The Rust Language Tutorial

1 Introduction
2 Getting started
3 Syntax basics
4 Control structures
5 Data structures
6 Functions
7 Destructors
8 Ownership
9 Boxes
10 Move semantics
11 Borrowed pointers
12 Dereferencing pointers
13 Vectors and strings
14 Closures
15 Methods
16 Generics
17 Modules and crates
18 What next?

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.

Rust 0.8-pre
fbeeeebf

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

The Rust compiler currently must be built from a tarball, unless you are on Windows, in which case using the installer is recommended.

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). As such, source builds require a connection to the Internet, to fetch snapshots, and an OS that can execute the available snapshot binaries.

Snapshot binaries are currently built and tested on several platforms:

- Windows (7, Server 2008 R2), x86 only
- Linux (various distributions), x86 and x86-64
- OSX 10.6 ("Snow Leopard") 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.

> ***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. Finally, `rustc` may need to be referred to as `rustc.exe`. It's a bummer, we know.

Rust 0.8-pre
fbeeeebf

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

- g++ 4.4 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-0.7.tar.gz
$ tar -xzf rust-0.7.tar.gz
$ cd rust-0.7
$ ./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; `rustdoc`, the API-documentation tool; `rustpkg`, the Rust package manager; `rusti`, the Rust REPL; and `rust`, a tool which acts both as a unified interface for them, and for a few common command line scenarios.

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?");
}
```

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 function), and then compile it, you'll see an error message like this:

```
hello.rs:2:4: 2:16 error: unresolved name: 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.

Rust 0.8-pre
fbeeeebf

2.2 Using the rust tool

While using `rustc` directly to generate your executables, and then running them manually is a perfectly valid way to test your code, for smaller projects, prototypes, or if you're a beginner, it might be more convenient to use the `rust` tool.

The `rust` tool provides central access to the other rust tools, as well as handy shortcuts for directly running source files. For example, if you have a file `foo.rs` in your current directory, `rust run foo.rs` would attempt to compile it and, if successful, directly run the resulting binary.

To get a list of all available commands, simply call `rust` without any argument.

2.3 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`.

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.

```
fn main() {
    /* A simple loop */
    loop {
        // A tricky calculation
        if universe::recalibrate() {
            return;
        }
    }
}
```

Rust 0.8-pre
fbeeeebf

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 = 0;

while count < 10 {
    println(fmt!("count: %?", count));
    count += 1;
}
```

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

```
static MONSTER_FACTOR: float = 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 `float`, 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 = 100;
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 price;
if item == "salad" {
    price = 3.50;
} else if item == "muffin" {
    price = 2.25;
} else {
    price = 2.00;
```

Rust 0.8-pre

fbeeeebf

```
}
```

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

```
let price =
    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 `()` (nil 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`), 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 assume that an unsuffixed integer literal has type `int`.

```
let a = 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
```

Rust 0.8-pre
fbeeeebf

There are three floating-point types: `float`, `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 `f`, `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. More on strings later.

The nil 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 (like ~ in C).

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

For 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.

```
let x: float = 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 `fmt!`, a `sprintf`-style text formatter that you will often see in examples.

`fmt!` supports most of the directives that printf supports, but unlike printf, will give you a compile-time error when the types of the directives don't match the types of the arguments.

```
println(fmt!("%s is %d", "the answer", 43));

// %? will conveniently print any type
println(fmt!("what is this thing: %?", mystery_object));
```

You can define your own syntax extensions with the macro system. For details, see the macro tutorial Rust 0.8-pre

fbeeeebf

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 { return 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 labelled with a pattern, and the code compares the value against each pattern in order until one matches. The matching pattern executes its corresponding arm.

```
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 an arrow `=>`, followed by an *action* (expression). 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. 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. The asterisk (`*`) is a different wildcard that can match

Rust 0.8-pre
fbeeeebf

one or more fields in an `enum` variant.

The patterns in a match arm are followed by a fat arrow, `=>`, then an expression to evaluate. Each case is separated by commas. It's often convenient to use a block expression for each case, in which case the commas are optional.

```
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 (`(float, float)`) which are explained in section 5.3. For now you can think of tuples as a list of items.

```
use std::float;
use std::num::atan;
fn angle(vector: (float, float)) -> float {
    let pi = float::consts::pi;
    match vector {
      (0f, y) if y < 0f => 1.5 * pi,
      (0f, y) => 0.5 * pi,
      (x, y) => atan(y / x)
    }
}
```

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, `(0f, 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.

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`.

```
let (a, b) = get_tuple_of_two_ints();
```

Rust 0.8-pre
fbeeeebf

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.

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 `loop` aborts the current iteration and continues with the next.

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

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

```
use std::int;
let mut x = 5;
loop {
    x += x - 3;
    if x % 5 == 0 { break; }
    println(int::to_str(x));
}
```

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

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: float,
    y: float
}
```

Inherited mutability means that any field of a struct may be mutable, if the struct is in a mutable slot (o a field of a struct in a mutable slot, and so forth).

Rust 0.8-pre
fbeeeebf

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.

```
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, ... }`:

```
match mypoint {
    Point { x: 0.0, y: yy } => { println(yy.to_str());                     }
    Point { x: xx,  y: yy } => { println(xx.to_str() + " " + yy.to_str()); }
}
```

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.

```
match mypoint {
    Point { x, _ } => { println(x.to_str()) }
}
```

5.2 Enums

Enums are datatypes that have several alternate representations. For example, consider the type shown earlier:

```
enum Shape {
    Circle(Point, float),
    Rectangle(Point, Point)
}
```

A value of this type is either a `Circle`, in which case it contains a `Point` struct and a float, 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.

The above declaration will define 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 (taking arguments of the specified types). So `Circle(Point { x: 0f, y: 0f }, 10f)` is the way to create a new circle.

Enum variants need not have parameters. This `enum` declaration, for example, is equivalent to a C enum:

Rust 0.8-pre
fbeeeebf

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

This declaration defines `North`, `East`, `South`, and `West` as constants, all of which have type `Direction`.

When an enum is C-like (that is, when none of the variants have parameters), it is possible to explicitly set the discriminator values to a constant value:

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

If an explicit discriminator is not 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 is C-like, you can apply the `as` cast operator to convert it to its discriminator value as an `int`.

For enum types with multiple variants, destructuring is the only way to get at their contents. All variant constructors can be used as patterns, as in this definition of `area`:

```
use std::float;
fn area(sh: Shape) -> float {
    match sh {
        Circle(_, size) => float::consts::pi * size * size,
        Rectangle(Point { x, y }, Point { x: x2, y: y2 }) => (x2 - x) * (y2 - y)
    }
}
```

You can write a lone _ to ignore an individual field, and can ignore all fields of a variant like: `Circle(*)`. As in their introduction form, nullary enum patterns are written without parentheses.

```
fn point_from_direction(dir: Direction) -> Point {
    match dir {
        North => Point { x:  0f, y:  1f },
        East  => Point { x:  1f, y:  0f },
        South => Point { x:  0f, y: -1f },
        West  => Point { x: -1f, y:  0f }
    }
}
```

Rust 0.8-pre
fbeeeebf

Enum variants may also be structs. For example:

```
use std::float;
enum Shape {
    Circle { center: Point, radius: float },
    Rectangle { top_left: Point, bottom_right: Point }
}
fn area(sh: Shape) -> float {
    match sh {
        Circle { radius: radius, _ } => float::consts::pi * square(radius),
        Rectangle { top_left: top_left, bottom_right: bottom_right } => {
            (bottom_right.x - top_left.x) * (bottom_right.y - top_left.y)
        }
    }
}
```

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 except for 0 (though you may consider unit, `()`, as the empty tuple if you like).

```
let mytup: (int, int, float) = (10, 20, 30.0);
match mytup {
  (a, b, c) => info!(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, float);
let mytup: MyTup = MyTup(10, 20, 30.0);
match mytup {
  MyTup(a, b, c) => info!(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);
```

Rust 0.8-pre
fbeeeebf

For convenience, you can extract the contents of such a struct with the dereference (`*`) unary operator:

```
let my_gizmo_id: GizmoId = GizmoId(10);
let id_int: int = *my_gizmo_id;
```

Types like this can be useful to differentiate between data that have the same 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.

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 `expr: 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 nil, `()`, 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 `()`.

Rust 0.8-pre
fbeeeebf

```
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, float)) -> 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 there are no dynamic failures from accessing freed resources. When a task fails, the destructors of all objects in the task are called.

The ~ sigil represents a unique handle for a memory allocation on the heap:

```
{
    // an integer allocated on the heap
    let y = ~10;
}
// 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. A box managed by the garbage collector starts a new ownership tree, and the destructo
is called when it is collected.

Rust 0.8-pre
fbeeeebf

```
// the struct owns the objects contained in the `x` and `y` fields
```

```
struct Foo { x: int, y: ~int }

{
    // `a` is the owner of the struct, and thus the owner of the struct's fields
    let a = Foo { x: 5, y: ~10 };
}
// when `a` goes out of scope, the destructor for the `~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: ~10 };
b.x = 10;
```

If an object doesn't contain garbage-collected boxes, it consists of a single ownership tree and is given the `Owned` trait which allows it to be sent between tasks. Custom destructors can only be implemented directly on types that are `Owned`, but garbage-collected boxes can still *contain* types with custom destructors.

9 Boxes

Many modern languages represent values as pointers to heap memory by default. In contrast, Rust, like C and C++, represents such types directly. Another way to say this is that aggregate data in Rust are *unboxed*. This means that if you `let x = Point { x: 1f, y: 1f };`, you are creating a struct on the stack. If you then copy it into a data structure, you copy the entire struct, not just a pointer.

For small structs like `Point`, this is usually more efficient than allocating memory and indirecting through a pointer. But for big structs, or mutable state, it can be useful to have a single copy on the stack or on the heap, and refer to that through a pointer.

9.1 Owned boxes

An owned box (~) is a uniquely owned allocation on the heap. It inherits the mutability and lifetime of the owner as it would if there was no box:

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

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

The purpose of an owned box is to add a layer of indirection in order to create recursive data structures or cheaply pass around an object larger than a pointer. Since an owned box has a unique owner, it can only be used to represent a tree data structure.

Rust 0.8-pre
fbeeeebf

The following struct won't compile, because the lack of indirection would mean it has an infinite size:

```
struct Foo {
    child: Option<Foo>
}
```

> ***Note:*** The `Option` type is an enum that represents an *optional* value. It's comparable to a nullable pointer in many other languages, but stores the contained value unboxed.

Adding indirection with an owned pointer allocates the child outside of the struct on the heap, which makes it a finite size and won't result in a compile-time error:

```
struct Foo {
    child: Option<~Foo>
}
```

9.2 Managed boxes

A managed box (`@`) is a heap allocation with the lifetime managed by a task-local garbage collector. It will be destroyed at some point after there are no references left to the box, no later than the end of the task. Managed boxes lack an owner, so they start a new ownership tree and don't inherit mutability. They do own the contained object, and mutability is defined by the type of the managed box (`@` or `@mut`). An object containing a managed box is not `Owned`, and can't be sent between tasks.

```
let a = @5; // immutable

let mut b = @5; // mutable variable, immutable box
b = @10;

let c = @mut 5; // immutable variable, mutable box
*c = 10;

let mut d = @mut 5; // mutable variable, mutable box
*d += 5;
d = @mut 15;
```

A mutable variable and an immutable variable can refer to the same box, given that their types are compatible. Mutability of a box is a property of its type, however, so for example a mutable handle to an immutable box cannot be assigned a reference to a mutable box.

```
let a = @1;     // immutable box
let b = @mut 2; // mutable box

let mut c : @int;       // declare a variable with type managed immutable int
let mut d : @mut int;   // and one of type managed mutable int
```

Rust 0.8-pre
fbeeeebf

```
c = a;          // box type is the same, okay
d = b;          // box type is the same, okay
```

```
// but b cannot be assigned to c, or a to d
c = b;          // error
```

10 Move semantics

Rust uses a shallow copy for parameter passing, assignment and returning values from functions. A shallow copy is considered a move of ownership if the ownership tree of the copied value includes an owned box or a type with a custom destructor. After a value has been moved, it can no longer be used from the source location and will not be destroyed there.

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

Since in owned boxes mutability is a property of the owner, not the box, mutable boxes may become immutable when they are moved, and vice-versa.

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

11 Borrowed pointers

Rust's borrowed pointers are a general purpose reference type. In contrast with owned boxes, where the holder of an owned box is the owner of the pointed-to memory, borrowed pointers never imply ownership. A pointer can be borrowed 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: float,
    y: float
}
```

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

```
let on_the_stack : Point  =  Point { x: 3.0, y: 4.0 };
let managed_box  : @Point = @Point { x: 5.0, y: 1.0 };
```

Rust 0.8-pre
fbeeeebf

```
let owned_box    : ~Point = ~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. For example, we might like to compute the distance between `on_the_stack` and `managed_box`, or between `managed_box` and `owned_box`. 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 borrowed pointers to do this:

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

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

```
compute_distance(&on_the_stack, managed_box);
compute_distance(managed_box, owned_box);
```

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 value. 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.

In the case of the boxes `managed_box` and `owned_box`, however, no explicit action is necessary. The compiler will automatically convert a box like `@point` or `~point` to a borrowed pointer like `&point`. This is another form of borrowing; in this case, the contents of the managed/owned box are being lent out.

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 borrowed pointer to go out of scope) before you can make full use of it again.

For a more in-depth explanation of borrowed pointers, read the borrowed pointer tutorial.

11.1 Freezing

Borrowing an immutable pointer to an object freezes it and prevents mutation. `Owned` objects have freezing enforced statically at compile-time.

```
let mut x = 5;
{
    let y = &x; // x is now frozen, it cannot be modified
}
```

Rust 0.8-pre

fbeeeebf

```
// x is now unfrozen again
```

Mutable managed boxes handle freezing dynamically when any of their contents are borrowed, and the task will fail if an attempt to modify them is made while they are frozen:

```
let x = @mut 5;
let y = x;
{
    let z = &*y; // the managed box is now frozen
    // modifying it through x or y will cause a task failure
}
// the box is now unfrozen again
```

12 Dereferencing pointers

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

```
let managed = @10;
let owned = ~20;
let borrowed = &30;

let sum = *managed + *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 managed = @mut 10;
let mut owned = ~20;

let mut value = 30;
let borrowed = &mut value;

*managed = *owned + 10;
*owned = *borrowed + 100;
*borrowed = *managed + 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.

```
let start = @Point { x: 10f, y: 20f };
let end = ~Point { x: (*start).x + 100f, y: (*start).y + 100f };
let rect = &Rectangle(*start, *end);
let area = (*rect).area();
```

Rust 0.8-pre
fbeeeebf

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.

```
let start = @Point { x: 10f, y: 20f };
let end = ~Point { x: start.x + 100f, y: start.y + 100f };
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

```
let point = &@~Point { x: 10f, y: 20f };
println(fmt!("%f", point.x));
```

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

13 Vectors and strings

A vector is a contiguous section of memory containing zero or more values of the same type. Like other types in Rust, vectors can be stored on the stack, the local heap, or the exchange heap. Borrowed pointers to vectors are also called 'slices'.

```
// A fixed-size stack vector
let stack_crayons: [Crayon, ..3] = [Almond, AntiqueBrass, Apricot];

// A borrowed pointer to stack-allocated vector
let stack_crayons: &[Crayon] = &[Aquamarine, Asparagus, AtomicTangerine];

// A local heap (managed) vector of crayons
let local_crayons: @[Crayon] = @[BananaMania, Beaver, Bittersweet];

// An exchange heap (owned) vector of crayons
let exchange_crayons: ~[Crayon] = ~[Black, BlizzardBlue, Blue];
```

The + operator means concatenation when applied to vector types.

```
let my_crayons = ~[Almond, AntiqueBrass, Apricot];
let your_crayons = ~[BananaMania, Beaver, Bittersweet];

// Add two vectors to create a new one
let our_crayons = my_crayons + your_crayons;

// .push_all() will append to a vector, provided it lives in a mutable slot
let mut my_crayons = my_crayons;
my_crayons.push_all(your_crayons);
```

Rust 0.8-pre
fbeeeebf

> ***Note:*** The above examples of vector addition use owned vectors. Some operations on slices and stack vectors are not yet well-supported. Owned vectors are often the most usable.

Square brackets denote indexing into a vector:

```
let crayons: [Crayon, ..3] = [BananaMania, Beaver, Bittersweet];
match crayons[0] {
    Bittersweet => draw_scene(crayons[0]),
    _ => ()
}
```

A vector can be destructured using pattern matching:

```
let numbers: [int, ..3] = [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
};
```

The elements of a vector *inherit the mutability of the vector*, and as such, individual elements may not be reassigned when the vector lives in an immutable slot.

```
let crayons: ~[Crayon] = ~[BananaMania, Beaver, Bittersweet];

crayons[0] = Apricot; // ERROR: Can't assign to immutable vector
```

Moving it into a mutable slot makes the elements assignable.

```
let crayons: ~[Crayon] = ~[BananaMania, Beaver, Bittersweet];

// Put the vector into a mutable slot
let mut mutable_crayons = crayons;

// Now it's mutable to the bone
mutable_crayons[0] = Apricot;
```

This is a simple example of Rust's *dual-mode data structures*, also referred to as *freezing and thawing*.

Strings are implemented with vectors of u8, though they have a distinct type. They support most of the same allocation options as vectors, though the string literal without a storage sigil (for example, `"foo"`) is treated differently than a comparable vector (`[foo]`). Whereas plain vectors are stack-allocated fixed-length vectors, plain strings are borrowed pointers to read-only (static) memory. All strings are immutable.

Rust 0.8-pre
fbeeeebf

```
// A plain string is a slice to read-only (static) memory
let stack_crayons: &str = "Almond, AntiqueBrass, Apricot";

// The same thing, but with the `&`
let stack_crayons: &str = &"Aquamarine, Asparagus, AtomicTangerine";

// A local heap (managed) string
let local_crayons: @str = @"BananaMania, Beaver, Bittersweet";

// An exchange heap (owned) string
let exchange_crayons: ~str = ~"Black, BlizzardBlue, Blue";
```

Both vectors and strings support a number of useful methods, defined in `std::vec` and `std::str`. Here are some examples.

```
let crayons = [Almond, AntiqueBrass, Apricot];

// Check the length of the vector
assert!(crayons.len() == 3);
assert!(!crayons.is_empty());

// Iterate over a vector, obtaining a pointer to each element
// (`for` is explained in the container/iterator tutorial)
for crayon in crayons.iter() {
    let delicious_crayon_wax = unwrap_crayon(*crayon);
    eat_crayon_wax(delicious_crayon_wax);
}

// Map vector elements
let crayon_names = crayons.map(|v| crayon_to_str(*v));
let favorite_crayon_name = crayon_names[0];

// Remove whitespace from before and after the string
let new_favorite_crayon_name = favorite_crayon_name.trim();

if favorite_crayon_name.len() > 5 {
   // Create a substring
   println(favorite_crayon_name.slice_chars(0, 5));
}
```

14 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:

Rust 0.8-pre
fbeeeebf

```
let foo = 10;

fn bar() -> int {
   return foo; // `bar` cannot refer to `foo`
}
```

Rust also supports *closures*, functions that can access variables in the enclosing scope.

```
fn call_closure_with_ten(b: &fn(int)) { b(10); }

let captured_var = 20;
let closure = |arg| println(fmt!("captured_var=%d, arg=%d", captured_var, arg));

call_closure_with_ten(closure);
```

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 types of the arguments are generally omitted, as is the return type, because the compiler can almost always infer them. In the rare case where the compiler needs assistance, though, the arguments and return types may be annotated.

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

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

```
let mut max = 0;
[1, 2, 3].map(|x| if *x > max { max = *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.

14.1 Managed closures

When you need to store a closure in a data structure, a stack closure will not do, since the compiler will refuse to let you store it. For this purpose, Rust provides a type of closure that has an arbitrary lifetime, written `@fn` (boxed closure, analogous to the `@` pointer type described earlier). This type of closure *is* first-class.

A managed closure does not directly access its environment, but merely copies out the values that it closes over into a private data structure. This means that it can not assign to these variables, and cannot observe updates to them.

Rust 0.8-pre
fbeeeebf

This code creates a closure that adds a given string to its argument, returns it from a function, and then calls it:

```
fn mk_appender(suffix: ~str) -> @fn(~str) -> ~str {
    // The compiler knows that we intend this closure to be of type @fn
    return |s| s + suffix;
}

fn main() {
    let shout = mk_appender(~"!");
    println(shout(~"hey ho, let's go"));
}
```

14.2 Owned closures

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

14.3 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 `&fn()`. 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 `&fn()`. That way, callers may pass any kind of closure.

```
fn call_twice(f: &fn()) { 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.

14.4 Do syntax

The do expression provides a way to treat higher-order functions (functions that take closures as arguments) as control structures.

Consider this function that iterates over a vector of integers, passing in a pointer to each integer in the vector:

Rust 0.8-pre

fbeeeebf

```
fn each(v: &[int], op: &fn(v: &int)) {
   let mut n = 0;
   while n < v.len() {
       op(&v[n]);
       n += 1;
   }
}
```

As a caller, if we use a closure to provide the final operator argument, we can write it in a way that has a pleasant, block-like structure.

```
each([1, 2, 3], |n| {
    do_some_work(n);
});
```

This is such a useful pattern that Rust has a special form of function call that can be written more like a built-in control structure:

```
do each([1, 2, 3]) |n| {
    do_some_work(n);
}
```

The call is prefixed with the keyword `do` and, instead of writing the final closure inside the argument list, it appears outside of the parentheses, where it looks more like a typical block of code.

`do` is a convenient way to create tasks with the `task::spawn` function. `spawn` has the signature `spawn(fn: ~fn())`. In other words, it is a function that takes an owned closure that takes no arguments.

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

do spawn() || {
    debug!("I'm a task, whatever");
}
```

Look at all those bars and parentheses -- that's two empty argument lists back to back. Since that is so unsightly, empty argument lists may be omitted from `do` expressions.

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

do spawn {
   debug!("Kablam!");
}
```

If you want to see the output of `debug!` statements, you will need to turn on `debug!` logging. To enable `debug!` logging, set the RUST_LOG environment variable to the name of your crate, which, fo a file named `foo.rs`, will be `foo` (e.g., with bash, `export RUST_LOG=foo`).

Rust 0.8-pre
fbeeeebf

15 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.

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

enum Shape {
    Circle(Point, float),
    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: 1f, y: 2f }, 3f);
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`, `@self`, or `~self`. A caller must in turn have a compatible pointer type to call the method.

```
impl Shape {
    fn draw_borrowed(&self) { ... }
    fn draw_managed(@self) { ... }
    fn draw_owned(~self) { ... }
    fn draw_value(self) { ... }
}

let s = Circle(Point { x: 1f, y: 2f }, 3f);
```

```
(@s).draw_managed();
(~s).draw_owned();
(&s).draw_borrowed();
s.draw_value();
```

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

```
// As with typical function arguments, managed and owned pointers
// are automatically converted to borrowed pointers

(@s).draw_borrowed();
(~s).draw_borrowed();

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

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

// ... and dereferenced and borrowed
(&@~s).draw_borrowed();
```

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) -> float { ... }
    fn new(area: float) -> Circle { ... }
}
```

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

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

16 Generics

Rust 0.8-pre
fbeeeebf

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: &fn(v: &T) -> U) -> ~[U] {
    let mut accumulator = ~[];
    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:

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

struct Stack<T> {
    elements: ~[T]
}

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

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`.

```
fn radius(shape: Shape) -> Option<float> {
    match shape {
        Circle(_, radius) => Some(radius),
        Rectangle(*)      => None
```

Rust 0.8-pre
fbeeeebf

```
    }
}
```

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.

16.1 Traits

Within a generic function the operations available on generic types are very limited. After all, since the function doesn't know what types it is operating on, it can't safely modify or query their values. 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 are a form of *bounded polymorphism*: a trait is a way of limiting the set of possible types that a type parameter could refer to.

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

This complicates handling of generic functions. If you have a type parameter `T`, can you copy values of that type? In Rust, you can't, and if you 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: that is, those that implement the `Clone` trait. In that case, we can explicitly create a second copy of the value we are returning using the `clone` keyword:

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

This says that we can call `head` on any type `T` as long as that type implements the `Clone` trait. When instantiating a generic function, you can only instantiate it with types that implement the correct trait, so you could not apply `head` to a type that does not implement `Clone`.

While most traits can be defined and implemented by user code, two 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 managed boxes, managed closures, or borrowed pointers.

Rust 0.8-pre
fbeeeebf

- `Freeze` - Constant (immutable) types. These are types that do not contain anything intrinsically mutable. Intrinsically mutable values include `@mut` and `Cell` in the standard library.

> ***Note:*** These two 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 defines 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(&self) {
        do self.explosivity.times {
            println("blam!");
        }
    }
}
```

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

16.2 Declaring and implementing traits

A trait consists of a set of methods without bodies, or may be empty, as is the case with `Send` and `Freeze`. For example, we could declare the trait `Printable` for things that can be printed to the console, with a single method:

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

Traits may be implemented for specific types with impls. An impl that implements a trait includes the name of the trait at the start of the definition, as in the following impls of `Printable` for `int` and `~str`.

```
impl Printable for int {
    fn print(&self) { println(fmt!("%d", *self)) }
}

impl Printable for ~str {
    fn print(&self) { println(*self) }
}
```

Rust 0.8-pre
fbeeeebf

Methods defined in an implementation of a trait may be called just like any other method, using dot notation, as in `1.print()`. Traits may themselves contain type parameters. A trait for generalized sequence types might look like the following:

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

impl<T> Seq<T> for ~[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 `len`, 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 Eq {
    fn equals(&self, other: &Self) -> bool;
}

// In an impl, `self` refers just to the value of the receiver
impl Eq 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::float::consts::pi;
trait Shape { fn new(area: float) -> Self; }
struct Circle { radius: float }
struct Square { length: float }

impl Shape for Circle {
```

Rust 0.8-pre
fbeeeebf

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

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

16.3 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.

```
fn print_all<T: Printable>(printable_things: ~[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 an array 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.

```
fn print_all<T: Printable + Clone>(printable_things: ~[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.

16.4 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:

```
trait Drawable { fn draw(&self); }

fn draw_all<T: Drawable>(shapes: ~[T]) {
    for shape in shapes.iter() { shape.draw(); }
}
```

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

```
fn draw_all(shapes: &[@Drawable]) {
    for shape in shapes.iter() { shape.draw(); }
}
```

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

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

let c: @Circle = @new_circle();
let r: @Rectangle = @new_rectangle();
draw_all([c as @Drawable, r as @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 an `~Drawable`.

```
// A managed object
let boxy: @Drawable = @new_circle() as @Drawable;
// An owned object
let owny: ~Drawable = ~new_circle() as ~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.

16.5 Trait inheritance

Rust 0.8-pre
fbeeeebf

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

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

```
use std::float::consts::pi;
struct CircleStruct { center: Point, radius: float }
impl Circle for CircleStruct {
    fn radius(&self) -> float { (self.area() / pi).sqrt() }
}
impl Shape for CircleStruct {
    fn area(&self) -> float { 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. Refering to the previous example of `trait Circle : Shape`:

```
fn radius_times_area<T: Circle>(c: T) -> float {
    // `c` is both a Circle and a Shape
    c.radius() * c.area()
}
```

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

```
use std::float::consts::pi;

let concrete = @CircleStruct{center:Point{x:3f,y:4f},radius:5f};
let mycircle: Circle = concrete as @Circle;
let nonsense = mycircle.radius() * mycircle.area();
```

> ***Note:*** Trait inheritance does not actually work with objects yet

16.6 Deriving implementations for traits

A small number of traits in `std` and `extra` 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 `Eq` and can be used with the equality operators, and that a value of type `ABC` can be randomly generated and converted to a

Rust 0.8-pre
fbeeeebf

string:

```
#[deriving(Eq)]
struct Circle { radius: float }

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

The full list of derivable traits is `Eq`, `TotalEq`, `Ord`, `TotalOrd`, `Encodable` `Decodable`, `Clone`, `DeepClone`, `IterBytes`, `Rand`, `Zero`, and `ToStr`.

17 Modules and crates

The Rust namespace is arranged in a hierarchy of modules. Each source (.rs) file represents a single module and may in turn contain additional modules.

```
mod farm {
    pub fn chicken() -> &str { "cluck cluck" }
    pub fn cow() -> &str { "mooo" }
}

fn main() {
    println(farm::chicken());
}
```

The contents of modules can be imported into the current scope with the `use` keyword, optionally giving it an alias. `use` may appear at the beginning of crates, `mod`s, `fn`s, and other blocks.

```
// Bring `chicken` into scope
use farm::chicken;

fn chicken_farmer() {
    // The same, but name it `my_chicken`
    use my_chicken = farm::chicken;
    ...
}
```

These farm animal functions have a new keyword, `pub`, attached to them. The `pub` keyword modifies an item's visibility, making it visible outside its containing module. An expression with `::`, like `farm::chicken`, can name an item outside of its containing module. Items, such as those declared with `fn`, `struct`, `enum`, `type`, or `static`, are module-private by default.

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. Note that fields and methods are *public* by default.

Rust 0.8-pre
fbeeeebf

```
pub mod farm {
    pub struct Farm {
        priv chickens: ~[Chicken],
        priv cows: ~[Cow],
        farmer: Human
    }

    impl Farm {
        priv fn feed_chickens(&self) { ... }
        priv fn feed_cows(&self) { ... }
        pub fn add_chicken(&self, c: Chicken) { ... }
    }

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

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

17.1 Crates

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

When compiling a single `.rs` source file, the file acts as the whole crate. You can compile it with the `--lib` compiler switch to create a shared library, or without, provided that your file contains a `fn main` somewhere, to create an executable.

Larger crates typically span multiple files and are, by convention, compiled from a source file with the `.rc` extension, called a *crate file*. The crate file extension distinguishes source files that represent crates from those that do not, but otherwise source files and crate files are identical.

A typical crate file declares attributes associated with the crate that may affect how the compiler processes the source. Crate attributes specify metadata used for locating and linking crates, the type of crate (library or executable), and control warning and error behavior, among other things. Crate files additionally declare the external crates they depend on as well as any modules loaded from other files.

```
// Crate linkage metadata
#[link(name = "farm", vers = "2.5", author = "mjh")];
```

Rust 0.8-pre

fbeeeebf

```
// Make a library ("bin" is the default)
#[crate_type = "lib"];

// Turn on a warning
#[warn(non_camel_case_types)]

// Link to the standard library
extern mod std;

// Load some modules from other files
mod cow;
mod chicken;
mod horse;

fn main() {
    ...
}
```

Compiling this file will cause `rustc` to look for files named `cow.rs`, `chicken.rs`, and `horse.rs` in the same directory as the `.rc` file, compile them all together, and, based on the presence of the `crate_type = "lib"` attribute, output a shared library or an executable. (If the line `#[crate_type = "lib"];` was omitted, `rustc` would create an executable.)

The `#[link(...)]` attribute provides meta information about the module, which other crates can use to load the right module. More about that later.

To have a nested directory structure for your source files, you can nest mods:

```
mod poultry {
    mod chicken;
    mod turkey;
}
```

The compiler will now look for `poultry/chicken.rs` and `poultry/turkey.rs`, and export their content in `poultry::chicken` and `poultry::turkey`. You can also provide a `poultry.rs` to add content to the `poultry` module itself.

17.2 Using other crates

The `extern mod` directive lets you use a crate (once it's been compiled into a library) from inside another crate. `extern mod` can appear at the top of a crate file or at the top of modules. It will cause the compiler to look in the library search path (which you can extend with the `-L` switch) for a compiled Rust library with the right name, then add a module with that crate's name into the local scope.

For example, `extern mod std` links the standard library.

When a comma-separated list of name/value pairs appears after `extern mod`, the compiler front-end matches these pairs against the attributes provided in the `link` attribute of the crate file. The front-enc

Rust 0.8-pre
fbeeeebf

will only select this crate for use if the actual pairs match the declared attributes. You can provide a name value to override the name used to search for the crate.

Our example crate declared this set of `link` attributes:

```
#[link(name = "farm", vers = "2.5", author = "mjh")];
```

Which you can then link with any (or all) of the following:

```
extern mod farm;
extern mod my_farm (name = "farm", vers = "2.5");
extern mod my_auxiliary_farm (name = "farm", author = "mjh");
```

If any of the requested metadata do not match, then the crate will not be compiled successfully.

17.3 A minimal example

Now for something that you can actually compile yourself, we have these two files:

```
// world.rs
#[link(name = "world", vers = "1.0")];
pub fn explore() -> &str { "world" }
```

```
// main.rs
extern mod world;
fn main() { println(~"hello " + world::explore()); }
```

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

```
> rustc --lib world.rs  # compiles libworld-94839cbfe144198-1.0.so
> rustc main.rs -L .    # compiles main
> ./main
"hello world"
```

Notice that the library produced contains the version in the filename as well as an inscrutable string of alphanumerics. These are both part of Rust's library versioning scheme. The alphanumerics are a hash representing the crate metadata.

17.4 The standard library

The Rust standard library provides runtime features required by the language, including the task scheduler and memory allocators, as well as library support for Rust built-in types, platform abstractions, and other commonly used features.

`std` includes modules corresponding to each of the integer types, each of the floating point types, the `bool` type, tuples, characters, strings, vectors, managed boxes, owned boxes, and unsafe and borrowed pointers. Additionally, `std` provides some pervasive types (`option` and `result`), task creation and communication primitives, platform abstractions (`os` and `path`), basic I/O abstractions

Rust 0.8-pre
fbeeeebf

(`io`), containers like `hashmap`, common traits (`kinds`, `ops`, `cmp`, `num`, `to_str`, `clone`), and complete bindings to the C standard library (`libc`).

17.4.1 Standard Library injection and the Rust prelude

`std` is imported at the topmost level of every crate by default, as if the first line of each crate was

```
extern mod std;
```

This means that the contents of std can be accessed from from any context with the `std::` path prefix, as in `use std::vec`, `use std::task::spawn`, etc.

Additionally, `std` contains a `prelude` module that reexports many of the most common standard modules, types and traits. The contents of the prelude are imported into every *module* by default. Implicitly, all modules behave as if they contained the following prologue:

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

18 What next?

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

- Borrowed pointers
- Tasks and communication
- Macros
- The foreign function interface
- Containers and iterators

There is further documentation on the wiki.

Rust 0.8-pre

fbeeeebf