Rust Crash Course

Agenda

- Basic syntax
- The borrow checker
- Practice problems
- Convenience types

- Slides from Rohan Kumar, Rahul Kumar, and Edward Zeng
  - Used in the course CS 162: Operating Systems and Systems Programming, Prof. John Kubiatowicz at Berkely University, USA

Why Rust

- Memory safe!
- Fast executables
- Fewer runtime bugs

Defining variables

- The compiler will try to guess the type

```
let x: i32 = 2;
let x = 2;
```

- Variables are **immutable** by default

```
// This won‘t compile
let x = 2;
x = 2;
```

```
// This is OK
let mut x = 2;
x = 2;
```

References

- Rust **references** are like pointers in C, but are:
  - Always valid (no pointers to freed memory)
  - Never null
- The Rust compiler checks these properties at **compile time**
- Reference with &, dereference with *

```rust
let x = 2;
let ptr = &x;
assert_eq!(*ptr, 2);
```

Mutable References

- By default, references are **immutable**.
- This won‘t work:

```
let mut x = 2;
let ptr = &x;
*ptr = 3; // Cannot mutate through an immutable reference
```

- If you want to modify a variable through a pointer, you must have a **mutable reference**

```
let mut x = 2;
let ptr = &mut x;
*ptr = 3; // This is OK
```

Mutable References

- To obtain a mutable reference, the variable itself must be mutable:

```
let x = 2;
let ptr = &mut x; // Cannot mutably reference an immutable variable
```

- Obtaining a reference in Rust is called **borrowing**.

Primitive types

- Signed integers: `i8, i16, i32, i64, i128, isize`
- Unsigned integers: `u8, u16, u32, u64, u128, usize`
- Floating point: `f32, f64`
- Boolean: `boolean`
- Character: `char` (use single quotes, e.g. `'a'`)

Compound types

Heinrich Heine
Universität Düsseldorf

- Tuples:

```
// The type annotation is unnecessary
let my_tuple: (i32, char, bool) = (162, 'X', true);
```

- Arrays:

```
// Arrays have a fixed size, as indicated in the (optional) type annotation
let nums: [i32, 5] = [1, 2, 3, 4, 5];
let second = nums[1];
```

- Structs:

```
struct Coordinate {
   x: i32,
   y: i32,
}

let point = Coordinate { x: 5, y: 3, };
```

Functions

- A function that squares the input:

```
fn square(x: i32) -> i32 {
   return x * x;
}
```

- Equivalent to:

```
fn square(x: i32) -> i32 {
   x * x
}
```

- If a function does not return a value, just omit the return type:

```
fn add_one(x: &mut i32) {
   *x += 1;
}
```

Hello World!

- A function that squares the input:

```
fn main() {
   println!(“Hello world!“);
}
```

- `println!` is a **macro**. The latter always have the excalamtion mark as suffix.
- The compiler expands macros during compilation and macros will be replaced by „regular“ Rust source code.

If statements

- Rust has conditional if statements. No parentheses for the boolean expressions!

```
let x = 4;
if x > 5 {
   println!("Greater than 5");
} else if x > 3 {
   println!("Greater than 3");
} else {
   println!("x was not big enough");
}
```

If statements

- All expressions can evalute to a value.

```
let x = 4;
let message = if x > 5 {
   "Greater than 5"
} else if x > 3 {
   "Greater than 3"
} else {
   "x was not big enough"
};
println!("{}", message)
```

Loops

```
loop {
   println!("stuck in a loop");
}
```

- Use break to exit

```
loop {
   break;
}
```

- Loop expressions can evalute to a value, just like any other expression:

```
let mut count = 0;
let three = loop {
   count += 1;
   if count >= 3 {
      break count;
   }
};
```

While loops

- Rust `while` loops are fairly striaghtforward:

```
let mut count = 5;
while count > 0 {
    count -= 1;
}
```

For loops

- For loop can iterate over a collection

```
let muts = [0, 1, 2, 3, 4];
for num in nums {
   println!("{}", num);
}
// Prints 0 1 2 3 4
```

- Range notion:

```
for num in 0..5 {
   println!("{}", num);
}
// Prints 0 1 2 3 4
```

Enums

- Useful when a type should have only a few possible values

```
enum Coin {
   Head,
   Tail,
}

let a = Coin::Head;
let b = Coin::Tail;
```

- Each value in an enum is called a **variant**.

Enums

- Enums can also store data!

```rust
enum OperatingSystem {
   Mac,
   Windows,
   Linux,
   Other(String),
}

let a = OperatingSystem::Linux;
let b = OperatingSystem::Other("Redox OS");
```

Matching

- Rust **match expressions** are like C `switch` statements. However, they must *always be exhaustive*.
- What is wrong here?

```
let num = 162;

match num {
   160 => println!(160);
   161 => println!(161);
   168 => println!(168);
}
```

- The match is not exhaustive! What if `num` was 164 or 10?

Matching

- This **is** valid:

```
let num = 162;

match num {
   160 => println!(160);
   161 => println!(161);
   168 => println!(168);
     _ => println!(“another case“);
}
```

- The underscore matches anything that was not already matched.
- Each pattern in the `match` statement is called a **match arm**.

Matching

- Matching is very useful in combination with enums. Match expressions can also evaluate to a value (just like any other expression).

```rust
enum OperatingSystem {
   Mac,
   Windows,
   Linux,
   Other(String),
}

fn os_name(os: OperatingSystem) -> String {
   match os {
      Mac        => "mac".to_string(),
      Windows    => "windows".to_string(),
      Linux      => "linuxc".to_string(),
      Other(s) => s,
   }
}
```

Impl blocks

- Suppose we have the following struct definition:

```
struct Vector {
    x: f64,
    y: f64,
}
```

- We might want to add 2 Vectors elementwise. Here is one way to do that:

```
fn add(v1: Vector, v2: Vector) -> Vector {
    x: v1.x + v2.x,
    y: v1.y + v2.y,
}
```

`impl` blocks

- We can also do the same thing using `impl` blocks, wich define methods related to a struct (or enum).

```rust
impl Vector {
   fn add(&self, other: Vector) -> Self {
      x: self.x + other.x,
      y: self.y + other.y,
   }
}

let sum = v1.add(&v2);

// Can also be called like this
let sum = Vector::add(&v1, &v2);
```

- `Self` is a shorthand for the type of the `impl` block (in this case, `Vector`).

Arguments to `impl` block functions

Heinrich Heine Universität Düsseldorf

- First argument is `&self`: the compiler will immutably borrow the object

```
let v1 = ...;
v1.add(&v2)
```

- Is approximately equivalent to

```
let v1 = ...;
let reference = &v1;
Vector::add(reference, &v2);
```

Arguments to `impl` block functions

- Now first argument is `&mut self`: the compiler will mutably borrow the object

```
impl Vector {
    fn double(&mut self) {
        self.x *= 2;
        self.y *= 2;
    }
}
```

```
let mut v1 = ...;
v1.double();
```

- Is approximately equivalent to

```
let mut v1 = ...;
let reference = &mut v1;
Vector::double(reference);
```

The Borrow Checker

Heinrich Heine
Universität Düsseldorf

- A very important concept
- At the beginning sometimes difficult

Why have a borrow checker?

- We are trying to solve the problem of when to allocate and deallocate memory

- In C, you have to do this manually using `malloc` and `free`.
- In Java, a garbage collector runs periodically to free objects that are no longer usable
- In Rust, the borrow checker automatically determines when a value is unusable, and inserts code to free it at that point

- Automatic memory management without the overhead of garbage collection!
(Sometimes also called compile-time garbage collection)

Borrow checker

- The **borrow checker** is what makes Rust *very* different from other languages

- The borrow checker verifies a set of rules at compile time. It does the magic of making sure your references are always valid.

- The borrow checker rules can initially seem mysterious.
  But they become easier with practice.

- We'll incrementally build up some of the basic borrow checking rules.

Basic rules

Heinrich Heine Universität Düsseldorf

- Here's one set of borrow checking rules:

- Every value has one and only one owner
- When a value's owner goes out of **scope**, the value is **dropped** (= freed)
- To manually drop a value v *early*, call `drop(v)`

- Is it possible to drop a value late?
- No. A value is dropped when it goes out of scope. You can modify your code so that the scope is larger, but you cannot call `drop(v)` if v is not in scope.

Scopes

- **Scopes** are enclosed in curly braces:

```rust
fn do_stuff() {
   let a = String::from("hello");
   {
    let b = String::from("goodbye");
    // Can access a and b
    // ...
    // b goes out of scope and is dropped
  }
  // Cannot access b here: it is out of scope
  // a is dropped at the end of the function
}
```

- Functions, loops, if statements etc. have their own scope. You can also create nested scopes using curly braces.

Moves

- Every value has one owner. Sometimes that owner can change. This is called a **move**
- Assignment moves values.
- This is invalid:

```
let s1 = String::from("my string");
let s2 = s1; // Ownership of the string moves from s1 to s2.

// s1 no longer owns the string, so we can't access data via s1.
println!("{}", s1); // This is an error; data has been moved out of s1.
```

- This is fine:

```
let s1 = String::from("my string");
let s2 = s1; // Ownership of the string moves from s1 to s2.
println!("{}", s2); // This is okay; s2 owns the string now.
```

Cloning

Heinrich Heine Universität Düsseldorf

- If you need multiple variables to own data, you can `clone` a value:

```
let s1 = String::from("my string");
let s2 = s1.clone(); // s2 is a clone of s1
println!("{}", s2); // This is okay; s2 owns its data.
println!("{}", s1); // This is okay; s1 also owns its data.
```

- Note that cloning is usually *expensive*. In this case, we are allocating memory for a string *twice*.

- In the previous examples, we only allocated space for the string once.

- Cloned values are completely independent of the value they were cloned from. If `s1` is modified, code using `s2` will not see those changes.

Copy

- There is one case in which a move behaves like a clone: when the type is Copy.
- Copy is a **trait** (ie. interface) that indicates that a value is copied whenever it is used.
- For example, integers are `Copy`:

```
let x = 5;
let y = x;
println!("{} {}", x, y);
```

- This is fine because the value in x (5) is copied, not moved. So x and y both own their values and can be accessed.

Copy

- In general, types that require heap allocations are not `Copy`.
- `Copy`: integers, floats, booleans, chars, immutable references, and compound types containing only `Copy` types.
- Not `Copy`: Strings, Vectors, mutable references, and compound types containing at least one non-`Copy` type.

Deriving Copy

- Consider the following struct:

```
struct Person {
   id: u64,
   age: u32,
}
```

- Is it `Copy`?
- **No**. But shouldn't it be `Copy`, since it only contains `Copy` types?
- We must explicitly tell the Rust compiler we wish to make it `Copy`

```
#[derive(Copy)]
struct Person {
   id: u64,
   age: u32,
}
```

- Can also make structs cloneable by adding `#[derive(Clone)]`.

More moves

Heinrich Heine Universität Düsseldorf

- Passing a value to a function moves the value:

```
fn main() {
   let s = String::from("hello");
   do_stuff(s);
   // s no longer accessible; it was moved into do_stuff
}

fn do_stuff(s: String) {
   // do stuff with s
}
```

More moves (2)

- Returning a value from a function moves the value to the caller:

```
fn main() {
   let s = get_string();
   // We can now use s, which owns the string
}

fn get_string() -> String {
   String::from("hello")
}
```

- (Aside: Rust functions generally don't take in `Strings`, but don't worry about this for now.)

Aliasing or mutability

- *You can have aliasing or mutability, but not both.*
- Aliasing:

```rust
let x = 162;
let p1 = &x;
let p2 = p1;
```

- x is **aliased**: multiple variables can read (not modify) the variable.

```rust
let mut x = 162;
let p1 = &mut x;
```

- x is **mutable**; p1 can modify the contents of x.
- This is *forbidden*:

```rust
let mut x = 162;
let p1 = &mut x;
let p2 = &mut x;
```

No dangling pointers

- Rust ensures that you don't create dangling references at *compile time*.
- This code won't compile:

```rust
fn get_string() -> &String {
    let s = String::from("hi");
    &s
}
// s is dropped at the end of this function,
// so &s would be a dangling pointer.
// The Rust compiler won't allow this.
```

- Key point: a reference can never outlive the value it points to!

Summary of borrowing rules

- References are always valid and non-null.
- Every value has one owner.
- Values are freed when their owner goes out of scope.
- Assignment moves values (unless the value is `Copy`).
- Values that allocate memory on the heap are usually not `Copy`.
- You can have one mutable reference, or multiple immutable references. But not both.
- A reference can never outlive its value.

Practice

Heinrich Heine Universität Düsseldorf

- Problems …

Problem 1


```
fn main() {
    let mut v = vec![5, 4, 3, 2];
    append_one(&v);
}

fn append_one(v: &mut Vec<i32>) {
    v.push(1);
}
```

- (The `vec!` macro just initializes a Vector, which is a dynamically sized array-based list. A `Vec` is not `Copy`.)
- Incorrect types: we're passing an *immutable* reference to a function that expects a *mutable* reference.

Problem 2

```
fn main() {
   let v = vec![5, 4, 3, 2];
   append_one(v);
   assert_eq!(v[4], 1);
}

fn append_one(mut v: Vec<i32>) {
   v.push(1);
}
```

- Use of moved value: v is moved when we call append_one, so we're not allowed to use it in the `assert_eq` statement.

Problem 3

Heinrich Heine
Universität Düsseldorf

```
fn main() {
   let mut v = vec![5, 4, 3, 2];
   let mut v = append_one(v);
   assert_eq!(v[4], 1);
}

fn append_one(mut v: Vec<i32>) -> Vec<i32> {
   v.push(1);
   v
}
```

- No problems here! v is moved into append_one, but append_one returns v.
  So v is moved back into the caller.

Problem 4


```
struct Rect {
   width: u32,
   height: u32,
}

impl Rect {
   fn transpose(&mut self) {
      let tmp = self.width;
      self.width = self.height;
      self.height = tmp;
   }
}

fn main() {
   let r = Rect { width: 2, height: 5 };
   let ptr = &r;
   r.transpose();
   assert_eq(ptr.width, 5);
}
```

- r is mutably borrowed (in `transpose`) while immutably borrowed (to `ptr`)! Not allowed.

Problem 5

```
fn main() {
    let v = vec![1, 2, 3, 4];
    let one = &vec[0];
    vec.push(5);
    println!("1 = {}", *one);
}
```

- Cannot borrow `vec` mutably (to `push`) while it is already immutably (to one) borrowed.
- Think about this: the call to push might result in the `vec` being realloc'd. This might invalidate the one pointer. borrowed while immutably borrowed! Not allowed.

Convenience types

Heinrich Heine
Universität Düsseldorf

Vectors

- Dynamically sized arrays.
- Create a `Vec`:

```
// The type annotation is needed if the compiler can't determine the element type.
let x: Vec<i32> = Vec::new();
```

- Or use the `vec!` macro:

```
let x = vec![1, 2, 3];
let y = vec![162; 3]; // Equivalent to vec![162, 162, 162].
```

- Operations on a `Vec`:

```
let mut x = vec![1, 2, 3];
assert_eq(x.len(), 3);
x.push(4);
```

Options

- Pointers are never null! What if you actually *want* something to be null?
- Use an `Option<T>`! Here's the definition of `Option`, from the standard library:

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

- Two possible cases: the option is either `None`, or it is `Some`.
- If an Option is `Some`, the value in the `Some` variant will always be a valid value of type `T`.
- The `T` is a generic type parameter. It behaves like generics in other languages, such as Java.
- This prevents us from having to manually define separate `Option` types for `i32`, `String`, etc.

Options

- Here's how you can use an option:

```rust
// This type annotation is not necessary.
let x: Option<i32> = Some(4);
assert!(x.is_some());
let y = x.unwrap(); // get the value out of Option
assert_eq!(y, 4);

// This type annotation IS necessary!
let z: Option<i32> = None;
assert!(z.is_none());

let w = Some(String::from("hello"));
match w {
   Some(s) => println!("{} world!", s),
   None => panic!("didn't expect to get here"),
}
```

Error handling

- Most languages handle errors via one of two ways:

- Try/catch exceptions (Python, Java, JavaScript, etc.)
- Returning a separate error value (Go)
- Many times, errors are not handled properly or are tedious to handle.

- Rust tries to make error handling easier. It's not always smooth sailing though.

Error handling

- The idiomatic way to handle errors in Rust is via `Result<T, E>`:

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

- Functions that may not complete successfully should return a `Result`.
- If the result is `Ok`, the caller can access the returned value (of generic type `T`).
- If the result is `Err`, additional information (of generic type `E`) about the error is returned.
- If a program cannot recover from an error, you can `panic!` instead of returning a `Result`. The panic will immediately terminate the program.

Error handling

- Example:

```
use std::fs::File;

fn main() {
   let f = File::open("hello.txt");
   let f = match f {
      Ok(file) => file,
      Err(error) => panic!("Problem opening the file: {:?}", error),
   };
}
```

- Opening a file is **fallible**, so `File::open` returns a `Result`. We check if the open was successful; if not, we panic and exit.

Error handling

- Matching on `Results` all the time can be tedious. If we know we are going to panic on an `Err`, we can use `unwrap()` instead:

```rust
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

- `unwrap()` panics on error; otherwise, it returns the data contained in the `Ok` variant.

Error handling

- Another shortcut is the `?` operator:

```
use std::fs::File; use std::io; use std::io::Read;

fn read_file() -> Result<String, io::Error> {
   let mut f = File::open("hello.txt")?;
   let mut s = String::new();
   f.read_to_string(&mut s)?;
   Ok(s)
}
```

- If a Result is an `Err`, `?` causes the function to return that `Err`. Otherwise, `?` unwraps the `Ok` variant.
- It's actually a bit more complicated than that. The `?` also attempts to do some type conversion: If your function returns `Result<T, E1>`, but you apply `?` to a `Result<U, E2>`, Rust will try to convert error `E2` into an error of type `E1`.

Further reading

- The Rust book
  - https://doc.rust-lang.org/stable/book/
- Rust by Example
  - https://doc.rust-lang.org/stable/rust-by-example/
- Smart pointers
  - https://doc.rust-lang.org/book/ch15-00-smart-pointers.html
- Generics, Traits, and Lifetimes
  - https://doc.rust-lang.org/book/ch10-00-generics.html