Rust For Systems Programmers

A Rust tutorial for experienced C and C++ programmers.

Jump to contents. Jump to contributing.

This tutorial is intended for programmers who already know how pointers and references work and are used to systems programming concepts such as integer widths and memory management. We intend to cover, primarily, the differences between Rust and C++ to get you writing Rust programs quickly without lots of fluff you probably already know.

Hopefully, Rust is a pretty intuitive language for C++ programmers. Most of the syntax is pretty similar. The big difference (in my experience) is that the sometimes vague concepts of good systems programming are strictly enforced by the compiler. This can be infuriating at first - there are things you want to do, but the compiler won't let you (at least in safe code), and sometimes these things *are* safe, but you can't convince the compiler of that. However, you'll quickly develop a good intuition for what is allowed. Communicating your own notions of memory safety to the compiler requires some new and sometimes complicated type annotations. But if you have a strong idea of lifetimes for your objects and experience with generic programming, they shouldn't be too tough to learn.

This tutorial started as a series of blog posts. Partly as an aid for me (@nrc) learning Rust (there is no better way to check that you have learnt something than to try and explain it to somebody else) and partly because I found the existing resources for learning Rust unsatisfactory - they spent too much time on the basics that I already knew and used higher level intuitions to describe concepts that could better be explained to me using lower level intuitions. Since then, the documentation for Rust has got *much* better, but I still think that existing C++ programmers are an audience who are a natural target for Rust, but are not particularly well catered for.

Contents

Other resources

- The Rust book/guide - the best place for learning Rust in general and probably the best place to go for a second opinion on stuff here or for stuff not covered.
- Rust API documentation - detailed documentation for the Rust libraries.
- The Rust reference manual - a little out of date in places, but thorough; good for looking up details.
- Discuss forum - general forum for discussion or questions about using and learning Rust.
- StackOverflow Rust questions - answers to many beginner and advanced questions about Rust, but be careful though - Rust has changed *a lot* over the years and some of the answers might be very out of date.
- A Firehose of Rust - a recorded talk introducing C++ programmers to how lifetimes, mutable aliasing, and move semantics work in Rust

Contributing

Yes please!

If you spot a typo or mistake, please submit a PR, don't be shy! Please feel free to file an issue for larger changes or for new chapters you'd like to see. I'd also be happy to see re-organisation of existing work or expanded examples, if you feel the tutorial could be improved in those ways.

If you'd like to contribute a paragraph, section, or chapter please do! If you want ideas for things to cover, see the list of issues, in particular those tagged new material. If you're not sure of something, please get in touch by pinging me here (@nrc) or on irc (nrc, on #rust or #rust-internals).

Style

Obviously, the intended audience is C++ programmers. The tutorial should concentrate on things that will be new to experienced C++ programmers, rather than a general audience (although, I don't assume the audience is familiar with the most recent versions of C++). I'd like to avoid too much basic material and definitely avoid too much overlap with other resources, in particular the Rust guide/book.

Work on edge case use cases (e.g., using a different build system from Cargo, or writing syntax extensions, using unstable APIs) is definitely welcome, as is in-depth work on topics already covered at a high level.

I'd like to avoid recipe-style examples for converting C++ code to Rust code, but small examples of this kind are OK.

Use of different formats (e.g., question and answer/FAQs, or larger worked examples) are welcome.

I don't plan on adding exercises or suggestions for mini-projects, but if you're interested in that, let me know.

I'm aiming for a fairly academic tone, but not too dry. All writing should be in English (British English, not American English; although I would be very happy to have localisations/translations into any language, including American English) and be valid GitHub markdown. For advice on writing style, grammar, punctuation, etc. see the Oxford Style Manual or The Economist Style Guide. Please limit width to 80 columns. I am a fan of the Oxford comma.

Don't feel like work has to be perfect to be submitted, I'm happy to edit and I'm sure other people will be in the future.

Introduction - hello world!

If you are using C or C++, it is probably because you have to - either you need low-level access to the system, or need every last drop of performance, or both. Rust aims to offer the same level of abstraction around memory, the same performance, but be safer and make you more productive.

Concretely, there are many languages out there that you might prefer to use to C++: Java, Scala, Haskell, Python, and so forth, but you can't because either the level of abstraction is too high (you don't get direct access to memory, you are forced to use garbage collection, etc.), or there are performance issues (either performance is unpredictable or it's simply not fast enough). Rust does not force you to use garbage collection, and as in C++, you get raw pointers to memory to play with. Rust subscribes to the 'pay for what you use' philosophy of C++. If you don't use a feature, then you don't pay any performance overhead for its existence. Furthermore, all language features in Rust have a predictable (and usually small) cost.

Whilst these constraints make Rust a (rare) viable alternative to C++, Rust also has benefits: it is memory safe - Rust's type system ensures that you don't get the kind of memory errors which are common in C++ - accessing un-initialised memory, and dangling pointers - all are impossible in Rust. Furthermore, whenever other constraints allow, Rust strives to prevent other safety issues too - for example, all array indexing is bounds checked (of course, if you want to avoid the cost, you can (at the expense of safety) - Rust allows you to do this in unsafe blocks, along with many other unsafe things. Crucially, Rust ensures that unsafety in unsafe blocks stays in unsafe blocks and can't affect the rest of your program). Finally, Rust takes many concepts from modern programming languages and introduces them to the systems language space. Hopefully, that makes programming in Rust more productive, efficient, and enjoyable.

In the rest of this section we'll download and install Rust, create a minimal Cargo project, and implement Hello World.

Getting Rust

You can get Rust from http://www.rust-lang.org/install.html. The downloads from there include the Rust compiler, standard libraries, and Cargo, which is a package manager and build tool for Rust.

Rust is available on three channels: stable, beta, and nightly. Rust works on a rapid-release, schedule with new releases every six weeks. On the release date, nightly becomes beta and beta becomes stable.

Nightly is updated every night and is ideal for users who want to experiment with cutting edge features and ensure that their libraries will work with future Rust.

Stable is the right choice for most users. Rust's stability guarantees only apply to the stable channel.

Beta is designed to mostly be used in users' CI to check that their code will continue to work as expected.

So, you probably want the stable channel. If you're on Linux or OS X, the easiest way to get it is to run

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

On Windows, a similarly easy way would be to run

```
choco install rust
```

For other ways to install, see http://www.rust-lang.org/install.html.

You can find the source at github.com/rust-lang/rust. To build the compiler, run `./configure && make rustc`. See building-from-source for more detailed instructions.

Hello World!

The easiest and most common way to build Rust programs is to use Cargo. To start a project called `hello` using Cargo, run `cargo new --bin hello`. This will create a new directory called `hello` inside which is a `Cargo.toml` file and a `src` directory with a file called `main.rs`.

`Cargo.toml` defines dependencies and other metadata about our project. We'll come back to it in detail later.

All our source code will go in the `src` directory. `main.rs` already contains a Hello World program. It looks like this:

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

To build the program, run `cargo build`. To build and run it, `cargo run`. If you do the latter, you should be greeted in the console. Success!

Cargo will have made a `target` directory and put the executable in there.

If you want to use the compiler directly you can run `rustc src/hello.rs` which will create an executable called `hello`. See `rustc --help` for lots of options.

OK, back to the code. A few interesting points - we use `fn` to define a function or method. `main()` is the default entry point for our programs (we'll leave program args for later). There are no separate declarations or header files as with C++. `println!` is Rust's equivalent of printf. The `!` means that it is a macro. A subset of the standard library is available without needing to be explicitly imported/included (the prelude). The `println!` macro is included as part of that subset.

Lets change our example a little bit:

```
fn main() {
    let world = "world";
    println!("Hello {}!", world);
}
```

`let` is used to introduce a variable, world is the variable name and it is a string (technically the type is `&'static str`, but more on that later). We don't need to specify the type, it will be inferred for us.

Using `{}` in the `println!` statement is like using `%s` in printf. In fact, it is a bit more general than that because Rust will try to convert the variable to a string if it is not one already1 (like `operator<<()` in C++). You can easily play around with this sort of thing - try multiple strings and using numbers (integer and float literals will work).

If you like, you can explicitly give the type of `world`:

```
let world: &'static str = "world";
```

In C++ we write `T x` to declare a variable `x` with type `T`. In Rust we write `x: T`, whether in `let` statements or function signatures, etc. Mostly we omit explicit types in `let` statements, but they are required for function arguments. Lets add another function to see it work:

```
fn foo(_x: &'static str) -> &'static str {
    "world"
}

fn main() {
    println!("Hello {}!", foo("bar"));
}
```

The function `foo` has a single argument `_x` which is a string literal (we pass it "bar" from `main`)2.

The return type for a function is given after `->`. If the function doesn't return anything (a void function in C++), we don't need to give a return type at all (as in `main`). If you want to be super-explicit, you can write `-> ()`, `()` is the void type in Rust.

You don't need the `return` keyword in Rust, if the last expression in a function body (or any other block, we'll see more of this later) is not finished with a semicolon, then it is the return value. So `foo` will return "world". The `return` keyword still exists so we can do early returns. You can replace `"world"` with `return "world";` and it will have the same effect.

Why?

I would like to motivate some of the language features above. Local type inference is convenient and useful without sacrificing safety or performance (it's even in modern versions of C++ now). A minor convenience is that language items are consistently denoted by keyword (`fn`, `let`, etc.), this makes scanning by eye or by tools easier, in general the syntax of Rust is simpler and more consistent than C++. The `println!` macro is safer than printf - the number of arguments is statically checked against the number of 'holes' in the string and the arguments are type checked. This means you can't make the printf mistakes of printing memory as if it had a different type or addressing memory further down the stack by mistake. These are fairly minor things, but I hope they illustrate the philosophy behind the design of Rust.

1 This is a programmer specified conversion which uses the `Display` trait, which works a bit like `toString` in Java. You can also use `{:?}` which gives a compiler generated representation which is sometimes useful for debugging. As with printf, there are many other options.

2 We don't actually use that argument in `foo`. Usually, Rust will warn us about this. By prefixing the argument name with `_` we avoid these warnings. In fact, we don't need to name the argument at all, we could just use `_`.

Control flow

If

The `if` statement is pretty much the same in Rust as C++. One difference is that the braces are mandatory, but parentheses around the expression being tested are not. Another is that `if` is an expression, so you can use it the same way as the ternary `?:` operator in C++ (remember from the previous section that if the last expression in a block is not terminated by a semi-colon, then it becomes the value of the block). There is no ternary `?:` in Rust. So, the following two functions do the same thing:

```rust
fn foo(x: i32) -> &'static str {
    let result: &'static str;
    if x < 10 {
        result = "less than 10";
    } else {
        result = "10 or more";
    }
    return result;
}

fn bar(x: i32) -> &'static str {
    if x < 10 {
        "less than 10"
    } else {
        "10 or more"
    }
}
```

(Why not `mut result`? The code in `foo` makes `result` immutable, it's just initialized in two possible places. Rust can see that by the time of `return result`, it is guaranteed to have been initialized.)

The first is a fairly literal translation of what you might write in C++. The second is better Rust style.

You can also write `let x = if ...`, etc.

Loops

Rust has while loops, again just like C++:

```rust
fn main() {
    let mut x = 10;
    while x > 0 {
        println!("Current value: {}", x);
        x -= 1;
    }
}
```

There is no `do...while` loop in Rust, but there is the `loop` statement which just loops forever:

```rust
fn main() {
    loop {
        println!("Just looping");
    }
}
```

Rust has `break` and `continue` just like C++.

For loops

Rust also has `for` loops, but these are a bit different. Lets say you have a vector of integers and you want to print them all (we'll cover vectors/arrays, iterators, and generics in more detail in the future. For now, know that a `Vec<T>` is a sequence of `T`s and `iter()` returns an iterator from anything you might reasonably want to iterate over). A simple `for` loop would look like:

```
fn print_all(all: Vec<i32>) {
    for a in all.iter() {
        println!("{}", a);
    }
}
```

TODO also &all/all instead of all.iter()

If we want to index over the indices of `all` (a bit more like a standard C++ for loop over an array), you could do

```
fn print_all(all: Vec<i32>) {
    for i in 0..all.len() {
        println!("{}: {}", i, all[i]);
    }
}
```

Hopefully, it is obvious what the `len` function does. TODO range notation

A more Rust-like equivalent of the preceding example would be to use an enumerating iterator:

```
fn print_all(all: Vec<i32>) {
    for (i, a) in all.iter().enumerate() {
        println!("{}: {}", i, a);
    }
}
```

Where `enumerate()` chains from the iterator `iter()` and yields the current count and the element during iteration.

The following example incorporates more advanced topics covered in the section on Borrowed Pointers. Let's say you have a vector of integers and want to call the function, passing the vector by reference and have the vector modified in place. Here the `for` loop uses a mutable iterator which gives mutable refererences - the `*` dereferencing should be familiar to C++ programmers:

```
fn double_all(all: &mut Vec<i32>) {
    for a in all.iter_mut() {
        *a += *a;
    }
}
```

Switch/Match

Rust has a match expression which is similar to a C++ switch statement, but much more powerful. This simple version should look pretty familiar:

```
fn print_some(x: i32) {
    match x {
        0 => println!("x is zero"),
        1 => println!("x is one"),
        10 => println!("x is ten"),
        y => println!("x is something else {}", y),
    }
}
```

There are some syntactic differences - we use `=>` to go from the matched value to the expression to execute, and the match arms are separated by `,` (that last `,` is optional). There are also some semantic differences which are not so obvious: the matched patterns must be exhaustive, that is all possible values of the matched expression (`x` in the above example) must be covered. Try removing the `y => ...` line and see what happens; that is because we only have matches for 0, 1, and 10, but there are obviously lots of other integers which don't get matched. In that last arm, `y` is bound to the value being matched (`x` in this case). We could also write:

```
fn print_some(x: i32) {
    match x {
        x => println!("x is something else {}", x)
    }
}
```

Here the `x` in the match arm introduces a new variable which hides the argument `x`, just like declaring a variable in an inner scope.

If we don't want to name the variable, we can use `_` for an unnamed variable, which is like having a wildcard match. If we don't want to do anything, we can provide an empty branch:

```
fn print_some(x: i32) {
    match x {
        0 => println!("x is zero"),
        1 => println!("x is one"),
        10 => println!("x is ten"),
        _ => {}
    }
}
```

Another semantic difference is that there is no fall through from one arm to the next so it works like `if...else if...else`.

We'll see in later posts that match is extremely powerful. For now I want to

introduce just a couple more features - the 'or' operator for values and `if` clauses on arms. Hopefully an example is self-explanatory:

```
fn print_some_more(x: i32) {
    match x {
        0 | 1 | 10 => println!("x is one of zero, one, or ten"),
        y if y < 20 => println!("x is less than 20, but not zero, one, or ten"),
        y if y == 200 => println!("x is 200 (but this is not very stylish)"),
        _ => {}
    }
}
```

Just like `if` expressions, `match` statements are actually expressions so we could re-write the last example as:

```
fn print_some_more(x: i32) {
    let msg = match x {
        0 | 1 | 10 => "one of zero, one, or ten",
        y if y < 20 => "less than 20, but not zero, one, or ten",
        y if y == 200 => "200 (but this is not very stylish)",
        _ => "something else"
    };

    println!("x is {}", msg);
}
```

Note the semi-colon after the closing brace, that is because the `let` statement is a statement and must take the form `let msg = ...;`. We fill the rhs with a match expression (which doesn't usually need a semi-colon), but the `let` statement does. This catches me out all the time.

Motivation: Rust match statements avoid the common bugs with C++ switch statements - you can't forget a `break` and unintentionally fall through; if you add a case to an enum (more later on) the compiler will make sure it is covered by your `match` statement.

Method call

Finally, just a quick note that methods exist in Rust, similarly to C++. They are always called via the `.` operator (no `->`, more on this in another post). We saw a few examples above (`len`, `iter`). We'll go into more detail in the future about how they are defined and called. Most assumptions you might make from C++ or Java are probably correct.

Primitive types and operators

Rust has pretty much the same arithmetic and logical operators as C++. `bool` is the same in both languages (as are the `true` and `false` literals). Rust has

similar concepts of integers, unsigned integers, and floats. However the syntax is a bit different. Rust uses `isize` to mean an integer and `usize` to mean an unsigned integer. These types are pointer sized. E.g., on a 32 bit system, `usize` means a 32 bit unsigned integer. Rust also has explicitly sized types which are `u` or `i` followed by 8, 16, 32, 64, or 128. So, for example, `u8` is an 8 bit unsigned integer and `i32` is a 32 bit signed integer. For floats, Rust has `f32` and `f64`.

Numeric literals can take suffixes to indicate their type. If no suffix is given, Rust tries to infer the type. If it can't infer, it uses `i32` or `f64` (if there is a decimal point). Examples:

```
fn main() {
    let x: bool = true;
    let x = 34;   // type i32
    // let x = 2147483648; // error: literal out of range for `i32`
    let x = 34isize;
    let x = 34usize;
    let x = 34u8;
    let x = 34i64;
    let x = 34f32;
}
```

As a side note, Rust lets you redefine variables so the above code is legal - each `let` statement creates a new variable `x` and hides the previous one. This is more useful than you might expect due to variables being immutable by default.

Numeric literals can be given as binary, octal, and hexadecimal, as well as decimal. Use the `0b`, `0o`, and `0x` prefixes, respectively. You can use an underscore anywhere in a numeric literal and it will be ignored. E.g,

```
fn main() {
    let x = 12;
    let x = 0b1100;
    let x = 0o14;
    let x = 0xe;
    let y = 0b_1100_0011_1011_0001;
}
```

Rust has chars and strings, but since they are Unicode, they are a bit different from C++. I'm going to postpone talking about them until after I've introduced pointers, references, and vectors (arrays).

Rust does not implicitly coerce numeric types. In general, Rust has much less implicit coercion and subtyping than C++. Rust uses the `as` keyword for explicit coercions and casting. Any numeric value can be cast to another numeric type. `as` cannot be used to convert from numeric types to boolean types, but the reverse can be done. E.g.,

```
fn main() {
    let x = 34usize as isize;   // cast usize to isize
```

```
    let x = 10 as f32;      // isize to float
    let x = 10.45f64 as i8; // float to i8 (loses precision)
    let x = 4u8 as u64;     // gains precision
    let x = 400u16 as u8;   // 144, loses precision (and thus changes the value)
    println!("`400u16 as u8` gives {}", x);
    let x = -3i8 as u8;     // 253, signed to unsigned (changes sign)
    println!("`-3i8 as u8` gives {}", x);
    //let x = 45 as bool;  // FAILS! (use 45 != 0 instead)
    let x = true as usize;  // cast bool to usize (gives a 1)
}}
```

Rust has the following operators:

Type	Operators
Numeric	`+`, `-`, `*`, `/`, `%`
Bitwise	`|`, `&`, `^`, `<<`, `>>`
Comparison	`==`, `!=`, `>`, `<`, `>=`, `<=`
Short-circuit logical	`||`, `&&`

All of these behave as in C++, however, Rust is a bit stricter about the types the operators can be applied to - the bitwise operators can only be applied to integers and the logical operators can only be applied to booleans. Rust has the `-` unary operator which negates a number. The `!` operator negates a boolean and inverts every bit on an integer type (equivalent to `~` in C++ in the latter case). Rust has compound assignment operators as in C++, e.g., `+=`, but does not have increment or decrement operators (e.g., `++`).

Unique pointers

Rust is a systems language and therefore must give you raw access to memory. It does this (as in C++) via pointers. Pointers are one area where Rust and C++ are very different, both in syntax and semantics. Rust enforces memory safety by type checking pointers. That is one of its major advantages over other languages. Although the type system is a bit complex, you get memory safety and bare-metal performance in return.

I had intended to cover all of Rust's pointers in one post, but I think the subject is too large. So this post will cover just one kind - unique pointers - and other kinds will be covered in follow up posts.

First, an example without pointers:

```
fn foo() {
    let x = 75;
```

```
    // ... do something with `x` ...
}
```

When we reach the end of `foo`, `x` goes out of scope (in Rust as in C++). That means the variable can no longer be accessed and the memory for the variable can be reused.

In Rust, for every type `T` we can write `Box<T>` for an owning (aka unique) pointer to `T`. We use `Box::new(...)` to allocate space on the heap and initialise that space with the supplied value. This is similar to `new` in C++. For example,

```
fn foo() {
    let x = Box::new(75);
}
```

Here `x` is a pointer to a location on the heap which contains the value `75`. `x` has type `Box<isize>`; we could have written `let x: Box<isize> = Box::new(75);`. This is similar to writing `int* x = new int(75);` in C++. Unlike in C++, Rust will tidy up the memory for us, so there is no need to call `free` or `delete`1. Unique pointers behave similarly to values - they are deleted when the variable goes out of scope. In our example, at the end of the function `foo`, `x` can no longer be accessed and the memory pointed at by `x` can be reused.

Owning pointers are dereferenced using the `*` as in C++. E.g.,

```
fn foo() {
    let x = Box::new(75);
    println!("`x` points to {}", *x);
}
```

As with primitive types in Rust, owning pointers and the data they point to are immutable by default. Unlike in C++, you can't have a mutable (unique) pointer to immutable data or vice versa. Mutability of the data follows from the pointer. E.g.,

```
fn foo() {
    let x = Box::new(75);
    let y = Box::new(42);
    // x = y;          // Not allowed, x is immutable.
    // *x = 43;        // Not allowed, *x is immutable.
    let mut x = Box::new(75);
    x = y;             // OK, x is mutable.
    *x = 43;           // OK, *x is mutable.
}
```

Owning pointers can be returned from a function and continue to live on. If they are returned, then their memory will not be freed, i.e., there are no dangling pointers in Rust. The memory will not leak. However, it will eventually go out of scope and then it will be freed. E.g.,

```
fn foo() -> Box<i32> {
    let x = Box::new(75);
    x
}

fn bar() {
    let y = foo();
    // ... use y ...
}
```

Here, memory is initialised in `foo`, and returned to `bar`. `x` is returned from `foo` and stored in `y`, so it is not deleted. At the end of `bar`, `y` goes out of scope and so the memory is reclaimed.

Owning pointers are unique (also called linear) because there can be only one (owning) pointer to any piece of memory at any time. This is accomplished by move semantics. When one pointer points at a value, any previous pointer can no longer be accessed. E.g.,

```
fn foo() {
    let x = Box::new(75);
    let y = x;
    // x can no longer be accessed
    // let z = *x;   // Error.
}
```

Likewise, if an owning pointer is passed to another function or stored in a field, it can no longer be accessed:

```
fn bar(y: Box<isize>) {
}

fn foo() {
    let x = Box::new(75);
    bar(x);
    // x can no longer be accessed
    // let z = *x;   // Error.
}
```

Rust's unique pointers are similar to C++ `std::unique_ptr`s. In Rust, as in C++, there can be only one unique pointer to a value and that value is deleted when the pointer goes out of scope. Rust does most of its checking statically rather than at runtime. So, in C++ accessing a unique pointer whose value has moved will result in a runtime error (since it will be null). In Rust this produces a compile time error and you cannot go wrong at runtime.

We'll see later that it is possible to create other pointer types which point at a unique pointer's value in Rust. This is similar to C++. However, in C++ this allows you to cause errors at runtime by holding a pointer to freed memory.

That is not possible in Rust (we'll see how when we cover Rust's other pointer types).

As shown above, owning pointers must be dereferenced to use their values. However, method calls automatically dereference, so there is no need for a `->` operator or to use `*` for method calls. In this way, Rust pointers are a bit similar to both pointers and references in C++. E.g.,

```
fn bar(x: Box<Foo>, y: Box<Box<Box<Box<Foo>>>>) {
    x.foo();
    y.foo();
}
```

Assuming that the type `Foo` has a method `foo()`, both these expressions are OK.

Calling `Box::new()` with an existing value does not take a reference to that value, it copies that value. So,

```
fn foo() {
    let x = 3;
    let mut y = Box::new(x);
    *y = 45;
    println!("x is still {}", x);
}
```

In general, Rust has move rather than copy semantics (as seen above with unique pointers). Primitive types have copy semantics, so in the above example the value `3` is copied, but for more complex values it would be moved. We'll cover this in more detail later.

Sometimes when programming, however, we need more than one reference to a value. For that, Rust has borrowed pointers. I'll cover those in the next post.

1 The `std::unique_ptr<T>`, introduced in C++11, is similar in some aspects to Rust's `Box<T>` but there are also significant differences.

Similarities: * The memory pointed to by a `std::unique_ptr<T>` in C++11 and a `Box<T>` in Rust is automatically released once the `std::unique_ptr<T>` goes out of the scope. * Both C++11's `std::unique_ptr<T>` and Rust's `Box<T>` only exhibit move semantics.

Differences:

1. C++11 allows for a `std::unique_ptr<T>` to be constructed from an existing pointer, thereby allowing multiple unique pointers to the same memory. This behaviour is not permitted with `Box<T>`.
2. Dereferencing a `std::unique_ptr<T>` that has been moved to another variable or function, causes undefined behavior in C++11. This would be caught at compile time in Rust.

3. Mutability or immutability does not go "through" `std::unique_ptr<T>` – dereferencing a `const std::unique_ptr<T>` still yields a mutable (non-`const`) reference to the underlying data. In Rust, an immutable `Box<T>` does not allow mutation of the data it points to.

`let x = Box::new(75)` in Rust may be interpreted as `const auto x = std::unique_ptr<const int>{new int{75}};` in C++11 and `const auto x = std::make_unique<const int>(75);` in C++14.

Borrowed pointers

In the last post I introduced unique pointers. This time I will talk about another kind of pointer which is much more common in most Rust programs: borrowed pointers (aka borrowed references, or just references).

If we want to have a reference to an existing value (as opposed to creating a new value on the heap and pointing to it, as with unique pointers), we must use `&`, a borrowed reference. These are probably the most common kind of pointer in Rust, and if you want something to fill in for a C++ pointer or reference (e.g., for passing a parameter to a function by reference), this is probably it.

We use the `&` operator to create a borrowed reference and to indicate reference types, and `*` to dereference them. The same rules about automatic dereferencing apply as for unique pointers. For example,

```
fn foo() {
    let x = &3;   // type: &i32
    let y = *x;   // 3, type: i32
    bar(x, *x);
    bar(&y, y);
}

fn bar(z: &i32, i: i32) {
    // ...
}
```

The `&` operator does not allocate memory (we can only create a borrowed reference to an existing value) and if a borrowed reference goes out of scope, no memory gets deleted.

Borrowed references are not unique - you can have multiple borrowed references pointing to the same value. E.g.,

```
fn foo() {
    let x = 5;                // type: i32
    let y = &x;               // type: &i32
    let z = y;                // type: &i32
    let w = y;                // type: &i32
```

```
    println!("These should all be 5: {} {} {}", *w, *y, *z);
}
```

Like values, borrowed references are immutable by default. You can also use `&mut` to take a mutable reference, or to denote mutable reference types. Mutable borrowed references are unique (you can only take a single mutable reference to a value, and you can only have a mutable reference if there are no immutable references). You can use a mutable reference where an immutable one is wanted, but not vice versa. Putting all that together in an example:

```
fn bar(x: &i32) { ... }
fn bar_mut(x: &mut i32) { ... }  // &mut i32 is a reference to an i32 which
                                 // can be mutated

fn foo() {
    let x = 5;
    //let xr = &mut x;      // Error - can't make a mutable reference to an
                            // immutable variable
    let xr = &x;            // Ok (creates an immutable ref)
    bar(xr);
    //bar_mut(xr);          // Error - expects a mutable ref

    let mut x = 5;
    let xr = &x;            // Ok (creates an immutable ref)
    //*xr = 4;              // Error - mutating immutable ref
    //let xr = &mut x;      // Error - there is already an immutable ref, so we
                            // can't make a mutable one

    let mut x = 5;
    let xr = &mut x;        // Ok (creates a mutable ref)
    *xr = 4;                // Ok
    //let xr = &x;          // Error - there is already a mutable ref, so we
                            // can't make an immutable one
    //let xr = &mut x;      // Error - can only have one mutable ref at a time
    bar(xr);                // Ok
    bar_mut(xr);            // Ok
}
```

Note that the reference may be mutable (or not) independently of the mutableness of the variable holding the reference. This is similar to C++ where pointers can be const (or not) independently of the data they point to. This is in contrast to unique pointers, where the mutableness of the pointer is linked to the mutableness of the data. For example,

```
fn foo() {
    let mut x = 5;
    let mut y = 6;
    let xr = &mut x;
```

```
    //xr = &mut y;        // Error xr is immutable

    let mut x = 5;
    let mut y = 6;
    let mut xr = &mut x;
    xr = &mut y;          // Ok

    let x = 5;
    let y = 6;
    let mut xr = &x;
    xr = &y;              // Ok - xr is mut, even though the referenced data is not
}
```

If a mutable value is borrowed, it becomes immutable for the duration of the borrow. Once the borrowed pointer goes out of scope, the value can be mutated again. This is in contrast to unique pointers, which once moved can never be used again. For example,

```
fn foo() {
    let mut x = 5;            // type: i32
    {
        let y = &x;           // type: &i32
        //x = 4;              // Error - x has been borrowed
        println!("{}", x);    // Ok - x can be read
    }
    x = 4;                    // OK - y no longer exists
}
```

The same thing happens if we take a mutable reference to a value - the value still cannot be modified. In general in Rust, data can only ever be modified via one variable or pointer. Furthermore, since we have a mutable reference, we can't take an immutable reference. That limits how we can use the underlying value:

```
fn foo() {
    let mut x = 5;            // type: i32
    {
        let y = &mut x;       // type: &mut i32
        //x = 4;              // Error - x has been borrowed
        //println!("{}", x);  // Error - requires borrowing x
    }
    x = 4;                    // OK - y no longer exists
}
```

Unlike C++, Rust won't automatically reference a value for you. So if a function takes a parameter by reference, the caller must reference the actual parameter. However, pointer types will automatically be converted to a reference:

```
fn foo(x: &i32) { ... }
```

```
fn bar(x: i32, y: Box<i32>) {
    foo(&x);
    // foo(x);   // Error - expected &i32, found i32
    foo(y);      // Ok
    foo(&*y);    // Also ok, and more explicit, but not good style
}
```

mut vs const

At this stage it is probably worth comparing `mut` in Rust to `const` in C++. Superficially they are opposites. Values are immutable by default in Rust and can be made mutable by using `mut`. Values are mutable by default in C++, but can be made constant by using `const`. The subtler and more important difference is that C++ const-ness applies only to the current use of a value, whereas Rust's immutability applies to all uses of a value. So in C++ if I have a `const` variable, someone else could have a non-const reference to it and it could change without me knowing. In Rust if you have an immutable variable, you are guaranteed it won't change.

As we mentioned above, all mutable variables are unique. So if you have a mutable value, you know it is not going to change unless you change it. Furthermore, you can change it freely since you know that no one else is relying on it not changing.

Borrowing and lifetimes

One of the primary safety goals of Rust is to avoid dangling pointers (where a pointer outlives the memory it points to). In Rust, it is impossible to have a dangling borrowed reference. It is only legal to create a borrowed reference to memory which will be alive longer than the reference (well, at least as long as the reference). In other words, the lifetime of the reference must be shorter than the lifetime of the referenced value.

That has been accomplished in all the examples in this post. Scopes introduced by `{}` or functions are bounds on lifetimes - when a variable goes out of scope its lifetime ends. If we try to take a reference to a shorter lifetime, such as in a narrower scope, the compiler will give us an error. For example,

```
fn foo() {
    let x = 5;
    let mut xr = &x;  // Ok - x and xr have the same lifetime
    {
        let y = 6;
        //xr = &y     // Error - xr will outlive y
    }                 // y is released here
}                     // x and xr are released here
```

In the above example, xr and y don't have the same lifetime because y starts later than xr, but it's the end of lifetimes which is more interesting, since you

can't reference a variable before it exists in any case - something else which Rust enforces and which makes it safer than C++.

Explicit lifetimes

After playing with borrowed pointers for a while, you'll probably come across borrowed pointers with an explicit lifetime. These have the syntax `&'a T` (cf. `&T`). They're kind of a big topic since I need to cover lifetime-polymorphism at the same time so I'll leave it for another post (there are a few more less common pointer types to cover first though). For now, I just want to say that `&T` is a shorthand for `&'a T` where `a` is the current scope, that is the scope in which the type is declared.

Reference counted and raw pointers

TODO add discussion of custom pointers and Deref trait (maybe later, not here)

So far we've covered unique and borrowed pointers. Unique pointers are very similar to the new std::unique_ptr in C++ and borrowed references are the 'default' pointer you usually reach for if you would use a pointer or reference in C++. Rust has a few more, rarer pointers either in the libraries or built in to the language. These are mostly similar to various kinds of smart pointers you might be used to in C++.

This post took a while to write and I still don't like it. There are a lot of loose ends here, both in my write up and in Rust itself. I hope some will get better with later posts and some will get better as the language develops. If you are learning Rust, you might even want to skip this stuff for now, hopefully you won't need it. Its really here just for completeness after the posts on other pointer types.

It might feel like Rust has a lot of pointer types, but it is pretty similar to C++ once you think about the various kinds of smart pointers available in libraries. In Rust, however, you are more likely to meet them when you first start learning the language. Because Rust pointers have compiler support, you are also much less likely to make errors when using them.

I'm not going to cover these in as much detail as unique and borrowed references because, frankly, they are not as important. I might come back to them in more detail later on.

Rc

Reference counted pointers come as part of the rust standard library. They are in the `std::rc` module (we'll cover modules soon-ish. The modules are the reason for the `use` incantations in the examples). A reference counted pointer to an object of type `T` has type `Rc<T>`. You create reference counted pointers

using a static method (which for now you can think of like C++'s, but we'll see later they are a bit different) - `Rc::new(...)` which takes a value to create the pointer to. This constructor method follows Rust's usual move/copy semantics (like we discussed for unique pointers) - in either case, after calling Rc::new, you will only be able to access the value via the pointer.

As with the other pointer types, the `.` operator does all the dereferencing you need it to. You can use `*` to manually dereference.

To pass a ref-counted pointer you need to use the `clone` method. This kinda sucks, and hopefully we'll fix that, but that is not for sure (sadly). You can take a (borrowed) reference to the pointed at value, so hopefully you don't need to clone too often. Rust's type system ensures that the ref-counted variable will not be deleted before any references expire. Taking a reference has the added advantage that it doesn't need to increment or decrement the ref count, and so will give better performance (although, that difference is probably marginal since Rc objects are limited to a single thread and so the ref count operations don't have to be atomic). As in C++, you can also take a reference to the Rc pointer.

An Rc example:

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

fn bar(x: Rc<i32>) { }
fn baz(x: &i32) { }

fn foo() {
    let x = Rc::new(45);
    bar(x.clone());   // Increments the ref-count
    baz(&*x);         // Does not increment
    println!("{}", 100 - *x);
}  // Once this scope closes, all Rc pointers are gone, so ref-count == 0
   // and the memory will be deleted.
```

Ref counted pointers are always immutable. If you want a mutable ref-counted object you need to use a RefCell (or Cell) wrapped in an `Rc`.

Cell and RefCell

Cell and RefCell are structs which allow you to 'cheat' the mutability rules. This is kind of hard to explain without first covering Rust data structures and how they work with mutability, so I'm going to come back to these slightly tricky objects later. For now, you should know that if you want a mutable, ref counted object you need a Cell or RefCell wrapped in an Rc. As a first approximation, you probably want Cell for primitive data and RefCell for objects with move semantics. So, for a mutable, ref-counted int you would use `Rc<Cell<int>>`.

*T - raw pointers

Finally, Rust has two kinds of raw pointers (aka unsafe pointers): `*const T` for an immutable raw pointer, and `*mut T` for a mutable raw pointer. They are created using `&` or `&mut` (you might need to specify a type to get a `*T` rather than a `&T` since the `&` operator can create either a borrowed reference or a raw pointer). Raw pointers are like C pointers, just a pointer to memory with no restrictions on how they are used (you can't do pointer arithmetic without casting, but you can do it that way if you must). Raw pointers are the only pointer type in Rust which can be null. There is no automatic dereferencing of raw pointers (so to call a method you have to write `(*x).foo()`) and no automatic referencing. The most important restriction is that they can't be dereferenced (and thus can't be used) outside of an unsafe block. In regular Rust code you can only pass them around.

So, what is unsafe code? Rust has strong safety guarantees, and (rarely) they prevent you doing something you need to do. Since Rust aims to be a systems language, it has to be able to do anything that is possible and sometimes that means doing things the compiler can't verify is safe. To accomplish that, Rust has the concept of unsafe blocks, marked by the `unsafe` keyword. In unsafe code you can do unsafe things - dereference a raw pointer, index into an array without bounds checking, call code written in another language via the FFI, or cast variables. Obviously, you have to be much more careful writing unsafe code than writing regular Rust code. In fact, you should only very rarely write unsafe code. Mostly it is used in very small chunks in libraries, rather than in client code. In unsafe code you must do all the things you normally do in C++ to ensure safety. Furthermore, you must manually ensure that you maintain the invariants which the compiler would usually enforce. Unsafe blocks allow you to manually enforce Rust's invariants, it does not allow you to break those invariants. If you do, you can introduce bugs both in safe and unsafe code.

An example of using an raw pointer:

```
fn foo() {
    let mut x = 5;
    let x_p: *mut i32 = &mut x;
    println!("x+5={}", add_5(x_p));
}

fn add_5(p: *mut i32) -> i32 {
    unsafe {
        if !p.is_null() { // Note that *-pointers do not auto-deref, so this is
                          // a method implemented on *i32, not i32.
            *p + 5
        } else {
            -1            // Not a recommended error handling strategy.
        }
```

```
    }
}
```

And that concludes our tour of Rust's pointers. Next time we'll take a break from pointers and look at Rust's data structures. We'll come back to borrowed references again in a later post though.

Destructuring

Last time we looked at Rust's data types. Once you have some data inside a structure, you will want to get that data out. For structs, Rust has field access, just like C++. For tuples, tuple structs, and enums you must use destructuring (there are various convenience functions in the library, but they use destructuring internally). Destructuring of data structures exists in C++ only since C++17, so it most likely familiar from languages such as Python or various functional languages. The idea is that just as you can initialize a data structure by filling out its fields with data from a bunch of local variables, you can fill out a bunch of local variables with data from a data structure. From this simple beginning, destructuring has become one of Rust's most powerful features. To put it another way, destructuring combines pattern matching with assignment into local variables.

Destructuring is done primarily through the let and match statements. The match statement is used when the structure being destructured can have different variants (such as an enum). A let expression pulls the variables out into the current scope, whereas match introduces a new scope. To compare:

```
fn foo(pair: (int, int)) {
    let (x, y) = pair;
    // we can now use x and y anywhere in foo

    match pair {
        (x, y) => {
            // x and y can only be used in this scope
        }
    }
}
```

The syntax for patterns (used after `let` and before `=>` in the above example) in both cases is (pretty much) the same. You can also use these patterns in argument position in function declarations:

```
fn foo((x, y): (int, int)) {
}
```

(Which is more useful for structs or tuple-structs than tuples).

Most initialisation expressions can appear in a destructuring pattern and they

can be arbitrarily complex. That can include references and primitive literals as well as data structures. For example,

```
struct St {
    f1: int,
    f2: f32
}

enum En {
    Var1,
    Var2,
    Var3(int),
    Var4(int, St, int)
}

fn foo(x: &En) {
    match x {
        &Var1 => println!("first variant"),
        &Var3(5) => println!("third variant with number 5"),
        &Var3(x) => println!("third variant with number {} (not 5)", x),
        &Var4(3, St { f1: 3, f2: x }, 45) => {
            println!("destructuring an embedded struct, found {} in f2", x)
        }
        &Var4(_, ref x, _) => {
            println!("Some other Var4 with {} in f1 and {} in f2", x.f1, x.f2)
        }
        _ => println!("other (Var2)")
    }
}
```

Note how we destructure through a reference by using `&` in the patterns and how we use a mix of literals (`5`, `3`, `St { ... }`), wildcards (`_`), and variables (`x`).

You can use `_` wherever a variable is expected if you want to ignore a single item in a pattern, so we could have used `&Var3(_)` if we didn't care about the integer. In the first `Var4` arm we destructure the embedded struct (a nested pattern) and in the second `Var4` arm we bind the whole struct to a variable. You can also use `..` to stand in for all fields of a tuple or struct. So if you wanted to do something for each enum variant but don't care about the content of the variants, you could write:

```
fn foo(x: En) {
    match x {
        Var1 => println!("first variant"),
        Var2 => println!("second variant"),
        Var3(..) => println!("third variant"),
        Var4(..) => println!("fourth variant")
    }
```

```
}
```

When destructuring structs, the fields don't need to be in order and you can use `..` to elide the remaining fields. E.g.,

```
struct Big {
    field1: int,
    field2: int,
    field3: int,
    field4: int,
    field5: int,
    field6: int,
    field7: int,
    field8: int,
    field9: int,
}

fn foo(b: Big) {
    let Big { field6: x, field3: y, ..} = b;
    println!("pulled out {} and {}", x, y);
}
```

As a shorthand with structs you can use just the field name which creates a local variable with that name. The let statement in the above example created two new local variables `x` and `y`. Alternatively, you could write

```
fn foo(b: Big) {
    let Big { field6, field3, .. } = b;
    println!("pulled out {} and {}", field3, field6);
}
```

Now we create local variables with the same names as the fields, in this case `field3` and `field6`.

There are a few more tricks to Rust's destructuring. Lets say you want a reference to a variable in a pattern. You can't use `&` because that matches a reference, rather than creates one (and thus has the effect of dereferencing the object). For example,

```
struct Foo {
    field: &'static int
}

fn foo(x: Foo) {
    let Foo { field: &y } = x;
}
```

Here, `y` has type `int` and is a copy of the field in `x`.

To create a reference to something in a pattern, you use the `ref` keyword. For

example,

```
fn foo(b: Big) {
    let Big { field3: ref x, ref field6, ..} = b;
    println!("pulled out {} and {}", *x, *field6);
}
```

Here, `x` and `field6` both have type `&int` and are references to the fields in `b`.

One last trick when destructuring is that if you are destructuring a complex object, you might want to name intermediate objects as well as individual fields. Going back to an earlier example, we had the pattern `&Var4(3, St{ f1: 3, f2: x }, 45)`. In that pattern we named one field of the struct, but you might also want to name the whole struct object. You could write `&Var4(3, s, 45)` which would bind the struct object to `s`, but then you would have to use field access for the fields, or if you wanted to only match with a specific value in a field you would have to use a nested match. That is not fun. Rust lets you name parts of a pattern using `@` syntax. For example `&Var4(3, s @ St{ f1: 3, f2: x }, 45)` lets us name both a field (`x`, for `f2`) and the whole struct (`s`).

That just about covers your options with Rust pattern matching. There are a few features I haven't covered, such as matching vectors, but hopefully you know how to use `match` and `let` and have seen some of the powerful things you can do. Next time I'll cover some of the subtle interactions between match and borrowing which tripped me up a fair bit when learning Rust.

Destructuring pt2 - match and borrowing

When destructuring there are some surprises in store where borrowing is concerned. Hopefully, nothing surprising once you understand borrowed references really well, but worth discussing (it took me a while to figure out, that's for sure. Longer than I realised, in fact, since I screwed up the first version of this blog post).

Imagine you have some `&Enum` variable `x` (where `Enum` is some enum type). You have two choices: you can match `*x` and list all the variants (`Variant1 => ...`, etc.) or you can match `x` and list reference to variant patterns (`&Variant1 => ...`, etc.). (As a matter of style, prefer the first form where possible since there is less syntactic noise). `x` is a borrowed reference and there are strict rules for how a borrowed reference can be dereferenced, these interact with match expressions in surprising ways (at least surprising to me), especially when you are modifying an existing enum in a seemingly innocuous way and then the compiler explodes on a match somewhere.

Before we get into the details of the match expression, lets recap Rust's rules for value passing. In C++, when assigning a value into a variable or passing it to a function there are two choices - pass-by-value and pass-by-reference. The former is the default case and means a value is copied either using a copy constructor

or a bitwise copy. If you annotate the destination of the parameter pass or assignment with `&`, then the value is passed by reference - only a pointer to the value is copied and when you operate on the new variable, you are also operating on the old value.

Rust has the pass-by-reference option, although in Rust the source as well as the destination must be annotated with `&`. For pass-by-value in Rust, there are two further choices - copy or move. A copy is the same as C++'s semantics (except that there are no copy constructors in Rust). A move copies the value but destroys the old value - Rust's type system ensures you can no longer access the old value. As examples, `i32` has copy semantics and `Box<i32>` has move semantics:

```
    fn foo() {
    let x = 7i;
    let y = x;                  // x is copied
    println!("x is {}", x);     // OK

    let x = box 7i;
    let y = x;                  // x is moved
    //println!("x is {}", x); // error: use of moved value: `x`
}
```

You can also choose to have copy semantics for user-defined types by implementing the `Copy` trait. One straightforward way to do that is to add `#[derive(Copy)]` before the definition of the `struct`. Not all user-defined types are allowed to implement the `Copy` trait. All fields of a type must implement `Copy` and the type must not have a destructor. Destructors probably need a post of their own, but for now, an object in Rust has a destructor if it implements the `Drop`trait. Just like C++, the destructor is executed just before an object is destroyed.

Now, it is important that a borrowed object is not moved, otherwise you would have a reference to the old object which is no longer valid. This is equivalent to holding a reference to an object which has been destroyed after going out of scope - it is a kind of dangling pointer. If you have a pointer to an object, there could be other references to it. So if an object has move semantics and you have a pointer to it, it is unsafe to dereference that pointer. (If the object has copy semantics, dereferencing creates a copy and the old object will still exist, so other references will be fine).

OK, back to match expressions. As I said earlier, if you want to match some `x` with type `&T` you can dereference once in the match clause or match the reference in every arm of the match expression. Example:

```
enum Enum1 {
    Var1,
    Var2,
    Var3
}
```

```
fn foo(x: &Enum1) {
    match *x {  // Option 1: deref here.
        Var1 => {}
        Var2 => {}
        Var3 => {}
    }

    match x {
        // Option 2: 'deref' in every arm.
        &Var1 => {}
        &Var2 => {}
        &Var3 => {}
    }
}
```

In this case you can take either approach because `Enum1` has copy semantics. Let's take a closer look at each approach: in the first approach we dereference `x` to a temporary variable with type `Enum1` (which copies the value in `x`) and then do a match against the three variants of `Enum1`. This is a 'one level' match because we don't go deep into the value's type. In the second approach there is no dereferencing. We match a value with type `&Enum1` against a reference to each variant. This match goes two levels deep - it matches the type (always a reference) and looks inside the type to match the referred type (which is `Enum1`).

Either way, we must ensure that we (that is, the compiler) respect Rust's invariants around moves and references - we must not move any part of an object if it is referenced. If the value being matched has copy semantics, that is trivial. If it has move semantics then we must make sure that moves don't happen in any match arm. This is accomplished either by ignoring data which would move, or making references to it (so we get by-reference passing rather than by-move).

```
enum Enum2 {
    // Box has a destructor so Enum2 has move semantics.
    Var1(Box<i32>),
    Var2,
    Var3
}

fn foo(x: &Enum2) {
    match *x {
        // We're ignoring nested data, so this is OK
        Var1(..) => {}
        // No change to the other arms.
        Var2 => {}
        Var3 => {}
    }
```

```
    match x {
        // We're ignoring nested data, so this is OK
        &Var1(..) => {}
        // No change to the other arms.
        &Var2 => {}
        &Var3 => {}
    }
}
```

In either approach we don't refer to any of the nested data, so none of it is moved. In the first approach, even though `x` is referenced, we don't touch its innards in the scope of the dereference (i.e., the match expression) so nothing can escape. We also don't bind the whole value (i.e., bind `*x` to a variable), so we can't move the whole object either.

We can take a reference to any variant in the second match, but not in the dereferenced version. So, in the second approach replacing the second arm with `a @ &Var2 => {}` is OK (`a` is a reference), but under the first approach we couldn't write `a @ Var2 => {}` since that would mean moving `*x` into `a`. We could write `ref a @ Var2 => {}` (in which `a` is also a reference), although it's not a construct you see very often.

But what about if we want to use the data nested inside `Var1`? We can't write:

```
match *x {
    Var1(y) => {}
    _ => {}
}
```

or

```
match x {
    &Var1(y) => {}
    _ => {}
}
```

because in both cases it means moving part of `x` into `y`. We can use the 'ref' keyword to get a reference to the data in `Var1`: `&Var1(ref y) => {}`. That is OK, because now we are not dereferencing anywhere and thus not moving any part of `x`. Instead we are creating a pointer which points into the interior of `x`.

Alternatively, we could destructure the Box (this match is going three levels deep): `&Var1(box y) => {}`. This is OK because `i32` has copy semantics and `y` is a copy of the `i32` inside the `Box` inside `Var1` (which is 'inside' a borrowed reference). Since `i32` has copy semantics, we don't need to move any part of `x`. We could also create a reference to the int rather than copy it: `&Var1(box ref y) => {}`. Again, this is OK, because we don't do any dereferencing and thus don't need to move any part of `x`. If the contents of the Box had move semantics, then we could not write `&Var1(box y) => {}`, we would be forced

to use the reference version. We could also use similar techniques with the first approach to matching, which look the same but without the first `&`. For example, `Var1(box ref y) => {}`.

Now lets get more complex. Lets say you want to match against a pair of reference-to-enum values. Now we can't use the first approach at all:

```
fn bar(x: &Enum2, y: &Enum2) {
    // Error: x and y are being moved.
    // match (*x, *y) {
    //     (Var2, _) => {}
    //     _ => {}
    // }

    // OK.
    match (x, y) {
        (&Var2, _) => {}
        _ => {}
    }
}
```

The first approach is illegal because the value being matched is created by dereferencing `x` and `y` and then moving them both into a new tuple object. So in this circumstance, only the second approach works. And of course, you still have to follow the rules above for avoiding moving parts of `x` and `y`.

If you do end up only being able to get a reference to some data and you need the value itself, you have no option except to copy that data. Usually that means using `clone()`. If the data doesn't implement clone, you're going to have to further destructure to make a manual copy or implement clone yourself.

What if we don't have a reference to a value with move semantics, but the value itself. Now moves are OK, because we know no one else has a reference to the value (the compiler ensures that if they do, we can't use the value). For example,

```
fn baz(x: Enum2) {
    match x {
        Var1(y) => {}
        _ => {}
    }
}
```

There are still a few things to be aware of. Firstly, you can only move to one place. In the above example we are moving part of `x` into `y` and we'll forget about the rest. If we wrote `a @ Var1(y) => {}` we would be attempting to move all of `x` into `a` and part of `x` into `y`. That is not allowed, an arm like that is illegal. Making one of `a` or `y` a reference (using `ref a`, etc.) is not an option either, then we'd have the problem described above where we move whilst holding a reference. We can make both `a` and `y` references and then we're OK -

neither is moving, so `x` remains intact and we have pointers to the whole and a part of it.

Similarly (and more common), if we have a variant with multiple pieces of nested data, we can't take a reference to one datum and move another. For example if we had a `Var4` declared as `Var4(Box<int>, Box<int>)` we can have a match arm which references both (`Var4(ref y, ref z) => {}`) or a match arm which moves both (`Var4(y, z) => {}`) but you cannot have a match arm which moves one and references the other (`Var4(ref y, z) => {}`). This is because a partial move still destroys the whole object, so the reference would be invalid.

Arrays and Vectors

Rust arrays are pretty different from C arrays. For starters they come in statically and dynamically sized flavours. These are more commonly known as fixed length arrays and slices. As we'll see, the former is kind of a bad name since both kinds of array have fixed (as opposed to growable) length. For a growable 'array', Rust provides the `Vec` collection.

Fixed length arrays

The length of a fixed length array is known statically and features in its type. E.g., `[i32; 4]` is the type of an array of `i32`s with length four.

Array literal and array access syntax is the same as C:

```
let a: [i32; 4] = [1, 2, 3, 4];     // As usual, the type annotation is optional.
println!("The second element is {}", a[1]);
```

You'll notice that array indexing is zero-based, just like C.

However, unlike C/C++1, array indexing is bounds checked. In fact all access to arrays is bounds checked, which is another way Rust is a safer language.

If you try to do `a[4]`, then you will get a runtime panic. Unfortunately, the Rust compiler is not clever enough to give you a compile time error, even when it is obvious (as in this example).

If you like to live dangerously, or just need to get every last ounce of performance out of your program, you can still get unchecked access to arrays. To do this, use the `get_unchecked` method on an array. Unchecked array accesses must be inside an unsafe block. You should only need to do this in the rarest circumstances.

Just like other data structures in Rust, arrays are immutable by default and mutability is inherited. Mutation is also done via the indexing syntax:

```
let mut a = [1, 2, 3, 4];
a[3] = 5;
println!("{:?}", a);
```

And just like other data, you can borrow an array by taking a reference to it:

```
fn foo(a: &[i32; 4]) {
    println!("First: {}; last: {}", a[0], a[3]);
}

fn main() {
    foo(&[1, 2, 3, 4]);
}
```

Notice that indexing still works on a borrowed array.

This is a good time to talk about the most interesting aspect of Rust arrays for C++ programmers - their representation. Rust arrays are value types: they are allocated on the stack like other values and an array object is a sequence of values, not a pointer to those values (as in C). So from our examples above, `let a = [1_i32, 2, 3, 4];` will allocate 16 bytes on the stack and executing `let b = a;` will copy 16 bytes. If you want a C-like array, you have to explicitly make a pointer to the array, this will give you a pointer to the first element.

A final point of difference between arrays in Rust and C++ is that Rust arrays can implement traits, and thus have methods. To find the length of an array, for example, you use `a.len()`.

Slices

A slice in Rust is just an array whose length is not known at compile time. The syntax of the type is just like a fixed length array, except there is no length: e.g., `[i32]` is a slice of 32 bit integers (with no statically known length).

There is a catch with slices: since the compiler must know the size of all objects in Rust, and it can't know the size of a slice, then we can never have a value with slice type. If you try and write `fn foo(x: [i32])`, for example, the compiler will give you an error.

So, you must always have pointers to slices (there are some very technical exceptions to this rule so that you can implement your own smart pointers, but you can safely ignore them for now). You must write `fn foo(x: &[i32])` (a borrowed reference to a slice) or `fn foo(x: *mut [i32])` (a mutable raw pointer to a slice), etc.

The simplest way to create a slice is by coercion. There are far fewer implicit coercions in Rust than there are in C++. One of them is the coercion from fixed length arrays to slices. Since slices must be pointer values, this is effectively a coercion between pointers. For example, we can coerce `&[i32; 4]` to `&[i32]`, e.g.,

```
let a: &[i32] = &[1, 2, 3, 4];
```

Here the right hand side is a fixed length array of length four, allocated on the stack. We then take a reference to it (type `&[i32; 4]`). That reference is coerced to type `&[i32]` and given the name `a` by the let statement.

Again, access is just like C (using `[...]`), and access is bounds checked. You can also check the length yourself by using `len()`. So clearly the length of the array is known somewhere. In fact all arrays of any kind in Rust have known length, since this is essential for bounds checking, which is an integral part of memory safety. The size is known dynamically (as opposed to statically in the case of fixed length arrays), and we say that slice types are dynamically sized types (DSTs, there are other kinds of dynamically sized types too, they'll be covered elsewhere).

Since a slice is just a sequence of values, the size cannot be stored as part of the slice. Instead it is stored as part of the pointer (remember that slices must always exist as pointer types). A pointer to a slice (like all pointers to DSTs) is a fat pointer - it is two words wide, rather than one, and contains the pointer to the data plus a payload. In the case of slices, the payload is the length of the slice.

So in the example above, the pointer `a` will be 128 bits wide (on a 64 bit system). The first 64 bits will store the address of the `1` in the sequence `[1, 2, 3, 4]`, and the second 64 bits will contain `4`. Usually, as a Rust programmer, these fat pointers can just be treated as regular pointers. But it is good to know about (it can affect the things you can do with casts, for example).

Slicing notation and ranges

A slice can be thought of as a (borrowed) view of an array. So far we have only seen a slice of the whole array, but we can also take a slice of part of an array. There is a special notation for this which is like the indexing syntax, but takes a range instead of a single integer. E.g., `a[0..4]`, which takes a slice of the first four elements of `a`. Note that the range is exclusive at the top and inclusive at the bottom. Examples:

```
let a: [i32; 4] = [1, 2, 3, 4];
let b: &[i32] = &a;   // Slice of the whole array.
let c = &a[0..4];     // Another slice of the whole array, also has type &[i32].
let c = &a[1..3];     // The middle two elements, &[i32].
let c = &a[1..];      // The last three elements.
let c = &a[..3];      // The first three elements.
let c = &a[..];       // The whole array, again.
let c = &b[1..3];     // We can also slice a slice.
```

Note that in the last example, we still need to borrow the result of slicing. The slicing syntax produces an unborrowed slice (type: `[i32]`) which we must then borrow (to give a `&[i32]`), even if we are slicing a borrowed slice.

Range syntax can also be used outside of slicing syntax. `a..b` produces an

iterator which runs from `a` to `b-1`. This can be combined with other iterators in the usual way, or can be used in `for` loops:

```
// Print all numbers from 1 to 10.
for i in 1..11 {
    println!("{}", i);
}
```

Vecs

A vector is heap allocated and is an owning reference. Therefore (and like `Box<_>`), it has move semantics. We can think of a fixed length array analogously to a value, a slice to a borrowed reference. Similarly, a vector in Rust is analogous to a `Box<_>` pointer.

It helps to think of `Vec<_>` as a kind of smart pointer, just like `Box<_>`, rather than as a value itself. Similarly to a slice, the length is stored in the 'pointer', in this case the 'pointer' is the Vec value.

A vector of `i32`s has type `Vec<i32>`. There are no vector literals, but we can get the same effect by using the `vec!` macro. We can also create an empty vector using `Vec::new()`:

```
let v = vec![1, 2, 3, 4];       // A Vec<i32> with length 4.
let v: Vec<i32> = Vec::new();   // An empty vector of i32s.
```

In the second case above, the type annotation is necessary so the compiler can know what the vector is a vector of. If we were to use the vector, the type annotation would probably not be necessary.

Just like arrays and slices, we can use indexing notation to get a value from the vector (e.g., `v[2]`). Again, these are bounds checked. We can also use slicing notation to take a slice of a vector (e.g., `&v[1..3]`).

The extra feature of vectors is that their size can change - they can get longer or shorter as needed. For example, `v.push(5)` would add the element `5` to the end of the vector (this would require that `v` is mutable). Note that growing a vector can cause reallocation, which for large vectors can mean a lot of copying. To guard against this you can pre-allocate space in a vector using `with_capacity`, see the Vec docs for more details.

The `Index` traits

Note for readers: there is a lot of material in this section that I haven't covered properly yet. If you're following the tutorial, you can skip this section, it is a somewhat advanced topic in any case.

The same indexing syntax used for arrays and vectors is also used for other collections, such as `HashMap`s. And you can use it yourself for your own collections. You opt-in to using the indexing (and slicing) syntax by implementing the `Index`

trait. This is a good example of how Rust makes available nice syntax to user types, as well as built-ins (`Deref` for dereferencing smart pointers, as well as `Add` and various other traits, work in a similar way).

The `Index` trait looks like

```
pub trait Index<Idx: ?Sized> {
    type Output: ?Sized;

    fn index(&self, index: Idx) -> &Self::Output;
}
```

`Idx` is the type used for indexing. For most uses of indexing this is `usize`. For slicing this is one of the `std::ops::Range` types. `Output` is the type returned by indexing, this will be different for each collection. For slicing it will be a slice, rather than the type of a single element. `index` is a method which does the work of getting the element(s) out of the collection. Note that the collection is taken by reference and the method returns a reference to the element with the same lifetime.

Let's look at the implementation for `Vec` to see how what an implementation looks like:

```
impl<T> Index<usize> for Vec<T> {
    type Output = T;

    fn index(&self, index: usize) -> &T {
        &(**self)[index]
    }
}
```

As we said above, indexing is done using `usize`. For a `Vec<T>`, indexing will return a single element of type `T`, thus the value of `Output`. The implementation of `index` is a bit weird - `(**self)` gets a view of the whole vec as a slice, then we use indexing on slices to get the element, and finally take a reference to it.

If you have your own collections, you can implement `Index` in a similar way to get indexing and slicing syntax for your collection.

Initialiser syntax

As with all data in Rust, arrays and vectors must be properly initialised. Often you just want an array full of zeros to start with and using the array literal syntax is a pain. So Rust gives you a little syntactic sugar to initialise an array full of a given value: `[value; len]`. So for example to create an array with length 100 full of zeros, we'd use `[0; 100]`.

Similarly for vectors, `vec![42; 100]` would give you a vector with 100 elements, each with the value 42.

The initial value is not limited to integers, it can be any expression. For array initialisers, the length must be an integer constant expression. For `vec!`, it can be any expression with type `usize`.

1 In C++11 there is `std::array<T, N>` that provides boundary checking when `at()` method is used.

Graphs and arena allocation

(Note you can run the examples in this chapter by downloading this directory and running `cargo run`).

Graphs are a bit awkward to construct in Rust because of Rust's stringent lifetime and mutability requirements. Graphs of objects are very common in OO programming. In this tutorial I'm going to go over a few different approaches to implementation. My preferred approach uses arena allocation and makes slightly advanced use of explicit lifetimes. I'll finish up by discussing a few potential Rust features which would make using such an approach easier.

A graph is a collection of nodes with edges between some of those nodes. Graphs are a generalisation of lists and trees. Each node can have multiple children and multiple parents (we usually talk about edges into and out of a node, rather than parents/children). Graphs can be represented by adjacency lists or adjacency matrices. The former is basically a node object for each node in the graph, where each node object keeps a list of its adjacent nodes. An adjacency matrix is a matrix of booleans indicating whether there is an edge from the row node to the column node. We'll only cover the adjacency list representation, adjacency matrices have very different issues which are less Rust-specific.

There are essentially two orthogonal problems: how to handle the lifetime of the graph and how to handle it's mutability.

The first problem essentially boils down to what kind of pointer to use to point to other nodes in the graph. Since graph-like data structures are recursive (the types are recursive, even if the data is not) we are forced to use pointers of some kind rather than have a totally value-based structure. Since graphs can be cyclic, and ownership in Rust cannot be cyclic, we cannot use `Box<Node>` as our pointer type (as we might do for tree-like data structures or linked lists).

No graph is truly immutable. Because there may be cycles, the graph cannot be created in a single statement. Thus, at the very least, the graph must be mutable during its initialisation phase. The usual invariant in Rust is that all pointers must either be unique or immutable. Graph edges must be mutable (at least during initialisation) and there can be more than one edge into any node, thus no edges are guaranteed to be unique. So we're going to have to do something a little bit advanced to handle mutability.

One solution is to use mutable raw pointers (`*mut Node`). This is the most flexible approach, but also the most dangerous. You must handle all the lifetime management yourself without any help from the type system. You can make very flexible and efficient data structures this way, but you must be very careful. This approach handles both the lifetime and mutability issues in one fell swoop. But it handles them by essentially ignoring all the benefits of Rust - you will get no help from the compiler here (it's also not particularly ergonomic since raw pointers don't automatically (de-)reference). Since a graph using raw pointers is not much different from a graph in C++, I'm not going to cover that option here.

The options you have for lifetime management are reference counting (shared ownership, using `Rc<...>`) or arena allocation (all nodes have the same lifetime, managed by an arena; using borrowed references `&...`). The former is more flexible (you can have references from outside the graph to individual nodes with any lifetime), the latter is better in every other way.

For managing mutability, you can either use `RefCell`, i.e., make use of Rust's facility for dynamic, interior mutability, or you can manage the mutability yourself (in this case you have to use `UnsafeCell` to communicate the interior mutability to the compiler). The former is safer, the latter is more efficient. Neither is particularly ergonomic.

Note that if your graph might have cycles, then if you use `Rc`, further action is required to break the cycles and not leak memory. Since Rust has no cycle collection of `Rc` pointers, if there is a cycle in your graph, the ref counts will never fall to zero, and the graph will never be deallocated. You can solve this by using `Weak` pointers in your graph or by manually breaking cycles when you know the graph should be destroyed. The former is more reliable. We don't cover either here, in our examples we just leak memory. The approach using borrowed references and arena allocation does not have this issue and is thus superior in that respect.

To compare the different approaches I'll use a pretty simple example. We'll just have a `Node` object to represent a node in the graph, this will hold some string data (representative of some more complex data payload) and a `Vec` of adjacent nodes (`edges`). We'll have an `init` function to create a simple graph of nodes, and a `traverse` function which does a pre-order, depth-first traversal of the graph. We'll use this to print the payload of each node in the graph. Finally, we'll have a `Node::first` method which returns a reference to the first adjacent node to the `self` node and a function `foo` which prints the payload of an individual node. These functions stand in for more complex operations involving manipulation of a node interior to the graph.

To try and be as informative as possible without boring you, I'll cover two combinations of possibilities: ref counting and `RefCell`, and arena allocation and `UnsafeCell`. I'll leave the other two combinations as an exercise.

`Rc<RefCell<Node>>`

See full example.

This is the safer option because there is no unsafe code. It is also the least efficient and least ergonomic option. It is pretty flexible though, nodes of the graph can be easily reused outside the graph since they are ref-counted. I would recommend this approach if you need a fully mutable graph, or need your nodes to exist independently of the graph.

The node structure looks like

```
struct Node {
    datum: &'static str,
    edges: Vec<Rc<RefCell<Node>>>,
}
```

Creating a new node is not too bad: `Rc::new(RefCell::new(Node { ... }))`. To add an edge during initialisation, you have to borrow the start node as mutable, and clone the end node into the Vec of edges (this clones the pointer, incrementing the reference count, not the actual node). E.g.,

```
let mut mut_root = root.borrow_mut();
mut_root.edges.push(b.clone());
```

The `RefCell` dynamically ensures that we are not already reading or writing the node when we write it.

Whenever you access a node, you have to use `.borrow()` to borrow the `RefCell`. Our `first` method has to return a ref-counted pointer, rather than a borrowed reference, so callers of `first` also have to borrow:

```
fn first(&self) -> Rc<RefCell<Node>> {
    self.edges[0].clone()
}

pub fn main() {
    let g = ...;
    let f = g.first();
    foo(&*f.borrow());
}
```

`&Node` and `UnsafeCell`

See full example.

In this approach we use borrowed references as edges. This is nice and ergonomic and lets us use our nodes with 'regular' Rust libraries which primarily operate with borrowed references (note that one nice thing about ref counted objects in Rust is that they play nicely with the lifetime system. We can create a borrowed reference into the `Rc` to directly (and safely) reference the data. In the previous

example, the `RefCell` prevents us doing this, but an `Rc`/`UnsafeCell` approach should allow it).

Destruction is correctly handled too - the only constraint is that all the nodes must be destroyed at the same time. Destruction and allocation of nodes is handled using an arena.

On the other hand, we do need to use quite a few explicit lifetimes. Unfortunately we don't benefit from lifetime elision here. At the end of the section I'll discuss some future directions for the language which could make things better.

During construction we will mutate our nodes which might be multiply referenced. This is not possible in safe Rust code, so we must initialise inside an `unsafe` block. Since our nodes are mutable and multiply referenced, we must use an `UnsafeCell` to communicate to the Rust compiler that it cannot rely on its usual invariants.

When is this approach feasible? The graph must only be mutated during initialisation. In addition, we require that all nodes in the graph have the same lifetime (we could relax these constraints somewhat to allow adding nodes later as long as they can all be destroyed at the same time). Similarly, we could rely on more complicated invariants for when the nodes can be mutated, but it pays to keep things simple, since the programmer is responsible for safety in those respects.

Arena allocation is a memory management technique where a set of objects have the same lifetime and can be deallocated at the same time. An arena is an object responsible for allocating and deallocating the memory. Since large chunks of memory are allocated and deallocated at once (rather than allocating individual objects), arena allocation is very efficient. Usually, all the objects are allocated from a contiguous chunk of memory, that improves cache coherency when you are traversing the graph.

In Rust, arena allocation is supported by the libarena crate and is used throughout the compiler. There are two kinds of arenas - typed and untyped. The former is more efficient and easier to use, but can only allocate objects of a single type. The latter is more flexible and can allocate any object. Arena allocated objects all have the same lifetime, which is a parameter of the arena object. The type system ensures references to arena allocated objects cannot live longer than the arena itself.

Our node struct must now include the lifetime of the graph, `'a`. We wrap our `Vec` of adjacent nodes in an `UnsafeCell` to indicate that we will mutate it even when it should be immutable:

```
struct Node<'a> {
    datum: &'static str,
    edges: UnsafeCell<Vec<&'a Node<'a>>>,
}
```

Our new function must also use this lifetime and must take as an argument the arena which will do the allocation:

```
fn new<'a>(datum: &'static str, arena: &'a TypedArena<Node<'a>>) -> &'a Node<'a> {
    arena.alloc(Node {
        datum: datum,
        edges: UnsafeCell::new(Vec::new()),
    })
}
```

We use the arena to allocate the node. The lifetime of the graph is derived from the lifetime of the reference to the arena, so the arena must be passed in from the scope which covers the graph's lifetime. For our examples, that means we pass it into the `init` method. (One could imagine an extension to the type system which allows creating values at scopes outside their lexical scope, but there are no plans to add such a thing any time soon). When the arena goes out of scope, the whole graph is destroyed (Rust's type system ensures that we can't keep references to the graph beyond that point).

Adding an edge is a bit different looking:

```
(*root.edges.get()).push(b);
```

We're essentially doing the obvious `root.edges.push(b)` to push a node (`b`) on to the list of edges. However, since `edges` is wrapped in an `UnsafeCell`, we have to call `get()` on it. That gives us a mutable raw pointer to edges (`*mut Vec<&Node>`), which allows us to mutate `edges`. However, it also requires us to manually dereference the pointer (raw pointers do not auto-deref), thus the `(*...)` construction. Finally, dereferencing a raw pointer is unsafe, so the whole lot has to be wrapped up in an unsafe block.

The interesting part of `traverse` is:

```
for n in &(*self.edges.get()) {
    n.traverse(f, seen);
}
```

We follow the previous pattern for getting at the edges list, which requires an unsafe block. In this case we know it is in fact safe because we must be post-initialisation and thus there will be no mutation.

Again, the `first` method follows the same pattern for getting at the `edges` list. And again must be in an unsafe block. However, in contrast to the graph using `Rc<RefCell<_>>`, we can return a straightforward borrowed reference to the node. That is very convenient. We can reason that the unsafe block is safe because we do no mutation and we are post-initialisation.

```
fn first(&'a self) -> &'a Node<'a> {
    unsafe {
        (*self.edges.get())[0]
```

```
    }
}
```

Future language improvements for this approach

I believe that arena allocation and using borrowed references are an important pattern in Rust. We should do more in the language to make these patterns safer and easier to use. I hope use of arenas becomes more ergonomic with the ongoing work on allocators. There are three other improvements I see:

Safe initialisation There has been lots of research in the OO world on mechanisms for ensuring mutability only during initialisation. How exactly this would work in Rust is an open research question, but it seems that we need to represent a pointer which is mutable and not unique, but restricted in scope. Outside that scope any existing pointers would become normal borrowed references, i.e., immutable *or* unique.

The advantage of such a scheme is that we have a way to represent the common pattern of mutable during initialisation, then immutable. It also relies on the invariant that, while individual objects are multiply owned, the aggregate (in this case a graph) is uniquely owned. We should then be able to adopt the reference and `UnsafeCell` approach, without the `UnsafeCell`s and the unsafe blocks, making that approach more ergonomic and more safer.

Alex Summers and Julian Viereck at ETH Zurich are investigating this further.

Generic modules The 'lifetime of the graph' is constant for any particular graph. Repeating the lifetime is just boilerplate. One way to make this more ergonomic would be to allow the graph module to be parameterised by the lifetime, so it would not need to be added to every struct, impl, and function. The lifetime of the graph would still need to be specified from outside the module, but hopefully inference would take care of most uses (as it does today for function calls).

See ref_graph_generic_mod.rs for how that might look. (We should also be able to use safe initialisation (proposed above) to remove the unsafe code).

See also this RFC issue.

This feature would vastly reduce the syntactic overhead of the reference and `UnsafeCell` approach.

Lifetime elision We currently allow the programmer to elide some lifetimes in function signatures to improve ergonomics. One reason the `&Node` approach to graphs is a bit ugly is because it doesn't benefit from any of the lifetime elision rules.

A common pattern in Rust is data structures with a common lifetime. References into such data structures give rise to types like `&'a Foo<'a>`, for example `&'a`

`Node<'a>` in the graph example. It would be nice to have an elision rule that helps in this case. I'm not really sure how it should work though.

Looking at the example with generic modules, it doesn't look like we need to extend the lifetime elision rules very much (I'm not actually sure if `Node::new` would work without the given lifetimes, but it seems like a fairly trivial extension to make it work if it doesn't). We might want to add some new rule to allow elision of module-generic lifetimes if they are the only ones in scope (other than `'static`), but I'm not sure how that would work with multiple in- scope lifetimes (see the `foo` and `init` functions, for example).

If we don't add generic modules, we might still be able to add an elision rule specifically to target `&'a Node<'a>`, not sure how though.

Closures and first-class functions

Closures and first-class and higher order functions are a core part of Rust. In C and C++ there are function pointers (and those weird member/method pointer things in C++ that I never got the hang of). However, they are used relatively rarely and are not very ergonomic. C++11 introduced lambdas, and these are pretty close to Rust closures, in particular they have a very similar implementation strategy.

To start with, I want to establish some intuition for these things. Then, we'll dive in to the details.

Lets say we have a function `foo`: `pub fn foo() -> u32 { 42 }`. Now let's imagine another function `bar` which takes a function as an argument (I'll leave `bar`'s signature for later): `fn bar(f: ...) { ... }`. We can pass `foo` to `bar` kind of like we would pass a function pointer in C: `bar(foo)`. In the body of `bar` we can call `f` as if it were a function: `let x = f();`.

We say that Rust has first-class functions because we can pass them around and use them like we can with other values. We say `bar` is a higher-order function because it takes a function as an argument, i.e., it is a function that operates on functions.

Closures in Rust are anonymous functions with a nice syntax. A closure `|x| x + 2` takes an argument and returns it with `2` added. Note that we don't have to give types for the arguments to a closure (they can usually be inferred). We also don't need to specify a return type. If we want the closure body to be more than just one expression, we can use braces: `|x: i32| { let y = x + 2; y }`. We can pass closures just like functions: `bar(|| 42)`.

The big difference between closures and other functions is that closures capture their environment. This means that we can refer to variables outside the closure from the closure. E.g.,

```
let x = 42;
bar(|| x);
```

Note how `x` is in scope in the closure.

We've seen closures before, used with iterators, and this is a common use case for them. E.g., to add a value to each element of a vector:

```
fn baz(v: Vec<i32>) -> Vec<i32> {
    let z = 3;
    v.iter().map(|x| x + z).collect()
}
```

Here `x` is an argument to the closure, each member of `v` will be passed as an `x`. `z` is declared outside of the closure, but because it's a closure, `z` can be referred to. We could also pass a function to map:

```
fn add_two(x: i32) -> i32 {
    x + 2
}

fn baz(v: Vec<i32>) -> Vec<i32> {
    v.iter().map(add_two).collect()
}
```

Note that Rust also allows declaring functions inside of functions. These are *not* closures - they can't access their environment. They are merely a convenience for scoping.

```
fn qux(x: i32) {
    fn quxx() -> i32 {
        x // ERROR x is not in scope.
    }

    let a = quxx();
}
```

Function types

Lets introduce a new example function:

```
fn add_42(x: i32) -> i64 {
    x as i64 + 42
}
```

As we saw before, we can store a function in a variable: `let a = add_42;`. The most precise type of `a` cannot be written in Rust. You'll sometimes see the compiler render it as `fn(i32) -> i64 {add_42}` in error messages. Each function has its own unique and anonymous type. `fn add_41(x: i32) -> i64` has a different type, even though it has the same signature.

We can write less precise types, for example, `let a: fn(i32) -> i64 = add_42;`. All function types with the same signature can be coerced to a `fn` type (which can be written by the programmer).

`a` is represented by the compiler as a function pointer, however, if the compiler knows the precise type, it doesn't actually use that function pointer. A call like a() is statically dispatched based on the type of a. If the compiler doesn't know the precise type (e.g., it only knows the fn type), then the call is dispatched using the function pointer in the value.

There are also `Fn` types (note the capital 'F'). These `Fn` types are bounds, just like traits (in fact they *are* traits, as we'll see later). `Fn(i32) -> i64` is a bound on the types of all function-like objects with that signature. When we take a reference to a function pointer, we're actually creating a trait object which is represented by a fat pointer (see DSTs).

To pass a function to another function, or to store the function in a field, we must write a type. We have several choices, we can either use either a `fn` type or a `Fn` type. The latter is better because it includes closures (and potentially other function-like things), whereas `fn` types don't. The `Fn` types are dynamically sized which means we cannot use them as value types. We must either pass function objects or use generics. Let's look at the generic approach first. For example,

```
fn bar<F>(f: F) -> i64
    where F: Fn(i32) -> i64
{
    f(0)
}
```

`bar` takes any function with the signature `Fn(i32) -> i64`, i.e., we can instantiate the `F` type parameter with any function-like type. We could call `bar(add_42)` to pass `add_42` to `bar` which would instantiate `F` with `add_42`'s anonymous type. We could also call `bar(add_41)` and that would work too.

You can also pass closures to `bar`, e.g., `bar(|x| x as i64)`. This works because closure types are also bounded by the `Fn` bound matching their signature (like functions, each closure has it's own anonymous type).

Finally, you can pass references to functions or closures too: `bar(&add_42)` or `bar(&|x| x as i64)`.

One could also write `bar` as `fn bar(f: &Fn(i32) -> i64) ...`. These two approaches (generics vs a function/trait object) have quite different semantics. In the generics case, `bar` will be monomorphised so when code is generated, the compiler know the exact type of `f`, that means it can be statically dispatched. If using a function object, the function is not monomorphised. The exact type of `f` is not known, and so the compiler must generate a virtual dispatch. The latter is slower, but the former will produce more code (one monomorphised function per type parameter instance).

There are actually more function traits than just `Fn`; there are `FnMut` and `FnOnce` too. These are used in the same way as `Fn`, e.g., `FnOnce(i32) -> i64`. A `FnMut` represents an object which can be called and can be mutated during that call. This doesn't apply to normal functions, but for closures it means the closure can mutate its environment. `FnOnce` is a function which can only be called (at most) once. Again, this is only relevant for closures.

`Fn`, `FnMut`, and `FnOnce` are in a sub-trait hierarchy. `Fn`s are `FnMut`s (because one can call a `Fn` function with permission to mutate and no harm is done, but the opposite is not true). `Fn`s and `FnMut`s are `FnOnce`s (because there is no harm done if a regular function is only called once, but not the opposite).

So, to make a higher-order function as flexible as possible, you should use the `FnOnce` bound, rather than the `Fn` bound (or use the `FnMut` bound if you must call the function more than once).

Methods

You can use methods in the same way as functions - take pointers to them store them in variables, etc. You can't use the dot syntax, you must explicitly name the method using the fully explicit form of naming (sometimes called UFCS for universal function call syntax). The `self` parameter is the first argument to the method. E.g.,

```
struct Foo;

impl Foo {
    fn bar(&self) {}
}

trait T {
    fn baz(&self);
}

impl T for Foo {
    fn baz(&self) {}
}

fn main() {
    // Inherent method.
    let x = Foo::bar;
    x(&Foo);

    // Trait method, note the fully explicit naming form.
    let y = <Foo as T>::baz;
    y(&Foo);
}
```

Generic functions

You can't take a pointer to a generic function and there is no way to express a generic function type. However, you can take a reference to a function if all its type parameters are instantiated. E.g.,

```
fn foo<T>(x: &T) {}

fn main() {
    let x = &foo::<i32>;
    x(&42);
}
```

There is no way to define a generic closure. If you need a closure to work over many types you can use trait objects, macros (for generating closures), or pass a closure which returns closures (each returned closure can operate on a different type).

Lifetime-generic functions and higher-ranked types

It *is* possible to have function types and closures which are generic over lifetimes.

Imagine we have a closure which takes a borrowed reference. The closure can work the same way no matter what lifetime the reference has (and indeed in the compiled code, the lifetime will have been erased). But, what does the type look like?

For example,

```
fn foo<F>(x: &Bar, f: F) -> &Baz
    where F: Fn(&Bar) -> &Baz
{
    f(x)
}
```

what are the lifetimes of the references here? In this simple example, we can use a single lifetime (no need for a generic closure):

```
fn foo<'b, F>(x: &'b Bar, f: F) -> &'b Baz
    where F: Fn(&'b Bar) -> &'b Baz
{
    f(x)
}
```

But what if we want `f` to work on inputs with different lifetimes? Then we need a generic function type:

```
fn foo<'b, 'c, F>(x: &'b Bar, y: &'c Bar, f: F) -> (&'b Baz, &'c Baz)
    where F: for<'a> Fn(&'a Bar) -> &'a Baz
{
```

```
    (f(x), f(y))
}
```

The novelty here is the `for<'a>` syntax, this is used to denote a function type which is generic over a lifetime. It is read "for all 'a, ...". In theoretical terms, the function type is universally quantified.

Note that we cannot hoist up `'a` to `foo` in the above example. Counter-example:

```
fn foo<'a, 'b, 'c, F>(x: &'b Bar, y: &'c Bar, f: F) -> (&'b Baz, &'c Baz)
    where F: Fn(&'a Bar) -> &'a Baz
{
    (f(x), f(y))
}
```

will not compile because when the compiler infers lifetimes for a call to `foo`, it must pick a single lifetime for `'a`, which it can't do if `'b` and `'c` are different.

A function type which is generic in this way is called a higher-ranked type. Lifetime variables at the outer level have rank one. Because `'a` in the above example cannot be moved to the outer level, it's rank is higher than one.

Calling functions with higher-ranked function type arguments is easy - the compiler will infer the lifetime parameters. E.g., `foo(&Bar { ... }, &Bar {...}, |b| &b.field)`.

In fact, most of the time you don't even need to worry about such things. The compiler will allow you to elide the quantified lifetimes in the same way that you are allowed to elide many lifetimes on function arguments. For example, the example above can just be written as

```
fn foo<'b, 'c, F>(x: &'b Bar, y: &'c Bar, f: F) -> (&'b Baz, &'c Baz)
    where F: Fn(&Bar) -> &Baz
{
    (f(x), f(y))
}
```

(and you only need `'b` and `'c` because it is a contrived example).

Where Rust sees a function type with a borrowed references, it will apply the usual elision rules, and quantify the elided variables at the scope of the function type (i.e., with higher rank).

You might be wondering why bother with all this complexity for what looks like a fairly niche use case. The real motivation is functions which take a function to operate on some data provided by the outer function. For example,

```
fn foo<F>(f: F)
    where F: Fn(&i32) // Fully explicit type: for<'a> Fn(&'a i32)
{
    let data = 42;
```

```
    f(&data)
}
```

In these cases, we *need* higher-ranked types. If we added a lifetime parameter to `foo` instead, we could never infer a correct lifetime. To see why, let's look at how it might work, consider `fn foo<'a, F: Fn(&'a i32')> ...`. Rust requires that any lifetime parameter must outlive the item it is declared on (if this were not the case, an argument with that lifetime could be used inside that function, where it is not guaranteed to be live). In the body of `foo` we use `f(&data)`, the lifetime Rust will infer for that reference will last (at most) from where `data` is declared to where it goes out of scope. Since `'a` must outlive `foo`, but that inferred lifetime does not, we cannot call `f` in this way.

However, with higher-ranked lifetimes `f` can accept any lifetime and so the anonymous one from `&data` is fine and the function type checks.

Enum constructors

This is something of a digression, but it is sometimes a useful trick. All variants of an enum define a function from the fields of the variant to the enum type. For example,

```
enum Foo {
    Bar,
    Baz(i32),
}
```

defines two functions, `Foo::Bar: Fn() -> Foo` and `Foo::Baz: Fn(i32) -> Foo`. We don't normally use the variants in this way, we treat them as data types rather than functions. But sometimes it is useful, for example if we have a list of `i32`s we can create a list of `Foo`s with

```
list_of_i32.iter().map(Foo::Baz).collect()
```

Closure flavours

A closure has two forms of input: the arguments which are passed to it explicitly and the variables it *captures* from its environment. Usually, everything about both kinds of input is inferred, but you can have more control if you want it.

For the arguments, you can declare types instead of letting Rust infer them. You can also declare a return type. Rather than writing `|x| { ... }` you can write `|x: i32| -> String { ... }`. Whether an argument is owned or borrowed is determined by the types (either declared or inferred).

For the captured variables, the type is mostly known from the environment, but Rust does a little extra magic. Should a variable be captured by reference or value? Rust infers this from the body of the closure. If possible, Rust captures by reference. E.g.,

```
fn foo(x: Bar) {
    let f = || { ... x ... };
}
```

All being well, in the body of `f`, `x` has the type `&Bar` with a lifetime bounded by the scope of `foo`. However, if `x` is mutated, then Rust will infer that the capture is by mutable reference, i.e., `x` has type `&mut Bar`. If `x` is moved in `f` (e.g., is stored into a variable or field with value type), then Rust infers that the variable must be captured by value, i.e., it has the type `Bar`.

This can be overridden by the programmer (sometimes necessary if the closure will be stored in a field or returned from a function). By using the `move` keyword in front of a closure. Then, all of the captured variables are captured by value. E.g., in `let f = move || { ... x ... };`, `x` would always have type `Bar`.

We talked earlier about the different function kinds: `Fn`, `FnMut`, and `FnOnce`. We can now explain why we need them. For closures, the mutable-ness and once-ness refer to the captured variables. If a capture mutates any of the variables it captures then it will have a `FnMut` type (note that this is completely inferred by the compiler, no annotation is necessary). If a variable is moved into a closure, i.e., it is captured by value (either because of an explicit `move` or due to inference), then the closure will have a `FnOnce` type. It would be unsafe to call such a closure multiple times because the captured variable would be moved more than once.

Rust will do its best to infer the most flexible type for the closure if it can.

Implementation

A closure is implemented as an anonymous struct. That struct has a field for each variable captured by the closure. It is lifetime-parametric with a single lifetime parameter which is a bound on the lifetime of captured variables. The anonymous struct implements a `call` method which is called to execute the closure.

For example, consider

```
fn main() {
    let x = Foo { ... };
    let f = |y| x.get_number() + y;
    let z = f(42);
}
```

the compiler treats this as

```
struct Closure14<'env> {
    x: &'env Foo,
}

// Not actually implemented like this, see below.
```

```
impl<'env> Closure14<'env> {
    fn call(&self, y: i32) -> i32 {
        self.x.get_number() + y
    }
}

fn main() {
    let x = Foo { ... };
    let f = Closure14 { x: x }
    let z = f.call(42);
}
```

As we mentioned above, there are three different function traits - `Fn`, `FnMut`, and `FnOnce`. In reality the `call` method is required by these traits rather than being in an inherent impl. `Fn` has a method `call` which takes `self` by reference, `FnMut` has `call_mut` taking `self` by mutable reference, and `FnOnce` has `call_once` which takes `self` by values.

When we've seen function types above, they look like `Fn(i32) -> i32` which doesn't look much like a trait type. There is a little bit of magic here. Rust allows this round bracket sugar only for function types. To desugar to a regular type (an 'angle bracket type'), the argument types are treated as a tuple type and passed as a type parameter and the return type as an associated type called `Output`. So, `Fn(i32) -> i32` is desugared to `Fn<(i32,), Output=i32>` and the `Fn` trait definition looks like

```
pub trait Fn<Args> : FnMut<Args> {
    fn call(&self, args: Args) -> Self::Output;
}
```

The implementation for `Closure14` above would therefore look more like

```
impl<'env> FnOnce<(i32,)> for Closure14<'env> {
    type Output = i32;
    fn call_once(self, args: (i32,)) -> i32 {
        ...
    }
}
impl<'env> FnMut<(i32,)> for Closure14<'env> {
    fn call_mut(&mut self, args: (i32,)) -> i32 {
        ...
    }
}
impl<'env> Fn<(i32,)> for Closure14<'env> {
    fn call(&self, args: (i32,)) -> i32 {
        ...
    }
}
```

You can find the function traits in core::ops

We talked above about how using generics gives static dispatch and using trait objects gives virtual dispatch. We can now see in a bit more detail why.

When we call `call`, it is a statically dispatched method call, there is no virtual dispatch. If we pass it to a monomorphised function, we still know the type statically, and we still get a static dispatch.

We can make the closure into a trait object, e.g., `&f` or `Box::new(f)` with types `&Fn(i32)->i32` or `Box<Fn(i32)->i32>`. These are pointer types, and because they are pointer-to-trait types, the pointers are fat pointers. That means they consist of the pointer to the data itself and a pointer to a vtable. The vtable is used to lookup the address of `call` (or `call_mut` or whatever).

You'll sometimes hear these two representations of closures called boxed and unboxed closures. An unboxed closure is the by-value version with static dispatch. A boxed version is the trait object version with dynamic dispatch. In the olden days, Rust only had boxed closures (and the system was quite a bit different).

References

- RFC 114 - Closures
- Finding Closure in Rust blog post
- RFC 387 - Higher ranked trait bounds
- Purging proc blog post

FIXME: relate to closures in C++ 11