changelog

28 March 2025: fixup Rc initial examples having wrong count in comments and second example not matching error

why are people still using C/C++?

Python, Java, ...are great languages

why are people using C, C++, etc.?

- which seem horrible for security?

history + good support

- lots of libraries in C, C++, ...

"zero overhead"

- safe languages don't make it easy to get "close to the machine"
- e.g. garbage collection overhead
- e.g. array checking overhead

no language VM — easier to distribute

why are people still using C/C++?

Python, Java, ...are great languages

why are people using C, C++, etc.?

- which seem horrible for security?

history + good support

- lots of libraries in C, C++, ...

"zero overhead"

- safe languages don't make it easy to get "close to the machine"
- e.g. garbage collection overhead
- e.g. array checking overhead

no language VM — easier to distribute

safety rules + escape hatch

idea: can avoid out-of-bounds, etc. with safety rules

...but safety rules don't allow us to do some things fast

so: have "escape hatch" to avoid safety checks in those cases

hope: code that uses escape hatch can be tightly checked

- good target for expensive program analysis

Java: unofficial escape hatch

Oracle JDK and OpenJDK come with a class called `com.sun.Unsafe`

Example methods:

```
public long allocateMemory(long size);
                          // returns pointer value
public void freeMemory(long address);
public long getLong(long address);
public void putLong(long address, long x);
```

can be used to, e.g., write “fast” IntArray class

so, if Java has escape hatch...

why do people not want to write
their performance-sensitive programs in Java?

hard to integrate code that uses escape hatch with normal Java code

hard to efficiently avoid dangling pointers when using escape hatch

- Is it safe to freeMemory from my FastIntArray class?

slow to pass garbage collected references to/from C/assembly code

hard to avoid using garbage collector

- garbage collector performance can be variable

Rust philosophy

default rules that only allow 'safe' things

- no dangling pointers
- no out-of-bounds accesses

escape hatch to use "raw" pointers or unchecked libraries

escape hatch can be used to write useful libraries

- e.g. Vector/ArrayList equivalent
- *expose interface that is safe*

simple Rust syntax (1a)

```
fn main() {
    println!("Hello, World!\n");
}
```

simple Rust syntax (1b)

```
fn main() {
    let name = "World";
    println!("Hello, {}!\n", name);
}
```

simple Rust syntax (2)

```
fn timesTwo(number: i32) -> i32 {
    return number * 2;
}

/* or last value automatically returned: */
fn timesTwo(number: i32) -> i32 {
    number * 2
}
```

simple Rust syntax (3)

```
struct Student {
    name: String,
    id: i32,
}

fn get_example_student() -> Student {
    Student {
        name: String::from("Example Fakelastname"),
        id: 42,
    }
}
```

simple Rust syntax (4)

```
fn factorial(number: i32) -> i32 {
    let mut result = 1;
    let mut index = 1;
    while index <= number {
        result *= index;
        index = index + 1;
    }
    return result;
}
```

simple Rust syntax (4)

```
fn factorial(number: i32) -> i32 {
    let mut result = 1;
    let mut index = 1;
    while index <= number {
        result = result * index;
        index = index + 1;
    }
    return result;
}
```

"result" is a mutable variable

type automatically inferred as i32 (32-bit int)

methods in Rust (1)

```
pub struct Rectangle {
    width: u32,
    height: u32
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn multiply_size(&mut self, amount: u32) {
        self.width *= amount;
        self.height *= amount;
    }
}

fn foo(rect: &mut Rectangle) {
    let size = rect.area();
    rect.multiple_size(4);
}
```

traits (1a)

```
// import std.cmp.PartialEq; import std.cmp.Eq;
pub struct Rectangle {
    width: u32,
    height: u32
}

impl PartialEq for Rectangle {
    fn eq(&self, other: &Rectangle) -> bool {
        self.width == other.width &&
        self.height == other.height
    }
}

impl Eq for Rectangle {}
```

traits (1b)

```
#[derive(PartialEq,Eq)]
pub struct Rectangle {
    width: u32,
    height: u32
}
```

traits (2)

```
trait ShapeOps {
    fn draw(&self, canvas: &mut Canvas);
}

impl ShapeOps for Rectangle {
    fn draw(&self, canvas: &mut Canvas) {
        ...
    }
}

impl ShapeOps for Square {
    fn draw(&self, canvas: &mut Canvas) {
        ...
    }
}
```

Rust references (1)

```
fn main() {
    let mut x: u32 = 42;

    {
        let y: &mut u32 = &mut x;
        *y = 100;
    }

    let z: &u32 = &x;

    println!("x = {}; z = {}", x, z);
}
```

Rust example with refs

```rust
use std::io;

fn main() {
    println!("Enter a number: ");

    let mut input = String::new();
    // could have also written:
    //   let mut input: String = String::new();

    io::stdin().read_line(&mut input);

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("That was not a number!");
    println!("Twice that number is: {}", number * 2);
}
```

Rust example with refs

```
use std::io;

fn main() {
    println!("En
    let mut input = String::new();
    // could have also written:
    //   let mut input: String = String::new();

    io::stdin().read_line(&mut input);

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("That was not a number!");
    println!("Twice that number is: {}", number * 2);
}
```

"input" is a mutable variable
type is automatically inferred as String

Rust example with refs

```
use std::io;

fn main() {
    println!("Enter a number: ");

    let mut input = String::new();
    // could have also written:
    //   let mut input: String = String::new();

    io::stdin().read_line(&mut input);

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("Tha
    println!("Twice
}
```

pass mutable reference to input

Rust example with refs

```
use std::io;

fn main() {
    println!("Enter a number: ");

    let mut input = String::new();
    // could have also written:
    //   let mut input: String = String::new();

    io::stdin().read_line(&mut input);

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("That was not a number!");
    println!("Twice that number is: {}", number * 2);
}
```

number is an immutable unsigned 32-bit integer

Rust example with refs+struct

```
struct Rectangle {
    width: u32, height u32
}
fn double_rectangle(rect: &mut Rectangle) {
    rect.width *= 2;
    rect.height *= 2;
}
fn show_rectangle(rect: &Rectangle) {
    println!("{}x{}", rect.width, rect.height);
}
fn main() {
    let mut rectangle = Rectangle { width: 4, height: 7 };
    double_rectangle(&mut rectangle);
    show_rectangle(&rectangle);
}
```

rules to stop dangling pointers (1)

objects have an single *owner*

owner is the only one allowed to modify an object

owner can give away ownership

simplest version: only owner can access object

never have multiple references to object — always move/copy

Rust objects and ownership (1a)

```rust
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0;
    for value in &vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let sum = mysum(vector);
    // **moves** vector into mysum()
         // philosophy: no implicit expensive copies

    println!("Sum is {}", sum);
    // ERROR
    println!("vector[0] is {}" , vector[0]);
}
```

Rust objects and ownership (1a)

```
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0;
    for value in &vector {
        total += value
    }
    re
}

fn foo
    le
    le
    //

    println!("Sum is {}", sum);
    // ERROR
    println!("vector[0] is {}" , vector[0]);
}
```

```
   Compiling lecture-demo v0.1.0 (file:///home/cr4bd/spring2017/cs4630/...
error[E0382]: use of moved value: vector
  --> src/main.rs:16:34
   |
13 |     let sum = mysum(vector);
   |                     ------ value moved here
...
16 |     println!("vector[0] is {}" , vector[0]);
   |                                  ^^^^^^ value used here after move
```

Rust objects and ownership (2)

```
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let sum = mysum(vector.clone());
    // give away a copy of vector instead
        // mysum will dispose, since it owns it

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

Rust objects and ownership (2)

```
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = v
    let sum = mysum(vector.clone());
    // give away a copy of vector instead
        // mysum will dispose, since it owns it

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

mysum borrows a copy

moving?

moving a Vec — really copying a pointer to an array and its size

cloning a Vec — making a copy of the array itself, too

Rust defaults to moving non-trivial types

some trivial types (u32, etc.) are copied by default

Rust objects and ownership (3)

```
fn mysum(vector: Vec<u32>) -> (u32, Vec<u32>) {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return (total, vector)
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let (sum, newVector) = mysum(vector);
    // give away vector, get it back

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

Rust objects and ownership (3)

```
fn mysum(vector: Vec<u32>) -> (u32, Vec<u32>) {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return (total, vector)
}

fn foo() {
    let vector: Vec
    let (sum, newVector) = mySum(vector);
    // give away vector, get it back

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

mysum “borrows” vector, then gives it back
uses pointers

ownership rules

exactly one owner at a time

giving away ownership means you *can't use object*

either give object new owner or deallocate

ownership rules

exactly one owner at a time

giving away ownership means you *can't use object*

common idiom — temporarily give away object

either give object new owner or deallocate

ownership exercise

If called like `p = foo(p)`, which follow single-owner rule?

```c
// (A)
char *foo(char *p) {
    free(p);
    return NULL;
}
```

```c
// (B)
char *foo(char *p) {
    p = realloc(p, strlen(p) + 100);
    strcat(p, "test");
    return p;
}
```

```c
// (C)
char *global;
char *foo(char *p) {
    if (p) free(p);
    return global;
}
```

```c
// (D)
char *foo(char *p) {
    p[0] = 'A';
    return p;
}
```

single owner rule limit

single owner prevents use-after-free, etc.

...but too limiting in practice

basically doesn't allow references:

```
let v = vec![1,2,3];
let x = &v;
// now both x, v reference same vector?
// violates single owner rule!
```

will extend rule to be more flexible with two changes:

- borrowing
- multiple readers, one writer

rules to stop dangling pointers (2)

objects have an single **owner**

owner can give away ownership permanently
- object is "moved"

*owner can let someone borrow object **temporarily***
- allows us to give away reference to object
- track as if object temporarily moved
- need to know how long object is borrowed for

can borrow object for reading **while others** also **reading**
- don't need to force exclusive owner/borrower of object
- need to distinguish mutable and immutable borrowing

only **modify** object when exactly one user
- owner or exclusive borrower

borrowing (1)

```
fn mysum(vector: &Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let sum = mysum(&vector);
    // automates (vector, sum) = mysum(vector) idea

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , vector[0]);
}
```

dangling pointers? (1a)

```c
int *dangling_pointer() {
    int array[3] = {1,2,3};
    return &array[0]; // not an error
}
```

```rust
fn dangling_pointer() -> &mut i32 {
    let array = vec![1,2,3];
    return &mut array[0]; // ERROR
}
```

dangling pointers? (1a)

```
int *dangling_pointer() {
    int array[3] = {1,2,3};
```

```
error[E0106]: missing lifetime specifier
  --> src/main.rs:19:25
   |
19 | fn dangling_pointer() -> &mut i32 {
   |                          ^ expected lifetime parameter
   |
   = help: this function's return type contains a borrowed value,
           but there is no value for it to be borrowed from
```

dangling pointers? (1b)

```
/* 'static = "valid forever" */
fn dangling_pointer() -> &'static mut i32 {
    let array = vec![1,2,3];
    return &mut array[0]; // ERROR
}
```

dangling pointers? (1b)

```
/* 'static = "valid forever" */
fn dangling_pointer() -> &'static mut i32 {
```

```
error[E0515]: cannot return value referencing local variable v
 --> src/lib.rs:3:12
  |
3 |     return &v[0];
  |            ^-^^^
  |            ||
  |            |v is borrowed here
  |            returns a value referencing data owned
  |            by the current function
```

```
}
```

dangling pointers? (2)

```
int *ptr;
int dangling_pointer(int *array) {
    ptr = &array[0];
    return array[0];
}
```

```
static mut ptr : &i32 = &0;
fn dangling_pointer(v: Vec<i32>) -> i32 {
    ptr = &v[0];
    return v[0];
}
```

dangling pointers? (2)

```
int
int

}
```

```
stat
fn d

}
```

```
error[E0597]: v does not live long enough
 --> src/lib.rs:3:12
  |
2 | fn dangling_pointer(v: Vec<i32>) -> i32 {
  |                        - binding v declared here
3 |     ptr = &v[0];
  |     -------^---
  |     |      |
  |     |      borrowed value does not live long enough
  |     assignment requires that v is borrowed for 'static
4 |     return v[0];
5 | }
  | - v dropped here while still borrowed
```

dangling pointers? (2)

```
int *ptr;
int d
    p
    r
}

stati
fn da
    p
    r
}
```

```
error[E0133]: use of mutable static is unsafe
              and requires unsafe block
 --> src/lib.rs:3:5
  |
3 |     ptr = &v[0];
  |     ^^^ use of mutable static
  |
  = note: mutable statics can be mutated by
          multiple threads: aliasing violations
          or data races will cause undefined behavior
```

borrowing (2a)

```
fn add1(vector: &mut Vec<u32>) {
    for value in vector {
        *value += 1
    }
}

fn foo() {
    let mut vector: Vec<u32> = vec![1, 2, 3];
    add1(&mut vector);
    println!("vector[0] is {}" , vector[0]);
}
```

borrowing (2b)

```
fn add1(vector: &mut Vec<u32>) {
    for value in vector {
        *value += 1
    }
}

fn foo() {
    let mut vector: Vec<u32> = vec![1, 2, 3];
    // what previous example was basically shorthand for
    {
        let borrowed = &mut vector;
        // borrowing vector here...
        add1(borrowed);
        // until here
    }
    println!("vector[0] is {}" , vector[0]);
}
```

borrow tracking

compiler finds *lifetime* of borrowing

- when is new reference to object created
- when is last use of reference to object

compiler checks for overlap with all other borrowings of that object

applying rules (1a)

single owner, someone can borrow temporarily

only modify if exactly one user

```
let mut x = 42;     // (1)  int x = 42;         // (1)
let p = &mut x;     // (2)  int *p = &x;        // (2).
*p = 10;            // (3)  *p = 10;            // (3)'
println!("{}", x); // (4)  printf("%d\n", x); // (4)
```

Exercise 1/2/3/4: The owner of x on line 1/2/3/4 is:

A. (original owner) the variable x
B. (borrowed) the pointer/reference p

applying rules (1b)

single owner, someone can borrow temporarily

only modify if exactly one user

```
let x = vec![vec![1],vec![2]];
let y = &mut x[0];
let z = &mut y[0];
y.push(4);
x.push(vec![4]);
*z += 1;
y.push(5);
```

exercise: what compile errors? how to fix and get same result?

applying rules (2)

single owner, someone can borrow temporarily

only modify if exactly one user

```
let mut x = 42;     // (1)
let p = &mut x;     // (2)
*p = 10;            // (3)
println!("{}", x); // (4)
*p = 11;            // (5)
```

```
int x = 42;         // (1)
int *p = &x;        // (2)
*p = 10;            // (3);
printf("%d\n", x); // (4)
*p = 11;            // (5)
```

Rust rufuses to compile left-side: x being used while borrowed by p

Which changes would avoid this problem?

A. use `*p` in the println!
B. make p mutable, reassign `p = &mut x` after line (4)
C. take a non-mutable reference to x instead of a mutable one

rules to stop dangling pointers (2)

objects have an single **owner**

owner can give away ownership permanently
- object is "moved"

owner can let someone borrow object ***temporarily***
- allows us to give away reference to object
- track as if object temporarily moved
- need to know *how long object is borrowed for*

can borrow object for reading **while others** also **reading**
- don't need to force exclusive owner/borrower of object
- need to distinguish mutable and immutable borrowing

only **modify** object when exactly one user
- owner or exclusive borrower

why lifetimes? (1)

```
let x = vec![1, 2, 3, 4];
let mut q = &x[1];
{
    let mut r = &x[1];
    let y = vec![5, 6, 7, 8];
    if random() == 0 {
        r = &y[1]; // SHOULD BE FINE
        q = &y[1]; // SHOULD BE ERROR
    }
    println!("{}", *r);
}
println!("{}", *q);
```

need to prevent q referring to values that live too long

why lifetimes? (2)

```
fn mystery(ptr: &i32, vec: &Vec<i32>) -> &i32 {...}

fn example() {
    ...
    let mut x = vec![1, 2, 3, 4];
    let mut q = &x[1];
    {
        let mut y = vec![5, 6, 7, 8];
        q = mystery(q, &y);
    }
    println!("{}", *q);
}
```

question: should assignment to be q from mystery be allowed?

lifetimes

every reference in Rust has a *lifetime*

intuitively: how long reference is usable

Rust compiler infers and checks lifetimes

lifetime rules

object is borrowed for duration of reference lifetime

- can't modify object during lifetime
- can't let object go out of scope during lifetime

lifetime of function args must include whole function call

references returned from function must have lifetimes

- based on arguments or static (valid for entire program)

references stored in structs must have lifetime longer than struct

lifetime inference

```
fn get_first(values: &Vec<String>) -> &String {
    return &values[0];
}
```

compiler infers lifetime of return value is same as input

lifetime hard cases

```
// ERROR:
fn get_first_matching(prefix: &str, values: &Vec<String>)
                                    -> &String {
    for item in values {
        if item.starts_with(prefix) {
            return item
        }
    }
    panic!()
}
```

this is a compile-error, because of the return value

compiler need to be told lifetime of return value

lifetime annotations

```
fn get_first_matching<'a, 'b>(prefix: &'a str, values: &'b Vec<String>)
                                -> &'b String {
    for item in values {
        if item.starts_with(prefix) {
            return item
        }
    }
    panic!()
}
```

prefix has lifetime a

values and returned string have lifetime b

lifetime annotations

```
fn get_first_matching<'a, 'b>(prefix: &'a str, values: &'b Vec<String>)
                              -> &'b String {
    for item in values {
        if item.starts_with(prefix) {
            return item
        }
    }
    panic!()
}

fn get_first(values: &Vec<String>) -> &String {
    let prefix: String = compute_prefix();
    return get_first_matching(&prefix, values)
    // prefix deallocated here
}
```

rules to stop dangling pointers (2)

objects have an single **owner**

owner can give away ownership permanently
- object is "moved"

owner can let someone borrow object **temporarily**
- allows us to give away reference to object
- track as if object temporarily moved
- need to know how long object is borrowed for

can borrow object for *reading* ***while others*** *also* ***reading***
- don't need to force exclusive owner/borrower of object
- need to distinguish mutable and immutable borrowing

only ***modify*** object when exactly one user
- owner or exclusive borrower

borrowing (3a)

```
fn add1(vector: &mut Vec<u32>) {
    for value in vector {
        *value += 1
    }
}

fn foo() {
    let mut vector: Vec<u32> = vec![1, 2, 3];
    let first_elem: &u32 = &vector[0];
    println!("*first_elem is {}", *first_elem);
    add1(&mut vector);
    println!("*first_elem is {}", *first_elem); // ERROR
}
```

borrowing (3a)

```
fn add1(vector: &mut Vec<u32>) {
    for value in vector {
        *value += 1
    }
}

fn main() {
    let mut vector = vec![1, 2, 3];
    let first_elem: &u32 = &vector[0];
    println!("*first_elem is {}", *first_elem);
    add1(&mut vector);
    println!("*first_elem is {}", *first_elem);
}
```

```
error[E0502]: cannot borrow vector as mutable because it is also borrowed as immutable
  --> src/main.rs:11:10
   |
9  |     let first_elem: &u32 = &vector[0];
   |                             ------ immutable borrow occurs here
10 |     println!("*first_elem is {}", *first_elem);
11 |     add1(&mut vector);
   |          ^^^^^^^^^^^ mutable borrow occurs here
12 |     println!("*first_elem is {}", *first_elem);
   |                                   ----------- immutable borrow later used here
```

borrowing (3b)

```
fn append1(vector: &mut Vec<u32>) {
    vector.push(1);
}

fn foo() {
    let mut vector: Vec<u32> = vec![1, 2, 3];
    let first_elem: &u32 = &vector[0];
    println!("*first_elem is {}", *first_elem);
    append1(&mut vector);
    println!("*first_elem is {}", *first_elem); // ERROR
}
```

borrowing (3b)

```
fn append1(vector: &mut Vec<u32>) {
    vector.push(1);
}

fn
}
```

```
error[E0502]: cannot borrow vector as mutable because it is also borrowed as immutable
  --> src/main.rs:11:10
   |
9  |     let first_elem: &u32 = &vector[0];
   |                             ------ immutable borrow occurs here
10 |     println!("*first_elem is {}", *first_elem);
11 |     add1(&mut vector);
   |          ^^^^^^^^^^^ mutable borrow occurs here
12 |     println!("*first_elem is {}", *first_elem);
   |                                   ----------- immutable borrow later used here
```

aside: find the bug

```
struct vec { int *data; int size; };
void append1(struct vec *v) {
    v.data = realloc(v.data, sizeof(int) * (v.size + 1));
    v.data[v.size] = 1;
    v.size += 1;
}

void foo() {
    struct vec vector;
    vector.data = malloc(sizeof(int) * 3);
    vector.data[0] = 1; vector.data[1] = 2; vector.data[2] = 3;
    vector.size = 3;
    int *first_elem = &vector.data[0];
    printf("*first_elem␣is␣%d\n", *first_elem);
    append1(&vector);
    printf("*first_elem␣is␣%d\n"",␣*first_elem);
}
```

borrowing (4a)

```
fn add1(vector: &mut Vec<u32>) {
    for value in vector {
        *value += 1
    }
}

fn foo() {
    let mut vector: Vec<u32> = vec![1, 2, 3];
    // (lifetime of first_elem starts here)
    let first_elem: &mut u32 = &mut vector[0];
    *first_elem += 1;
    // (lifetime of first_elem ends here)
    add1(&mut vector);
    println!("vector is {:?}", vector);  // [3, 3, 4]
}
```

borrowing (4b)

```
fn add1(vector: &mut Vec<u32>) {
    for value in vector {
        *value += 1
    }
}

fn foo() {
    let mut vector: Vec<u32> = vec![1, 2, 3];
    // (lifetime of first_elem starts here)
    let first_elem: &mut u32 = &mut vector[0];
    add1(&mut vector);
    *first_elem += 1; // ERROR, two mutable borrowings of vector
    // (lifetime of first_elem ends here)
    println!("vector is {:?}", vector);
}
```

borrowing (4b)

```
fn add1(vector: &mut Vec<u32>) {
    for value in vector {
        *value += 1
}

fn
```

```
error[E0499]: cannot borrow vector as mutable more than once at a time
  --> src/main.rs:11:10
   |
9  |     let first_elem: &mut u32 = &mut vector[0];
   |                                    ------ first mutable borrow occurs here
10 |     *first_elem += 1;
11 |     add1(&mut vector);
   |          ^^^^^^^^^^^ second mutable borrow occurs here
12 |     println!("first_elem is {}", *first_elem);
   |                                   ---------- first borrow later used here
```

```
    println!("vector is {:?}", vector);
}
```

borrowing (4c)

```
fn foo() {
    let mut vector: Vec<u32> = vec![1, 2, 3];
    let first_elem: &mut u32 = &mut vector[0];
    vector[1] += 2;  // ERROR: two mutable borrowings
    *first_elem += 1;
}
```

borrowing (4c)

```
fn foo() {
    let mut vector: Vec<u32> = vec![1, 2, 3];
```

```
[E0499]: cannot borrow vector as mutable more than once at a time
 src/main.rs:10:5

    let first_elem: &mut u32 = &mut vector[0];
                                    ------ first mutable borrow occurs
    vector[1] += 2;
    ^^^^^^ second mutable borrow occurs here
    *first_elem += 1;
    ---------------- first borrow later used here
```

restricting modification

```
fn modifyVector(vector: &mut Vec<u32>) { ... }
fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    for value in &vector {
        if value == 2 {
            modifyVector(&mut vector) // ERROR
        }
    }
}
```

value is reference to vector element

can't give away vector

Rust dropping (1)

```
struct Example {}

impl Drop for Example {
    fn drop(&mut self) {
        println!("in Example's drop")
    }
}

fn main() {
    {
        let t = Example {};
        println!("A");
    }
    println!("B");
}
```

output: A(newline)in Example's drop(newline)B

Rust dropping (2)

```
struct Example {}

impl Drop for Example {
    fn drop(&mut self) {
        println!("in Example's drop")
    }
}

fn main() {
    let q: Example;
    {
        let t = Example {};
        println!("A");
        q = t;
    }
    println!("B");
}
```

output: A(newline)B(newline)in Example's drop

Rust dropping (3)

```
#[derive(Clone)] struct Example {}

impl Drop for Example {
    fn drop(&mut self) {
        println!("in drop")
    }
}

fn main() {
    let q: Example;
    {
        let t = Example {};
        println!("A");
        q = t.clone();
    }
    println!("B");
}
```

output: A(newline)in drop(newline)B(newline)in drop

preview: wrapper objects as tool

to manage memory, make objects with custom `drop` functions

creating object: allocates memory; droping: frees memory

Rust compiler will insert drop calls automatically

...and borrowing will enforce error if object still in use

aside: RAII and C++

common C++ idea that Rust is copying:
Resource Acquisition is Initialization (RAII)

will show "smart pointer" types where idea prominent in C++

...but C++ lacks compiler borrowing/etc. enforcement

idea probably didn't start with C++, though most prominent current version

what about dynamic allocation?

saw Rust's Vec class — equivalent to C++ vector/Java ArrayList

idea: Vec wraps a heap allocation of an array

owner of Vec "owns" heap allocation

- delete when no owner

also `Box` class — wraps heap allocation of a single value

- can be used like reference to object
- similar to C++ `unique_ptr`

Box

```rust
pub fn f1a() -> Box<u32> {
    let mut value: Box<u32> =
            Box::new(0);
    *value += 1;
    q(*value);
    value
}

pub fn f1b() {
    let mut value: Box<u32> =
            Box::new(0);
    *value += 1;
    q(*value);
}

pub fn f2() -> u32 {
    let mut value: u32 = 0;
    value += 1;
    q(value);
    value
}
```

```c
unsigned *f1a() {
    unsigned *value =
            malloc(sizeof(unsigned));
    *value += 1;
    q(*value);
    return value;
}

unsigned *f1b() {
    unsigned *value =
            malloc(sizeof(unsigned));
    *value += 1;
    q(*value);
    free(value);
}

unsigned f2() {
    unsigned value = 0;
    value += 1;
    q(value);
    return value
}
```

escape hatch

Rust lets you avoid compiler's mechanisms

implement your own

unsafe keyword

how Vec is implemented

deep inside Vec (1)

```
pub struct Vec<T> {
    buf: RawVec<T>, // interface to malloc
    len: usize,
}

impl<T> Vec<T> {
    ...
    pub fn truncate(&mut self, len: usize) {
        unsafe {
            // drop any extra elements
            while len < self.len {
                // decrement len before the drop_in_place(), so a panic on Drop
                // doesn't re-drop the just-failed value.
                self.len -= 1;
                let len = self.len;
                ptr::drop_in_place(self.get_unchecked_mut(len));
            }
        }
    }
    ...
}
```

deep inside Vec (1)

```
impl<T> Vec<T> {
    ...
    pub const fn as_mut_ptr(&mut self) -> *mut T {
        ...
        self.buf.ptr()
    }
    ...
    pub fn push(&mut self, value: T) {
        // Inform codegen that the length does not change across grow_one().
        let len = self.len;
        // This will panic or abort if we would allocate > isize::MAX bytes
        // or if the length increment would overflow for zero-sized types.
        if len == self.buf.capacity() {
            self.buf.grow_one();
        }
        unsafe {
            let end = self.as_mut_ptr().add(len);
            ptr::write(end, value);
            self.len = len + 1;
        }
    }
    ...
}
```

raw pointers (1)

```
fn main() {
    let mut y: Vec<u32> = vec![10,20,30,40];
    let x: *mut u32 = &mut y[0];
    println!("{:?}", x);
        // 0x5d82222f9b10
    println!("{}", unsafe{*x});
        // 10
    println!("{}", unsafe{*(x.wrapping_add(1))});
        // 20
    unsafe{*x = 11};
    println!("{:?}", y);
        // [11, 20, 30, 40]
}
```

raw pointers (2)

```
fn main() {
    let z: &mut u32;
    {
        let mut y: Vec<u32> = vec![10,20,30,40];
        let x: *mut u32 = &mut y[0];
        z = unsafe {&mut *x};
        *z = 12;
        println!("{}", z);
            // 12
        println!("{:?}", y);
            // [12, 20, 30, 40]
    }
    println!("{}", z);
        // 1134576980 (accesses freed memory!)
}
```

raw pointers (3)

```
use libc;

fn main() {
    let x: *mut u32 = unsafe { libc::malloc(16) as *mut u32 }
    let pointer_value: usize = x as usize;
    println!("{:?} {:#x}", x, pointer_value);
        // 0x58a7196d5b10 0x58a7196d5b10
    unsafe{*x = 0x123456;}
    unsafe{*(x.add(1)) = 0x234567;}
    let y: *mut u32 = (pointer_value+1) as *mut u32;
    println!("{:#x}", unsafe{*y});
        // release mode: 0x67001234
        // debug mode: thread 'main' panicked at src/main.rs:
        //      misaligned pointer dereference: ....
}
```

unsafe action at a distance

raw pointers = like pointers in C, basically

need `unsafe` keyword to use

need to explicitly dereference (*) to read unlike references

can be used to create normal references

...if done incorrectly, can cause all sorts of problems

Rust escape hatch support

escape hatch: make new reference-like object

(...implemented by returning temporary 'real' references)

callbacks on ownership ending (normally deallocation)

Rust compiler enforces that ref-like object not in use when free call made

choice of what happens on copy

alternative rule: reference counting

keep track of number of references

increment count when making new 'clone' of reference

decrement when reference goes away

- Rust borrowing rules will enforce it is not used when this happens

delete when count goes to zero

- Rust automatically calls destructor — no programmer effort

explicit operation to make new reference

Rust implement with Rc type ("counted reference")

Ref Counting Example (0a)

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

fn main() {
    let s_ref: &String;
    let s1: Rc<String>;
    {
        let s2: Rc<String> = Rc::new(String::from("example"));
        s1 = Rc::clone(&s2);
        s_ref = &*s1;
        println!("{s1} {s_ref} {s2}");
            // example example example
        println!("count={}", Rc::strong_count(&s1));
            // count=2
    }
    println!("count={}", Rc::strong_count(&s1));
        // count=1
    println!("{s1} {s_ref}");
        // example example
}
```

Ref Counting Example (0b)

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

fn main() {
    let s_ref: &String;
    let s1: Rc<String>;
    {
        let s2: Rc<String> = Rc::new(String::from("example"));
        s1 = Rc::clone(&s2);
        s_ref = &*s2;
        println!("{s1} {s_ref} {s2}");
        println!("count={}", Rc::strong_count(&s0));
    }
    println!("count={}", Rc::strong_count(&s1));
    println!("{s1} {s_ref}");  // ERROR
}
```

Ref Counting Example (0b)

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

fn main() {
    let s
    let s
    {
        l
        s
        s
        p
        p
    }
    print
    print
}
```

```
error[E0597]: s2 does not live long enough
  --> src/main.rs:9:19
   |
7  |         let s2: Rc<String> = Rc::new(String::from("example"));
   |             -- binding s2 declared here
8  |         s1 = Rc::clone(&s2);
9  |         s_ref = &*s2;
   |                 ^^ borrowed value does not live long enough
...
14 |     }
   |     - s2 dropped here while still borrowed
...
17 |     println!("{}", s_ref);
   |                    ----- borrow later used here
```

Rc<> into real ref

can turn Rc into real ref

borrowing rule enforces that Rc object stays around

allows us to ensure it's not drop()d while still in use

...and use it with functions that expect 'normal' reference

Ref Counting Example (1)

```
struct Grade {
    score: i32, studentName: String, assignmentName: String,
}
struct Student {
    name: String,
    grades: Vec<Rc<Grade>>,
}
struct Assignment {
    name: String,
    grades: Vec<Rc<Grade>>
}

fn add_grade(student: &mut Student, assignment: &mut Assignment, score: i32) {
    let grade = Rc::new(Grade {
        score: score,
        studentName: student.name.clone(),
        assignmentName: assignment.name.clone(),
    })
    student.grades.push(Rc::clone(&grade));
    assignment.grades.push(Rc::clone(&grade));
    println!("Added grade with score={}", grade.score);
}
```

Rc limitations

Rc: only gives references to *read-only* objects

- cannot enforce "only one mutable reference" rule
- ...we'll look at doing that next, but needs more bookkeeping

Rc: allows memory leaks via circular references

- correct way to handle ciruclar references: Weak
- ...but not enforcement that it is used when needed

aside: Weak

```
struct Foo {
    my_bar: Rc<Bar>,
    ...
}
struct Bar {
    my_foo: Weak<Foo>.
    ...
}
...
let bar: Bar = ...;
...
match bar.my_foo.upgrade() {
    Some(foo_rc) => {
        // foo_rc is an Rc<Foo>
        ...
    },
    None => {
        // the foo object was deleted
    }
}
```

Rust escape hatch support

escape hatch: make new reference-like object

- (...implemented by returning temporary 'real' references)

Rc: `Rc<T>` acts like `&T`

callbacks on ownership ending (normally deallocation)

- Rust compiler enforces that ref-like object not in use when free call made
- Rc: deallocating `Rc<T>` decrements shared count
- Rc: real object only decremented on count $==$ 0

choice of what happens on copy

- Rc: no implicit copy; explicit `clone` operation increments count

Rc implementation (approx) (0)

```
struct RcInner<T: ?Sized> {
    strong: Cell<usize>,    // <-- count of Rc<T>s pointing to this
    weak: Cell<usize>,      // <-- count of Weak<T>s pointing to this
    value: T,               // <-- actual data
}

pub struct Rc<T: ?Sized> {
    ptr: NonNull<RcInner<T>>,
    phantom: PhantomData<RcInner<T>>, // <- so compiler infers what operat
}
```

NonNull = raw pointer wrapper

Cell = container for mutable object (have to copy in/out of)

Rc implementation (approx) (1)

```
impl<T: ?Sized> Clone for Rc<T> {
    ...
    fn clone(&self) -> Rc<T> {
        self.inc_strong(); // <-- increment reference count
        Rc { ptr: self.ptr }
    }
}
```

Rc implementation (approx) (2)

```
unsafe impl<#[may_dangle] T: ?Sized> Drop for Rc<T> {
    ...
    fn drop(&mut self) { // <-- compilers calls on deallocation
        unsafe {
            self.inner().dec_strong();
            if self.inner().strong() == 0 {
                self.drop_slow();
            }
        }
    }
    ...
}
```

Rc implementation (approx) (3)

```
impl<T: ?Sized> Deref for Rc<T> {
    type Target = T;

    #[inline(always)]
    fn deref(&self) -> &T {
        &self.inner().value
    }
}
```

observation: returned reference still has lifetime

compiler will enforce that extracted reference only used when Rc object valid

C++ has ‘smart pointers’

probably what inspired Rust Box, Rc, etc.

`std::shared_ptr` (like Rc), `std::unique_ptr` (like Box)

> like Rust Deref/DerefMut, implement `operator*` and `operator->` to act like ‘normal’ pointers

like Rust, internally return temporary real references/pointers

problem: no compiler enforcement of ownership rules

can accidentally use ‘temporary’ reference/pointer for too long

reference counting elsewhere

a lot of programs do manual reference counting

- example: lots of stuff in Linux kernel

Swift, Perl, Python implicitly do reference counting for ‘normal’ references

- means normal references not ‘zero-overhead’
- more predictable object deallocation than ‘normal’ garbage collection

alternate rule: tracked usage (RefCell)

runtime-enforced version of Rust borrowing rules

`borrow_mut()`: give `&mut T`-like object only if other active borrows

- increment counter of active mutable borrows (instead of tracking statically)
- decrement counter when `&mut T` no longer in use

`borrow()`: give &T-like object only if no active mutable borrows

- increment counter of active immutable borrows
- decrement counter when &T no longer in use

RefCell example (0)

```
fn myadd(x: &RefCell<i32>, y: &RefCell<i32>, z: &RefCell<i32>) {
    let mut x_value = x.borrow_mut();
    let y_value = y.borrow();
    let z_value = z.borrow();
    *x_value += *y_value;
    *x_value += *z_value;
    println!("{}, {}, {}", x_value, y_value, z_value);
}

fn main() {
    let x: RefCell<i32> = RefCell::new(1);
    let y: RefCell<i32> = RefCell::new(2);
    let z: RefCell<i32> = RefCell::new(3);

    myadd(&x, &y, &z); // 6, 2, 3
    myadd(&x, &y, &y); // 10, 2, 2
    myadd(&x, &x, &x); // RUNTIME ERROR
}
```

RefCell example (1)

```
fn appendsum(x: &RefCell<Vec<i32>>, y: &RefCell<Vec<i32>>, z: &RefCell<Vec<i32>>)
    let mut x_value = x.borrow_mut();
    let y_value = y.borrow();
    let z_value = z.borrow();
    let i = 0;
    for (y_number, z_number) in y_value.iter().zip(z_value.iter()) {
        x_value.push(y_number + z_number);
    }
    println!("{:?}", *x_value)
}

fn main() {
    let x: RefCell<Vec<i32>> = RefCell::new(vec![1]);
    let y: RefCell<Vec<i32>> = RefCell::new(vec![2]);
    let z: RefCell<Vec<i32>> = RefCell::new(vec![3]);

    appendsum(&x, &y, &z);
    appendsum(&x, &y, &y);
    appendsum(&x, &x, &x);
}
```

Rust escape hatch support

escape hatch: make new reference-like types

- (...implemented by returning temporary 'real' references)

RefCell: `borrow_mut()` method gives mutable-ref-like object

- crashed if conflicting borrowing active

RefCell<T> x: `x.borrow()` gives immutable-ref-like object

- crashed if conflicting borrowing active

callbacks on ownership ending (normally deallocation)

- Rust compiler enforces that ref-like object not in use when free call made
- RefCell<T> — deallocate normally

choice of what happens on move/copy

- RefCell<T> —- no implicit copy of RefCell or ref-like objects

approx RefCell implementation (1)

```
pub struct RefCell<T: ?Sized> {
    // mutable integer
        // set to -1 on mutable borrow
        // incremented on immutable borrow
    borrow: Cell<BorrowFlag>,
    value: UnsafeCell<T>,
}

pub fn borrow_mut(&self) -> RefMut<'_, T> { ... }

pub struct RefMut<'b, T: ?Sized + 'b> {
    value: NonNull<T>,
    borrow: BorrowRefMut<'b>,
    ...
}
```

RefMut = type that acts like reference (`Deref`, `DerefMut`)

BorrowRefMut = contains pointer to Cell<BorrowFlag>, handles updates

approx RefCell implementation (RefMut)

```
pub struct RefMut<'b, T: ?Sized + 'b> {
    value: NonNull<T>,
    borrow: BorrowRefMut<'b>,
    ...
}

impl<T: ?Sized> DerefMut for RefMut<'_, T> {
    fn deref_mut(&mut self) -> &mut T {
        unsafe { self.value.as_mut() }
    }
}
```

RefMut idea

RefMut acts like a mutable reference value

- DerefMut implementation allows conversion to reference when needed
  done automatically when you try to use like reference
  (also similar non-mutable reference wrapper type)

compiler automatically `drop`s when it goes out of scope

compiler knows RefMut contains item with lifetime `'b`

- compile-time error to use for too long

approx RefCell implementation (BorrowRefMut)

```
struct BorrowRefMut<'b> {
    borrow: &'b Cell<BorrowFlag>,
}
impl Drop for BorrowRefMut<'_> {
    fn drop(&mut self) {
        let borrow = self.borrow.get(); self.borrow.set(borrow + 1);
    }
}
impl<'b> BorrowRefMut<'b> {
    fn new(borrow: &'b Cell<BorrowFlag>) -> Option<BorrowRefMut<'b>> {
        match borrow.get() {
            UNUSED => {
                borrow.set(UNUSED - 1); Some(BorrowRefMut { borrow })
            }
            _ => None,
        }
    }
}
```

example: concurreny UAF bug

```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/main.c
208. static const struct ieee80211_ops cw1200_ops = {
        ......
215.    .hw_scan = cw1200_hw_scan,
        ......
223.    .bss_info_changed = cw1200_bss_info_changed,
        ......
238. };

FILE: linux-4.19/drivers/net/wireless/st/cw1200/scan.c
 54. int cw1200_hw_scan(...) {
        ......
 91.    mutex_lock(&priv->conf_mutex);
        ......
123.    mutex_unlock(&priv->conf_mutex);
125.    if (frame.skb)
126.       dev_kfree_skb(frame.skb); // FREE
        ......
129. }

FILE: linux-4.19/drivers/net/wireless/st/cw1200/sta.c
1799. void cw1200_bss_info_changed(...) {
         ......
1807.    mutex_lock(&priv->conf_mutex);
         ......
1849.    cw1200_upload_beacon(...);
         ......
2075.    mutex_unlock(&priv->conf_mutex);
         ......
2081. }
-------------------------------------------
2189. static int cw1200_upload_beacon(...) {
         ......
2221.    mgmt = (void *)frame.skb->data; // READ
         ......
2238. }
```

Figure 2: A reported bug in the *cw1200* driver in Linux 4.19

Figure from Bai, Lawall, Chen and Mu (Usenix ATC'19)

"Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux drivers"

bug in a wireless networking driver

data races

Rusts rules around modification built assuming concurrency

OSes and other "systems programming" applications use multiple cores/threads

particular problem: value being used from multiple threads at same time

data races from use-after-free

given x: Rc<Foo> variable calling x.clone() on two cores

- some variable shared between two cores
- reference counting will prevent use-after-free, right?

```
x.clone on core A                x.clone on core B
-------------------------------------------------
x.inc_strong():
  temp <- self.count
                                 x.inc_strong():
                                   temp <- self.count
                                   self.count <- temp +
  self.count <- temp + 1
```

problem: reference count one too low!

Rust solution?

one option: require Rc implementation to handle mutiple cores

- problem: not zero overhead

Rust solution: different types for multithreaded/multicore code

two "traits" to mark custom types:

- Sync: can be used from multiple cores/threads at once
- Send: can be moves from one thread to another

two implementations of referenc counting

- Rc: not suitable for multicore, not marked Sync/Send
- Arc: is suitable for multicore, slower than Rc probably

other things languages can enforce?

saw: enforcing no use-after-free

lots of coding conventions we might try to enforce:

code's runtime does not depend on secret data

- secret data has different type
- variable time operations prohibited with secret data

sensitive data not passed to wrong place

- sensitive data has different type
- assignment to wrong places is a type error

code has bounded runtime

- langauge prohibits not unbounded loops, recursion, etc.

other ‘smart pointers’ Rust supports

saw: Rc (reference count), RefCell (track active borrows)

Box — wrap heap-allocation, free when it goes out of scope

OnceCell — allow setting value exactly once + unlimited reads after set

Arc — atomic reference count — multithread reference count

Mutex — enforce one thread at a time by waiting

RwLock — enforce one writer + any number of reads by waiting

...

exercise: which smart pointer?

Rc, Arc (reference counting, w/ or w/o threading support

RefCell (borrowing, check at runtime)

Weak (reference counting, but don't contribute to count — works with Rc)

Mutex (with multicore, one-at-a-time by waiting)

say I have flight reservation system with Flight objects that have references to Ticket objects and vice-versa,
and Customer objects that have references to Ticket objects and vice-versa?

Rust linked list

not actually a good idea

use `Box<...>` to represent object on the heap

no null, use `Option<Box<...>>` to represent pointer.

Rust linked list (not recommended)

```
struct LinkedListNode {
    value: u32,
    next: Option<Box<LinkedListNode>>,
}

fn allocate_list() -> LinkedListNode {
    return LinkedListNode {
        value: 1,
        next: Some(Box::new(LinkedListNode {
            value: 2,
            next: Some(Box::new(LinkedListNode {
                value: 3,
                next: None
            }))
        }))
    }
}
```

why the box? (1)

```
struct LinkedListNode { // ERROR
    value: u32,
    next: Option<LinkedListNode>,
}

// error[E0072]: recursive type `LinkedListNode` has infinite size
```

why the box? (2)

```
struct LinkedListNode { // ERROR
    value: u32,
    next: Option<&LinkedListNode>,
}
// error[E0106]: missing lifetime specifier
//  --> src/main.rs:48:18
//   |
// 48 |     next: Option<&LinkedListNode>,
//   |                  ^ expected lifetime parameter
```

zero-overhead

normal case — lifetimes — have no overhead

compiler proves safety, generates code with no bookkeeping

other policies (e.g. reference counting) do

...but can implement new ones if not good enough

other things languages can enforce?

saw: enforcing no use-after-free

lots of coding conventions we might try to enforce:

code's runtime does not depend on secret data

- secret data has different type
- variable time operations prohibited with secret data

sensitive data not passed to wrong place

- sensitive data has different type
- assignment to wrong places is a type error

code has bounded runtime

- langauge prohibits not unbounded loops, recursion, etc.

some constant time ideas

FaCT: A DSL for Timing-Sensitive Computation

Sunjay Cauligi[†] Gary Soeller[†] Brian Johannesmeyer[†] Fraser Brown[⋆] Riad S. Wahby[⋆]

John Renner[†] Benjamin Grégoire[♠] Gilles Barthe[♣♦] Ranjit Jhala[†] Deian Stefan[†]

[†]UC San Diego, USA [⋆]Stanford, USA [♠]INRIA Sophia Antipolis, France
[♣]MPI for Security and Privacy, Germany [♦]IMDEA Software Institute, Spain

CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem

CONRAD WATT, University of Cambridge, UK
JOHN RENNER, University of California San Diego, USA
NATALIE POPESCU, University of California San Diego, USA
SUNJAY CAULIGI, University of California San Diego, USA
DEIAN STEFAN, University of California San Diego, USA

FaCT, PLDI 2019; CT-Wasm, POPL 2019

constant-time programming languages

active research area, no consensus on what works best

common approach: separate type for **secret** data

compiler or language virtual machine disallows variable-time operations using secret data

no secret-based array lookup (cache timing varies)

e.g. array[secret_value] $\rightarrow$ compile error (type mismatch)

no secret-based integer division (usually variable speed instruction)

...

explicit operations for any secret to non-secret conversions

backup slides

Rust linked list

not actually a good idea

use `Box<...>` to represent object on the heap

no null, use `Option<Box<...>>` to represent pointer.

Rust linked list (not recommended)

```
struct LinkedListNode {
    value: u32,
    next: Option<Box<LinkedListNode>>,
}

fn allocate_list() -> LinkedListNode {
    return LinkedListNode {
        value: 1,
        next: Some(Box::new(LinkedListNode {
            value: 2,
            next: Some(Box::new(LinkedListNode {
                value: 3,
                next: None
            }))
        }))
    }
}
```

why the box? (1)

```
struct LinkedListNode { // ERROR
    value: u32,
    next: Option<LinkedListNode>,
}

// error[E0072]: recursive type `LinkedListNode` has infinite size
```

why the box? (2)

```
struct LinkedListNode { // ERROR
    value: u32,
    next: Option<&LinkedListNode>,
}
// error[E0106]: missing lifetime specifier
//  --> src/main.rs:48:18
//     |
// 48  |     next: Option<&LinkedListNode>,
//     |                  ^ expected lifetime parameter
```