The Rust Language Tutorial

July 27, 2014

0.12.0-pre (2c50add48 2014-07-27 23:03:52 -0700)

Contents

The tutorial is undergoing a complete re-write as the Guide. Until it's ready, this tutorial is the right place to come to start learning Rust. Please submit improvements as pull requests, but understand that eventually it will be going away.

1 Introduction

Rust is a programming language with a focus on type safety, memory safety, concurrency and performance. It is intended for writing large-scale, high-performance software that is free from several classes of common errors. Rust has a sophisticated memory model that encourages efficient data structures and safe concurrency patterns, forbidding invalid memory accesses that would otherwise cause segmentation faults. It is statically typed and compiled ahead of time.

As a multi-paradigm language, Rust supports writing code in procedural, functional and object-oriented styles. Some of its pleasant high-level features include:

- **Type inference.** Type annotations on local variable declarations are optional.
- **Safe task-based concurrency.** Rust's lightweight tasks do not share memory, instead communicating through messages.

- **Higher-order functions.** Efficient and flexible closures provide iteration and other control structures
- **Pattern matching and algebraic data types.** Pattern matching on Rust's enumeration types (a more powerful version of C's enums, similar to algebraic data types in functional languages) is a compact and expressive way to encode program logic.
- **Polymorphism.** Rust has type-parametric functions and types, type classes and OO-style interfaces.

1.1 Scope

This is an introductory tutorial for the Rust programming language. It covers the fundamentals of the language, including the syntax, the type system and memory model, generics, and modules. Additional tutorials cover specific language features in greater depth.

This tutorial assumes that the reader is already familiar with one or more languages in the C family. Understanding of pointers and general memory management techniques will help.

1.2 Conventions

Throughout the tutorial, language keywords and identifiers defined in example code are displayed in `code font`.

Code snippets are indented, and also shown in a monospaced font. Not all snippets constitute whole programs. For brevity, we'll often show fragments of programs that don't compile on their own. To try them out, you might have to wrap them in `fn main() { ... }`, and make sure they don't contain references to names that aren't actually defined.

> *Warning:* Rust is a language under ongoing development. Notes about potential changes to the language, implementation deficiencies, and other caveats appear offset in blockquotes.

2 Getting started

There are two ways to install the Rust compiler: by building from source or by downloading prebuilt binaries or installers for your platform. The install page contains links to download binaries for both the nightly build and the most current Rust major release. For Windows and OS X, the install page provides links to native installers.

> *Note:* Windows users should read the detailed Getting started notes on the wiki. Even when using the binary installer, the Windows build requires a MinGW installation, the precise details of which are not discussed here.

For Linux and OS X, the install page provides links to binary tarballs. To install the Rust compiler from the from a binary tarball, download the binary package, extract it, and execute the `install.sh` script in the root directory of the package.

To build the Rust compiler from source, you will need to obtain the source through Git or by downloading the source package from the install page.

Since the Rust compiler is written in Rust, it must be built by a precompiled "snapshot" version of itself (made in an earlier state of development). The source build automatically fetches these snapshots from the Internet on our supported platforms.

Snapshot binaries are currently built and tested on several platforms:

- Windows (7, 8, Server 2008 R2), x86 only
- Linux (2.6.18 or later, various distributions), x86 and x86-64
- OSX 10.7 (Lion) or greater, x86 and x86-64

You may find that other platforms work, but these are our "tier 1" supported build environments that are most likely to work.

To build from source you will also need the following prerequisite packages:

- g++ 4.7 or clang++ 3.x
- python 2.6 or later (but not 3.x)
- perl 5.0 or later
- gnu make 3.81 or later
- curl

If you've fulfilled those prerequisites, something along these lines should work.

```
$ curl -O http://static.rust-lang.org/dist/rust-nightly.tar.gz
$ tar -xzf rust-nightly.tar.gz
$ cd rust-nightly
$ ./configure
$ make && make install
```

You may need to use `sudo make install` if you do not normally have permission to modify the destination directory. The install locations can be adjusted by

passing a `--prefix` argument to `configure`. Various other options are also supported: pass `--help` for more information on them.

When complete, `make install` will place several programs into `/usr/local/bin`: `rustc`, the Rust compiler, and `rustdoc`, the API-documentation tool.

2.1 Compiling your first program

Rust program files are, by convention, given the extension `.rs`. Say we have a file `hello.rs` containing this program:

```
fn main() {
    println!("hello?");
}
```

> *Note:* An identifier followed by an exclamation point, like `println!`, is a macro invocation. Macros are explained later; for now just remember to include the exclamation point.

If the Rust compiler was installed successfully, running `rustc hello.rs` will produce an executable called `hello` (or `hello.exe` on Windows) which, upon running, will likely do exactly what you expect.

The Rust compiler tries to provide useful information when it encounters an error. If you introduce an error into the program (for example, by changing `println!` to some nonexistent macro), and then compile it, you'll see an error message like this:

```
hello.rs:2:5: 2:24 error: macro undefined: 'print_with_unicorns'
hello.rs:2     print_with_unicorns!("hello?");
               ^~~~~~~~~~~~~~~~~~~
```

In its simplest form, a Rust program is a `.rs` file with some types and functions defined in it. If it has a `main` function, it can be compiled to an executable. Rust does not allow code that's not a declaration to appear at the top level of the file: all statements must live inside a function. Rust programs can also be compiled as libraries, and included in other programs, even ones not written in Rust.

2.2 Editing Rust code

There are vim highlighting and indentation scripts in the Rust source distribution under `src/etc/vim/`. There is an emacs mode under `src/etc/emacs/` called `rust-mode`, but do read the instructions included in that directory. In particular, if you are running emacs 24, then using emacs's internal package manager to

install `rust-mode` is the easiest way to keep it up to date. There is also a package for Sublime Text 2, available both standalone and through Sublime Package Control, and support for Kate under `src/etc/kate`.

A community-maintained list of available Rust tooling is on the wiki.

There is ctags support via `src/etc/ctags.rust`, but many other tools and editors are not yet supported. If you end up writing a Rust mode for your favorite editor, let us know so that we can link to it.

3 Syntax basics

Assuming you've programmed in any C-family language (C++, Java, JavaScript, C#, or PHP), Rust will feel familiar. Code is arranged in blocks delineated by curly braces; there are control structures for branching and looping, like the familiar `if` and `while`; function calls are written `myfunc(arg1, arg2)`; operators are written the same and mostly have the same precedence as in C; comments are again like C; module names are separated with double-colon (`::`) as with C++.

The main surface difference to be aware of is that the condition at the head of control structures like `if` and `while` does not require parentheses, while their bodies *must* be wrapped in braces. Single-statement, unbraced bodies are not allowed.

```
# mod universe { pub fn recalibrate() -> bool { true } }
fn main() {
    /* A simple loop */
    loop {
        // A tricky calculation
        if universe::recalibrate() {
            return;
        }
    }
}
```

The `let` keyword introduces a local variable. Variables are immutable by default. To introduce a local variable that you can re-assign later, use `let mut` instead.

```
let hi = "hi";
let mut count = 0i;

while count < 10 {
    println!("count is {}", count);
    count += 1;
}
```

Although Rust can almost always infer the types of local variables, you can specify a variable's type by following it in the `let` with a colon, then the type name. Static items, on the other hand, always require a type annotation.

```
static MONSTER_FACTOR: f64 = 57.8;
let monster_size = MONSTER_FACTOR * 10.0;
let monster_size: int = 50;
```

Local variables may shadow earlier declarations, as in the previous example: `monster_size` was first declared as a `f64`, and then a second `monster_size` was declared as an `int`. If you were to actually compile this example, though, the compiler would determine that the first `monster_size` is unused and issue a warning (because this situation is likely to indicate a programmer error). For occasions where unused variables are intentional, their names may be prefixed with an underscore to silence the warning, like `let _monster_size = 50;`.

Rust identifiers start with an alphabetic character or an underscore, and after that may contain any sequence of alphabetic characters, numbers, or underscores. The preferred style is to write function, variable, and module names with lowercase letters, using underscores where they help readability, while writing types in camel case.

```
let my_variable = 100i;
type MyType = int;     // primitive types are _not_ camel case
```

3.1 Expressions and semicolons

Though it isn't apparent in all code, there is a fundamental difference between Rust's syntax and predecessors like C. Many constructs that are statements in C are expressions in Rust, allowing code to be more concise. For example, you might write a piece of code like this:

```
# let item = "salad";
let price: f64;
if item == "salad" {
    price = 3.50;
} else if item == "muffin" {
    price = 2.25;
} else {
    price = 2.00;
}
```

But, in Rust, you don't have to repeat the name `price`:

```
# let item = "salad";
let price: f64 =
    if item == "salad" {
        3.50
    } else if item == "muffin" {
        2.25
    } else {
        2.00
    };
```

Both pieces of code are exactly equivalent: they assign a value to `price` depending on the condition that holds. Note that there are no semicolons in the blocks of the second snippet. This is important: the lack of a semicolon after the last statement in a braced block gives the whole block the value of that last expression.

Put another way, the semicolon in Rust *ignores the value of an expression.* Thus, if the branches of the `if` had looked like `{ 4; }`, the above example would simply assign `()` (unit or void) to `price`. But without the semicolon, each branch has a different value, and `price` gets the value of the branch that was taken.

In short, everything that's not a declaration (declarations are `let` for variables; `fn` for functions; and any top-level named items such as traits, enum types, and static items) is an expression, including function bodies.

```
fn is_four(x: int) -> bool {
   // No need for a return statement. The result of the expression
   // is used as the return value.
   x == 4
}
```

3.2 Primitive types and literals

There are general signed and unsigned integer types, `int` and `uint`, as well as 8-, 16-, 32-, and 64-bit variants, `i8`, `u16`, etc. Integers can be written in decimal (`144`), hexadecimal (`0x90`), octal (`0o70`), or binary (`0b10010000`) base. Each integral type has a corresponding literal suffix that can be used to indicate the type of a literal: `i` for `int`, `u` for `uint`, `i8` for the `i8` type.

In the absence of an integer literal suffix, Rust will infer the integer type based on type annotations and function signatures in the surrounding program. In the absence of any type information at all, Rust will report an error and request that the type be specified explicitly.

```
let a: int = 1;  // `a` is an `int`
let b = 10i;     // `b` is an `int`, due to the `i` suffix
```

```
let c = 100u;    // `c` is a `uint`
let d = 1000i32; // `d` is an `i32`
```

There are two floating-point types: `f32`, and `f64`. Floating-point numbers are written `0.0`, `1e6`, or `2.1e-4`. Like integers, floating-point literals are inferred to the correct type. Suffixes `f32`, and `f64` can be used to create literals of a specific type.

The keywords `true` and `false` produce literals of type `bool`.

Characters, the `char` type, are four-byte Unicode codepoints, whose literals are written between single quotes, as in `'x'`. Just like C, Rust understands a number of character escapes, using the backslash character, such as `\n`, `\r`, and `\t`. String literals, written between double quotes, allow the same escape sequences, and do no other processing, unlike languages such as PHP or shell.

On the other hand, raw string literals do not process any escape sequences. They are written as `r##"blah"##`, with a matching number of zero or more `#` before the opening and after the closing quote, and can contain any sequence of characters except their closing delimiter. More on strings later.

The unit type, written `()`, has a single value, also written `()`.

3.3 Operators

Rust's set of operators contains very few surprises. Arithmetic is done with `*`, `/`, `%`, `+`, and `-` (multiply, quotient, remainder, add, and subtract). `-` is also a unary prefix operator that negates numbers. As in C, the bitwise operators `>>`, `<<`, `&`, `|`, and `^` are also supported.

Note that, if applied to an integer value, `!` flips all the bits (bitwise NOT, like `~` in C).

The comparison operators are the traditional `==`, `!=`, `<`, `>`, `<=`, and `>=`. Short-circuiting (lazy) boolean operators are written `&&` (and) and `||` (or).

For compile-time type casting, Rust uses the binary `as` operator. It takes an expression on the left side and a type on the right side and will, if a meaningful conversion exists, convert the result of the expression to the given type. Generally, `as` is only used with the primitive numeric types or pointers, and is not overloadable. `transmute` can be used for unsafe C-like casting of same-sized types.

```
let x: f64 = 4.0;
let y: uint = x as uint;
assert!(y == 4u);
```

3.4 Syntax extensions

Syntax extensions are special forms that are not built into the language, but are instead provided by the libraries. To make it clear to the reader when a name refers to a syntax extension, the names of all syntax extensions end with `!`. The standard library defines a few syntax extensions, the most useful of which is `format!`, a `sprintf`-like text formatter that you will often see in examples, and its related family of macros: `print!`, `println!`, and `write!`.

`format!` draws syntax from Python, but contains many of the same principles that printf has. Unlike printf, `format!` will give you a compile-time error when the types of the directives don't match the types of the arguments.

```
// `{}` will print the "default format" of a type
println!("{} is {}", "the answer", 43i);
```

```
extern crate debug;

# fn main() {
# let mystery_object = ();
// `{:?}` will conveniently print any type,
// but requires the `debug` crate to be linked in
println!("what is this thing: {:?}", mystery_object);
# }
```

You can define your own syntax extensions with the macro system. For details, see the macro tutorial. Note that macro definition is currently considered an unstable feature.

4 Control structures

4.1 Conditionals

We've seen `if` expressions a few times already. To recap, braces are compulsory, an `if` can have an optional `else` clause, and multiple `if`/`else` constructs can be chained together:

```
if false {
    println!("that's odd");
} else if true {
    println!("right");
} else {
    println!("neither true nor false");
}
```

The condition given to an `if` construct *must* be of type `bool` (no implicit conversion happens). If the arms are blocks that have a value, this value must be of the same type for every arm in which control reaches the end of the block:

```
fn signum(x: int) -> int {
    if x < 0 { -1 }
    else if x > 0 { 1 }
    else { 0 }
}
```

4.2 Pattern matching

Rust's `match` construct is a generalized, cleaned-up version of C's `switch` construct. You provide it with a value and a number of *arms*, each labeled with a pattern, and the code compares the value against each pattern in order until one matches. The matching pattern executes its corresponding arm.

```
let my_number = 1i;
match my_number {
  0     => println!("zero"),
  1 | 2 => println!("one or two"),
  3..10 => println!("three to ten"),
  _     => println!("something else")
}
```

Unlike in C, there is no "falling through" between arms: only one arm executes, and it doesn't have to explicitly `break` out of the construct when it is finished.

A `match` arm consists of a *pattern*, then a fat arrow `=>`, followed by an *action* (expression). Each case is separated by commas. It is often convenient to use a block expression for each case, in which case the commas are optional as shown below. Literals are valid patterns and match only their own value. A single arm may match multiple different patterns by combining them with the pipe operator (`|`), so long as every pattern binds the same set of variables (see "destructuring" below). Ranges of numeric literal patterns can be expressed with two dots, as in `M..N`. The underscore (`_`) is a wildcard pattern that matches any single value. (`..`) is a different wildcard that can match one or more fields in an `enum` variant.

```
# let my_number = 1i;
match my_number {
  0 => { println!("zero") }
  _ => { println!("something else") }
}
```

`match` constructs must be *exhaustive*: they must have an arm covering every possible case. For example, the typechecker would reject the previous example if the arm with the wildcard pattern was omitted.

A powerful application of pattern matching is *destructuring*: matching in order to bind names to the contents of data types.

> *Note:* The following code makes use of tuples (`(f64, f64)`) which are explained in section 5.3. For now you can think of tuples as a list of items.

```
use std::f64;
fn angle(vector: (f64, f64)) -> f64 {
    let pi = f64::consts::PI;
    match vector {
      (0.0, y) if y < 0.0 => 1.5 * pi,
      (0.0, _) => 0.5 * pi,
      (x, y) => (y / x).atan()
    }
}
```

A variable name in a pattern matches any value, *and* binds that name to the value of the matched value inside of the arm's action. Thus, `(0.0, y)` matches any tuple whose first element is zero, and binds `y` to the second element. `(x, y)` matches any two-element tuple, and binds both elements to variables. `(0.0,_)` matches any tuple whose first element is zero and does not bind anything to the second element.

A subpattern can also be bound to a variable, using `variable @ pattern`. For example:

```
# let age = 23i;
match age {
    a @ 0..20 => println!("{} years old", a),
    _ => println!("older than 21")
}
```

Any `match` arm can have a guard clause (written `if EXPR`), called a *pattern guard*, which is an expression of type `bool` that determines, after the pattern is found to match, whether the arm is taken or not. The variables bound by the pattern are in scope in this guard expression. The first arm in the `angle` example shows an example of a pattern guard.

You've already seen simple `let` bindings, but `let` is a little fancier than you've been led to believe. It, too, supports destructuring patterns. For example, you can write this to extract the fields from a tuple, introducing two variables at once: `a` and `b`.

```
# fn get_tuple_of_two_ints() -> (int, int) { (1, 1) }
let (a, b) = get_tuple_of_two_ints();
```

Let bindings only work with *irrefutable* patterns: that is, patterns that can never fail to match. This excludes `let` from matching literals and most `enum` variants as binding patterns, since most such patterns are not irrefutable. For example, this will not compile:

```
{ignore} let (a, 2) = (1, 2);
```

4.3 Loops

`while` denotes a loop that iterates as long as its given condition (which must have type `bool`) evaluates to `true`. Inside a loop, the keyword `break` aborts the loop, and `continue` aborts the current iteration and continues with the next.

```
let mut cake_amount = 8i;
while cake_amount > 0 {
    cake_amount -= 1;
}
```

`loop` denotes an infinite loop, and is the preferred way of writing `while true`:

```
let mut x = 5u;
loop {
    x += x - 3;
    if x % 5 == 0 { break; }
    println!("{}", x);
}
```

This code prints out a weird sequence of numbers and stops as soon as it finds one that can be divided by five.

There is also a for-loop that can be used to iterate over a range of numbers:

```
for n in range(0u, 5) {
    println!("{}", n);
}
```

The snippet above prints integer numbers under 5 starting at 0.

More generally, a for loop works with anything implementing the `Iterator` trait. Data structures can provide one or more methods that return iterators over their contents. For example, strings support iteration over their contents in various ways:

```
let s = "Hello";
for c in s.chars() {
    println!("{}", c);
}
```

The snippet above prints the characters in "Hello" vertically, adding a new line after each character.

5 Data structures

5.1 Structs

Rust struct types must be declared before they are used using the `struct` syntax: `struct Name { field1: T1, field2: T2 [, ...] }`, where `T1`, `T2`, ... denote types. To construct a struct, use the same syntax, but leave off the `struct`: for example: `Point { x: 1.0, y: 2.0 }`.

Structs are quite similar to C structs and are even laid out the same way in memory (so you can read from a Rust struct in C, and vice-versa). Use the dot operator to access struct fields, as in `mypoint.x`.

```
struct Point {
    x: f64,
    y: f64
}
```

Structs have "inherited mutability", which means that any field of a struct may be mutable, if the struct is in a mutable slot.

With a value (say, `mypoint`) of such a type in a mutable location, you can do `mypoint.y += 1.0`. But in an immutable location, such an assignment to a struct without inherited mutability would result in a type error.

```
# struct Point { x: f64, y: f64 }
let mut mypoint = Point { x: 1.0, y: 1.0 };
let origin = Point { x: 0.0, y: 0.0 };

mypoint.y += 1.0; // `mypoint` is mutable, and its fields as well
origin.y += 1.0; // ERROR: assigning to immutable field
```

`match` patterns destructure structs. The basic syntax is `Name { fieldname: pattern, ... }`:

```
# struct Point { x: f64, y: f64 }
# let mypoint = Point { x: 0.0, y: 0.0 };
match mypoint {
    Point { x: 0.0, y: yy } => println!("{}", yy),
    Point { x: xx,  y: yy } => println!("{} {}", xx, yy)
}
```

In general, the field names of a struct do not have to appear in the same order they appear in the type. When you are not interested in all the fields of a struct, a struct pattern may end with `, ..` (as in `Name { field1, .. }`) to indicate that you're ignoring all other fields. Additionally, struct fields have a shorthand matching form that simply reuses the field name as the binding name.

```
# struct Point { x: f64, y: f64 }
# let mypoint = Point { x: 0.0, y: 0.0 };
match mypoint {
    Point { x, .. } => println!("{}", x)
}
```

5.2 Enums

Enums are datatypes with several alternate representations. A simple `enum` defines one or more constants, all of which have the same type:

```
enum Direction {
    North,
    East,
    South,
    West
}
```

Each variant of this enum has a unique and constant integral discriminator value. If no explicit discriminator is specified for a variant, the value defaults to the value of the previous variant plus one. If the first variant does not have a discriminator, it defaults to 0. For example, the value of `North` is 0, `East` is 1, `South` is 2, and `West` is 3.

When an enum has simple integer discriminators, you can apply the `as` cast operator to convert a variant to its discriminator value as an `int`:

```
# enum Direction { North, East, South, West }
println!( "North => {}", North as int );
```

It is possible to set the discriminator values to chosen constant values:

```
enum Color {
  Red = 0xff0000,
  Green = 0x00ff00,
  Blue = 0x0000ff
}
```

Variants do not have to be simple values; they may be more complex:

```
# struct Point { x: f64, y: f64 }
enum Shape {
    Circle(Point, f64),
    Rectangle(Point, Point)
}
```

A value of this type is either a `Circle`, in which case it contains a `Point` struct and a f64, or a `Rectangle`, in which case it contains two `Point` structs. The run-time representation of such a value includes an identifier of the actual form that it holds, much like the "tagged union" pattern in C, but with better static guarantees.

This declaration defines a type `Shape` that can refer to such shapes, and two functions, `Circle` and `Rectangle`, which can be used to construct values of the type.

To create a new `Circle`, write:

```
# struct Point { x: f64, y: f64 }
# enum Shape { Circle(Point, f64), Rectangle(Point, Point) }
let circle = Circle(Point { x: 0.0, y: 0.0 }, 10.0);
```

All of these variant constructors may be used as patterns. The only way to access the contents of an enum instance is the destructuring of a match. For example:

```
use std::f64;

# struct Point {x: f64, y: f64}
# enum Shape { Circle(Point, f64), Rectangle(Point, Point) }
fn area(sh: Shape) -> f64 {
    match sh {
        Circle(_, size) => f64::consts::PI * size * size,
        Rectangle(Point { x, y }, Point { x: x2, y: y2 }) => (x2 - x) * (y2 - y)
    }
}

let rect = Rectangle(Point { x: 0.0, y: 0.0 }, Point { x: 2.0, y: 2.0 });
println!("area: {}", area(rect));
```

Use a lone `_` to ignore an individual field. Ignore all fields of a variant like: `Circle(..)`. Nullary enum patterns are written without parentheses:

```
# struct Point { x: f64, y: f64 }
# enum Direction { North, East, South, West }
fn point_from_direction(dir: Direction) -> Point {
    match dir {
        North => Point { x:  0.0, y:  1.0 },
        East  => Point { x:  1.0, y:  0.0 },
        South => Point { x:  0.0, y: -1.0 },
        West  => Point { x: -1.0, y:  0.0 }
    }
}
```

Enum variants may also be structs. For example:

```
#![feature(struct_variant)]
use std::f64;

# struct Point { x: f64, y: f64 }
# fn square(x: f64) -> f64 { x * x }
enum Shape {
    Circle { center: Point, radius: f64 },
    Rectangle { top_left: Point, bottom_right: Point }
}
fn area(sh: Shape) -> f64 {
    match sh {
        Circle { radius: radius, .. } => f64::consts::PI * square(radius),
        Rectangle { top_left: top_left, bottom_right: bottom_right } => {
            (bottom_right.x - top_left.x) * (top_left.y - bottom_right.y)
        }
    }
}

fn main() {
    let rect = Rectangle {
        top_left: Point { x: 0.0, y: 0.0 },
        bottom_right: Point { x: 2.0, y: -2.0 }
    };
    println!("area: {}", area(rect));
}
```

> *Note:* This feature of the compiler is currently gated behind the `#[feature(struct_variant)]` directive. More about these directives can be found in the manual.

5.3 Tuples

Tuples in Rust behave exactly like structs, except that their fields do not have names. Thus, you cannot access their fields with dot notation. Tuples can have any arity (number of elements) except for 0 (though you may consider unit, `()`, as the empty tuple if you like).

```
let mytup: (int, int, f64) = (10, 20, 30.0);
match mytup {
  (a, b, c) => println!("{}", a + b + (c as int))
}
```

5.4 Tuple structs

Rust also has *tuple structs*, which behave like both structs and tuples, except that, unlike tuples, tuple structs have names (so `Foo(1, 2)` has a different type from `Bar(1, 2)`), and tuple structs' *fields* do not have names.

For example:

```
struct MyTup(int, int, f64);
let mytup: MyTup = MyTup(10, 20, 30.0);
match mytup {
  MyTup(a, b, c) => println!("{}", a + b + (c as int))
}
```

There is a special case for tuple structs with a single field, which are sometimes called "newtypes" (after Haskell's "newtype" feature). These are used to define new types in such a way that the new name is not just a synonym for an existing type but is rather its own distinct type.

```
struct GizmoId(int);
```

Types like this can be useful to differentiate between data that have the same underlying type but must be used in different ways.

```
struct Inches(int);
struct Centimeters(int);
```

The above definitions allow for a simple way for programs to avoid confusing numbers that correspond to different units. Their integer values can be extracted with pattern matching:

```
# struct Inches(int);
let length_with_unit = Inches(10);
let Inches(integer_length) = length_with_unit;
println!("length is {} inches", integer_length);
```

6 Functions

We've already seen several function definitions. Like all other static declarations, such as `type`, functions can be declared both at the top level and inside other functions (or in modules, which we'll come back to later). The `fn` keyword introduces a function. A function has an argument list, which is a parenthesized list of `name: type` pairs separated by commas. An arrow `->` separates the argument list and the function's return type.

```
fn line(a: int, b: int, x: int) -> int {
    return a * x + b;
}
```

The `return` keyword immediately returns from the body of a function. It is optionally followed by an expression to return. A function can also return a value by having its top-level block produce an expression.

```
fn line(a: int, b: int, x: int) -> int {
    a * x + b
}
```

It's better Rust style to write a return value this way instead of writing an explicit `return`. The utility of `return` comes in when returning early from a function. Functions that do not return a value are said to return unit, `()`, and both the return type and the return value may be omitted from the definition. The following two functions are equivalent.

```
fn do_nothing_the_hard_way() -> () { return (); }

fn do_nothing_the_easy_way() { }
```

Ending the function with a semicolon like so is equivalent to returning `()`.

```
fn line(a: int, b: int, x: int) -> int { a * x + b  }
fn oops(a: int, b: int, x: int) -> ()  { a * x + b; }

assert!(8 == line(5, 3, 1));
assert!(() == oops(5, 3, 1));
```

As with `match` expressions and `let` bindings, function arguments support pattern destructuring. Like `let`, argument patterns must be irrefutable, as in this example that unpacks the first value from a tuple and returns it.

```
fn first((value, _): (int, f64)) -> int { value }
```

7 Destructors

A *destructor* is a function responsible for cleaning up the resources used by an object when it is no longer accessible. Destructors can be defined to handle the release of resources like files, sockets and heap memory.

Objects are never accessible after their destructor has been called, so no dynamic failures are possible from accessing freed resources. When a task fails, destructors of all objects in the task are called.

The `box` operator performs memory allocation on the heap:

```
{
    // an integer allocated on the heap
    let y = box 10i;
}
// the destructor frees the heap memory as soon as `y` goes out of scope
```

Rust includes syntax for heap memory allocation in the language since it's commonly used, but the same semantics can be implemented by a type with a custom destructor.

8 Ownership

Rust formalizes the concept of object ownership to delegate management of an object's lifetime to either a variable or a task-local garbage collector. An object's owner is responsible for managing the lifetime of the object by calling the destructor, and the owner determines whether the object is mutable.

Ownership is recursive, so mutability is inherited recursively and a destructor destroys the contained tree of owned objects. Variables are top-level owners and destroy the contained object when they go out of scope.

```
// the struct owns the objects contained in the `x` and `y` fields
struct Foo { x: int, y: Box<int> }

{
    // `a` is the owner of the struct, and thus the owner of the struct's fields
    let a = Foo { x: 5, y: box 10 };
}
// when `a` goes out of scope, the destructor for the `Box<int>` in the struct's
// field is called

// `b` is mutable, and the mutability is inherited by the objects it owns
let mut b = Foo { x: 5, y: box 10 };
b.x = 10;
```

If an object doesn't contain any non-`Send` types, it consists of a single ownership tree and is itself given the `Send` trait which allows it to be sent between tasks. Custom destructors can only be implemented directly on types that are `Send`, but non-`Send` types can still *contain* types with custom destructors. Example of types which are not `Send` are `Gc<T>` and `Rc<T>`, the shared-ownership types.

> *Note:* See a later chapter for a discussion about `Gc<T>` and `Rc<T>`, and the chapter about traits for a discussion about `Send`.

9 Implementing a linked list

An `enum` is a natural fit for describing a linked list, because it can express a `List` type as being *either* the end of the list (`Nil`) or another node (`Cons`). The full definition of the `Cons` variant will require some thought.

```
enum List {
    Cons(...), // an incomplete definition of the next element in a List
    Nil        // the end of a List
}
```

The obvious approach is to define `Cons` as containing an element in the list along with the next `List` node. However, this will generate a compiler error.

```
// error: illegal recursive enum type; wrap the inner value in a box to make it
// representable
enum List {
    Cons(u32, List), // an element (`u32`) and the next node in the list
    Nil
}
```

This error message is related to Rust's precise control over memory layout, and solving it will require introducing the concept of *boxing*.

9.1 Boxes

A value in Rust is stored directly inside the owner. If a `struct` contains four `u32` fields, it will be four times as large as a single `u32`.

```
use std::mem::size_of; // bring `size_of` into the current scope, for convenience

struct Foo {
    a: u32,
```

```
    b: u32,
    c: u32,
    d: u32
}

assert_eq!(size_of::<Foo>(), size_of::<u32>() * 4);

struct Bar {
    a: Foo,
    b: Foo,
    c: Foo,
    d: Foo
}

assert_eq!(size_of::<Bar>(), size_of::<u32>() * 16);
```

Our previous attempt at defining the `List` type included an `u32` and a `List` directly inside `Cons`, making it at least as big as the sum of both types. The type was invalid because the size was infinite!

An *owned box* (`Box`, located in the `std::owned` module) uses a dynamic memory allocation to provide the invariant of always being the size of a pointer, regardless of the contained type. This can be leveraged to create a valid `List` definition:

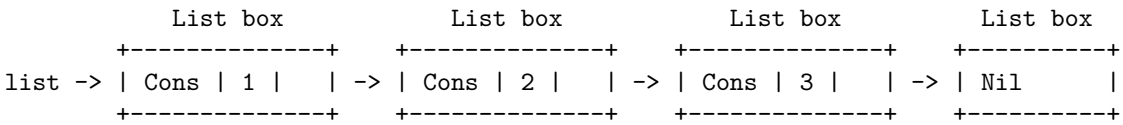
```
enum List {
    Cons(u32, Box<List>),
    Nil
}
```

Defining a recursive data structure like this is the canonical example of an owned box. Much like an unboxed value, an owned box has a single owner and is therefore limited to expressing a tree-like data structure.

Consider an instance of our `List` type:

```
# enum List {
#     Cons(u32, Box<List>),
#     Nil
# }
let list = Cons(1, box Cons(2, box Cons(3, box Nil)));
```

It represents an owned tree of values, inheriting mutability down the tree and being destroyed along with the owner. Since the `list` variable above is immutable, the whole list is immutable. The memory allocation itself is the box, while the owner holds onto a pointer to it:



> *Note:* the above diagram shows the logical contents of the enum. The actual memory layout of the enum may vary. For example, for the `List` enum shown above, Rust guarantees that there will be no enum tag field in the actual structure. See the language reference for more details.

An owned box is a common example of a type with a destructor. The allocated memory is cleaned up when the box is destroyed.

9.2 Move semantics

Rust uses a shallow copy for parameter passing, assignment and returning from functions. Passing around the `List` will copy only as deep as the pointer to the box rather than doing an implicit heap allocation.

```
# enum List {
#     Cons(u32, Box<List>),
#     Nil
# }
let xs = Cons(1, box Cons(2, box Cons(3, box Nil)));
let ys = xs; // copies `Cons(u32, pointer)` shallowly
```

Rust will consider a shallow copy of a type with a destructor like `List` to *move ownership* of the value. After a value has been moved, the source location cannot be used unless it is reinitialized.

```
# enum List {
#     Cons(u32, Box<List>),
#     Nil
# }
let mut xs = Nil;
let ys = xs;

// attempting to use `xs` will result in an error here

xs = Nil;

// `xs` can be used again
```

A destructor call will only occur for a variable that has not been moved from, as it is only called a single time.

Avoiding a move can be done with the library-defined `clone` method:

```
let x = box 5i;
let y = x.clone(); // `y` is a newly allocated box
let z = x; // no new memory allocated, `x` can no longer be used
```

The `clone` method is provided by the `Clone` trait, and can be derived for our `List` type. Traits will be explained in detail later.

```
#[deriving(Clone)]
enum List {
    Cons(u32, Box<List>),
    Nil
}

let x = Cons(5, box Nil);
let y = x.clone();

// `x` can still be used!

let z = x;

// and now, it can no longer be used since it has been moved
```

The mutability of a value may be changed by moving it to a new owner:

```
let r = box 13i;
let mut s = r; // box becomes mutable
*s += 1;
let t = s; // box becomes immutable
```

A simple way to define a function prepending to the `List` type is to take advantage of moves:

```
enum List {
    Cons(u32, Box<List>),
    Nil
}

fn prepend(xs: List, value: u32) -> List {
    Cons(value, box xs)
```

```
}

let mut xs = Nil;
xs = prepend(xs, 1);
xs = prepend(xs, 2);
xs = prepend(xs, 3);
```

However, this is not a very flexible definition of `prepend` as it requires ownership of a list to be passed in rather than just mutating it in-place.

9.3 References

The obvious signature for a `List` equality comparison is the following:

```
fn eq(xs: List, ys: List) -> bool { /* ... */ }
```

However, this will cause both lists to be moved into the function. Ownership isn't required to compare the lists, so the function should take *references* (&T) instead.

```
fn eq(xs: &List, ys: &List) -> bool { /* ... */ }
```

A reference is a *non-owning* view of a value. A reference can be obtained with the `&` (address-of) operator. It can be dereferenced by using the `*` operator. In a pattern, such as `match` expression branches, the `ref` keyword can be used to bind to a variable name by-reference rather than by-value. A recursive definition of equality using references is as follows:

```
# enum List {
#     Cons(u32, Box<List>),
#     Nil
# }
fn eq(xs: &List, ys: &List) -> bool {
    // Match on the next node in both lists.
    match (xs, ys) {
        // If we have reached the end of both lists, they are equal.
        (&Nil, &Nil) => true,
        // If the current elements of both lists are equal, keep going.
        (&Cons(x, box ref next_xs), &Cons(y, box ref next_ys))
                if x == y => eq(next_xs, next_ys),
        // If the current elements are not equal, the lists are not equal.
        _ => false
    }
```

```
}

let xs = Cons(5, box Cons(10, box Nil));
let ys = Cons(5, box Cons(10, box Nil));
assert!(eq(&xs, &ys));
```

> *Note:* Rust doesn't guarantee tail-call optimization, but LLVM is able to handle a simple case like this with optimizations enabled.

9.4 Lists of other types

Our `List` type is currently always a list of 32-bit unsigned integers. By leveraging Rust's support for generics, it can be extended to work for any element type.

The `u32` in the previous definition can be substituted with a type parameter:

> *Note:* The following code introduces generics, which are explained in a dedicated section.

```
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil
}
```

The old `List` of `u32` is now available as `List<u32>`. The `prepend` definition has to be updated too:

```
# enum List<T> {
#     Cons(T, Box<List<T>>),
#     Nil
# }
fn prepend<T>(xs: List<T>, value: T) -> List<T> {
    Cons(value, box xs)
}
```

Generic functions and types like this are equivalent to defining specialized versions for each set of type parameters.

Using the generic `List<T>` works much like before, thanks to type inference:

```
# enum List<T> {
#     Cons(T, Box<List<T>>),
#     Nil
# }
```

```
# fn prepend<T>(xs: List<T>, value: T) -> List<T> {
#     Cons(value, box xs)
# }
let mut xs = Nil; // Unknown type! This is a `List<T>`, but `T` can be anything.
xs = prepend(xs, 10i); // Here the compiler infers `xs`'s type as `List<int>`.
xs = prepend(xs, 15i);
xs = prepend(xs, 20i);
```

The code sample above demonstrates type inference making most type annotations optional. It is equivalent to the following type-annotated code:

```
# enum List<T> {
#     Cons(T, Box<List<T>>),
#     Nil
# }
# fn prepend<T>(xs: List<T>, value: T) -> List<T> {
#     Cons(value, box xs)
# }
let mut xs: List<int> = Nil::<int>;
xs = prepend::<int>(xs, 10);
xs = prepend::<int>(xs, 15);
xs = prepend::<int>(xs, 20);
```

In declarations, the language uses `Type<T, U, V>` to describe a list of type parameters, but expressions use `identifier::<T, U, V>`, to disambiguate the `<` operator.

9.5 Defining list equality with generics

Generic functions are type-checked from the definition, so any necessary properties of the type must be specified up-front. Our previous definition of list equality relied on the element type having the `==` operator available, and took advantage of the lack of a destructor on `u32` to copy it without a move of ownership.

We can add a *trait bound* on the `PartialEq` trait to require that the type implement the `==` operator. Two more `ref` annotations need to be added to avoid attempting to move out the element types:

```
# enum List<T> {
#     Cons(T, Box<List<T>>),
#     Nil
# }
fn eq<T: PartialEq>(xs: &List<T>, ys: &List<T>) -> bool {
    // Match on the next node in both lists.
```

```
    match (xs, ys) {
        // If we have reached the end of both lists, they are equal.
        (&Nil, &Nil) => true,
        // If the current elements of both lists are equal, keep going.
        (&Cons(ref x, box ref next_xs), &Cons(ref y, box ref next_ys))
                if x == y => eq(next_xs, next_ys),
        // If the current elements are not equal, the lists are not equal.
        _ => false
    }
}

let xs = Cons('c', box Cons('a', box Cons('t', box Nil)));
let ys = Cons('c', box Cons('a', box Cons('t', box Nil)));
assert!(eq(&xs, &ys));
```

This would be a good opportunity to implement the `PartialEq` trait for our list type, making the `==` and `!=` operators available. We'll need to provide an `impl` for the `PartialEq` trait and a definition of the `eq` method. In a method, the `self` parameter refers to an instance of the type we're implementing on.

```
# enum List<T> {
#     Cons(T, Box<List<T>>),
#     Nil
# }
impl<T: PartialEq> PartialEq for List<T> {
    fn eq(&self, ys: &List<T>) -> bool {
        // Match on the next node in both lists.
        match (self, ys) {
            // If we have reached the end of both lists, they are equal.
            (&Nil, &Nil) => true,
            // If the current elements of both lists are equal, keep going.
            (&Cons(ref x, box ref next_xs), &Cons(ref y, box ref next_ys))
                    if x == y => next_xs == next_ys,
            // If the current elements are not equal, the lists are not equal.
            _ => false
        }
    }
}

let xs = Cons(5i, box Cons(10i, box Nil));
let ys = Cons(5i, box Cons(10i, box Nil));
// The methods below are part of the PartialEq trait,
// which we implemented on our linked list.
assert!(xs.eq(&ys));
assert!(!xs.ne(&ys));
```

```
// The PartialEq trait also allows us to use the shorthand infix operators.
assert!(xs == ys);    // `xs == ys` is short for `xs.eq(&ys)`
assert!(!(xs != ys)); // `xs != ys` is short for `xs.ne(&ys)`
```

10 More on boxes

The most common use case for owned boxes is creating recursive data structures like a binary search tree. Rust's trait-based generics system (covered later in the tutorial) is usually used for static dispatch, but also provides dynamic dispatch via boxing. Values of different types may have different sizes, but a box is able to *erase* the difference via the layer of indirection they provide.

In uncommon cases, the indirection can provide a performance gain or memory reduction by making values smaller. However, unboxed values should almost always be preferred when they are usable.

Note that returning large unboxed values via boxes is unnecessary. A large value is returned via a hidden output parameter, and the decision on where to place the return value should be left to the caller:

```
fn foo() -> (u64, u64, u64, u64, u64, u64) {
    (5, 5, 5, 5, 5, 5)
}

let x = box foo(); // allocates a box, and writes the integers directly to it
```

Beyond the properties granted by the size, an owned box behaves as a regular value by inheriting the mutability and lifetime of the owner:

```
let x = 5i; // immutable
let mut y = 5i; // mutable
y += 2;

let x = box 5i; // immutable
let mut y = box 5i; // mutable
*y += 2; // the `*` operator is needed to access the contained value
```

11 References

In contrast with owned boxes, where the holder of an owned box is the owner of the pointed-to memory, references never imply ownership - they are "borrowed".

You can borrow a reference to any object, and the compiler verifies that it cannot outlive the lifetime of the object.

As an example, consider a simple struct type, `Point`:

```
struct Point {
    x: f64,
    y: f64
}
```

We can use this simple definition to allocate points in many different ways. For example, in this code, each of these local variables contains a point, but allocated in a different location:

```
# struct Point { x: f64, y: f64 }
let on_the_stack :     Point  =     Point { x: 3.0, y: 4.0 };
let on_the_heap  : Box<Point> = box Point { x: 7.0, y: 9.0 };
```

Suppose we want to write a procedure that computes the distance between any two points, no matter where they are stored. One option is to define a function that takes two arguments of type point—that is, it takes the points by value. But this will cause the points to be copied when we call the function. For points, this is probably not so bad, but often copies are expensive. So we'd like to define a function that takes the points by pointer. We can use references to do this:

```
# struct Point { x: f64, y: f64 }
fn compute_distance(p1: &Point, p2: &Point) -> f64 {
    let x_d = p1.x - p2.x;
    let y_d = p1.y - p2.y;
    (x_d * x_d + y_d * y_d).sqrt()
}
```

Now we can call `compute_distance()` in various ways:

```
# struct Point{ x: f64, y: f64 };
# let on_the_stack :     Point  =     Point { x: 3.0, y: 4.0 };
# let on_the_heap  : Box<Point> = box Point { x: 7.0, y: 9.0 };
# fn compute_distance(p1: &Point, p2: &Point) -> f64 { 0.0 }
compute_distance(&on_the_stack, &*on_the_heap);
```

Here the `&` operator is used to take the address of the variable `on_the_stack`; this is because `on_the_stack` has the type `Point` (that is, a struct value) and we have to take its address to get a reference. We also call this *borrowing* the

local variable `on_the_stack`, because we are creating an alias: that is, another route to the same data.

Likewise, in the case of `owned_box`, the `&` operator is used in conjunction with the `*` operator to take a reference to the contents of the box.

Whenever a value is borrowed, there are some limitations on what you can do with the original. For example, if the contents of a variable have been lent out, you cannot send that variable to another task, nor will you be permitted to take actions that might cause the borrowed value to be freed or to change its type. This rule should make intuitive sense: you must wait for a borrowed value to be returned (that is, for the reference to go out of scope) before you can make full use of it again.

For a more in-depth explanation of references and lifetimes, read the references and lifetimes guide.

11.1 Freezing

Lending an &-pointer to an object freezes the pointed-to object and prevents mutation—even if the object was declared as `mut`. `Freeze` objects have freezing enforced statically at compile-time. An example of a non-`Freeze` type is `RefCell<T>`.

```
let mut x = 5i;
{
    let y = &x; // `x` is now frozen. It cannot be modified or re-assigned.
}
// `x` is now unfrozen again
# x = 3;
```

12 Dereferencing pointers

Rust uses the unary star operator (`*`) to access the contents of a box or pointer, similarly to C.

```
let owned = box 10i;
let borrowed = &20i;

let sum = *owned + *borrowed;
```

Dereferenced mutable pointers may appear on the left hand side of assignments. Such an assignment modifies the value that the pointer points to.

```
let mut owned = box 10i;

let mut value = 20i;
let borrowed = &mut value;

*owned = *borrowed + 100;
*borrowed = *owned + 1000;
```

Pointers have high operator precedence, but lower precedence than the dot operator used for field and method access. This precedence order can sometimes make code awkward and parenthesis-filled.

```
# struct Point { x: f64, y: f64 }
# enum Shape { Rectangle(Point, Point) }
# impl Shape { fn area(&self) -> int { 0 } }
let start = box Point { x: 10.0, y: 20.0 };
let end = box Point { x: (*start).x + 100.0, y: (*start).y + 100.0 };
let rect = &Rectangle(*start, *end);
let area = (*rect).area();
```

To combat this ugliness the dot operator applies *automatic pointer dereferencing* to the receiver (the value on the left-hand side of the dot), so in most cases, explicitly dereferencing the receiver is not necessary.

```
# struct Point { x: f64, y: f64 }
# enum Shape { Rectangle(Point, Point) }
# impl Shape { fn area(&self) -> int { 0 } }
let start = box Point { x: 10.0, y: 20.0 };
let end = box Point { x: start.x + 100.0, y: start.y + 100.0 };
let rect = &Rectangle(*start, *end);
let area = rect.area();
```

You can write an expression that dereferences any number of pointers automatically. For example, if you feel inclined, you could write something silly like

```
# struct Point { x: f64, y: f64 }
let point = &box Point { x: 10.0, y: 20.0 };
println!("{:f}", point.x);
```

The indexing operator (`[]`) also auto-dereferences.

13 Vectors and strings

A vector is a contiguous block of memory containing zero or more values of the same type. Rust also supports vector reference types, called slices, which are a view into a block of memory represented as a pointer and a length.

Strings are represented as vectors of `u8`, with the guarantee of containing a valid UTF-8 sequence.

Fixed-size vectors are an unboxed block of memory, with the element length as part of the type. A fixed-size vector owns the elements it contains, so the elements are mutable if the vector is mutable. Fixed-size strings do not exist.

```
// A fixed-size vector
let numbers = [1i, 2, 3];
let more_numbers = numbers;

// The type of a fixed-size vector is written as `[Type, ..length]`
let five_zeroes: [int, ..5] = [0, ..5];
```

A unique vector is dynamically sized, and has a destructor to clean up allocated memory on the heap. A unique vector owns the elements it contains, so the elements are mutable if the vector is mutable.

```
use std::string::String;

// A dynamically sized vector (unique vector)
let mut numbers = vec![1i, 2, 3];
numbers.push(4);
numbers.push(5);

// The type of a unique vector is written as `Vec<int>`
let more_numbers: Vec<int> = numbers.move_iter().map(|i| i+1).collect();

// The original `numbers` value can no longer be used, due to move semantics.

let mut string = String::from_str("fo");
string.push_char('o');
```

Slices are similar to fixed-size vectors, but the length is not part of the type. They simply point into a block of memory and do not have ownership over the elements.

```
// A slice
let xs = &[1, 2, 3];
```

```
// Slices have their type written as `&[int]`
let ys: &[int] = xs;

// Other vector types coerce to slices
let three = [1, 2, 3];
let zs: &[int] = three;

// An unadorned string literal is an immutable string slice
let string = "foobar";

// A string slice type is written as `&str`
let view: &str = string.slice(0, 3);
```

Square brackets denote indexing into a slice or fixed-size vector:

```
let crayons: [&str, ..3] = ["BananaMania", "Beaver", "Bittersweet"];
println!("Crayon 2 is '{}'", crayons[2]);
```

Mutable slices also exist, just as there are mutable references. However, there are no mutable string slices. Strings are a multi-byte encoding (UTF-8) of Unicode code points, so they cannot be freely mutated without the ability to alter the length.

```
let mut xs = [1i, 2i, 3i];
let view = xs.mut_slice(0, 2);
view[0] = 5;

// The type of a mutable slice is written as `&mut [T]`
let ys: &mut [int] = &mut [1i, 2i, 3i];
```

A slice or fixed-size vector can be destructured using pattern matching:

```
let numbers: &[int] = &[1, 2, 3];
let score = match numbers {
    [] => 0,
    [a] => a * 10,
    [a, b] => a * 6 + b * 4,
    [a, b, c, ..rest] => a * 5 + b * 3 + c * 2 + rest.len() as int
};
```

Both vectors and strings support a number of useful methods, defined in `std::vec`, `std::slice`, and `std::str`.

14 Ownership escape hatches

Ownership can cleanly describe tree-like data structures, and references provide non-owning pointers. However, more flexibility is often desired and Rust provides ways to escape from strict single parent ownership.

The standard library provides the `std::rc::Rc` pointer type to express *shared ownership* over a reference counted box. As soon as all of the `Rc` pointers go out of scope, the box and the contained value are destroyed.

```
use std::rc::Rc;

// A fixed-size array allocated in a reference-counted box
let x = Rc::new([1i, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
let y = x.clone(); // a new owner
let z = x; // this moves `x` into `z`, rather than creating a new owner

assert!(*z == [1i, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

// the variable is mutable, but not the contents of the box
let mut a = Rc::new([10, 9, 8, 7, 6, 5, 4, 3, 2, 1]);
a = z;
```

A garbage collected pointer is provided via `std::gc::Gc`, with a task-local garbage collector having ownership of the box. It allows the creation of cycles, and the individual `Gc` pointers do not have a destructor.

```
use std::gc::GC;

// A fixed-size array allocated in a garbage-collected box
let x = box(GC) [1i, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let y = x; // does not perform a move, unlike with `Rc`
let z = x;

assert!(*z == [1i, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
```

With shared ownership, mutability cannot be inherited so the boxes are always immutable. However, it's possible to use *dynamic* mutability via types like `std::cell::Cell` where freezing is handled via dynamic checks and can fail at runtime.

The `Rc` and `Gc` types are not sendable, so they cannot be used to share memory between tasks. Safe immutable and mutable shared memory is provided by the `sync::arc` module.

> *Note:* See a later chapter for a discussion about `Send` and sendable types.

15 Closures

Named functions, like those we've seen so far, may not refer to local variables declared outside the function: they do not close over their environment (sometimes referred to as "capturing" variables in their environment). For example, you couldn't write the following:

```
let x = 3;

// `fun` cannot refer to `x`
fn fun() -> () { println!("{}", x); }
```

A *closure* does support accessing the enclosing scope; below we will create 2 *closures* (nameless functions). Compare how `||` replaces `()` and how they try to access `x`:

```
let x = 3;

// `fun` is an invalid definition
fn  fun       () -> () { println!("{}", x) }  // cannot capture from enclosing scope
let closure = || -> () { println!("{}", x) }; // can capture from enclosing scope

// `fun_arg` is an invalid definition
fn  fun_arg       (arg: int) -> () { println!("{}", arg + x) }  // cannot capture
let closure_arg = |arg: int| -> () { println!("{}", arg + x) }; // can capture
//                        ^
// Requires a type because the implementation needs to know which `+` to use.
// In the future, the implementation may not need the help.

fun();          // Still won't work
closure();      // Prints: 3

fun_arg(7);     // Still won't work
closure_arg(7); // Prints: 10
```

Closures begin with the argument list between vertical bars and are followed by a single expression. Remember that a block, `{ <expr1>; <expr2>; ... }`, is considered a single expression: it evaluates to the result of the last expression it contains if that expression is not followed by a semicolon, otherwise the block evaluates to `()`, the unit value.

In general, return types and all argument types must be specified explicitly for function definitions. (As previously mentioned in the Functions section, omitting the return type from a function declaration is synonymous with an explicit declaration of return type unit, `()`.)

```
fn  fun   (x: int)         { println!("{}", x) } // this is same as saying `-> ()`
fn  square(x: int) -> uint { (x * x) as uint }   // other return types are explicit

// Error: mismatched types: expected `()` but found `uint`
fn  badfun(x: int)         { (x * x) as uint }
```

On the other hand, the compiler can usually infer both the argument and return types for a closure expression; therefore they are often omitted, since both a human reader and the compiler can deduce the types from the immediate context. This is in contrast to function declarations, which require types to be specified and are not subject to type inference. Compare:

```
// `fun` as a function declaration cannot infer the type of `x`, so it must be provided
fn  fun       (x: int) { println!("{}", x) }
let closure = |x     | { println!("{}", x) }; // infers `x: int`, return type `()`

// For closures, omitting a return type is *not* synonymous with `-> ()`
let add_3   = |y     | { 3i + y }; // infers `y: int`, return type `int`.

fun(10);            // Prints 10
closure(20);        // Prints 20
closure(add_3(30)); // Prints 33

fun("String"); // Error: mismatched types

// Error: mismatched types
// inference already assigned `closure` the type `|int| -> ()`
closure("String");
```

In cases where the compiler needs assistance, the arguments and return types may be annotated on closures, using the same notation as shown earlier. In the example below, since different types provide an implementation for the operator `*`, the argument type for the `x` parameter must be explicitly provided.

```
// Error: the type of `x` must be known to be used with `x * x`
let square = |x     | -> uint { (x * x) as uint };
```

In the corrected version, the argument type is explicitly annotated, while the return type can still be inferred.

```
let square_explicit = |x: int| -> uint { (x * x) as uint };
let square_infer    = |x: int|         { (x * x) as uint };

println!("{}", square_explicit(20));  // 400
println!("{}", square_infer(-20));    // 400
```

There are several forms of closure, each with its own role. The most common, called a *stack closure*, has type `||` and can directly access local variables in the enclosing scope.

```
let mut max = 0;
let f = |x: int| if x > max { max = x };
for x in [1, 2, 3].iter() {
    f(*x);
}
```

Stack closures are very efficient because their environment is allocated on the call stack and refers by pointer to captured locals. To ensure that stack closures never outlive the local variables to which they refer, stack closures are not first-class. That is, they can only be used in argument position; they cannot be stored in data structures or returned from functions. Despite these limitations, stack closures are used pervasively in Rust code.

15.1 Owned closures

Owned closures, written `proc`, hold on to things that can safely be sent between processes. They copy the values they close over, but they also own them: that is, no other code can access them. Owned closures are used in concurrent code, particularly for spawning tasks.

Closures can be used to spawn tasks. A practical example of this pattern is found when using the `spawn` function, which starts a new task.

```
use std::task::spawn;

// proc is the closure which will be spawned.
spawn(proc() {
    println!("I'm a new task")
});
```

15.2 Closure compatibility

Rust closures have a convenient subtyping property: you can pass any kind of closure (as long as the arguments and return types match) to functions that expect a `||`. Thus, when writing a higher-order function that only calls its function argument, and does nothing else with it, you should almost always declare the type of that argument as `||`. That way, callers may pass any kind of closure.

```
fn call_twice(f: ||) { f(); f(); }
let closure = || { "I'm a closure, and it doesn't matter what type I am"; };
fn function() { "I'm a normal function"; }
call_twice(closure);
call_twice(function);
```

> *Note:* Both the syntax and the semantics will be changing in small ways. At the moment they can be unsound in some scenarios, particularly with non-copyable types.

16 Methods

Methods are like functions except that they always begin with a special argument, called `self`, which has the type of the method's receiver. The `self` argument is like `this` in C++ and many other languages. Methods are called with dot notation, as in `my_vec.len()`.

Implementations, written with the `impl` keyword, can define methods on most Rust types, including structs and enums. As an example, let's define a `draw` method on our `Shape` enum.

```
# fn draw_circle(p: Point, f: f64) { }
# fn draw_rectangle(p: Point, p: Point) { }
struct Point {
    x: f64,
    y: f64
}

enum Shape {
    Circle(Point, f64),
    Rectangle(Point, Point)
}

impl Shape {
    fn draw(&self) {
        match *self {
            Circle(p, f) => draw_circle(p, f),
            Rectangle(p1, p2) => draw_rectangle(p1, p2)
        }
    }
}

let s = Circle(Point { x: 1.0, y: 2.0 }, 3.0);
s.draw();
```

This defines an *implementation* for `Shape` containing a single method, `draw`. In most respects the `draw` method is defined like any other function, except for the name `self`.

The type of `self` is the type on which the method is implemented, or a pointer thereof. As an argument it is written either `self`, `&self`, or `self: TYPE`. A caller must in turn have a compatible pointer type to call the method.

```
# fn draw_circle(p: Point, f: f64) { }
# fn draw_rectangle(p: Point, p: Point) { }
# struct Point { x: f64, y: f64 }
# enum Shape {
#     Circle(Point, f64),
#     Rectangle(Point, Point)
# }
impl Shape {
    fn draw_reference(&self) { /* ... */ }
    fn draw_owned(self: Box<Shape>) { /* ... */ }
    fn draw_value(self) { /* ... */ }
}

let s = Circle(Point { x: 1.0, y: 2.0 }, 3.0);

(&s).draw_reference();
(box s).draw_owned();
s.draw_value();
```

Methods typically take a reference self type, so the compiler will go to great lengths to convert a callee to a reference.

```
# fn draw_circle(p: Point, f: f64) { }
# fn draw_rectangle(p: Point, p: Point) { }
# struct Point { x: f64, y: f64 }
# enum Shape {
#     Circle(Point, f64),
#     Rectangle(Point, Point)
# }
# impl Shape {
#    fn draw_reference(&self) { /* ... */ }
#    fn draw_owned(self: Box<Shape>) { /* ... */ }
#    fn draw_value(self) { /* ... */ }
# }
# let s = Circle(Point { x: 1.0, y: 2.0 }, 3.0);
// As with typical function arguments, owned pointers
// are automatically converted to references
```

```
(box s).draw_reference();

// Unlike typical function arguments, the self value will
// automatically be referenced ...
s.draw_reference();

// ... and dereferenced
(& &s).draw_reference();

// ... and dereferenced and borrowed
(&box s).draw_reference();
```

Implementations may also define standalone (sometimes called "static") methods. The absence of a `self` parameter distinguishes such methods. These methods are the preferred way to define constructor functions.

```
impl Circle {
    fn area(&self) -> f64 { /* ... */ }
    fn new(area: f64) -> Circle { /* ... */ }
}
```

To call such a method, just prefix it with the type name and a double colon:

```
use std::f64::consts::PI;
struct Circle { radius: f64 }
impl Circle {
    fn new(area: f64) -> Circle { Circle { radius: (area / PI).sqrt() } }
}
let c = Circle::new(42.5);
```

17 Generics

Throughout this tutorial, we've been defining functions that act only on specific data types. With type parameters we can also define functions whose arguments have generic types, and which can be invoked with a variety of types. Consider a generic `map` function, which takes a function `function` and a vector `vector` and returns a new vector consisting of the result of applying `function` to each element of `vector`:

```
fn map<T, U>(vector: &[T], function: |v: &T| -> U) -> Vec<U> {
    let mut accumulator = Vec::new();
    for element in vector.iter() {
        accumulator.push(function(element));
```

```
    }
    return accumulator;
}
```

When defined with type parameters, as denoted by `<T, U>`, this function can be applied to any type of vector, as long as the type of `function`'s argument and the type of the vector's contents agree with each other.

Inside a generic function, the names of the type parameters (capitalized by convention) stand for opaque types. All you can do with instances of these types is pass them around: you can't apply any operations to them or pattern-match on them. Note that instances of generic types are often passed by pointer. For example, the parameter `function()` is supplied with a pointer to a value of type `T` and not a value of type `T` itself. This ensures that the function works with the broadest set of types possible, since some types are expensive or illegal to copy and pass by value.

Generic `type`, `struct`, and `enum` declarations follow the same pattern:

```
type Set<T> = std::collections::HashMap<T, ()>;

struct Stack<T> {
    elements: Vec<T>
}

enum Option<T> {
    Some(T),
    None
}
# fn main() {}
```

These declarations can be instantiated to valid types like `Set<int>`, `Stack<int>`, and `Option<int>`.

The last type in that example, `Option`, appears frequently in Rust code. Because Rust does not have null pointers (except in unsafe code), we need another way to write a function whose result isn't defined on every possible combination of arguments of the appropriate types. The usual way is to write a function that returns `Option<T>` instead of `T`.

```
# struct Point { x: f64, y: f64 }
# enum Shape { Circle(Point, f64), Rectangle(Point, Point) }
fn radius(shape: Shape) -> Option<f64> {
    match shape {
        Circle(_, radius) => Some(radius),
        Rectangle(..)     => None
```

```
    }
}
```

The Rust compiler compiles generic functions very efficiently by *monomorphizing* them. *Monomorphization* is a fancy name for a simple idea: generate a separate copy of each generic function at each call site, a copy that is specialized to the argument types and can thus be optimized specifically for them. In this respect, Rust's generics have similar performance characteristics to C++ templates.

17.1 Traits

Within a generic function—that is, a function parameterized by a type parameter, say, `T`—the operations we can do on arguments of type `T` are quite limited. After all, since we don't know what type `T` will be instantiated with, we can't safely modify or query values of type `T`. This is where *traits* come into play. Traits are Rust's most powerful tool for writing polymorphic code. Java developers will see them as similar to Java interfaces, and Haskellers will notice their similarities to type classes. Rust's traits give us a way to express *bounded polymorphism*: by limiting the set of possible types that a type parameter could refer to, they expand the number of operations we can safely perform on arguments of that type.

As motivation, let us consider copying of values in Rust. The `clone` method is not defined for values of every type. One reason is user-defined destructors: copying a value of a type that has a destructor could result in the destructor running multiple times. Therefore, values of types that have destructors cannot be copied unless we explicitly implement `clone` for them.

This complicates handling of generic functions. If we have a function with a type parameter `T`, can we copy values of type `T` inside that function? In Rust, we can't, and if we try to run the following code the compiler will complain.

```
// This does not compile
fn head_bad<T>(v: &[T]) -> T {
    v[0] // error: copying a non-copyable value
}
```

However, we can tell the compiler that the `head` function is only for copyable types. In Rust, copyable types are those that *implement the `Clone` trait*. We can then explicitly create a second copy of the value we are returning by calling the `clone` method:

```
// This does
fn head<T: Clone>(v: &[T]) -> T {
    v[0].clone()
}
```

The bounded type parameter `T: Clone` says that `head` can be called on an argument of type `&[T]` for any `T`, so long as there is an implementation of the `Clone` trait for `T`. When instantiating a generic function, we can only instantiate it with types that implement the correct trait, so we could not apply `head` to a vector whose elements are of some type that does not implement `Clone`.

While most traits can be defined and implemented by user code, three traits are automatically derived and implemented for all applicable types by the compiler, and may not be overridden:

- `Send` - Sendable types. Types are sendable unless they contain references.
- `Share` - Types that are *threadsafe* These are types that are safe to be used across several threads with access to a `&T` pointer. `Mutex<T>` is an example of a *sharable* type with internal mutable data.
- `'static` - Non-borrowed types. These are types that do not contain any data whose lifetime is bound to a particular stack frame. These are types that do not contain any references, or types where the only contained references have the `'static` lifetime. (For more on named lifetimes and their uses, see the references and lifetimes guide.)

  Note: These built-in traits were referred to as 'kinds' in earlier iterations of the language, and often still are.

Additionally, the `Drop` trait is used to define destructors. This trait provides one method called `drop`, which is automatically called when a value of the type that implements this trait is destroyed, either because the value went out of scope or because the garbage collector reclaimed it.

```
struct TimeBomb {
    explosivity: uint
}

impl Drop for TimeBomb {
    fn drop(&mut self) {
        for _ in range(0, self.explosivity) {
            println!("blam!");
        }
    }
}
```

It is illegal to call `drop` directly. Only code inserted by the compiler may call it.

17.2 Declaring and implementing traits

At its simplest, a trait is a set of zero or more *method signatures*. For example, we could declare the trait `Printable` for things that can be printed to the console, with a single method signature:

```
trait Printable {
    fn print(&self);
}
```

We say that the `Printable` trait *provides* a `print` method with the given signature. This means that we can call `print` on an argument of any type that implements the `Printable` trait.

Rust's built-in `Send` and `Share` types are examples of traits that don't provide any methods.

Traits may be implemented for specific types with impls. An impl for a particular trait gives an implementation of the methods that trait provides. For instance, the following impls of `Printable` for `int` and `String` give implementations of the `print` method.

```
# trait Printable { fn print(&self); }
impl Printable for int {
    fn print(&self) { println!("{}", *self) }
}

impl Printable for String {
    fn print(&self) { println!("{}", *self) }
}

# 1.print();
# ("foo".to_string()).print();
```

Methods defined in an impl for a trait may be called just like any other method, using dot notation, as in `1.print()`.

17.3 Default method implementations in trait definitions

Sometimes, a method that a trait provides will have the same implementation for most or all of the types that implement that trait. For instance, suppose that we wanted `bool`s and `f32`s to be printable, and that we wanted the implementation of `print` for those types to be exactly as it is for `int`, above:

```
# trait Printable { fn print(&self); }
impl Printable for f32 {
    fn print(&self) { println!("{}", *self) }
}

impl Printable for bool {
    fn print(&self) { println!("{}", *self) }
}

# true.print();
# 3.14159.print();
```

This works fine, but we've now repeated the same definition of `print` in three places. Instead of doing that, we can simply include the definition of `print` right in the trait definition, instead of just giving its signature. That is, we can write the following:

```
extern crate debug;

# fn main() {
trait Printable {
    // Default method implementation
    fn print(&self) { println!("{:?}", *self) }
}

impl Printable for int {}

impl Printable for String {
    fn print(&self) { println!("{}", *self) }
}

impl Printable for bool {}

impl Printable for f32 {}

# 1.print();
# ("foo".to_string()).print();
# true.print();
# 3.14159.print();
# }
```

Here, the impls of `Printable` for `int`, `bool`, and `f32` don't need to provide an implementation of `print`, because in the absence of a specific implementation, Rust just uses the *default method* provided in the trait definition. Depending on the trait, default methods can save a great deal of boilerplate code from having

to be written in impls. Of course, individual impls can still override the default method for `print`, as is being done above in the impl for `String`.

17.4 Type-parameterized traits

Traits may be parameterized by type variables. For example, a trait for generalized sequence types might look like the following:

```
trait Seq<T> {
    fn length(&self) -> uint;
}

impl<T> Seq<T> for Vec<T> {
    fn length(&self) -> uint { self.len() }
}
```

The implementation has to explicitly declare the type parameter that it binds, `T`, before using it to specify its trait type. Rust requires this declaration because the `impl` could also, for example, specify an implementation of `Seq<int>`. The trait type (appearing between `impl` and `for`) *refers* to a type, rather than defining one.

The type parameters bound by a trait are in scope in each of the method declarations. So, re-declaring the type parameter `T` as an explicit type parameter for `length`, in either the trait or the impl, would be a compile-time error.

Within a trait definition, `Self` is a special type that you can think of as a type parameter. An implementation of the trait for any given type `T` replaces the `Self` type parameter with `T`. The following trait describes types that support an equality operation:

```
// In a trait, `self` refers to the self argument.
// `Self` refers to the type implementing the trait.
trait PartialEq {
    fn equals(&self, other: &Self) -> bool;
}

// In an impl, `self` refers just to the value of the receiver
impl PartialEq for int {
    fn equals(&self, other: &int) -> bool { *other == *self }
}
```

Notice that in the trait definition, `equals` takes a second parameter of type `Self`. In contrast, in the `impl`, `equals` takes a second parameter of type `int`, only using `self` as the name of the receiver.

Just as in type implementations, traits can define standalone (static) methods. These methods are called by prefixing the method name with the trait name and a double colon. The compiler uses type inference to decide which implementation to use.

```
use std::f64::consts::PI;
trait Shape { fn new(area: f64) -> Self; }
struct Circle { radius: f64 }
struct Square { length: f64 }

impl Shape for Circle {
    fn new(area: f64) -> Circle { Circle { radius: (area / PI).sqrt() } }
}
impl Shape for Square {
    fn new(area: f64) -> Square { Square { length: area.sqrt() } }
}

let area = 42.5;
let c: Circle = Shape::new(area);
let s: Square = Shape::new(area);
```

17.5 Bounded type parameters and static method dispatch

Traits give us a language for defining predicates on types, or abstract properties that types can have. We can use this language to define *bounds* on type parameters, so that we can then operate on generic types.

```
# trait Printable { fn print(&self); }
fn print_all<T: Printable>(printable_things: Vec<T>) {
    for thing in printable_things.iter() {
        thing.print();
    }
}
```

Declaring `T` as conforming to the `Printable` trait (as we earlier did with `Clone`) makes it possible to call methods from that trait on values of type `T` inside the function. It will also cause a compile-time error when anyone tries to call `print_all` on a vector whose element type does not have a `Printable` implementation.

Type parameters can have multiple bounds by separating them with `+`, as in this version of `print_all` that copies elements.

```
# trait Printable { fn print(&self); }
fn print_all<T: Printable + Clone>(printable_things: Vec<T>) {
    let mut i = 0;
    while i < printable_things.len() {
        let copy_of_thing = printable_things[i].clone();
        copy_of_thing.print();
        i += 1;
    }
}
```

Method calls to bounded type parameters are *statically dispatched*, imposing no more overhead than normal function invocation, so are the preferred way to use traits polymorphically.

This usage of traits is similar to Haskell type classes.

17.6 Trait objects and dynamic method dispatch

The above allows us to define functions that polymorphically act on values of a single unknown type that conforms to a given trait. However, consider this function:

```
# type Circle = int; type Rectangle = int;
# impl Drawable for int { fn draw(&self) {} }
# fn new_circle() -> int { 1 }
trait Drawable { fn draw(&self); }

fn draw_all<T: Drawable>(shapes: Vec<T>) {
    for shape in shapes.iter() { shape.draw(); }
}
# let c: Circle = new_circle();
# draw_all(vec![c]);
```

You can call that on a vector of circles, or a vector of rectangles (assuming those have suitable `Drawable` traits defined), but not on a vector containing both circles and rectangles. When such behavior is needed, a trait name can alternately be used as a type, called an *object*.

```
# trait Drawable { fn draw(&self); }
fn draw_all(shapes: &[Box<Drawable>]) {
    for shape in shapes.iter() { shape.draw(); }
}
```

In this example, there is no type parameter. Instead, the `Box<Drawable>` type denotes any owned box value that implements the `Drawable` trait. To construct such a value, you use the `as` operator to cast a value to an object:

```
# type Circle = int; type Rectangle = bool;
# trait Drawable { fn draw(&self); }
# fn new_circle() -> Circle { 1 }
# fn new_rectangle() -> Rectangle { true }
# fn draw_all(shapes: &[Box<Drawable>]) {}

impl Drawable for Circle { fn draw(&self) { /* ... */ } }
impl Drawable for Rectangle { fn draw(&self) { /* ... */ } }

let c: Box<Circle> = box new_circle();
let r: Box<Rectangle> = box new_rectangle();
draw_all([c as Box<Drawable>, r as Box<Drawable>]);
```

We omit the code for `new_circle` and `new_rectangle`; imagine that these just return `Circle`s and `Rectangle`s with a default size. Note that, like strings and vectors, objects have dynamic size and may only be referred to via one of the pointer types. Other pointer types work as well. Casts to traits may only be done with compatible pointers so, for example, an `&Circle` may not be cast to a `Box<Drawable>`.

```
# type Circle = int; type Rectangle = int;
# trait Drawable { fn draw(&self); }
# impl Drawable for int { fn draw(&self) {} }
# fn new_circle() -> int { 1 }
# fn new_rectangle() -> int { 2 }
// An owned object
let owny: Box<Drawable> = box new_circle() as Box<Drawable>;
// A borrowed object
let stacky: &Drawable = &new_circle() as &Drawable;
```

Method calls to trait types are *dynamically dispatched*. Since the compiler doesn't know specifically which functions to call at compile time, it uses a lookup table (also known as a vtable or dictionary) to select the method to call at runtime.

This usage of traits is similar to Java interfaces.

There are some built-in bounds, such as `Send` and `Share`, which are properties of the components of types. By design, trait objects don't know the exact type of their contents and so the compiler cannot reason about those properties.

You can instruct the compiler, however, that the contents of a trait object must ascribe to a particular bound with a trailing colon (`:`). These are examples of valid types:

```
trait Foo {}
trait Bar<T> {}
```

```
fn sendable_foo(f: Box<Foo + Send>) { /* ... */ }
fn shareable_bar<T: Share>(b: &Bar<T> + Share) { /* ... */ }
```

When no colon is specified (such as the type `Box<Foo>`), it is inferred that the value ascribes to no bounds. They must be added manually if any bounds are necessary for usage.

Builtin kind bounds can also be specified on closure types in the same way (for example, by writing `fn:Send()`), and the default behaviours are the same as for traits of the same storage class.

17.7 Trait inheritance

We can write a trait declaration that *inherits* from other traits, called *supertraits*. Types that implement a trait must also implement its supertraits. For example, we can define a `Circle` trait that inherits from `Shape`.

```
trait Shape { fn area(&self) -> f64; }
trait Circle : Shape { fn radius(&self) -> f64; }
```

Now, we can implement `Circle` on a type only if we also implement `Shape`.

```
use std::f64::consts::PI;
# trait Shape { fn area(&self) -> f64; }
# trait Circle : Shape { fn radius(&self) -> f64; }
# struct Point { x: f64, y: f64 }
# fn square(x: f64) -> f64 { x * x }
struct CircleStruct { center: Point, radius: f64 }
impl Circle for CircleStruct {
    fn radius(&self) -> f64 { (self.area() / PI).sqrt() }
}
impl Shape for CircleStruct {
    fn area(&self) -> f64 { PI * square(self.radius) }
}
```

Notice that methods of `Circle` can call methods on `Shape`, as our `radius` implementation calls the `area` method. This is a silly way to compute the radius of a circle (since we could just return the `radius` field), but you get the idea.

In type-parameterized functions, methods of the supertrait may be called on values of subtrait-bound type parameters. Referring to the previous example of `trait Circle : Shape`:

```
# trait Shape { fn area(&self) -> f64; }
# trait Circle : Shape { fn radius(&self) -> f64; }
fn radius_times_area<T: Circle>(c: T) -> f64 {
    // `c` is both a Circle and a Shape
    c.radius() * c.area()
}
```

Likewise, supertrait methods may also be called on trait objects.

```
use std::f64::consts::PI;
# trait Shape { fn area(&self) -> f64; }
# trait Circle : Shape { fn radius(&self) -> f64; }
# struct Point { x: f64, y: f64 }
# struct CircleStruct { center: Point, radius: f64 }
# impl Circle for CircleStruct { fn radius(&self) -> f64 { (self.area() / PI).sqrt() } }
# impl Shape for CircleStruct { fn area(&self) -> f64 { PI * square(self.radius) } }
# fn square(x: f64) -> f64 { x * x }

let concrete = box CircleStruct{center:Point{x:3.0,y:4.0},radius:5.0};
let mycircle: Box<Circle> = concrete as Box<Circle>;
let nonsense = mycircle.radius() * mycircle.area();
```

Note: Trait inheritance does not actually work with objects yet

17.8 Deriving implementations for traits

A small number of traits in can have implementations that can be automatically derived. These instances are specified by placing the `deriving` attribute on a data type declaration. For example, the following will mean that `Circle` has an implementation for `PartialEq` and can be used with the equality operators, and that a value of type `ABC` can be randomly generated and converted to a string:

```
extern crate rand;
use std::rand::{task_rng, Rng};

#[deriving(PartialEq)]
struct Circle { radius: f64 }

#[deriving(Rand, Show)]
enum ABC { A, B, C }

fn main() {
    // Use the Show trait to print "A, B, C."
    println!("{}, {}, {}", A, B, C);
```

```
    let mut rng = task_rng();

    // Use the Rand trait to generate a random variants.
    for _ in range(0i, 10) {
        println!("{}", rng.gen::<ABC>());
    }
}
```

The full list of derivable traits is `PartialEq`, `Eq`, `PartialOrd`, `Ord`, `Encodable`, `Decodable`, `Clone`, `Hash`, `Rand`, `Default`, `Zero`, `FromPrimitive` and `Show`.

18 Crates and the module system

Rust's module system is very powerful, but because of that also somewhat complex. Nevertheless, this section will try to explain every important aspect of it.

18.1 Crates

In order to speak about the module system, we first need to define the medium it exists in:

Let's say you've written a program or a library, compiled it, and got the resulting binary. In Rust, the content of all source code that the compiler directly had to compile in order to end up with that binary is collectively called a 'crate'.

For example, for a simple hello world program your crate only consists of this code:

```
// `main.rs`
fn main() {
    println!("Hello world!");
}
```

A crate is also the unit of independent compilation in Rust: `rustc` always compiles a single crate at a time, from which it produces either a library or an executable.

Note that merely using an already compiled library in your code does not make it part of your crate.

18.2 The module hierarchy

For every crate, all the code in it is arranged in a hierarchy of modules starting with a single root module. That root module is called the 'crate root'.

All modules in a crate below the crate root are declared with the `mod` keyword:

```
// This is the crate root

mod farm {
    // This is the body of module 'farm' declared in the crate root.

    fn chicken() { println!("cluck cluck"); }
    fn cow() { println!("mooo"); }

    mod barn {
        // Body of module 'barn'

        fn hay() { println!("..."); }
    }
}

fn main() {
    println!("Hello farm!");
}
```

As you can see, your module hierarchy is now three modules deep: There is the crate root, which contains your `main()` function, and the module `farm`. The module `farm` also contains two functions and a third module `barn`, which contains a function `hay`.

18.3 Paths and visibility

We've now defined a nice module hierarchy. But how do we access the items in it from our `main` function? One way to do it is to simply fully qualifying it:

```
mod farm {
    fn chicken() { println!("cluck cluck"); }
    // ...
}

fn main() {
    println!("Hello chicken!");

    ::farm::chicken(); // Won't compile yet, see further down
}
```

The `::farm::chicken` construct is what we call a 'path'.

Because it's starting with a `::`, it's also a 'global path', which qualifies an item by its full path in the module hierarchy relative to the crate root.

If the path were to start with a regular identifier, like `farm::chicken`, it would be a 'local path' instead. We'll get to them later.

Now, if you actually tried to compile this code example, you'll notice that you get a `function 'chicken' is private` error. That's because by default, items (`fn`, `struct`, `static`, `mod`, ...) are private.

To make them visible outside their containing modules, you need to mark them *public* with `pub`:

```
mod farm {
    pub fn chicken() { println!("cluck cluck"); }
    pub fn cow() { println!("mooo"); }
    // ...
}

fn main() {
    println!("Hello chicken!");
    ::farm::chicken(); // This compiles now
}
```

Visibility restrictions in Rust exist only at module boundaries. This is quite different from most object-oriented languages that also enforce restrictions on objects themselves. That's not to say that Rust doesn't support encapsulation: both struct fields and methods can be private. But this encapsulation is at the module level, not the struct level.

Fields are *private* by default, and can be made *public* with the `pub` keyword:

```
mod farm {
# pub type Chicken = int;
# struct Human(int);
# impl Human { pub fn rest(&self) { } }
# pub fn make_me_a_farm() -> Farm { Farm { chickens: vec![], farmer: Human(0) } }
    pub struct Farm {
        chickens: Vec<Chicken>,
        pub farmer: Human
    }

    impl Farm {
        fn feed_chickens(&self) { /* ... */ }
        pub fn add_chicken(&self, c: Chicken) { /* ... */ }
```

```
    }

    pub fn feed_animals(farm: &Farm) {
        farm.feed_chickens();
    }
}

fn main() {
    let f = make_me_a_farm();
    f.add_chicken(make_me_a_chicken());
    farm::feed_animals(&f);
    f.farmer.rest();

    // This wouldn't compile because both are private:
    // `f.feed_chickens();`
    // `let chicken_counter = f.chickens.len();`
}
# fn make_me_a_farm() -> farm::Farm { farm::make_me_a_farm() }
# fn make_me_a_chicken() -> farm::Chicken { 0 }
```

Exact details and specifications about visibility rules can be found in the Rust manual.

18.4 Files and modules

One important aspect of Rust's module system is that source files and modules are not the same thing. You define a module hierarchy, populate it with all your definitions, define visibility, maybe put in a `fn main()`, and that's it.

The only file that's relevant when compiling is the one that contains the body of your crate root, and it's only relevant because you have to pass that file to `rustc` to compile your crate.

In principle, that's all you need: You can write any Rust program as one giant source file that contains your crate root and everything else in `mod ... { ... }` declarations.

However, in practice you usually want to split up your code into multiple source files to make it more manageable. Rust allows you to move the body of any module into its own source file. If you declare a module without its body, like `mod foo;`, the compiler will look for the files `foo.rs` and `foo/mod.rs` inside some directory (usually the same as of the source file containing the `mod foo;` declaration). If it finds either, it uses the content of that file as the body of the module. If it finds both, that's a compile error.

To move the content of `mod farm` into its own file, you can write:

```
// `main.rs` - contains body of the crate root
mod farm; // Compiler will look for `farm.rs` and `farm/mod.rs`

fn main() {
    println!("Hello farm!");
    ::farm::cow();
}

// `farm.rs` - contains body of module 'farm' in the crate root
pub fn chicken() { println!("cluck cluck"); }
pub fn cow() { println!("mooo"); }

pub mod barn {
    pub fn hay() { println!("..."); }
}
# fn main() { }
```

In short, `mod foo;` is just syntactic sugar for `mod foo { /* content of <...>/foo.rs or <...>/foo/mod.rs */ }`.

This also means that having two or more identical `mod foo;` declarations somewhere in your crate hierarchy is generally a bad idea, just like copy-and-paste-ing a module into multiple places is a bad idea. Both will result in duplicate and mutually incompatible definitions.

When `rustc` resolves these module declarations, it starts by looking in the parent directory of the file containing the `mod foo` declaration. For example, given a file with the module body:

```
// `src/main.rs`
mod plants;
mod animals {
    mod fish;
    mod mammals {
        mod humans;
    }
}
```

The compiler will look for these files, in this order:

```
src/plants.rs
src/plants/mod.rs

src/animals/fish.rs
src/animals/fish/mod.rs
```

```
src/animals/mammals/humans.rs
src/animals/mammals/humans/mod.rs
```

Keep in mind that identical module hierarchies can still lead to different path lookups depending on how and where you've moved a module body to its own file. For example, if you move the `animals` module into its own file:

```
// `src/main.rs`
mod plants;
mod animals;

// `src/animals.rs` or `src/animals/mod.rs`
mod fish;
mod mammals {
    mod humans;
}
```

...then the source files of `mod animals`'s submodules can either be in the same directory as the animals source file or in a subdirectory of its directory. If the animals file is `src/animals.rs`, `rustc` will look for:

```
src/animals.rs
    src/fish.rs
    src/fish/mod.rs

    src/mammals/humans.rs
    src/mammals/humans/mod.rs
```

If the animals file is `src/animals/mod.rs`, `rustc` will look for:

```
src/animals/mod.rs
    src/animals/fish.rs
    src/animals/fish/mod.rs

    src/animals/mammals/humans.rs
    src/animals/mammals/humans/mod.rs
```

These rules allow you to write small modules consisting of single source files which can live in the same directory as well as large modules which group submodule source files in subdirectories.

If you need to override where `rustc` will look for the file containing a module's source code, use the `path` compiler directive. For example, to load a `classified` module from a different file:

```
#[path="../../area51/alien.rs"]
mod classified;
```

18.5 Importing names into the local scope

Always referring to definitions in other modules with their global path gets old really fast, so Rust has a way to import them into the local scope of your module: `use`-statements.

They work like this: At the beginning of any module body, `fn` body, or any other block you can write a list of `use`-statements, consisting of the keyword `use` and a **global path** to an item without the `::` prefix. For example, this imports `cow` into the local scope:

```
use farm::cow;
# mod farm { pub fn cow() { println!("I'm a hidden ninja cow!") } }
# fn main() { cow() }
```

The path you give to `use` is per default global, meaning relative to the crate root, no matter how deep the module hierarchy is, or whether the module body it's written in is contained in its own file. (Remember: files are irrelevant.)

This is different from other languages, where you often only find a single import construct that combines the semantic of `mod foo;` and `use`-statements, and which tend to work relative to the source file or use an absolute file path - Ruby's `require` or C/C++'s `#include` come to mind.

However, it's also possible to import things relative to the module of the `use`-statement: Adding a `super::` in front of the path will start in the parent module, while adding a `self::` prefix will start in the current module:

```
# mod workaround {
# pub fn some_parent_item(){ println!("...") }
# mod foo {
use super::some_parent_item;
use self::some_child_module::some_item;
# pub fn bar() { some_parent_item(); some_item() }
# pub mod some_child_module { pub fn some_item() {} }
# }
# }
```

Again - relative to the module, not to the file.

Imports are also shadowed by local definitions: For each name you mention in a module/block, `rust` will first look at all items that are defined locally, and only if that results in no match look at items you brought in scope with corresponding `use` statements.

```
# // FIXME: Allow unused import in doc test
```

```
use farm::cow;
// ...
# mod farm { pub fn cow() { println!("Hidden ninja cow is hidden.") } }
fn cow() { println!("Mooo!") }

fn main() {
    cow() // resolves to the locally defined `cow()` function
}
```

To make this behavior more obvious, the rule has been made that `use`-statement always need to be written before any declaration, like in the example above. This is a purely artificial rule introduced because people always assumed they shadowed each other based on order, despite the fact that all items in rust are mutually recursive, order independent definitions.

One odd consequence of that rule is that `use` statements also go in front of any `mod` declaration, even if they refer to things inside them:

```
use farm::cow;
mod farm {
    pub fn cow() { println!("Moooooo?") }
}

fn main() { cow() }
```

This is what our `farm` example looks like with `use` statements:

```
use farm::chicken;
use farm::cow;
use farm::barn;

mod farm {
    pub fn chicken() { println!("cluck cluck"); }
    pub fn cow() { println!("mooo"); }

    pub mod barn {
        pub fn hay() { println!("..."); }
    }
}

fn main() {
    println!("Hello farm!");

    // Can now refer to those names directly:
    chicken();
```

```
    cow();
    barn::hay();
}
```

And here an example with multiple files:

```
// `a.rs` - crate root
use b::foo;
use b::c::bar;
mod b;
fn main() {
    foo();
    bar();
}

// `b/mod.rs`
pub mod c;
pub fn foo() { println!("Foo!"); }

// `b/c.rs`
pub fn bar() { println!("Bar!"); }
```

There also exist two short forms for importing multiple names at once:

1. Explicit mention multiple names as the last element of an `use` path:

```
use farm::{chicken, cow};
# mod farm {
#     pub fn cow() { println!("Did I already mention how hidden and ninja I am?") }
#     pub fn chicken() { println!("I'm Bat-chicken, guardian of the hidden tutorial code.")
# }
# fn main() { cow(); chicken() }
```

2. Import everything in a module with a wildcard:

```
# #![feature(globs)]
use farm::*;
# mod farm {
#     pub fn cow() { println!("Bat-chicken? What a stupid name!") }
#     pub fn chicken() { println!("Says the 'hidden ninja' cow.") }
# }
# fn main() { cow(); chicken() }
```

Note: This feature of the compiler is currently gated behind the `#![feature(globs)]` directive. More about these directives can be found in the manual.

However, that's not all. You can also rename an item while you're bringing it into scope:

```
use egg_layer = farm::chicken;
# mod farm { pub fn chicken() { println!("Laying eggs is fun!")  } }
// ...

fn main() {
    egg_layer();
}
```

In general, `use` creates a local alias: An alternate path and a possibly different name to access the same item, without touching the original, and with both being interchangeable.

18.6 Reexporting names

It is also possible to reexport items to be accessible under your module.

For that, you write `pub use`:

```
mod farm {
    pub use self::barn::hay;

    pub fn chicken() { println!("cluck cluck"); }
    pub fn cow() { println!("mooo"); }

    mod barn {
        pub fn hay() { println!("..."); }
    }
}

fn main() {
    farm::chicken();
    farm::cow();
    farm::hay();
}
```

Just like in normal `use` statements, the exported names merely represent an alias to the same thing and can also be renamed.

The above example also demonstrate what you can use `pub use` for: The nested `barn` module is private, but the `pub use` allows users of the module `farm` to access a function from `barn` without needing to know that `barn` exists.

In other words, you can use it to decouple a public api from its internal implementation.

18.7 Using libraries

So far we've only talked about how to define and structure your own crate.

However, most code out there will want to use preexisting libraries, as there really is no reason to start from scratch each time you start a new project.

In Rust terminology, we need a way to refer to other crates.

For that, Rust offers you the `extern crate` declaration:

```
// `num` ships with Rust.
extern crate num;

fn main() {
    // The rational number '1/2':
    let one_half = ::num::rational::Ratio::new(1i, 2);
}
```

A statement of the form `extern crate foo;` will cause `rustc` to search for the crate `foo`, and if it finds a matching binary it lets you use it from inside your crate.

The effect it has on your module hierarchy mirrors aspects of both `mod` and `use`:

- Like `mod`, it causes `rustc` to actually emit code: The linkage information the binary needs to use the library `foo`.
- But like `use`, all `extern crate` statements that refer to the same library are interchangeable, as each one really just presents an alias to an external module (the crate root of the library you're linking against).

Remember how `use`-statements have to go before local declarations because the latter shadows the former? Well, `extern crate` statements also have their own rules in that regard: Both `use` and local declarations can shadow them, so the rule is that `extern crate` has to go in front of both `use` and local declarations.

Which can result in something like this:

```
extern crate num;

use farm::dog;
use num::rational::Ratio;

mod farm {
    pub fn dog() { println!("woof"); }
}

fn main() {
    farm::dog();
    let a_third = Ratio::new(1i, 3);
}
```

It's a bit weird, but it's the result of shadowing rules that have been set that way because they model most closely what people expect to shadow.

18.8 Crate metadata and settings

For every crate you can define a number of metadata items, such as link name, version or author. You can also toggle settings that have crate-global consequences. Both mechanism work by providing attributes in the crate root.

For example, Rust uniquely identifies crates by their link metadata, which includes the link name and the version. It also hashes the filename and the symbols in a binary based on the link metadata, allowing you to use two different versions of the same library in a crate without conflict.

Therefore, if you plan to compile your crate as a library, you should annotate it with that information:

```
# #![allow(unused_attribute)]
// `lib.rs`

# #![crate_type = "lib"]
#![crate_id = "farm#2.5"]

// ...
# fn farm() {}
```

You can also specify crate id information in a `extern crate` statement. For example, these `extern crate` statements would both accept and select the crate define above:

```
extern crate farm;
```

```
extern crate farm = "farm#2.5";
extern crate my_farm = "farm";
```

Other crate settings and metadata include things like enabling/disabling certain errors or warnings, or setting the crate type (library or executable) explicitly:

```
# #![allow(unused_attribute)]
// `lib.rs`
// ...

// This crate is a library ("bin" is the default)
#![crate_id = "farm#2.5"]
#![crate_type = "lib"]

// Turn on a warning
#[warn(non_camel_case_types)]
# fn farm() {}
```

18.9 A minimal example

Now for something that you can actually compile yourself.

We define two crates, and use one of them as a library in the other.

```
# #![allow(unused_attribute)]
// `world.rs`
#![crate_id = "world#0.42"]

# mod secret_module_to_make_this_test_run {
pub fn explore() -> &'static str { "world" }
# }

// `main.rs`
extern crate world;
fn main() { println!("hello {}", world::explore()); }
```

Now compile and run like this (adjust to your platform if necessary):

```
$ rustc --crate-type=lib world.rs  # compiles libworld-<HASH>-0.42.rlib
$ rustc main.rs -L .               # compiles main
$ ./main
"hello world"
```

Notice that the library produced contains the version in the file name as well as an inscrutable string of alphanumerics. As explained in the previous paragraph, these are both part of Rust's library versioning scheme. The alphanumerics are a hash representing the crate's id.

18.10 The standard library and the prelude

While reading the examples in this tutorial, you might have asked yourself where all those magical predefined items like `range` are coming from.

The truth is, there's nothing magical about them: They are all defined normally in the `std` library, which is a crate that ships with Rust.

The only magical thing that happens is that `rustc` automatically inserts this line into your crate root:

```
extern crate std;
```

As well as this line into every module body:

```
use std::prelude::*;
```

The role of the `prelude` module is to re-export common definitions from `std`.

This allows you to use common types and functions like `Option<T>` or `range` without needing to import them. And if you need something from `std` that's not in the prelude, you just have to import it with an `use` statement.

For example, it re-exports `range` which is defined in `std::iter::range`:

```
use iter_range = std::iter::range;

fn main() {
    // `range` is imported by default
    for _ in range(0u, 10) {}

    // Doesn't hinder you from importing it under a different name yourself
    for _ in iter_range(0u, 10) {}

    // Or from not using the automatic import.
    for _ in ::std::iter::range(0u, 10) {}
}
```

Both auto-insertions can be disabled with an attribute if necessary:

```
# #![allow(unused_attribute)]
// In the crate root:
#![no_std]

# #![allow(unused_attribute)]
// In any module:
#![no_implicit_prelude]
```

See the API documentation for details.

19 What next?

Now that you know the essentials, check out any of the additional guides on individual topics.

- Pointers
- Lifetimes
- Tasks and communication
- Macros
- The foreign function interface
- Containers and iterators
- Documenting Rust code
- Testing Rust code
- The Rust Runtime

There is further documentation on the wiki, however those tend to be even more out of date than this document.