CS 131 Discussion

Week 5: Midterm Mayhem

matt wang (he/him)
✉️ matt@matthewwang.me
🔗 mattxw.com/131slides
📣 mattxw.com/131feedback

Discussion Agenda

1. feedback from last week
2. selected hw answers (as warmup)
3. midterm review
4. mid-quarter feedback

~ break ~

5. language of the week
6. project 2 overview + tips
7. extra time: midterm/proj q&a

snacc of the week

Feedback / Iteration

Thank you for giving feedback!

Most of this week’s feedback is about the project.

- **you now get your full grade on submission.**
  - **advice: submit early, submit frequently!**
- pushed back the due date for Project 2 by 2 days – now **Nov 8**
- new to the spec:
  - “what we won’t test you on” section
  - more clear instructions on *what* to submit

What I need from you: **please start early!**
(there’s only *so many* emails/CW posts I can respond to on Sunday)

last week’s homework!

(abridged)

People seemed to struggle with…

- **Q1: Subtypes & Supertypes** (const vs non-const, int vs float)
- **Q2B/2C** (dynamic scoping, LEGB / block-scoping, shadowing)
- Q3: (not *really* going to test you like this)
  - people contradicted their own answers for 1A!
  - `a + b` in Python doesn't guarantee that `a` and `b` are numbers :)
- Q4: C++ is *weakly-typed*!

Suggestion: review these ones in particular when it's midterm time! (now)

- *aside: these hw questions are ~ slightly harder~ than the exam ones!*

Q1C

C++ has `float`, `int`, `const float`, and `const int` (among other number types). Which of these are supertypes or subtypes of each other, and which are unrelated?

Q1C: floats and ints

C++ has `float`, `int`, `const float`, and `const int` (among other number types). Which of these are supertypes or subtypes of each other, and which are unrelated?

In *almost all* languages, `float` and `int` have **no subtyping relationship.**

Q1C: floats and ints

C++ has `float`, `int`, `const float`, and `const int` (among other number types). Which of these are supertypes or subtypes of each other, and which are unrelated?

In *almost all* languages, `float` and `int` have **no subtyping relationship.**

What is a value that can be a `float` but not an `int`?

Q1C: floats and ints

C++ has `float`, `int`, `const float`, and `const int` (among other number types). Which of these are supertypes or subtypes of each other, and which are unrelated?

In *almost all* languages, `float` and `int` have **no subtyping relationship.**

What is a value that can be a `float` but not an `int`?

Any decimal! Ex: `3.14`

Q1C: floats and ints

C++ has `float`, `int`, `const float`, and `const int` (among other number types). Which of these are supertypes or subtypes of each other, and which are unrelated?

In *almost all* languages, `float` and `int` have **no subtyping relationship.**

What is a value that can be a `float` but not an `int`?

Any decimal! Ex: `3.14`

What is a value that can be an `int` but not a `float`?

Q1C: floats and ints

C++ has `float`, `int`, `const float`, and `const int` (among other number types). Which of these are supertypes or subtypes of each other, and which are unrelated?

In *almost all* languages, `float` and `int` have **no subtyping relationship.**

What is a value that can be a `float` but not an `int`?

Any decimal! Ex: `3.14`

What is a value that can be an `int` but not a `float`?

Large numbers! Ex: 999999999999999999

(matt shows you in ghci and python and JS)

Q1C: const vs non-const

C++ has `float`, `int`, `const float`, and `const int` (among other number types). Which of these are supertypes or subtypes of each other, and which are unrelated?

Non-const types are subtypes of the const version!

Q1C: const vs non-const

C++ has `float`, `int`, `const float`, and `const int` (among other number types). Which of these are supertypes or subtypes of each other, and which are unrelated?

Non-const types are subtypes of the const version!

- values are the same!
- but …

```
void printAndAssign(float* f) {
    cout << *f << endl;
    *f = 21.0;
}
```

```
float floaty = 42.0;
const float consty = 42.0;

printAndAssign(&floaty); // compiles
printAndAssign(&consty); // error: invalid conversion from
                         // 'const float*' to 'float*'
```

Q1C: const vs non-const

C++ has `float`, `int`, `const float`, and `const int` (among other number types). Which of these are supertypes or subtypes of each other, and which are unrelated?

Non-const types are subtypes of the const version!

- values are the same!
- but …
- **cannot always use a `const float` when we need a `float`!**

Q2B: JavaScript

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
```

Prints:

- `2`
- `x is not defined`

Q2B: JavaScript

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
```

Prints:

- `2`
- `x is not defined`

Matt’s intuition:

1. First, what’s printing what?

Q2B: JavaScript

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
```

Prints:

- 2
- x is not defined

Matt's intuition:

1. **First, what's printing what?**

Q2B: JavaScript

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
```

Prints:

- 2
- **`x is not defined`**

Matt’s intuition:

1. First, what’s printing what?
    - **no global variables!**

Q2B: JavaScript

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
```

Prints:

- **2**
- `x is not defined`

Matt's intuition:

1. First, what's printing what?
   - no global variables!

but, this is the "last value" of x!

Q2B: JavaScript

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
```

Prints:

- **2**
- `x is not defined`

Matt's intuition:

1. First, what's printing what?
   - no global variables!
   - **not dynamically scoped**

but, this is the "last value" of x!

Q2B: JavaScript

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
```

Prints:

- `2`
- `x is not defined`

Matt's intuition:

1. First, what's printing what?
    - no global variables!
    - **not dynamically scoped <-> lexical scoping**

Q2B: JavaScript

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
```

Prints:

- `2`
- `x is not defined`

Matt's intuition:

1. First, what's printing what?
   - no global variables!
   - **not dynamically scoped <-> lexical scoping**
2. What kind of lexical scoping?

Q2B: JavaScript

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
```

Prints:

- `2`
- `x is not defined`

Matt's intuition:

1. First, what's printing what?
   - no global variables!
   - **not dynamically scoped <-> lexical scoping**
2. What kind of lexical scoping?

not an error!

Q2B: JavaScript

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
```

Prints:

- `2`
- `x is not defined`

Matt's intuition:

1. First, what's printing what?
   - no global variables!
   - **not dynamically scoped <-> lexical scoping**
2. What kind of lexical scoping?

not an error!
and, not global

Q2B: JavaScript

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
```

Prints:

- `2`
- `x is not defined`

Matt's intuition:

1. First, what's printing what?
   - no global variables!
   - **not dynamically scoped <-> lexical scoping**
2. What kind of lexical scoping?
   - **function scoping!** (~ LEGB)

Q2C: Rust

```
fn main() {
  let x = 0;
  {
    let x = 1;
    println!(x);  // prints 1
  }

  println!(x);    // prints 0

  let x = "Mystery Language";
  println!(x);    // prints 'Mystery Language'
}
```

Q2C: Rust

```
fn main() {
  let x = 0;
  {
    let x = 1;
    println!(x);  // prints 1
  }

  println!(x);    // prints 0

  let x = "Mystery Language";
  println!(x);    // prints '...'
}
```

What we're told: statically typed, scope of `let x = 0;` is only lines 2, 7, 8, and 9.

Matt's intuition:

1. **Shadowing**

Q2C: Rust

```
fn main() {
  let x = 0;
  {
    let x = 1;
    println!(x);  // prints 1
  }

  println!(x);    // prints 0

  let x = "Mystery Language";
  println!(x);    // prints '...'
}
```

What we're told: statically typed, scope of `let x = 0;` is only lines 2, 7, 8, and 9.

Matt's intuition:

1. Shadowing
2. **Lexical Scoping** (to blocks)

Q2C: Rust

```rust
fn main() {
  let x = 0;
  {
    let x = 1;
    println!(x);  // prints 1
  }

  println!(x);    // prints 0

  let x = "Mystery Language";
  println!(x);    // prints '...'
}
```

What we're told: statically typed, scope of `let x = 0;` is only lines 2, 7, 8, and 9.

Matt's intuition:

1. Shadowing
2. Lexical Scoping (to blocks)

huh????????
It looks like we're changing the type of a var...
But this language is statically typed...

Q2C: Rust

```rust
fn main() {
  let x = 0;
  {
    let x = 1;
    println!(x);  // prints 1
  }

  println!(x);    // prints 0

  let x = "Mystery Language";
  println!(x);    // prints '...'
}
```

What we're told: statically typed, **scope of** **`let x = 0;` is only lines 2, 7, 8, and 9.**

Matt's intuition:

1. Shadowing
2. Lexical Scoping (to blocks)

oh! at this line, the scope of the previous x is over!!

language must be **creating a new variable**

Q2C: Rust

```
fn main() {
  let x = 0;
  {
    let x = 1;
    println!(x);  // prints 1
  }

  println!(x);    // prints 0

  let x = "Mystery Language";
  println!(x);    // prints '...'
}
```

What we're told: statically typed, **scope of `let x = 0;` is only lines 2, 7, 8, and 9.**

Matt's intuition:

1. **Shadowing** (including *within same scope*)
2. Lexical Scoping (to blocks)

oh! at this line, the scope of the previous x is over!!

language must be **creating a new variable**

overview: this week’s HW!

- memory management
- parameter passing
- casts vs conversions

Goal of this (shorter) hw: mostly midterm prep!

general midterm things

- generally: similar to homeworks in style, slightly easier
  - write some code
  - short-answer questions
- matt tips^{TM}
  - review the HW problems you got wrong
  - review the HW problems I listed week-on-week!
  - with a friend, explain a concept in < 5 sentences or with an example
- see carey’s email (on midterm topics, open-book, logistics)
- coming soon: **practice midterm**

concepts you are responsible for (read this later)

- Haskell: pattern matching, guards, currying, partial application, algebraic data types, immutability, and general syntax/types
- Python: object references & object model & implications, classes/OOP, syntax
- Haskell & Python: lists & list comprehensions, lambdas & first-class functions, closures, map & filter & reduce/fold
- Types: static/dynamic/gradual, strong/weak, subtype/supertype, casting/conversion, duck typing
- Scoping & Binding: lexical/dynamic, lifetime vs scope, pass-by value/object reference/reference, garbage collection
- A lil' bit of parameter passing

skills

- Writing code!
    - Includes with constraints (ex: **using a list comprehension, ...**)
    - From scratch or fill-in-the-blank
- Reading code!
    - Given __, why does ___ occur? Would ___ occur?
    - Classify that language!
    - What should this output?
- Answering conceptual questions!
    - See: hw

midterm q&a

ask me anything!

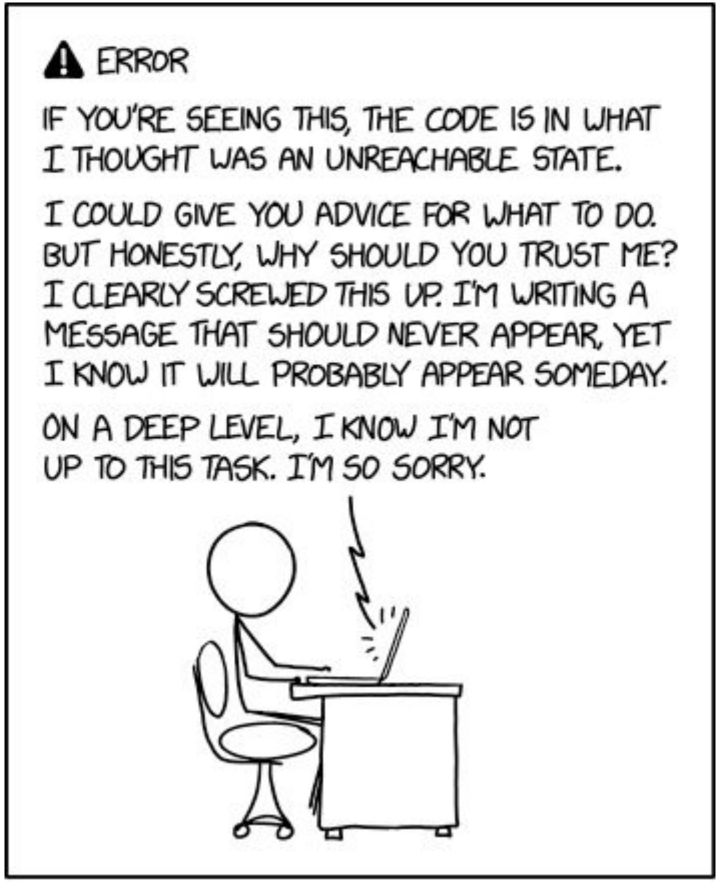
~ mid-quarter feedback ~

give me **advice!**

- what can I do to make the rest of this quarter amazing?
- what about the next time I TA 131?

To get to this slide:
mattxw.com/131slides

https://forms.gle/MB8rimXyRbqErrMu5


ERROR
IF YOU'RE SEEING THIS, THE CODE IS IN WHAT I THOUGHT WAS AN UNREACHABLE STATE.
I COULD GIVE YOU ADVICE FOR WHAT TO DO. BUT HONESTLY, WHY SHOULD YOU TRUST ME? I CLEARLY SCREWED THIS UP. I'M WRITING A MESSAGE THAT SHOULD NEVER APPEAR, YET I KNOW IT WILL PROBABLY APPEAR SOMEDAY.
ON A DEEP LEVEL, I KNOW I'M NOT UP TO THIS TASK. I'M SO SORRY.

NEVER WRITE ERROR MESSAGES TIRED.

~ break ~

discussion will resume at 10:55

language of the week

🥁🥁🥁🥁🥁

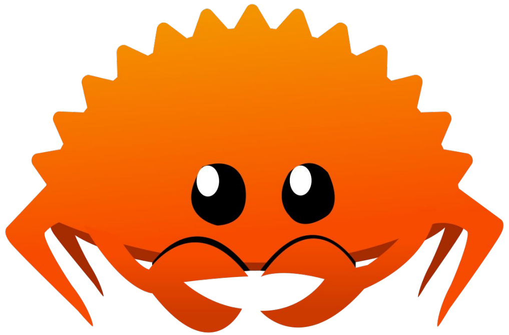
Rust

A better C (for systems programming)?

Some neat things about Rust:

- **compile-time memory safety**
- first-class **functional programming!!**
- super, super good tooling (rust-analyzer)

Also has a *super vibrant* community, and is well-loved: Rust is the **most-loved language 7 years running in the SO Dev Survey!**



Rust & Memory Safety

Rust does not allow (and catches at compile time):

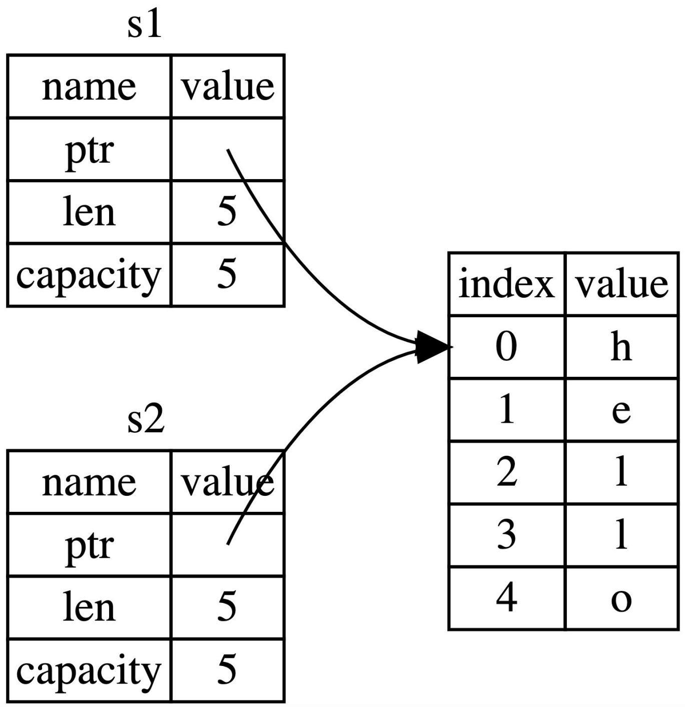
- data races in memory
  - no unwanted memory aliasing
- dangling pointers / use-after-free / double-free
- null-related issues (null doesn’t exist!)
- and more!

What’s the secret sauce?

```rust
let s1 = String::from("hello");
let s2 = s1;

// ... some stuff happens

println!("{}, world!", s1);
```

First: let’s review our memory model. What’s probably going on?

```rust
let s1 = String::from("hello");
let s2 = s1;

// ... some stuff happens

println!("{}, world!", s1);
```



In Rust (and most langs): **a shared reference!**

```rust
let s1 = String::from("hello");
let s2 = s1;

// ... some stuff happens

println!("{}, world!", s1);
```

Okay, so … what could go wrong?

```
let s1 = String::from("hello");
let s2 = s1;

delete(s2);

println!("{}, world!", s1);
```

This is *really bad*! We're inadvertently also affecting `s1`!

Then, this becomes a use-after-free!

Yikes!!

Rust & Memory Safety

Rust does not allow (and catches at compile time):

- data races in memory
    - no unwanted memory aliasing
- dangling pointers / use-after-free / double-free
- null-related issues (null doesn’t exist!)
- and more!

What’s the secret sauce? **Lifetimes, bindings, and references!**

i.e. what we’re learning in class!!

```rust
let s1 = String::from("hello");
let s2 = s1;
```

$\leftarrow$ Here, s1's lifetime ends.

```rust
// ... some stuff happens

println!("{}, world!", s1);
```

Rust's solution!

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` (in Nightly builds, run with -Z
macro-backtrace for more info)

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` due to previous error
```

Then, **this becomes a compile-time error in Rust.**

“but matt”, you say,
“that seems really inconvenient…”

and you’d be right!

there are, of course, still ways to work with strings in Rust.
(for example – maybe you want to copy the string?)

bigger picture: *like with functional programming,* **Rust adds restrictions to your code.** The tradeoff?

Memory Safety!

(and it turns out, this tradeoff is often worth it!)

project overview / walkthrough

Useful Project Links:

- Spec
- Gradescope
- Template + Autograder (optional)

Brewin to Brewin++

New features/changes:

- **static typing for all variables**
  - still: no implicit conversions
- variables are declared first, with default values
- **lexical scoping to blocks/functions, including shadowing**
- functions argument by **value** or **reference**
- functions have an **explicit return type**
  - includes changes to result variables

Brewin to Brewin++

Things that **have not changed**:

- supported types are still just `int, bool, string`
- types still need to match for operations/assignment, no conversions
- error handling
- **still interpreted** (even though it’s static!)
- **how we grade your project**
  - **no syntax errors**
  - **format of program**

```
func main void
  var int a
  assign a 5
  if > a 0
    funccall print a    # prints 5
    var string a        # legal: shadows our original a variable
    assign a "foobar"
    funccall print a    # prints foobar
    var bool b
  endif
  funccall print a      # prints 5
  funccall print b      # Name error: b is out of scope
endfunc
```

```
# Equivalent to: bool absval(int val, int& change_me)
func absval val:int change_me:refint bool
  if < val 0
    assign change_me * -1 val
    return True
  else
    assign change_me val
    return False
  endif
endfunc

func main void
  var int val output
  assign val -5
  funccall absval val output
  funccall print "The absolute value is: " output
  funccall print "Did I negate the input value? " resultb
endfunc
```

Using Your Project 1

First, **look at what you got on Project 1.** If you got:

- 50: great job! Feel free to use your solution!
- 35-49: you probably only missed a few edge cases. Take a look at them (they're published!) and fix them. Using Carey's solution is also fine!
- <35: you're probably missing a good chunk of the project.
  - A good exercise would be to fix the problems yourself, ...
  - But, using Carey's solution is totally good too!

Detour: Carey’s Solution!

- split up into several files
  - managers for environments and functions -> a helpful abstraction
- before interpreting
  - tasks: rm newlines, compute indentation, tokenize, initialize managers, find main
  - tokenize: remove comments first, then search for strings, then split on spaces
- design decisions
  - handlers for every type of starting token
    - endwhile: linear search backwards to find start (w/ indentation)
    - funccall/return: manage instruction pointer stack
  - stores value-type tuples for each variable
  - evaluates expressions with a stack (not recursively)
  - does not use many Python libraries!

Not all decisions have a “right” answer.

~ tips and tricks ~

Please. *Please.* ***Please.***

Start Early!!!

(and test/submit frequently)

Tips on Starting: Concepts

This project requires good conceptual understanding of:

- **lexical scoping** and related data structures (**stacks!**)
- **pass by value** and **pass by reference**
- to a lesser extent, **static typing**

You should review these:

- so you can implement them in the project
- and because they'll be on the midterm!

Tips on Starting: Pragmatic

You shouldn’t implement all the changes at once. Instead, scaffold them.

For example, start by:

- just implementing pass-by-value, or ...
- just implementing function scoping (no shadowing/blocks), or ...
- just implementing type annotations

Then, slowly add each bit. Some of these are much easier to add in isolation than others, and some depend on others (mainly static typing).

Test frequently!

Using the Grading Framework

The grading framework “should” be easy to use. **Requires Python 3.10+**

1. Clone the autograder repo
2. Add an `interpreterv2.py`
3. Run `python3 tester.py 2`

When submitting, **submit your .py files. Do not submit grader.zip!**

To add extra test cases:

- add `.src`, `.exp`, `.in` (optional) files to `testsv2` or `failsv2`
- adjust `successes`/`failures` in `generate_test_suite_v2` (`tester.py`)

Advice from Reviewing Project 1

Start early! Read the spec!

- largest struggle: **overcomplication!**
  - not everything needs to be recursive (ex: expression evaluation w/ a stack)
  - not everything needs to be “optimal” (ex: finding strings in expressions)
  - you don’t **need** most external libraries (ex shlex)
- write good error messages *for yourself*
- implement the spec, not Python!
- don’t sweat the small stuff until you finish the big stuff

Did I mention … **start early? And read the spec?**

Misc Advice

- stacks are *still* your friend!
- build a dependency tree, and see what’s critical and what’s isolated
    - what does reference types rely on? what about function return types?
- questions for you:
    - Is a Brewin’ program a valid Brewin++ program? Why or why not?
    - Why did Carey create a function manager? Environment manager?
    - Can you come up with edge cases, like I talked about last week?
- you will *definitely* want to write your own test cases, but you can share!

~ post-discussion survey ~

always appreciate the feedback!

- do you have feedback **for the class**
- **how helpful was today?**

see you next week ~

https://forms.gle/33gPkKDfajrrQrZ88

midterm / project q&a

(feel free to leave!)