Rust programming

Module 2: Foundations of Rust

Unit 3

Advanced Syntax

Learning objectives

Composite types

Types redux

We have previously looked at some of the basic types in the Rust typesystem

- Primitives (integers, floats, booleans, characters)
- Compounds (tuples, arrays)
- Most of the types we looked at were `Copy`
- Borrowing will make more sense when we look at some more ways we can type our data

Structuring data

Rust has two important ways to structure data

- structs
- enums
- ~~unions~~

Structs

A struct is similar to a tuple, but this time the combined type gets its own name

```rust
struct ControlPoint(f64, f64, bool);
```

This is an example of a *tuple struct*. You can access the fields in the struct the same way as with tuples:

```rust
fn main() {
  let cp = ControlPoint(10.5, 12.3, true);
  println!("{}", cp.0); // prints 10.5
}
```

Structs

Much more common though are structs with named fields

```rust
struct ControlPoint {
  x: f64,
  y: f64,
  enabled: bool,
}
```

- We can add a little more purpose to each field
- No need to keep our indexing up to date when we add or remove a field

```rust
fn main() {
  let cp = ControlPoint {
    x: 10.5,
    y: 12.3,
    enabled: true,
  };
  println!("{}", cp.x); // prints 10.5
}
```

Enumerations

One of the more powerful kinds of types in Rust are enumerations

```rust
enum IpAddressType {
  Ipv4,
  Ipv6,
}
```

- An enumeration (listing) of different *variants*
- Each variant is an alternative value of the enum, you pick a single value to create an instance
- Each variant has a discriminant (hidden by default)
  - a numeric value (`` `isize` `` by default, can be changed by using `` `#[repr(numeric_type)]` ``) used to determine the variant that the enumeration holds
  - one cannot rely on the fact that the discriminant is an `` `isize` ``, the compiler may always decide to optimize it

```rust
fn main() {
  let ip_type = IpAddressType::Ipv4;
}
```

Enumerations

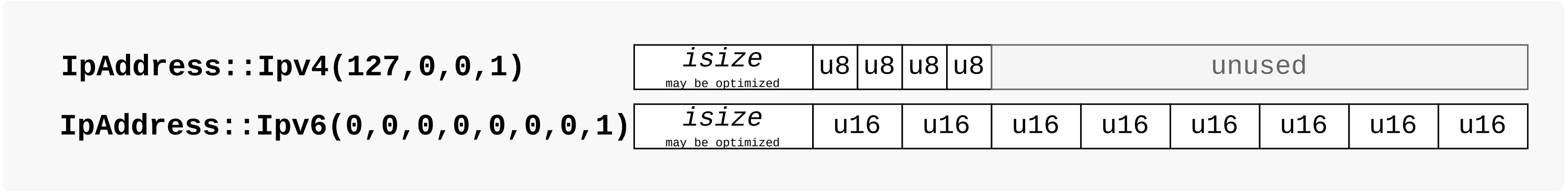
Enums get more powerful, because each variant can have associated data with it

```rust
enum IpAddress {
  Ipv4(u8, u8, u8, u8),                            // = 0 (default discriminant)
  Ipv6(u16, u16, u16, u16, u16, u16, u16, u16),    // = 1 (default discriminant)
}
```

- This way, the associated data and the variant are bound together
- Impossible to create an ipv6 address while only giving a 32 bits integer

```rust
fn main() {
  let ipv4_home = IpAddress::Ipv4(127, 0, 0, 1);
  let ipv6_home = IpAddress::Ipv6(0, 0, 0, 0, 0, 0, 0, 1);
}
```

- An enum always is as large as the largest variant plus the size of the discriminant


IpAddress::Ipv4(127,0,0,1)
isize
may be optimized
u8 u8 u8 u8
unused
IpAddress::Ipv6(0,0,0,0,0,0,0,1)
isize
may be optimized
u16 u16 u16 u16 u16 u16 u16 u16

Pattern matching

Extracting data from `enum`

- We must ensure we interpret `enum` data correctly
- Use pattern matching to do so

Pattern matching

Using the `if let [pattern] = [value]` statement

```rust
fn accept_ipv4(ip: IpAddress) {
  if let IpAddress::Ipv4(a, b, _, _) = ip {
    println!("Accepted, first octet is {} and second is {}", a, b);
  }
}
```

- `a` and `b` introduce local variables within the body of the if that contain the values of those fields
- The underscore (`_`) can be used to accept any value

Match

Pattern matching is very powerful if combined with the match statement

```rust
fn accept_home(ip: IpAddress) {
  match ip {
    IpAddress::Ipv4(127, 0, 0, 1) => {
      println!("You are home!");
    },
    IpAddress::Ipv6(0, 0, 0, 0, 0, 0, 0, 1) => {
      println!("You are in your new home!");
    },
    _ => {
      println!("You are not home");
    },
  }
}
```

- Every part of the match is called an arm
- A match is exhaustive, meaning all possible values must be handled by one of the match arms
- You can use a catch-all `_` arm to catch any remaining cases if there are any left

Match as an expression

The match statement can even be used as an expression

```rust
fn get_first_byte(ip: IpAddress) {
  let first_byte = match ip {
    IpAddress::Ipv4(a, _, _, _) => a,
    IpAddress::Ipv6(a, _, _, _, _, _, _, _) => a / 256 as u8,
  };
  println!("The first byte was: {}", first_byte);
}
```

- The match arms can return a value, but their types have to match
- Note how here we do not need a catch all (`` `_ =>` ``) arm because all cases have already been handled by the two arms

`impl` blocks

`impl` blocks

To associate functions to `structs` and `enums`, we use `impl` blocks

```rust
fn main() {
  let x = Some(42);
  let unwrapped = x.unwrap();
  println!("{}", unwrapped);
}
```

- The syntax `x.y()` looks similar to how we accessed a field in a struct
- We can define functions on our types using impl blocks
- Impl blocks can be defined on any type, not just structs (with some limitations)

`impl` blocks

```rust
enum IpAddress {
  Ipv4(u8, u8, u8, u8),
  Ipv6(u16, u16, u16, u16, u16, u16, u16, u16),
}

impl IpAddress {
  fn as_u32(&self) -> Option<u32> {
    match self {
      IpAddress::Ipv4(a, b, c, d) => a << 24 + b << 16 + c << 8 + d
      _ => None,_
    }
  }
}

fn main() {
  let addr = IpAddress::Ipv4(127, 0, 0, 1);
  println!("{:?}", addr.as_u32());
}
```

`self` and `Self`

- The `self` parameter defines how the method can be used.
- The `Self` type is a shorthand for the type on which the current implementation is specified.

```rust
struct Foo(i32);

impl Foo {
  fn consume(self) -> Self {            // Takes `Foo` by value, returns `Foo`
    Self(self.0 + 1)
  }

  fn borrow(&self) -> &i32 {            // Takes immutable reference of `Foo`
    &self.0
  }

  fn borrow_mut(&mut self) -> &mut i32 { // Takes mutable reference of `Foo`
    &mut self.0
  }

  fn new() -> Self {                     // Associated function, returns `Foo`
    Self(0)
  }
}
```

`impl` blocks, the `self` parameter

The self parameter is called the *receiver*.

- The `self` parameter is always the first and it always has the type on which it was defined
- We never specify the type of the `self` parameter
- We can optionally prepend `&` or `&mut ` to `self` to indicate that we take a value by reference
- Absence of a `self` parameter means that the function is an associated function instead

```rust
fn main () {
  let mut f = Foo::new();
  println!("{}", f.borrow());
  *f.borrow_mut() = 10;
  let g = f.consume();
  println!("{}", g.borrow());
}
```

Optionals and Error handling

Generics

Structs become even more powerful if we introduce a little of generics

```rust
struct PointFloat(f64, f64);
struct PointInt(i64, i64);
```

We are repeating ourselves here, what if we could write a data structure for both of these cases?

```rust
struct Point<T>(T, T);

fn main() {
  let float_point: Point<f64> = Point(10.0, 10.0);
  let int_point: Point<i64> = Point(10, 10);
}
```

Generics are much more powerful, but this is all we need for now

Option

A quick look into the basic enums available in the standard library

- Rust does not have null, but you can still define variables that optionally do not have a value
- For this you can use the `` `Option<T>` `` enum

```rust
enum Option<T> {
  Some(T),
  None,
}

fn main() {
  let some_int = Option::Some(42);
  let no_string: Option<String> = Option::None;
}
```

Option

A quick look into the basic enums available in the standard library

- Rust does not have null, but you can still define variables that optionally do not have a value
- For this you can use the `` `Option<T>` `` enum

```rust
enum Option<T> {
  Some(T),
  None,
}

fn main() {
  let some_int = Some(42);
  let no_string: Option<String> = None;
}
```

Error handling

What would we do when there is an error?

```rust
fn divide(x: i64, y: i64) -> i64 {
  if y == 0 {
    // what to do now?
  } else {
    x / y
  }
}
```

Error handling

What would we do when there is an error?

```rust
fn divide(x: i64, y: i64) -> i64 {
  if y == 0 {
    panic!("Cannot divide by zero");
  } else {
    x / y
  }
}
```

- A panic in Rust is the most basic way to handle errors
- A panic error is an all or nothing kind of error
- A panic will immediately stop running the current thread/program and instead immediately work to shut it down, using one of two methods:
  - Unwinding: going up through the stack and making sure that each value is cleaned up
  - Aborting: ignore everything and immediately exit the thread/program
- Only use panic in small programs if normal error handling would also exit the program
- Avoid using panic in library code or other reusable components

Error handling

What would we do when there is an error? We could try and use the option enum instead of panicking

```rust
fn divide(x: i64, y: i64) -> Option<i64> {
  if y == 0 {
    None
  } else {
    Some(x / y)
  }
}
```

Result

Another really powerful enum is the result, which is even more useful if we think about error handling

```rust
enum Result<T, E> {
  Ok(T),
  Err(E),
}

enum DivideError {
  DivisionByZero,
  CannotDivideOne,
}

fn divide(x: i64, y: i64) -> Result<i64, DivideError> {
  if x == 1 {
    Err(DivideError::CannotDivideOne)
  } else if y == 0 {
    Err(DivideError::DivisionByZero)
  } else {
    Ok(x / y)
  }
}
```

Handling results

Now that we have a function that returns a result we have to think about how we handle that error at the call-site

```rust
fn div_zero_fails() {
  match divide(10, 0) {
    Ok(div) => println!("{}", div),
    Err(e) => panic!("Could not divide by zero"),
  }
}
```

- We made the signature of the `` `divide` `` function explicit in how it can fail
- The user of the function can now decide what to do, even if it is panicking
- Note: just as with `` `Option` `` we never have to use `` `Result::Ok` `` and `` `Result::Err` `` because they have been made available globally

Handling results

Especially when writing initial prototyping code you will often find yourself wanting to write error handling code later, Rust has a useful utility function to help you for both `Option` and `Result`:

```rust
fn div_zero_fails() {
  let div = divide(10, 0).unwrap();
  println!("{}", div);
}
```

- Unwrap checks if the Result/Option is `Ok(x)` or `Some(x)` respectively and then return that `x`, otherwise it will panic your program with an error message
- Having unwraps all over the place is generally considered a bad practice
- Sometimes you can ensure that an error won't occur, in such cases `unwrap` can be a good solution

Handling results

Especially when writing initial prototyping code you will often find yourself wanting to write error handling code later, Rust has a useful utility function to help you for both `` `Option` `` and `` `Result` ``:

```rust
fn div_zero_fails() {
  let div = divide(10, 0).unwrap_or(-1);
  println!("{}", div);
}
```

Besides unwrap, there are some other useful utility functions

- `` `unwrap_or(val)` ``: If there is an error, use the value given to unwrap_or instead
- `` `unwrap_or_default()` ``: Use the default value for that type if there is an error
- `` `expect(msg)` ``: Same as unwrap, but instead pass a custom error message
- `` `unwrap_or_else(fn)` ``: Same as unwrap_or, but instead call a function that generates a value in case of an error

Result and the `?` operator

Results are so common that there is a special operator associated with them, the `?` operator

```
fn can_fail() -> Result<i64, DivideError> {
  let intermediate_result = match divide(10, 0) {
    Ok(ir) => ir,
    Err(e) => return Err(e),
  };

  match divide(intermediate_result, 0) {
    Ok(sec) => Ok(sec * 2),
    Err(e) => Err(e),
  }
}
```

Look how this function changes if we use the `?` operator

```
fn can_fail() -> Result<i64, DivideError> {
  let intermediate_result = divide(10, 0)?;
  Ok(divide(intermediate_result, 0)? * 2)
}
```

Result and the `?` operator

```rust
fn can_fail() -> Result<i64, DivideError> {
  let intermediate_result = divide(10, 0)?;
  Ok(divide(intermediate_result, 0)? * 2)
}
```

- The `?` operator does an implicit match, if there is an error, that error is then immediately returned and the function returns early
- If the result is `Ok()` then the value is extracted and we can continue right away

`Vec`

`Vec`: storing more of the same

The vector is an array that can grow

- Compare this to the array we previously saw, which has a fixed size

```rust
fn main() {
  let arr = [1, 2];
  println!("{:?}", arr);

  let mut nums = Vec::new();
  nums.push(1);
  nums.push(2);
  println!("{:?}", nums);
}
```

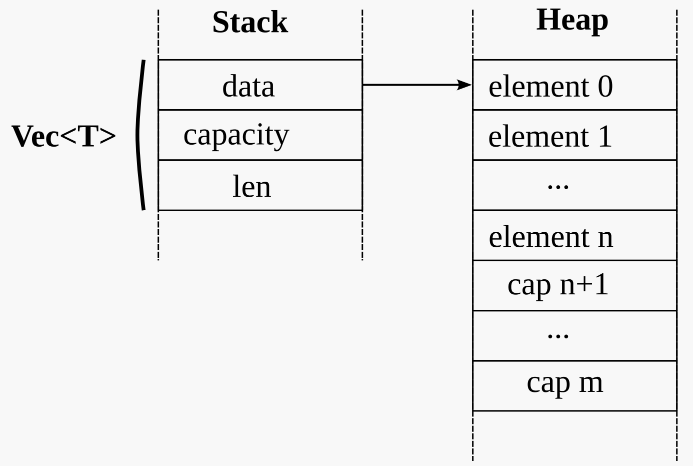
`Vec`

`Vec` is such a common type that there is an easy way to initialize it with values that looks similar to arrays

```rust
fn main() {
  let mut nums = vec![1, 2];
  nums.push(3);
  println!("{:?}", nums);
}
```

`Vec`: memory layout

How can a vector grow? Things on the stack need to be of a fixed size



Slices

Vectors and arrays

What if we wanted to write a sum function, we could define one for arrays of a specific size:

```rust
fn sum(data: &[i64; 10]) -> i64 {
  let mut total = 0;
  for val in data {
    total += val;
  }
  total
}
```

Vectors and arrays

Or one for just vectors:

```rust
fn sum(data: &Vec<i64>) -> i64 {
  let mut total = 0;
  for val in data {
    total += val;
  }
  total
}
```

Slices

What if we want something to work on arrays of any size? Or what if we want to support summing up only parts of a vector?

- A slice is a dynamically sized view into a contiguous sequence
- Contiguous: elements are layed out in memory such that they are evenly spaced
- Dynamically sized: the size of the slice is not stored in the type, but is determined at runtime
- View: a slice is never an owned data structure
- Slices are typed as `` `[T]` ``, where `` `T` `` is the type of the elements in the slice

Slices

```rust
fn sum(data: [i64]) -> i64 {
  let mut total = 0;
  for val in data {
    total += val;
  }
  total
}

fn main() {
  let data = vec![10, 11, 12, 13, 14];
  println!("{}", sum(data));
}
```

```
   Compiling playground v0.0.1 (/playground)
error[E0277]: the size for values of type `[i64]` cannot be known at compilation time
 --> src/main.rs:1:8
  |
1 | fn sum(data: [i64]) -> i64 {
  |        ^^^^ doesn't have a size known at compile-time
  |
  = help: the trait `Sized` is not implemented for `[i64]`
help: function arguments must have a statically known size, borrowed types always have a known size
```

Slices

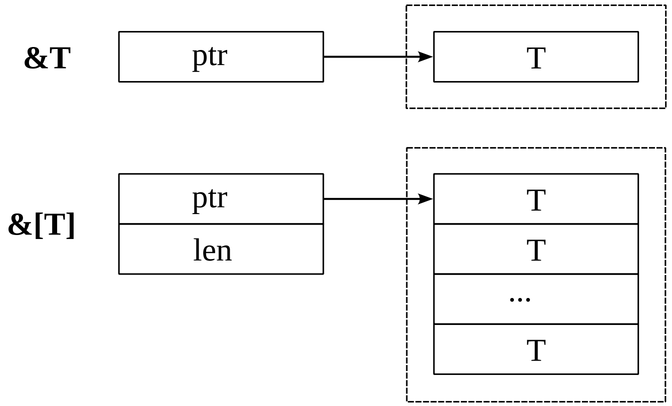
```rust
fn sum(data: &[i64]) -> i64 {
  let mut total = 0;
  for val in data {
    total += val;
  }
  total
}

fn main() {
  let data = vec![10, 11, 12, 13, 14];
  println!("{}", sum(&data));
}
```

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.89s
     Running `target/debug/playground`
60
```

Slices

- `[T]` is an incomplete type: we need to know how many `T` there are
- Types that have a known compile time size implement the `Sized` trait, raw slices do **not** implement it
- Slices must always be behind a reference type, i.e. `&[T]` and `&mut [T]` (but also `Box<[T]>` etc)
- The length of the slice is always stored together with the reference



Creating slices

Because we cannot create slices out of thin air, they have to be located somewhere. There are three possible ways to create slices:

- Using a borrow
  - We can borrow from arrays and vectors to create a slice of their entire contents
- Using ranges
  - We can use ranges to create a slice from parts of a vector or array
- Using a literal (for immutable slices only)
  - We can have memory statically available from our compiled binary

Creating slices

Using a borrow

```rust
fn sum(data: &[i32]) -> i32 { /* ... */ }

fn main() {
  let v = vec![1, 2, 3, 4, 5, 6];
  let total = sum(&v);
  println!("{}", total);
}
```

Creating slices

Using ranges

```rust
fn sum(data: &[i32]) -> i32 { /* ... */ }

fn main() {
  let v = vec![0, 1, 2, 3, 4, 5, 6];
  let all = sum(&v[..]);
  let except_first = sum(&v[1..]);
  let except_last = sum(&v[..5]);
  let except_ends = sum(&v[1..5]);
}
```

- The range `` `start..end` `` contains all values `` `x` `` with `` `start <= x < end` ``.
- Note: you can also use ranges on their own, for example in a for loop:

```rust
fn main() {
  for i in 0..10 {
    println!("{}", i);
  }
}
```

Creating slices

From a literal

```rust
fn sum(data: &[i32]) -> i32 { todo!("Sum all items in `data`") }

fn get_v_arr() -> &'static [i32] {
    &[0, 1, 2, 3, 4, 5, 6]
}

fn main() {
  let all = sum(get_v_arr());
}
```

- Interestingly `get_v_arr` works, even though the literal looks like it would only exist temporarily
- Literals actually exist during the entire lifetime of the program
- `&'static` here is used to indicate that this slice will exist the entire lifetime of the program

Strings

We have already seen the `` `String` `` type being used before, but let’s dive a little deeper

- Strings are used to represent text
- In Rust they are always valid UTF-8
- Their data is stored on the heap
- A String is almost the same as `` `Vec<u8>` `` with extra checks to prevent creating invalid text

Strings

Let’s take a look at some strings

```rust
fn main() {
  let s = String::from("Hello world\nSee you!");
  println!("{:?}", s.split_once(" "));
  println!("{}", s.len());
  println!("{:?}", s.starts_with("Hello"));
  println!("{}", s.to_uppercase());
  for line in s.lines() {
    println!("{}", line);
  }
}
```

String literals

We have already seen string literals being used while constructing a string. The string literal is what arrays are to vectors

```rust
fn main() {
  let s1 = "Hello world";
  let s2 = String::from("Hello world");
}
```

String literals

We have already seen string literals being used while constructing a string. The string literal is what arrays are to vectors

```rust
fn main() {
  let s1: &'static str = "Hello world";
  let s2: String = String::from("Hello world");
}
```

- `s1` is actually a slice, a string slice

String literals

We have already seen string literals being used while constructing a string. The string literal is what arrays are to vectors

```rust
fn main() {
  let s1: &str = "Hello world";
  let s2: String = String::from("Hello world");
}
```

- `s1` is actually a slice, a string slice

`str` - the string slice

It should be possible to have a reference to part of a string. But what is it?

- Not `[u8]`: not every sequence of bytes is valid UTF-8
- Not `[char]`: we could not create a slice from a string since it is stored as UTF-8 encoded bytes
- We introduce a new special kind of slice: `str`
- For string slices we do not use brackets!

`` `str` ``, `` `String` ``, `` `[T; N]` ``, `` `Vec` ``

Static	Dynamic	Borrowed
`` `[T; N]` ``	`` `Vec<T>` ``	`` `&[T]` ``
-	`` `String` ``	`` `&str` ``

- There is no static variant of str
- This would only be useful if we wanted strings of an exact length
- But just like we had the static slice literals, we can use `` `&'static str` `` literals for that instead!

`String` or `str`

When do we use `String` and when do we use `str`?

```rust
fn string_len(data: &String) -> usize {
  data.len()
}
```

`String` or `str`

When do we use String and when do we use str?

```rust
fn string_len(data: &str) -> usize {
  data.len()
}
```

- Prefer `&str` over `String` whenever possible
- If you need to mutate a string you might try `&mut str`, but you cannot change a slice's length
- Use `String` or `&mut String` if you need to fully mutate the string

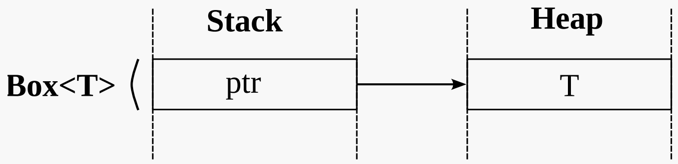
Smart pointers

Put it in a `Box`

That pointer from the stack to the heap, how do we create such a thing?

- Boxing something is the way to store a value on the heap
- A `Box` uniquely owns that value, there is no one else that also owns that same value
- Even if the type inside the box is `Copy`, the box itself is not, move semantics apply to a box.

```rust
fn main() {
  // put an integer on the heap
  let boxed_int = Box::new(10);
}
```



Boxing

There are several reasons to box a variable on the heap

- When something is too large to move around
- We need something that is sized dynamically
- For writing recursive data structures

```rust
struct Node {
  data: Vec<u8>,
  parent: Box<Node>,
}
```

To Do

Issue: tweedegolf/teach-rs#68

Interior mutability

To do:

Issue: tweedegolf/teach-rs#67

Summary