1_Data types_en

May 2, 2023

1 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**.

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.

2 Scalar Types

Rust has four primary scalar types: integers, floating-point numbers, Booleans, and characters.

2.1 Integer types

Signed integer types start with **i**, unsigned integers start with **u**. The number of bits is explicitly stated by the data type (u32, i64), except for the **isize** and **usize** types which depend on the computer architecture (64 bits on a 64-bit architecture and 32 bits on a 32-bits architecture). If you're unsure of the type of integer to use, Rust's defaults are generally good choices, and integer types default to i32: this type is generally the fastest, even on 64-bit systems. The compiler can usually infer what type we want to use based on the value and how we use it.

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

[ ]:
```
let guess = 5; // defaults to i32

let y: u32 = 5; // explicit type declaration using type annotation
```

In cases where many types are possible, such as when we convert a String to a numeric type, type annotation is mandatory.

[ ]:
```
fn main() {
    let guess: u32 = "42".parse().expect("Not a number!");
    println!("My number is {}", guess);
}
main();
```

TODO: try to remove the type assertion on `let guess: u32` in the code above and re-run it to see what it does.

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

    // Integer subtraction
    // TO DO:
    let x = 1i32 - 2; //short syntax: value followed by a type
    println!("1 - 2 = {}", x);

    // Syntax with underscore to improve readability
    println!("One million is written as {}", 1_000_000u32);
}
main();
```

TODO: Try changing `1i32` to `1u32` to see why the type is important

2.2 Floating-point types

Rust's floating-point types are f32 and f64, which are 32 bits and 64 bits in size, respectively. The default type is f64 because on modern CPUs it's roughly the same speed as f32 but is capable of more precision.

[ ]:
```
let x = 2.0; // f64

let y: f32 = 3.0; // f32
```

2.3 Numeric operators

[ ]:
```
// addition
let sum = 5 + 10;

// subtraction
let difference = 95.5 - 4.3;

// multiplication
let product = 4 * 30;

// division
let quotient = 56.7 / 32.2;
```

```
// remainder
let remainder = 43 % 5;
```

2.4 Logical and bitwise operators

[ ]:
```
// Logical operators
    println!("true AND false is {}", true && false);
    println!("true OR false is {}", true || false);
    println!("NOT true is {}", !true);

// Bitwise operators
    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);
```

2.5 Boolean types

[ ]:
```
let t = true;

let f: bool = false; // with explicit type annotation
```

2.6 The Character Type

The most primitive alphabetic type, char literals are specified with single quotes, as opposed to string literals, which use double quotes. Rust's char type is four bytes in size and represents a Unicode Scalar Value, which means it can represent a lot more than just ASCII. Accented letters; Chinese, Japanese, and Korean characters; emoji; and zero-width spaces are all valid char values in Rust.

[ ]:
```
let c = 'z';
let z = ' ';
let heart_eyed_cat = ' ';
```

2.7 String literals

A string literal is a sequence of any Unicode characters enclosed within two U+0022 (double-quote) characters, with the exception of U+0022 itself, which must be escaped by a preceding U+005C character ( ).

> Note that internally, Rust `String`s are always represented as UTF-8 encoded strings and are always valid. It is NOT possible to construct a Rust `String` that contains an invalid UTF-8 sequence.

Line-break characters are allowed in string literals. Normally they represent themselves (i.e. no translation), but as a special exception, when an unescaped U+005C character ( ) occurs immediately before the newline (U+000A), the U+005C character, the newline, and all whitespace at the beginning of the next line are ignored. Thus a and b are equal:

[ ]:
```
let a = "foobar";
let b = "foo\
         bar";
```

2.8 Type conversions

To convert from one type to another, here is a useful reference: http://carols10cents.github.io/rust-conversion-reference/

- Use `as` to coerce primitive types.

[ ]:
```
let x = 6.5; // implicit f64
let y = x as u32; // coerce the f64 to u32
println!("{}", y);
let z: String = x.to_string(); // convert to String representation
println!("{}", z);
```

2.9 Constants

[ ]:
```
const N: i32 = 5;
```

Unlike let bindings, you must annotate the type of a const. The standard library defines some useful constants:

[ ]:
```
const MAX: i32 = std::i32::MAX;
const MIN: i32 = std::i32::MIN;
```

[ ]:
```
println!("The i32 max value is: {}", std::i32::MAX);
println!("The u32 max value is: {}", std::u32::MAX);
```

3 Compound Types

3.1 The Tuple Type

A tuple is a general way of grouping together some number of other values with a variety of types into one compound type. Tuples have a *fixed length*: once declared, they cannot grow or shrink in size.

We create a tuple by writing a comma-separated list of values inside parentheses. Each position in the tuple has a type, and the types of the different values in the tuple don't have to be the same.

[ ]:
```
let tup: (i32, f64, u8) = (500, 6.4, 1);
```

To get the individual values out of a tuple, we can use *pattern matching* to destructure a tuple value or we can *access a tuple element directly* by using a period (.) followed by the index of the value we want to access.

[ ]:
```
let tup = (500, 6.4, 1); // type inference can be used

let (x, y, z) = tup; // this is tuple destructuring

println!("The value of y is: {}", y);
println!("The tuple is: {:?}", tup); // the :? printing option will be␣
 ↪explained later on
```

[ ]:
```
let x: (i32, f64, u8) = (500, 6.4, 1);

let five_hundred = x.0; // access to tuple members

let six_point_four = x.1;

let one = x.2;
```

3.2 The Array Type

Unlike a tuple, every element of an array must have the same type. Arrays in Rust are different from arrays in some other languages because *arrays in Rust have a fixed length*, like tuples. Arrays are useful when you want your data allocated on the stack rather than the heap.

[ ]:
```
let a: [i32; 5] = [1, 2, 3, 4, 5];
let first = a[0];
let second = a[1];
println!("First element in the array: {}", first);
println!("Second element in the array: {}", second);
```

Invalid array element access

[ ]:
```
let a = [1, 2, 3, 4, 5];
let index = (|| 10)(); // we will see this syntax later
// TODO: try with `let index = 10;` what do you get?

let element = a[index];

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

The compilation didn't produce any errors, but the program resulted in a runtime error and didn't exit successfully. When you attempt to access an element using indexing, Rust will check that the index you've specified is less than the array length. If the index is greater than or equal to the array length, Rust will panic.

This is the first example of Rust's safety principles in action. In many low-level languages, this kind of check is not done, and when you provide an incorrect index, invalid memory can be accessed. Rust protects you against this kind of error by immediately exiting instead of allowing the memory access and continuing.

4 Exercises

Exercise 1: Use the correct data structure.

Sample test data:

Matrix	
1.1	1.2
2.1	2.2

[ ]:
```
fn main() {
    // TO DO (1) Define a list of months, or days of the week

    // TO DO (2) Define a constant for the value epsilon = 1e-11

    // TO DO (3) Define a 2 X 2 matrix of real numbers and display the␣
 ↪transposed matrix

}
main();
```

Solution:

[ ]:
```
fn main() {
    // TO DO (1)
    let months: [&str; 12] = ["January", "February", "March", "April", "May",␣
 ↪"June", "July",
                "August", "September", "October", "November", "December"];
    let days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday",␣
 ↪"Saturday", "Sunday"];
    // TO DO (2)
    const EPSILON: f64 = 1e-11_f64;
    // TO DO (3)
    let matrix: ((f32,f32),(f32,f32)) = ((1.1, 1.2), (2.1, 2.2));
    println!("{} {}", (matrix.0).0, (matrix.1).0);
    println!("{} {}", (matrix.0).1, (matrix.1).1);

}
main();
```