PLC2024 Lecture 09, 08 Feb 2024

Rust

- Rust resources: https://www.rust-lang.org/
- Installing Rust: https://www.rust-lang.org/tools/install
- Documentation: https://www.rust-lang.org/learn

Typing

- Static (Java, Haskell) vs dynamic (Python)
  - Ideally, type errors should be caught at compile-time (static)
  - Dynamic --- type is determined by current value, type of a variable can change over time
- Implicit (Haskell, Python) vs explicit (Java declarations)
  - Implicit + static $\implies$ type inference
- Degrees of strictness
  - Is mixed mode arithmetic allowed? e.g., `x = 1.5 + 3`
  - Can numbers be intepreted as booleans? `if len(l) { ... }`
- Rust types
  - Static
  - Mostly implicit, but *must* declare types for function signatures
  - **Very** strict!

Rust program

- **Not** object oriented
- Program is a collection of functions
- Execution begins with `main()`
- Read documentation about how to compile
- `cargo` to build Rust projects

Hello world!

In [2]:
```
fn main(){
    println!("Hello world");
}
```

In [3]:
```
main()
```

```
Hello world
```

Out[3]:
```
()
```

- `!` after `println` signifies it is a macro, not a function --- will worry about this later
- This function returns nothing, so return value is `()`

Variables

- Declare variables using `let` and assign a value
- Value implicitly fixes type

In [4]:
```
fn var1(){
    let x = 55;
    println!("Value of x is {x}"); // Insert value in string, Version 1
}
```

In [5]:
```
var1()
```

```
Value of x is 55
```

Out[5]:
```
()
```

In [6]:
```
fn var2(){
    let x = 55;
    println!("Value of x is {}",x); // Insert value in string, Version 2
}
```

In [7]:
```
var2()
```

```
Value of x is 55
```

Out[7]:
```
()
```

- What if we try to update the value of `x`?

In [8]:
```
fn var3(){
    let x = 55;
    x = 66;
    println!("Value of x is {}",x); // Insert value in string, Version 2
}
```

```
[unused_assignments] Error: value assigned to `x` is never read
   ┌─[command_8:1:1]
   │
 2 │     let x = 55;
   │         ┬
   │         └── warning: value assigned to `x` is never read
───┘

[E0384] Error: cannot assign twice to immutable variable `x`
   ┌─[command_8:1:1]
   │
 2 │     let x = 55;
   │         ┬
   │         └── first assignment to `x`
   │         │
   │         └── help: consider making this binding mutable: `mut x`
 3 │     x = 66;
   │     ─┬─
   │      └──── cannot assign twice to immutable variable
   │
   │ Note: You can change an existing variable to mutable like: `let mut x = x;`
───┘
```

- Rust variables are **immutable** by default
  - Like variables in mathematics
  - Let $x = 4$ ... means x is an arbitrary but fixed value
- Need to add a qualifier `mut` to say that a variable is mutable
  - Notice the useful error message, suggesting that we add the qualifier `mut`

In [9]:
```
fn var4(){
    let mut x = 55;
    x = 66;
```

```
    println!("Value of x is {}",x); // Insert value in string, Version 2
}
```

In [10]:
```
var4()
```

```
Value of x is 66
```

Out[10]:
```
()
```

Constants

- Immutable variables are not the same as constants
- Declare constants explicitly
  - So far we have used implicit typing
  - Constants need to be typed explicitly -- Rust uses older Algol/Pascal style `var: type` notation for typing rather than C/Java style `type var`
  - Will describe Rust types shortly
- Constants can have *global* scope, declared outside all functions

In [11]:
```
const PI_APPROX: f32 = 3.1415927;
fn const1(){
    println!("Value of pi is approximately {}",PI_APPROX);
}
```

In [12]:
```
const1()
```

```
Value of pi is approximately 3.1415927
```

Out[12]:
```
()
```

Shadowing

- Redeclaring a variable *shadows* the earlier definition
- Can change the type with each fresh declaration (but why?)

In [13]:
```
let x = 0.0;
let x = 5;
println!("value of x is {}",x);
```

```
value of x is 5
```

- But cannot change the type of a mutable variable

In [14]:
```
let mut x = 0.0;
x = 5;
println!("value of x is {}",x);
```

```
[E0308] Error: mismatched types
   ┌─[command_14:1:1]
   │
 1 │ let mut x = 0.0;
   │             ─┬─
   │              └── expected due to this value
 2 │ x = 5;
   │     ┬
   │     └── expected floating-point number, found integer
───┘
```

Scalar types

- Signed integers: `i8`, `i32`, `i64`, `isize` -- explicitly specify number of bits, last version uses the underlying architecture default
- Unsigned integers: `u8`, `u32`, `u64`, `usize`
- Floats: `f32`, `f64`
- Boolean: `bool` --- values are `true` and `false`
- Charactre: `char` --- write with single quote, `'a'`, uses UTF-8, upto 4 bytes per character

- Implicit vs explicit typing
  - Normally Rust deduces type from value assigned in `let`
  - Can also explicitly annotate type

In [15]:
```
let y: f32 = 5.0;
println!("Value of y is {}",y);
```

```
Value of y is 5
```

- Strict typing
  - Cannot have mixed int/float expressions --- use `as type` to "recast" a type
  - Arithmetic expressions cannot replace boolean expressions -- convention that `0` is `false`, non-zero is `true` etc does not work

In [16]:
```
let mut x = 5.8;
x = x * 7;
println!("Value of x is {}",x);
```

```
[E0277] Error: cannot multiply `{float}` by `{integer}`
   ┌─[command_16:1:1]
   │
 2 │ x = x * 7;
   │       ┬
   │       └── no implementation for `{float} * {integer}`
───┘
```

In [17]:
```
let mut x = 5.8;
x = x * 7 as f32;
println!("Value of x is {}",x);
```

```
Value of x is 40.600002
```

Defining functions

- Functions are defined using `fn`
- Need to provide explicit types for arguments and return value
- Notation for return value uses `->` like Haskell

In [18]:
```
fn addtwo(x : i32, y: i32) -> i32 {
    return x + y;
}
```

In [19]:
```
let a = addtwo(17,42);
println!("Value of a is {}",a);
```

```
Value of a is 59
```

Expressions

- Functions implicitly return last *expression* evaluated
- Can rewrite our function as below

In [20]:
```
fn addtwoexpr(x : i32, y: i32) -> i32 {
    x + y
}
```

In [21]:
```
let a = addtwoexpr(17,42);
println!("Value of a is {}",a);
```

```
Value of a is 59
```

- An expression should not have a semicolon at the end
- Semicolon turns the expression into a *statement*
  - Note again the helpful compiler error message

In [22]:
```
fn addtwosemicolon(x : i32, y: i32) -> i32 {
    x + y;
}
```

```
[E0308] Error: mismatched types
   ┌─[command_22:1:1]
   │
 1 │ fn addtwosemicolon(x : i32, y: i32) -> i32 {
   │    ──────┬──────                       ─┬─
   │          └──────────────────────────────── implicitly returns `()` as its body
has no tail or `return` expression
   │                                          │
   │                                          └── expected `i32`, found `()`
 2 │     x + y;
   │          ┬
   │          └── help: remove this semicolon to return this value: ``
───┘
```

Control flow

- `if` boolean-expression `{ ... } else {....}`
- loops: `while` boolean-expression `{...}`, `loop {...}`, `for`
- `loop` requires a `break`, else infinite
- `for` runs over elements from an iterator --- later

In [23]:
```
fn signum1(x: i32) -> i32{
    if x < 0 {return -1;}
    else if x == 0 {return 0;}
    else {1}
}
```

In [24]:
```
signum1(-7)
```

Out[24]:
```
-1
```

- `if` is itself an expression, so can do a conditional assignment

In [25]:
```
fn signum2(y: i32) -> i32{
    let x = if y < 0 {-1} else if y == 0 {0} else {1};
    return x;
}
```

In [26]:
```
signum2(0)
```

Out[26]:
```
0
```

- This cryptic `if` expression suffices

In [27]:
```
fn signum3(y: i32) -> i32{
    if y < 0 {-1} else if y == 0 {0} else {1}
}
```

In [28]:
```
signum3(77)
```

Out[28]:
```
1
```

Copying values

- `x = y` for values stored on the stack copies the value
- `x = y` for values stored on the heap creates an *alias* -- both `x` and `y` refer to the same value on the heap
  - Useful to avoid copying large values, and to pass heap objects to a function
  - However, leads to subtle errors because updating `y` indirectly updates `x`
  - Also, releasing memory through `y` results in a *dangling pointer* at `x`
- Rust introduces a concept called **ownership** to address these issues