Little Languages

Lecture 10:

Dynamic Memory and Recursive Types in Rust

Before lecture: **Start VM and pull 590 materials from upstream**.

Then…

```
$ cd 590-material-<you>
$ git pull upstream master
$ cd 590-material-<you>/lecture/<today>
```

Heap Values in Rust with **Box<T>**

```
fn main() {
    let a: char = 'a';
    let a_ref: &char = &a;

    let b: Box<char> = Box::new('b');
    let b_ref: &Box<char> = &b;
    let b_content_ref: &char = &(*b);

    println!("a:             {}", a);
    println!("a_ref:         {:p}", a_ref);
    println!("b:             {}", b);
    println!("b_ref:         {:p}", b_ref);
    println!("b_content_ref: {:p}", b_content_ref);
}
```

- How can *any* stack value in Rust be moved to the heap? By *boxing* it.
- The **Box<T>** type allows you to designate a value as a pointer to its location on the heap.
- Unlike in C/Java, a **Box<T>** can *never be null*.
- Unlike in C, the *allocation* and *freeing* of the heap memory is handled by the Box.

```
a:             a
a_ref:         0x7fff5a7e6cbc
b:             b
b_ref:         0x7fff5a7e6cc8
b_content_ref: 0x7f33e7621008
```

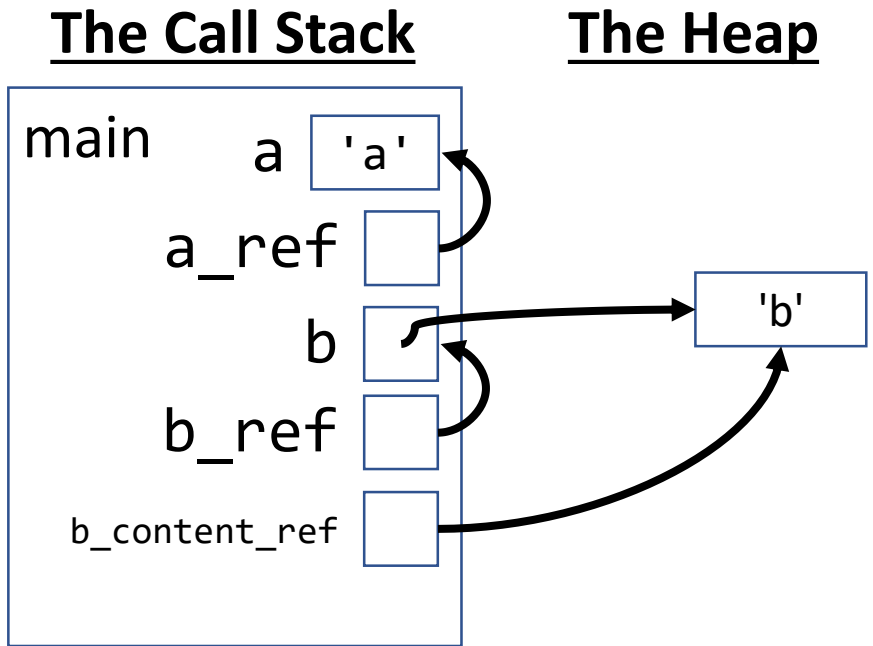
Box<T> representation in memory

```
fn main() {
    let a: char = 'a';
    let a_ref: &char = &a;

    let b: Box<char> = Box::new('b');
    let b_ref: &Box<char> = &b;
    let b_content_ref: &char = &(*b);

    println!("a:             {}", a);
    println!("a_ref:         {:p}", a_ref);
    println!("b:             {}", b);
    println!("b_ref:         {:p}", b_ref);
    println!("b_content_ref: {:p}", b_content_ref);
}
```

A Box's representation on the stack is just a pointer to its value on the heap. There is "zero overhead" versus a heap pointer in C.



How is a **Box<T>** able to release its dynamic memory?

- Recall heap values in C require you to release memory manually via `free` when the values are no longer needed.
- Rust automatically drops values for you when their lifetime ends.
- So how is Box, whose stack value is just a pointer, able to free its memory allocation when it is dropped?
- There is a special **Drop** trait that structs and enums can implement.
  - Its **drop** function is called *just before* Rust drops the its value. This gives you the ability to perform additional clean-up when a value is dropped.

```
#[derive(Debug)]
struct Bigram {
    first: char,
    second: char,
}

impl Drop for Bigram {
    fn drop(&mut self) {
        println!("DROP - {:?}", self);
    }
}
```

Demo of the **Drop** trait (1 of 2)

```
fn main() {
    let a = Bigram {
        first: 'a',
        second: 'b',
    };

    let y = Box::new(Bigram {
        first: 'y',
        second: 'z',
    });

    println!("{:?}", a);
    println!("{:?}", y);

    // Drop when scope ends
}
```

- Notice two Bigram structs are established and printed, then the program ends.
- Before the program ends, there is additional output thanks to the Drop implementation shown on the previous slide.
- Notice the *order* values are dropped in.
  - *Why?*

```
Bigram { first: 'a', second: 'b' }
Bigram { first: 'y', second: 'z' }
DROP - Bigram { first: 'y', second: 'z' }
DROP - Bigram { first: 'a', second: 'b' }
```

Demo of the **Drop** trait (2 of 2)

```
fn main() {
    let a = bigram_builder();
    println!("main a:   {:?}", a);
}

fn bigram_builder() -> Bigram {
    let local1 = Bigram {
        first: 'a',
        second: 'b',
    };
    println!("New:      {:?}", local1);

    let local2 = Bigram {
        first: 'x',
        second: 'y',
    };
    println!("New:      {:?}", local2);

    local2
}
```

Printing messages when a function is dropped allows us to *instrument* a value's lifetime in Rust.

In the example left (PollEv.com/compunc)

1. How many Bigram drops occur?
2. In what order do the drops occur?

Representation of Structures in Memory

- Knowing a char's size in Rust is 4 bytes, how many bytes does a Bigram require?

```
// Find a value's size in memory.
use std::mem::size_of_val;

struct Bigram {
    first: char,
    second: char,
}

fn main() {
    let a = 'a';
    let b = Bigram {
        first: 'b',
        second: 'c',
    };

    println!("{}", size_of_val(&a));
    println!("{}", size_of_val(&b));
}
```

Representation of Structures in Memory

```
#[derive(Debug)]
enum NGram {
    Unigram { first: char },
    Bigram { first: char, second: char },
}

// Import function to tell us size in memory.
use std::mem::size_of_val;

fn main() {
    let unigram = NGram::Unigram { first: 'a' };
    let bigram = NGram::Bigram {
        first: 'b',
        second: 'c',
    };

    println!("unigram: {}", size_of_val(&unigram));
    println!("bigram:  {}", size_of_val(&bigram));
}
```

- How many bytes does an NGram::Unigram require? NGram::Bigram?

Nested Struct Size Cost

- Assume the size of an ID is 2 bytes:
  - One byte for variant tag
  - One byte for data

- How many bytes does a value of type Animal require?

```
#[derive(Debug)]
enum Animal {
    Dog { age: u8, id: ID },
    Stray { id: ID },
}

#[derive(Debug)]
enum ID {
    Tag { first_initial: u8 },
    Chip { id: u8 },
}

fn main() {
    let nelli = Animal::Dog {
        age: 6,
        id: ID::Tag { first_initial: b'n' },
    };
    println!("{:?}", nelli);
}
```

Recursive Enums - i.e. a Linked List

- What if you want to represent a linked list with an enum?
- How many bytes are required to represent an **`NGram`**?
- Answered? Question: Who owns **`b`** when the **`println`** statement is encountered?

```
enum NGram {
    Node { data: char, next: NGram },
    End
}
use self::NGram::{Node, End};

fn main() {
    let b = Node { data: 'b', next: End };
    let a = Node { data: 'a', next: b };
    println!("a: {}", size_of_val(&a));
}
```

Dynamic Memory to the Rescue

```
#[derive(Debug)]
enum NGram {
    Node { data: char, next: Box<NGram> },
    End
}
use self::NGram::{End, Node};

fn main() {
    let b = Node { data: 'b', next: Box::new(End) };
    println!("b size: {}", size_of_val(&b));
    let a = Node { data: 'a', next: Box::new(b) };
    println!("a:      {:?}", a);
    println!("a size: {}", size_of_val(&a));
}
```

- The previous example felt natural and possible because in memory managed languages *reference types are always boxed* *(nullable)* *pointers.*
- To avoid the infinite size structure problem, recursive types must hold references to descendants *not* their actual values.
- To achieve the same semantics in Rust, you'll **Box** recursively typed, descendent values.
  - In C, you would use a pointer.

Preview: Recursive Traversals

```
fn traverse_recur(ngram: &NGram) -> String {
    match ngram {
        Node { data, next } => format!("{} -> {}", data, traverse_recur(&next)),
        End => String::from("End"),
    }
}
```

- Beautiful recursive solutions are possible in Rust thanks to the combination of:
    1. Rust being an *expression language*
    2. The **`match`** expression's concise ability to distinguish variants of enums
    3. The **`match`** expression's ability to assign matched data to local variables

- Sadly, though, in systems programs iterative (looping) solutions are preferred for their performance characteristics.
- There is a postponed RFC (suggested change) in Rust for Tail-Call Optimization that would make some recursive solutions as performant as iterative solutions in specific cases.

Preview: Iterative Traversals

- Iterative traversals like the one shown right are more common.
- Notice we have a mutable ref cursor that starts at the "head" NGram passed in. It gets reassigned.
- Pattern matching with if-let and while-let statements is also powerful!

```
fn traverse(ngram: &NGram) -> String {
    let mut result = String::new();
    let mut cursor = ngram;
    while let Node { data, next } = cursor {
        result.push_str(&format!("{} -> ", data));
        cursor = &next;
    }
    result.push_str("End");
    result
}
```