-
-
-
-
-

Rust By Example

Search this book ...

Rust by Example

Rust is a modern systems programming language focusing on safety, speed, and concurrency. It accomplishes these goals by being memory safe without using garbage collection.

Rust by Example (RBE) is a collection of runnable examples that illustrate various Rust concepts and standard libraries. To get even more out of these examples, don't forget to install Rust locally and check out the official docs. Additionally for the curious, you can also check out the source code for this site.

Now let's begin!

-
-
-
-
-

Rust by Example

Rust is a modern systems programming language focusing on safety, speed, and concurrency. It accomplishes these goals by being memory safe without using garbage collection.

Rust by Example (RBE) is a collection of runnable examples that illustrate various Rust concepts and standard libraries. To get even more out of these examples, don't forget to install Rust locally and check out the official docs. Additionally for the curious, you can also check out the source code for this site.

Now let's begin!

- Hello World - Start with a traditional Hello World program.
- Primitives - Learn about signed integers, unsigned integers and other primitives.
- Custom Types - `struct` and `enum`.
- Variable Bindings - mutable bindings, scope, shadowing.
- Types - Learn about changing and defining types.
- Conversion
- Expressions
- Flow Control - `if`/`else`, `for`, and others.
- Functions - Learn about Methods, Closures and High Order Functions.
- Modules - Organize code using modules
- Crates - A crate is a compilation unit in Rust. Learn to create a library.
- Attributes - An attribute is metadata applied to some module, crate or item.
- Generics - Learn about writing a function or data type which can work for multiple types of arguments.
- Scoping rules - Scopes play an important part in ownership, borrowing, and lifetimes.
- Traits - A trait is a collection of methods defined for an unknown type: `Self`
- Macros
- Error handling - Learn Rust way of handling failures.
- Std library types - Learn about some custom types provided by `std` library.
- Std misc - More custom types for file handling, threads.
- Testing - All sorts of testing in Rust.
- Meta - Documentation, Benchmarking.
- Unsafe Operations

Rust By Example

Search this book ...

Hello World

This is the source code of the traditional Hello World program.

```
// This is a comment, and will be ignored by the compiler
// You can test this code by clicking the "Run" button over there ->
```

```
// or if prefer to use your keyboard, you can use the "Ctrl + Enter" shortcut

// This code is editable, feel free to hack it!
// You can always return to the original code by clicking the "Reset" button ->

// This is the main function
fn main() {
    // The statements here will be executed when the compiled binary is called

    // Print text to the console
    println!("Hello World!");
}
```

`println!` is a *[macro](#)* that prints text to the console.

A binary can be generated using the Rust compiler: `rustc`.

```
$ rustc hello.rs
```

`rustc` will produce a `hello` binary that can be executed.

```
$ ./hello
Hello World!
```

Activity

Click 'Run' above to see the expected output. Next, add a new line with a second `println!` macro so that the output shows:

```
Hello World!
I'm a Rustacean!
```

-
-
-
-
-

Rust By Example

Search this book ...

Comments

Any program requires comments and indeed Rust supports a few different varieties:

- *Regular comments* which are ignored by the compiler:

- `// Line comments which go to the end of the line.`
- `/* Block comments which go to the closing delimiter. */`

- *Doc comments* which are parsed into HTML library documentation:

- `/// Generate library docs for the following item.`
- `//! Generate library docs for the enclosing item.`

```
fn main() {
    // This is an example of a line comment
    // Notice how there are two slashes at the beginning of the line
    // And that nothing written inside these will be read by the compiler

    // println!("Hello, world!");

    // Run it. See? Now try deleting the two slashes, and run it again.

    /*
     * This is another type of comment, the block comment. In general,
     * the line comment is the recommended comment style however the
     * block comment is extremely useful for temporarily disabling
     * a large chunk of code. /* Block comments can be /* nested, */ */
     * so it takes only a few keystrokes to comment out all the lines
     * in this main() function. /*/*/* Try it yourself! */*/*/
     */

    /*
    Note, the previous column of `*` was entirely for style. There's
    no actual need for it.
    */

    // Observe how block comments allow easy expression manipulation
    // which line comments do not. Deleting the comment delimiters
    // will change the result:
    let x = 5 + /* 90 + */ 5;
    println!("Is `x` 10 or 100? x = {}", x);
}
```

See also:

Library documentation

Formatted print

Printing is handled by a series of `macros` defined in `std::fmt` some of which include:

- `format!`: write formatted text to `String`
- `print!`: same as `format!` but the text is printed to the console (io::stdout).
- `println!`: same as `print!` but a newline is appended.
- `eprint!`: same as `format!` but the text is printed to the standard error (io::stderr).
- `eprintln!`: sames as `eprint!` but a newline is appended.

All parse text in the same fashion. A plus is that the formatting correctness will be checked at compile time.

```
fn main() {
    // In general, the `{}` will be automatically replaced with any
    // arguments. These will be stringified.
    println!("{} days", 31);

    // Without a suffix, 31 becomes an i32. You can change what type 31 is,
    // with a suffix.

    // There are various optional patterns this works with. Positional
    // arguments can be used.
    println!("{0}, this is {1}. {1}, this is {0}", "Alice", "Bob");

    // As can named arguments.
    println!("{subject} {verb} {object}",
             object="the lazy dog",
             subject="the quick brown fox",
             verb="jumps over");

    // Special formatting can be specified after a `:`.
    println!("{} of {:b} people know binary, the other half doesn't", 1, 2);

    // You can right-align text with a specified width. This will output
    // "     1". 5 white spaces and a "1".
    println!("{number:>width$}", number=1, width=6);

    // You can pad numbers with extra zeroes. This will output "000001".
    println!("{number:>0width$}", number=1, width=6);

    // It will even check to make sure the correct number of arguments are
    // used.
    println!("My name is {0}, {1} {0}", "Bond");
    // FIXME ^ Add the missing argument: "James"

    // Create a structure which contains an `i32`. Name it `Structure`.
    #[allow(dead_code)]
    struct Structure(i32);

    // However, custom types such as this structure require more complicated
    // handling. This will not work.
    println!("This struct `{}` won't print...", Structure(3));
    // FIXME ^ Comment out this line.
}
```

`std::fmt` contains many `traits` which govern the display of text. The base form of two important ones are listed below:

- `fmt::Debug`: Uses the `{:?}` marker. Format text for debugging purposes.
- `fmt::Display`: Uses the `{}` marker. Format text in a more elegant, user friendly fashion.

Here, `fmt::Display` was used because the std library provides implementations for these types. To print text for custom types, more steps are required.

Activities

- Fix the two issues in the above code (see FIXME) so that it runs without error.
- Add a `println!` macro that prints: `Pi is roughly 3.142` by controlling the number of decimal places shown. For the purposes of this exercise, use `let pi = 3.141592` as an estimate for Pi. (Hint: you may need to check the `std::fmt` documentation for setting the number of decimals to display)

See also

`std::fmt`, `macros`, `struct`, and `traits`

Rust By Example

Search this book ...

Debug

All types which want to use `std::fmt` formatting `traits` require an implementation to be printable. Automatic implementations are only provided for types such as in the `std` library. All others *must* be manually implemented somehow.

The `fmt::Debug` `trait` makes this very straightforward. *All* types can `derive` (automatically create) the `fmt::Debug` implementation. This is not true for `fmt::Display` which must be manually implemented.

```
# #![allow(unused_variables)]
#fn main() {
// This structure cannot be printed either with `fmt::Display` or
// with `fmt::Debug`
struct UnPrintable(i32);

// The `derive` attribute automatically creates the implementation
// required to make this `struct` printable with `fmt::Debug`.
#[derive(Debug)]
struct DebugPrintable(i32);
#}
```

All `std` library types automatically are printable with `{:?}` too:

```
// Derive the `fmt::Debug` implementation for `Structure`. `Structure`
// is a structure which contains a single `i32`.
#[derive(Debug)]
struct Structure(i32);

// Put a `Structure` inside of the structure `Deep`. Make it printable
// also.
#[derive(Debug)]
struct Deep(Structure);

fn main() {
    // Printing with `{:?}` is similar to with `{}`.
    println!("{:?} months in a year.", 12);
    println!("{1:?} {0:?} is the {actor:?} name.",
             "Slater",
             "Christian",
             actor="actor's");

    // `Structure` is printable!
    println!("Now {:?} will print!", Structure(3));

    // The problem with `derive` is there is no control over how
    // the results look. What if I want this to just show a `7`?
    println!("Now {:?} will print!", Deep(Structure(7)));
}
```

So `fmt::Debug` definitely makes this printable but sacrifices some elegance. Rust also provides "pretty printing" with `{:#?}`.

```
#[derive(Debug)]
struct Person<'a> {
    name: &'a str,
    age: u8
}

fn main() {
    let name = "Peter";
    let age = 27;
    let peter = Person { name, age };

    // Pretty print
    println!("{:#?}", peter);
}
```

One can manually implement `fmt::Display` to control the display.

See also

attributes, `derive`, `std::fmt`, and `struct`

-

Rust By Example

Search this book ...

Display

`fmt::Debug` hardly looks compact and clean, so it is often advantageous to customize the output appearance. This is done by manually implementing `fmt::Display`, which uses the `{}` print marker. Implementing it looks like this:

```
# #![allow(unused_variables)]
#fn main() {
// Import (via `use`) the `fmt` module to make it available.
use std::fmt;

// Define a structure which `fmt::Display` will be implemented for. This is simply
// a tuple struct containing an `i32` bound to the name `Structure`.
struct Structure(i32);

// In order to use the `{}` marker, the trait `fmt::Display` must be implemented
// manually for the type.
impl fmt::Display for Structure {
    // This trait requires `fmt` with this exact signature.
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // Write strictly the first element into the supplied output
        // stream: `f`. Returns `fmt::Result` which indicates whether the
        // operation succeeded or failed. Note that `write!` uses syntax which
        // is very similar to `println!`.
        write!(f, "{}", self.0)
    }
}
#}
```

`fmt::Display` may be cleaner than `fmt::Debug` but this presents a problem for the `std` library. How should ambiguous types be displayed? For example, if the `std` library implemented a single style for all `Vec<T>`, what style should it be? Either of these two?

- `Vec<path>`: `/:/etc:/home/username:/bin` (split on `:`)
- `Vec<number>`: `1,2,3` (split on `,`)

No, because there is no ideal style for all types and the `std` library doesn't presume to dictate one. `fmt::Display` is not implemented for `Vec<T>` or for any other generic containers. `fmt::Debug` must then be used for these generic cases.

This is not a problem though because for any new *container* type which is *not* generic,`fmt::Display` can be implemented.

```
use std::fmt; // Import `fmt`

// A structure holding two numbers. `Debug` will be derived so the results can
// be contrasted with `Display`.
#[derive(Debug)]
struct MinMax(i64, i64);

// Implement `Display` for `MinMax`.
impl fmt::Display for MinMax {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // Use `self.number` to refer to each positional data point.
        write!(f, "({}, {})", self.0, self.1)
    }
}

// Define a structure where the fields are nameable for comparison.
#[derive(Debug)]
struct Point2D {
    x: f64,
    y: f64,
}

// Similarly, implement for Point2D
impl fmt::Display for Point2D {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // Customize so only `x` and `y` are denoted.
        write!(f, "x: {}, y: {}", self.x, self.y)
    }
}

fn main() {
    let minmax = MinMax(0, 14);

    println!("Compare structures:");
    println!("Display: {}", minmax);
    println!("Debug: {:?}", minmax);

    let big_range =   MinMax(-300, 300);
    let small_range = MinMax(-3, 3);

    println!("The big range is {big} and the small is {small}",
             small = small_range,
             big = big_range);

    let point = Point2D { x: 3.3, y: 7.2 };

    println!("Compare points:");
    println!("Display: {}", point);
    println!("Debug: {:?}", point);

    // Error. Both `Debug` and `Display` were implemented but `{:b}`
    // requires `fmt::Binary` to be implemented. This will not work.
    // println!("What does Point2D look like in binary: {:b}?", point);
}
```

So, `fmt::Display` has been implemented but `fmt::Binary` has not, and therefore cannot be used. `std::fmt` has many such `traits` and each requires its own implementation. This is detailed further in `std::fmt`.

Activity

After checking the output of the above example, use the `Point2D` struct as guide to add a Complex struct to the example. When printed in the same way, the output should be:

```
Display: 3.3 +7.2i
Debug: Complex { real: 3.3, imag: 7.2 }
```

See also

`derive`, `std::fmt`, macros, `struct`, `trait`, and use

-
-
-
-
-

Rust By Example

Search this book ...

Testcase: List

Implementing `fmt::Display` for a structure where the elements must each be handled sequentially is tricky. The problem is that each `write!` generates a `fmt::Result`. Proper handling of this requires dealing with *all* the results. Rust provides the `?` operator for exactly this purpose.

Using `?` on `write!` looks like this:

```
// Try `write!` to see if it errors. If it errors, return
// the error. Otherwise continue.
write!(f, "{}", value)?;
```

Alternatively, you can also use the `try!` macro, which works the same way. This is a bit more verbose and no longer recommended, but you may still see it in older Rust code. Using `try!` looks like this:

```
try!(write!(f, "{}", value));
```

With `?` available, implementing `fmt::Display` for a `Vec` is straightforward:

```
use std::fmt; // Import the `fmt` module.

// Define a structure named `List` containing a `Vec`.
struct List(Vec<i32>);

impl fmt::Display for List {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // Extract the value using tuple indexing
        // and create a reference to `vec`.
        let vec = &self.0;

        write!(f, "[")?;

        // Iterate over `vec` in `v` while enumerating the iteration
        // count in `count`.
        for (count, v) in vec.iter().enumerate() {
            // For every element except the first, add a comma.
            // Use the ? operator, or try!, to return on errors.
            if count != 0 { write!(f, ", ")?; }
            write!(f, "{}", v)?;
        }

        // Close the opened bracket and return a fmt::Result value
        write!(f, "]")
    }
}

fn main() {
    let v = List(vec![1, 2, 3]);
    println!("{}", v);
}
```

Activity

Try changing the program so that the index of each element in the vector is also printed. The new output should look like this:

```
[0: 1, 1: 2, 2: 3]
```

See also

`for`, `ref`, `Result`, `struct`, `?`, and `vec!`

-
-
-
-
-

Rust By Example

Search this book ...

Formatting

We've seen that formatting is specified via a *format string*:

- `format!("{}", foo)` -> `"3735928559"`
- `format!("0x{:X}", foo)` -> `"0xDEADBEEF"`
- `format!("0o{:o}", foo)` -> `"0o33653337357"`

The same variable (`foo`) can be formatted differently depending on which *argument type* is used: `X` vs `o` vs *unspecified*.

This formatting functionality is implemented via traits, and there is one trait for each argument type. The most common formatting trait is `Display`, which handles cases where the argument type is left unspecified: `{}` for instance.

```
use std::fmt::{self, Formatter, Display};

struct City {
    name: &'static str,
    // Latitude
    lat: f32,
    // Longitude
    lon: f32,
}

impl Display for City {
    // `f` is a buffer, this method must write the formatted string into it
    fn fmt(&self, f: &mut Formatter) -> fmt::Result {
        let lat_c = if self.lat >= 0.0 { 'N' } else { 'S' };
        let lon_c = if self.lon >= 0.0 { 'E' } else { 'W' };

        // `write!` is like `format!`, but it will write the formatted string
        // into a buffer (the first argument)
        write!(f, "{}: {:.3}Â°{} {:.3}Â°{}",
               self.name, self.lat.abs(), lat_c, self.lon.abs(), lon_c)
    }
}

#[derive(Debug)]
struct Color {
    red: u8,
    green: u8,
    blue: u8,
}

fn main() {
    for city in [
        City { name: "Dublin", lat: 53.347778, lon: -6.259722 },
        City { name: "Oslo", lat: 59.95, lon: 10.75 },
        City { name: "Vancouver", lat: 49.25, lon: -123.1 },
    ].iter() {
        println!("{}", *city);
    }
    for color in [
        Color { red: 128, green: 255, blue: 90 },
        Color { red: 0, green: 3, blue: 254 },
        Color { red: 0, green: 0, blue: 0 },
    ].iter() {
        // Switch this to use {} once you've added an implementation
        // for fmt::Display
        println!("{:?}", *color);
    }
}
```

You can view a full list of formatting traits and their argument types in the `std::fmt` documentation.

Activity

Add an implementation of the `fmt::Display` trait for the `Color` struct above so that the output displays as:

```
RGB (128, 255, 90) 0x80FF5A
RGB (0, 3, 254) 0x0003FE
RGB (0, 0, 0) 0x000000
```

Two hints if you get stuck:

- You may need to list each color more than once,
- You can pad with zeros to a width of 2 with `:02`.

See also

`std::fmt`

-
-
-
-
-

Rust By Example

Search this book ...

Primitives

Rust provides access to a wide variety of `primitives`. A sample includes:

Scalar Types

- signed integers: `i8`, `i16`, `i32`, `i64`, `i128` and `isize` (pointer size)
- unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128` and `usize` (pointer size)
- floating point: `f32`, `f64`
- `char` Unicode scalar values like `'a'`, `'α'` and `'∞'` (4 bytes each)
- `bool` either `true` or `false`
- and the unit type `()`, whose only possible value is an empty tuple: `()`

Despite the value of a unit type being a tuple, it is not considered a compound type because it does not contain multiple values.

Compound Types

- arrays like `[1, 2, 3]`
- tuples like `(1, true)`

Variables can always be *type annotated*. Numbers may additionally be annotated via a *suffix* or *by default*. Integers default to `i32` and floats to `f64`. Note that Rust can also infer types from context.

```
fn main() {
    // Variables can be type annotated.
    let logical: bool = true;

    let a_float: f64 = 1.0;  // Regular annotation
    let an_integer   = 5i32; // Suffix annotation

    // Or a default will be used.
    let default_float   = 3.0; // `f64`
    let default_integer = 7;   // `i32`

    // A type can also be inferred from context
    let mut inferred_type = 12; // Type i64 is inferred from another line
    inferred_type = 4294967296i64;

    // A mutable variable's value can be changed.
    let mut mutable = 12; // Mutable `i32`
    mutable = 21;

    // Error! The type of a variable can't be changed.
    mutable = true;

    // Variables can be overwritten with shadowing.
    let mutable = true;
}
```

See also:

the `std` library, `mut`, inference, and shadowing

-
-
-
-
-

Rust By Example

Search this book ...

Literals and operators

Integers `1`, floats `1.2`, characters `'a'`, strings `"abc"`, booleans `true` and the unit type `()` can be expressed using literals.

Integers can, alternatively, be expressed using hexadecimal, octal or binary notation using either of these prefixes: `0x`, `0o` or `0b`.

Underscores can be inserted in numeric literals to improve readability, e.g. `1_000` is the same as `1000`, and `0.000_001` is the same as `0.000001`.

We need to tell the compiler the type of the literals we use. For now, we'll use the `u32` suffix to indicate that the literal is an unsigned 32-bit integer, and the `i32` suffix to indicate that it's a signed 32-bit integer.

The operators available and their precedence [in Rust] are similar to other [C-like languages].

```
fn main() {
    // Integer addition
    println!("1 + 2 = {}", 1u32 + 2);

    // Integer subtraction
    println!("1 - 2 = {}", 1i32 - 2);
    // TODO ^ Try changing `1i32` to `1u32` to see why the type is important

    // Short-circuiting boolean logic
    println!("true AND false is {}", true && false);
    println!("true OR false is {}", true || false);
    println!("NOT true is {}", !true);

    // Bitwise operations
    println!("0011 AND 0101 is {:04b}", 0b0011u32 & 0b0101);
    println!("0011 OR 0101 is {:04b}", 0b0011u32 | 0b0101);
    println!("0011 XOR 0101 is {:04b}", 0b0011u32 ^ 0b0101);
    println!("1 << 5 is {}", 1u32 << 5);
    println!("0x80 >> 2 is 0x{:x}", 0x80u32 >> 2);

    // Use underscores to improve readability!
    println!("One million is written as {}", 1_000_000u32);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Tuples

A tuple is a collection of values of different types. Tuples are constructed using parentheses `()`, and each tuple itself is a value with type signature `(T1, T2, ...)`, where `T1`, `T2` are the types of its members. Functions can use tuples to return multiple values, as tuples can hold any number of values.

```
// Tuples can be used as function arguments and as return values
fn reverse(pair: (i32, bool)) -> (bool, i32) {
    // `let` can be used to bind the members of a tuple to variables
    let (integer, boolean) = pair;

    (boolean, integer)
}

// The following struct is for the activity.
#[derive(Debug)]
struct Matrix(f32, f32, f32, f32);

fn main() {
    // A tuple with a bunch of different types
    let long_tuple = (1u8, 2u16, 3u32, 4u64,
                      -1i8, -2i16, -3i32, -4i64,
                      0.1f32, 0.2f64,
                      'a', true);

    // Values can be extracted from the tuple using tuple indexing
    println!("long tuple first value: {}", long_tuple.0);
    println!("long tuple second value: {}", long_tuple.1);

    // Tuples can be tuple members
    let tuple_of_tuples = ((1u8, 2u16, 2u32), (4u64, -1i8), -2i16);

    // Tuples are printable
    println!("tuple of tuples: {:?}", tuple_of_tuples);

    // But long Tuples cannot be printed
    // let too_long_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13);
    // println!("too long tuple: {:?}", too_long_tuple);
    // TODO ^ Uncomment the above 2 lines to see the compiler error

    let pair = (1, true);
    println!("pair is {:?}", pair);
```

```
    println!("the reversed pair is {:?}", reverse(pair));

    // To create one element tuples, the comma is required to tell them apart
    // from a literal surrounded by parentheses
    println!("one element tuple: {:?}", (5u32,));
    println!("just an integer: {:?}", (5u32));

    //tuples can be destructured to create bindings
    let tuple = (1, "hello", 4.5, true);

    let (a, b, c, d) = tuple;
    println!("{:?}, {:?}, {:?}, {:?}", a, b, c, d);

    let matrix = Matrix(1.1, 1.2, 2.1, 2.2);
    println!("{:?}", matrix);

}
```

Activity

1. *Recap*: Add the `fmt::Display` trait to the Matrix `struct` in the above example, so that if you switch from printing the debug format `{:?}` to the display format `{}`, you see the following output:

   ```
   ( 1.1 1.2 )
   ( 2.1 2.2 )
   ```

   You may want to refer back to the example for print display.

2. Add a `transpose` function using the `reverse` function as a template, which accepts a matrix as an argument, and returns a matrix in which two elements have been swapped. For example:

   ```
   println!("Matrix:\n{}", matrix);
   println!("Transpose:\n{}", transpose(matrix));
   ```

   results in the output:

   ```
   Matrix:
   ( 1.1 1.2 )
   ( 2.1 2.2 )
   Transpose:
   ( 1.1 2.1 )
   ( 1.2 2.2 )
   ```

-
-
-
-
-

Rust By Example

Search this book ...

Arrays and Slices

An array is a collection of objects of the same type `T`, stored in contiguous memory. Arrays are created using brackets `[]`, and their size, which is known at compile time, is part of their type signature `[T; size]`.

Slices are similar to arrays, but their size is not known at compile time. Instead, a slice is a two-word object, the first word is a pointer to the data, and the second word is the length of the slice. The word size is the same as usize, determined by the processor architecture eg 64 bits on an x86-64. Slices can be used to borrow a section of an array, and have the type signature `&[T]`.

```
use std::mem;

// This function borrows a slice
fn analyze_slice(slice: &[i32]) {
    println!("first element of the slice: {}", slice[0]);
    println!("the slice has {} elements", slice.len());
}

fn main() {
    // Fixed-size array (type signature is superfluous)
    let xs: [i32; 5] = [1, 2, 3, 4, 5];

    // All elements can be initialized to the same value
    let ys: [i32; 500] = [0; 500];

    // Indexing starts at 0
    println!("first element of the array: {}", xs[0]);
    println!("second element of the array: {}", xs[1]);

    // `len` returns the size of the array
    println!("array size: {}", xs.len());

    // Arrays are stack allocated
    println!("array occupies {} bytes", mem::size_of_val(&xs));

    // Arrays can be automatically borrowed as slices
    println!("borrow the whole array as a slice");
    analyze_slice(&xs);
```

```
    // Slices can point to a section of an array
    println!("borrow a section of the array as a slice");
    analyze_slice(&ys[1 .. 4]);

    // Out of bound indexing yields a panic
    println!("{}", xs[5]);
}
```

Rust By Example

Search this book ...

Custom Types

Rust custom data types are formed mainly through the two keywords:

- `struct`: define a structure
- `enum`: define an enumeration

Constants can also be created via the `const` and `static` keywords.

Rust By Example

Search this book ...

Structures

There are three types of structures ("structs") that can be created using the `struct` keyword:

- Tuple structs, which are, basically, named tuples.
- The classic C structs
- Unit structs, which are field-less, are useful for generics.

```
#[derive(Debug)]
struct Person<'a> {
    name: &'a str,
    age: u8,
}

// A unit struct
struct Nil;

// A tuple struct
struct Pair(i32, f32);

// A struct with two fields
struct Point {
    x: f32,
    y: f32,
}

// Structs can be reused as fields of another struct
#[allow(dead_code)]
struct Rectangle {
    p1: Point,
    p2: Point,
}

fn main() {
    // Create struct with field init shorthand
```

```
    let name = "Peter";
    let age = 27;
    let peter = Person { name, age };

    // Print debug struct
    println!("{:?}", peter);


    // Instantiate a `Point`
    let point: Point = Point { x: 0.3, y: 0.4 };

    // Access the fields of the point
    println!("point coordinates: ({}, {})", point.x, point.y);

    // Make a new point by using struct update syntax to use the fields of our other one
    let new_point = Point { x: 0.1, ..point };
    // `new_point.y` will be the same as `point.y` because we used that field from `point`
    println!("second point: ({}, {})", new_point.x, new_point.y);

    // Destructure the point using a `let` binding
    let Point { x: my_x, y: my_y } = point;

    let _rectangle = Rectangle {
        // struct instantiation is an expression too
        p1: Point { x: my_y, y: my_x },
        p2: point,
    };

    // Instantiate a unit struct
    let _nil = Nil;

    // Instantiate a tuple struct
    let pair = Pair(1, 0.1);

    // Access the fields of a tuple struct
    println!("pair contains {:?} and {:?}", pair.0, pair.1);

    // Destructure a tuple struct
    let Pair(integer, decimal) = pair;

    println!("pair contains {:?} and {:?}", integer, decimal);
}
```

Activity

1. Add a function `rect_area` which calculates the area of a rectangle (try using nested destructuring).
2. Add a function `square` which takes a `Point` and a `f32` as arguments, and returns a `Rectangle` with its lower left corner on the point, and a width and height corresponding to the `f32`.

See also:

`attributes` and destructuring

-
-
-
-
-

Rust By Example

Search this book ...

Enums

The `enum` keyword allows the creation of a type which may be one of a few different variants. Any variant which is valid as a `struct` is also valid as an `enum`.

```
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

// Create an `enum` to classify a web event. Note how both
// names and type information together specify the variant:
// `PageLoad != PageUnload` and `KeyPress(char) != Paste(String)`.
// Each is different and independent.
enum WebEvent {
    // An `enum` may either be `unit-like`,
    PageLoad,
    PageUnload,
    // like tuple structs,
    KeyPress(char),
    Paste(String),
    // or like structures.
    Click { x: i64, y: i64 },
}

// A function which takes a `WebEvent` enum as an argument and
// returns nothing.
fn inspect(event: WebEvent) {
    match event {
        WebEvent::PageLoad => println!("page loaded"),
        WebEvent::PageUnload => println!("page unloaded"),
        // Destructure `c` from inside the `enum`.
        WebEvent::KeyPress(c) => println!("pressed '{}'.", c),
```

```
        WebEvent::Paste(s) => println!("pasted \"{}\".", s),
        // Destructure `Click` into `x` and `y`.
        WebEvent::Click { x, y } => {
            println!("clicked at x={}, y={}.", x, y);
        },
    }
}

fn main() {
    let pressed = WebEvent::KeyPress('x');
    // `to_owned()` creates an owned `String` from a string slice.
    let pasted  = WebEvent::Paste("my text".to_owned());
    let click   = WebEvent::Click { x: 20, y: 80 };
    let load    = WebEvent::PageLoad;
    let unload  = WebEvent::PageUnload;

    inspect(pressed);
    inspect(pasted);
    inspect(click);
    inspect(load);
    inspect(unload);
}
```

See also:

`attributes`, `match`, `fn`, and `String`

-
-
-
-
-

Rust By Example

Search this book ...

use

The `use` declaration can be used so manual scoping isn't needed:

```
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

enum Status {
    Rich,
    Poor,
}

enum Work {
    Civilian,
    Soldier,
}

fn main() {
    // Explicitly `use` each name so they are available without
    // manual scoping.
    use Status::{Poor, Rich};
    // Automatically `use` each name inside `Work`.
    use Work::*;

    // Equivalent to `Status::Poor`.
    let status = Poor;
    // Equivalent to `Work::Civilian`.
    let work = Civilian;

    match status {
        // Note the lack of scoping because of the explicit `use` above.
        Rich => println!("The rich have lots of money!"),
        Poor => println!("The poor have no money..."),
    }

    match work {
        // Note again the lack of scoping.
        Civilian => println!("Civilians work!"),
        Soldier  => println!("Soldiers fight!"),
    }
}
```

See also:

`match` and `use`

-
-

-
-
-

Rust By Example

Search this book ...

C-like

`enum` can also be used as C-like enums.

```
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

// enum with implicit discriminator (starts at 0)
enum Number {
    Zero,
    One,
    Two,
}

// enum with explicit discriminator
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff,
}

fn main() {
    // `enums` can be cast as integers.
    println!("zero is {}", Number::Zero as i32);
    println!("one is {}", Number::One as i32);

    println!("roses are #{:06x}", Color::Red as i32);
    println!("violets are #{:06x}", Color::Blue as i32);
}
```

See also:

casting

-
-
-
-
-

Rust By Example

Search this book ...

Testcase: linked-list

A common use for `enums` is to create a linked-list:

```
use List::*;

enum List {
    // Cons: Tuple struct that wraps an element and a pointer to the next node
    Cons(u32, Box<List>),
    // Nil: A node that signifies the end of the linked list
    Nil,
}

// Methods can be attached to an enum
impl List {
    // Create an empty list
    fn new() -> List {
        // `Nil` has type `List`
        Nil
    }

    // Consume a list, and return the same list with a new element at its front
    fn prepend(self, elem: u32) -> List {
        // `Cons` also has type List
        Cons(elem, Box::new(self))
    }

    // Return the length of the list
    fn len(&self) -> u32 {
```

```
        // `self` has to be matched, because the behavior of this method
        // depends on the variant of `self`
        // `self` has type `&List`, and `*self` has type `List`, matching on a
        // concrete type `T` is preferred over a match on a reference `&T`
        match *self {
            // Can't take ownership of the tail, because `self` is borrowed;
            // instead take a reference to the tail
            Cons(_, ref tail) => 1 + tail.len(),
            // Base Case: An empty list has zero length
            Nil => 0
        }
    }

    // Return representation of the list as a (heap allocated) string
    fn stringify(&self) -> String {
        match *self {
            Cons(head, ref tail) => {
                // `format!` is similar to `print!`, but returns a heap
                // allocated string instead of printing to the console
                format!("{}, {}", head, tail.stringify())
            },
            Nil => {
                format!("Nil")
            },
        }
    }
}

fn main() {
    // Create an empty linked list
    let mut list = List::new();

    // Prepend some elements
    list = list.prepend(1);
    list = list.prepend(2);
    list = list.prepend(3);

    // Show the final state of the list
    println!("linked list has length: {}", list.len());
    println!("{}", list.stringify());
}
```

See also:

`Box` and methods

-
-
-
-
-

Rust By Example

Search this book ...

constants

Rust has two different types of constants which can be declared in any scope including global. Both require explicit type annotation:

- `const`: An unchangeable value (the common case).
- `static`: A possibly `mutable` variable with `'static` lifetime.

One special case is the `"string"` literal. It can be assigned directly to a `static` variable without modification because its type signature: `&'static str` has the required lifetime of `'static`. All other reference types must be specifically annotated so that they fulfill the `'static` lifetime. This may seem minor though because the required explicit annotation hides the distinction.

```
// Globals are declared outside all other scopes.
static LANGUAGE: &'static str = "Rust";
const  THRESHOLD: i32 = 10;

fn is_big(n: i32) -> bool {
    // Access constant in some function
    n > THRESHOLD
}

fn main() {
    let n = 16;

    // Access constant in the main thread
    println!("This is {}", LANGUAGE);
    println!("The threshold is {}", THRESHOLD);
    println!("{} is {}", n, if is_big(n) { "big" } else { "small" });

    // Error! Cannot modify a `const`.
    THRESHOLD = 5;
    // FIXME ^ Comment out this line
}
```

See also:

The `const`/`static` RFC, `'static` lifetime

Variable Bindings

Rust provides type safety via static typing. Variable bindings can be type annotated when declared. However, in most cases, the compiler will be able to infer the type of the variable from the context, heavily reducing the annotation burden.

Values (like literals) can be bound to variables, using the `let` binding.

```
fn main() {
    let an_integer = 1u32;
    let a_boolean = true;
    let unit = ();

    // copy `an_integer` into `copied_integer`
    let copied_integer = an_integer;

    println!("An integer: {:?}", copied_integer);
    println!("A boolean: {:?}", a_boolean);
    println!("Meet the unit value: {:?}", unit);

    // The compiler warns about unused variable bindings; these warnings can
    // be silenced by prefixing the variable name with an underscore
    let _unused_variable = 3u32;

    let noisy_unused_variable = 2u32;
    // FIXME ^ Prefix with an underscore to suppress the warning
}
```


Mutability

Variable bindings are immutable by default, but this can be overridden using the `mut` modifier.

```
fn main() {
    let _immutable_binding = 1;
    let mut mutable_binding = 1;

    println!("Before mutation: {}", mutable_binding);

    // Ok
    mutable_binding += 1;

    println!("After mutation: {}", mutable_binding);

    // Error!
    _immutable_binding += 1;
    // FIXME ^ Comment out this line
}
```

The compiler will throw a detailed diagnostic about mutability errors.

-
-
-
-

Rust By Example

Search this book ...

Scope and Shadowing

Variable bindings have a scope, and are constrained to live in a *block*. A block is a collection of statements enclosed by braces `{}`. Also, variable shadowing is allowed.

```
fn main() {
    // This binding lives in the main function
    let long_lived_binding = 1;

    // This is a block, and has a smaller scope than the main function
    {
        // This binding only exists in this block
        let short_lived_binding = 2;

        println!("inner short: {}", short_lived_binding);

        // This binding *shadows* the outer one
        let long_lived_binding = 5_f32;

        println!("inner long: {}", long_lived_binding);
    }
    // End of the block

    // Error! `short_lived_binding` doesn't exist in this scope
    println!("outer short: {}", short_lived_binding);
    // FIXME ^ Comment out this line

    println!("outer long: {}", long_lived_binding);

    // This binding also *shadows* the previous binding
    let long_lived_binding = 'a';

    println!("outer long: {}", long_lived_binding);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Declare first

It's possible to declare variable bindings first, and initialize them later. However, this form is seldom used, as it may lead to the use of uninitialized variables.

```
fn main() {
    // Declare a variable binding
    let a_binding;

    {
        let x = 2;

        // Initialize the binding
        a_binding = x * x;
    }

    println!("a binding: {}", a_binding);

    let another_binding;

    // Error! Use of uninitialized binding
    println!("another binding: {}", another_binding);
    // FIXME ^ Comment out this line

    another_binding = 1;

    println!("another binding: {}", another_binding);
}
```

The compiler forbids use of uninitialized variables, as this would lead to undefined behavior.

Rust By Example

Search this book ...

Types

Rust provides several mechanisms to change or define the type of primitive and user defined types. The following sections cover:

- Casting between primitive types
- Specifying the desired type of literals
- Using type inference
- Aliasing types

Rust By Example

Search this book ...

Casting

Rust provides no implicit type conversion (coercion) between primitive types. But, explicit type conversion (casting) can be performed using the `as` keyword.

Rules for converting between integral types follow C conventions generally, except in cases where C has undefined behavior. The behavior of all casts between integral types is well defined in Rust.

```
// Suppress all warnings from casts which overflow.
#![allow(overflowing_literals)]

fn main() {
    let decimal = 65.4321_f32;

    // Error! No implicit conversion
    let integer: u8 = decimal;
    // FIXME ^ Comment out this line

    // Explicit conversion
    let integer = decimal as u8;
    let character = integer as char;

    println!("Casting: {} -> {} -> {}", decimal, integer, character);

    // when casting any value to an unsigned type, T,
    // std::T::MAX + 1 is added or subtracted until the value
    // fits into the new type

    // 1000 already fits in a u16
    println!("1000 as a u16 is: {}", 1000 as u16);

    // 1000 - 256 - 256 - 256 = 232
    // Under the hood, the first 8 least significant bits (LSB) are kept,
    // while the rest towards the most significant bit (MSB) get truncated.
    println!("1000 as a u8 is : {}", 1000 as u8);
    // -1 + 256 = 255
    println!("  -1 as a u8 is : {}", (-1i8) as u8);

    // For positive numbers, this is the same as the modulus
    println!("1000 mod 256 is : {}", 1000 % 256);

    // When casting to a signed type, the (bitwise) result is the same as
    // first casting to the corresponding unsigned type. If the most significant
```

```
    // bit of that value is 1, then the value is negative.

    // Unless it already fits, of course.
    println!(" 128 as a i16 is: {}", 128 as i16);
    // 128 as u8 -> 128, whose two's complement in eight bits is:
    println!(" 128 as a i8 is : {}", 128 as i8);

    // repeating the example above
    // 1000 as u8 -> 232
    println!("1000 as a u8 is : {}", 1000 as u8);
    // and the two's complement of 232 is -24
    println!(" 232 as a i8 is : {}", 232 as i8);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Literals

Numeric literals can be type annotated by adding the type as a suffix. As an example, to specify that the literal `42` should have the type `i32`, write `42i32`.

The type of unsuffixed numeric literals will depend on how they are used. If no constraint exists, the compiler will use `i32` for integers, and `f64` for floating-point numbers.

```
fn main() {
    // Suffixed literals, their types are known at initialization
    let x = 1u8;
    let y = 2u32;
    let z = 3f32;

    // Unsuffixed literal, their types depend on how they are used
    let i = 1;
    let f = 1.0;

    // `size_of_val` returns the size of a variable in bytes
    println!("size of `x` in bytes: {}", std::mem::size_of_val(&x));
    println!("size of `y` in bytes: {}", std::mem::size_of_val(&y));
    println!("size of `z` in bytes: {}", std::mem::size_of_val(&z));
    println!("size of `i` in bytes: {}", std::mem::size_of_val(&i));
    println!("size of `f` in bytes: {}", std::mem::size_of_val(&f));
}
```

There are some concepts used in the previous code that haven't been explained yet, here's a brief explanation for the impatient readers:

- `fun(&foo)` is used to pass an argument to a function *by reference*, rather than by value (`fun(foo)`). For more details see borrowing.
- `std::mem::size_of_val` is a function, but called with its *full path*. Code can be split in logical units called *modules*. In this case, the `size_of_val` function is defined in the `mem` module, and the `mem` module is defined in the `std` *crate*. For more details, see modules and crates.

-
-
-
-
-

Rust By Example

Search this book ...

Inference

The type inference engine is pretty smart. It does more than looking at the type of the r-value during an initialization. It also looks at how the variable is used afterwards to infer its type. Here's an advanced example of type inference:

```
fn main() {
    // Because of the annotation, the compiler knows that `elem` has type u8.
    let elem = 5u8;
```

```
    // Create an empty vector (a growable array).
    let mut vec = Vec::new();
    // At this point the compiler doesn't know the exact type of `vec`, it
    // just knows that it's a vector of something (`Vec<_>`).

    // Insert `elem` in the vector.
    vec.push(elem);
    // Aha! Now the compiler knows that `vec` is a vector of `u8`s (`Vec<u8>`)
    // TODO ^ Try commenting out the `vec.push(elem)` line

    println!("{:?}", vec);
}
```

No type annotation of variables was needed, the compiler is happy and so is the programmer!

-
-
-
-
-

Rust By Example

Search this book ...

Aliasing

The `type` statement can be used to give a new name to an existing type. Types must have `CamelCase` names, or the compiler will raise a warning. The exception to this rule are the primitive types: `usize`, `f32`, etc.

```
// `NanoSecond` is a new name for `u64`.
type NanoSecond = u64;
type Inch = u64;

// Use an attribute to silence warning.
#[allow(non_camel_case_types)]
type u64_t = u64;
// TODO ^ Try removing the attribute

fn main() {
    // `NanoSecond` = `Inch` = `u64_t` = `u64`.
    let nanoseconds: NanoSecond = 5 as u64_t;
    let inches: Inch = 2 as u64_t;

    // Note that type aliases *don't* provide any extra type safety, because
    // aliases are *not* new types
    println!("{} nanoseconds + {} inches = {} unit?",
             nanoseconds,
             inches,
             nanoseconds + inches);
}
```

The main use of aliases is to reduce boilerplate; for example the `IoResult<T>` type is an alias for the `Result<T, IoError>` type.

See also:

Attributes

-
-
-
-
-

Rust By Example

Search this book ...

Conversion

Rust addresses conversion between types by the use of traits. The generic conversions will use the `From` and `Into` traits. However there are more specific ones for the more common cases, in particular when converting to and from `String`s.

-
-
-
-
-

Rust By Example

Search this book ...

`From` and `Into`

The `From` and `Into` traits are inherently linked, and this is actually part of its implementation. If you are able to convert type A from type B, then it should be easy to believe that we should be able to convert type B to type A.

`From`

The `From` trait allows for a type to define how to create itself from another type, hence providing a very simple mechanism for converting between several types. There are numerous implementations of this trait within the standard library for conversion of primitive and common types.

For example we can easily convert a `str` into a `String`

```
# #![allow(unused_variables)]
#fn main() {
let my_str = "hello";
let my_string = String::from(my_str);
#}
```

We can do similar for defining a conversion for our own type.

```
use std::convert::From;

#[derive(Debug)]
struct Number {
    value: i32,
}

impl From<i32> for Number {
    fn from(item: i32) -> Self {
        Number { value: item }
    }
}

fn main() {
    let num = Number::from(30);
    println!("My number is {:?}", num);
}
```

`Into`

The `Into` trait is simply the reciprocal of the `From` trait. That is, if you have implemented the `From` trait for your type you get the `Into` implementation for free.

Using the `Into` trait will typically require specification of the type to convert into as the compiler is unable to determine this most of the time. However this is a small trade-off considering we get the functionality for free.

```
use std::convert::From;

#[derive(Debug)]
struct Number {
    value: i32,
}

impl From<i32> for Number {
    fn from(item: i32) -> Self {
        Number { value: item }
    }
}

fn main() {
    let int = 5;
    // Try removing the type declaration
    let num: Number = int.into();
    println!("My number is {:?}", num);
}
```

-
-
-

To and from Strings

`ToString`

To convert any type to a `String` it is as simple as implementing the `ToString` trait for the type.

```
use std::string::ToString;

struct Circle {
    radius: i32
}

impl ToString for Circle {
    fn to_string(&self) -> String {
        format!("Circle of radius {:?}", self.radius)
    }
}

fn main() {
    let circle = Circle { radius: 6 };
    println!("{}", circle.to_string());
}
```

Parsing a String

One of the more common types to convert a string into is a number. The idiomatic approach to this is to use the `parse` function and provide the type for the function to parse the string value into, this can be done either without type inference or using the 'turbofish' syntax.

This will convert the string into the type specified so long as the `FromStr` trait is implemented for that type. This is implemented for numerous types within the standard library. To obtain this functionality on a user defined type simply implement the `FromStr` trait for that type.

```
fn main() {
    let parsed: i32 = "5".parse().unwrap();
    let turbo_parsed = "10".parse::<i32>().unwrap();

    let sum = parsed + turbo_parsed;
    println!{"Sum: {:?}", sum};
}
```


Rust By Example

Search this book ...

Expressions

A Rust program is (mostly) made up of a series of statements:

```
fn main() {
    // statement
    // statement
    // statement
}
```

There are a few kinds of statements in Rust. The most common two are declaring a variable binding, and using a `;` with an expression:

```
fn main() {
    // variable binding
    let x = 5;

    // expression;
    x;
    x + 1;
    15;
}
```

Blocks are expressions too, so they can be used as r-values in assignments. The last expression in the block will be assigned to the l-value. However, if the last expression of the block ends with a semicolon, the return value will be `()`.

```
fn main() {
    let x = 5u32;

    let y = {
        let x_squared = x * x;
        let x_cube = x_squared * x;

        // This expression will be assigned to `y`
        x_cube + x_squared + x
    };

    let z = {
        // The semicolon suppresses this expression and `()` is assigned to `z`
        2 * x;
    };

    println!("x is {:?}", x);
    println!("y is {:?}", y);
    println!("z is {:?}", z);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Flow Control

An essential part of any programming languages are ways to modify control flow: `if`/`else`, `for`, and others. Let's talk about them in Rust.

-
-
-
-
-

Rust By Example

Search this book ...

if/else

Branching with `if`-`else` is similar to other languages. Unlike many of them, the boolean condition doesn't need to be surrounded by parentheses, and each condition is followed by a block. `if`-`else` conditionals are expressions, and, all branches must return the same type.

```
fn main() {
    let n = 5;

    if n < 0 {
        print!("{} is negative", n);
    } else if n > 0 {
        print!("{} is positive", n);
    } else {
        print!("{} is zero", n);
    }

    let big_n =
        if n < 10 && n > -10 {
            println!(", and is a small number, increase ten-fold");

            // This expression returns an `i32`.
            10 * n
        } else {
            println!(", and is a big number, half the number");

            // This expression must return an `i32` as well.
            n / 2
```

```
            // TODO ^ Try suppressing this expression with a semicolon.
        };
    //   ^ Don't forget to put a semicolon here! All `let` bindings need it.

    println!("{} -> {}", n, big_n);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

loop

Rust provides a `loop` keyword to indicate an infinite loop.

The `break` statement can be used to exit a loop at anytime, whereas the `continue` statement can be used to skip the rest of the iteration and start a new one.

```
fn main() {
    let mut count = 0u32;

    println!("Let's count until infinity!");

    // Infinite loop
    loop {
        count += 1;

        if count == 3 {
            println!("three");

            // Skip the rest of this iteration
            continue;
        }

        println!("{}", count);

        if count == 5 {
            println!("OK, that's enough");

            // Exit this loop
            break;
        }
    }
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Nesting and labels

It's possible to `break` or `continue` outer loops when dealing with nested loops. In these cases, the loops must be annotated with some `'label`, and the label must be passed to the `break`/`continue` statement.

```
#![allow(unreachable_code)]

fn main() {
    'outer: loop {
        println!("Entered the outer loop");

        'inner: loop {
            println!("Entered the inner loop");

            // This would break only the inner loop
```

```
            //break;

            // This breaks the outer loop
            break 'outer;
        }

        println!("This point will never be reached");
    }

    println!("Exited the outer loop");
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Returning from loops

One of the uses of a `loop` is to retry an operation until it succeeds. If the operation returns a value though, you might need to pass it to the rest of the code: put it after the `break`, and it will be returned by the `loop` expression.

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    assert_eq!(result, 20);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

while

The `while` keyword can be used to loop until a condition is met.

Let's write the infamous FizzBuzz using a `while` loop.

```
fn main() {
    // A counter variable
    let mut n = 1;

    // Loop while `n` is less than 101
    while n < 101 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
        }

        // Increment counter
        n += 1;
```

```
    }
}
```

-
-
-
-
-

Rust By Example

Search this book ...

for loops

for and range

The `for in` construct can be used to iterate through an `Iterator`. One of the easiest ways to create an iterator is to use the range notation `a..b`. This yields values from `a` (inclusive) to `b` (exclusive) in steps of one.

Let's write FizzBuzz using `for` instead of `while`.

```
fn main() {
    // `n` will take the values: 1, 2, ..., 100 in each iteration
    for n in 1..101 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
        }
    }
}
```

Alternatively, `a..=b` can be used for a range that is inclusive on both ends. The above can be written as:

```
fn main() {
    // `n` will take the values: 1, 2, ..., 100 in each iteration
    for n in 1..=100 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
        }
    }
}
```

for and iterators

The `for in` construct is able to interact with an `Iterator` in several ways. As discussed in with the Iterator trait, if not specified, the `for` loop will apply the `into_iter` function on the collection provided to convert the collection into an iterator. This is not the only means to convert a collection into an iterator however, the other functions available include `iter` and `iter_mut`.

These 3 functions will return different views of the data within your collection.

- `iter` - This borrows each element of the collection through each iteration. Thus leaving the collection untouched and available for reuse after the loop.

```
fn main() {
    let names = vec!["Bob", "Frank", "Ferris"];

    for name in names.iter() {
        match name {
            &"Ferris" => println!("There is a rustacean among us!"),
            _ => println!("Hello {}", name),
        }
    }
}
```

- `into_iter` - This consumes the collection so that on each iteration the exact data is provided. Once the collection has been consumed it is no longer available for reuse as it has been 'moved' within the loop.

```
fn main() {
    let names = vec!["Bob", "Frank", "Ferris"];

    for name in names.into_iter() {
        match name {
            "Ferris" => println!("There is a rustacean among us!"),
            _ => println!("Hello {}", name),
        }
    }
}
```

* `iter_mut` - This mutably borrows each element of the collection, allowing for the collection to be modified in place.

```
fn main() {
    let mut names = vec!["Bob", "Frank", "Ferris"];

    for name in names.iter_mut() {
        match name {
            &mut "Ferris" => println!("There is a rustacean among us!"),
            _ => println!("Hello {}", name),
        }
    }
}
```

In the above snippets note the type of `match` branch, that is the key difference in the types or iteration. The difference in type then of course implies differing actions that are able to be performed.

See also

Iterator

*
*
*
*
*

Rust By Example

Search this book ...

match

Rust provides pattern matching via the `match` keyword, which can be used like a C `switch`.

```
fn main() {
    let number = 13;
    // TODO ^ Try different values for `number`

    println!("Tell me about {}", number);
    match number {
        // Match a single value
        1 => println!("One!"),
        // Match several values
        2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
        // Match an inclusive range
        13...19 => println!("A teen"),
        // Handle the rest of cases
        _ => println!("Ain't special"),
    }

    let boolean = true;
    // Match is an expression too
    let binary = match boolean {
        // The arms of a match must cover all the possible values
        false => 0,
        true => 1,
        // TODO ^ Try commenting out one of these arms
    };

    println!("{} -> {}", boolean, binary);
}
```

*
*
*
*
*

Rust By Example

Search this book ...

Destructuring

A `match` block can destructure items in a variety of ways.

- Destructuring Enums
- Destructuring Pointers
- Destructuring Structures
- Destructuring Tuples

-
-
-
-
-

Rust By Example

Search this book ...

tuples

Tuples can be destructured in a `match` as follows:

```
fn main() {
    let pair = (0, -2);
    // TODO ^ Try different values for `pair`

    println!("Tell me about {:?}", pair);
    // Match can be used to destructure a tuple
    match pair {
        // Destructure the second
        (0, y) => println!("First is `0` and `y` is `{:?}`", y),
        (x, 0) => println!("`x` is `{:?}` and last is `0`", x),
        _      => println!("It doesn't matter what they are"),
        // `_` means don't bind the value to a variable
    }
}
```

See also:

Tuples

-
-
-
-
-

Rust By Example

Search this book ...

enums

An `enum` is destructured similarly:

```
// `allow` required to silence warnings because only
// one variant is used.
#[allow(dead_code)]
enum Color {
    // These 3 are specified solely by their name.
    Red,
    Blue,
    Green,
    // These likewise tie `u32` tuples to different names: color models.
    RGB(u32, u32, u32),
    HSV(u32, u32, u32),
    HSL(u32, u32, u32),
    CMY(u32, u32, u32),
    CMYK(u32, u32, u32, u32),
}

fn main() {
    let color = Color::RGB(122, 17, 40);
```

```
    // TODO ^ Try different variants for `color`

    println!("What color is it?");
    // An `enum` can be destructured using a `match`.
    match color {
        Color::Red   => println!("The color is Red!"),
        Color::Blue  => println!("The color is Blue!"),
        Color::Green => println!("The color is Green!"),
        Color::RGB(r, g, b) =>
            println!("Red: {}, green: {}, and blue: {}!", r, g, b),
        Color::HSV(h, s, v) =>
            println!("Hue: {}, saturation: {}, value: {}!", h, s, v),
        Color::HSL(h, s, l) =>
            println!("Hue: {}, saturation: {}, lightness: {}!", h, s, l),
        Color::CMY(c, m, y) =>
            println!("Cyan: {}, magenta: {}, yellow: {}!", c, m, y),
        Color::CMYK(c, m, y, k) =>
            println!("Cyan: {}, magenta: {}, yellow: {}, key (black): {}!",
                c, m, y, k),
        // Don't need another arm because all variants have been examined
    }
}
```

See also:

`#[allow(...)]`, color models and `enum`

-
-
-
-
-

Rust By Example

Search this book ...

pointers/ref

For pointers, a distinction needs to be made between destructuring and dereferencing as they are different concepts which are used differently from a language like `C`.

- Dereferencing uses `*`
- Destructuring uses `&`, `ref`, and `ref mut`

```
fn main() {
    // Assign a reference of type `i32`. The `&` signifies there
    // is a reference being assigned.
    let reference = &4;

    match reference {
        // If `reference`s is pattern matched against `&val`, it results
        // in a comparison like:
        // `&i32`
        // `&val`
        // ^ We see that if the matching `&`s are dropped, then the `i32`
        // should be assigned to `val`.
        &val => println!("Got a value via destructuring: {:?}", val),
    }

    // To avoid the `&`, you dereference before matching.
    match *reference {
        val => println!("Got a value via dereferencing: {:?}", val),
    }

    // What if you don't start with a reference? `reference` was a `&`
    // because the right side was already a reference. This is not
    // a reference because the right side is not one.
    let _not_a_reference = 3;

    // Rust provides `ref` for exactly this purpose. It modifies the
    // assignment so that a reference is created for the element; this
    // reference is assigned.
    let ref _is_a_reference = 3;

    // Accordingly, by defining 2 values without references, references
    // can be retrieved via `ref` and `ref mut`.
    let value = 5;
    let mut mut_value = 6;

    // Use `ref` keyword to create a reference.
    match value {
        ref r => println!("Got a reference to a value: {:?}", r),
    }

    // Use `ref mut` similarly.
    match mut_value {
        ref mut m => {
            // Got a reference. Gotta dereference it before we can
            // add anything to it.
            *m += 10;
            println!("We added 10. `mut_value`: {:?}", m);
        },
    }
```

```
}
```

-
-
-
-
-

Rust By Example

Search this book ...

structs

Similarly, a `struct` can be destructured as shown:

```
fn main() {
    struct Foo { x: (u32, u32), y: u32 }

    // destructure members of the struct
    let foo = Foo { x: (1, 2), y: 3 };
    let Foo { x: (a, b), y } = foo;

    println!("a = {}, b = {},  y = {} ", a, b, y);

    // you can destructure structs and rename the variables,
    // the order is not important

    let Foo { y: i, x: j } = foo;
    println!("i = {:?}, j = {:?}", i, j);

    // and you can also ignore some variables:
    let Foo { y, .. } = foo;
    println!("y = {}", y);

    // this will give an error: pattern does not mention field `x`
    // let Foo { y } = foo;
}
```

See also:

[Structs](), [The ref pattern]()

-
-
-
-
-

Rust By Example

Search this book ...

Guards

A `match` *guard* can be added to filter the arm.

```
fn main() {
    let pair = (2, -2);
    // TODO ^ Try different values for `pair`

    println!("Tell me about {:?}", pair);
    match pair {
        (x, y) if x == y => println!("These are twins"),
        // The ^ `if condition` part is a guard
        (x, y) if x + y == 0 => println!("Antimatter, kaboom!"),
        (x, _) if x % 2 == 1 => println!("The first one is odd"),
        _ => println!("No correlation..."),
    }
}
```

See also:

[Binding](#)

Indirectly accessing a variable makes it impossible to branch and use that variable without re-binding. `match` provides the `@` sigil for binding values to names:

```
// A function `age` which returns a `u32`.
fn age() -> u32 {
    15
}

fn main() {
    println!("Tell me type of person you are");

    match age() {
        0             => println!("I'm not born yet I guess"),
        // Could `match` 1 ... 12 directly but then what age
        // would the child be? Instead, bind to `n` for the
        // sequence of 1 .. 12. Now the age can be reported.
        n @ 1  ... 12 => println!("I'm a child of age {:?}", n),
        n @ 13 ... 19 => println!("I'm a teen of age {:?}", n),
        // Nothing bound. Return the result.
        n             => println!("I'm an old person of age {:?}", n),
    }
}
```

[See also:](#)

[functions](#)

[if let](#)

For some use cases, when matching enums, `match` is awkward. For example:

```
# #![allow(unused_variables)]
#fn main() {
// Make `optional` of type `Option<i32>`
let optional = Some(7);

match optional {
    Some(i) => {
        println!("This is a really long string and `{:?}`", i);
        // ^ Needed 2 indentations just so we could destructure
        // `i` from the option.
    },
    _ => {},
    // ^ Required because `match` is exhaustive. Doesn't it seem
    // like wasted space?
};

#}
```

`if let` is cleaner for this use case and in addition allows various failure options to be specified:

```
fn main() {
    // All have type `Option<i32>`
    let number = Some(7);
    let letter: Option<i32> = None;
    let emoticon: Option<i32> = None;

    // The `if let` construct reads: "if `let` destructures `number` into
    // `Some(i)`, evaluate the block (`{}`).
    if let Some(i) = number {
        println!("Matched {:?}!", i);
    }

    // If you need to specify a failure, use an else:
    if let Some(i) = letter {
        println!("Matched {:?}!", i);
    } else {
        // Destructure failed. Change to the failure case.
        println!("Didn't match a number. Let's go with a letter!");
    };

    // Provide an altered failing condition.
    let i_like_letters = false;

    if let Some(i) = emoticon {
        println!("Matched {:?}!", i);
    // Destructure failed. Evaluate an `else if` condition to see if the
    // alternate failure branch should be taken:
    } else if i_like_letters {
        println!("Didn't match a number. Let's go with a letter!");
    } else {
        // The condition evaluated false. This branch is the default:
        println!("I don't like letters. Let's go with an emoticon :)!");
    };
}
```

In the same way, `if let` can be used to match any enum value:

```
// Our example enum
enum Foo {
    Bar,
    Baz,
    Qux(u32)
}

fn main() {
    // Create example variables
    let a = Foo::Bar;
    let b = Foo::Baz;
    let c = Foo::Qux(100);

    // Variable a matches Foo::Bar
    if let Foo::Bar = a {
        println!("a is foobar");
    }

    // Variable b does not match Foo::Bar
    // So this will print nothing
    if let Foo::Bar = b {
        println!("b is foobar");
    }

    // Variable c matches Foo::Qux which has a value
    // Similar to Some() in the previous example
    if let Foo::Qux(value) = c {
        println!("c is {}", value);
    }
}
```

See also:

`enum`, `Option`, and the RFC

-
-
-
-
-

Rust By Example

Search this book ...

while let

Similar to `if let`, `while let` can make awkward `match` sequences more tolerable. Consider the following sequence that increments `i`:

```
# #![allow(unused_variables)]
#fn main() {
// Make `optional` of type `Option<i32>`
let mut optional = Some(0);
```

```
// Repeatedly try this test.
loop {
    match optional {
        // If `optional` destructures, evaluate the block.
        Some(i) => {
            if i > 9 {
                println!("Greater than 9, quit!");
                optional = None;
            } else {
                println!("`i` is `{:?}`. Try again.", i);
                optional = Some(i + 1);
            }
            // ^ Requires 3 indentations!
        },
        // Quit the loop when the destructure fails:
        _ => { break; }
        // ^ Why should this be required? There must be a better way!
    }
}
#}
```

Using `while let` makes this sequence much nicer:

```
fn main() {
    // Make `optional` of type `Option<i32>`
    let mut optional = Some(0);

    // This reads: "while `let` destructures `optional` into
    // `Some(i)`, evaluate the block (`{}`). Else `break`.
    while let Some(i) = optional {
        if i > 9 {
            println!("Greater than 9, quit!");
            optional = None;
        } else {
            println!("`i` is `{:?}`. Try again.", i);
            optional = Some(i + 1);
        }
        // ^ Less rightward drift and doesn't require
        // explicitly handling the failing case.
    }
    // ^ `if let` had additional optional `else`/`else if`
    // clauses. `while let` does not have these.
}
```

See also:

`enum`, `Option`, and the RFC

-
-
-
-
-

Rust By Example

Search this book ...

Functions

Functions are declared using the `fn` keyword. Its arguments are type annotated, just like variables, and, if the function returns a value, the return type must be specified after an arrow `->`.

The final expression in the function will be used as return value. Alternatively, the `return` statement can be used to return a value earlier from within the function, even from inside loops or `if`s.

Let's rewrite FizzBuzz using functions!

```
// Unlike C/C++, there's no restriction on the order of function definitions
fn main() {
    // We can use this function here, and define it somewhere later
    fizzbuzz_to(100);
}

// Function that returns a boolean value
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    // Corner case, early return
    if rhs == 0 {
        return false;
    }

    // This is an expression, the `return` keyword is not necessary here
    lhs % rhs == 0
}

// Functions that "don't" return a value, actually return the unit type `()`
fn fizzbuzz(n: u32) -> () {
    if is_divisible_by(n, 15) {
        println!("fizzbuzz");
    } else if is_divisible_by(n, 3) {
        println!("fizz");
    } else if is_divisible_by(n, 5) {
        println!("buzz");
```

```
    } else {
        println!("{}", n);
    }
}

// When a function returns `()`, the return type can be omitted from the
// signature
fn fizzbuzz_to(n: u32) {
    for n in 1..n + 1 {
        fizzbuzz(n);
    }
}
```


Methods

Methods are functions attached to objects. These methods have access to the data of the object and its other methods via the `self` keyword. Methods are defined under an `impl` block.

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

// Implementation block, all `Point` methods go in here
impl Point {
    // This is a static method
    // Static methods don't need to be called by an instance
    // These methods are generally used as constructors
    fn origin() -> Point {
        Point { x: 0.0, y: 0.0 }
    }

    // Another static method, taking two arguments:
    fn new(x: f64, y: f64) -> Point {
        Point { x: x, y: y }
    }
}

struct Rectangle {
    p1: Point,
    p2: Point,
}

impl Rectangle {
    // This is an instance method
    // `&self` is sugar for `self: &Self`, where `Self` is the type of the
    // caller object. In this case `Self` = `Rectangle`
    fn area(&self) -> f64 {
        // `self` gives access to the struct fields via the dot operator
        let Point { x: x1, y: y1 } = self.p1;
        let Point { x: x2, y: y2 } = self.p2;

        // `abs` is a `f64` method that returns the absolute value of the
        // caller
        ((x1 - x2) * (y1 - y2)).abs()
    }

    fn perimeter(&self) -> f64 {
        let Point { x: x1, y: y1 } = self.p1;
        let Point { x: x2, y: y2 } = self.p2;

        2.0 * ((x1 - x2).abs() + (y1 - y2).abs())
    }

    // This method requires the caller object to be mutable
    // `&mut self` desugars to `self: &mut Self`
    fn translate(&mut self, x: f64, y: f64) {
        self.p1.x += x;
        self.p2.x += x;

        self.p1.y += y;
        self.p2.y += y;
    }
}

// `Pair` owns resources: two heap allocated integers
struct Pair(Box<i32>, Box<i32>);

impl Pair {
    // This method "consumes" the resources of the caller object
    // `self` desugars to `self: Self`
    fn destroy(self) {
        // Destructure `self`
        let Pair(first, second) = self;

        println!("Destroying Pair({}, {})", first, second);

        // `first` and `second` go out of scope and get freed
```

```
    }
}

fn main() {
    let rectangle = Rectangle {
        // Static methods are called using double colons
        p1: Point::origin(),
        p2: Point::new(3.0, 4.0),
    };

    // Instance methods are called using the dot operator
    // Note that the first argument `&self` is implicitly passed, i.e.
    // `rectangle.perimeter()` === `Rectangle::perimeter(&rectangle)`
    println!("Rectangle perimeter: {}", rectangle.perimeter());
    println!("Rectangle area: {}", rectangle.area());

    let mut square = Rectangle {
        p1: Point::origin(),
        p2: Point::new(1.0, 1.0),
    };

    // Error! `rectangle` is immutable, but this method requires a mutable
    // object
    //rectangle.translate(1.0, 0.0);
    // TODO ^ Try uncommenting this line

    // Okay! Mutable objects can call mutable methods
    square.translate(1.0, 1.0);

    let pair = Pair(Box::new(1), Box::new(2));

    pair.destroy();

    // Error! Previous `destroy` call "consumed" `pair`
    //pair.destroy();
    // TODO ^ Try uncommenting this line
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Closures

Closures in Rust, also called lambda expressions or lambdas, are functions that can capture the enclosing environment. For example, a closure that captures the x variable:

```
|val| val + x
```

The syntax and capabilities of closures make them very convenient for on the fly usage. Calling a closure is exactly like calling a function. However, both input and return types *can* be inferred and input variable names *must* be specified.

Other characteristics of closures include:

- using `||` instead of `()` around input variables.
- optional body delimination (`{}`) for a single expression (mandatory otherwise).
- the ability to capture the outer environment variables.

```
fn main() {
    // Increment via closures and functions.
    fn  function            (i: i32) -> i32 { i + 1 }

    // Closures are anonymous, here we are binding them to references
    // Annotation is identical to function annotation but is optional
    // as are the `{}` wrapping the body. These nameless functions
    // are assigned to appropriately named variables.
    let closure_annotated = |i: i32| -> i32 { i + 1 };
    let closure_inferred  = |i     |          i + 1  ;

    let i = 1;
    // Call the function and closures.
    println!("function: {}", function(i));
    println!("closure_annotated: {}", closure_annotated(i));
    println!("closure_inferred: {}", closure_inferred(i));

    // A closure taking no arguments which returns an `i32`.
    // The return type is inferred.
    let one = || 1;
    println!("closure returning one: {}", one());

}
```

-

-
-
-
-

Rust By Example

Search this book ...

Capturing

Closures are inherently flexible and will do what the functionality requires to make the closure work without annotation. This allows capturing to flexibly adapt to the use case, sometimes moving and sometimes borrowing. Closures can capture variables:

- by reference: `&T`
- by mutable reference: `&mut T`
- by value: `T`

They preferentially capture variables by reference and only go lower when required.

```
fn main() {
    use std::mem;

    let color = "green";

    // A closure to print `color` which immediately borrows (`&`)
    // `color` and stores the borrow and closure in the `print`
    // variable. It will remain borrowed until `print` goes out of
    // scope. `println!` only requires `by reference` so it doesn't
    // impose anything more restrictive.
    let print = || println!("`color`: {}", color);

    // Call the closure using the borrow.
    print();
    print();

    let mut count = 0;

    // A closure to increment `count` could take either `&mut count`
    // or `count` but `&mut count` is less restrictive so it takes
    // that. Immediately borrows `count`.
    //
    // A `mut` is required on `inc` because a `&mut` is stored inside.
    // Thus, calling the closure mutates the closure which requires
    // a `mut`.
    let mut inc = || {
        count += 1;
        println!("`count`: {}", count);
    };

    // Call the closure.
    inc();
    inc();

    //let _reborrow = &mut count;
    // ^ TODO: try uncommenting this line.

    // A non-copy type.
    let movable = Box::new(3);

    // `mem::drop` requires `T` so this must take by value. A copy type
    // would copy into the closure leaving the original untouched.
    // A non-copy must move and so `movable` immediately moves into
    // the closure.
    let consume = || {
        println!("`movable`: {:?}", movable);
        mem::drop(movable);
    };

    // `consume` consumes the variable so this can only be called once.
    consume();
    //consume();
    // ^ TODO: Try uncommenting this line.
}
```

Using `move` before vertical pipes forces closure to take ownership of captured variables:

```
fn main() {
    // `Vec` has non-copy semantics.
    let haystack = vec![1, 2, 3];

    let contains = move |needle| haystack.contains(needle);

    println!("{}", contains(&1));
    println!("{}", contains(&4));

    // `println!("There're {} elements in vec", haystack.len());`
    // ^ Uncommenting above line will result in compile-time error
    // because borrow checker doesn't allow re-using variable after it
    // has been moved.

    // Removing `move` from closure's signature will cause closure
    // to borrow _haystack_ variable immutably, hence _haystack_ is still
    // available and uncommenting above line will not cause an error.
}
```

See also:

Box and std::mem::drop

-
-
-
-
-

Rust By Example

Search this book ...

As input parameters

While Rust chooses how to capture variables on the fly mostly without type annotation, this ambiguity is not allowed when writing functions. When taking a closure as an input parameter, the closure's complete type must be annotated using one of a few `traits`. In order of decreasing restriction, they are:

- `Fn`: the closure captures by reference (`&T`)
- `FnMut`: the closure captures by mutable reference (`&mut T`)
- `FnOnce`: the closure captures by value (`T`)

On a variable-by-variable basis, the compiler will capture variables in the least restrictive manner possible.

For instance, consider a parameter annotated as `FnOnce`. This specifies that the closure *may* capture by `&T`, `&mut T`, or `T`, but the compiler will ultimately choose based on how the captured variables are used in the closure.

This is because if a move is possible, then any type of borrow should also be possible. Note that the reverse is not true. If the parameter is annotated as `Fn`, then capturing variables by `&mut T` or `T` are not allowed.

In the following example, try swapping the usage of `Fn`, `FnMut`, and `FnOnce` to see what happens:

```
// A function which takes a closure as an argument and calls it.
fn apply<F>(f: F) where
    // The closure takes no input and returns nothing.
    F: FnOnce() {
    // ^ TODO: Try changing this to `Fn` or `FnMut`.

    f();
}

// A function which takes a closure and returns an `i32`.
fn apply_to_3<F>(f: F) -> i32 where
    // The closure takes an `i32` and returns an `i32`.
    F: Fn(i32) -> i32 {

    f(3)
}

fn main() {
    use std::mem;

    let greeting = "hello";
    // A non-copy type.
    // `to_owned` creates owned data from borrowed one
    let mut farewell = "goodbye".to_owned();

    // Capture 2 variables: `greeting` by reference and
    // `farewell` by value.
    let diary = || {
        // `greeting` is by reference: requires `Fn`.
        println!("I said {}.", greeting);

        // Mutation forces `farewell` to be captured by
        // mutable reference. Now requires `FnMut`.
        farewell.push_str("!!!");
        println!("Then I screamed {}.", farewell);
        println!("Now I can sleep. zzzzz");

        // Manually calling drop forces `farewell` to
        // be captured by value. Now requires `FnOnce`.
        mem::drop(farewell);
    };

    // Call the function which applies the closure.
    apply(diary);

    // `double` satisfies `apply_to_3`'s trait bound
    let double = |x| 2 * x;

    println!("3 doubled: {}", apply_to_3(double));
}
```

See also:

std::mem::drop, Fn, FnMut, and FnOnce

Type anonymity

Closures succinctly capture variables from enclosing scopes. Does this have any consequences? It surely does. Observe how using a closure as a function parameter requires generics, which is necessary because of how they are defined:

```
# #![allow(unused_variables)]
#fn main() {
// `F` must be generic.
fn apply<F>(f: F) where
    F: FnOnce() {
    f();
}
#}
```

When a closure is defined, the compiler implicitly creates a new anonymous structure to store the captured variables inside, meanwhile implementing the functionality via one of the `traits`: `Fn`, `FnMut`, or `FnOnce` for this unknown type. This type is assigned to the variable which is stored until calling.

Since this new type is of unknown type, any usage in a function will require generics. However, an unbounded type parameter `<T>` would still be ambiguous and not be allowed. Thus, bounding by one of the `traits`: `Fn`, `FnMut`, or `FnOnce` (which it implements) is sufficient to specify its type.

```
// `F` must implement `Fn` for a closure which takes no
// inputs and returns nothing - exactly what is required
// for `print`.
fn apply<F>(f: F) where
    F: Fn() {
    f();
}

fn main() {
    let x = 7;

    // Capture `x` into an anonymous type and implement
    // `Fn` for it. Store it in `print`.
    let print = || println!("{}", x);

    apply(print);
}
```

See also:

A thorough analysis, `Fn`, `FnMut`, and `FnOnce`

Input functions

Since closures may be used as arguments, you might wonder if the same can be said about functions. And indeed they can! If you declare a function that takes a closure as parameter, then any function that satisfies the trait bound of that closure can be passed as a parameter.

```
// Define a function which takes a generic `F` argument
// bounded by `Fn`, and calls it
fn call_me<F: Fn()>(f: F) {
    f();
}

// Define a wrapper function satisfying the `Fn` bound
```

```
fn function() {
    println!("I'm a function!");
}

fn main() {
    // Define a closure satisfying the `Fn` bound
    let closure = || println!("I'm a closure!");

    call_me(closure);
    call_me(function);
}
```

As an additional note, the `Fn`, `FnMut`, and `FnOnce` `traits` dictate how a closure captures variables from the enclosing scope.

See also:

`Fn`, `FnMut`, and `FnOnce`

-
-
-
-
-

Rust By Example

Search this book ...

As output parameters

Closures as input parameters are possible, so returning closures as output parameters should also be possible. However, returning closure types are problematic because Rust currently only supports returning concrete (non-generic) types. Anonymous closure types are, by definition, unknown and so returning a closure is only possible by making it concrete. This can be done via boxing.

The valid traits for returns are slightly different than before:

- `Fn`: normal
- `FnMut`: normal
- `FnOnce`: There are some unusual things at play here, so the `FnBox` type is currently needed, and is unstable. This is expected to change in the future.

Beyond this, the `move` keyword must be used, which signals that all captures occur by value. This is required because any captures by reference would be dropped as soon as the function exited, leaving invalid references in the closure.

```
fn create_fn() -> Box<Fn()> {
    let text = "Fn".to_owned();

    Box::new(move || println!("This is a: {}", text))
}

fn create_fnmut() -> Box<FnMut()> {
    let text = "FnMut".to_owned();

    Box::new(move || println!("This is a: {}", text))
}

fn main() {
    let fn_plain = create_fn();
    let mut fn_mut = create_fnmut();

    fn_plain();
    fn_mut();
}
```

See also:

Boxing, `Fn`, `FnMut`, and Generics.

-
-
-
-
-

Examples in `std`

This section contains a few examples of using closures from the `std` library.

Rust By Example

Search this book ...

Iterator::any

`Iterator::any` is a function which when passed an iterator, will return `true` if any element satisfies the predicate. Otherwise `false`. Its signature:

```
pub trait Iterator {
    // The type being iterated over.
    type Item;

    // `any` takes `&mut self` meaning the caller may be borrowed
    // and modified, but not consumed.
    fn any<F>(&mut self, f: F) -> bool where
        // `FnMut` meaning any captured variable may at most be
        // modified, not consumed. `Self::Item` states it takes
        // arguments to the closure by value.
        F: FnMut(Self::Item) -> bool {}
}

fn main() {
    let vec1 = vec![1, 2, 3];
    let vec2 = vec![4, 5, 6];

    // `iter()` for vecs yields `&i32`. Destructure to `i32`.
    println!("2 in vec1: {}", vec1.iter()     .any(|&x| x == 2));
    // `into_iter()` for vecs yields `i32`. No destructuring required.
    println!("2 in vec2: {}", vec2.into_iter().any(| x| x == 2));

    let array1 = [1, 2, 3];
    let array2 = [4, 5, 6];

    // `iter()` for arrays yields `&i32`.
    println!("2 in array1: {}", array1.iter()     .any(|&x| x == 2));
    // `into_iter()` for arrays unusually yields `&i32`.
    println!("2 in array2: {}", array2.into_iter().any(|&x| x == 2));
}
```

See also:

`std::iter::Iterator::any`

Rust By Example

Search this book ...

Iterator::find

`Iterator::find` is a function which when passed an iterator, will return the first element which satisfies the predicate as an `Option`. Its signature:

```
pub trait Iterator {
    // The type being iterated over.
    type Item;

    // `find` takes `&mut self` meaning the caller may be borrowed
    // and modified, but not consumed.
    fn find<P>(&mut self, predicate: P) -> Option<Self::Item> where
        // `FnMut` meaning any captured variable may at most be
        // modified, not consumed. `&Self::Item` states it takes
        // arguments to the closure by reference.
        P: FnMut(&Self::Item) -> bool {}
}

fn main() {
    let vec1 = vec![1, 2, 3];
    let vec2 = vec![4, 5, 6];

    // `iter()` for vecs yields `&i32`.
    let mut iter = vec1.iter();
    // `into_iter()` for vecs yields `i32`.
    let mut into_iter = vec2.into_iter();

    // A reference to what is yielded is `&&i32`. Destructure to `i32`.
    println!("Find 2 in vec1: {:?}", iter     .find(|&&x| x == 2));
    // A reference to what is yielded is `&i32`. Destructure to `i32`.
    println!("Find 2 in vec2: {:?}", into_iter.find(| &x| x == 2));

    let array1 = [1, 2, 3];
    let array2 = [4, 5, 6];

    // `iter()` for arrays yields `&i32`
    println!("Find 2 in array1: {:?}", array1.iter()     .find(|&&x| x == 2));
    // `into_iter()` for arrays unusually yields `&i32`
    println!("Find 2 in array2: {:?}", array2.into_iter().find(|&&x| x == 2));
}
```

See also:

`std::iter::Iterator::find`

-
-
-
-
-

Rust By Example

Search this book ...

Higher Order Functions

Rust provides Higher Order Functions (HOF). These are functions that take one or more functions and/or produce a more useful function. HOFs and lazy iterators give Rust its functional flavor.

```
fn is_odd(n: u32) -> bool {
    n % 2 == 1
}

fn main() {
    println!("Find the sum of all the squared odd numbers under 1000");
    let upper = 1000;

    // Imperative approach
    // Declare accumulator variable
    let mut acc = 0;
    // Iterate: 0, 1, 2, ... to infinity
    for n in 0.. {
        // Square the number
        let n_squared = n * n;

        if n_squared >= upper {
            // Break loop if exceeded the upper limit
            break;
        } else if is_odd(n_squared) {
            // Accumulate value, if it's odd
            acc += n_squared;
        }
    }
    println!("imperative style: {}", acc);

    // Functional approach
    let sum_of_squared_odd_numbers: u32 =
        (0..).map(|n| n * n)                             // All natural numbers squared
             .take_while(|&n_squared| n_squared < upper) // Below upper limit
             .filter(|&n_squared| is_odd(n_squared))     // That are odd
             .fold(0, |acc, n_squared| acc + n_squared); // Sum them
    println!("functional style: {}", sum_of_squared_odd_numbers);
}
```

Option and Iterator implement their fair share of HOFs.

Diverging functions

Diverging functions never return. They are marked using `!`, which is an empty type.

```
# #![allow(unused_variables)]
#fn main() {
fn foo() -> ! {
    panic!("This call never returns.");
}
#}
```

As opposed to all the other types, this one cannot be instantiated, because the set of all possible values this type can have is empty. Note, that it is different from the `()` type, which has exactly one possible value.

For example, this functions returns as usual, although there is no information in the return value.

```
fn some_fn() {
    ()
}

fn main() {
    let a: () = some_fn();
    println!("This functions returns and you can see this line.")
}
```

As opposed to this function, which will never return the control back to the caller.

```
#![feature(never_type)]

fn main() {
    let x: ! = panic!("This call never returns.");
    println!("You will never see this line!");
}
```

Although this might seem like an abstract concept, it is in fact very useful and often handy. The main advantage of this type is that it can be cast to any other one and therefore used at places where an exact type is required, for instance in `match` branches. This allows us to write code like this:

```
fn main() {
    fn sum_odd_numbers(up_to: u32) -> u32 {
        let mut acc = 0;
        for i in 0..up_to {
            // Notice that the return type of this match expression must be u32
            // because of the type of the "addition" variable.
            let addition: u32 = match i%2 == 1 {
                // The "i" variable is of type u32, which is perfectly fine.
                true => i,
                // On the other hand, the "continue" expression does not return
                // u32, but it is still fine, because it never returns and therefore
                // does not violate the type requirements of the match expression.
                false => continue,
            };
            acc += addition;
        }
        acc
    }
    println!("Sum of odd numbers up to 9 (excluding): {}", sum_odd_numbers(9));
}
```

It is also the return type of functions that loop forever (e.g. `loop {}`) like network servers or functions that terminates the process (e.g. `exit()`).

Modules

Rust provides a powerful module system that can be used to hierarchically split code in logical units (modules), and manage visibility (public/private) between them.

A module is a collection of items: functions, structs, traits, `impl` blocks, and even other modules.

-
-
-
-
-

Rust By Example

Search this book ...

Visibility

By default, the items in a module have private visibility, but this can be overridden with the `pub` modifier. Only the public items of a module can be accessed from outside the module scope.

```
// A module named `my_mod`
mod my_mod {
    // Items in modules default to private visibility.
    fn private_function() {
        println!("called `my_mod::private_function()`");
    }

    // Use the `pub` modifier to override default visibility.
    pub fn function() {
        println!("called `my_mod::function()`");
    }

    // Items can access other items in the same module,
    // even when private.
    pub fn indirect_access() {
        print!("called `my_mod::indirect_access()`, that\n> ");
        private_function();
    }

    // Modules can also be nested
    pub mod nested {
        pub fn function() {
            println!("called `my_mod::nested::function()`");
        }

        #[allow(dead_code)]
        fn private_function() {
            println!("called `my_mod::nested::private_function()`");
        }

        // Functions declared using `pub(in path)` syntax are only visible
        // within the given path. `path` must be a parent or ancestor module
        pub(in my_mod) fn public_function_in_my_mod() {
            print!("called `my_mod::nested::public_function_in_my_mod()`, that\n > ");
            public_function_in_nested()
        }

        // Functions declared using `pub(self)` syntax are only visible within
        // the current module
        pub(self) fn public_function_in_nested() {
            println!("called `my_mod::nested::public_function_in_nested");
        }

        // Functions declared using `pub(super)` syntax are only visible within
        // the parent module
        pub(super) fn public_function_in_super_mod() {
            println!("called my_mod::nested::public_function_in_super_mod");
        }
    }

    pub fn call_public_function_in_my_mod() {
        print!("called `my_mod::call_public_funcion_in_my_mod()`, that\n> ");
        nested::public_function_in_my_mod();
        print!("> ");
        nested::public_function_in_super_mod();
    }

    // pub(crate) makes functions visible only within the current crate
    pub(crate) fn public_function_in_crate() {
        println!("called `my_mod::public_function_in_crate()");
    }

    // Nested modules follow the same rules for visibility
    mod private_nested {
```

```
        #[allow(dead_code)]
        pub fn function() {
            println!("called `my_mod::private_nested::function()`");
        }
    }
}

fn function() {
    println!("called `function()`");
}

fn main() {
    // Modules allow disambiguation between items that have the same name.
    function();
    my_mod::function();

    // Public items, including those inside nested modules, can be
    // accessed from outside the parent module.
    my_mod::indirect_access();
    my_mod::nested::function();
    my_mod::call_public_function_in_my_mod();

    // pub(crate) items can be called from anywhere in the same crate
    my_mod::public_function_in_crate();

    // pub(in path) items can only be called from within the mode specified
    // Error! function `public_function_in_my_mod` is private
    //my_mod::nested::public_function_in_my_mod();
    // TODO ^ Try uncommenting this line

    // Private items of a module cannot be directly accessed, even if
    // nested in a public module:

    // Error! `private_function` is private
    //my_mod::private_function();
    // TODO ^ Try uncommenting this line

    // Error! `private_function` is private
    //my_mod::nested::private_function();
    // TODO ^ Try uncommenting this line

    // Error! `private_nested` is a private module
    //my_mod::private_nested::function();
    // TODO ^ Try uncommenting this line
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Struct visibility

Structs have an extra level of visibility with their fields. The visibility defaults to private, and can be overridden with the `pub` modifier. This visibility only matters when a struct is accessed from outside the module where it is defined, and has the goal of hiding information (encapsulation).

```
mod my {
    // A public struct with a public field of generic type `T`
    pub struct OpenBox<T> {
        pub contents: T,
    }

    // A public struct with a private field of generic type `T`
    #[allow(dead_code)]
    pub struct ClosedBox<T> {
        contents: T,
    }

    impl<T> ClosedBox<T> {
        // A public constructor method
        pub fn new(contents: T) -> ClosedBox<T> {
            ClosedBox {
                contents: contents,
            }
        }
    }
}

fn main() {
    // Public structs with public fields can be constructed as usual
    let open_box = my::OpenBox { contents: "public information" };

    // and their fields can be normally accessed.
    println!("The open box contains: {}", open_box.contents);

    // Public structs with private fields cannot be constructed using field names.
    // Error! `ClosedBox` has private fields
    //let closed_box = my::ClosedBox { contents: "classified information" };
    // TODO ^ Try uncommenting this line

    // However, structs with private fields can be created using
    // public constructors
    let _closed_box = my::ClosedBox::new("classified information");
```

```
    // and the private fields of a public struct cannot be accessed.
    // Error! The `contents` field is private
    //println!("The closed box contains: {}", _closed_box.contents);
    // TODO ^ Try uncommenting this line
}
```

See also:

generics and methods

-
-
-
-
-

Rust By Example

Search this book ...

The `use` declaration

The `use` declaration can be used to bind a full path to a new name, for easier access.

```
// Bind the `deeply::nested::function` path to `other_function`.
use deeply::nested::function as other_function;

fn function() {
    println!("called `function()`");
}

mod deeply {
    pub mod nested {
        pub fn function() {
            println!("called `deeply::nested::function()`");
        }
    }
}

fn main() {
    // Easier access to `deeply::nested::function`
    other_function();

    println!("Entering block");
    {
        // This is equivalent to `use deeply::nested::function as function`.
        // This `function()` will shadow the outer one.
        use deeply::nested::function;
        function();

        // `use` bindings have a local scope. In this case, the
        // shadowing of `function()` is only in this block.
        println!("Leaving block");
    }

    function();
}
```

-
-
-
-
-

Rust By Example

Search this book ...

`super` and `self`

The `super` and `self` keywords can be used in the path to remove ambiguity when accessing items and to prevent unnecessary hardcoding of paths.

```
fn function() {
```

```
    println!("called `function()`");
}

mod cool {
    pub fn function() {
        println!("called `cool::function()`");
    }
}

mod my {
    fn function() {
        println!("called `my::function()`");
    }

    mod cool {
        pub fn function() {
            println!("called `my::cool::function()`");
        }
    }

    pub fn indirect_call() {
        // Let's access all the functions named `function` from this scope!
        print!("called `my::indirect_call()`, that\n> ");

        // The `self` keyword refers to the current module scope - in this case `my`.
        // Calling `self::function()` and calling `function()` directly both give
        // the same result, because they refer to the same function.
        self::function();
        function();

        // We can also use `self` to access another module inside `my`:
        self::cool::function();

        // The `super` keyword refers to the parent scope (outside the `my` module).
        super::function();

        // This will bind to the `cool::function` in the *crate* scope.
        // In this case the crate scope is the outermost scope.
        {
            use cool::function as root_function;
            root_function();
        }
    }
}

fn main() {
    my::indirect_call();
}
```

-
-
-
-
-

Rust By Example

Search this book ...

File hierarchy

Modules can be mapped to a file/directory hierarchy. Let's break down the visibility example in files:

```
$ tree .
.
|-- my
|   |-- inaccessible.rs
|   |-- mod.rs
|   `-- nested.rs
`-- split.rs
```

In `split.rs`:

```
// This declaration will look for a file named `my.rs` or `my/mod.rs` and will
// insert its contents inside a module named `my` under this scope
mod my;

fn function() {
    println!("called `function()`");
}

fn main() {
    my::function();

    function();

    my::indirect_access();

    my::nested::function();
}
```

In `my/mod.rs`:

```
// Similarly `mod inaccessible` and `mod nested` will locate the `nested.rs`
// and `inaccessible.rs` files and insert them here under their respective
```

```
// modules
mod inaccessible;
pub mod nested;

pub fn function() {
    println!("called `my::function()`");
}

fn private_function() {
    println!("called `my::private_function()`");
}

pub fn indirect_access() {
    print!("called `my::indirect_access()`, that\n> ");

    private_function();
}
```

In `my/nested.rs`:

```
pub fn function() {
    println!("called `my::nested::function()`");
}

#[allow(dead_code)]
fn private_function() {
    println!("called `my::nested::private_function()`");
}
```

In `my/inaccessible.rs`:

```
#[allow(dead_code)]
pub fn public_function() {
    println!("called `my::inaccessible::public_function()`");
}
```

Let's check that things still work as before:

```
$ rustc split.rs && ./split
called `my::function()`
called `function()`
called `my::indirect_access()`, that
> called `my::private_function()`
called `my::nested::function()`
```


Crates

A crate is a compilation unit in Rust. Whenever `rustc some_file.rs` is called, `some_file.rs` is treated as the *crate file*. If `some_file.rs` has `mod` declarations in it, then the contents of the module files would be inserted in places where `mod` declarations in the crate file are found, *before* running the compiler over it. In other words, modules do *not* get compiled individually, only crates get compiled.

A crate can be compiled into a binary or into a library. By default, `rustc` will produce a binary from a crate. This behavior can be overridden by passing the `--crate-type` flag to `rustc`.

Library

Let's create a library, and then see how to link it to another crate.

```
pub fn public_function() {
    println!("called rary's `public_function()`");
}

fn private_function() {
    println!("called rary's `private_function()`");
}

pub fn indirect_access() {
    print!("called rary's `indirect_access()`, that\n> ");

    private_function();
}
```

```
$ rustc --crate-type=lib rary.rs
$ ls lib*
library.rlib
```

Libraries get prefixed with "lib", and by default they get named after their crate file, but this default name can be overridden using the `crate_name` attribute.

-
-
-
-
-

`extern crate`

To link a crate to this new library, the `extern crate` declaration must be used. This will not only link the library, but also import all its items under a module named the same as the library. The visibility rules that apply to modules also apply to libraries.

```
// Link to `library`, import items under the `rary` module
extern crate rary;

fn main() {
    rary::public_function();

    // Error! `private_function` is private
    //rary::private_function();

    rary::indirect_access();
}
```

```
# Where library.rlib is the path to the compiled library, assumed that it's
# in the same directory here:
$ rustc executable.rs --extern rary=library.rlib && ./executable
called rary's `public_function()`
called rary's `indirect_access()`, that
> called rary's `private_function()`
```

-
-
-
-
-

Rust By Example

Search this book ...

Cargo

`cargo` is the official Rust package management tool. It has lots of really useful features to improve code quality and developer velocity! These include

- Dependency management and integration with crates.io (the official Rust package registry)

- Awareness of unit tests
- Awareness of benchmarks

This chapter will go through some quick basics, but you can find the comprehensive docs in The Cargo Book.

Rust By Example

Search this book ...

Dependencies

Most programs have dependencies on some libraries. If you have ever managed dependencies by hand, you know how much of a pain this can be. Luckily, the Rust ecosystem comes standard with `cargo`! `cargo` can manage dependencies for a project.

To create a new Rust project,

```
# A binary
cargo new foo

# OR A library
cargo new --lib foo
```

For the rest of this chapter, I will assume we are making a binary, rather than a library, but all of the concepts are the same.

After the above commands, you should see something like this:

```
foo
â”œâ”€â”€ Cargo.toml
â””â”€â”€ src
    â””â”€â”€ main.rs
```

The `main.rs` is the root source file for your new project -- nothing new there. The `Cargo.toml` is the config file for `cargo` for this project (`foo`). If you look inside it, you should see something like this:

```
[package]
name = "foo"
version = "0.1.0"
authors = ["mark"]

[dependencies]
```

The `name` field under `package` determines the name of the project. This is used by `crates.io` if you publish the crate (more later). It is also the name of the output binary when you compile.

The `version` field is a crate version number using Semantic Versioning.

The `authors` field is a list of authors used when publishing the crate.

The `dependencies` section lets you add a dependency for your project.

For example, suppose that I want my program to have a great CLI. You can find lots of great packages on crates.io (the official Rust package registry). One popular choice is clap. As of this writing, the most recent published version of `clap` is `2.27.1`. To add a dependency to our program, we can simply add the following to our `Cargo.toml` under `dependencies`: `clap = "2.27.1"`. And of course, `extern crate clap` in `main.rs`, just like normal. And that's it! You can start using `clap` in your program.

`cargo` also supports other types of dependencies. Here is just a small sampling:

```
[package]
name = "foo"
version = "0.1.0"
authors = ["mark"]

[dependencies]
clap = "2.27.1" # from crates.io
rand = { git = "https://github.com/rust-lang-nursery/rand" } # from online repo
bar = { path = "../bar" } # from a path in the local filesystem
```

`cargo` is more than a dependency manager. All of the available configuration options are listed in the format specification of `Cargo.toml`.

To build our project we can execute `cargo build` anywhere in the project directory (including subdirectories!). We can also do `cargo run` to build and run. Notice that these commands will resolve all dependencies, download crates if needed, and build everything, including your crate. (Note that it only rebuilds what it has not already built, similar to `make`).

Voila! That's all there is to it!

-
-
-
-
-

Rust By Example

Search this book ...

Conventions

In the previous chapter, we saw the following directory hierarchy:

```
foo
â”œâ”€â”€ Cargo.toml
â””â”€â”€ src
    â””â”€â”€ main.rs
```

Suppose that we wanted to have two binaries in the same project, though. What then?

It turns out that `cargo` supports this. The default binary name is `main.rs`, as we saw before, but you can add additional binaries by placing them in a `bin/` directory:

```
foo
â”œâ”€â”€ Cargo.toml
â””â”€â”€ src
    â”œâ”€â”€ main.rs
    â””â”€â”€ bin
        â””â”€â”€ my_other_bin.rs
```

To tell `cargo` to compile or run this binary as opposed to the default or other binaries, we just pass `cargo` the `--bin my_other_bin` flag, where `my_other_bin` is the name of the binary we want to work with.

In addition to extra binaries, `cargo` supports more features such as benchmarks, tests, and examples.

In the next chapter, we will look more closely at tests.

-
-
-
-
-

Rust By Example

Search this book ...

Testing

As we know testing is integral to any piece of software! Rust has first-class support for unit and integration testing (see this chapter in TRPL).

From the testing chapters linked above, we see how to write unit tests and integration tests. Organizationally, we can place unit tests in the modules they test and integration tests in their own `tests/` directory:

```
foo
â”œâ”€â”€ Cargo.toml
â”œâ”€â”€ src
â”‚   â””â”€â”€ main.rs
â””â”€â”€ tests
    â”œâ”€â”€ my_test.rs
    â””â”€â”€ my_other_test.rs
```

Each file in `tests` is a separate integration test.

`cargo` naturally provides an easy way to run all of your tests!

```
cargo test
```

You should see output like this:

```
$ cargo test
   Compiling blah v0.1.0 (file:///nobackup/blah)
    Finished dev [unoptimized + debuginfo] target(s) in 0.89 secs
```

```
     Running target/debug/deps/blah-d3b32b97275ec472

running 3 tests
test test_bar ... ok
test test_baz ... ok
test test_foo_bar ... ok
test test_foo ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

You can also run tests whose name matches a pattern:

```
cargo test test_foo
```

```
$ cargo test test_foo
   Compiling blah v0.1.0 (file:///nobackup/blah)
    Finished dev [unoptimized + debuginfo] target(s) in 0.35 secs
     Running target/debug/deps/blah-d3b32b97275ec472

running 2 tests
test test_foo ... ok
test test_foo_bar ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out
```

One word of caution: Cargo may run multiple tests concurrently, so make sure that they don't race with each other. For example, if they all output to a file, you should make them write to different files.

Rust By Example

Search this book ...

Attributes

An attribute is metadata applied to some module, crate or item. This metadata can be used to/for:

- conditional compilation of code
- set crate name, version and type (binary or library)
- disable lints (warnings)
- enable compiler features (macros, glob imports, etc.)
- link to a foreign library
- mark functions as unit tests
- mark functions that will be part of a benchmark

When attributes apply to a whole crate, their syntax is `#![crate_attribute]`, and when they apply to a module or item, the syntax is `#[item_attribute]` (notice the missing bang `!`).

Attributes can take arguments with different syntaxes:

- `#[attribute = "value"]`
- `#[attribute(key = "value")]`
- `#[attribute(value)]`

Attributes can have multiple values and can be separated over multiple lines, too:

```
#[attribute(value, value2)]


#[attribute(value, value2, value3,
            value4, value5)]
```

Rust By Example

Search this book ...

dead_code

The compiler provides a `dead_code` *lint* that will warn about unused functions. An *attribute* can be used to disable the lint.

```
fn used_function() {}

// `#[allow(dead_code)]` is an attribute that disables the `dead_code` lint
#[allow(dead_code)]
fn unused_function() {}

fn noisy_unused_function() {}
// FIXME ^ Add an attribute to suppress the warning

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

Note that in real programs, you should eliminate dead code. In these examples we'll allow dead code in some places because of the interactive nature of the examples.

Rust By Example

Search this book ...

Crates

The `crate_type` attribute can be used to tell the compiler whether a crate is a binary or a library (and even which type of library), and the `crate_name` attribute can be used to set the name of the crate.

However, it is important to note that both the `crate_type` and `crate_name` attributes have **no** effect whatsoever when using Cargo, the Rust package manager. Since Cargo is used for the majority of Rust projects, this means real-world uses of `crate_type` and `crate_name` are relatively limited.

```
// This crate is a library
#![crate_type = "lib"]
// The library is named "rary"
#![crate_name = "rary"]

pub fn public_function() {
    println!("called rary's `public_function()`");
}

fn private_function() {
    println!("called rary's `private_function()`");
}

pub fn indirect_access() {
    print!("called rary's `indirect_access()`, that\n> ");

    private_function();
}
```

When the `crate_type` attribute is used, we no longer need to pass the `--crate-type` flag to `rustc`.

```
$ rustc lib.rs
$ ls lib*
library.rlib
```

Search this book ...

cfg

Conditional compilation is possible through two different operators:

- the `cfg` attribute: `#[cfg(...)]` in attribute position
- the `cfg!` macro: `cfg!(...)` in boolean expressions

Both utilize identical argument syntax.

```
// This function only gets compiled if the target OS is linux
#[cfg(target_os = "linux")]
fn are_you_on_linux() {
    println!("You are running linux!");
}

// And this function only gets compiled if the target OS is *not* linux
#[cfg(not(target_os = "linux"))]
fn are_you_on_linux() {
    println!("You are *not* running linux!");
}

fn main() {
    are_you_on_linux();

    println!("Are you sure?");
    if cfg!(target_os = "linux") {
        println!("Yes. It's definitely linux!");
    } else {
        println!("Yes. It's definitely *not* linux!");
    }
}
```

See also:

the reference, `cfg!`, and macros.

Search this book ...

Custom

Some conditionals like `target_os` are implicitly provided by `rustc`, but custom conditionals must be passed to `rustc` using the `--cfg` flag.

```
#[cfg(some_condition)]
fn conditional_function() {
    println!("condition met!");
}

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

Try to run this to see what happens without the custom `cfg` flag.

With the custom `cfg` flag:

```
$ rustc --cfg some_condition custom.rs && ./custom
condition met!
```

Generics

Generics is the topic of generalizing types and functionalities to broader cases. This is extremely useful for reducing code duplication in many ways, but can call for rather involving syntax. Namely, being generic requires taking great care to specify over which types a generic type is actually considered valid. The simplest and most common use of generics is for type parameters.

A type parameter is specified as generic by the use of angle brackets and upper camel case: `<Aaa, Bbb, ...>`. "Generic type parameters" are typically represented as `<T>`. In Rust, "generic" also describes anything that accepts one or more generic type parameters `<T>`. Any type specified as a generic type parameter is generic, and everything else is concrete (non-generic).

For example, defining a *generic function* named `foo` that takes an argument `T` of any type:

```
fn foo<T>(arg: T) { ... }
```

Because `T` has been specified as a generic type parameter using `<T>`, it is considered generic when used here as `(arg: T)`. This is the case even if `T` has previously been defined as a `struct`.

This example shows some of the syntax in action:

```
// A concrete type `A`.
struct A;

// In defining the type `Single`, the first use of `A` is not preceded by `<A>`.
// Therefore, `Single` is a concrete type, and `A` is defined as above.
struct Single(A);
//            ^ Here is `Single`s first use of the type `A`.

// Here, `<T>` precedes the first use of `T`, so `SingleGen` is a generic type.
// Because the type parameter `T` is generic, it could be anything, including
// the concrete type `A` defined at the top.
struct SingleGen<T>(T);

fn main() {
    // `Single` is concrete and explicitly takes `A`.
    let _s = Single(A);

    // Create a variable `_char` of type `SingleGen<char>`
    // and give it the value `SingleGen('a')`.
    // Here, `SingleGen` has a type parameter explicitly specified.
    let _char: SingleGen<char> = SingleGen('a');

    // `SingleGen` can also have a type parameter implicitly specified:
    let _t    = SingleGen(A); // Uses `A` defined at the top.
    let _i32  = SingleGen(6); // Uses `i32`.
    let _char = SingleGen('a'); // Uses `char`.
}
```

See also:

`structs`

Functions

The same set of rules can be applied to functions: a type `T` becomes generic when preceded by `<T>`.

Using generic functions sometimes requires explicitly specifying type parameters. This may be the case if the function is called where the return type is generic, or if the compiler doesn't have enough information to infer the necessary type parameters.

A function call with explicitly specified type parameters looks like: `fun::<A, B, ...>()`.

```
struct A;          // Concrete type `A`.
struct S(A);       // Concrete type `S`.
struct SGen<T>(T); // Generic type `SGen`.

// The following functions all take ownership of the variable passed into
```

```
// them and immediately go out of scope, freeing the variable.

// Define a function `reg_fn` that takes an argument `_s` of type `S`.
// This has no `<T>` so this is not a generic function.
fn reg_fn(_s: S) {}

// Define a function `gen_spec_t` that takes an argument `_s` of type `SGen<T>`.
// It has been explicitly given the type parameter `A`, but because `A` has not
// been specified as a generic type parameter for `gen_spec_t`, it is not generic.
fn gen_spec_t(_s: SGen<A>) {}

// Define a function `gen_spec_i32` that takes an argument `_s` of type `SGen<i32>`.
// It has been explicitly given the type parameter `i32`, which is a specific type.
// Because `i32` is not a generic type, this function is also not generic.
fn gen_spec_i32(_s: SGen<i32>) {}

// Define a function `generic` that takes an argument `_s` of type `SGen<T>`.
// Because `SGen<T>` is preceded by `<T>`, this function is generic over `T`.
fn generic<T>(_s: SGen<T>) {}

fn main() {
    // Using the non-generic functions
    reg_fn(S(A));          // Concrete type.
    gen_spec_t(SGen(A));   // Implicitly specified type parameter `A`.
    gen_spec_i32(SGen(6)); // Implicitly specified type parameter `i32`.

    // Explicitly specified type parameter `char` to `generic()`.
    generic::<char>(SGen('a'));

    // Implicitly specified type parameter `char` to `generic()`.
    generic(SGen('c'));
}
```

See also:

functions and `structs`

-
-
-
-
-

Rust By Example

Search this book ...

Implementation

Similar to functions, implementations require care to remain generic.

```
# #![allow(unused_variables)]
#fn main() {
struct S; // Concrete type `S`
struct GenericVal<T>(T,); // Generic type `GenericVal`

// impl of GenericVal where we explicitly specify type parameters:
impl GenericVal<f32> {} // Specify `f32`
impl GenericVal<S> {} // Specify `S` as defined above

// `<T>` Must precede the type to remain generic
impl <T> GenericVal<T> {}
#}

struct Val {
    val: f64
}

struct GenVal<T>{
    gen_val: T
}

// impl of Val
impl Val {
    fn value(&self) -> &f64 { &self.val }
}

// impl of GenVal for a generic type `T`
impl <T> GenVal<T> {
    fn value(&self) -> &T { &self.gen_val }
}

fn main() {
    let x = Val { val: 3.0 };
    let y = GenVal { gen_val: 3i32 };

    println!("{}, {}", x.value(), y.value());
}
```

See also:

functions returning references, `impl`, and `struct`

-
-
-
-
-

Rust By Example

Search this book ...

Traits

Of course `traits` can also be generic. Here we define one which reimplements the `Drop` `trait` as a generic method to `drop` itself and an input.

```
// Non-copyable types.
struct Empty;
struct Null;

// A trait generic over `T`.
trait DoubleDrop<T> {
    // Define a method on the caller type which takes an
    // additional single parameter `T` and does nothing with it.
    fn double_drop(self, _: T);
}

// Implement `DoubleDrop<T>` for any generic parameter `T` and
// caller `U`.
impl<T, U> DoubleDrop<T> for U {
    // This method takes ownership of both passed arguments,
    // deallocating both.
    fn double_drop(self, _: T) {}
}

fn main() {
    let empty = Empty;
    let null  = Null;

    // Deallocate `empty` and `null`.
    empty.double_drop(null);

    //empty;
    //null;
    // ^ TODO: Try uncommenting these lines.
}
```

See also:

`Drop`, `struct`, and `trait`

-
-
-
-
-

Rust By Example

Search this book ...

Bounds

When working with generics, the type parameters often must use traits as *bounds* to stipulate what functionality a type implements. For example, the following example uses the trait `Display` to print and so it requires `T` to be bound by `Display`; that is, `T` *must* implement `Display`.

```
// Define a function `printer` that takes a generic type `T` which
// must implement trait `Display`.
fn printer<T: Display>(t: T) {
    println!("{}", t);
}
```

Bounding restricts the generic to types that conform to the bounds. That is:

```
struct S<T: Display>(T);
```

```
// Error! `Vec<T>` does not implement `Display`. This
// specialization will fail.
let s = S(vec![1]);
```

Another effect of bounding is that generic instances are allowed to access the methods of traits specified in the bounds. For example:

```
// A trait which implements the print marker: `{:?}`.
use std::fmt::Debug;

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Rectangle {
    fn area(&self) -> f64 { self.length * self.height }
}

#[derive(Debug)]
struct Rectangle { length: f64, height: f64 }
#[allow(dead_code)]
struct Triangle  { length: f64, height: f64 }

// The generic `T` must implement `Debug`. Regardless
// of the type, this will work properly.
fn print_debug<T: Debug>(t: &T) {
    println!("{:?}", t);
}

// `T` must implement `HasArea`. Any function which meets
// the bound can access `HasArea`'s function `area`.
fn area<T: HasArea>(t: &T) -> f64 { t.area() }

fn main() {
    let rectangle = Rectangle { length: 3.0, height: 4.0 };
    let _triangle = Triangle  { length: 3.0, height: 4.0 };

    print_debug(&rectangle);
    println!("Area: {}", area(&rectangle));

    //print_debug(&_triangle);
    //println!("Area: {}", area(&_triangle));
    // ^ TODO: Try uncommenting these.
    // | Error: Does not implement either `Debug` or `HasArea`.
}
```

As an additional note, `where` clauses can also be used to apply bounds in some cases to be more expressive.

See also:

`std::fmt`, `struct`s, and `trait`s

Rust By Example

Search this book ...

Testcase: empty bounds

A consequence of how bounds work is that even if a `trait` doesn't include any functionality, you can still use it as a bound. `Eq` and `Ord` are examples of such `trait`s from the `std` library.

```
struct Cardinal;
struct BlueJay;
struct Turkey;

trait Red {}
trait Blue {}

impl Red for Cardinal {}
impl Blue for BlueJay {}

// These functions are only valid for types which implement these
// traits. The fact that the traits are empty is irrelevant.
fn red<T: Red>(_: &T)   -> &'static str { "red" }
fn blue<T: Blue>(_: &T) -> &'static str { "blue" }

fn main() {
    let cardinal = Cardinal;
    let blue_jay = BlueJay;
    let _turkey   = Turkey;

    // `red()` won't work on a blue jay nor vice versa
    // because of the bounds.
    println!("A cardinal is {}", red(&cardinal));
    println!("A blue jay is {}", blue(&blue_jay));
    //println!("A turkey is {}", red(&_turkey));
    // ^ TODO: Try uncommenting this line.
}
```

See also:

`std::cmp::Eq`, `std::cmp::Ord`s, and `trait`s

-
-
-
-
-

Rust By Example

Search this book ...

Multiple bounds

Multiple bounds can be applied with a `+`. Like normal, different types are separated with `,`.

```
use std::fmt::{Debug, Display};

fn compare_prints<T: Debug + Display>(t: &T) {
    println!("Debug: `{:?}`", t);
    println!("Display: `{}`", t);
}

fn compare_types<T: Debug, U: Debug>(t: &T, u: &U) {
    println!("t: `{:?}`", t);
    println!("u: `{:?}`", u);
}

fn main() {
    let string = "words";
    let array = [1, 2, 3];
    let vec = vec![1, 2, 3];

    compare_prints(&string);
    //compare_prints(&array);
    // TODO ^ Try uncommenting this.

    compare_types(&array, &vec);
}
```

See also:

`std::fmt` and `trait`s

-
-
-
-
-

Rust By Example

Search this book ...

Where clauses

A bound can also be expressed using a `where` clause immediately before the opening `{`, rather than at the type's first mention. Additionally, `where` clauses can apply bounds to arbitrary types, rather than just to type parameters.

Some cases that a `where` clause is useful:

- When specifying generic types and bounds separately is clearer:

```
impl <A: TraitB + TraitC, D: TraitE + TraitF> MyTrait<A, D> for YourType {}

// Expressing bounds with a `where` clause
impl <A, D> MyTrait<A, D> for YourType where
    A: TraitB + TraitC,
    D: TraitE + TraitF {}
```

- When using a `where` clause is more expressive than using normal syntax. The `impl` in this example cannot be directly expressed without a `where` clause:

```
use std::fmt::Debug;

trait PrintInOption {
    fn print_in_option(self);
}

// Because we would otherwise have to express this as `T: Debug` or
// use another method of indirect approach, this requires a `where` clause:
impl<T> PrintInOption for T where
    Option<T>: Debug {
    // We want `Option<T>: Debug` as our bound because that is what's
    // being printed. Doing otherwise would be using the wrong bound.
    fn print_in_option(self) {
        println!("{:?}", Some(self));
    }
}

fn main() {
    let vec = vec![1, 2, 3];

    vec.print_in_option();
}
```

See also:

RFC, `struct`, and `trait`

-
-
-
-
-

Rust By Example

Search this book ...

New Type Idiom

The `newtype` idiom gives compile time guarantees that the right type of value is supplied to a program.

For example, an age verification function that checks age in years, *must* be given a value of type `Years`.

```
struct Years(i64);

struct Days(i64);

impl Years {
    pub fn to_days(&self) -> Days {
        Days(self.0 * 365)
    }
}


impl Days {
    /// truncates partial years
    pub fn to_years(&self) -> Years {
        Years(self.0 / 365)
    }
}

fn old_enough(age: &Years) -> bool {
    age.0 >= 18
}

fn main() {
    let age = Years(5);
    let age_days = age.to_days();
    println!("Old enough {}", old_enough(&age));
    println!("Old enough {}", old_enough(&age_days.to_years()));
    // println!("Old enough {}", old_enough(&age_days));
}
```

Uncomment the last print statement to observe that the type supplied must be `Years`.

See also:

`structs`

-
-
-

-
-

Rust By Example

Search this book ...

Associated items

"Associated Items" refers to a set of rules pertaining to `item`s of various types. It is an extension to `trait` generics, and allows `traits` to internally define new items.

One such item is called an *associated type*, providing simpler usage patterns when the `trait` is generic over its container type.

See also:

RFC

-
-
-
-
-

Rust By Example

Search this book ...

The Problem

A `trait` that is generic over its container type has type specification requirements - users of the `trait` *must* specify all of its generic types.

In the example below, the `Contains` `trait` allows the use of the generic types `A` and `B`. The trait is then implemented for the `Container` type, specifying `i32` for `A` and `B` so that it can be used with `fn difference()`.

Because `Contains` is generic, we are forced to explicitly state *all* of the generic types for `fn difference()`. In practice, we want a way to express that `A` and `B` are determined by the *input* `C`. As you will see in the next section, associated types provide exactly that capability.

```
struct Container(i32, i32);

// A trait which checks if 2 items are stored inside of container.
// Also retrieves first or last value.
trait Contains<A, B> {
    fn contains(&self, &A, &B) -> bool; // Explicitly requires `A` and `B`.
    fn first(&self) -> i32; // Doesn't explicitly require `A` or `B`.
    fn last(&self) -> i32;  // Doesn't explicitly require `A` or `B`.
}

impl Contains<i32, i32> for Container {
    // True if the numbers stored are equal.
    fn contains(&self, number_1: &i32, number_2: &i32) -> bool {
        (&self.0 == number_1) && (&self.1 == number_2)
    }

    // Grab the first number.
    fn first(&self) -> i32 { self.0 }

    // Grab the last number.
    fn last(&self) -> i32 { self.1 }
}

// `C` contains `A` and `B`. In light of that, having to express `A` and
// `B` again is a nuisance.
fn difference<A, B, C>(container: &C) -> i32 where
    C: Contains<A, B> {
    container.last() - container.first()
}

fn main() {
    let number_1 = 3;
    let number_2 = 10;

    let container = Container(number_1, number_2);

    println!("Does container contain {} and {}: {}",
        &number_1, &number_2,
        container.contains(&number_1, &number_2));
    println!("First number: {}", container.first());
    println!("Last number: {}", container.last());
```

```
    println!("The difference is: {}", difference(&container));
}
```

See also:

`structs`, and `traits`

-
-
-
-
-

Rust By Example

Search this book ...

Associated types

The use of "Associated types" improves the overall readability of code by moving inner types locally into a trait as *output* types. Syntax for the `trait` definition is as follows:

```
# #![allow(unused_variables)]
#fn main() {
// `A` and `B` are defined in the trait via the `type` keyword.
// (Note: `type` in this context is different from `type` when used for
// aliases).
trait Contains {
    type A;
    type B;

    // Updated syntax to refer to these new types generically.
    fn contains(&self, &Self::A, &Self::B) -> bool;
}
#}
```

Note that functions that use the `trait Contains` are no longer required to express `A` or `B` at all:

```
// Without using associated types
fn difference<A, B, C>(container: &C) -> i32 where
    C: Contains<A, B> { ... }

// Using associated types
fn difference<C: Contains>(container: &C) -> i32 { ... }
```

Let's rewrite the example from the previous section using associated types:

```
struct Container(i32, i32);

// A trait which checks if 2 items are stored inside of container.
// Also retrieves first or last value.
trait Contains {
    // Define generic types here which methods will be able to utilize.
    type A;
    type B;

    fn contains(&self, &Self::A, &Self::B) -> bool;
    fn first(&self) -> i32;
    fn last(&self) -> i32;
}

impl Contains for Container {
    // Specify what types `A` and `B` are. If the `input` type
    // is `Container(i32, i32)`, the `output` types are determined
    // as `i32` and `i32`.
    type A = i32;
    type B = i32;

    // `&Self::A` and `&Self::B` are also valid here.
    fn contains(&self, number_1: &i32, number_2: &i32) -> bool {
        (&self.0 == number_1) && (&self.1 == number_2)
    }
    // Grab the first number.
    fn first(&self) -> i32 { self.0 }

    // Grab the last number.
    fn last(&self) -> i32 { self.1 }
}

fn difference<C: Contains>(container: &C) -> i32 {
    container.last() - container.first()
}

fn main() {
    let number_1 = 3;
    let number_2 = 10;

    let container = Container(number_1, number_2);

    println!("Does container contain {} and {}: {}",
        &number_1, &number_2,
        container.contains(&number_1, &number_2));
```

```
    println!("First number: {}", container.first());
    println!("Last number: {}", container.last());

    println!("The difference is: {}", difference(&container));
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Phantom type parameters

A phantom type parameter is one that doesn't show up at runtime, but is checked statically (and only) at compile time.

Data types can use extra generic type parameters to act as markers or to perform type checking at compile time. These extra parameters hold no storage values, and have no runtime behavior.

In the following example, we combine std::marker::PhantomData with the phantom type parameter concept to create tuples containing different data types.

```
use std::marker::PhantomData;

// A phantom tuple struct which is generic over `A` with hidden parameter `B`.
#[derive(PartialEq)] // Allow equality test for this type.
struct PhantomTuple<A, B>(A,PhantomData<B>);

// A phantom type struct which is generic over `A` with hidden parameter `B`.
#[derive(PartialEq)] // Allow equality test for this type.
struct PhantomStruct<A, B> { first: A, phantom: PhantomData<B> }

// Note: Storage is allocated for generic type `A`, but not for `B`.
//       Therefore, `B` cannot be used in computations.

fn main() {
    // Here, `f32` and `f64` are the hidden parameters.
    // PhantomTuple type specified as `<char, f32>`.
    let _tuple1: PhantomTuple<char, f32> = PhantomTuple('Q', PhantomData);
    // PhantomTuple type specified as `<char, f64>`.
    let _tuple2: PhantomTuple<char, f64> = PhantomTuple('Q', PhantomData);

    // Type specified as `<char, f32>`.
    let _struct1: PhantomStruct<char, f32> = PhantomStruct {
        first: 'Q',
        phantom: PhantomData,
    };
    // Type specified as `<char, f64>`.
    let _struct2: PhantomStruct<char, f64> = PhantomStruct {
        first: 'Q',
        phantom: PhantomData,
    };

    // Compile-time Error! Type mismatch so these cannot be compared:
    //println!("_tuple1 == _tuple2 yields: {}",
    //          _tuple1 == _tuple2);

    // Compile-time Error! Type mismatch so these cannot be compared:
    //println!("_struct1 == _struct2 yields: {}",
    //          _struct1 == _struct2);
}
```

See also:

Derive, struct, and TupleStructs

-
-
-
-
-

Rust By Example

Search this book ...

Testcase: unit clarification

A useful method of unit conversions can be examined by implementing `Add` with a phantom type parameter. The `Add` `trait` is examined below:

```
// This construction would impose: `Self + RHS = Output`
// where RHS defaults to Self if not specified in the implementation.
pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}

// `Output` must be `T<U>` so that `T<U> + T<U> = T<U>`.
impl<U> Add for T<U> {
    type Output = T<U>;
    ...
}
```

The whole implementation:

```
use std::ops::Add;
use std::marker::PhantomData;

/// Create void enumerations to define unit types.
#[derive(Debug, Clone, Copy)]
enum Inch {}
#[derive(Debug, Clone, Copy)]
enum Mm {}

/// `Length` is a type with phantom type parameter `Unit`,
/// and is not generic over the length type (that is `f64`).
///
/// `f64` already implements the `Clone` and `Copy` traits.
#[derive(Debug, Clone, Copy)]
struct Length<Unit>(f64, PhantomData<Unit>);

/// The `Add` trait defines the behavior of the `+` operator.
impl<Unit> Add for Length<Unit> {
     type Output = Length<Unit>;

    // add() returns a new `Length` struct containing the sum.
    fn add(self, rhs: Length<Unit>) -> Length<Unit> {
        // `+` calls the `Add` implementation for `f64`.
        Length(self.0 + rhs.0, PhantomData)
    }
}

fn main() {
    // Specifies `one_foot` to have phantom type parameter `Inch`.
    let one_foot:  Length<Inch> = Length(12.0, PhantomData);
    // `one_meter` has phantom type parameter `Mm`.
    let one_meter: Length<Mm>   = Length(1000.0, PhantomData);

    // `+` calls the `add()` method we implemented for `Length<Unit>`.
    //
    // Since `Length` implements `Copy`, `add()` does not consume
    // `one_foot` and `one_meter` but copies them into `self` and `rhs`.
    let two_feet = one_foot + one_foot;
    let two_meters = one_meter + one_meter;

    // Addition works.
    println!("one foot + one_foot = {:?} in", two_feet.0);
    println!("one meter + one_meter = {:?} mm", two_meters.0);

    // Nonsensical operations fail as they should:
    // Compile-time Error: type mismatch.
    //let one_feter = one_foot + one_meter;
}
```

See also:

[Borrowing (`&`)], [Bounds (`X: Y`)], [enum], [impl & self], [Overloading], [ref], [Traits (`X for Y`)], and [TupleStructs].

-
-
-
-
-

Rust By Example

Search this book ...

Scoping rules

Scopes play an important part in ownership, borrowing, and lifetimes. That is, they indicate to the compiler when borrows are valid, when resources can be freed, and when variables are created or destroyed.

-
-
-
-
-

Rust By Example

Search this book ...

RAII

Variables in Rust do more than just hold data in the stack: they also *own* resources, e.g. `Box<T>` owns memory in the heap. Rust enforces RAII (Resource Acquisition Is Initialization), so whenever an object goes out of scope, its destructor is called and its owned resources are freed.

This behavior shields against *resource leak* bugs, so you'll never have to manually free memory or worry about memory leaks again! Here's a quick showcase:

```
// raii.rs
fn create_box() {
    // Allocate an integer on the heap
    let _box1 = Box::new(3i32);

    // `_box1` is destroyed here, and memory gets freed
}

fn main() {
    // Allocate an integer on the heap
    let _box2 = Box::new(5i32);

    // A nested scope:
    {
        // Allocate an integer on the heap
        let _box3 = Box::new(4i32);

        // `_box3` is destroyed here, and memory gets freed
    }

    // Creating lots of boxes just for fun
    // There's no need to manually free memory!
    for _ in 0u32..1_000 {
        create_box();
    }

    // `_box2` is destroyed here, and memory gets freed
}
```

Of course, we can double check for memory errors using `valgrind`:

```
$ rustc raii.rs && valgrind ./raii
==26873== Memcheck, a memory error detector
==26873== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==26873== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==26873== Command: ./raii
==26873==
==26873==
==26873== HEAP SUMMARY:
==26873==     in use at exit: 0 bytes in 0 blocks
==26873==   total heap usage: 1,013 allocs, 1,013 frees, 8,696 bytes allocated
==26873==
==26873== All heap blocks were freed -- no leaks are possible
==26873==
==26873== For counts of detected and suppressed errors, rerun with: -v
==26873== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

No leaks here!

Destructor

The notion of a destructor in Rust is provided through the `Drop` trait. The destructor is called when the resource goes out of scope. This trait is not required to be implemented for every type, only implement it for your type if you require its own destructor logic.

Run the below example to see how the `Drop` trait works. When the variable in the `main` function goes out of scope the custom destructor will be invoked.

```
struct ToDrop;

impl Drop for ToDrop {
    fn drop(&mut self) {
        println!("ToDrop is being dropped");
    }
}

fn main() {
    let x = ToDrop;
    println!("Made a ToDrop!");
}
```

See also:

Box

Ownership and moves

Because variables are in charge of freeing their own resources, **resources can only have one owner**. This also prevents resources from being freed more than once. Note that not all variables own resources (e.g. references).

When doing assignments (`let x = y`) or passing function arguments by value (`foo(x)`), the *ownership* of the resources is transferred. In Rust-speak, this is known as a *move*.

After moving resources, the previous owner can no longer be used. This avoids creating dangling pointers.

```
// This function takes ownership of the heap allocated memory
fn destroy_box(c: Box<i32>) {
    println!("Destroying a box that contains {}", c);

    // `c` is destroyed and the memory freed
}

fn main() {
    // _Stack_ allocated integer
    let x = 5u32;

    // *Copy* `x` into `y` - no resources are moved
    let y = x;

    // Both values can be independently used
    println!("x is {}, and y is {}", x, y);

    // `a` is a pointer to a _heap_ allocated integer
    let a = Box::new(5i32);

    println!("a contains: {}", a);

    // *Move* `a` into `b`
    let b = a;
    // The pointer address of `a` is copied (not the data) into `b`.
    // Both are now pointers to the same heap allocated data, but
    // `b` now owns it.

    // Error! `a` can no longer access the data, because it no longer owns the
    // heap memory
    //println!("a contains: {}", a);
    // TODO ^ Try uncommenting this line

    // This function takes ownership of the heap allocated memory from `b`
    destroy_box(b);

    // Since the heap memory has been freed at this point, this action would
    // result in dereferencing freed memory, but it's forbidden by the compiler
    // Error! Same reason as the previous Error
    //println!("b contains: {}", b);
    // TODO ^ Try uncommenting this line
}
```


Mutability

Mutability of data can be changed when ownership is transferred.

```
fn main() {
    let immutable_box = Box::new(5u32);

    println!("immutable_box contains {}", immutable_box);

    // Mutability error
    //*immutable_box = 4;

    // *Move* the box, changing the ownership (and mutability)
    let mut mutable_box = immutable_box;

    println!("mutable_box contains {}", mutable_box);

    // Modify the contents of the box
    *mutable_box = 4;

    println!("mutable_box now contains {}", mutable_box);
}
```

Rust By Example

Search this book ...

Borrowing

Most of the time, we'd like to access data without taking ownership over it. To accomplish this, Rust uses a *borrowing* mechanism. Instead of passing objects by value (`T`), objects can be passed by reference (`&T`).

The compiler statically guarantees (via its borrow checker) that references *always* point to valid objects. That is, while references to an object exist, the object cannot be destroyed.

```
// This function takes ownership of a box and destroys it
fn eat_box_i32(boxed_i32: Box<i32>) {
    println!("Destroying box that contains {}", boxed_i32);
}

// This function borrows an i32
fn borrow_i32(borrowed_i32: &i32) {
    println!("This int is: {}", borrowed_i32);
}

fn main() {
    // Create a boxed i32, and a stacked i32
    let boxed_i32 = Box::new(5_i32);
    let stacked_i32 = 6_i32;

    // Borrow the contents of the box. Ownership is not taken,
    // so the contents can be borrowed again.
    borrow_i32(&boxed_i32);
    borrow_i32(&stacked_i32);

    {
        // Take a reference to the data contained inside the box
        let _ref_to_i32: &i32 = &boxed_i32;

        // Error!
        // Can't destroy `boxed_i32` while the inner value is borrowed.
        eat_box_i32(boxed_i32);
        // FIXME ^ Comment out this line

        // `_ref_to_i32` goes out of scope and is no longer borrowed.
    }

    // `boxed_i32` can now give up ownership to `eat_box` and be destroyed
    eat_box_i32(boxed_i32);
}
```

Mutability

Mutable data can be mutably borrowed using `&mut T`. This is called a *mutable reference* and gives read/write access to the borrower. In contrast, `&T` borrows the data via an immutable reference, and the borrower can read the data but not modify it:

```
#[allow(dead_code)]
#[derive(Clone, Copy)]
struct Book {
    // `&'static str` is a reference to a string allocated in read only memory
    author: &'static str,
    title: &'static str,
    year: u32,
}

// This function takes a reference to a book
fn borrow_book(book: &Book) {
    println!("I immutably borrowed {} - {} edition", book.title, book.year);
}

// This function takes a reference to a mutable book and changes `year` to 2014
fn new_edition(book: &mut Book) {
    book.year = 2014;
    println!("I mutably borrowed {} - {} edition", book.title, book.year);
}

fn main() {
    // Create an immutable Book named `immutabook`
    let immutabook = Book {
        // string literals have type `&'static str`
        author: "Douglas Hofstadter",
        title: "GÃ¶del, Escher, Bach",
        year: 1979,
    };

    // Create a mutable copy of `immutabook` and call it `mutabook`
    let mut mutabook = immutabook;

    // Immutably borrow an immutable object
    borrow_book(&immutabook);

    // Immutably borrow a mutable object
    borrow_book(&mutabook);

    // Borrow a mutable object as mutable
    new_edition(&mut mutabook);

    // Error! Cannot borrow an immutable object as mutable
    new_edition(&mut immutabook);
    // FIXME ^ Comment out this line
}
```

See also:

`static`

-
-
-
-
-

Freezing

When data is immutably borrowed, it also *freezes*. *Frozen* data can't be modified via the original object until all references to it go out of scope:

```
fn main() {
    let mut _mutable_integer = 7i32;

    {
        // Borrow `_mutable_integer`
        let _large_integer = &_mutable_integer;

        // Error! `_mutable_integer` is frozen in this scope
        _mutable_integer = 50;
        // FIXME ^ Comment out this line

        // `_large_integer` goes out of scope
    }

    // Ok! `_mutable_integer` is not frozen in this scope
    _mutable_integer = 3;
```

```
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Aliasing

Data can be immutably borrowed any number of times, but while immutably borrowed, the original data can't be mutably borrowed. On the other hand, only *one* mutable borrow is allowed at a time. The original data can be borrowed again only *after* the mutable reference goes out of scope.

```
struct Point { x: i32, y: i32, z: i32 }

fn main() {
    let mut point = Point { x: 0, y: 0, z: 0 };

    {
        let borrowed_point = &point;
        let another_borrow = &point;

        // Data can be accessed via the references and the original owner
        println!("Point has coordinates: ({}, {}, {})",
                 borrowed_point.x, another_borrow.y, point.z);

        // Error! Can't borrow point as mutable because it's currently
        // borrowed as immutable.
        //let mutable_borrow = &mut point;
        // TODO ^ Try uncommenting this line

        // Immutable references go out of scope
    }

    {
        let mutable_borrow = &mut point;

        // Change data via mutable reference
        mutable_borrow.x = 5;
        mutable_borrow.y = 2;
        mutable_borrow.z = 1;

        // Error! Can't borrow `point` as immutable because it's currently
        // borrowed as mutable.
        //let y = &point.y;
        // TODO ^ Try uncommenting this line

        // Error! Can't print because `println!` takes an immutable reference.
        //println!("Point Z coordinate is {}", point.z);
        // TODO ^ Try uncommenting this line

        // Ok! Mutable references can be passed as immutable to `println!`
        println!("Point has coordinates: ({}, {}, {})",
                 mutable_borrow.x, mutable_borrow.y, mutable_borrow.z);

        // Mutable reference goes out of scope
    }

    // Immutable references to point are allowed again
    let borrowed_point = &point;
    println!("Point now has coordinates: ({}, {}, {})",
             borrowed_point.x, borrowed_point.y, borrowed_point.z);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

The ref pattern

When doing pattern matching or destructuring via the `let` binding, the `ref` keyword can be used to take references to the fields of a struct/tuple. The example below shows a few instances where this can be useful:

```
#[derive(Clone, Copy)]
struct Point { x: i32, y: i32 }

fn main() {
    let c = 'Q';

    // A `ref` borrow on the left side of an assignment is equivalent to
    // an `&` borrow on the right side.
    let ref ref_c1 = c;
    let ref_c2 = &c;

    println!("ref_c1 equals ref_c2: {}", *ref_c1 == *ref_c2);

    let point = Point { x: 0, y: 0 };

    // `ref` is also valid when destructuring a struct.
    let _copy_of_x = {
        // `ref_to_x` is a reference to the `x` field of `point`.
        let Point { x: ref ref_to_x, y: _ } = point;

        // Return a copy of the `x` field of `point`.
        *ref_to_x
    };

    // A mutable copy of `point`
    let mut mutable_point = point;

    {
        // `ref` can be paired with `mut` to take mutable references.
        let Point { x: _, y: ref mut mut_ref_to_y } = mutable_point;

        // Mutate the `y` field of `mutable_point` via a mutable reference.
        *mut_ref_to_y = 1;
    }

    println!("point is ({}, {})", point.x, point.y);
    println!("mutable_point is ({}, {})", mutable_point.x, mutable_point.y);

    // A mutable tuple that includes a pointer
    let mut mutable_tuple = (Box::new(5u32), 3u32);

    {
        // Destructure `mutable_tuple` to change the value of `last`.
        let (_, ref mut last) = mutable_tuple;
        *last = 2u32;
    }

    println!("tuple is {:?}", mutable_tuple);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Lifetimes

A *lifetime* is a construct the compiler (or more specifically, its *borrow checker*) uses to ensure all borrows are valid. Specifically, a variable's lifetime begins when it is created and ends when it is destroyed. While lifetimes and scopes are often referred to together, they are not the same.

Take, for example, the case where we borrow a variable via `&`. The borrow has a lifetime that is determined by where it is declared. As a result, the borrow is valid as long as it ends before the lender is destroyed. However, the scope of the borrow is determined by where the reference is used.

In the following example and in the rest of this section, we will see how lifetimes relate to scopes, as well as how the two differ.

```
// Lifetimes are annotated below with lines denoting the creation
// and destruction of each variable.
// `i` has the longest lifetime because its scope entirely encloses
// both `borrow1` and `borrow2`. The duration of `borrow1` compared
// to `borrow2` is irrelevant since they are disjoint.
fn main() {
    let i = 3; // Lifetime for `i` starts. â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
    //                                                     â”‚
    { //                                                   â”‚
        let borrow1 = &i; // `borrow1` lifetime starts. â”€â”€â”â”‚
        //                                                â”‚â”‚
        println!("borrow1: {}", borrow1); //              â”‚â”‚
    } // `borrow1 ends. â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜â”‚
    //                                                     â”‚
    //                                                     â”‚
    { //                                                   â”‚
        let borrow2 = &i; // `borrow2` lifetime starts. â”€â”€â”â”‚
```

```
        //                                              â”‚â”‚
        println!("borrow2: {}", borrow2); //            â”‚â”‚
    } // `borrow2` ends. â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜â”‚
    //                                                   â”‚
}   // Lifetime ends. â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜
```

Note that no names or types are assigned to label lifetimes. This restricts how lifetimes will be able to be used as we will see.

-
-
-
-
-

Rust By Example

Search this book ...

Explicit annotation

The borrow checker uses explicit lifetime annotations to determine how long references should be valid. In cases where lifetimes are not elided[1], Rust requires explicit annotations to determine what the lifetime of a reference should be. The syntax for explicitly annotating a lifetime uses an apostrophe character as follows:

```
foo<'a>
// `foo` has a lifetime parameter `'a`
```

Similar to closures, using lifetimes requires generics. Additionally, this lifetime syntax indicates that the lifetime of `foo` may not exceed that of `'a`. Explicit annotation of a type has the form `&'a T` where `'a` has already been introduced.

In cases with multiple lifetimes, the syntax is similar:

```
foo<'a, 'b>
// `foo` has lifetime parameters `'a` and `'b`
```

In this case, the lifetime of `foo` cannot exceed that of either `'a` *or* `'b`.

See the following example for explicit lifetime annotation in use:

```
// `print_refs` takes two references to `i32` which have different
// lifetimes `'a` and `'b`. These two lifetimes must both be at
// least as long as the function `print_refs`.
fn print_refs<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("x is {} and y is {}", x, y);
}

// A function which takes no arguments, but has a lifetime parameter `'a`.
fn failed_borrow<'a>() {
    let _x = 12;

    // ERROR: `_x` does not live long enough
    //let y: &'a i32 = &_x;
    // Attempting to use the lifetime `'a` as an explicit type annotation
    // inside the function will fail because the lifetime of `&_x` is shorter
    // than that of `y`. A short lifetime cannot be coerced into a longer one.
}

fn main() {
    // Create variables to be borrowed below.
    let (four, nine) = (4, 9);

    // Borrows (`&`) of both variables are passed into the function.
    print_refs(&four, &nine);
    // Any input which is borrowed must outlive the borrower.
    // In other words, the lifetime of `four` and `nine` must
    // be longer than that of `print_refs`.

    failed_borrow();
    // `failed_borrow` contains no references to force `'a` to be
    // longer than the lifetime of the function, but `'a` is longer.
    // Because the lifetime is never constrained, it defaults to `'static`.
}
```

1

elision implicitly annotates lifetimes and so is different.

See also:

generics and closures

-
-

Rust By Example

Search this book ...

Functions

Ignoring elision, function signatures with lifetimes have a few constraints:

- any reference *must* have an annotated lifetime.
- any reference being returned *must* have the same lifetime as an input or be `static`.

Additionally, note that returning references without input is banned if it would result in returning references to invalid data. The following example shows off some valid forms of functions with lifetimes:

```
// One input reference with lifetime `'a` which must live
// at least as long as the function.
fn print_one<'a>(x: &'a i32) {
    println!("`print_one`: x is {}", x);
}

// Mutable references are possible with lifetimes as well.
fn add_one<'a>(x: &'a mut i32) {
    *x += 1;
}

// Multiple elements with different lifetimes. In this case, it
// would be fine for both to have the same lifetime `'a`, but
// in more complex cases, different lifetimes may be required.
fn print_multi<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("`print_multi`: x is {}, y is {}", x, y);
}

// Returning references that have been passed in is acceptable.
// However, the correct lifetime must be returned.
fn pass_x<'a, 'b>(x: &'a i32, _: &'b i32) -> &'a i32 { x }

//fn invalid_output<'a>() -> &'a String { &String::from("foo") }
// The above is invalid: `'a` must live longer than the function.
// Here, `&String::from("foo")` would create a `String`, followed by a
// reference. Then the data is dropped upon exiting the scope, leaving
// a reference to invalid data to be returned.

fn main() {
    let x = 7;
    let y = 9;

    print_one(&x);
    print_multi(&x, &y);

    let z = pass_x(&x, &y);
    print_one(z);

    let mut t = 3;
    add_one(&mut t);
    print_one(&t);
}
```

See also:

functions

Rust By Example

Search this book ...

Methods

Methods are annotated similarly to functions:

```
struct Owner(i32);

impl Owner {
    // Annotate lifetimes as in a standalone function.
    fn add_one<'a>(&'a mut self) { self.0 += 1; }
    fn print<'a>(&'a self) {
        println!("`print`: {}", self.0);
    }
}

fn main() {
    let mut owner  = Owner(18);

    owner.add_one();
    owner.print();
}
```

See also:

methods

-
-
-
-
-

Rust By Example

Search this book ...

Structs

Annotation of lifetimes in structures are also similar to functions:

```
// A type `Borrowed` which houses a reference to an
// `i32`. The reference to `i32` must outlive `Borrowed`.
#[derive(Debug)]
struct Borrowed<'a>(&'a i32);

// Similarly, both references here must outlive this structure.
#[derive(Debug)]
struct NamedBorrowed<'a> {
    x: &'a i32,
    y: &'a i32,
}

// An enum which is either an `i32` or a reference to one.
#[derive(Debug)]
enum Either<'a> {
    Num(i32),
    Ref(&'a i32),
}

fn main() {
    let x = 18;
    let y = 15;

    let single = Borrowed(&x);
    let double = NamedBorrowed { x: &x, y: &y };
    let reference = Either::Ref(&x);
    let number    = Either::Num(y);

    println!("x is borrowed in {:?}", single);
    println!("x and y are borrowed in {:?}", double);
    println!("x is borrowed in {:?}", reference);
    println!("y is *not* borrowed in {:?}", number);
}
```

See also:

`struct`s

-
-
-
-
-

Bounds

Just like generic types can be bounded, lifetimes (themselves generic) use bounds as well. The `:` character has a slightly different meaning here, but `+` is the same. Note how the following read:

1. `T: 'a`: *All* references in `T` must outlive lifetime `'a`.
2. `T: Trait + 'a`: Type `T` must implement trait `Trait` and *all* references in `T` must outlive `'a`.

The example below shows the above syntax in action:

```
use std::fmt::Debug; // Trait to bound with.

#[derive(Debug)]
struct Ref<'a, T: 'a>(&'a T);
// `Ref` contains a reference to a generic type `T` that has
// an unknown lifetime `'a`. `T` is bounded such that any
// *references* in `T` must outlive `'a`. Additionally, the lifetime
// of `Ref` may not exceed `'a`.

// A generic function which prints using the `Debug` trait.
fn print<T>(t: T) where
    T: Debug {
    println!("`print`: t is {:?}", t);
}

// Here a reference to `T` is taken where `T` implements
// `Debug` and all *references* in `T` outlive `'a`. In
// addition, `'a` must outlive the function.
fn print_ref<'a, T>(t: &'a T) where
    T: Debug + 'a {
    println!("`print_ref`: t is {:?}", t);
}

fn main() {
    let x = 7;
    let ref_x = Ref(&x);

    print_ref(&ref_x);
    print(ref_x);
}
```

See also:

generics, bounds in generics, and multiple bounds in generics

-
-
-
-
-

Rust By Example

Search this book ...

Coercion

A longer lifetime can be coerced into a shorter one so that it works inside a scope it normally wouldn't work in. This comes in the form of inferred coercion by the Rust compiler, and also in the form of declaring a lifetime difference:

```
// Here, Rust infers a lifetime that is as short as possible.
// The two references are then coerced to that lifetime.
fn multiply<'a>(first: &'a i32, second: &'a i32) -> i32 {
    first * second
}

// `<'a: 'b, 'b>` reads as lifetime `'a` is at least as long as `'b`.
// Here, we take in an `&'a i32` and return a `&'b i32` as a result of coercion.
fn choose_first<'a: 'b, 'b>(first: &'a i32, _: &'b i32) -> &'b i32 {
    first
}

fn main() {
    let first = 2; // Longer lifetime

    {
        let second = 3; // Shorter lifetime

        println!("The product is {}", multiply(&first, &second));
        println!("{} is the first", choose_first(&first, &second));
    };
}
```

Static

A `'static` lifetime is the longest possible lifetime, and lasts for the lifetime of the running program. A `'static` lifetime may also be coerced to a shorter lifetime. There are two ways to make a variable with `'static` lifetime, and both are stored in the read-only memory of the binary:

- Make a constant with the `static` declaration.
- Make a `string` literal which has type: `&'static str`.

See the following example for a display of each method:

```
// Make a constant with `'static` lifetime.
static NUM: i32 = 18;

// Returns a reference to `NUM` where its `'static`
// lifetime is coerced to that of the input argument.
fn coerce_static<'a>(_: &'a i32) -> &'a i32 {
    &NUM
}

fn main() {
    {
        // Make a `string` literal and print it:
        let static_string = "I'm in read-only memory";
        println!("static_string: {}", static_string);

        // When `static_string` goes out of scope, the reference
        // can no longer be used, but the data remains in the binary.
    }

    {
        // Make an integer to use for `coerce_static`:
        let lifetime_num = 9;

        // Coerce `NUM` to lifetime of `lifetime_num`:
        let coerced_static = coerce_static(&lifetime_num);

        println!("coerced_static: {}", coerced_static);
    }

    println!("NUM: {} stays accessible!", NUM);
}
```

See also:

[`'static` constants]

Elision

Some lifetime patterns are overwhelmingly common and so the borrow checker will implicitly add them to save typing and to improve readability. This process of implicit addition is called elision. Elision exists in Rust solely because these patterns are common.

The following code shows a few examples of elision. For a more comprehensive description of elision, see [lifetime elision] in the book.

```
// `elided_input` and `annotated_input` essentially have identical signatures
// because the lifetime of `elided_input` is elided by the compiler:
fn elided_input(x: &i32) {
    println!("`elided_input`: {}", x);
}

fn annotated_input<'a>(x: &'a i32) {
    println!("`annotated_input`: {}", x);
}

// Similarly, `elided_pass` and `annotated_pass` have identical signatures
// because the lifetime is added implicitly to `elided_pass`:
fn elided_pass(x: &i32) -> &i32 { x }

fn annotated_pass<'a>(x: &'a i32) -> &'a i32 { x }

fn main() {
    let x = 3;

    elided_input(&x);
    annotated_input(&x);

    println!("`elided_pass`: {}", elided_pass(&x));
    println!("`annotated_pass`: {}", annotated_pass(&x));
}
```

See also:

elision

-
-
-
-
-

Rust By Example

Search this book ...

Traits

A `trait` is a collection of methods defined for an unknown type: `Self`. They can access other methods declared in the same trait.

Traits can be implemented for any data type. In the example below, we define `Animal`, a group of methods. The `Animal` `trait` is then implemented for the `Sheep` data type, allowing the use of methods from `Animal` with a `Sheep`.

```
struct Sheep { naked: bool, name: &'static str }

trait Animal {
    // Static method signature; `Self` refers to the implementor type.
    fn new(name: &'static str) -> Self;

    // Instance method signatures; these will return a string.
    fn name(&self) -> &'static str;
    fn noise(&self) -> &'static str;

    // Traits can provide default method definitions.
    fn talk(&self) {
        println!("{} says {}", self.name(), self.noise());
    }
}

impl Sheep {
    fn is_naked(&self) -> bool {
        self.naked
    }

    fn shear(&mut self) {
        if self.is_naked() {
            // Implementor methods can use the implementor's trait methods.
            println!("{} is already naked...", self.name());
        } else {
            println!("{} gets a haircut!", self.name);

            self.naked = true;
        }
    }
}

// Implement the `Animal` trait for `Sheep`.
impl Animal for Sheep {
    // `Self` is the implementor type: `Sheep`.
    fn new(name: &'static str) -> Sheep {
        Sheep { name: name, naked: false }
    }

    fn name(&self) -> &'static str {
        self.name
    }

    fn noise(&self) -> &'static str {
        if self.is_naked() {
            "baaaaah?"
        } else {
```

```
            "baaaaah!"
        }
    }

    // Default trait methods can be overridden.
    fn talk(&self) {
        // For example, we can add some quiet contemplation.
        println!("{} pauses briefly... {}", self.name, self.noise());
    }
}

fn main() {
    // Type annotation is necessary in this case.
    let mut dolly: Sheep = Animal::new("Dolly");
    // TODO ^ Try removing the type annotations.

    dolly.talk();
    dolly.shear();
    dolly.talk();
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Derive

The compiler is capable of providing basic implementations for some traits via the `#[derive]` attribute. These traits can still be manually implemented if a more complex behavior is required.

The following is a list of derivable traits:

- Comparison traits: `Eq`, `PartialEq`, `Ord`, `PartialOrd`
- `Clone`, to create `T` from `&T` via a copy.
- `Copy`, to give a type 'copy semantics' instead of 'move semantics'
- `Hash`, to compute a hash from `&T`.
- `Default`, to create an empty instance of a data type.
- `Debug`, to format a value using the `{:?}` formatter.

```
// `Centimeters`, a tuple struct that can be compared
#[derive(PartialEq, PartialOrd)]
struct Centimeters(f64);

// `Inches`, a tuple struct that can be printed
#[derive(Debug)]
struct Inches(i32);

impl Inches {
    fn to_centimeters(&self) -> Centimeters {
        let &Inches(inches) = self;

        Centimeters(inches as f64 * 2.54)
    }
}

// `Seconds`, a tuple struct no additional attributes
struct Seconds(i32);

fn main() {
    let _one_second = Seconds(1);

    // Error: `Seconds` can't be printed; it doesn't implement the `Debug` trait
    //println!("One second looks like: {:?}", _one_second);
    // TODO ^ Try uncommenting this line

    // Error: `Seconds` can't be compared; it doesn't implement the `PartialEq` trait
    //let _this_is_true = (_one_second == _one_second);
    // TODO ^ Try uncommenting this line

    let foot = Inches(12);

    println!("One foot equals {:?}", foot);

    let meter = Centimeters(100.0);

    let cmp =
        if foot.to_centimeters() < meter {
            "smaller"
        } else {
            "bigger"
        };

    println!("One foot is {} than one meter.", cmp);
}
```

See also:

Operator Overloading

In Rust, many of the operators can be overloaded via traits. That is, some operators can be used to accomplish different tasks based on their input arguments. This is possible because operators are syntactic sugar for method calls. For example, the `+` operator in `a + b` calls the `add` method (as in `a.add(b)`). This `add` method is part of the `Add` trait. Hence, the `+` operator can be used by any implementor of the `Add` trait.

A list of the traits, such as `Add`, that overload operators can be found in `core::ops`.

```
use std::ops;

struct Foo;
struct Bar;

#[derive(Debug)]
struct FooBar;

#[derive(Debug)]
struct BarFoo;

// The `std::ops::Add` trait is used to specify the functionality of `+`.
// Here, we make `Add<Bar>` - the trait for addition with a RHS of type `Bar`.
// The following block implements the operation: Foo + Bar = FooBar
impl ops::Add<Bar> for Foo {
    type Output = FooBar;

    fn add(self, _rhs: Bar) -> FooBar {
        println!("> Foo.add(Bar) was called");

        FooBar
    }
}

// By reversing the types, we end up implementing non-commutative addition.
// Here, we make `Add<Foo>` - the trait for addition with a RHS of type `Foo`.
// This block implements the operation: Bar + Foo = BarFoo
impl ops::Add<Foo> for Bar {
    type Output = BarFoo;

    fn add(self, _rhs: Foo) -> BarFoo {
        println!("> Bar.add(Foo) was called");

        BarFoo
    }
}

fn main() {
    println!("Foo + Bar = {:?}", Foo + Bar);
    println!("Bar + Foo = {:?}", Bar + Foo);
}
```

See Also

Add, Syntax Index

Drop

The Drop trait only has one method: `drop`, which is called automatically when an object goes out of scope. The main use of the `Drop` trait is to free the resources that the implementor instance owns.

`Box`, `Vec`, `String`, `File`, and `Process` are some examples of types that implement the `Drop` trait to free resources. The `Drop` trait can also be manually implemented for any custom data type.

The following example adds a print to console to the `drop` function to announce when it is called.

```
struct Droppable {
    name: &'static str,
}

// This trivial implementation of `drop` adds a print to console.
impl Drop for Droppable {
    fn drop(&mut self) {
        println!("> Dropping {}", self.name);
    }
}

fn main() {
    let _a = Droppable { name: "a" };

    // block A
    {
        let _b = Droppable { name: "b" };

        // block B
        {
            let _c = Droppable { name: "c" };
            let _d = Droppable { name: "d" };

            println!("Exiting block B");
        }
        println!("Just exited block B");

        println!("Exiting block A");
    }
    println!("Just exited block A");

    // Variable can be manually dropped using the `drop` function
    drop(_a);
    // TODO ^ Try commenting this line

    println!("end of the main function");

    // `_a` *won't* be `drop`ed again here, because it already has been
    // (manually) `drop`ed
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Iterators

The `Iterator` trait is used to implement iterators over collections such as arrays.

The trait requires only a method to be defined for the `next` element, which may be manually defined in an `impl` block or automatically defined (as in arrays and ranges).

As a point of convenience for common situations, the `for` construct turns some collections into iterators using the `.into_iterator()` method.

```
struct Fibonacci {
    curr: u32,
    next: u32,
}

// Implement `Iterator` for `Fibonacci`.
// The `Iterator` trait only requires a method to be defined for the `next` element.
impl Iterator for Fibonacci {
    type Item = u32;

    // Here, we define the sequence using `.curr` and `.next`.
    // The return type is `Option<T>`:
    //     * When the `Iterator` is finished, `None` is returned.
    //     * Otherwise, the next value is wrapped in `Some` and returned.
    fn next(&mut self) -> Option<u32> {
        let new_next = self.curr + self.next;

        self.curr = self.next;
        self.next = new_next;

        // Since there's no endpoint to a Fibonacci sequence, the `Iterator`
```

```
        // will never return `None`, and `Some` is always returned.
        Some(self.curr)
    }
}

// Returns a Fibonacci sequence generator
fn fibonacci() -> Fibonacci {
    Fibonacci { curr: 1, next: 1 }
}

fn main() {
    // `0..3` is an `Iterator` that generates: 0, 1, and 2.
    let mut sequence = 0..3;

    println!("Four consecutive `next` calls on 0..3");
    println!("> {:?}", sequence.next());
    println!("> {:?}", sequence.next());
    println!("> {:?}", sequence.next());
    println!("> {:?}", sequence.next());

    // `for` works through an `Iterator` until it returns `None`.
    // Each `Some` value is unwrapped and bound to a variable (here, `i`).
    println!("Iterate through 0..3 using `for`");
    for i in 0..3 {
        println!("> {}", i);
    }

    // The `take(n)` method reduces an `Iterator` to its first `n` terms.
    println!("The first four terms of the Fibonacci sequence are: ");
    for i in fibonacci().take(4) {
        println!("> {}", i);
    }

    // The `skip(n)` method shortens an `Iterator` by dropping its first `n` terms.
    println!("The next four terms of the Fibonacci sequence are: ");
    for i in fibonacci().skip(4).take(4) {
        println!("> {}", i);
    }

    let array = [1u32, 3, 3, 7];

    // The `iter` method produces an `Iterator` over an array/slice.
    println!("Iterate the following array {:?}", &array);
    for i in array.iter() {
        println!("> {}", i);
    }
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Clone

When dealing with resources, the default behavior is to transfer them during assignments or function calls. However, sometimes we need to make a copy of the resource as well.

The `Clone` trait helps us do exactly this. Most commonly, we can use the `.clone()` method defined by the `Clone` trait.

```
// A unit struct without resources
#[derive(Debug, Clone, Copy)]
struct Nil;

// A tuple struct with resources that implements the `Clone` trait
#[derive(Clone, Debug)]
struct Pair(Box<i32>, Box<i32>);

fn main() {
    // Instantiate `Nil`
    let nil = Nil;
    // Copy `Nil`, there are no resources to move
    let copied_nil = nil;

    // Both `Nil`s can be used independently
    println!("original: {:?}", nil);
    println!("copy: {:?}", copied_nil);

    // Instantiate `Pair`
    let pair = Pair(Box::new(1), Box::new(2));
    println!("original: {:?}", pair);

    // Copy `pair` into `moved_pair`, moves resources
    let moved_pair = pair;
    println!("copy: {:?}", moved_pair);

    // Error! `pair` has lost its resources
    //println!("original: {:?}", pair);
    // TODO ^ Try uncommenting this line

    // Clone `moved_pair` into `cloned_pair` (resources are included)
    let cloned_pair = moved_pair.clone();
```

```
    // Drop the original pair using std::mem::drop
    drop(moved_pair);

    // Error! `moved_pair` has been dropped
    //println!("copy: {:?}", moved_pair);
    // TODO ^ Try uncommenting this line

    // The result from .clone() can still be used!
    println!("clone: {:?}", cloned_pair);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

macro_rules!

Rust provides a powerful macro system that allows metaprogramming. As you've seen in previous chapters, macros look like functions, except that their name ends with a bang `!`, but instead of generating a function call, macros are expanded into source code that gets compiled with the rest of the program. However, unlike macros in C and other languages, Rust macros are expanded into abstract syntax trees, rather than string preprocessing, so you don't get unexpected precedence bugs.

Macros are created using the `macro_rules!` macro.

```
// This is a simple macro named `say_hello`.
macro_rules! say_hello {
    // `()` indicates that the macro takes no argument.
    () => (
        // The macro will expand into the contents of this block.
        println!("Hello!");
    )
}

fn main() {
    // This call will expand into `println!("Hello");`
    say_hello!()
}
```

So why are macros useful?

1. Don't repeat yourself. There are many cases where you may need similar functionality in multiple places but with different types. Often, writing a macro is a useful way to avoid repeating code. (More on this later)

2. Domain-specific languages. Macros allow you to define special syntax for a specific purpose. (More on this later)

3. Variadic interfaces. Sometime you want to define an interface that takes a variable number of arguments. An example is `println!` which could take any number of arguments, depending on the format string!. (More on this later)

-
-
-
-
-

Rust By Example

Search this book ...

Syntax

In following subsections, we will show how to define macros in Rust. There are three basic ideas:

- Patterns and Designators
- Overloading

Rust By Example

Search this book ...

Designators

The arguments of a macro are prefixed by a dollar sign `$` and type annotated with a *designator*:

```
macro_rules! create_function {
    // This macro takes an argument of designator `ident` and
    // creates a function named `$func_name`.
    // The `ident` designator is used for variable/function names.
    ($func_name:ident) => (
        fn $func_name() {
            // The `stringify!` macro converts an `ident` into a string.
            println!("You called {:?}()",
                     stringify!($func_name));
        }
    )
}

// Create functions named `foo` and `bar` with the above macro.
create_function!(foo);
create_function!(bar);

macro_rules! print_result {
    // This macro takes an expression of type `expr` and prints
    // it as a string along with its result.
    // The `expr` designator is used for expressions.
    ($expression:expr) => (
        // `stringify!` will convert the expression *as it is* into a string.
        println!("{:?} = {:?}",
                 stringify!($expression),
                 $expression);
    )
}

fn main() {
    foo();
    bar();

    print_result!(1u32 + 1);

    // Recall that blocks are expressions too!
    print_result!({
        let x = 1u32;

        x * x + 2 * x - 1
    });
}
```

This is a list of all the designators:

- `block`
- `expr` is used for expressions
- `ident` is used for variable/function names
- `item`
- `pat` (*pattern*)
- `path`
- `stmt` (*statement*)
- `tt` (*token tree*)
- `ty` (*type*)

Rust By Example

Search this book ...

Overload

Macros can be overloaded to accept different combinations of arguments. In that regard, `macro_rules!` can work similarly to a match block:

```
// `test!` will compare `$left` and `$right`
// in different ways depending on how you invoke it:
macro_rules! test {
    // Arguments don't need to be separated by a comma.
    // Any template can be used!
    ($left:expr; and $right:expr) => (
        println!("{:?} and {:?} is {:?}",
                 stringify!($left),
                 stringify!($right),
                 $left && $right)
    );
    // ^ each arm must end with a semicolon.
    ($left:expr; or $right:expr) => (
        println!("{:?} or {:?} is {:?}",
                 stringify!($left),
                 stringify!($right),
                 $left || $right)
    );
}

fn main() {
    test!(1i32 + 1 == 2i32; and 2i32 * 2 == 4i32);
    test!(true; or false);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Repeat

Macros can use `+` in the argument list to indicate that an argument may repeat at least once, or `*`, to indicate that the argument may repeat zero or more times.

In the following example, surrounding the matcher with `$(...),+` will match one or more expression, separated by commas. Also note that the semicolon is optional on the last case.

```
// `min!` will calculate the minimum of any number of arguments.
macro_rules! find_min {
    // Base case:
    ($x:expr) => ($x);
    // `$x` followed by at least one `$y,`
    ($x:expr, $($y:expr),+) => (
        // Call `find_min!` on the tail `$y`
        std::cmp::min($x, find_min!($($y),+))
    )
}

fn main() {
    println!("{}", find_min!(1u32));
    println!("{}", find_min!(1u32 + 2 , 2u32));
    println!("{}", find_min!(5u32, 2u32 * 3, 4u32));
}
```

-
-
-
-
-

Rust By Example

Search this book ...

DRY (Don't Repeat Yourself)

Macros allow writing DRY code by factoring out the common parts of functions and/or test suites. Here is an example that implements and tests the `+=`, `*=` and `-=` operators on `Vec<T>`:

```
use std::ops::{Add, Mul, Sub};

macro_rules! assert_equal_len {
    // The `tt` (token tree) designator is used for
    // operators and tokens.
    ($a:ident, $b: ident, $func:ident, $op:tt) => (
        assert!($a.len() == $b.len(),
                "{:?}: dimension mismatch: {:?} {:?} {:?}",
                stringify!($func),
                ($a.len(),),
                stringify!($op),
                ($b.len(),));
    )
}

macro_rules! op {
    ($func:ident, $bound:ident, $op:tt, $method:ident) => (
        fn $func<T: $bound<T, Output=T> + Copy>(xs: &mut Vec<T>, ys: &Vec<T>) {
            assert_equal_len!(xs, ys, $func, $op);

            for (x, y) in xs.iter_mut().zip(ys.iter()) {
                *x = $bound::$method(*x, *y);
                // *x = x.$method(*y);
            }
        }
    )
}

// Implement `add_assign`, `mul_assign`, and `sub_assign` functions.
op!(add_assign, Add, +=, add);
op!(mul_assign, Mul, *=, mul);
op!(sub_assign, Sub, -=, sub);

mod test {
    use std::iter;
    macro_rules! test {
        ($func: ident, $x:expr, $y:expr, $z:expr) => {
            #[test]
            fn $func() {
                for size in 0usize..10 {
                    let mut x: Vec<_> = iter::repeat($x).take(size).collect();
                    let y: Vec<_> = iter::repeat($y).take(size).collect();
                    let z: Vec<_> = iter::repeat($z).take(size).collect();

                    super::$func(&mut x, &y);

                    assert_eq!(x, z);
                }
            }
        }
    }

    // Test `add_assign`, `mul_assign` and `sub_assign`
    test!(add_assign, 1u32, 2u32, 3u32);
    test!(mul_assign, 2u32, 3u32, 6u32);
    test!(sub_assign, 3u32, 2u32, 1u32);
}
```

```
$ rustc --test dry.rs && ./dry
running 3 tests
test test::mul_assign ... ok
test test::add_assign ... ok
test test::sub_assign ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```

-
-
-
-
-

Rust By Example

Search this book ...

Domain Specific Languages (DSLs)

A DSL is a mini "language" embedded in a Rust macro. It is completely valid Rust because the macro system expands into normal Rust constructs, but it looks like a small language. This allows you to define concise or intuitive syntax for some special functionality (within bounds).

Suppose that I want to define a little calculator API. I would like to supply an expression an have the output printed to console.

```
macro_rules! calculate {
    (eval $e:expr) => {{
```

```
        {
            let val: usize = $e; // Force types to be integers
            println!("{} = {}", stringify!{$e}, val);
        }
    }};
}

fn main() {
    calculate! {
        eval 1 + 2 // hehehe `eval` is _not_ a Rust keyword!
    }

    calculate! {
        eval (1 + 2) * (3 / 4)
    }
}
```

Output:

```
1 + 2 = 3
(1 + 2) * (3 / 4) = 0
```

This was a very simple example, but much more complex interfaces have been developed, such as `lazy_static` or `clap`.

-
-
-
-
-

Rust By Example

Search this book ...

Variadic Interfaces

A *variadic* interface takes an arbitrary number of arguments. For example, `println!` can take an arbitrary number of arguments, as determined by the format string.

We can extend our `calculate!` macro from the previous section to be variadic:

```
macro_rules! calculate {
    // The pattern for a single `eval`
    (eval $e:expr) => {{
        {
            let val: usize = $e; // Force types to be integers
            println!("{} = {}", stringify!{$e}, val);
        }
    }};

    // Decompose multiple `eval`s recursively
    (eval $e:expr, $(eval $es:expr),+) => {{
        calculate! { eval $e }
        calculate! { $(eval $es),+ }
    }};
}

fn main() {
    calculate! { // Look ma! Variadic `calculate!`!
        eval 1 + 2,
        eval 3 + 4,
        eval (2 * 3) + 1
    }
}
```

Output:

```
1 + 2 = 3
3 + 4 = 7
(2 * 3) + 1 = 7
```

-
-
-
-
-

Rust By Example

Search this book ...

Error handling

Error handling is the process of handling the possibility of failure. For example, failing to read a file and then continuing to use that *bad* input would clearly be problematic. Noticing and explicitly managing those errors saves the rest of the program from various pitfalls.

There are various ways to deal with errors in Rust, which are described in the following subchapters. They all have more or less subtle differences and different use cases. As a rule of thumb:

An explicit `panic` is mainly useful for tests and dealing with unrecoverable errors. For prototyping it can be useful, for example when dealing with functions that haven't been implemented yet, but in those cases the more descriptive `unimplemented` is better. In tests `panic` is a reasonable way to explicitly fail.

The `Option` type is for when a value is optional or when the lack of a value is not an error condition. For example the parent of a directory - `/` and `C:` don't have one. When dealing with `Option`s, `unwrap` is fine for prototyping and cases where it's absolutely certain that there is guaranteed to be a value. However `expect` is more useful since it lets you specify an error message in case something goes wrong anyway.

When there is a chance that things do go wrong and the caller has to deal with the problem, use `Result`. You can `unwrap` and `expect` them as well (please don't do that unless it's a test or quick prototype).

For a more rigorous discussion of error handling, refer to the error handling section in the official book.

-
-
-
-
-

Rust By Example

Search this book ...

`panic`

The simplest error handling mechanism we will see is `panic`. It prints an error message, starts unwinding the task, and usually exits the program. Here, we explicitly call `panic` on our error condition:

```
fn give_princess(gift: &str) {
    // Princesses hate snakes, so we need to stop if she disapproves!
    if gift == "snake" { panic!("AAAaaaaa!!!!"); }

    println!("I love {}s!!!!!", gift);
}

fn main() {
    give_princess("teddy bear");
    give_princess("snake");
}
```

-
-
-
-
-

Rust By Example

Search this book ...

`Option` & `unwrap`

In the last example, we showed that we can induce program failure at will. We told our program to `panic` if the princess received an inappropriate gift - a snake. But what if the princess expected a gift and didn't receive one? That case would be just as bad, so it needs to be handled!

We *could* test this against the null string (`""`) as we do with a snake. Since we're using Rust, let's instead have the compiler point out cases where there's no gift.

An `enum` called `Option<T>` in the `std` library is used when absence is a possibility. It manifests itself as one of two "options":

- `Some(T)`: An element of type `T` was found
- `None`: No element was found

These cases can either be explicitly handled via `match` or implicitly with `unwrap`. Implicit handling will either return the inner element or `panic`.

Note that it's possible to manually customize `panic` with [expect](), but `unwrap` otherwise leaves us with a less meaningful output than explicit handling. In the following example, explicit handling yields a more controlled result while retaining the option to `panic` if desired.

```
// The commoner has seen it all, and can handle any gift well.
// All gifts are handled explicitly using `match`.
fn give_commoner(gift: Option<&str>) {
    // Specify a course of action for each case.
    match gift {
        Some("snake") => println!("Yuck! I'm throwing that snake in a fire."),
        Some(inner)   => println!("{}? How nice.", inner),
        None          => println!("No gift? Oh well."),
    }
}

// Our sheltered princess will `panic` at the sight of snakes.
// All gifts are handled implicitly using `unwrap`.
fn give_princess(gift: Option<&str>) {
    // `unwrap` returns a `panic` when it receives a `None`.
    let inside = gift.unwrap();
    if inside == "snake" { panic!("AAAaaaaa!!!!"); }

    println!("I love {}s!!!!!", inside);
}

fn main() {
    let food  = Some("cabbage");
    let snake = Some("snake");
    let void  = None;

    give_commoner(food);
    give_commoner(snake);
    give_commoner(void);

    let bird = Some("robin");
    let nothing = None;

    give_princess(bird);
    give_princess(nothing);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Combinators: `map`

`match` is a valid method for handling `Option`s. However, you may eventually find heavy usage tedious, especially with operations only valid with an input. In these cases, [combinators]() can be used to manage control flow in a modular fashion.

`Option` has a built in method called `map()`, a combinator for the simple mapping of `Some -> Some` and `None -> None`. Multiple `map()` calls can be chained together for even more flexibility.

In the following example, `process()` replaces all functions previous to it while staying compact.

```
#![allow(dead_code)]

#[derive(Debug)] enum Food { Apple, Carrot, Potato }

#[derive(Debug)] struct Peeled(Food);
#[derive(Debug)] struct Chopped(Food);
#[derive(Debug)] struct Cooked(Food);

// Peeling food. If there isn't any, then return `None`.
// Otherwise, return the peeled food.
fn peel(food: Option<Food>) -> Option<Peeled> {
    match food {
        Some(food) => Some(Peeled(food)),
        None       => None,
    }
}

// Chopping food. If there isn't any, then return `None`.
// Otherwise, return the chopped food.
fn chop(peeled: Option<Peeled>) -> Option<Chopped> {
    match peeled {
        Some(Peeled(food)) => Some(Chopped(food)),
        None               => None,
    }
}
```

```
// Cooking food. Here, we showcase `map()` instead of `match` for case handling.
fn cook(chopped: Option<Chopped>) -> Option<Cooked> {
    chopped.map(|Chopped(food)| Cooked(food))
}

// A function to peel, chop, and cook food all in sequence.
// We chain multiple uses of `map()` to simplify the code.
fn process(food: Option<Food>) -> Option<Cooked> {
    food.map(|f| Peeled(f))
        .map(|Peeled(f)| Chopped(f))
        .map(|Chopped(f)| Cooked(f))
}

// Check whether there's food or not before trying to eat it!
fn eat(food: Option<Cooked>) {
    match food {
        Some(food) => println!("Mmm. I love {:?}", food),
        None       => println!("Oh no! It wasn't edible."),
    }
}

fn main() {
    let apple = Some(Food::Apple);
    let carrot = Some(Food::Carrot);
    let potato = None;

    let cooked_apple = cook(chop(peel(apple)));
    let cooked_carrot = cook(chop(peel(carrot)));
    // Let's try the simpler looking `process()` now.
    let cooked_potato = process(potato);

    eat(cooked_apple);
    eat(cooked_carrot);
    eat(cooked_potato);
}
```

See also:

closures, `Option`, `Option::map()`

-
-
-
-
-

Rust By Example

Search this book ...

Combinators: `and_then`

`map()` was described as a chainable way to simplify `match` statements. However, using `map()` on a function that returns an `Option<T>` results in the nested `Option<Option<T>>`. Chaining multiple calls together can then become confusing. That's where another combinator called `and_then()`, known in some languages as flatmap, comes in.

`and_then()` calls its function input with the wrapped value and returns the result. If the `Option` is `None`, then it returns `None` instead.

In the following example, `cookable_v2()` results in an `Option<Food>`. Using `map()` instead of `and_then()` would have given an `Option<Option<Food>>`, which is an invalid type for `eat()`.

```
#![allow(dead_code)]

#[derive(Debug)] enum Food { CordonBleu, Steak, Sushi }
#[derive(Debug)] enum Day { Monday, Tuesday, Wednesday }

// We don't have the ingredients to make Sushi.
fn have_ingredients(food: Food) -> Option<Food> {
    match food {
        Food::Sushi => None,
        _           => Some(food),
    }
}

// We have the recipe for everything except Cordon Bleu.
fn have_recipe(food: Food) -> Option<Food> {
    match food {
        Food::CordonBleu => None,
        _                => Some(food),
    }
}

// To make a dish, we need both the ingredients and the recipe.
// We can represent the logic with a chain of `match`es:
fn cookable_v1(food: Food) -> Option<Food> {
    match have_ingredients(food) {
        None       => None,
        Some(food) => match have_recipe(food) {
            None       => None,
            Some(food) => Some(food),
        },
    }
}
```

```
// This can conveniently be rewritten more compactly with `and_then()`:
fn cookable_v2(food: Food) -> Option<Food> {
    have_ingredients(food).and_then(have_recipe)
}

fn eat(food: Food, day: Day) {
    match cookable_v2(food) {
        Some(food) => println!("Yay! On {:?} we get to eat {:?}.", day, food),
        None       => println!("Oh no. We don't get to eat on {:?}?", day),
    }
}

fn main() {
    let (cordon_bleu, steak, sushi) = (Food::CordonBleu, Food::Steak, Food::Sushi);

    eat(cordon_bleu, Day::Monday);
    eat(steak, Day::Tuesday);
    eat(sushi, Day::Wednesday);
}
```

See also:

closures, `Option`, and `Option::and_then()`

-
-
-
-
-

Rust By Example

Search this book ...

`Result`

`Result` is a richer version of the `Option` type that describes possible *error* instead of possible *absence*.

That is, `Result<T, E>` could have one of two outcomes:

- `Ok<T>`: An element `T` was found
- `Err<E>`: An error was found with element `E`

By convention, the expected outcome is `Ok` while the unexpected outcome is `Err`.

Like `Option`, `Result` has many methods associated with it. `unwrap()`, for example, either yields the element `T` or `panic`s. For case handling, there are many combinators between `Result` and `Option` that overlap.

In working with Rust, you will likely encounter methods that return the `Result` type, such as the `parse()` method. It might not always be possible to parse a string into the other type, so `parse()` returns a `Result` indicating possible failure.

Let's see what happens when we successfully and unsuccessfully `parse()` a string:

```
fn multiply(first_number_str: &str, second_number_str: &str) -> i32 {
    // Let's try using `unwrap()` to get the number out. Will it bite us?
    let first_number = first_number_str.parse::<i32>().unwrap();
    let second_number = second_number_str.parse::<i32>().unwrap();
    first_number * second_number
}

fn main() {
    let twenty = multiply("10", "2");
    println!("double is {}", twenty);

    let tt = multiply("t", "2");
    println!("double is {}", tt);
}
```

In the unsuccessful case, `parse()` leaves us with an error for `unwrap()` to `panic` on. Additionally, the `panic` exits our program and provides an unpleasant error message.

To improve the quality of our error message, we should be more specific about the return type and consider explicitly handling the error.

-
-
-
-
-

Rust By Example

Search this book ...

map for Result

Panicking in the previous example's `multiply` does not make for robust code. Generally, we want to return the error to the caller so it can decide what is the right way to respond to errors.

We first need to know what kind of error type we are dealing with. To determine the `Err` type, we look to `parse()`, which is implemented with the `FromStr` trait for `i32`. As a result, the `Err` type is specified as `ParseIntError`.

In the example below, the straightforward `match` statement leads to code that is overall more cumbersome.

```
use std::num::ParseIntError;

// With the return type rewritten, we use pattern matching without `unwrap()`.
fn multiply(first_number_str: &str, second_number_str: &str) -> Result<i32, ParseIntError> {
    match first_number_str.parse::<i32>() {
        Ok(first_number)  => {
            match second_number_str.parse::<i32>() {
                Ok(second_number)  => {
                    Ok(first_number * second_number)
                },
                Err(e) => Err(e),
            }
        },
        Err(e) => Err(e),
    }
}

fn print(result: Result<i32, ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    // This still presents a reasonable answer.
    let twenty = multiply("10", "2");
    print(twenty);

    // The following now provides a much more helpful error message.
    let tt = multiply("t", "2");
    print(tt);
}
```

Luckily, `Option`'s `map`, `and_then`, and many other combinators are also implemented for `Result`. `Result` contains a complete listing.

```
use std::num::ParseIntError;

// As with `Option`, we can use combinators such as `map()`.
// This function is otherwise identical to the one above and reads:
// Modify n if the value is valid, otherwise pass on the error.
fn multiply(first_number_str: &str, second_number_str: &str) -> Result<i32, ParseIntError> {
    first_number_str.parse::<i32>().and_then(|first_number| {
        second_number_str.parse::<i32>().map(|second_number| first_number * second_number)
    })
}

fn print(result: Result<i32, ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    // This still presents a reasonable answer.
    let twenty = multiply("10", "2");
    print(twenty);

    // The following now provides a much more helpful error message.
    let tt = multiply("t", "2");
    print(tt);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

aliases for `Result`

How about when we want to reuse a specific `Result` type many times? Recall that Rust allows us to create aliases. Conveniently, we can define one for the specific `Result` in question.

At a module level, creating aliases can be particularly helpful. Errors found in a specific module often have the same `Err` type, so a single alias can succinctly define *all* associated `Results`. This is so useful that the `std` library even supplies one: `io::Result`!

Here's a quick example to show off the syntax:

```
use std::num::ParseIntError;

// Define a generic alias for a `Result` with the error type `ParseIntError`.
type AliasedResult<T> = Result<T, ParseIntError>;

// Use the above alias to refer to our specific `Result` type.
fn multiply(first_number_str: &str, second_number_str: &str) -> AliasedResult<i32> {
    first_number_str.parse::<i32>().and_then(|first_number| {
        second_number_str.parse::<i32>().map(|second_number| first_number * second_number)
    })
}

// Here, the alias again allows us to save some space.
fn print(result: AliasedResult<i32>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    print(multiply("10", "2"));
    print(multiply("t", "2"));
}
```

See also:

`io::Result`

-
-
-
-
-

Rust By Example

Search this book ...

Early returns

In the previous example, we explicitly handled the errors using combinators. Another way to deal with this case analysis is to use a combination of `match` statements and *early returns*.

That is, we can simply stop executing the function and return the error if one occurs. For some, this form of code can be easier to both read and write. Consider this version of the previous example, rewritten using early returns:

```
use std::num::ParseIntError;

fn multiply(first_number_str: &str, second_number_str: &str) -> Result<i32, ParseIntError> {
    let first_number = match first_number_str.parse::<i32>() {
        Ok(first_number)  => first_number,
        Err(e) => return Err(e),
    };

    let second_number = match second_number_str.parse::<i32>() {
        Ok(second_number)  => second_number,
        Err(e) => return Err(e),
    };

    Ok(first_number * second_number)
}

fn print(result: Result<i32, ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    print(multiply("10", "2"));
    print(multiply("t", "2"));
}
```

At this point, we've learned to explicitly handle errors using combinators and early returns. While we generally want to avoid panicking, explicitly handling all of our errors is cumbersome.

In the next section, we'll introduce `?` for the cases where we simply need to `unwrap` without possibly inducing `panic`.

-
-
-
-
-

Rust By Example

Search this book ...

Introducing `?`

Sometimes we just want the simplicity of `unwrap` without the possibility of a `panic`. Until now, `unwrap` has forced us to nest deeper and deeper when what we really wanted was to get the variable *out*. This is exactly the purpose of `?`.

Upon finding an `Err`, there are two valid actions to take:

1. `panic!` which we already decided to try to avoid if possible
2. `return` because an `Err` means it cannot be handled

`?` is *almost*[1] exactly equivalent to an `unwrap` which `return`s instead of `panic`s on `Err`s. Let's see how we can simplify the earlier example that used combinators:

```
use std::num::ParseIntError;

fn multiply(first_number_str: &str, second_number_str: &str) -> Result<i32, ParseIntError> {
    let first_number = first_number_str.parse::<i32>()?;
    let second_number = second_number_str.parse::<i32>()?;

    Ok(first_number * second_number)
}

fn print(result: Result<i32, ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    print(multiply("10", "2"));
    print(multiply("t", "2"));
}
```

The `try!` macro

Before there was `?`, the same functionality was achieved with the `try!` macro. The `?` operator is now recommended, but you may still find `try!` when looking at older code. The same `multiply` function from the previous example would look like this using `try!`:

```
use std::num::ParseIntError;

fn multiply(first_number_str: &str, second_number_str: &str) -> Result<i32, ParseIntError> {
    let first_number = try!(first_number_str.parse::<i32>());
    let second_number = try!(second_number_str.parse::<i32>());

    Ok(first_number * second_number)
}

fn print(result: Result<i32, ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    print(multiply("10", "2"));
    print(multiply("t", "2"));
}
```

1

See re-enter ? for more details.

-
-
-
-

-

Rust By Example

Search this book ...

Multiple error types

The previous examples have always been very convenient; `Result`s interact with other `Result`s and `Option`s interact with other `Option`s.

Sometimes an `Option` needs to interact with a `Result`, or a `Result<T, Error1>` needs to interact with a `Result<T, Error2>`. In those cases, we want to manage our different error types in a way that makes them composable and easy to interact with.

In the following code, two instances of `unwrap` generate different error types. `Vec::first` returns an `Option`, while `parse::<i32>` returns a `Result<i32, ParseIntError>`:

```
fn double_first(vec: Vec<&str>) -> i32 {
    let first = vec.first().unwrap(); // Generate error 1
    2 * first.parse::<i32>().unwrap() // Generate error 2
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    println!("The first doubled is {}", double_first(numbers));

    println!("The first doubled is {}", double_first(empty));
    // Error 1: the input vector is empty

    println!("The first doubled is {}", double_first(strings));
    // Error 2: the element doesn't parse to a number
}
```

Over the next sections, we'll see several strategies for handling these kind of problems.

-
-
-
-
-

Rust By Example

Search this book ...

Pulling `Results` out of `Options`

The most basic way of handling mixed error types is to just embed them in each other.

```
use std::num::ParseIntError;

fn double_first(vec: Vec<&str>) -> Option<Result<i32, ParseIntError>> {
    vec.first().map(|first| {
        first.parse::<i32>().map(|n| 2 * n)
    })
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    println!("The first doubled is {:?}", double_first(numbers));

    println!("The first doubled is {:?}", double_first(empty));
    // Error 1: the input vector is empty

    println!("The first doubled is {:?}", double_first(strings));
    // Error 2: the element doesn't parse to a number
}
```

There are times when we'll want to stop processing on errors (like with [`?`]) but keep going when the `Option` is `None`. A couple of combinators come in handy to swap the `Result` and `Option`.

```
use std::num::ParseIntError;

fn double_first(vec: Vec<&str>) -> Result<Option<i32>, ParseIntError> {
    let opt = vec.first().map(|first| {
        first.parse::<i32>().map(|n| 2 * n)
    });
```

```
    let opt = opt.map_or(Ok(None), |r| r.map(Some))?;

    Ok(opt)
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    println!("The first doubled is {:?}", double_first(numbers));
    println!("The first doubled is {:?}", double_first(empty));
    println!("The first doubled is {:?}", double_first(strings));
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Defining an error type

Sometimes it simplifies the code to mask all of the different errors with a single type of error. We'll show this with a custom error.

Rust allows us to define our own error types. In general, a "good" error type:

- Represents different errors with the same type
- Presents nice error messages to the user
- Is easy to compare with other types
  - Good: `Err(EmptyVec)`
  - Bad: `Err("Please use a vector with at least one element".to_owned())`
- Can hold information about the error
  - Good: `Err(BadChar(c, position))`
  - Bad: `Err("+ cannot be used here".to_owned())`
- Composes well with other errors

```
use std::error;
use std::fmt;
use std::num::ParseIntError;

type Result<T> = std::result::Result<T, DoubleError>;

#[derive(Debug, Clone)]
// Define our error types. These may be customized for our error handling cases.
// Now we will be able to write our own errors, defer to an underlying error
// implementation, or do something in between.
struct DoubleError;

// Generation of an error is completely separate from how it is displayed.
// There's no need to be concerned about cluttering complex logic with the display style.
//
// Note that we don't store any extra info about the errors. This means we can't state
// which string failed to parse without modifying our types to carry that information.
impl fmt::Display for DoubleError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "invalid first item to double")
    }
}

// This is important for other errors to wrap this one.
impl error::Error for DoubleError {
    fn description(&self) -> &str {
        "invalid first item to double"
    }

    fn cause(&self) -> Option<&error::Error> {
        // Generic error, underlying cause isn't tracked.
        None
    }
}

fn double_first(vec: Vec<&str>) -> Result<i32> {
    vec.first()
       // Change the error to our new type.
       .ok_or(DoubleError)
       .and_then(|s| s.parse::<i32>()
            // Update to the new error type here also.
            .map_err(|_| DoubleError)
            .map(|i| 2 * i))
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}
```

```
fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

-
-
-
-
-

Rust By Example

Search this book ...

`Box`ing errors

A way to write simple code while preserving the original errors is to `Box` them. The drawback is that the underlying error type is only known at runtime and not statically determined.

The stdlib helps in boxing our errors by having `Box` implement conversion from any type that implements the `Error` trait into the trait object `Box<Error>`, via `From`.

```
use std::error;
use std::fmt;
use std::num::ParseIntError;

// Change the alias to `Box<error::Error>`.
type Result<T> = std::result::Result<T, Box<error::Error>>;

#[derive(Debug, Clone)]
struct EmptyVec;

impl fmt::Display for EmptyVec {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "invalid first item to double")
    }
}

impl error::Error for EmptyVec {
    fn description(&self) -> &str {
        "invalid first item to double"
    }

    fn cause(&self) -> Option<&error::Error> {
        // Generic error, underlying cause isn't tracked.
        None
    }
}

fn double_first(vec: Vec<&str>) -> Result<i32> {
    vec.first()
       .ok_or_else(|| EmptyVec.into())  // Converts to Box
       .and_then(|s| s.parse::<i32>()
            .map_err(|e| e.into())  // Converts to Box
            .map(|i| 2 * i))
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

See also:

Dynamic dispatch and `Error` trait

-
-
-

Rust By Example

Search this book ...

Other uses of ?

Notice in the previous example that our immediate reaction to calling `parse` is to `map` the error from a library error into a boxed error:

```
.and_then(|s| s.parse::<i32>()
    .map_err(|e| e.into())
```

Since this is a simple and common operation, it would be convenient if it could be elided. Alas, because `and_then` is not sufficiently flexible, it cannot. However, we can instead use `?`.

`?` was previously explained as either `unwrap` or `return Err(err)`. This is only mostly true. It actually means `unwrap` or `return Err(From::from(err))`. Since `From::from` is a conversion utility between different types, this means that if you `?` where the error is convertible to the return type, it will convert automatically.

Here, we rewrite the previous example using `?`. As a result, the `map_err` will go away when `From::from` is implemented for our error type:

```
use std::error;
use std::fmt;
use std::num::ParseIntError;

// Change the alias to `Box<error::Error>`.
type Result<T> = std::result::Result<T, Box<error::Error>>;

#[derive(Debug)]
struct EmptyVec;

impl fmt::Display for EmptyVec {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "invalid first item to double")
    }
}

impl error::Error for EmptyVec {
    fn description(&self) -> &str {
        "invalid first item to double"
    }

    fn cause(&self) -> Option<&error::Error> {
        // Generic error, underlying cause isn't tracked.
        None
    }
}

// The same structure as before but rather than chain all `Results`
// and `Options` along, we `?` to get the inner value out immediately.
fn double_first(vec: Vec<&str>) -> Result<i32> {
    let first = vec.first().ok_or(EmptyVec)?;
    let parsed = first.parse::<i32>()?;
    Ok(2 * parsed)
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

This is actually fairly clean now. Compared with the original `panic`, it is very similar to replacing the `unwrap` calls with `?` except that the return types are `Result`. As a result, they must be destructured at the top level.

See also:

`From::from` and `?`

Rust By Example

Search this book ...

Wrapping errors

An alternative to boxing errors is to wrap them in your own error type.

```
use std::error;
use std::num::ParseIntError;
use std::fmt;

type Result<T> = std::result::Result<T, DoubleError>;

#[derive(Debug)]
enum DoubleError {
    EmptyVec,
    // We will defer to the parse error implementation for their error.
    // Supplying extra info requires adding more data to the type.
    Parse(ParseIntError),
}

impl fmt::Display for DoubleError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            DoubleError::EmptyVec =>
                write!(f, "please use a vector with at least one element"),
            // This is a wrapper, so defer to the underlying types' implementation of `fmt`.
            DoubleError::Parse(ref e) => e.fmt(f),
        }
    }
}

impl error::Error for DoubleError {
    fn description(&self) -> &str {
        match *self {
            DoubleError::EmptyVec => "empty vectors not allowed",
            // This already impls `Error`, so defer to its own implementation.
            DoubleError::Parse(ref e) => e.description(),
        }
    }

    fn cause(&self) -> Option<&error::Error> {
        match *self {
            DoubleError::EmptyVec => None,
            // The cause is the underlying implementation error type. Is implicitly
            // cast to the trait object `&error::Error`. This works because the
            // underlying type already implements the `Error` trait.
            DoubleError::Parse(ref e) => Some(e),
        }
    }
}

// Implement the conversion from `ParseIntError` to `DoubleError`.
// This will be automatically called by `?` if a `ParseIntError`
// needs to be converted into a `DoubleError`.
impl From<ParseIntError> for DoubleError {
    fn from(err: ParseIntError) -> DoubleError {
        DoubleError::Parse(err)
    }
}

fn double_first(vec: Vec<&str>) -> Result<i32> {
    let first = vec.first().ok_or(DoubleError::EmptyVec)?;
    let parsed = first.parse::<i32>()?;

    Ok(2 * parsed)
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

This adds a bit more boilerplate for handling errors and might not be needed in all applications. There are some libraries that can take care of the boilerplate for you.

See also:

`From::from` and `Enums`

-
-
-
-
-

Iterating over `Results`

An `Iter::map` operation might fail, for example:

```
fn main() {
    let strings = vec!["tofu", "93", "18"];
    let possible_numbers: Vec<_> = strings
        .into_iter()
        .map(|s| s.parse::<i32>())
        .collect();
    println!("Results: {:?}", possible_numbers);
}
```

Let's step through strategies for handling this.

Ignore the failed items with `filter_map()`

`filter_map` calls a function and filters out the results that are `None`.

```
fn main() {
    let strings = vec!["tofu", "93", "18"];
    let numbers: Vec<_> = strings
        .into_iter()
        .map(|s| s.parse::<i32>())
        .filter_map(Result::ok)
        .collect();
    println!("Results: {:?}", numbers);
}
```

Fail the entire operation with `collect()`

`Result` implements `FromIter` so that a vector of results (`Vec<Result<T, E>>`) can be turned into a result with a vector (`Result<Vec<T>, E>`). Once an `Result::Err` is found, the iteration will terminate.

```
fn main() {
    let strings = vec!["tofu", "93", "18"];
    let numbers: Result<Vec<_>, _> = strings
        .into_iter()
        .map(|s| s.parse::<i32>())
        .collect();
    println!("Results: {:?}", numbers);
}
```

This same technique can be used with `Option`.

Collect all valid values and failures with `partition()`

```
fn main() {
    let strings = vec!["tofu", "93", "18"];
    let (numbers, errors): (Vec<_>, Vec<_>) = strings
        .into_iter()
        .map(|s| s.parse::<i32>())
        .partition(Result::is_ok);
    println!("Numbers: {:?}", numbers);
    println!("Errors: {:?}", errors);
}
```

When you look at the results, you'll note that everything is still wrapped in `Result`. A little more boilerplate is needed for this.

```
fn main() {
    let strings = vec!["tofu", "93", "18"];
    let (numbers, errors): (Vec<_>, Vec<_>) = strings
        .into_iter()
        .map(|s| s.parse::<i32>())
        .partition(Result::is_ok);
    let numbers: Vec<_> = numbers.into_iter().map(Result::unwrap).collect();
    let errors: Vec<_> = errors.into_iter().map(Result::unwrap_err).collect();
    println!("Numbers: {:?}", numbers);
    println!("Errors: {:?}", errors);
}
```

Search this book ...

Std library types

The `std` library provides many custom types which expands drastically on the `primitives`. Some of these include:

- growable `Strings` like: `"hello world"`
- growable vectors: `[1, 2, 3]`
- optional types: `Option<i32>`
- error handling types: `Result<i32, i32>`
- heap allocated pointers: `Box<i32>`

See also:

primitives and the std library

-
-
-
-
-

Rust By Example

Search this book ...

Box, stack and heap

All values in Rust are stack allocated by default. Values can be *boxed* (allocated in the heap) by creating a `Box<T>`. A box is a smart pointer to a heap allocated value of type `T`. When a box goes out of scope, its destructor is called, the inner object is destroyed, and the memory in the heap is freed.

Boxed values can be dereferenced using the `*` operator; this removes one layer of indirection.

```
use std::mem;

#[allow(dead_code)]
#[derive(Debug, Clone, Copy)]
struct Point {
    x: f64,
    y: f64,
}

#[allow(dead_code)]
struct Rectangle {
    p1: Point,
    p2: Point,
}

fn origin() -> Point {
    Point { x: 0.0, y: 0.0 }
}

fn boxed_origin() -> Box<Point> {
    // Allocate this point in the heap, and return a pointer to it
    Box::new(Point { x: 0.0, y: 0.0 })
}

fn main() {
    // (all the type annotations are superfluous)
    // Stack allocated variables
    let point: Point = origin();
    let rectangle: Rectangle = Rectangle {
        p1: origin(),
        p2: Point { x: 3.0, y: 4.0 }
    };

    // Heap allocated rectangle
    let boxed_rectangle: Box<Rectangle> = Box::new(Rectangle {
        p1: origin(),
        p2: origin()
    });

    // The output of functions can be boxed
    let boxed_point: Box<Point> = Box::new(origin());

    // Double indirection
    let box_in_a_box: Box<Box<Point>> = Box::new(boxed_origin());

    println!("Point occupies {} bytes in the stack",
             mem::size_of_val(&point));
    println!("Rectangle occupies {} bytes in the stack",
             mem::size_of_val(&rectangle));

    // box size = pointer size
    println!("Boxed point occupies {} bytes in the stack",
             mem::size_of_val(&boxed_point));
    println!("Boxed rectangle occupies {} bytes in the stack",
             mem::size_of_val(&boxed_rectangle));
```

```
    println!("Boxed box occupies {} bytes in the stack",
             mem::size_of_val(&box_in_a_box));

    // Copy the data contained in `boxed_point` into `unboxed_point`
    let unboxed_point: Point = *boxed_point;
    println!("Unboxed point occupies {} bytes in the stack",
             mem::size_of_val(&unboxed_point));
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Vectors

Vectors are re-sizable arrays. Like slices, their size is not known at compile time, but they can grow or shrink at any time. A vector is represented using 3 words: a pointer to the data, its length, and its capacity. The capacity indicates how much memory is reserved for the vector. The vector can grow as long as the length is smaller than the capacity. When this threshold needs to be surpassed, the vector is reallocated with a larger capacity.

```
fn main() {
    // Iterators can be collected into vectors
    let collected_iterator: Vec<i32> = (0..10).collect();
    println!("Collected (0..10) into: {:?}", collected_iterator);

    // The `vec!` macro can be used to initialize a vector
    let mut xs = vec![1i32, 2, 3];
    println!("Initial vector: {:?}", xs);

    // Insert new element at the end of the vector
    println!("Push 4 into the vector");
    xs.push(4);
    println!("Vector: {:?}", xs);

    // Error! Immutable vectors can't grow
    collected_iterator.push(0);
    // FIXME ^ Comment out this line

    // The `len` method yields the current size of the vector
    println!("Vector size: {}", xs.len());

    // Indexing is done using the square brackets (indexing starts at 0)
    println!("Second element: {}", xs[1]);

    // `pop` removes the last element from the vector and returns it
    println!("Pop last element: {:?}", xs.pop());

    // Out of bounds indexing yields a panic
    println!("Fourth element: {}", xs[3]);
    // FIXME ^ Comment out this line

    // `Vector`s can be easily iterated over
    println!("Contents of xs:");
    for x in xs.iter() {
        println!("> {}", x);
    }

    // A `Vector` can also be iterated over while the iteration
    // count is enumerated in a separate variable (`i`)
    for (i, x) in xs.iter().enumerate() {
        println!("In position {} we have value {}", i, x);
    }

    // Thanks to `iter_mut`, mutable `Vector`s can also be iterated
    // over in a way that allows modifying each value
    for x in xs.iter_mut() {
        *x *= 3;
    }
    println!("Updated vector: {:?}", xs);
}
```

More `Vec` methods can be found under the std::vec module

-
-
-
-
-

Rust By Example

Search this book ...

Strings

There are two types of strings in Rust: `String` and `&str`.

A `String` is stored as a vector of bytes (`Vec<u8>`), but guaranteed to always be a valid UTF-8 sequence. `String` is heap allocated, growable and not null terminated.

`&str` is a slice (`&[u8]`) that always points to a valid UTF-8 sequence, and can be used to view into a `String`, just like `&[T]` is a view into `Vec<T>`.

```
fn main() {
    // (all the type annotations are superfluous)
    // A reference to a string allocated in read only memory
    let pangram: &'static str = "the quick brown fox jumps over the lazy dog";
    println!("Pangram: {}", pangram);

    // Iterate over words in reverse, no new string is allocated
    println!("Words in reverse");
    for word in pangram.split_whitespace().rev() {
        println!("> {}", word);
    }

    // Copy chars into a vector, sort and remove duplicates
    let mut chars: Vec<char> = pangram.chars().collect();
    chars.sort();
    chars.dedup();

    // Create an empty and growable `String`
    let mut string = String::new();
    for c in chars {
        // Insert a char at the end of string
        string.push(c);
        // Insert a string at the end of string
        string.push_str(", ");
    }

    // The trimmed string is a slice to the original string, hence no new
    // allocation is performed
    let chars_to_trim: &[char] = &[' ', ','];
    let trimmed_str: &str = string.trim_matches(chars_to_trim);
    println!("Used characters: {}", trimmed_str);

    // Heap allocate a string
    let alice = String::from("I like dogs");
    // Allocate new memory and store the modified string there
    let bob: String = alice.replace("dog", "cat");

    println!("Alice says: {}", alice);
    println!("Bob says: {}", bob);
}
```

More `str`/`String` methods can be found under the std::str and std::string modules

Literals and escapes

There are multiple ways to write string literals with special characters in them. All result in a similar`&str` so it's best to use the form that is the most convenient to write. Similarly there are multiple ways to write byte string literals, which all result in `&[u8; N]`.

Generally special characters are escaped with a backslash character: `\`. This way you can add any character to your string, even unprintable ones and ones that you don't know how to type. If you want a literal backslash, escape it with another one: `\\`

String or character literal delimiters occuring within a literal must be escaped: `"\""`, `'\''`.

```
fn main() {
    // You can use escapes to write bytes by their hexadecimal values...
    let byte_escape = "I'm writing \x52\x75\x73\x74!";
    println!("What are you doing\x3F (\\x3F means ?) {}", byte_escape);

    // ...or Unicode code points.
    let unicode_codepoint = "\u{211D}";
    let character_name = "\"DOUBLE-STRUCK CAPITAL R\"";

    println!("Unicode character {} (U+211D) is called {}",
                unicode_codepoint, character_name );


    let long_string = "String literals
                        can span multiple lines.
                        The linebreak and indentation here ->\
                        <- can be escaped too!";
    println!("{}", long_string);
}
```

Sometimes there are just too many characters that need to be escaped or it's just much more convenient to write a string out as-is. This is where raw string literals come into play.

```
fn main() {
    let raw_str = r"Escapes don't work here: \x3F \u{211D}";
    println!("{}", raw_str);

    // If you need quotes in a raw string, add a pair of #s
    let quotes = r#"And then I said: "There is no escape!""#;
    println!("{}", quotes);

    // If you need "# in your string, just use more #s in the delimiter.
    // There is no limit for the number of #s you can use.
    let longer_delimiter = r###"A string with "# in it. And even "##!"###;
    println!("{}", longer_delimiter);
}
```

Want a string that's not UTF-8? (Remember, `str` and `String` must be valid UTF-8) Or maybe you want an array of bytes that's mostly text? Byte strings to the rescue!

```
use std::str;

fn main() {
    // Note that this is not actually a &str
    let bytestring: &[u8; 20] = b"this is a bytestring";

    // Byte arrays don't have Display so printing them is a bit limited
    println!("A bytestring: {:?}", bytestring);

    // Bytestrings can have byte escapes...
    let escaped = b"\x52\x75\x73\x74 as bytes";
    // ...but no unicode escapes
    // let escaped = b"\u{211D} is not allowed";
    println!("Some escaped bytes: {:?}", escaped);


    // Raw bytestrings work just like raw strings
    let raw_bytestring = br"\u{211D} is not escaped here";
    println!("{:?}", raw_bytestring);

    // Converting a byte array to str can fail
    if let Ok(my_str) = str::from_utf8(raw_bytestring) {
        println!("And the same as text: '{}'", my_str);
    }

    let quotes = br#"You can also use "fancier" formatting, \
                    like with normal raw strings"#;

    // Bytestrings don't have to be UTF-8
    let shift_jis = b"\x82\xe6\x82\xa8\x82\xb1\x82"; // "ã,ˆã†ã“ã" in SHIFT-JIS

    // But then they can't always be converted to str
    match str::from_utf8(shift_jis) {
        Ok(my_str) => println!("Conversion successful: '{}'", my_str),
        Err(e) => println!("Conversion failed: {:?}", e),
    };
}
```

For conversions between character encodings check out the [encoding]() crate.

A more detailed listing of the ways to write string literals and escape characters is given in the ['Tokens' chapter]() of the Rust Reference.

-
-
-
-
-

Rust By Example

Search this book ...

Option

Sometimes it's desirable to catch the failure of some parts of a program instead of calling `panic!`; this can be accomplished using the `Option` enum.

The `Option<T>` enum has two variants:

- `None`, to indicate failure or lack of value, and
- `Some(value)`, a tuple struct that wraps a `value` with type `T`.

```
// An integer division that doesn't `panic!`
fn checked_division(dividend: i32, divisor: i32) -> Option<i32> {
    if divisor == 0 {
        // Failure is represented as the `None` variant
        None
    } else {
        // Result is wrapped in a `Some` variant
        Some(dividend / divisor)
    }
}

// This function handles a division that may not succeed
fn try_division(dividend: i32, divisor: i32) {
    // `Option` values can be pattern matched, just like other enums
    match checked_division(dividend, divisor) {
        None => println!("{} / {} failed!", dividend, divisor),
        Some(quotient) => {
            println!("{} / {} = {}", dividend, divisor, quotient)
        },
    }
}

fn main() {
    try_division(4, 2);
    try_division(1, 0);

    // Binding `None` to a variable needs to be type annotated
    let none: Option<i32> = None;
    let _equivalent_none = None::<i32>;

    let optional_float = Some(0f32);
```

```
    // Unwrapping a `Some` variant will extract the value wrapped.
    println!("{:?} unwraps to {:?}", optional_float, optional_float.unwrap());

    // Unwrapping a `None` variant will `panic!`
    println!("{:?} unwraps to {:?}", none, none.unwrap());
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Result

We've seen that the `Option` enum can be used as a return value from functions that may fail, where `None` can be returned to indicate failure. However, sometimes it is important to express *why* an operation failed. To do this we have the `Result` enum.

The `Result<T, E>` enum has two variants:

- `Ok(value)` which indicates that the operation succeeded, and wraps the `value` returned by the operation. (`value` has type `T`)
- `Err(why)`, which indicates that the operation failed, and wraps `why`, which (hopefully) explains the cause of the failure. (`why` has type `E`)

```
mod checked {
    // Mathematical "errors" we want to catch
    #[derive(Debug)]
    pub enum MathError {
        DivisionByZero,
        NonPositiveLogarithm,
        NegativeSquareRoot,
    }

    pub type MathResult = Result<f64, MathError>;

    pub fn div(x: f64, y: f64) -> MathResult {
        if y == 0.0 {
            // This operation would `fail`, instead let's return the reason of
            // the failure wrapped in `Err`
            Err(MathError::DivisionByZero)
        } else {
            // This operation is valid, return the result wrapped in `Ok`
            Ok(x / y)
        }
    }

    pub fn sqrt(x: f64) -> MathResult {
        if x < 0.0 {
            Err(MathError::NegativeSquareRoot)
        } else {
            Ok(x.sqrt())
        }
    }

    pub fn ln(x: f64) -> MathResult {
        if x <= 0.0 {
            Err(MathError::NonPositiveLogarithm)
        } else {
            Ok(x.ln())
        }
    }
}

// `op(x, y)` === `sqrt(ln(x / y))`
fn op(x: f64, y: f64) -> f64 {
    // This is a three level match pyramid!
    match checked::div(x, y) {
        Err(why) => panic!("{:?}", why),
        Ok(ratio) => match checked::ln(ratio) {
            Err(why) => panic!("{:?}", why),
            Ok(ln) => match checked::sqrt(ln) {
                Err(why) => panic!("{:?}", why),
                Ok(sqrt) => sqrt,
            },
        },
    }
}

fn main() {
    // Will this fail?
    println!("{}", op(1.0, 10.0));
}
```

-
-

?

Chaining results using match can get pretty untidy; luckily, the `?` operator can be used to make things pretty again. `?` is used at the end of an expression returning a `Result`, and is equivalent to a match expression, where the `Err(err)` branch expands to an early `Err(From::from(err))`, and the `Ok(ok)` branch expands to an `ok` expression.

```
mod checked {
    #[derive(Debug)]
    enum MathError {
        DivisionByZero,
        NonPositiveLogarithm,
        NegativeSquareRoot,
    }

    type MathResult = Result<f64, MathError>;

    fn div(x: f64, y: f64) -> MathResult {
        if y == 0.0 {
            Err(MathError::DivisionByZero)
        } else {
            Ok(x / y)
        }
    }

    fn sqrt(x: f64) -> MathResult {
        if x < 0.0 {
            Err(MathError::NegativeSquareRoot)
        } else {
            Ok(x.sqrt())
        }
    }

    fn ln(x: f64) -> MathResult {
        if x <= 0.0 {
            Err(MathError::NonPositiveLogarithm)
        } else {
            Ok(x.ln())
        }
    }

    // Intermediate function
    fn op_(x: f64, y: f64) -> MathResult {
        // if `div` "fails", then `DivisionByZero` will be `return`ed
        let ratio = div(x, y)?;

        // if `ln` "fails", then `NonPositiveLogarithm` will be `return`ed
        let ln = ln(ratio)?;

        sqrt(ln)
    }

    pub fn op(x: f64, y: f64) {
        match op_(x, y) {
            Err(why) => panic!(match why {
                MathError::NonPositiveLogarithm
                    => "logarithm of non-positive number",
                MathError::DivisionByZero
                    => "division by zero",
                MathError::NegativeSquareRoot
                    => "square root of negative number",
            }),
            Ok(value) => println!("{}", value),
        }
    }
}

fn main() {
    checked::op(1.0, 10.0);
}
```

Be sure to check the documentation, as there are many methods to map/compose `Result`.

panic!

The `panic!` macro can be used to generate a panic and start unwinding its stack. While unwinding, the runtime will take care of freeing all the resources *owned* by the thread by calling the destructor of all its objects.

Since we are dealing with programs with only one thread, `panic!` will cause the program to report the panic message and exit.

```
// Re-implementation of integer division (/)
fn division(dividend: i32, divisor: i32) -> i32 {
    if divisor == 0 {
        // Division by zero triggers a panic
        panic!("division by zero");
    } else {
        dividend / divisor
    }
}

// The `main` task
fn main() {
    // Heap allocated integer
    let _x = Box::new(0i32);

    // This operation will trigger a task failure
    division(3, 0);

    println!("This point won't be reached!");

    // `_x` should get destroyed at this point
}
```

Let's check that `panic!` doesn't leak memory.

```
$ rustc panic.rs && valgrind ./panic
==4401== Memcheck, a memory error detector
==4401== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4401== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==4401== Command: ./panic
==4401==
thread '<main>' panicked at 'division by zero', panic.rs:5
==4401==
==4401== HEAP SUMMARY:
==4401==     in use at exit: 0 bytes in 0 blocks
==4401==   total heap usage: 18 allocs, 18 frees, 1,648 bytes allocated
==4401==
==4401== All heap blocks were freed -- no leaks are possible
==4401==
==4401== For counts of detected and suppressed errors, rerun with: -v
==4401== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rust By Example

HashMap

Where vectors store values by an integer index, `HashMap`s store values by key. `HashMap` keys can be booleans, integers, strings, or any other type that implements the `Eq` and `Hash` traits. More on this in the next section.

Like vectors, `HashMap`s are growable, but HashMaps can also shrink themselves when they have excess space. You can create a HashMap with a certain starting capacity using `HashMap::with_capacity(uint)`, or use `HashMap::new()` to get a HashMap with a default initial capacity (recommended).

```
use std::collections::HashMap;

fn call(number: &str) -> &str {
    match number {
        "798-1364" => "We're sorry, the call cannot be completed as dialed.
            Please hang up and try again.",
        "645-7689" => "Hello, this is Mr. Awesome's Pizza. My name is Fred.
            What can I get for you today?",
        _ => "Hi! Who is this again?"
    }
}

fn main() {
    let mut contacts = HashMap::new();

    contacts.insert("Daniel", "798-1364");
    contacts.insert("Ashley", "645-7689");
    contacts.insert("Katie", "435-8291");
    contacts.insert("Robert", "956-1745");

    // Takes a reference and returns Option<&V>
    match contacts.get(&"Daniel") {
```

```
        Some(&number) => println!("Calling Daniel: {}", call(number)),
        _ => println!("Don't have Daniel's number."),
    }

    // `HashMap::insert()` returns `None`
    // if the inserted value is new, `Some(value)` otherwise
    contacts.insert("Daniel", "164-6743");

    match contacts.get(&"Ashley") {
        Some(&number) => println!("Calling Ashley: {}", call(number)),
        _ => println!("Don't have Ashley's number."),
    }

    contacts.remove(&"Ashley");

    // `HashMap::iter()` returns an iterator that yields
    // (&'a key, &'a value) pairs in arbitrary order.
    for (contact, &number) in contacts.iter() {
        println!("Calling {}: {}", contact, call(number));
    }
}
```

For more information on how hashing and hash maps (sometimes called hash tables) work, have a look at [Hash Table Wikipedia](https://en.wikipedia.org/wiki/Hash_table)

[Alternate/custom key types]()

Any type that implements the `Eq` and `Hash` traits can be a key in `HashMap`. This includes:

- `bool` (though not very useful since there is only two possible keys)
- `int`, `uint`, and all variations thereof
- `String` and `&str` (protip: you can have a `HashMap` keyed by `String` and call `.get()` with an `&str`)

Note that `f32` and `f64` do *not* implement `Hash`, likely because [floating-point precision errors]() would make using them as hashmap keys horribly error-prone.

All collection classes implement `Eq` and `Hash` if their contained type also respectively implements `Eq` and `Hash`. For example, `Vec<T>` will implement `Hash` if `T` implements `Hash`.

You can easily implement `Eq` and `Hash` for a custom type with just one line: `#[derive(PartialEq, Eq, Hash)]`

The compiler will do the rest. If you want more control over the details, you can implement `Eq` and/or `Hash` yourself. This guide will not cover the specifics of implementing `Hash`.

To play around with using a `struct` in `HashMap`, let's try making a very simple user logon system:

```
use std::collections::HashMap;

// Eq requires that you derive PartialEq on the type.
#[derive(PartialEq, Eq, Hash)]
struct Account<'a>{
    username: &'a str,
    password: &'a str,
}

struct AccountInfo<'a>{
    name: &'a str,
    email: &'a str,
}

type Accounts<'a> = HashMap<Account<'a>, AccountInfo<'a>>;

fn try_logon<'a>(accounts: &Accounts<'a>,
        username: &'a str, password: &'a str){
    println!("Username: {}", username);
    println!("Password: {}", password);
    println!("Attempting logon...");

    let logon = Account {
        username: username,
        password: password,
    };

    match accounts.get(&logon) {
        Some(account_info) => {
            println!("Successful logon!");
            println!("Name: {}", account_info.name);
            println!("Email: {}", account_info.email);
        },
        _ => println!("Login failed!"),
    }
}

fn main(){
```

```
    let mut accounts: Accounts = HashMap::new();

    let account = Account {
        username: "j.everyman",
        password: "password123",
    };

    let account_info = AccountInfo {
        name: "John Everyman",
        email: "j.everyman@email.com",
    };

    accounts.insert(account, account_info);

    try_logon(&accounts, "j.everyman", "psasword123");

    try_logon(&accounts, "j.everyman", "password123");
}
```

-
-
-
-
-

Rust By Example

Search this book ...

HashSet

Consider a `HashSet` as a `HashMap` where we just care about the keys (`HashSet<T>` is, in actuality, just a wrapper around `HashMap<T, ()>`).

"What's the point of that?" you ask. "I could just store the keys in a `Vec`."

A `HashSet`'s unique feature is that it is guaranteed to not have duplicate elements. That's the contract that any set collection fulfills. `HashSet` is just one implementation. (see also: `BTreeSet`)

If you insert a value that is already present in the `HashSet`, (i.e. the new value is equal to the existing and they both have the same hash), then the new value will replace the old.

This is great for when you never want more than one of something, or when you want to know if you've already got something.

But sets can do more than that.

Sets have 4 primary operations (all of the following calls return an iterator):

- `union`: get all the unique elements in both sets.
- `difference`: get all the elements that are in the first set but not the second.
- `intersection`: get all the elements that are only in *both* sets.
- `symmetric_difference`: get all the elements that are in one set or the other, but *not* both.

Try all of these in the following example:

```
use std::collections::HashSet;

fn main() {
    let mut a: HashSet<i32> = vec![1i32, 2, 3].into_iter().collect();
    let mut b: HashSet<i32> = vec![2i32, 3, 4].into_iter().collect();

    assert!(a.insert(4));
    assert!(a.contains(&4));

    // `HashSet::insert()` returns false if
    // there was a value already present.
    assert!(b.insert(4), "Value 4 is already in set B!");
    // FIXME ^ Comment out this line

    b.insert(5);

    // If a collection's element type implements `Debug`,
    // then the collection implements `Debug`.
    // It usually prints its elements in the format `[elem1, elem2, ...]`
    println!("A: {:?}", a);
    println!("B: {:?}", b);

    // Print [1, 2, 3, 4, 5] in arbitrary order
    println!("Union: {:?}", a.union(&b).collect::<Vec<&i32>>());

    // This should print [1]
    println!("Difference: {:?}", a.difference(&b).collect::<Vec<&i32>>());

    // Print [2, 3, 4] in arbitrary order.
    println!("Intersection: {:?}", a.intersection(&b).collect::<Vec<&i32>>());

    // Print [1, 5]
    println!("Symmetric Difference: {:?}",
```

```
            a.symmetric_difference(&b).collect::<Vec<&i32>>());
}
```

(Examples are adapted from the documentation.)

Rust By Example

Search this book ...

Std misc

Many other types are provided by the std library to support things such as:

- Threads
- Channels
- File I/O

These expand beyond what the primitives provide.

See also:

primitives and the std library

Rust By Example

Search this book ...

Threads

Rust provides a mechanism for spawning native OS threads via the `spawn` function, the argument of this function is a moving closure.

```
use std::thread;

static NTHREADS: i32 = 10;

// This is the `main` thread
fn main() {
    // Make a vector to hold the children which are spawned.
    let mut children = vec![];

    for i in 0..NTHREADS {
        // Spin up another thread
        children.push(thread::spawn(move || {
            println!("this is thread number {}", i);
        }));
    }

    for child in children {
        // Wait for the thread to finish. Returns a result.
        let _ = child.join();
    }
}
```

These threads will be scheduled by the OS.

-
-
-
-
-

Rust By Example

Search this book ...

Testcase: map-reduce

Rust makes it very easy to parallelise data processing, without many of the headaches traditionally associated with such an attempt.

The standard library provides great threading primitives out of the box. These, combined with Rust's concept of Ownership and aliasing rules, automatically prevent data races.

The aliasing rules (one writable reference XOR many readable references) automatically prevent you from manipulating state that is visible to other threads. (Where synchronisation is needed, there are synchronisation primitives like `Mutex`es or `Channel`s.)

In this example, we will calculate the sum of all digits in a block of numbers. We will do this by parcelling out chunks of the block into different threads. Each thread will sum its tiny block of digits, and subsequently we will sum the intermediate sums produced by each thread.

Note that, although we're passing references across thread boundaries, Rust understands that we're only passing read-only references, and that thus no unsafety or data races can occur. Because we're `move`-ing the data segments into the thread, Rust will also ensure the data is kept alive until the threads exit, so no dangling pointers occur.

```
use std::thread;

// This is the `main` thread
fn main() {

    // This is our data to process.
    // We will calculate the sum of all digits via a threaded  map-reduce algorithm.
    // Each whitespace separated chunk will be handled in a different thread.
    //
    // TODO: see what happens to the output if you insert spaces!
    let data = "86967897737416471853297327050364959
11861322575564723963297542624962850
70856234701860851907960690014725639
38397966707106094172783238747669219
52380795257888236525459303330302837
58495327135744041048897885734297812
69920216438980873548808413720956532
16278424637452589860345374828574668";

    // Make a vector to hold the child-threads which we will spawn.
    let mut children = vec![];

    /*************************************************************************
     * "Map" phase
     *
     * Divide our data into segments, and apply initial processing
     ************************************************************************/

    // split our data into segments for individual calculation
    // each chunk will be a reference (&str) into the actual data
    let chunked_data = data.split_whitespace();

    // Iterate over the data segments.
    // .enumerate() adds the current loop index to whatever is iterated
    // the resulting tuple "(index, element)" is then immediately
    // "destructured" into two variables, "i" and "data_segment" with a
    // "destructuring assignment"
    for (i, data_segment) in chunked_data.enumerate() {
        println!("data segment {} is \"{}\"", i, data_segment);

        // Process each data segment in a separate thread
        //
        // spawn() returns a handle to the new thread,
        // which we MUST keep to access the returned value
        //
        // 'move || -> u32' is syntax for a closure that:
        // * takes no arguments ('||')
        // * takes ownership of its captured variables ('move') and
        // * returns an unsigned 32-bit integer ('-> u32')
        //
        // Rust is smart enough to infer the '-> u32' from
        // the closure itself so we could have left that out.
        //
        // TODO: try removing the 'move' and see what happens
        children.push(thread::spawn(move || -> u32 {
            // Calculate the intermediate sum of this segment:
            let result = data_segment
                        // iterate over the characters of our segment..
                        .chars()
                        // .. convert text-characters to their number value..
                        .map(|c| c.to_digit(10).expect("should be a digit"))
                        // .. and sum the resulting iterator of numbers
                        .sum();

            // println! locks stdout, so no text-interleaving occurs
            println!("processed segment {}, result={}", i, result);

            // "return" not needed, because Rust is an "expression language", the
            // last evaluated expression in each block is automatically its value.
            result
```

```
        }));
    }

    /*************************************************************************
     * "Reduce" phase
     *
     * Collect our intermediate results, and combine them into a final result
     ************************************************************************/

    // collect each thread's intermediate results into a new Vec
    let mut intermediate_sums = vec![];
    for child in children {
        // collect each child thread's return-value
        let intermediate_sum = child.join().unwrap();
        intermediate_sums.push(intermediate_sum);
    }

    // combine all intermediate sums into a single final sum.
    //
    // we use the "turbofish" ::<> to provide sum() with a type hint.
    //
    // TODO: try without the turbofish, by instead explicitly
    // specifying the type of final_result
    let final_result = intermediate_sums.iter().sum::<u32>();

    println!("Final sum result: {}", final_result);
}
```

Assignments

It is not wise to let our number of threads depend on user inputted data. What if the user decides to insert a lot of spaces? Do we *really* want to spawn 2,000 threads? Modify the program so that the data is always chunked into a limited number of chunks, defined by a static constant at the beginning of the program.

See also:

- Threads
- vectors and iterators
- closures, move semantics and `move` closures
- destructuring assignments
- turbofish notation to help type inference
- unwrap vs. expect
- enumerate

Rust By Example

Search this book ...

Channels

Rust provides asynchronous `channels` for communication between threads. Channels allow a unidirectional flow of information between two end-points: the `Sender` and the `Receiver`.

```
use std::sync::mpsc::{Sender, Receiver};
use std::sync::mpsc;
use std::thread;

static NTHREADS: i32 = 3;

fn main() {
    // Channels have two endpoints: the `Sender<T>` and the `Receiver<T>`,
    // where `T` is the type of the message to be transferred
    // (type annotation is superfluous)
    let (tx, rx): (Sender<i32>, Receiver<i32>) = mpsc::channel();

    for id in 0..NTHREADS {
        // The sender endpoint can be copied
        let thread_tx = tx.clone();

        // Each thread will send its id via the channel
        thread::spawn(move || {
            // The thread takes ownership over `thread_tx`
            // Each thread queues a message in the channel
            thread_tx.send(id).unwrap();

            // Sending is a non-blocking operation, the thread will continue
            // immediately after sending its message
            println!("thread {} finished", id);
        });
    }

    // Here, all the messages are collected
```

```
    let mut ids = Vec::with_capacity(NTHREADS as usize);
    for _ in 0..NTHREADS {
        // The `recv` method picks a message from the channel
        // `recv` will block the current thread if there are no messages available
        ids.push(rx.recv());
    }

    // Show the order in which the messages were sent
    println!("{:?}", ids);
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Path

The `Path` struct represents file paths in the underlying filesystem. There are two flavors of `Path`: `posix::Path`, for UNIX-like systems, and `windows::Path`, for Windows. The prelude exports the appropriate platform-specific `Path` variant.

A `Path` can be created from an `OsStr`, and provides several methods to get information from the file/directory the path points to.

Note that a `Path` is *not* internally represented as an UTF-8 string, but instead is stored as a vector of bytes (`Vec<u8>`). Therefore, converting a `Path` to a `&str` is *not* free and may fail (an `Option` is returned).

```
use std::path::Path;

fn main() {
    // Create a `Path` from an `&'static str`
    let path = Path::new(".");

    // The `display` method returns a `Show`able structure
    let _display = path.display();

    // `join` merges a path with a byte container using the OS specific
    // separator, and returns the new path
    let new_path = path.join("a").join("b");

    // Convert the path into a string slice
    match new_path.to_str() {
        None => panic!("new path is not a valid UTF-8 sequence"),
        Some(s) => println!("new path is {}", s),
    }
}
```

Be sure to check at other `Path` methods (`posix::Path` or `windows::Path`) and the `Metadata` struct.

See also

OsStr and Metadata.

-
-
-
-
-

Rust By Example

Search this book ...

File I/O

The `File` struct represents a file that has been opened (it wraps a file descriptor), and gives read and/or write access to the underlying file.

Since many things can go wrong when doing file I/O, all the `File` methods return the `io::Result<T>` type, which is an alias for `Result<T, io::Error>`.

This makes the failure of all I/O operations *explicit*. Thanks to this, the programmer can see all the failure paths, and is encouraged to handle them in a proactive manner.

open

The `open` static method can be used to open a file in read-only mode.

A `File` owns a resource, the file descriptor and takes care of closing the file when it is `drop`ed.

```
use std::error::Error;
use std::fs::File;
use std::io::prelude::*;
use std::path::Path;

fn main() {
    // Create a path to the desired file
    let path = Path::new("hello.txt");
    let display = path.display();

    // Open the path in read-only mode, returns `io::Result<File>`
    let mut file = match File::open(&path) {
        // The `description` method of `io::Error` returns a string that
        // describes the error
        Err(why) => panic!("couldn't open {}: {}", display,
                                                   why.description()),
        Ok(file) => file,
    };

    // Read the file contents into a string, returns `io::Result<usize>`
    let mut s = String::new();
    match file.read_to_string(&mut s) {
        Err(why) => panic!("couldn't read {}: {}", display,
                                                   why.description()),
        Ok(_) => print!("{} contains:\n{}", display, s),
    }

    // `file` goes out of scope, and the "hello.txt" file gets closed
}
```

Here's the expected successful output:

```
$ echo "Hello World!" > hello.txt
$ rustc open.rs && ./open
hello.txt contains:
Hello World!
```

(You are encouraged to test the previous example under different failure conditions: `hello.txt` doesn't exist, or `hello.txt` is not readable, etc.)

create

The `create` static method opens a file in write-only mode. If the file already existed, the old content is destroyed. Otherwise, a new file is created.

```
static LOREM_IPSUM: &'static str =
"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
";

use std::error::Error;
use std::io::prelude::*;
use std::fs::File;
use std::path::Path;

fn main() {
    let path = Path::new("out/lorem_ipsum.txt");
    let display = path.display();

    // Open a file in write-only mode, returns `io::Result<File>`
    let mut file = match File::create(&path) {
        Err(why) => panic!("couldn't create {}: {}",
                           display,
                           why.description()),
        Ok(file) => file,
    };

    // Write the `LOREM_IPSUM` string to `file`, returns `io::Result<()>`
    match file.write_all(LOREM_IPSUM.as_bytes()) {
        Err(why) => {
            panic!("couldn't write to {}: {}", display,
                                               why.description())
        },
        Ok(_) => println!("successfully wrote to {}", display),
    }
}
```

Here's the expected successful output:

```
$ mkdir out
$ rustc create.rs && ./create
successfully wrote to out/lorem_ipsum.txt
$ cat out/lorem_ipsum.txt
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
```

(As in the previous example, you are encouraged to test this example under failure conditions.)

There is also a more generic `open_mode` method that can open files in other modes like: read+write, append, etc.

-
-
-
-
-

Rust By Example

Search this book ...

Child processes

The `process::Output` struct represents the output of a finished child process, and the `process::Command` struct is a process builder.

```
use std::process::Command;

fn main() {
    let output = Command::new("rustc")
        .arg("--version")
        .output().unwrap_or_else(|e| {
            panic!("failed to execute process: {}", e)
    });

    if output.status.success() {
        let s = String::from_utf8_lossy(&output.stdout);

        print!("rustc succeeded and stdout was:\n{}", s);
    } else {
        let s = String::from_utf8_lossy(&output.stderr);

        print!("rustc failed and stderr was:\n{}", s);
    }
}
```

(You are encouraged to try the previous example with an incorrect flag passed to `rustc`)

-

-
-
-
-

Rust By Example

Search this book ...

Pipes

The `std::Child` struct represents a running child process, and exposes the `stdin`, `stdout` and `stderr` handles for interaction with the underlying process via pipes.

```
use std::error::Error;
use std::io::prelude::*;
use std::process::{Command, Stdio};

static PANGRAM: &'static str =
"the quick brown fox jumped over the lazy dog\n";

fn main() {
    // Spawn the `wc` command
    let process = match Command::new("wc")
                                .stdin(Stdio::piped())
                                .stdout(Stdio::piped())
                                .spawn() {
        Err(why) => panic!("couldn't spawn wc: {}", why.description()),
        Ok(process) => process,
    };

    // Write a string to the `stdin` of `wc`.
    //
    // `stdin` has type `Option<ChildStdin>`, but since we know this instance
    // must have one, we can directly `unwrap` it.
    match process.stdin.unwrap().write_all(PANGRAM.as_bytes()) {
        Err(why) => panic!("couldn't write to wc stdin: {}",
                           why.description()),
        Ok(_) => println!("sent pangram to wc"),
    }

    // Because `stdin` does not live after the above calls, it is `drop`ed,
    // and the pipe is closed.
    //
    // This is very important, otherwise `wc` wouldn't start processing the
    // input we just sent.

    // The `stdout` field also has type `Option<ChildStdout>` so must be unwrapped.
    let mut s = String::new();
    match process.stdout.unwrap().read_to_string(&mut s) {
        Err(why) => panic!("couldn't read wc stdout: {}",
                           why.description()),
        Ok(_) => print!("wc responded with:\n{}", s),
    }
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Wait

If you'd like to wait for a `process::Child` to finish, you must call `Child::wait`, which will return a `process::ExitStatus`.

```
use std::process::Command;

fn main() {
    let mut child = Command::new("sleep").arg("5").spawn().unwrap();
    let _result = child.wait().unwrap();

    println!("reached end of main");
}
```

```
$ rustc wait.rs && ./wait
# `wait` keeps running for 5 seconds until the `sleep 5` command finishes
reached end of main
```

Rust By Example

Search this book ...

Filesystem Operations

The `std::fs` module contains several functions that deal with the filesystem.

```
use std::fs;
use std::fs::{File, OpenOptions};
use std::io;
use std::io::prelude::*;
use std::os::unix;
use std::path::Path;

// A simple implementation of `% cat path`
fn cat(path: &Path) -> io::Result<String> {
    let mut f = File::open(path)?;
    let mut s = String::new();
    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}

// A simple implementation of `% echo s > path`
fn echo(s: &str, path: &Path) -> io::Result<()> {
    let mut f = File::create(path)?;

    f.write_all(s.as_bytes())
}

// A simple implementation of `% touch path` (ignores existing files)
fn touch(path: &Path) -> io::Result<()> {
    match OpenOptions::new().create(true).write(true).open(path) {
        Ok(_) => Ok(()),
        Err(e) => Err(e),
    }
}

fn main() {
    println!("`mkdir a`");
    // Create a directory, returns `io::Result<()>`
    match fs::create_dir("a") {
        Err(why) => println!("! {:?}", why.kind()),
        Ok(_) => {},
    }

    println!("`echo hello > a/b.txt`");
    // The previous match can be simplified using the `unwrap_or_else` method
    echo("hello", &Path::new("a/b.txt")).unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });

    println!("`mkdir -p a/c/d`");
    // Recursively create a directory, returns `io::Result<()>`
    fs::create_dir_all("a/c/d").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });

    println!("`touch a/c/e.txt`");
    touch(&Path::new("a/c/e.txt")).unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });

    println!("`ln -s ../b.txt a/c/b.txt`");
    // Create a symbolic link, returns `io::Result<()>`
    if cfg!(target_family = "unix") {
        unix::fs::symlink("../b.txt", "a/c/b.txt").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
        });
    }

    println!("`cat a/c/b.txt`");
    match cat(&Path::new("a/c/b.txt")) {
        Err(why) => println!("! {:?}", why.kind()),
        Ok(s) => println!("> {}", s),
    }

    println!("`ls a`");
    // Read the contents of a directory, returns `io::Result<Vec<Path>>`
    match fs::read_dir("a") {
        Err(why) => println!("! {:?}", why.kind()),
        Ok(paths) => for path in paths {
            println!("> {:?}", path.unwrap().path());
        },
    }

    println!("`rm a/c/e.txt`");
    // Remove a file, returns `io::Result<()>`
    fs::remove_file("a/c/e.txt").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
```

```
    });

    println!("`rmdir a/c/d`");
    // Remove an empty directory, returns `io::Result<()>`
    fs::remove_dir("a/c/d").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });
}
```

Here's the expected successful output:

```
$ rustc fs.rs && ./fs
`mkdir a`
`echo hello > a/b.txt`
`mkdir -p a/c/d`
`touch a/c/e.txt`
`ln -s ../b.txt a/c/b.txt`
`cat a/c/b.txt`
> hello
`ls a`
> "a/b.txt"
> "a/c"
`rm a/c/e.txt`
`rmdir a/c/d`
```

And the final state of the `a` directory is:

```
$ tree a
a
|-- b.txt
`-- c
    `-- b.txt -> ../b.txt

1 directory, 2 files
```

An alternative way to define the function `cat` is with `?` notation:

```
fn cat(path: &Path) -> io::Result<String> {
    let mut f = File::open(path)?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

See also:

`cfg!`

-
-
-
-
-

Rust By Example

Search this book ...

Program arguments

Standard Library

The command line arguments can be accessed using `std::env::args`, which returns an iterator that yields a `String` for each argument:

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    // The first argument is the path that was used to call the program.
    println!("My path is {}.", args[0]);

    // The rest of the arguments are the passed command line parameters.
    // Call the program like this:
    //   $ ./args arg1 arg2
    println!("I got {:?} arguments: {:?}.", args.len() - 1, &args[1..]);
}
```

```
$ ./args 1 2 3
My path is ./args.
I got 3 arguments: ["1", "2", "3"].
```

Crates

Alternatively, there are numerous crates that can provide extra functionality when creating command-line applications. The Rust Cookbook exhibits best practices on how to use one of the more popular command line argument crates, `clap`.

-
-
-
-
-

Rust By Example

Search this book ...

Argument parsing

Matching can be used to parse simple arguments:

```
use std::env;

fn increase(number: i32) {
    println!("{}", number + 1);
}

fn decrease(number: i32) {
    println!("{}", number - 1);
}

fn help() {
    println!("usage:
match_args <string>
    Check whether given string is the answer.
match_args {{increase|decrease}} <integer>
    Increase or decrease given integer by one.");
}

fn main() {
    let args: Vec<String> = env::args().collect();

    match args.len() {
        // no arguments passed
        1 => {
            println!("My name is 'match_args'. Try passing some arguments!");
        },
        // one argument passed
        2 => {
            match args[1].parse() {
                Ok(42) => println!("This is the answer!"),
                _ => println!("This is not the answer."),
            }
        },
        // one command and one argument passed
        3 => {
            let cmd = &args[1];
            let num = &args[2];
            // parse the number
            let number: i32 = match num.parse() {
                Ok(n) => {
                    n
                },
                Err(_) => {
                    eprintln!("error: second argument not an integer");
                    help();
                    return;
                },
            };
            // parse the command
            match &cmd[..] {
                "increase" => increase(number),
                "decrease" => decrease(number),
                _ => {
                    eprintln!("error: invalid command");
                    help();
                },
            }
        },
        // all the other cases
        _ => {
            // show a help message
            help();
        }
    }
}
```

```
$ ./match_args Rust
This is not the answer.
$ ./match_args 42
This is the answer!
$ ./match_args do something
error: second argument not an integer
usage:
match_args <string>
    Check whether given string is the answer.
match_args {increase|decrease} <integer>
    Increase or decrease given integer by one.
$ ./match_args do 42
error: invalid command
usage:
match_args <string>
    Check whether given string is the answer.
match_args {increase|decrease} <integer>
    Increase or decrease given integer by one.
```

```
$ ./match_args increase 42
43
```

-
-
-
-
-

Rust By Example

Search this book ...

Foreign Function Interface

Rust provides a Foreign Function Interface (FFI) to C libraries. Foreign functions must be declared inside an `extern` block annotated with a `#[link]` attribute containing the name of the foreign library.

```
use std::fmt;

// this extern block links to the libm library
#[link(name = "m")]
extern {
    // this is a foreign function
    // that computes the square root of a single precision complex number
    fn csqrtf(z: Complex) -> Complex;

    fn ccosf(z: Complex) -> Complex;
}

// Since calling foreign functions is considered unsafe,
// it's common to write safe wrappers around them.
fn cos(z: Complex) -> Complex {
    unsafe { ccosf(z) }
}

fn main() {
    // z = -1 + 0i
    let z = Complex { re: -1., im: 0. };

    // calling a foreign function is an unsafe operation
    let z_sqrt = unsafe { csqrtf(z) };

    println!("the square root of {:?} is {:?}", z, z_sqrt);

    // calling safe API wrapped around unsafe operation
    println!("cos({:?}) = {:?}", z, cos(z));
}

// Minimal implementation of single precision complex numbers
#[repr(C)]
#[derive(Clone, Copy)]
struct Complex {
    re: f32,
    im: f32,
}

impl fmt::Debug for Complex {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        if self.im < 0. {
            write!(f, "{}-{}i", self.re, -self.im)
        } else {
            write!(f, "{}+{}i", self.re, self.im)
        }
    }
}
```

-
-
-
-
-

Rust By Example

Search this book ...

Testing

Rust is a programming language that cares a lot about correctness and it includes support for writing software tests within the language itself.

Testing comes in three styles:

- Unit testing.
- Doc testing.
- Integration testing.

Also Rust has support for specifying additional dependencies for tests:

- Dev-dependencies

See Also

- The Book chapter on testing
- API Guidelines on doc-testing

Rust By Example

Search this book ...

Unit testing

Tests are Rust functions that verify that the non-test code is functioning in the expected manner. The bodies of test functions typically perform some setup, run the code we want to test, then assert whether the results are what we expect.

Most unit tests go into a `tests` mod with the `#[cfg(test)]` attribute. Test functions are marked with the `#[test]` attribute.

Tests fail when something in the test function panics. There are some helper macros:

- `assert!(expression)` - panics if expression evaluates to `false`.
- `assert_eq!(left, right)` and `assert_ne!(left, right)` - testing left and right expressions for equality and inequality respectively.

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

// This is a really bad adding function, its purpose is to fail in this
// example.
#[allow(dead_code)]
fn bad_add(a: i32, b: i32) -> i32 {
    a - b
}

#[cfg(test)]
mod tests {
    // Note this useful idiom: importing names from outer (for mod tests) scope.
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(1, 2), 3);
    }

    #[test]
    fn test_bad_add() {
        // This assert would fire and test will fail.
        // Please note, that private functions can be tested too!
        assert_eq!(bad_add(1, 2), 3);
    }
}
```

Tests can be run with `cargo test`.

```
$ cargo test

running 2 tests
test tests::test_bad_add ... FAILED
test tests::test_add ... ok

failures:

---- tests::test_bad_add stdout ----
        thread 'tests::test_bad_add' panicked at 'assertion failed: `(left == right)`
  left: `-1`,
 right: `3`', src/lib.rs:21:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

```
failures:
    tests::test_bad_add

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

Testing panics

To check functions that should panic under certain circumstances, use attribute `#[should_panic]`. This attribute accepts optional parameter `expected =` with the text of the panic message. If your function can panic in multiple ways, it helps make sure your test is testing the correct panic.

```
pub fn divide_non_zero_result(a: u32, b: u32) -> u32 {
    if b == 0 {
        panic!("Divide-by-zero error");
    } else if a < b {
        panic!("Divide result is zero");
    }
    a / b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_divide() {
        assert_eq!(divide_non_zero_result(10, 2), 5);
    }

    #[test]
    #[should_panic]
    fn test_any_panic() {
        divide_non_zero_result(1, 0);
    }

    #[test]
    #[should_panic(expected = "Divide result is zero")]
    fn test_specific_panic() {
        divide_non_zero_result(1, 10);
    }
}
```

Running these tests gives us:

```
$ cargo test

running 3 tests
test tests::test_any_panic ... ok
test tests::test_divide ... ok
test tests::test_specific_panic ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests tmp-test-should-panic

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Running specific tests

To run specific tests one may specify the test name to `cargo test` command.

```
$ cargo test test_any_panic
running 1 test
test tests::test_any_panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out

   Doc-tests tmp-test-should-panic

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

To run multiple tests one may specify part of a test name that matches all the tests that should be run.

```
$ cargo test panic
running 2 tests
test tests::test_any_panic ... ok
test tests::test_specific_panic ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out

   Doc-tests tmp-test-should-panic

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Ignoring tests

Tests can be marked with the`#[ignore]` attribute to exclude some tests. Or to run them with command `cargo test -- --ignored`

```
# #![allow(unused_variables)]
#fn main() {
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(2, 2), 4);
```

```
    }

    #[test]
    fn test_add_hundred() {
        assert_eq!(add(100, 2), 102);
        assert_eq!(add(2, 100), 102);
    }

    #[test]
    #[ignore]
    fn ignored_test() {
        assert_eq!(add(0, 0), 0);
    }
}
#}
```

```
$ cargo test
running 1 test
test tests::ignored_test ... ignored

test result: ok. 0 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out

   Doc-tests tmp-ignore

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

$ cargo test -- --ignored
running 1 test
test tests::ignored_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests tmp-ignore

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

-
-
-
-
-

Rust By Example

Search this book ...

Documentation testing

The primary way of documenting a Rust project is through annotating the source code. Documentation comments are written in markdown and support code blocks in them. Rust takes care about correctness, so these code blocks are compiled and used as tests.

````
/// First line is a short summary describing function.
///
/// The next lines present detailed documentation. Code blocks start with
/// triple backquotes and have implicit `fn main()` inside
/// and `extern crate <cratename>`. Assume we're testing `doccomments` crate:
///
/// ```
/// let result = doccomments::add(2, 3);
/// assert_eq!(result, 5);
/// ```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

/// Usually doc comments may include sections "Examples", "Panics" and "Failures".
///
/// The next function divides two numbers.
///
/// # Examples
///
/// ```
/// let result = doccomments::div(10, 2);
/// assert_eq!(result, 5);
/// ```
///
/// # Panics
///
/// The function panics if the second argument is zero.
///
/// ```rust,should_panic
/// // panics on division by zero
/// doccomments::div(10, 0);
/// ```
pub fn div(a: i32, b: i32) -> i32 {
    if b == 0 {
        panic!("Divide-by-zero error");
    }

    a / b
}
````

Tests can be run with `cargo test`:

```
$ cargo test
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests doccomments

running 3 tests
test src/lib.rs - add (line 7) ... ok
test src/lib.rs - div (line 21) ... ok
test src/lib.rs - div (line 31) ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Motivation behind documentation tests

The main purpose of documentation tests is to serve as an examples that exercise the functionality, which is one of the most important guidelines. It allows using examples from docs as complete code snippets. But using `?` makes compilation fail since `main` returns `unit`. The ability to hide some source lines from documentation comes to the rescue: one may write `fn try_main() -> Result<(), ErrorType>`, hide it and `unwrap` it in hidden `main`. Sounds complicated? Here's an example:

```
/// Using hidden `try_main` in doc tests.
///
/// ```
/// # // hidden lines start with `#` symbol, but they're still compileable!
/// # fn try_main() -> Result<(), String> { // line that wraps the body shown in doc
/// let res = try::try_div(10, 2)?;
/// # Ok(()) // returning from try_main
/// # }
/// # fn main() { // starting main that'll unwrap()
/// #    try_main().unwrap(); // calling try_main and unwrapping
/// #                         // so that test will panic in case of error
/// # }
pub fn try_div(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err(String::from("Divide-by-zero"))
    } else {
        Ok(a / b)
    }
}
```

See Also

- RFC505 on documentation style
- API Guidelines on documentation guidelines

Rust By Example

Integration testing

Unit tests are testing one module in isolation at a time: they're small and can test private code. Integration tests are external to your crate and use only its public interface in the same way any other code would. Their purpose is to test that many parts of your library work correctly together.

Cargo looks for integration tests in `tests` directory next to `src`.

File `src/lib.rs`:

```
// Assume that crate is called adder, will have to extern it in integration test.
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

File with test: `tests/integration_test.rs`:

```
// extern crate we're testing, same as any other code would do.
extern crate adder;

#[test]
fn test_add() {
    assert_eq!(adder::add(3, 2), 5);
}
```

Running tests with `cargo test` command:

```
$ cargo test
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

     Running target/debug/deps/integration_test-bcd60824f5fbfe19
```

```
running 1 test
test test_add ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Each Rust source file in `tests` directory is compiled as a separate crate. One way of sharing some code between integration tests is making module with public functions, importing and using it within tests.

File `tests/common.rs`:

```
pub fn setup() {
    // some setup code, like creating required files/directories, starting
    // servers, etc.
}
```

File with test: `tests/integration_test.rs`

```
// extern crate we're testing, same as any other code will do.
extern crate adder;

// importing common module.
mod common;

#[test]
fn test_add() {
    // using common code.
    common::setup();
    assert_eq!(adder::add(3, 2), 5);
}
```

Modules with common code follow the ordinary modules rules, so it's ok to create common module as `tests/common/mod.rs`.

-
-
-
-
-

Rust By Example

Search this book ...

Development dependencies

Sometimes there is a need to have a dependencies for tests (examples, benchmarks) only. Such dependencies are added to `Cargo.toml` in `[dev-dependencies]` section. These dependencies are not propagated to other packages which depend on this package.

One such example is using a crate that extends standard `assert!` macros.
File `Cargo.toml`:

```
# standard crate data is left out
[dev-dependencies]
pretty_assertions = "0.4.0"
```

File `src/lib.rs`:

```
// externing crate for test-only use
#[cfg(test)]
#[macro_use]
extern crate pretty_assertions;

pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(2, 3), 5);
    }
}
```

See Also

Cargo docs on specifying dependencies.

Meta

Some topics aren't exactly relevant to how you program but provide you tooling or infrastructure support which just makes things better for everyone. These topics include:

- Documentation: Generate library documentation for users via the included `rustdoc`.
- Testing: Create testsuites for libraries to give confidence that your library does exactly what it's supposed to.
- Benchmarking: Create benchmarks for functionality to be confident that they run quickly.

Rust By Example

Search this book ...

Documentation

Doc comments are very useful for big projects that require documentation. When running Rustdoc, these are the comments that get compiled into documentation. They are denoted by a `///`, and support Markdown.

```
#![crate_name = "doc"]

/// A human being is represented here
pub struct Person {
    /// A person must have a name, no matter how much Juliet may hate it
    name: String,
}

impl Person {
    /// Returns a person with the name given them
    ///
    /// # Arguments
    ///
    /// * `name` - A string slice that holds the name of the person
    ///
    /// # Example
    ///
    /// ```
    /// // You can have rust code between fences inside the comments
    /// // If you pass --test to Rustdoc, it will even test it for you!
    /// use doc::Person;
    /// let person = Person::new("name");
    /// ```
    pub fn new(name: &str) -> Person {
        Person {
            name: name.to_string(),
        }
    }

    /// Gives a friendly hello!
    ///
    /// Says "Hello, [name]" to the `Person` it is called on.
    pub fn hello(& self) {
        println!("Hello, {}!", self.name);
    }
}

fn main() {
    let john = Person::new("John");

    john.hello();
}
```

To run the tests, first build the code as a library, then tell rustdoc where to find the library so it can link it into each doctest program:

```
$ rustc doc.rs --crate-type lib
$ rustdoc --test --extern doc="libdoc.rlib" doc.rs
```

(When you run `cargo test` on a library crate, Cargo will automatically generate and run the correct rustc and rustdoc commands.)

-
-
-
-
-

Rust By Example

Search this book ...

Unsafe Operations

As an introduction to this section, to borrow from the official docs, "one should try to minimize the amount of unsafe code in a code base." With that in mind, let's get started! Unsafe blocks in Rust are used to bypass protections put in place by the compiler; specifically, there are four primary things that unsafe blocks are used for:

- dereferencing raw pointers
- calling a function over FFI (but this is covered in a previous chapter of the book)
- calling functions which are `unsafe`
- inline assembly

Raw Pointers

Raw pointers `*` and references `&T` function similarly, but references are always safe because they are guaranteed to point to valid data due to the borrow checker. Dereferencing a raw pointer can only be done through an unsafe block.

```
fn main() {
    let raw_p: *const u32 = &10;

    unsafe {
        assert!(*raw_p == 10);
    }
}
```

Calling Unsafe Functions

Some functions can be declared as `unsafe`, meaning it is the programmer's responsibility to ensure correctness instead of the compiler's. One example of this is `std::slice::from_raw_parts` which will create a slice given a pointer to the first element and a length.

```
use std::slice;

fn main() {
    let some_vector = vec![1, 2, 3, 4];

    let pointer = some_vector.as_ptr();
    let length = some_vector.len();

    unsafe {
        let my_slice: &[u32] = slice::from_raw_parts(pointer, length);

        assert_eq!(some_vector.as_slice(), my_slice);
    }
}
```

For `slice::from_raw_parts`, one of the assumptions which *must* be upheld is that the pointer passed in points to valid memory and that the memory pointed to is of the correct type. If these invariants aren't upheld then the program's behaviour is undefined and there is no knowing what will happen.