Concepts of programming languages

Rust

Floris Schild, Mats Veldhuizen, Ruben Schenkuizen, Tobias van Driessel, Tom Freijsen

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Memory management

Different systems

- Garbage collection

Memory management

Different systems

- Garbage collection
- Smart pointers (reference counting)

Memory management

Different systems

- Garbage collection
- Smart pointers (reference counting)
- Ownership

Memory management

Different systems

- Garbage collection
- Smart pointers (reference counting)
- Ownership

Key difference: When are objects on the heap deallocated?

Ownership

- Every value has one variable that is its owner.

Ownership

- Every value has one variable that is its owner.

Example

```
let x = String::from("hello");
```

The variable *x* is the owner of the value 5.

The string object will be deallocated when *x* goes out of scope.

Ownership

Another example

```
fn say_hello(name: String) {
    println!("Hello {}", name);
}

let x = String::from("Wouter");
say_hello(x);
say_hello(x); // Error!
```

Ownership

Another example

```
fn say_hello(name: String) {
    println!("Hello {}", name);
}

let x = String::from("Wouter");
say_hello(x);
say_hello(x); // Error!
```

The ownership is passed into *say_hello*, which deallocates our string when its scope ends.

Borrowing

- Pass a reference to the function
- Ownership is not transferred
- Immutable (by default)

```
fn say_hello(name: &String) {
    println!("Hello {}", name);
}

let x = String::from("Wouter");
say_hello(&x);
say_hello(&x); // Works!
```

Mutable references

```
fn double(value: &mut isize) {
    *value = *value * 2;
}

let mut x: isize = 3;
double(&mut x);
// x = 6
```

Slices

- Taking a reference to part of a collection

```
let some_numbers = [0, 1, 2, 3, 4, 5];
let slice = &some_numbers[0..3]; // [0, 1, 2]
```

Smart pointers

- Sometimes you may want more freedom
- For these situations, reference counting is also possible in Rust

```
let a = Rc::new(String::from("Blue"));
```

Smart pointers

- Box
- Enables recursive data types

Smart pointers

```
enum List {
    Cons(isize, Box<List>),
    Nil
}

let list = List::Cons(1,
                Box::new(List::Cons(2,
                Box::new(List::Cons(3,
                Box::new(List::Nil))))));

// list = (1 : (2 : (3 : [])))
```

Reference counting using smart pointers

```
struct Node {
    value: isize,
    next: Vec<Rc<Node>>
}

let d = Rc::new(Node { value: 8, next: vec![] });
let b = Rc::new(Node { value: 3, next: vec![Rc::clone(&d)] });
let c = Rc::new(Node { value: 5, next: vec![Rc::clone(&d)] });
let a = Rc::new(Node { value: 2, next: vec![Rc::clone(&b),
                                            Rc::clone(&c)] });
```