Data Types

Every value in Rust is of a certain *data type*, which tells Rust what kind of data is being specified so it knows how to work with that data. We'll look at two data type subsets: scalar and compound.

Keep in mind that Rust is a *statically typed* language, which means that it must know the types of all variables at compile time. The compiler can usually infer what type we want to use based on the value and how we use it. In cases when many types are possible, such as when we converted a `String` to a numeric type using `parse` in the "Comparing the Guess to the Secret Number" section in Chapter 2, we must add a type annotation, like this:

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

If we don't add the `: u32` type annotation shown in the preceding code, Rust will display the following error, which means the compiler needs more information from us to know which type we want to use:

```
$ cargo build
   Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^
  |
help: consider giving `guess` an explicit type
  |
2 |     let guess: _ = "42".parse().expect("Not a number!");
  |              +++

For more information about this error, try `rustc --explain E0282`.
error: could not compile `no_type_annotations` due to previous error
```

You'll see different type annotations for other data types.

Scalar Types

A *scalar* type represents a single value. Rust has four primary scalar types: integers, floating-point numbers, Booleans, and characters. You may recognize these from other programming languages. Let's jump into how they work in Rust.

Integer Types

An *integer* is a number without a fractional component. We used one integer type in Chapter 2, the `u32` type. This type declaration indicates that the value it's associated with should be an unsigned integer (signed integer types start with `i` instead of `u`) that takes up 32 bits of space. Table 3-1 shows the built-in integer types in Rust. We can use any of these variants to declare the type of an integer value.

Table 3-1: Integer Types in Rust

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

Each variant can be either signed or unsigned and has an explicit size. *Signed* and *unsigned* refer to whether it's possible for the number to be negative—in other words, whether the number needs to have a sign with it (signed) or whether it will only ever be positive and can therefore be represented without a sign (unsigned). It's like writing numbers on paper: when the sign matters, a number is shown with a plus sign or a minus sign; however, when it's safe to assume the number is positive, it's shown with no sign. Signed numbers are stored using two's complement representation.

Each signed variant can store numbers from $-(2^{n-1})$ to $2^{n-1} - 1$ inclusive, where *n* is the number of bits that variant uses. So an `i8` can store numbers from $-(2^7)$ to $2^7 - 1$, which equals -128 to 127. Unsigned variants can store numbers from 0 to $2^n - 1$, so a `u8` can store numbers from 0 to $2^8 - 1$, which equals 0 to 255.

Additionally, the `isize` and `usize` types depend on the architecture of the computer your program is running on, which is denoted in the table as "arch": 64 bits if you're on a 64-bit architecture and 32 bits if you're on a 32-bit architecture.

You can write integer literals in any of the forms shown in Table 3-2. Note that number literals that can be multiple numeric types allow a type suffix, such as `57u8`, to designate the type. Number literals can also use `_` as a visual separator to make the number easier to read, such as `1_000`, which will have the same value as if you had specified `1000`.

Table 3-2: Integer Literals in Rust

Number literals	Example
Decimal	`98_222`
Hex	`0xff`
Octal	`0o77`
Binary	`0b1111_0000`
Byte (`u8` only)	`b'A'`

So how do you know which type of integer to use? If you're unsure, Rust's defaults are generally good places to start: integer types default to `i32`. The primary situation in which you'd use `isize` or `usize` is when indexing some sort of collection.