

CS 4414/5416: Systems Programming

Memory Layouts and Smart Pointers

David Bindel

2026-09-03

Warm-up exercise

```
fn get_letter_id(c: char) -> Result<usize, char> {  
    let c = c.to_ascii_lowercase();  
    if c.is_ascii_lowercase() {  
        Ok((u32::from(c) as usize)-97)  
    } else {  
        Err(c)  
    }  
}  
  
fn get_word_hist(word: &str) -> Result<[u8; 26], char> {  
    todo!("Form a histogram of counting a..z instances in word")  
}
```

Recap

Plan for this week

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

Where we're going (revised)

- Next Tue
 - Testing
 - Iterators and closures
 - Cargo and crates
- Next Thu
 - Dynamic traits and vtables
 - Advanced Rust, further resources
- Fri: Process registry lab, part 2
- And then on to performance tuning.

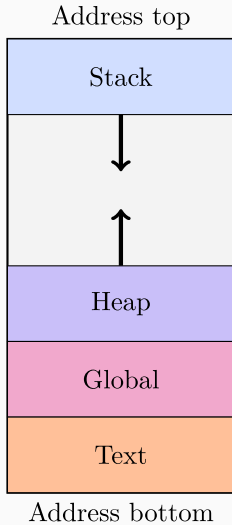
- Project list is posted!
 - First project repository is linked
 - Check-in submission due next *Tuesday*
 - Final submission due two Mondays hence
- We have Google cloud credits! See Ed for details
- Add deadline is Sep 8 – ticket if you need it!

- Write your name and netid on class sheets
 - Print netid, ideally in pen!
- To simplify the end of class
 - Pass class sheets right and up
 - Pass *blank* sheets left and up

Memory Layouts

- Program runs in own *virtual memory*
 - Hardware and OS map virtual to physical memory
 - More about virtual memory in a few lectures
- Virtual address space is partitioned into segments

Stack, heap, and global



Segments (high to low):

1. Stack segment
2. Heap segment
3. Global data segment
4. Text segment (program code)

Keep track of per-call state

- Function arguments
- Return address
- Local variables (fixed size)

Frame is popped when call returns

Can't touch this

```
int* bad_idea()  
{  
    int x = 1;  
    return &x;  
}
```

- Caller receives an invalid pointer!
 - Only makes sense until the frame is popped
 - Ignores valid *lifetime* of referenced data
- Legal C (but undefined behavior)
- Borrow checker prevents this in Rust

- Place for data that outlives calls
 - Also for variable-length allocations
 - Valid lifetime is decoupled from calls
- Managed (semi)-explicitly in Rust, C/C++

- Allocator map says what's available
 - Bad allocation patterns *fragment* the heap
 - Allocation/free is inexpensive, *not* totally free
 - Resizing may cause copying (which can get expensive)
- In Rust, release automatically at end of life
 - **drop** call when owner goes out of scope
 - Can choose to *leak* in some cases

Used for

- String literals
- Other compile-time constants
- **static** items
 - Associated with '**static**' lifetime (more on this soon)

Data size/placement is set at compile time.

Memory representations

- Stack, heap, and global are *where* data is stored
- What about the representations?
- Scalar types should be familiar from 3410
- What about tuples, structs, and enums?
 - Or more elaborate things?

Structs in C

```
struct pair_t {  
    uint8_t tag;  
    uint64_t value;  
};
```

Representation in memory:

- 1 byte for **tag**
- 7 bytes of padding (for *alignment* of **value**)
- 8 bytes for **value**

Usual advice: order fields by decreasing size to minimize alignment padding

Struct and tuple representation

```
fn main() {  
    println!("{}",  
        size_of::<(usize,u8)>(),  
        size_of::<usize>(), size_of::<u8>());  
}
```

- Mostly just list the fields
 - Compiler is allowed to reorder
 - This is different from C!
- Even with reordering, may need to pad for alignment
 - Ex: a **u64** usually starts at an 8-byte boundary (an address that is a multiple of 8)

Enum representation

```
#[repr(u16)]  
enum HttpErrorCodes {  
    Continue = 100,  
    OK = 200,  
    Forbidden = 403,  
    BadRequest = 404,  
}
```

- Simple enums (no payload) are coded as a small integer
- Can force values with = **value** (as in C)
- The **repr** macro makes it a specific integer type
- Use **code as u16** to get value (**as** is like a cast)

Enum representation

```
enum ArticlePages {  
    Pages(u32, u32),  
    Online  
}  
  
fn main() {  
    println!("{}", std::mem::size_of::<ArticlePages>());  
}
```

- Enums with payloads are coded as a *tagged union*
- Must reserve space for tag plus largest variant

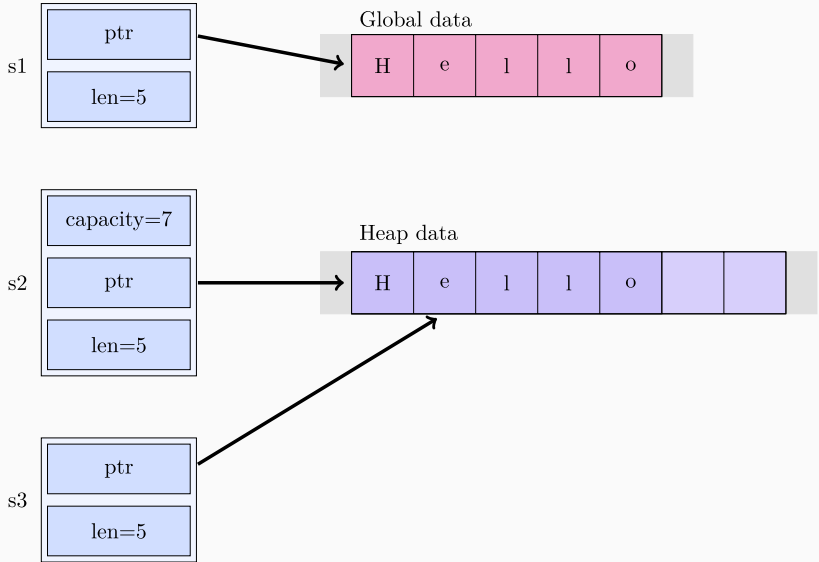
String layout example

```
fn foo(s3: &str) { /* ... */ }  
fn main() {  
    let s1 = "Hello";  
    let s2 = String::from(s1);  
    foo(&s2[1..5]);  
}
```

Pretend this was not optimized to nothing.

What is the memory picture?

String layout picture



- Types of metadata objects
 - Raw pointers (and references) are just addresses
 - Slices are “fat pointers” (address and length)
 - **String** is a **struct** with capacity, address, length
- Capacity and length parameters are in *bytes*
 - Note UTF-8 coded characters vary in length
- Metadata is stored separately from data

For a **String**:

- The *metadata object* owns the heap data
- Transfer of ownership means copying metadata object
 - Or just renaming it – up to the compiler
- Only one owner avoids multiple frees
- Compilers track *lifetime* to avoid use-after-free

Strings and cloning

```
fn hello(s: String) {  
    println!("Hello, {s}");  
}  
  
fn main() {  
    let s = String::from("World");  
    hello(s.clone());  
    hello(s);  
}
```

The `clone` method (trait `Clone`) usually makes a *deep copy*

- Copy the *data* and make new metadata
 - New metadata has different pointer, to new data!
- `hello` consumes `s` and clone (call `drop` at return)

- `String::from(s)` copies data from `s`. Why?
- What is the size of an `Option<u16>`? An `Option<u32>`?
- Did the person next to you ever have pets? What type?

Pointer types

- Basic pointer types (non-owning)
 - References: `&T` and `&mut T`
 - Slices: `&[T]` and `&mut [T]`
 - Raw pointers: `*const T` and `*mut T` (unsafe)
- Smart pointer types
 - `Box<T>`: Owning pointer
 - `Rc<T>`: Reference counted smart pointer
 - `RefCell<T>`: Run-time checked references
- Many containers (`String`, `Vec`) use pointers internally.

```
fn main() {  
    let mut v = Box::new(5u8);  
    *v += 1;  
    println!("{}", v);  
}
```

- A **Box<T>** is an owning pointer to a **T** on the heap
- Implements the **Deref** trait, ***** accesses contents
 - Can therefore call **T** methods directly on **Box<T>**
- Also gives us borrow operator (**&** and **&mut**)

Box type and linked lists

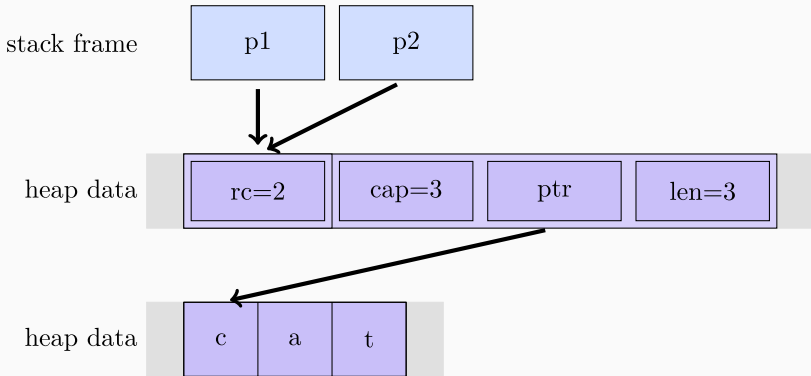
```
#[derive(Debug)]
enum List<T> {
    Link {data: T, next: Box<List<T>>},
    Empty
}

fn main() {
    let mut l: List<u8> = List::Empty;
    for i in (0..5).rev() {
        l = List::Link {data: i, next: Box::new(l)};
    }
    println!("{:?}", l);
}
```

```
use std::rc::Rc;

fn main() {
    let p1: Rc<u8> = Rc::new(String::from("cat"));
    let p2 = Rc::clone(&p1);
    println!("{p1:?} and {p2:?}");
}
```

Rc types



- `rc` is the strong reference count
 - For concurrent code, need *atomic updates* of `rc`
 - Will talk about `Arc` (and `Mutex`) later in semester

- Both **p1** and **p2** point to the same heap data
- Data is stored along with a reference counter
- Can lead to leakage with cycles
 - But cycles are hard to create
 - Requires *interior mutability*

- Single owner (like `Box<T>`)
- Usual rules apply
 - One mutable ref or many immutable (but not both)
 - References must always be valid
- But with `RefCell`, enforce borrow safety *at runtime*
- Good for a little mutability (e.g. logging)

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

fn main() {
    let x = RefCell::new(5);
    {
        let mut y = x.borrow_mut();
        *y += 1;
    }
    println!("{x:?}");
}
```

- Can change contents of `x` *even though* `x` is immutable
- This is *interior mutability*
- OK, but violating borrow rules causes runtime panic

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?

Borrowing and Lifetimes

Basic rules

- Each entity has one owner
- One mutable ref or many immutable refs (not both)
- References must always be valid

... with some addenda for smart pointers

Passing and returning

```
/// Get anagram class by index
pub fn get_class(&self, id: usize) -> &[usize] {
    let lo = self.class_offsets[id];
    let hi = self.class_offsets[id + 1];
    &self.word_ids[lo..hi]
}
```

- So far, we mostly passed references *into* functions
- Can't **drop** reference passed from caller while in callee!
- What about references passed *out* of functions?

What could go wrong?

Passing out references seems dangerous!

- Could I violate reference exclusivity rules?
- What about reference validity rules?

Want to relate input and output references somehow.

Why does this work?

```
/// Get anagram class by index
pub fn get_class(&self, id: usize) -> &[usize] {
    let lo = self.class_offsets[id];
    let hi = self.class_offsets[id + 1];
    &self.word_ids[lo..hi]
}
```

- One reference goes in (`&self`)
- One slice comes out
- Assume you borrowed out of `&self`
 - So `&self` (and return value) *remain borrowed* on return
 - This is a *lifetime* constraint
- If assumption is wrong, you need to annotate

Implicit lifetime tracking

```
#[derive(Debug)]
struct Point2d(f64, f64);

fn borrow1(p: &Point2d) -> &f64 { return &p.0; }
fn get1(p: &Point2d) -> f64      { return p.0; }
fn change2(p: &mut Point2d)      { p.1 = 1.0; }

fn main() {
    let mut p = Point2d(0.0, 0.0);
    //let x = borrow1(&p); // Compile error
    let x = get1(&p);      // This is OK
    change2(&mut p);
    println!("{x} {p:?}");
}
```

Explicit lifetimes

```
fn ifelse<'a>(cond: bool, x: &'a str, y: &'a str) -> &'a str {  
    if cond { x } else { y }  
}  
  
fn main() {  
    let x = "vanilla";  
    let y = "chocolate";  
    println!("{}", ifelse(false, x, y));  
}
```

- 'a is a *lifetime* parameter
- Name doesn't really matter; where it appears does
- Here, compiler assumes *both* inputs borrowed on output

Lifetimes in compound types

```
struct Highlight<'doc>(&'doc str);

fn main() {
    let doc = String::from("I like rabbits!");
    let rabbit = Highlight(&doc[7..13]);
    println!("{}", rabbit.0);
}
```

When returning a reference, can skip lifetime parameter if

- Only one reference is passed into a function
- Lifetime of return is the same as `&self` in a method

Otherwise, need an explicit parameter.

Wrap-up

- Project 1 is launched!
 - Check-in is *Tuesday*, Sep 8, 11:59 PM
 - Project is due Sep 14 at 11:59 PM
- Partner finding social today, Gates 114, 4:30-5:30
- Consulting hours and Ed Discussions are open!
- Lab 2 is tomorrow

What would you like to hear about from me
(course material or otherwise)?

- Write your name and netid on class sheets
 - Print netid, ideally in pen!
- To simplify the end of class
 - Pass class sheets right and up
 - Pass *blank* sheets left and up