Variables, types, mutatbility, functions, and control flow

Variables

By default variables are immutable in Rust. Once you assign a value to it you cannot change it without explicitly marking the variable as mutable.

```
fn main() {
  let x = 5;
  x = 6; // compiler error
}
```

In order to make your variable mutable you must mark it with the `mut` keyword before the variable name like so:

```
fn main() {
let mut x = 5;
    println!("{x}");
    x = 6;
    println!("{x}");
}
```

Constants

To declare a constant variable, a variable that you cannot change, you use the `const` keyword. Constants must have its value annotated. Meaning you have to explicitly give it its data type otherwise, it will not work.

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

Shadowing

You are allowed to re-declare a variable of the same name. And the type of the variable can even be different, and this is referred to as shadowing.

```
fn main() {
    let x = 5;
    let x = x + 1; // this is allowed
}
```

The final value of the variable `x` will be `6` because the `x` it is referring to in the assignment is still referring to the old variable, before it is updated as a new variable.

Shadowing is different than marking a variable as `mut` because the data type that you can reassign the data to can be completely different. In addition, if you don't use `let` and the variable isn't marked as `mut` then you will get a compile-time error. It isn't shadowing.

Shadowing is good if you want to reuse the same variable name after performing some transformation on the original value. For example

```
let spaces = "    ";
let spaces = spaces.len(); // This will update variable spaces to be the number of spaces
```

You cannot do this the data is mutable, because you cannot change the data type of the variable if it is marked as mutable. You can only change it to the value of the same type, not to a different type.

Data types

In Rust every value has a data type, since Rust is a statically typed language so it will know how much memory to allocate for each piece of data that you use at compile time. Usually, the compiler can infer the type for simple assignments, but if there are many type that are possible, you must add a type annotation, to tell the compiler that this is the type for the data.

```
let guess: u32 = "42".parse().expect("Not a number");
```

In this case, you have to add the `unsigned 32-bit` integer annotation, otherwise, Rust will display an error since the compiler need more information about the type `guess` is.

Scalar types

Scalar type represents a single value, primitives per say. Integers, floating-point numbers, booleans, and characters are scalar type in Rust.

1. Integer

An integer is a number without fractional component. There are many variant to an integer as listed below

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u126`
arch	`isize`	`usize`

Unsigned to remind you are only positive numbers, and they have the range from:

$[0, 2^n - 1]$

Signed to remind you are numbers that can be negative, in Rust signed numbers are stored using two's complement representation so they have the range from:

$[-2^{(n - 1)}, 2^{(n - 1)} - 1]$

Arch type will depend on the architecture of your computer. If your computer is 64-bit, then arch is 64-bit, if your computer is 32-bit then arch is 32-bit.

You can write number in any base.

1. Decimal: Use `_` to serve as delimiter for a bigger number like `87_321` for 87,321
2. Hexadecimal: Prefix the hexadecimal number with `0x`
3. Octal: Prefix the octal number with `0o`
4. Binary: Prefix the binary with `0b`
5. Byte (u8 type): Prefix the byte with `b'<byte goes here>'`

Rust supporst all of the basic math operations that you would expect.

2. Floating-point types

There are two types of precision `f32` and `f64`. Default is `f64` for floating point, has more precision.

3. Boolean type

Can have the two possible value `true` and `false`. Boolean are one byte in size, and in Rust is defined using the `bool` type.

4. Character type

Character type holds four bytes of letter. You specify `char` literal with single quotes, not double quotes which denotes string literals. It is four bytes in size so it can represent lots more character than just ASCII, Chinese, Japanese, and Korean characters.

Compound types

You can group multiple values into one type, Rust have two primitive compound types, tuples and array.

Tuple type

General way fo grouping together a number of values with variety of types into one compound type.

Tuple has a fixed length, once they are declared, the size cannot grow or shrink.

You can create tuple by writing a comma-separated list of values inside parentheses. Every index in the tuple has a type and you can add the optional type annotation:

```
fn main() {
  let tup: (i32, f64, u8) = (500, 6.4, 1);
  let tup2 = (true, true, false);
}
```

To access the element inside the tuple according to their indices you would use the `.` accessor. For example:

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

You can also get the values out from the tuple by using a destructure assignment like so:

```
let tup (500, 6.4, 1);
let (x, y, z) = tup;

println!("The value of y is : {y});
```

Array type

You can declare an array by using comma-separated list inside square brackets:

```
let a = [1, 2, 3, 4, 5];
```

You can also write the array's type using square brackets with the type of each element, semicolon, and then the number of the elements in the array like so:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

You can access each array element by indexing into the indices like so:

```
let a = [1, 2, 3,];
let first = a[0];
let second = a[1];
```

Accessing an element out of bound will raise an runtime error, in the case of Rust, it will be an panic. The program terminates and exists with an error.

Functions

Use snake case for function and variable name just like in Python.

To define a function in Rust you would use the `fn` keyword followed by the function name, then the parameters that are passed into the function.

Rust doesn't care where you define the function, as long as it is in the file you can use it even if it is defined after the place you defined the function.

Parameters

If your function takes parameter then you function header must have its type annotated.

Statements and expressions

Statements are code that are ran and do not contain a return value

Expressions on the other hand evaluates to a return value

Unlike Ruby, assignment in Rust is a statement not an expression, meaning you cannot do something like:

```
let x = (let y = 6);
```

The `let` statement does not return 6, but in Ruby it does, which also make `x` equal to 6. But in Rust this will result in compilation error.

Return value

Functions can also return values, you do not name the return value, but you have to declare their type after an arrow like so

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();
    println!("The value is {x}");
}
```

In this case the function `five` returns the integer value 5.

Return value can be explicitly done by using the `return` keyword, or the return value can be implicit if the returned value is the last expression in the function, so you do not include the semicolon, if that expression is the last expression in the function that you want to return. If you include a semicolon to the expression then it will become a statement and therefore that value is not returned.

```
fn foo() -> i32 {
    return 3;
}

// Or you can do
fn foo() -> i32 {
    3   // do not include semicolon! Then it will not be returning it! and will be an error
}
```

Comments

Comments can be done via `//` there is no C style comments in Rust.

Control flow

if expressions

```
let number = 3;

if number < 5 {
println!("Condition is met");
}
else if number == 6 {
println("Condition exactly met");
}
else {
println!("Condition isn't met");
}
```

You can use if statement within a let statement to do conditional assignments

```
let condition = true;
let number = if condition { 5 } else { 6 };
```

If the condition is true, then it will assign 5 to variable `number`, if false then it will assign 6. In addition, the type that the if statement evaluate to must be the type type, which means this is wrong:

```
let condition = true;
let number = if condition { 5 } else { "six" };
```

This will result in compiler error because the of the if statement branch don't match. One is an integer the other is a string!

While loops

```
let mut number = 3;

while number != 0 {
println!("The number is {number}");
    number -= 1;
}
```

For loop

You can use a for loop to loop through each element of an array:

```
let a = [1, 2, 3, 4, 5, 6];
for num in a {
```

```
println!("Value is {num}");
}
```

Range with for loop, `Range` can be denoted using `start..end` to generate a sequence of number without including `end`:

```
for number in (1..4) {
println!("{number}");
}
```

Revision #2
Created 2023-01-14 23:44:39 UTC by Tamarine
Updated 2023-01-18 22:50:01 UTC by Tamarine