Loops

Rust currently provides three approaches to performing some kind of iterative activity. They are: `loop`, `while` and `for`. Each approach has its own set of uses.

loop

The infinite `loop` is the simplest form of loop available in Rust. Using the keyword `loop`, Rust provides a way to loop indefinitely until some terminating statement is reached. Rust's infinite `loop`s look like this:

```
loop {
    println!("Loop forever!");
}
```

while

Rust also has a `while` loop. It looks like this:

```
let mut x = 5; // mut x: i32
let mut done = false; // mut done: bool

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

`while` loops are the correct choice when you're not sure how many times you need to loop.

If you need an infinite loop, you may be tempted to write this:

```
while true {
```

However, `loop` is far better suited to handle this case:

```
loop {
```

Rust's control-flow analysis treats this construct differently than a `while true`, since we know that it will always loop. In general, the more information we can give to the compiler, the better it can do with safety and code generation, so you should always prefer `loop` when you plan to loop infinitely.

for

The `for` loop is used to loop a particular number of times. Rust's `for` loops work a bit differently than in other systems languages, however. Rust's `for` loop doesn't look like this "C-style" `for` loop:

```
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

Instead, it looks like this:

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

In slightly more abstract terms,

```
for var in expression {
    code
}
```

The expression is an item that can be converted into an iterator using `IntoIterator`. The iterator gives back a series of elements, one element per iteration of the loop. That value is then bound to the name `var`, which is valid for the loop body. Once the body is over, the next value is fetched from the iterator, and we loop another time. When there are no more values, the `for` loop is over.

In our example, `0..10` is an expression that takes a start and an end position, and gives an iterator over those values. The upper bound is exclusive, though, so our loop will print `0` through `9`, not `10`.

Rust does not have the "C-style" `for` loop on purpose. Manually controlling each element of the loop is complicated and error prone, even for experienced C developers.

Enumerate

When you need to keep track of how many times you have already looped, you can use the `.enumerate()` function.

On ranges:

```
for (index, value) in (5..10).enumerate() {
    println!("index = {} and value = {}", index, value);
}
```

Outputs:

```
index = 0 and value = 5
index = 1 and value = 6
index = 2 and value = 7
index = 3 and value = 8
index = 4 and value = 9
```

Don't forget to add the parentheses around the range.

On iterators:

```
let lines = "hello\nworld".lines();

for (linenumber, line) in lines.enumerate() {
    println!("{}: {}", linenumber, line);
}
```

Outputs:

```
0: hello
1: world
```

Ending iteration early

Let's take a look at that `while` loop we had earlier:

```
let mut x = 5;
let mut done = false;

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

We had to keep a dedicated `mut` boolean variable binding, `done`, to know when we should exit out of the loop. Rust has two keywords to help us with modifying iteration: `break` and `continue`.

In this case, we can write the loop in a better way with `break`:

```
let mut x = 5;

loop {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 { break; }
}
```

We now loop forever with `loop` and use `break` to break out early. Issuing an explicit `return` statement will also serve to terminate the loop early.

`continue` is similar, but instead of ending the loop, it goes to the next iteration. This will only print the odd numbers:

```
for x in 0..10 {
    if x % 2 == 0 { continue; }

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

Loop labels

You may also encounter situations where you have nested loops and need to specify which one your `break` or `continue` statement is for. Like most other languages, Rust's `break` or `continue` apply to the innermost loop. In a situation where you would like to `break` or `continue` for one of the outer loops, you can use labels to specify which loop the `break` or `continue` statement applies to.

In the example below, we `continue` to the next iteration of `outer` loop when `x` is even, while we `continue` to the next iteration of `inner` loop when y is even. So it will execute the `println!` when both `x` and `y` are odd.

```
'outer: for x in 0..10 {
    'inner: for y in 0..10 {
        if x % 2 == 0 { continue 'outer; } // Continues the loop over
`x`.
        if y % 2 == 0 { continue 'inner; } // Continues the loop over
`y`.
        println!("x: {}, y: {}", x, y);
    }
}
```