

CS 4414/5416: Systems Programming

Testing, Cargo, Functional Features

David Bindel

2026-09-08

Warm-up exercise

```
use std::cell::RefCell;

fn main() {
    let x = RefCell::new(String::from("cat"));
    let mut y = x.borrow_mut();
}
```

- What might memory look like for representing this?
- What did the person next to you do this weekend?

Recap

- Last Tue
 - Traits and generics
 - Error handling
 - Modules and libraries (on your own)
- Last Thu
 - Memory layouts
 - Smart pointer types
 - Ownership, lifetimes, borrowing revisited
- Fri: Process registry lab, part 1

Where we're going

- Today
 - Closures and iterators
 - Cargo and crates
 - Testing
- Thu
 - Dynamic traits and vtables
 - Advanced Rust, student choice
- Fri: Process registry lab, part 2
- And then on to performance tuning.

- Updated prompt and code is posted (and grader is ready)
- Check-in is due *today* (not yesterday) at 11:59 PM
- Final submission due Monday, 9/14 at 11:59 PM
- For check-in, can submit just **anagram.rs** if needed

- Prelim is evening of Sep 22
- Will post conflict form info soon
- Will all be material from first three weeks
 - You may bring one page as “cheat sheet”
- Viable questions
 - Things like I asked in class
 - Things that appeared in P1
- I will try not to be *deliberately* tricky

Closures

```
let f1 = |x| x+1;  
let f2 = |x| { x+1 };  
let f3 = |x: usize| -> usize { x+1 };  
let f4 = |(x,y)| x+y;
```

- Several syntactic variants
- Often used where the tersest syntax makes sense
 - Can often suppress types (via type inference)

```
let n = 5;  
let plusn = |x: isize| { x + n };
```

- Closures can access variables in environment!
- How does this interact with ownership?

Ownership and captures

```
let mut num = 5;  
let plus_num = |x: i32| x + num;  
let y = &mut num; // Error!
```

- Default is to borrow references to data
- Only borrow mutable reference if needed
- Seems like a problem for returning closures
 - What if they capture stack frame data?

```
fn main() {  
    let mut total = 0;  
    let mut update_total = |x| { total += x; };  
    update_total(1);  
    update_total(2);  
    println!("{}", total);  
}
```

Move closures

```
let mut num = 5;  
let plus_num = move |x: i32| x + num;  
let y = &mut num; // OK
```

- **move** keyword says to move rather than borrow
- For **Copy** data, results in a copy
- Even for **Copy** data, makes a difference

Closures are syntactic sugar for:

- Make a new **struct** type
 - implements trait **Fn**, **FnMut**, or **FnOnce**
- Populate with any needed environment
- Execute code using **struct** fields

- **Fn**: Only borrows shared refs to captures
- **FnMut**: Borrows mutable refs to captures
- **FnOnce**: Moves data
 - Call consumes closure and values
- When *taking* a closure, prefer **FnOnce**
- When *defining* a closure, prefer **Fn**

Using closure types

FnOnce is a trait, not a concrete type, so:

```
fn invoke_callback<T>(c : T) -> usize
where
    T: FnOnce(usize) -> usize {
    c(12345)
}

fn main() {
    println!("{}", invoke_callback(|x| x+1))
}
```

Using closure types

Can also use `impl` argument:

```
fn invoke_callback(c: impl FnOnce(usize)->usize) -> usize {  
    c(12345)  
}  
  
fn main() {  
    println!("{}", invoke_callback(|x| x+1));  
}
```

Using closure types

```
fn invoke_callback(c: impl FnOnce(usize)->usize) -> usize {
    c(12345)
}

fn main() {
    let mut acc = 0;
    let mut c = |x| { acc += x; acc+1 };
    println!("{}", invoke_callback(&mut c), invoke_callback(&mut c));
}
```

Exercise

```
fn simpsons(f: ...) -> f64 { // TODO: What is the type signature?  
    (f(0.0)+4.0*f(0.5)+f(1.0))/6.0;  
}  
  
fn main() {  
    let s = ... // TODO: Call simpsons on cos (a method of f64)  
    println!("{}", s);  
}
```

Iterators

Iterator traits

```
pub trait Iterator {  
    type Item;  
    fn next(&mut self) -> Option<Self::Item>;  
    // Lots of methods with default implementations  
}
```

The `next` method

- Returns `Some(item)` if there is one available
- Returns `None` if the collection is exhausted
- Updates to the next item

```
fn main() {  
    let v = vec![1, 1, 2, 3, 5];  
    for x in v {  
        println!("{x}");  
    }  
}
```

- `for` loop consumes an iterator

```
pub trait IntoIterator {  
    type Item;  
    type IntoIter;  
    fn into_iter(self) -> Self::IntoIter;  
}
```

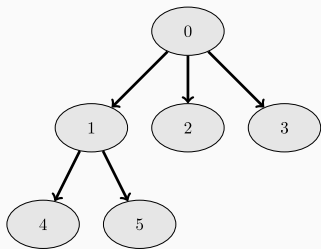
- Collections implement **IntoIterator** for default iterator
- Have seen this implicitly already (iterate over **Vec**)

```
fn main() {  
    let result: i32 = (1..=100)  
        .filter(|x| {x % 3 == 0 || x % 7 == 0})  
        .map(|x| x * x)  
        .sum();  
    println!("Mystery number is {result}");  
}
```

```
let v: Vec<i32> = (1..=10).map(|x| x * x).collect();
```

- The `collect` method collects iterator output
 - Sometimes materializing data is *not* what we want!
- Need to specify type we are collecting into

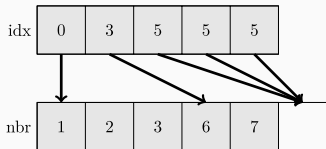
Iterator example



Compressed sparse graph storage:

- **nbr** stores packed neighbor lists
- **idx** stores neighbor list offsets
- `nbr[idx[i]..idx[i+1]]` is neighbors of *i*

Iterate through edges (*i*, *j*)?



Iterator example

```
// Neighbors of node i are in neighbors[idx[i]..idx[i+1]]
struct CompressedSparseGraph {
    neighbors: Vec<usize>,
    idx: Vec<usize>
}

struct EdgeIterator<'a> {
    g: &'a CompressedSparseGraph, // Graph reference
    i: usize, // Current first node
    k: usize // Offset of second node in neighbors
}

impl CompressedSparseGraph {
    fn edges<'a>(&'a self) -> EdgeIterator<'a> {
        EdgeIterator { g: &self, i: 0, k: 0 }
    }
}
```

Iterator example

```
impl<'a> Iterator for EdgeIterator<'a> {  
    type Item = (usize,usize);  
    fn next(&mut self) -> Option<Self::Item> {  
        if self.k < self.g.neighbors.len() {  
            let result = (self.i, self.g.neighbors[self.k]);  
            self.k += 1;  
            if self.k >= self.g.idx[self.i+1] {  
                self.i += 1;  
            }  
            Some(result)  
        } else {  
            None  
        }  
    }  
}
```

Iterator example

```
fn main() {  
    let idx = vec![0,1,2,2];  
    let neighbors = vec![1,2];  
    let g = CompressedSparseGraph{neighbors, idx};  
    for (i,j) in g.edges() {  
        println!("{i},{j}");  
    }  
}
```

Alternate strategy:

- Build iterator for edges on each node
- Iterate over edge list iterators
- Combine edge list iterators with `flatten`

Zero-cost abstractions?

- Explicit loops may not beat iterator chains!
 - Compiles to something pretty close to `for` loops
 - Unsurprising, given that `for` uses iterators!
- But some things to consider in inner loops
 - Helps to provide `size_hint` if `collecting`
 - Can combine phases (`filter_map` vs `filter` + `map`)

Exercise

```
/// Adapt an iterator to discard entries outside lo-hi
/// This should work for iterators over any T implementing Ord
fn censor<T>(..., lo: ..., hi: ...) -> ...
where
    T: ...
{
    ...
}
```

Cargo

Swiss army knife!

```
cargo new      # Start a new project
cargo fmt      # Reformat to Rust standard
cargo doc      # Build package documentation
cargo clean    # Clean up directory
cargo add      # Add external crate dependency to Cargo.toml
cargo build    # Build software
cargo run      # Re-build if needed and run
cargo clippy   # Lints your code
cargo test     # Run tests
cargo bench    # Run benchmarks
cargo package  # Build a package
cargo install  # Install a package from Crates.io
```

... and more – see the Cargo book!

Profiles let you alter compiler settings

- Specified in `Cargo.toml` in `[profile]` table
- Default profiles: `dev`, `release`, `test`, `bench`
 - Usual `dev` corresponds to debug mode
 - Use `--release` flag if you want your code to go fast!

Build scripts

```
// build.rs (example from the Cargo Book)

fn main() {
    cc::Build::new()
        .file("src/hello.c")
        .compile("hello");
    println!("cargo::rerun-if-changed=src/hello.c");
}
```

Build script is **build.rs** in package root

- Written in Rust (vs a **Makefile** or something)
- Not restricted to running **rustc**
 - Can auto-generate code, build C codes, etc
- Cargo doing lifting (e.g. around dependency management)

`cargo clippy` runs a *linter*

- Mostly tells you what could be done better
- You need not always follow its advice!
- ... but it's worth running it

- `cargo package` creates a `.crate` file
 - Ex `p1_maxagram-0.1.0.crate`
 - This is really a Unix tarball format
- Starting with P1 final submission, please upload `crate` files!

NB: Changed crate name to `p1_maxagram` in P1

- `cargo publish` uploads a crate to a registry
 - Default registry is <https://crates.io>
 - You can install packages from the registry!
 - With a login, you can push your own
- Warning: supply chain attacks!
 - People *do* publish malicious code
 - AI assistance makes this easier than ever
 - Maintainers try to manage, but caveat emptor

- `cargo install` installs things in `cargo` space
- Any executable `cargo-*` can be a `cargo` command
- Ex: `cargo install cargo-llvm-cov` to get `cargo llvm-cov`

Testing

*“Beware of bugs in the above code; I have only proved it correct,
not tried it.”*

— Don Knuth, 1977

Tests should *initially* fail

- How else will you know that they're actually testing?
- Tricky tests help you clarify corner cases
- Testability is part of design
- Extreme version is test-driven-development (TDD)

There are implications for P1.

- Small (maybe private) functions and data structures?
- Behavior involving interacting system components?
- Parameterized behavior?
- Behavior involving large workloads?
 - Return later to stress tests and benchmarks

Unit tests

```
/// In lib.rs or the like...
fn important_widget(x: usize) -> usize { /* do it */ }

#[cfg(test)]
mod test {

    #[test]
    fn test_important_widget() {
        assert_eq(important_widget(12345), 678910);
    }
}
```

Goal: Check functions, small data structures, etc

- Lives in code files in **src** tree
- `#[cfg(test)]` to only build on test
- Mark test functions with `#[test]`

Integration tests

```
/// In tests/test_suite.rs
use my_crate::data::Data;

fn data_setup() -> Data { /* ... */ }

#[test]
fn check_data_integrity() {
    let data = data_setup();
    assert!(data.integrity_checks());
}

#[test]
fn check_data_update() {
    let data = data_setup();
    data.update(1234);
    assert_eq!(data.getter(), 1234);
}
```

- Separate files in `tests`
- Again mark test functions with `#[test]`
- Not “in” the crate
 - Need to use crate’s external interface

- `assert!` (condition and optional string)
- `assert_eq!`, `assert_ne!`
- `#[should_panic(expected = "less than")]`
- Or `assert!(value.is_err())` to check recoverable errors

Filtering tests

```
cargo test                # Run all the tests
cargo test --test test_suite # Run just test_suite
cargo test --release       # Run all tests in release mode
```

Wrapping up

Some nice things about the Rust *ecosystem*:

- Lots of good documentation and users
- Actively maintained and updated
- Single standard tools (**cargo**, **rustc**, **rustdoc**)
- Standardized build and package management with **cargo**
- Many external crates on <https://crates.io>

Language choice is never just about the language!

- What do you want to hear about on Thursday?
- Are you worried about the prelim?
 - If so, how can we help?