*. Use the (often clumsy) circumlocutions in terms of raw pointers.
- Creating a reference to uninitialised memory is UB, *even if the reference is not read before the memory is initialised*. Use `MaybeUninit`.
- The automatic destructor-calling of variables that go out of scope interacts very dangerously with attempts at manual lifetime management. This can make Unsafe Rust even more hazardous than C in some cases! Use `ManuallyDrop`.
- With `#[repr(transparent)] struct X(Y)`, you may *not* assume that *things containing* X have the same layout as things containing Y. For example transmuting between `Option<Y>` and `Option<X>` is wrong.
- `mem::transmute` is an extremely powerful hammer and should be used with great care.

(Here we distinguish references `&T` from raw pointers `*T`. Safe Rust cannot use raw pointers, only references.)

# 7 Error handling

Rust has two parallel runtime error handling mechanisms: panics, and `Result` / `?`.

Do not use panics for anything except unrecoverable discovery of a programming error (eg, assertion failure).

## 7.1 `Result` / `?`

Rust has exceptionally good in-language support for functions which either return successfully, or return an error (such as an error code). This is the usual error handling style in Rust programs.

The core is the `Result` type in the standard library:

```
pub enum Result<T,E> { Ok(T), Err(E) }
```

and a postfix operator `?`. `?` applied to an `Ok` simply unwraps the inner success value `T`. `?` applied to an `Err` causes the containing function to return `Err(E)` after converting the error `E` to the containing function's error return type (using `From`).

An unfortunate downside is that all the returns from a fallible function must be written `Ok(r)` (or `return Ok(r)`). One must write `Ok(())` at the end of a function which would otherwise fall off the end implicitly returning `()`. The `fehler` macro library addresses this problem; due to language limitations it is not perfect, but even so it greatly improves the ergonomics. (For some reason `crates.io` has failed to render `fehler`'s `README.md`.)

The compiler will tell you if you forget to write a needed `?`. (If you tried to use the return value for something, it would have the wrong type; in case you don't, `Result` is marked `#[must_use]`, generating a warning.)

(`?` can also be used with `Option`.)

In quick-and-dirty programs it is common to call `unwrap` (or `expect`), on a `Result`; these panic on errors. But, the return type from `main` can be a suitable `Result`. This, plus use of `?` and a portmanteau error type like `anyhow::Error`, is usually better even in a prototype because it avoids writing `unwrap` calls that should be removed later to make the code production-ready.

### 7.1.1 Error types

The error type in a `Result` is generic.

The available and useful range of error types is too extensive to discuss here. But, consider:

- `anyhow` (or `eyre`) for a boxed portmanteau error type; good for application programs which need to aggregate many kinds of error.
- `thiserror` for defining your own error enum; good when you're writing a library.
- Defining your own unit struct as the error type for a specific function or scenario. (Perhaps several such.)

- `std::io::Error` if you primarily deal with OS errors.

In a sophisticated program errors often start out near the bottom of this list, and are progressively wrapped/converted into types nearer the top of the list.

#### 7.1.2 Crate- and module-local `Result` and `Error`

Some modules (including, for example, `std::io`) define their own type called `Error` and their own `Result` to go with it.

This can be confusing. You can tell such a `Result` from `std::result::Result` (which is in the language prelude) because it will only have one type parameter: the success value: e.g. `Result<()>` instead of `Result<(), io::Error>`.

Exercise discretion before importing an unqualified `Result` that isn't `std::result::Result`, without renaming it. Consider whether maybe `fehler`'s default `#[throws]` (meaning `#[throws(Error)]`) would be a better answer.

### 7.2 Panic

A panic is a synchronous unrecoverable failure of program execution, similar in some respects to a C++ exception.

Panics can be caused explicitly by `panic!()`, `assert!`, etc. The language will sometimes generate panics itself: for example, on arithmetic overflow in debug builds, or array bounds violation. There are no null pointer exceptions because references are never null - an optional reference is `Option<&T>`.

Libraries will sometimes generate panics, in cases of serious trouble. This should be documented, usually in an explicit `Panics` heading.

Typically panics produce a program crash with optional stack trace. Depending on the compilation settings, panics can perhaps be caught and recovered from, which involves unwinding including destroying the local variables in the unwound stack frames.

It is highly unidiomatic and inadvisable to use panics for handling of expected exceptional cases (eg, file not found). The very highest quality libraries offer completely panic-free versions of their functionality.

# 8 Macros and metaprogramming

## 8.1 Overview

Rust itself has two macro systems and cargo has a hook for e.g. build-time code generation. Each has its own section in this chapter:

- "Macros by example" `macro_rules!`
- Procedural macros `proc_macro`
- `build.rs`

**8.1.1 Macro invocation syntax** Roughly orthogonally to the two macro implementation methods, there are (broadly) three macro namespaces, with different invocation syntaxes:

- "function-like": Invoked as `macro!(...)` (where `macro` is the name of the macro). They can expand to expressions, blocks, types, items, etc.

  You can write `macro!{..}`, `macro!(..)` or `macro![..]` as you like: the macro cannot distinguish these cases, but there is generally a conventional invocation style for each macro.
- Attributes: `#[macro]` (before some language construct). The macro can filter/alter the decorated thing.
- `#[derive(Macro)]` before a struct, enum, or union. The macro does not modify the decorated data structure, but it takes it as input and can generate *additional* code.

You can qualify the macro name with a crate or module path. The rules for macro name scope, export, import, etc. are odd, and can be confusing in unusual cases.

## 8.2 "Macros by example" `macro_rules!`

`macro_rules!` defines a (function-like) macro in terms of template matches and and substitutions.

```
macro_rules! name { { template1 } => { replacement1 }, ... }
name!{ ... }
```

The contents of the macro invocation are matched against the templates in turn, stopping at the first one that matches.

Non-literal text in the template is introduced with `$`:

```
$binding:syntaxtype     syntaxtype can be one of
    block expr ident item lifetime literal
    meta pat pat_param path stmt tt ty vis
$( ... )?
$( ... )*  $( ... ),*    could be other separators besides ,
$( ... )+  $( ... ),+    could be other separators besides ,
```

In the replacement, write just `$binding` (without the syntax type).

Curious points:

- `macro_rules!` macros are partially hygienic. They can't introduce or refer to a local name in the caller's namespace, but the macro body must usually still qualify the global names it uses!
- The repetition and optional constructs have nontrivial rules to relate repetitions in the substitution to repetitions in the template, to find the the number of repetitions for the output.
- Use of the syntax item bindings has a side effect of transforming that part of the input from an unstructured token stream into a pseudo-token representing an AST node. This can cause trouble if the result is fed to further macros.
- The syntax item bindings have annoying rules about what they can be followed by. These rules appear intended to remove shift/reduce conflicts and therefore remove ambiguity, but of course the first-match over the whole set of patterns provides the ability to parse ambiguous grammars. Additionally, the rules (and indeed precisely what these tokens match) have not 100% kept pace with Rust's language evolution. The usual way to deal with this is simply to define one's macro to take `,` or `;` delimiters, whenever this problem arises. Sometimes one will resort to open-coding an ad-hoc parser which munches tokens `$tt` from the input one at a type into a "parsed" representation.
- Macros which are lexically in scope at, and precede, the call site do not need qualified names (and can be entirely local). To make a macro available elsewhere, write `#[macro_export]` before it. This will cause the macro to exist as a name in the toplevel of the current *crate* (not in the current module), from where it can be `use`d. (Rust 2015 has even odder scoping rules.)
- The provided input template keywords cannot match generics, function signatures, and many other important elements of Rust. Macro authors must accept only handling a restricted subset of the language, or using anomalous syntax (eg `[ ]` for generics instead of `< >`).

There are many details which are too fiddly to go into here.

If you want to do something exciting in a `macro_rules!` macro, the `paste` token pasting crate may be helpful.

## 8.3 Procedural macros `proc_macro`

Rust's 2nd macro system is very powerful and forms the basis for many advanced library facilities. The basic idea is: a proc_macro is a function from a `TokenStream` to a `TokenStream`.

The macro can arbitrarily modify the tokens as they pass through, and/or generate new tokens. There are libraries for parsing the token stream into an AST representation of Rust, and for quasiquoting.

proc_macros can be "function-like", but they can also be `#[attribute]`s (often used for code and particularly function transformations), or `#[derive(macro)]`s. Many important Rust facilities and libraries are derive macros.

`#[derive]` macros can define helper `#[attributes]` which may then be sprinkled on the type or its fields. These are inert, but since all attributes are visible to the macro, they can influence it. (These helper attributes are not namespaced.)

proc_macros operate at a syntactic, not semantic level. They do not have access to compiler symbol tables, or type information (other than types appearing lexically in the macro input).

Each macro invocation is independent. There is no way to pass information from one proc_macro invocation to the proc_macro expansion code for another invocation. However, it is usually possible to achieve the desired results by writing independent syntactic macros, that expand to Rust code which causes the compiler to correlate the relevant information, and calculate the implications, after all the macros have been expanded.

It is possible to have macros generate other macro definitions, at the cost of introducing some scoping/resolution order issues. Broadly, a macro-defined macro can only be available, within its crate, in lexically-subsequent code; outside its crate it can be available anywhere.

### 8.3.1 Practicalities

For Reasons, a proc_macro must be a separate crate, and, in cargo terms, package. Usually a proc_macro needs some non-macro support, or is just an affordance to help use some non-macro Rust. It is conventional to wrap a proc_macro in a package containing the non-macro code, and to use `use` to re-export the macro.

You should probably maintain the macro package as another member of a cargo workspace, alongside the non-macro facade/utilities. The macro crate ends up as a separate package on `crates.io`. It is conventional to call it `...-macros` or `...-derive`.

To write a proc_macro, you will probably want to refer to the chapter in the Reference and use some of these libraries:

- `syn` for parsing a `TokenStream` into an AST.
- `proc-macro-error` for providing pleasant error messages.
- `proc_macro2` to arrange that your main macro functionality can be tested outside of the rustc macro context.
- `quote` for quasiquoting macro output.

proc_macros are entirely unhygienic. In your macro output, you must fully qualify the names of everything you use, even things from std. The `Span` of identifiers determines their hygiene context.

## 8.4 build.rs

cargo supports running code at build-time, by providing a file `build.rs` in the toplevel containing appropriate functions. This can run arbitrary code, and includes the ability to generate `*.rs` files to be included in the current crate build.

This is an awkward way to to organise build-time code generation, because Rust is not an ideal language for writing build rules (although it can make a good language for generating Rust code).

`build.rs` can be the best choice if you want very portable build-time code generation, since it doesn't rely on anything but the Rust system that you were depending on anyway.

# 9 Async Rust

## 9.1 Introduction

Rust has an `async` system for cooperative multitasking, based on **futures** (which are rather like *promises* in e.g. JavaScript), but have some important novel features.

Async Rust is considerably less mature than the rest of the language. It can achieve rather higher performance (including, for example, lower power use in embedded setups, and better scaleability in highly concurrent server applications). But it comes at the cost of additional inconvenience and hazards.

Many important libraries (especially web libraries) provide (only) async interfaces.

I recommend using ordinary (synchronous) Rust, with multithreading where you need concurrency, unless you have a reason to do otherwise.

Good reasons to do otherwise might include: using async libraries; expecting a very wide deployment of your program; tight performance, efficiency or scaleabilty requirements; or working in a completely single-threaded environment.

Effectively, async Rust is a different dialect.

Work is ongoing to try to improve async Rust, and remove some of the rough edges.

## 9.2 Fundamentals

A magic trait `Future<Output=T>` represents an uncompleted asynchronous process.

Syntactic sugar `async { }` for both functions and blocks tells the compiler to convert the contained code into a state machine implementing the `Future` trait. An `async fn foo() -> T` actually returns `impl Future<Output=T>`.

Local variables (including lexical captures, for `async` blocks) become members of the state machine data structure, which is an anonymous type whose internals are hidden but which `impl Future`.

The special keyword constrution `.await` is to be applied to a `Future`. It introduces a yield (await) point into the generated state machine.

Utilities, types, and combinators are available for evaluating multiple futures in parallel and getting the answer from whichever finishes first (`select!`) or all of the answers (`join!`), async "iterators" (`Stream`), and so on.

The overall result is that, at a high level, much code can be written in a direct imperative style, without explicit state machines.

The usual Rust memory-safety guarantees are retained.

**9.2.1 Innards** Futures have one method, `poll`, which either returns `Ready(T)` or `Pending`.

`poll` takes a Context which has an associated `Waker`. When the future returns `Pending`, it is supposed to have recorded the `Waker` somewhere so that when the task can make progress, the `Waker` is woken.

An async Rust program contains a contraption known as the **executor** which is responsible for creating tasks (typically, it provides a `spawn` facility), keeping track of which are ready, and calling `poll` repeatedly so that the program makes progress.

## 9.3 Practicalities

**9.3.1 Choosing a runtime** The executor is not supplied by the Rust language itself. Multiple executors are available, as libraries. In practice, one needs async inter-task communication facilities, IO utilities, and so on.

The executor, and many of these other facilities, are generally provided by the async **runtime**. Many useful facilities turn out to be runtime-specific. In practice, library authors have in many cases been forced to choose a specific runtime.

Most of the important libraries use **Tokio**, a mature production-quality runtime (which actually predates, but now uses, modern async Rust language features).

Worth mentioning is `smol`, which might be good for small mostly-standalone projects.

There is also "`async-std`". Despite the name and strapline etc., "`async-std`" is not an official emanation of the Rust Project. This name grab in itself leaves a bad taste in my mouth. Also, "`async-std`" seems less comprehensive than Tokio in some areas. I prefer Tokio's APIs.

There are some glue libraries to help with bridging the gaps between different runtimes, such as `async_executors`.

**9.3.2 Mixing and matching sync and async; thread context** In a larger program, or one which makes use of diverse libraries, it can be necessary to mix-and-match sync and async code. Unlike in many other languages with async features, this is possible in Rust. There are facilities for calling async code from sync, and vice versa.

But there are gotchas. Specifically, there are complex rules about what kind of function you can call from what runtime context (ie, in what kind of thread).

For example, if you call tokio::runtime::Handle::block_on from a non-async function, to run async code from within non-async code, thinking you are not in an async execution context, but in fact the current thread is a Tokio executor thread, it will panic. Of course a sync veneer over an async library might not know if it's been called, indirectly, from an async task. If you think this might happen, you're supposed to use `spawn_blocking`.

This kind of thing complicates the liberal use of the sync/async gateway facilities. The rules, while documented, are hard to make sense of without a full mental model of the whole runtime, threading, and executor system. They are hard to follow without a full mental model of the whole program structure, including (sometimes) library implementation choices.

Complex programs may have multiple async executors and runtimes: a common way to make a sync veneer over an async library is to instantiate a "pet" executor.

**9.3.3 Pin** The state machines generated by `async { }` can contain local variables which are references to other local variables. But! Rust does not support self-referential data structures, because they cannot be moved without invalidating their internal pointers.

The solution to this is a type `Pin` which is used to wrap references (and smart pointers), and guarantees that the referenced data does not move. The type judo is confusing to think about, and is also awkward to use in practice.

Many types involved in futures (especially those you find in "manual" `impl Future`) end up with `Pin` wrappers, in a form of syntactic vinegar. Pinning brings more problems: even ordinary struct field access (projection) is not straightforward on a pinned object!

See the docs for `std::pin` and the crates `pin-project` and `pin-project-lite`.

**9.3.4 Anonymous future types, traits, etc.** Futures are not quite first-class objects in Rust. In particular, like closures, `async` blocks and `fn`s have anonymous types - types that cannot be named. But it is often necessary to store futures in structures, return them from functions (especially trait methods), and so on.

Because the type of an `async` block cannot be named, it cannot be made into an associated type in a trait implementation. So trait methods cannot simply be async.

The `impl Trait` existential type feature is nearly enough to solve this, but because one cannot write `impl Trait` anywhere except as a function return, it is often not sufficient.

If a trait method returns a different type for different implementations of the trait, it must be a nominal type, which is not possible if the function is an `async fn` (and therefore returns an anonymous future type). The usual workaround for async trait methods to return `Box<dyn Future<Output=_>>`. This is suboptimal because it requires an additional heap allocation, and runtime despatch. This workaround has been neatly productised in the `async-trait` macro package.

**9.3.5 Cancellation safety** Unlike most other languages' async systems, Rust futures are inert: they don't run unless they are polled, by an executor.

If a future is no longer needed, it is simply dropped. This can happen quite easily, for example if `select!` is used, or if a future is put explicitly into a data structure and then dropped at some point.

The effect from the point of view of an `async { }` is that the code simply stops running at some `.await`, effectively-unpredictably, discarding all of the local state.

Many straightforward-looking implementations of common tasks such as reading from incoming streams can lose data, or become desynchronised, if the local variables containing partially-processed data are simply discarded, and the algorithm later restarted from the beginning by a re-creation of the same future (eg, the next iteration of a loop containing a `select!`).

A type, future, data structure, or method, is said to be **cancellation-safe** if the underlying data structure is such that things do not malfunction if the future is dropped before completion.

There is no compiler support to ensure cancellation-safety and cancellation bugs turn up in real-world async Rust code with depressing frequency. Avoiding them is a matter of vigilance (and careful study of API docs).

While cancellation bugs do not affect the program's core memory safety, they often have security implications, because they can easily result in frame desynchronisation of network streams and other alarming consequences.

**9.3.6 Send** Most async Rust executors are multithreaded and will move tasks from thread to thread at whim. This means that every future in such a task must be `Send`, meaning it can safely be sent between threads. Therefore the local variables in async code must all be `Send`; captured references must be to `Sync` types.

Most concrete Rust types are in fact `Send`, but many generic types are not `Send` unless explicitly constrained. So `Send` (or, sometimes, `Sync`) bounds must be added, sometimes in surprising places.

The compiler errors do a pretty good job at pointing out the type or variable which is the root cause of a lack of `Send` but this is still a nuisance.

Futures don't *have* to be `Send`. In a single-threaded environment, working with non-`Send` futures is totally possible. But usually lack of `Send` is just an omission.

**9.3.7 Error messages** Async Rust has a tendency to produce rather opaque error messages referring to opaque types missing bounds, and other abstruse diagnostics.

You will get used to them, but it is in stark contrast to the rest of the language.

**9.3.8 Libraries and utilities** It is not entirely straightforward to find the right libraries to use. Matters are complicated by older decoy libraries from prior incarnations of Rust's approach to async.

You will end up using, at least:

- `std`'s builtin futures support: std::task, std::future;
- utilities from your runtime, eg: Tokio's modules and macros.
- utilities from the `futures` crate.

Unfortunately, many of these don't lend themselves to convenient blanket imports (although you should consider `use futures::prelude::*`).

Futures-related items share names with non-async thread tools (eg, `Mutex`, `mpsc`, etc., can mean different things). You will often want to use both sync and async tools in the same program. (In particular, a sync `Mutex` is often right.)

Importing the sub-module names is little better because the useful modules have generic names:

- `futures::future` vs `std::future`
- `tokio::process` vs `std::process`
- `tokio::task` vs `futures::task` vs `std::task`
- `tokio::stream` vs `futures::stream` vs the decoy (nightly-only) `std::stream`

Sometimes you'll want to use all of these in one program. Finding and naming anything is a chore!

# 10 FFI

Rust has a range of FFI support for interworking with other languages.

## 10.1 Raw C FFI

Built into the language, you can write `extern "C" { ... }` and both define and call C functions.

You have to write out a Rust version of the prototype of the C function. This is somewhat subtle, especially if the types are nontrivial. This is all, of course, `unsafe`.

You can exchange both references and raw pointers with C. If you use references, it is up to you to define the lifetimes and aliasing behaviour on the Rust side in a way consistent with the behaviour of the C. You will need to obey both C's and Rust's aliasing rules!

An `Option<&T>` is represented at the FFI as a pointer, despite the `Option`. This is because a reference `&T` cannot be null. So a null pointer corresponds to `None`. Nullable pointer arguments must appear in Rust as `Option<&T>` (or `*T`); existence of an actually-null `&T` is instant UB.

The FFI chapter in the Nomicon is comprehensive. You may also need to look at Type Layout in the Reference.

Rust's various string types are typically not the same as the platform's. Use std::ffi.

The raw FFI system has no direct interworking with C++ (but see below).

It is usual to wrap up `unsafe` FFI interfaces with a safe-to-call veneer. These are often in different crates, for separate compilation reasons etc., in which case conventionally the unsafe FFI crate is called `...-sys`.

## 10.2 FFI support crates

There are a range of crates which allow convenient interworking with a variety of languages.

- C++: `cxx`
- Python: `inline-python`, `pyo3`
- JS/DOM/WASM: `wasm-bindgen` (do *not* use `wasm-pack`), `web-sys`. `rusty_v8`.
- Java: `j4rs`, `jni`

There are others available - look on `lib.rs`.

## 10.3 FFI use in practice

The ecosystem contains Rust bindings to many C and C++ libraries. Look for a binding before writing one. However, because such bindings are largely `unsafe`, and often cannot be statically verified, correctness and quality are important considerations.

Sometimes FFI bindings to C libraries are in competition with whole replacement libraries written in Rust.

## 10.4 Alternatives - consider serde, json, etc.

serde can make it very easy for Rust to exchange marshalled data with code written in other languages.

This is often a more effective approach, especially when talking to scripting languages.

# 11 Documentation and testing

## 11.1 rustdoc

Rust's documentation generator, rustdoc, can automatically generate API documentation from appropriate comments in the Rust source.

You document an item with a `///` comment, like this:

```
/// pigpiod tick (\[us])
pub type Tick = Word;
```

`/** */` works too but is uglier and less idiomatic. `//!` is an "inner doc comment" which lives inside the thing it is documenting, and is normally used only for crates and modules.

The doc comments are in a Markdown dialect.

Rust community conventions value high-quality documentation, and especially, documentation which describes the semantics, details, and fine points of an API.

The Rust Standard Library documentation is built using rustdoc.

To invoke rustdoc to document your crate, run `cargo doc`. It will produce documentation for all your dependencies too, by default. It's nice to have that locally.

You can use include syntax, to include your `README.md` in your crate's top-level rustdoc docs too: `#![doc=include_str!("../README.md")]`.

See the Rustdoc Book.

## 11.2 Tests

Functions marked `#[test]` are treated as unit tests. They are run by `cargo test`. Multiple tests may be run at once, in different threads of a single process, so these unit test functions should avoid process-wide disruption. Panicking on failure is fine.

It is often convenient to put tests together in a module, marked `#[cfg(test)]`, if for no other reason than to avoid dead code warnings for code which exists just to support tests.

cargo supports other layouts for the test source code. The cargo documentation describes a difference between "integration tests" and "unit tests" but there is no real distinction between how they are treated or run; the distinction is just layout opinions.

For real integration tests, including anything that wants to run any executables produced by this crate, it is necessary to step outside cargo.

See also the section on Testing in the Rust Book.

## 11.3 Doctests

Code examples written like this are automatically treated as doctests:

```
/// ```
/// let hello = String::from("Hello, world!");
/// ```
```

`cargo test` compiles and runs them.

Writing ```` ```ignore ```` at the start suppresses this.

Lines inside the test starting with `#` are still used as part of the test, but don't come out in the documentation:

```
/// use std::fs::File;
///
/// # if cfg!(unix) {
/// let _ = File::open("/dev/null").unwrap();
/// # }
```

It is generally not considered good form to use this feature to hide `use`; after all, the reader will probably want the same `use` and hiding it is Really Not Helping.

## 11.4 Test annotations

Annotations are available for `#[test]` functions and doctests, including in particular `should_panic`:

```
#[test]
#[should_panic]
fn panics() { panic!() }
```

```
/// ```should_panic
/// panic!();
/// ```
```

# 12 Stability

The Rust Project and community value providing a stable platform, but also want to be able to make progress and changes.

There are a number of facilities and practices which try to achieve both, with a surprising degree of success.

## 12.1 Rust language, release channels

The Rust language itself (the compiler, the standard library, and some of the core tools) has a bespoke stability and release scheme:

There are three "channels", each representing a moving target. Stable is released periodically (about every 6 weeks). Beta is a pre-view of the next Stable and exists mostly to be tested.

The big difference is between Nightly and Beta/Stable (henceforth and elsewhere, Stable).

`rustup` can manage multiple versions of Rust. The `cargo`, `rustc`, etc. in `~/.cargo/bin` (on your `PATH`) are actually links to rustup so that you can invoke a different version with e.g. `cargo +nightly build`.

### 12.1.1 Nightly

Nightly provides numerous features which are explicitly denoted unstable. These are sometimes introduced experimentally. They are in any case subject to change without notice.

Each nightly language feature must be explicitly enabled by the use of `#![feature(something)]` at the start of the crate toplevel. Unstable command line options generally require adding `-Z unstable-options`.

There are even features which are known to be incomplete, broken, or maybe even unsound, for which an additional `#![allow(incomplete_features)]` is required.

### 12.1.2 Stable

Conversely Stable Rust aims to keep existing code working, almost entirely successfully.

Considerable care is taken when stabilising a feature, that the API and implementation is good, and that it doesn't paint Rust into unfortunate corners.

"Breaking changes" (defined as any change to the contract of the language or library or tool which might invalidate a previously-correct use) are very much the exception. Rarely, they are still considered, but they are handled very cautiously, including theoretical and practical assessment of the likely fallout.

(Actually, Stable Rust is actually simply a stabilised release branch of Nightly, so it does contain the code for all the unstable features. But measures are taken to prevent the use

of unstable features in the stable compiler. This allows the Rust Project to main one main line of development containing both the unstable work, and improvements to the stable compiler.)

## 12.2 Editions

Orthogonally to the different release channels, there are Editions of Rust. Currently, Rust 2015, 2018, and 2021 (supported by Rust 1.56, Oct 2021).

Each edition is a dialect, even with different syntax. The same compiler supports all the editions. The edition is specified at the level of a crate, and a single program may contain code from several editions.

This allows the language to evolve without breaking old code.

## 12.3 API stability management tools

The Rust language contains several features intended to allow a library API designer to warn or prevent users from relying on API properties which might change in the future.

For example, `#[non_exhaustive]` on data types which prevents an API consumer from writing code which would break when a new field or variant was added.

`impl Trait`, visibility specifiers, newtypes, and trait sealing, are also useful.

The standard library makes very extensive use of these facilities, and sets an example which the better crates largely follow.

When designing an API, you might want to take a look at the Rust Project's Rust API Guidelines. But do treat them as *opinionated guidelines*, not *rules*.

## 12.4 Libraries - semver

The Rust community has strong expectations about the API stability of Rust libraries (crates).

Cargo implements a modified semver scheme, and crates are generally expected to choose a cargo-semver-incompatible version for releases with breaking changes. The community will typically expect that any inadvertent breaking changes are reverted or fixed.

The semver scheme is like official semver, but with an additional compatibility rule for `0.x.y` versions where (for example) `0.x.(y+1)` satisfies a dependency on `0.x.y`. (In official semver, no `0.x` version is treated as compatible in any way with any other.)

That cargo expects there to be stability rules for `0.x` versions has made it feasible for many crate authors to avoid publishing a `1.0`, and inevitably many have failed to do so, for all the usual kinds of reasons. Many important and perfectly decent, stable, and reliable Rust libraries still have `0.x` version numbers.

Multiple versions of the same library can end up in the same program, and are then treated as entirely disjoint by the language. If they need to interoperate, special measures must be taken. For example, when the `log` crate makes a new incompatible release, an update is published with the old version number which is actually a compatibility facade over the new version, so that programs ending up containing a single instance of the library and its crucial global state.

# 13 Cargo

The `cargo` tool, which is used to build any nontrivial Rust program, will automatically download and build all the dependencies (from `crates.io`, typically) and (together with `rustc`) manage reuse of previous builds etc.

cargo is super-convenient for the common use cases, but also has serious problems.

## 13.1 Basics

A (git) tree can be a **workspace** containing multiple **packages**. Each package can contain multiple `rustc` **crates** (eg, a library and several binaries), but informally people often say "crate" to mean "package".

When publishing to `crates.io`, each package becomes separate.

cargo needs some metadata, from a file `Cargo.toml` in the toplevel.

cargo can often infer the intended libraries and executables in a conventionally-laid-out package. There are knobs to override these conventions. In particular it is fairly easy (and a good idea) to avoid the proliferation of `src` directories in each subdirectory of a workspace.

It is a good idea to start a new project with `cargo init`. Unlike some similar tools in other languages, the resulting tree does not contain much boilerplate.

If you make a project from scratch do not forget to include `edition = "2018"` (or similar). See Editions.

cargo maintains a calculated dependency resolution (versions and hashes of all dependencies) in `Cargo.lock`. It is conventional to commit that file for packages generating binaries, and omit it for libraries (where my personal practice is to commit `Cargo.lock.example`.)

By default cargo only operates on the crate in the cwd. If you want it to build/test/whatever the whole workspace, you must say `--workspace`.

## 13.2 Security implications

cargo and the `crates.io` ecosystem have some troublesome security properties. Since I have not seen this discussed in depth elsewhere, I will do so here.

cargo's model is heavily influenced by `npm`, whose ecosystem and usual methods of use have an appalling security record.

The Rust libraries are much less atomised than npm's. In a typical project one may end up using a handful, dozens or maybe hundreds of dependencies, but not the thousands upon thousands one sees with npm.

Both cargo and `rustc` will *run*, at build-time, code supplied by the packages they are building. There are no restrictions on what that code might do.

The `crates.io` package repository contains tarballs, and there is no mechanical linkage or machine-readable traceability of those crate tarballs back to the git repositories they were hopefully originally created from. (The `crates.io` index is maintained in git but cargo does not look at the git history of the index and does not mind if the index history rewinds, which it has done occasionally.)

Some of the more important libraries are part of library collections managed by multiple-person umbrella institutions. But many necessary libraries are standalone and owned and maintained by a single Rust developer.

### 13.2.1 Strategies

There are tools to help with the software supply chain management problem, such as `cargo-supply-chain`, `cargo-audit` and `cargo-crev`. The Rustsec advisory database even records advisories for APIs which are *capable of misuse*, even if there is no known real-world bug,

Some OS distros (e.g. Debian) are starting to maintain reasonable collections of Rust packages within the distro package repository. This puts your OS distro between you and the raw data from `crates.io`, which is likely to reduce your risk. To do this, you will probably want to configure cargo's source replacement not to look at `crates.io` but to look at your distro packages instead (sorry, link needs JS).

You may also consider some kind of privsep, where packages are built in a container or VM of some kind.

One approach is to keep all of the Rust code, and run all of the tools and the generated code, in the privsep environment. But this is not always very convenient for day-to-day development.

I have a tool `nailing-cargo` (sorry, link needs JS) which can help maintain a convenient workflow even when one doesn't want to run the Rust system in one's main environment.

## 13.3 Other problems and limitations

cargo is very easy for simple cases.

But it has limitations, bugs, and inflexibilities. Unlike most of the rest of Rust, important problems can remain outstanding for years. Some awkward limitations are even deliberate policy on the part of cargo upstream.

The situation is too complex to document here, but here are some of the key issues you may run into:

Out-of-tree builds are supported in theory, but in practice the information needed to successfully run a nontrivial test suite (or complex code generator) in an out-of-tree build is not provided to the crates being compiled. The ecosystem infrastructure does not use out-of-tree builds. So many crates' tests do not work out-of-tree, and some crates do not build. (You *can* arrange for the `target` directory to be somewhere else, if you don't mind the build still needing write access to the source tree.)

Although a stated goal of cargo is to be embeddable into other build systems, cargo does not expose the interfaces necessary to do this well. It's hard to know when to rerun cargo and when cargo's outputs changed. It's hard to get cargo to build precisely what's needed. If you want to run cargo inside `make`, you will need to resort to stamp files, and live with it sometimes doing unnecessary work.

It is not possible to have a completely local (unpublished) dependency without baking the path on the local filesystem into the depending packages' source tree.

`nailing-cargo` and other tools may help with some of these issues.

# 14 Libraries

There are many excellent Rust libraries (and also many poor ones of course). These are all collected at `crates.io`, the Rust language-specific package repository.

For most programs, use of ecosystem library packages is a practical necessity.

Rust's excellent metaprogramming system makes it possible for libraries to provide facilities that resemble bespoke language features.

When searching for libraries, usually use the opinionated catalogue at `lib.rs`. Or use "recent downloads" for the search order on `crates.io`, which is inexact but is likely to give you fate-sharing with the rest of the community, at least.

## 14.1 Libraries you should know about

- `itertools`. Superb collection of extra iterator combinators.
- `fehler`; `thiserror`; `anyhow` (or `eyre`). Error handling.
- `num`, `num-traits`, `num-derive`. Not just for "numeric" code - helpful integer conversions etc. too.
- `strum`. Iterate over enum variants; enums to strings, etc.
- `slab`, `generational_arena` or `slotmap`. Heap storage tools which safely sidestep borrowck (and are fast).
- `index_vec`. `arrayvec`. `indexmap`. Variations on `Vec` and `HashMap`.
- `easy-ext`. Conveniently define methods on other people's types.
- `rayon`: Semi-magical safe multicore parallelism as a drop-in replacement for std's serial iterators.
- `parking_lot`. Alternatives to the standard mutex etc. `parking_lot::Mutex` is const-initialisable; `std`'s only in Rust 1.63 (August 2022).
- `crossbeam`: other tools for multithreaded programming, including scoped threads.
- `chrono` (or `time`) for human-readable date/time handling API is a bit funky. Be sure to use `chrono-tz` on Unix.
- `libc` and `nix`. Take your pick. (Maybe consider `rustix`.)
- `lazy_static`, `once_cell` for data to be initialised once.

## 14.2 Libraries for specific purposes

- `log` (and `env_logger`, etc.); `tracing`.

- `regex` (and `lazy-regex`), `glob`, `tempfile`, `rand`, `either`, `void`.
- `ndarray`, `ndarray-linalg`, etc. Vectors, matrices, linear algebra.
- Cryptography: `ring`, `rustls`, Rust Crypto; see Sylvain Kerkour's 2021 writeup.
- `bstr`: Stringish methods on byte strings that are *hopefully* UTF-8 (but might not be).
- `bytemuck`: Reinterpret-casting of plain data.

## 14.3 `serde`

serde is a serialisation/deserialisation framework.

It defines a data model, and provides automatic translation of ordinary Rust `structs` to and from that model.

Ecosystem libraries provide concrete implementations for a wide variety of data formats, and some interesting data format metaprogramming tools.

The result is a superb capability to handle a wide variety of data marshalling problems. serde is especially good for ad-hoc data structures and structures whose definition is owned by a Rust project.

serde and its ecosystem are considerably better for many tasks than anything available in any other programming environment.

Generally, the resulting code is a fully monomorphised open-coded marshaller specialised for the specific data structure(s) and format(s), so performance is good but the code size can be very large.

## 14.4 Web tools and frameworks

Most Rust web tools are async.

Use `reqwest` or `ureq` for making HTTP requests.

Use `hyper` for a raw HTTP client or server, but consider using `reqwest` (client) or a web framework (server) instead.

Rust is well supplied with web frameworks, but it is hard to choose.

- I have been using Rocket for some years, But development is rather slow - there still isn't a final release of `rocket 0.5`. You should use the 0.5 preview.
- `axum` is from the Tokio team, but quite new.
- `actix-web` is very popular, but sometimes lacking attention to detail.
- `rouille` is sync. Yay! But I haven't tried it.
- You should perhaps also consider: `warp`.

## 14.5 Command line parsing: `clap`

If you are writing a command line program you should probably use `clap`. It allows declarative definition of command line options.

Unfortunately, `clap` has some problems.

- Serious problems handling options which override each other. There is a facility for this but it is not convenient and its algorithm is fundamentally wrong.
- General failure to follow (at least by default) well-established Unix option parsing conventions.

To illustrate: it is quite awkward even to provide a conventional pair of mutually-overriding `--foo` and `--no-foo` options.

In practice, using `clap` means accepting that one's program will have an imperfect and sometimes-balky command line syntax.

There are alternatives, notably `getopts`, `gumdrop` and `argparse`, but they are much less popular and less well maintained. I sometimes use `argparse` where I want a fine-tuned option parser, but it is quite odd and the docs are not great.

# Colophon

This is a guide to the Rust programming language. It was written by me, Ian Jackson, and I am responsible for the content and opinions.

Last revised and reviewed December 2022. (First edition September 2021.)

**Canonical location** The rendered document can be found here:

`https://www.chiark.greenend.org.uk/~ianmdlvl/rust-polyglot/`

There is also a single HTML page version and a PDF.

**Contributing** Contributions are very welcome, ideally via Issue or Merge Request:

`https://salsa.debian.org/iwj/rust-polyglot/`

I am happy to hear contrary views, especially about the recommendations about particular crates. However, I will make the final decision about the content of this guide.

**Format, building**: The document is in the intersection of mdbook and pandoc Markdown, in the `src/` directory. To format to HTML you will need to `cargo install mdbook` and run `make`, but untested contributions are welcome.

**Legal**: Please be sure to indicate your agreement with the declarations in the Developer Certificate of Origin, for example by adding a `Signed-off-by` line to your commits.

**Acknowledging your contribution**: If you would like to be acknowledged in the list below, please add your name there (as part of your MR).

**Privacy**: Note of course that since this guide is maintained in git, your contribution and any acknowledgement will be permanently recorded in the git history for reasons of traceability, auditability, transparency, and acknowledgement.

**Acknowledgements** Thanks for helpful review, comments and suggestions from: Simon Tatham, Mark Wooding, Daniel Silverstone, and others.

Thanks to Mark Wooding for the LaTeX/PDF arrangements.

```
  Copyright (c) 2021-2022 Ian Jackson and contributors

  Permission is hereby granted, free of charge, to any person
  obtaining a copy of this software and associated documentation files
  (the "Software"), to deal in the Software without restriction,
  including without limitation the rights to use, copy, modify, merge,
  publish, distribute, sublicense, and/or sell copies of the Software,
  and to permit persons to whom the Software is furnished to do so,
  subject to the following conditions:

  The above copyright notice and this permission notice shall be
  included in all copies or substantial portions of the Software.

  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
  EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
  MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
  NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
  BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
  ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
  CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
  SOFTWARE.

For a list of the contributors to this guide, see the git history.


Formalities
-----------

Individual files generally contain the following tag (or similar)
in the copyright notice, instead of the full licence grant text:
  SPDX-License-Identifier: MIT
As is conventional, this should be read as a licence grant.

Contributions to Rust for the Polyglot Programmer are accepted based
on the git commit Signed-off-by convention, by which the contributors'
certify their contributions according to the Developer Certificate of
Origin version 1.1 - see the file DEVELOPER-CERTIFICATE.

If you create a new file please be sure to add an appropriate licence
header, probably something like this:
// Copyright by contributors to Rust for the Polyglot Programmer
// SPDX-License-Identifier: MIT
// There is NO WARRANTY.
```

#### Developer Certificate of Origin (`DEVELOPER-CERTIFICATE`)

```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
1 Letterman Drive
```

Suite D4700
San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

---

formatted 2022-12-20 01:06:45 UTC

git commit `fa5def7537c413d3fea155c2d252c6cc2724a82f`