Primitive Types

The Rust language has a number of types that are considered 'primitive'. This means that they're built-in to the language. Rust is structured in such a way that the standard library also provides a number of useful types built on top of these ones, as well, but these are the most primitive.

Booleans

Rust has a built-in boolean type, named `bool`. It has two values, `true` and `false`:

```
let x = true;

let y: bool = false;
```

A common use of booleans is in `if` conditionals.

You can find more documentation for `bool`s in the standard library documentation.

char

The `char` type represents a single Unicode scalar value. You can create `char`s with a single tick: (`'`)

```
let x = 'x';
let two_hearts = '💕';
```

Unlike some other languages, this means that Rust's `char` is not a single byte, but four.

You can find more documentation for `char`s in the standard library documentation.

Numeric types

Rust has a variety of numeric types in a few categories: signed and unsigned, fixed and variable, floating-point and integer.

These types consist of two parts: the category, and the size. For example, `u16` is an unsigned type with sixteen bits of size. More bits lets you have bigger numbers.

If a number literal has nothing to cause its type to be inferred, it defaults:

```
let x = 42; // `x` has type `i32`.

let y = 1.0; // `y` has type `f64`.
```

Here's a list of the different numeric types, with links to their documentation in the standard library:

- i8
- i16
- i32
- i64
- u8
- u16
- u32
- u64
- isize
- usize
- f32
- f64

Let's go over them by category:

Signed and Unsigned

Integer types come in two varieties: signed and unsigned. To understand the difference, let's consider a number with four bits of size. A signed, four-bit number would let you store numbers from `-8` to `+7`. Signed numbers use "two's complement representation". An unsigned four bit number, since it does not need to store negatives, can store values from `0` to `+15`.

Unsigned types use a `u` for their category, and signed types use `i`. The `i` is for 'integer'. So `u8` is an eight-bit unsigned number, and `i8` is an eight-bit signed number.

Fixed-size types

Fixed-size types have a specific number of bits in their representation. Valid bit sizes are `8`, `16`, `32`, and `64`. So, `u32` is an unsigned, 32-bit integer, and `i64` is a signed, 64-bit integer.

Variable-size types

Rust also provides types whose particular size depends on the underlying machine architecture. Their range is sufficient to express the size of any collection, so these types have 'size' as the category. They come in signed and unsigned varieties which account for two types: `isize` and `usize`.

Floating-point types

Rust also has two floating point types: `f32` and `f64`. These correspond to IEEE-754 single and double precision numbers.

Arrays

Like many programming languages, Rust has list types to represent a sequence of things. The most basic is the *array*, a fixed-size list of elements of the same type. By default, arrays are immutable.

```
let a = [1, 2, 3]; // a: [i32; 3]
let mut m = [1, 2, 3]; // m: [i32; 3]
```

Arrays have type `[T; N]`. We'll talk about this `T` notation in the generics section. The `N` is a compile-time constant, for the length of the array.

There's a shorthand for initializing each element of an array to the same value. In this example, each element of `a` will be initialized to `0`:

```
let a = [0; 20]; // a: [i32; 20]
```

You can get the number of elements in an array `a` with `a.len()`:

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

println!("a has {} elements", a.len());
```

You can access a particular element of an array with *subscript notation*:

```
let names = ["Graydon", "Brian", "Niko"]; // names: [&str; 3]

println!("The second name is: {}", names[1]);
```

Subscripts start at zero, like in most programming languages, so the first name is `names[0]` and the second name is `names[1]`. The above example prints `The second name is: Brian`. If you try to use a subscript that is not in the array, you will get an error: array access is bounds-checked at run-time. Such errant access is the source of many bugs in other systems programming languages.

You can find more documentation for `array`s in the standard library documentation.

Slices

A 'slice' is a reference to (or "view" into) another data structure. They are useful for allowing safe, efficient access to a portion of an array without copying. For example, you might want to reference only one line of a file read into memory. By nature, a slice is not created directly, but from an existing variable binding. Slices have a defined length, and can be mutable or immutable.

Internally, slices are represented as a pointer to the beginning of the data and a length.

Slicing syntax

You can use a combo of `&` and `[]` to create a slice from various things. The `&` indicates that slices are similar to references, which we will cover in detail later in this section. The `[]` s, with a range, let you define the length of the slice:

```
let a = [0, 1, 2, 3, 4];
let complete = &a[..]; // A slice containing all of the elements in `a`.
let middle = &a[1..4]; // A slice of `a`: only the elements `1`, `2`,
and `3`.
```

Slices have type `&[T]`. We'll talk about that `T` when we cover generics.

You can find more documentation for slices in the standard library documentation.

`str`

Rust's `str` type is the most primitive string type. As an unsized type, it's not very useful by itself, but becomes useful when placed behind a reference, like `&str`. We'll elaborate further when we cover Strings and references.

You can find more documentation for `str` in the standard library documentation.

Tuples

A tuple is an ordered list of fixed size. Like this:

```
let x = (1, "hello");
```

The parentheses and commas form this two-length tuple. Here's the same code, but with the type annotated:

```
let x: (i32, &str) = (1, "hello");
```

As you can see, the type of a tuple looks like the tuple, but with each position having a type name rather than the value. Careful readers will also note that tuples are heterogeneous: we have an `i32` and a `&str` in this tuple. In systems programming languages, strings are a bit more complex than in other languages. For now, read `&str` as a *string slice*, and we'll learn more soon.

You can assign one tuple into another, if they have the same contained types and

arity. Tuples have the same arity when they have the same length.

```
let mut x = (1, 2); // x: (i32, i32)
let y = (2, 3); // y: (i32, i32)

x = y;
```

You can access the fields in a tuple through a *destructuring let*. Here's an example:

```
let (x, y, z) = (1, 2, 3);

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

Remember before when I said the left-hand side of a `let` statement was more powerful than assigning a binding? Here we are. We can put a pattern on the left-hand side of the `let`, and if it matches up to the right-hand side, we can assign multiple bindings at once. In this case, `let` "destructures" or "breaks up" the tuple, and assigns the bits to three bindings.

This pattern is very powerful, and we'll see it repeated more later.

You can disambiguate a single-element tuple from a value in parentheses with a comma:

```
(0,); // A single-element tuple.
(0); // A zero in parentheses.
```

Tuple Indexing

You can also access fields of a tuple with indexing syntax:

```
let tuple = (1, 2, 3);

let x = tuple.0;
let y = tuple.1;
let z = tuple.2;

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

Like array indexing, it starts at zero, but unlike array indexing, it uses a `.`, rather than `[]`s.

You can find more documentation for tuples in the standard library documentation.

Functions

Functions also have a type! They look like this:

```
fn foo(x: i32) -> i32 { x }

let x: fn(i32) -> i32 = foo;
```

In this case, `x` is a 'function pointer' to a function that takes an `i32` and returns an `i32`.