More Syntax and Borrowing

`struct`, `enum`, `impl`, `match`, and the Borrow Checker

Cooper Pierce & Jack Duvall

Carnegie Mellon University

Outline

structs

Like many other languages, Rust supports structs.
We can have traditional, C-style structs:

```
struct Student {
    andrewid: [u8; 8],
    name: String,
    section: char,
}
```

structs

Like many other languages, Rust supports structs.
We can have traditional, C-style structs:

```
struct Student {
    andrewid: [u8; 8],
    name: String,
    section: char,
}
```

or named tuple style structs:

```
struct Fraction(u32, u32);
```

structs

Like many other languages, Rust supports structs.
We can have traditional, C-style structs:

```rust
struct Student {
    andrewid: [u8; 8],
    name: String,
    section: char,
}
```

or named tuple style structs:

```rust
struct Fraction(u32, u32);
```

or unit structs:

```rust
struct Refl;
```

Every field of a struct must be assigned a value when initialising it.

```rust
let jack = Student {
    andrewid: [b'j', b'r', b'd', b'u', b'v', b'a', b'l', b'l'],
    name: String::from("Jack Duvall"),
    section: 'A',
};
```

Every field of a struct must be assigned a value when initialising it.

```
let jack = Student {
    andrewid: [b'j', b'r', b'd', b'u', b'v', b'a', b'l', b'l'],
    name: String::from("Jack Duvall"),
    section: 'A',
};
```

If there are local variables with the same name, we can shortcut this somewhat:

```
// Dereference because this gives a reference to the array.
let andrewid = *b"cppierce";
let name = String::from("Cooper Pierce");
let section = 'A';
let cooper = Student { andrewid, name, section };
```

Member access for structs is similar to C, with the exception of eliminating `->`. A period `.` is used for both acessing through reference and direct access.

```
assert_ne!(cooper.andrewid, jack.andrewid);

let s = &cooper;
assert_eq!(cooper.name, s.name);
```

Member access for structs is similar to C, with the exception of eliminating `->`. A period `.` is used for both acessing through reference and direct access.

```
assert_ne!(cooper.andrewid, jack.andrewid);

let s = &cooper;
assert_eq!(cooper.name, s.name);
```

Fields of named-tuple structs are accessed the same as tuples.

```
let f = Fraction(3, 10);
fn get_denominator(f: Fraction) -> u32 { f.1 }
```

Member access for structs is similar to C, with the exception of eliminating ->. A period . is used for both acessing through reference and direct access.

```
assert_ne!(cooper.andrewid, jack.andrewid);

let s = &cooper;
assert_eq!(cooper.name, s.name);
```

Fields of named-tuple structs are accessed the same as tuples.

```
let f = Fraction(3, 10);
fn get_denominator(f: Fraction) -> u32 { f.1 }
```

Unit structs behave exactly like the unnamed unit ():

```
let x: Refl = Refl;
```

enums

Rust also has enums. Both C-style "named constants" like

```rust
enum Weekday {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday
}
```

enums

Rust also has enums. Both C-style "named constants" like

```
enum Weekday {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday
}
```

which are kept in their own namespace (like C++ `enum class`es):

```
let today = Weekday::Wednesday;
```

And also more functionally-inspiried ones with data:

```
enum Number {
    Rational { numer: u32, denom: u32, sign: bool }
    Float(f64),
    Int(i32),
    Infinity,
}
```

And also more functionally-inspiried ones with data:

```
enum Number {
    Rational { numer: u32, denom: u32, sign: bool }
    Float(f64),
    Int(i32),
    Infinity,
}
```

Which we can use similarily:

```
let f = Number::Float(1.6);
let r = Number::Rational { numer: 3, denom: 8, sign: true };
```

And also more functionally-inspiried ones with data:

```
enum Number {
    Rational { numer: u32, denom: u32, sign: bool }
    Float(f64),
    Int(i32),
    Infinity,
}
```

Which we can use similarily:

```
let f = Number::Float(1.6);
let r = Number::Rational { numer: 3, denom: 8, sign: true };
```

What would an enum for `sign` look like?

Outline

`if` expressions

Similar to functional programming languages, `if` does not introduce a statement, but instead an expression.

`if` expressions

Similar to functional programming languages, `if` does not introduce a statement, but instead an expression.

So while we can do

```
let x;
if some_condition {
    x = 7;
} else {
    x = 9
}
```

You'd typically see

```
let x = if some_condition { 7 } else { 9 };
```

If we omit the else branch the if branch must evaluate to unit—()

```
if is_admin(user) {
    println!("Hello administrator!");
}
```

If we omit the else branch the if branch must evaluate to unit—`()`

```
if is_admin(user) {
    println!("Hello administrator!");
}
```

Note that any expression followed by a semicolon will be an expression which discards the result and evaluates to unit.

`while` loops

We have the typical while loop:

```
fn exp(mut n: i32) -> i32 {
    let mut b = 2;
    let mut x = 1;
    while n > 1 {
        if n % 2 == 1 {
            x = x * b;
        }
        b *= b;
        n /= 2;
    }
    x * b
}
```

`for` loops

and iterator-based for loops:

```
let nums = [1, 2, 3, 4, 5];
for n in nums {
    println!("{}", n);
}
```

for loops

and iterator-based for loops:

```
let nums = [1, 2, 3, 4, 5];
for n in nums {
    println!("{}", n);
}
```

Range types are often useful here:

```
for i in 0..n {
    println("{} squared is {}", i, i * i);
}
```

`loop` loops

In addition, we also have an unconditional loop construct:

```
loop {
    println!("Hi again!");
}
```

loop loops

In addition, we also have an unconditional loop construct:

```
loop {
    println!("Hi again!");
}
```

This is more useful when using `break`

```
let prime = loop {
    let p = gen_random_number();
    if miller_rabin(p) {
        break p;
    }
};
```

Outline

We can add associated functions and methods to a struct or enum we've defined by using an `impl` block.

```rust
struct Rectangle {
    width: u32,
    height: u32,
}
```

```rust
impl Rectangle {
    fn unit() -> Self {
        Self { width: 1, height: 1 }
    }

    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

Invoking an associated function is done by qualifying it with the type

```rust
let unit_square = Rectangle::unit();
```

Invoking an associated function is done by qualifying it with the type

```rust
let unit_square = Rectangle::unit();
```

and methods are typically invoked using a dot:

```rust
let r = Rectangle { width: 4, height: 7 };
assert_eq!(unit_square.area(), 1);
assert_eq!(r.area(), 28);
```

Outline

match expressions

What if we want to deal with many possible branching choices for an expression?

```rust
fn fib(n: u32) -> u32 {
    match n {
        0 | 1 => 0,
        n => fib(n - 1) + fib(n - 2),
    }
}
```

This is a bit more useful when dealing with enums

```
enum Coin { Penny, Nickel, Dime, Quarter }

impl Coin {
    fn value(&self) -> u32 {
        match self {
            Coin::Penny   =>  1,
            Coin::Nickel  =>  5,
            Coin::Dime    => 10,
            Coin::Quarter => 25,
        }
    }
}
```

Most of all when the enum has data

```
enum Transmission {
    Incoming(String)
    Done,
}

fn listen(&mut p: Port) {
    loop {
        match p.receive() {
            Transmission::Incoming(s) => {
                println!(s);
            }
            Done => return,
        }
    }
}
```

Sometimes we can employ more specific pattern matching constructs to simplfy code.

Sometimes we can employ more specific pattern matching constructs to simplfy code.

```
enum Transmission {
    Incoming(String)
    Done,
}

fn listen(&mut p: Port) {
    while let Transmission::Incoming(s) = p.receive() {
        println!(s);
    }
}
```

Likewise, there's also `if let`. However, you'll essentially always want to use `match` if you have two or more things to do.

Outline

Recall: Stack and Heap

- Regions of memory you can store data in
- Stack:
- Heap:

Recall: Stack and Heap

- Regions of memory you can store data in
- Stack:
  - Local to current function invocation
  - Data ideally has known size at compile time (or a reasonable upper bound)
  - Automatically (logically) freed when function exits
- Heap:

Recall: Stack and Heap

- Regions of memory you can store data in
- Stack:
  - Local to current function invocation
  - Data ideally has known size at compile time (or a reasonable upper bound)
  - Automatically (logically) freed when function exits
- Heap:
  - Persistent across function calls; not thread-local
  - Data can have unknown size
  - Some level of explicit memory management (gc, `malloc`/`free`, refcounting, dtors, etc..)

Definitions

- Value: The actual representation of some object
- Variable: A name corresponding to that representation

```
// The variable x has a value of 98008
let x = 98008;
```

More Definitions

- Scope: A region of code where a variable is valid
- Dropping: The process of running a value's destructor

More Definitions

- Scope: A region of code where a variable is valid
- Dropping: The process of running a value's destructor
  - think: popping stack frame or calling `free()`

Ownership Rules

- Each value in Rust has a single variable called its owner.
- There can only be one owner at a time.
- When the owner exits its scope, the value will be dropped.
- See also https://doc.rust-lang.org/stable/book/ch04-01-what-is-ownership.html

Upcoming Ownership Examples

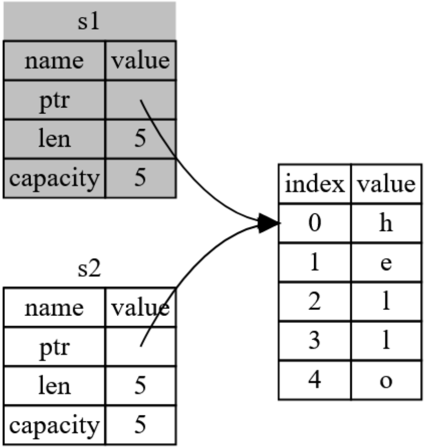
- Simple Move
- Move Into Function
- Move Out of Function
- Cloning

Ownership Example: Simple Move

```rust
let x = 5;
let y = x; // `x` can be copied efficiently, so the data is just
// copied into `y`
println!("{}", x); // This is OK

let s1 = String::from("hello");
let s2 = s1; // `s2` now "owns" the data that `s1` used to refer to
println!("{}", s1); // So this is an error
```

Ownership Example: Simple Move



Ownership Example: Move Into Function

```
fn makes_copy(x: i32)        { println!("{}", x); }

fn take_ownership(x: String) { println!("{}", x); }

fn main() {
    let x = 5;
    makes_copy(x);
    println!("{}", x);

    let y = String::from("hello");
    take_ownership(y);
    println!("{}", y); // !
}
```

Ownership: Cloning

```rust
let s1 = String::from("hello");
let s2 = s1.clone(); // different and distinct from s1
```

Ownership: Cloning

- What if you have data that can't be automatically copied, but you still want a copy?

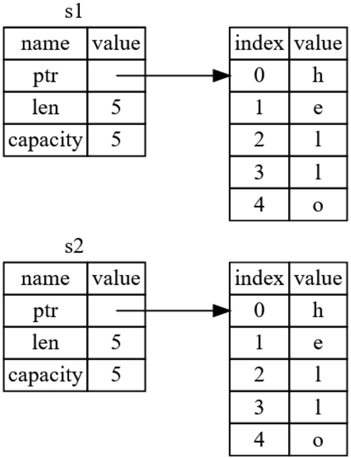
```rust
let s1 = String::from("hello");
let s2 = s1.clone(); // different and distinct from s1
```

Ownership: Cloning

- What if you have data that can't be automatically copied, but you still want a copy?
- Solution: `.clone()` the data!

```
let s1 = String::from("hello");
let s2 = s1.clone(); // different and distinct from s1
```

Ownership: Cloning: Diagram



When Can I Copy Or Clone?

- Copy: whenever a type implements the `Copy` trait!
- Clone: whenever a type implements the `Clone` trait!
- We'll get into traits more next lecture
- Important: the programmer implementing the struct decides if (and for `Clone`, how) these operations are allowed
  - Restriction on `Copy`: every field/variant must be `Copy`
  - If something is `Copy`, it must also be `Clone`

Outline

Reference Pitfalls

In many other languages with references (e.g., C++) there are a number of potential pitfalls:

```
int main() {
    auto v = std::vector<int>{1, 2, 3, 4};
    auto x = &v[1];
    v.push_back(5);
    *x = 0;
    std::cout << v[1] << std::endl;
    return 0;
}
```

What's wrong?

Reference Pitfalls

In many other languages with references (e.g., C++) there are a number of potential pitfalls:

```
int main() {
    auto v = std::vector<int>{1, 2, 3, 4};
    auto x = &v[1];
    v.push_back(5);
    *x = 0;
    std::cout << v[1] << std::endl;
    return 0;
}
```

What's wrong?

By changing `v`, we invalidate the reference `x`!

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an "owner".

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an “owner”.
- There can only be one owner.

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an "owner".
- There can only be one owner.
- When ownership of the value ends, the value will be "dropped" (think deallocated/destructed).

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an "owner".
- There can only be one owner.
- When ownership of the value ends, the value will be "dropped" (think deallocated/destructed).
- You can have as many shared borrows (`&`) as you want, all at the same time ...

Rules for Borrowing

In Rust, this *cannot happen*, because borrowing has restrictions:

- Every value has an "owner".
- There can only be one owner.
- When ownership of the value ends, the value will be "dropped" (think deallocated/destructed).
- You can have as many shared borrows (`&`) as you want, all at the same time ...
- ... but, you can only have one exclusive borrow (`&mut`), and not at the same time as any shared borrow.

References: Pointers But Better

References: Pointers But Better

- Reference: "You don't own this value, but you can still access it"

References: Pointers But Better

- Reference: “You don’t own this value, but you can still access it”
    - Value is called ”borrowed”

References: Pointers But Better

- Reference: “You don’t own this value, but you can still access it”
  - Value is called ”borrowed”
- Two types: Immutable and Mutable (more accurately: “shared” and “exclusive”)

References: Pointers But Better

- Reference: “You don’t own this value, but you can still access it”
  - Value is called ”borrowed”
- Two types: Immutable and Mutable (more accurately: “shared” and “exclusive”)
- Guarantee: it’s always valid to access memory through a reference!

References: Pointers But Better

- Reference: “You don’t own this value, but you can still access it”
  - Value is called ”borrowed”
- Two types: Immutable and Mutable (more accurately: “shared” and “exclusive”)
- Guarantee: it’s always valid to access memory through a reference!
  - Not the case with pointers

Immutable References

```
&Ty
```

- Only let you read
- Any number can exist at one point, so long as there's no mutable references to the object at the same time.

Immutable References: Example

```
let x: i32 = 5;
let x_ref: &i32 = &x;

// Ok to have more than one immutable ref
let x_ref2: &i32 = &x;

// Immutable reference is Copy
let x_ref3: &i32 = x_ref;

// Ok, i32 is Copy---can "move out of" reference to one
let y: i32 = *x_ref;
```

Mutable References

```
&mut Ty
```

- Let you read and write
- Can only be made if the underlying object is also mutable
- Only one can exist at a time

Mutable References: Example

```
let x: i32 = 5;

// Error: x isn't mut
let x_mut_ref: &mut i32 = &mut x;

let mut y: i32 = 6;
let y_mut_ref: &mut i32 = &mut y;

// Error: y_mut_ref
let y_mut_ref2: &mut i32 = &mut y;

// Error: mut ref isn't Copy
let y_mut_ref3: &mut i32 = y_mut_ref;
*y_mut_ref += 2;
```

Tomorrow

- Function types
- Closures
- More advanced ownership semantics