Changelog

Corrections made in this version not in first posting:

12 April 2017: slide 15: correct arrow from B-freed to D-freed

12 April 2017: slide 42: correct phrasing on what is borrowed

fuzzing assignment

target: a program that reindents C source files
- from FreeBSD, modified to run on Linux
- original uses sandboxing — so probably not actual security iussues

tool: american fuzzy lop

along with AddressSanitizer — find crashes
- probably buffer overflows

crashes are easy to find — so won't have to fuzz for long
- but in real scenario would run fuzzer for hours/days
- (or until coverage is very good)

alternative techniques

memory error detectors — to help with software **testing**

- reliably detect single-byte overwrites, use-after-free
- bitmap for every bit of memory — should this be accessed
- **not** suitable for stopping exploits
- examples: AddressSanitizer, Valgrind MemCheck

automatic testing tools — run programs to trigger memory bugs

static analysis — analyze programs and either

- find likely memory bugs, or
- prove absence of memory bugs

better programming languages

other program analysis

other design points than symbolic execution

higher level of abstractions:

can avoid path explosion

ignore irrelevant parts of the program

focus on finding common bug patterns

the easy case (1)

```
int vulnerable() {
    int buffer[100];
    ...
    return buffer[100];
}
```

the easy case (2)

```
void vulnerable(char *input) {
    char buffer[100];
    strcpy(buffer, input);
}
```

the easy case (3)

```
void vulnerable() {
    ...
    ...
    if (some_condition()) {
        free(some_variable);
        ...
        use(some_variable->data);
    }
}
```

the somewhat easy case (1)

```
void vulnerable(int *input, int num_items) {
    int buffer[100];
    int size = min(num_items * sizeof(int), sizeof(buffer) * s
    memcpy(buffer, input, size);
}
```

the somewhat easy case (2)

```
int vulnerable(char *input) {
    char buffer[100] = ...;
    return buffer[input[0]];
}
```

complete versus sound

complete versus sound

- complete: no false positive
  - says error — actually a memory error
- sound: no false negative
  - says no error — actually no memory errors

many real analyzers neither complete nor sound

sometimes assisted by programmer annotations

- e.g. “this pointer should not be null”

a brief look

we won't talk fully about program analysis

more general/flexible than symbolic execution

can avoid analyzing irrelevant parts of the program

can generalize to make analysis practical

one idea: simpler models

model for use-after-free, pointer is:

- allocated
- freed

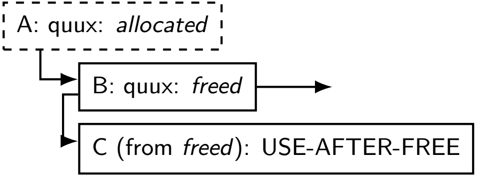
just track this logical state for each pointer

ignore everything else

assume all code is reachable

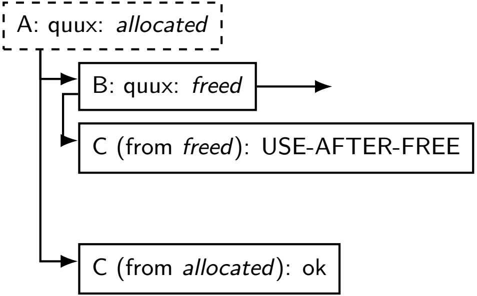
checking use-after-free (1)

```c
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ...
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ...
}
```



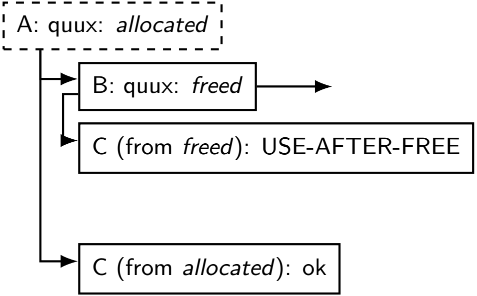
checking use-after-free (1)

```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ...
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ...
}
```



checking use-after-free (1)

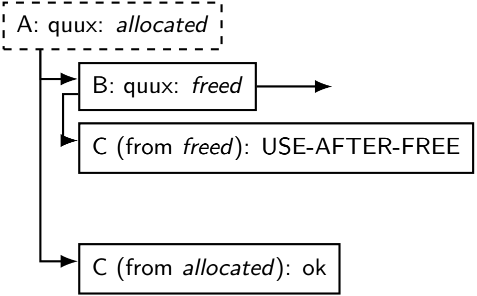
```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ...
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ...
}
```



static analysis can give warning — probably bad

checking use-after-free (1)

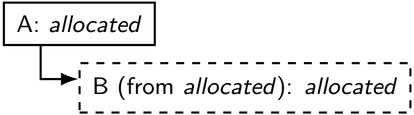
```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ...
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ...
}
```



static analysis can give warning — probably bad
but maybe `Complex(foo) == !Complex(bar)`

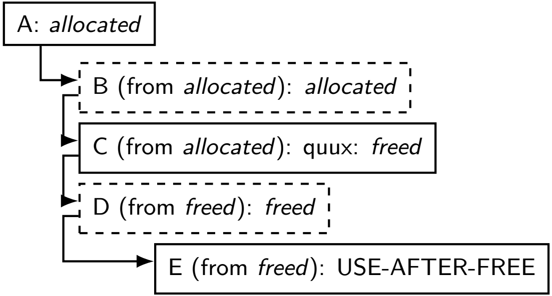
checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



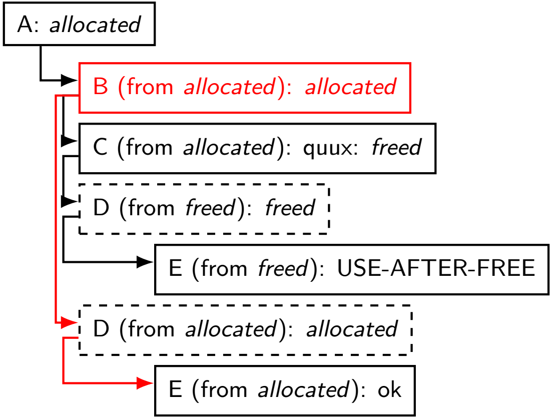
checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



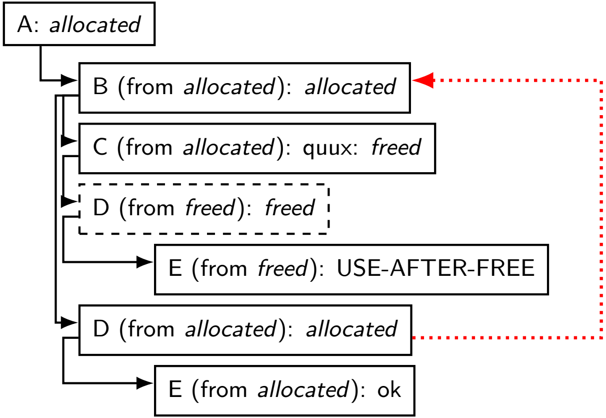
checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



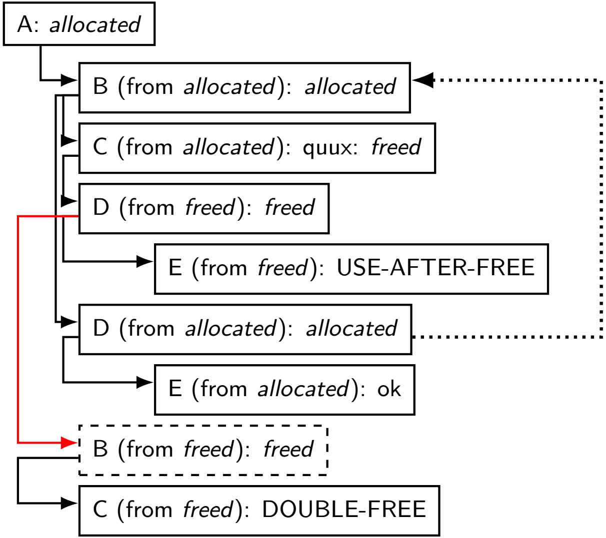
checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



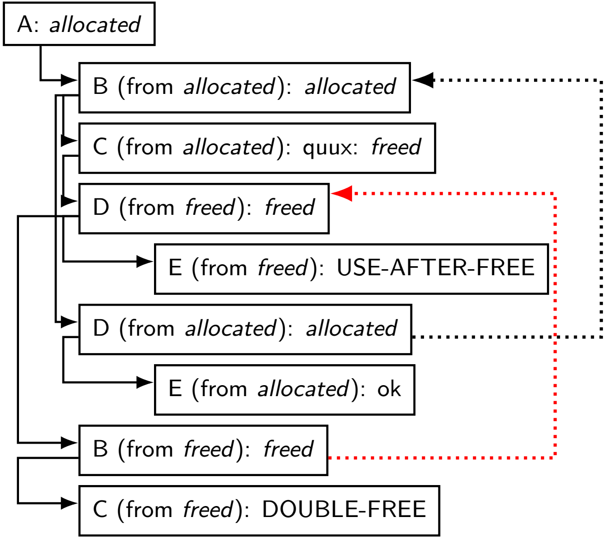
checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



checking use-after-free (2)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (someFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
}
```



static analysis over symbolic execution

can deal with hard cases by being imprecise

- can't try every path? generalize
- generate false positives and/or false negatives

can deal with hard cases with *annotations*

- "I promise this value is allocated here"
- "I promise this value is freed here"

avoiding false positives

after finding error, search for program path to triggers it

good time to use symbolic-execution-like techniques

can use heuristics to decide path is too unlikely

static analysis practicality

good at finding some kinds of bugs

array out-of-bounds probably not one

excellent for “bug patterns” like:

```
struct Foo* foo;
...
foo = malloc(sizeof(struct Bar));
```

false positive rates are often 20+% or more

somet tools assume lots of annotations

static analysis tools

Coverity, Fortify — commerical static analysis tools

Splint — unmaintained?

- written by David Evans and his research group in the late 90s/early 00s

FindBugs (Java)

clang-analyzer — part of Clang compiler

Microsoft's Static Driver Verifier — required for Windows drivers:

- mostly checks correct usage of Windows APIs

alternative techniques

memory error detectors — to help with software **testing**
- reliably detect single-byte overwrites, use-after-free
- bitmap for every bit of memory — should this be accessed
- **not** suitable for stopping exploits
- examples: AddressSanitizer, Valgrind MemCheck

automatic testing tools — run programs to trigger memory bugs

static analysis — analyze programs and either
- find likely memory bugs, or
- prove absence of memory bugs

better programming languages

better programming languages

get better information from programmer

ideal: eliminate memory errors without making program slower

some overlap with static analysis

- information used to prove no memory errors

example: “smart pointer” libraries for C++

example: Rust

safety rules

rule for avoiding bounds errors in C

- always pass around array buffers with size
- always check size

this is easy to enforce at compile-time

- Java does it

but problem: what about when I don't want overhead of checking?

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

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 (1)

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

simple Rust syntax (2)

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

simple Rust syntax (3)

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

fn get_example_student() -> Student {
    return 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(n
    let mut re
    let mut in
    while index <= number {
        result *= index;
        index = index + 1;
    }
    return result;
}
```

“input” is a mutable variable

type automatically inferred as i32 (32-bit int)

Rust references

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

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

    let z: &u32 = &x;

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

Rust example

```
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

```
use std::io;

fn main() {
    println!("Ent
    
    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

```rust
use std::io;

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

    let mut input = 
    // 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);
}
```

pass mutable reference to input

Rust example

```
use std::io;

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

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

    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

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 (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);
    // **moves** vector into mysum()
         // philosophy: no implicit expensive copies

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

Rust objects and ownership (1)

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

    }
    ret
}

fn foo
    let
    let
    //
        // philosophy: no implicit expensive copies

    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> = 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]);
}
```

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<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]);
}
```

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

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**

- must know when object is given back

only **modify** object when exactly one user

- owner or exclusive borrower

borrowing

```
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?

```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?

```
int *dangling_pointer() {
```

```
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
```

```
}
```

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**

- must know when object is given back

only **modify** object when exactly one user

- owner or exclusive borrower

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**

  must know when object is given back

only **modify** object when exactly one user

  owner or exclusive borrower

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
        }
    }
}
```

trying to give away mutable reference

…while the for loop has a reference

data races

Rusts rules around modification built assuming concurrency

idea: multiple processes/threads running at same time might use value

safe policy: all reading *or* only one at a time

if multiple at a time: problems are called “data races”

data races for use-after-free

```
Expand Vec                    | Read Vec
----------------------------------------------------
                              | mov pointer, %rax
                              | ...
mov $100, %rdi                |
call malloc                   |
mov pointer, %rdi             |
mov %rax, pointer             |
call free                     |
                              | ...
                              | mov (%rax), %rax
```

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
- basically same as Vec except one element

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**

- must know when object is given back

only **modify** object when exactly one user

- owner or exclusive borrower

ownership is enough?

what if my program is more complicated than single owner I borrow from?

exercise: when are cases where this doesn't work?

think of data structures you've implemented

exercise: what are other rules to prevent dangling pointers?

escape hatch

Rust lets you avoid compiler's mechanisms

implement your own

unsafe keyword

how Vec is implemented

deep inside Vec

```
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));
            }
        }
    }
    ...
}
```

Rust escape hatch support

escape hatch: make new reference-like types

callbacks on ownership ending (normally deallocation)

choice of what happens on move/copy

alternative rule: reference counting

keep track of number of references

delete when count goes to zero

Rust automatically calls destructor — no programmer effort

Rust implement with Rc type ("counted reference")

Ref Counting Example

```
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: i32,
        studentName: student.name,
        assignmentName: assignment.name,
    })
    student.grades.push(grade.clone())
    assignment.grades.push(grade.clone())
}
```

Rust escape hatch support

escape hatch: make new reference-like types

Rc: Rc<T> acts like &T

callbacks on ownership ending (normally deallocation)

Rc: deallocating Rc<T> decrements shared count

choice of what happens on move/copy

Rc: transferring Rc makes new copy, increments shared count

Rc implementationed (annotated) (1)

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

Rc implementation (annotated) (2)

```
unsafe impl<#[may_dangle] T: ?Sized> Drop for Rc<T> {
    ...
    fn drop(&mut self) { // <-- compilers calls on deallocation
        unsafe {
            let ptr = *self.ptr;

            self.dec_strong(); // <-- decrement reference cont
            if self.strong() == 0 { // if ref count is 0
                // destroy the contained object
                ptr::drop_in_place(&mut (*ptr).value);
                ...
            }
        }
    }
    ...
}
```

other policies Rust supports

RefCell — borrowing, but check at runtime, not compile-time

detect at runtime if used while already used

internally: destructo call when returned object goes out of scope

Weak — reference-counting, but don't contribute to count

detect at runtime if used with count $= 0$

...

other policies Rust supports

RefCell — borrowing, but check at runtime, not compile-time

detect at runtime if used while already used

internally: destructo call when returned object goes out of scope

Weak — reference-counting, but don't contribute to count

detect at runtime if used with count $= 0$

...

shared idea of all policies

normal users only use 'safe' interfaces

advanced users can make new 'safe' interfaces

- find something that makes their use-case works

no overhead if compiler can prove lifetimes

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

plans for the future

command injection bugs

web browser security

I am flexible — different topics you want?

- sandboxing (another mitigation)
- synchornization-related security bugs
- static analysis?
- new mitigations proposed in research?
- other?

one idea: bounding values

model values as abstraction

somewhat similar to *types*

example: lowest/highest value every integer can contain

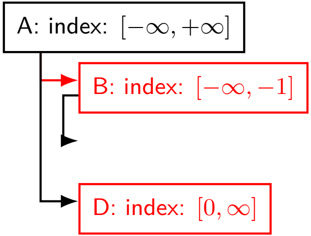
oversimplified array bounds

```
int data[100];
int getItem() {
    int index = getFromUser();
    // A
    if (index < 0) {
        // B
        index = 0;
        // C
    } else {
        // D
        if (index > 99) {
            // E
            index = 99;
            // F
        } else {
            // G
        }
        // H
    }
    // I
    return data[index];
}
```

A: index: $[-\infty, +\infty]$

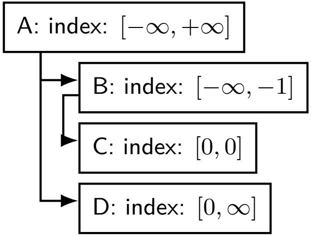
oversimplified array bounds

```
int data[100];
int getItem() {
    int index = getFromUser();
    // A
    if (index < 0) {
        // B
        index = 0;
        // C
    } else {
        // D
        if (index > 99) {
            // E
            index = 99;
            // F
        } else {
            // G
        }
        // H
    }
    // I
    return data[index];
}
```



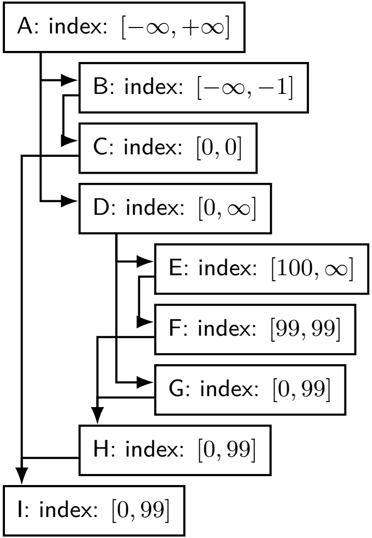
oversimplified array bounds

```
int data[100];
int getItem() {
    int index = getFromUser();
    // A
    if (index < 0) {
        // B
        index = 0;
        // C
    } else {
        // D
        if (index > 99) {
            // E
            index = 99;
            // F
        } else {
            // G
        }
        // H
    }
    // I
    return data[index];
}
```



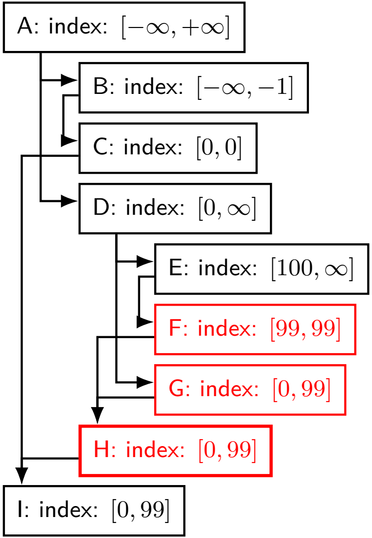
oversimplified array bounds

```
int data[100];
int getItem() {
    int index = getFromUser();
    // A
    if (index < 0) {
        // B
        index = 0;
        // C
    } else {
        // D
        if (index > 99) {
            // E
            index = 99;
            // F
        } else {
            // G
        }
        // H
    }
    // I
    return data[index];
}
```



oversimplified array bounds

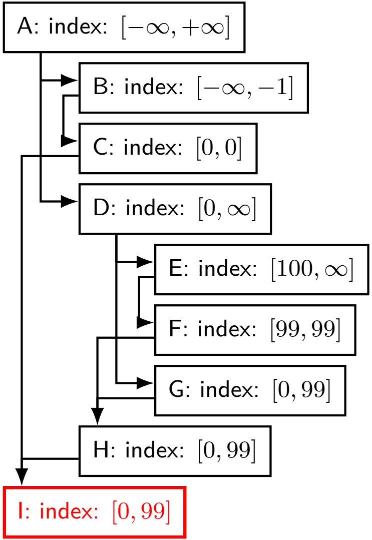
```
int data[100];
int getItem() {
    int index = getFromUser();
    // A
    if (index < 0) {
        // B
        index = 0;
        // C
    } else {
        // D
        if (index > 99) {
            // E
            index = 99;
            // F
        } else {
            // G
        }
        // H
    }
    // I
    return data[index];
}
```



conservative rule:
take union of ranges

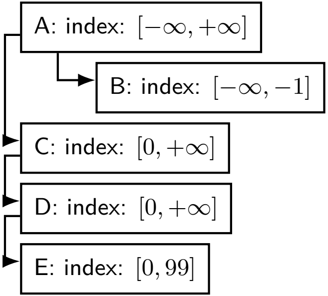
oversimplified array bounds

```
int data[100];
int getItem() {
    int index = getFromUser();
    // A
    if (index < 0) {
        // B
        index = 0;
        // C
    } else {
        // D
        if (index > 99) {
            // E
            index = 99;
            // F
        } else {
            // G
        }
        // H
    }
    // I
    return data[index];
}
```



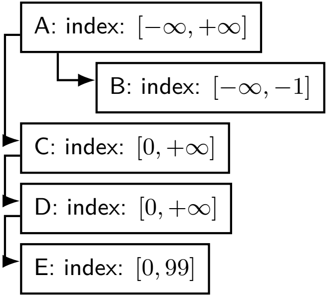
oversimplified array bounds (2)

```
int data[100];
int getItem() {
    int index = getFromUser();
    int functionToUse = getFromUser();
    // A
    if (index < 0) {
        // B
        return ERROR;
    }
    // C
    switch (functionToUse) {
        ...
        // unknown, hard to understand code
        // that doesn't change index
        // and might not even terminate
        ...
    }
    // D
    if (index > 100) {
        index = 99;
    }
    // E
```



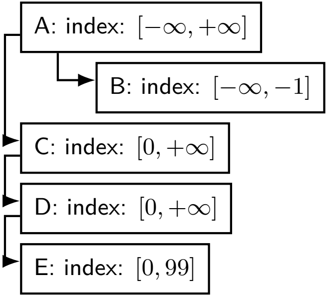
oversimplified array bounds (2)

```
int data[100];
int getItem() {
    int index = getFromUser();
    int functionToUse = getFromUser();
    // A
    if (index < 0) {
        // B
        return ERROR;
    }
    // C
    switch (functionToUse) {
        ...
        // unknown, hard to understand code
        // that doesn't change index
        // and might not even terminate
        ...
    }
    // D
    if (index > 100) {
        index = 99;
    }
```



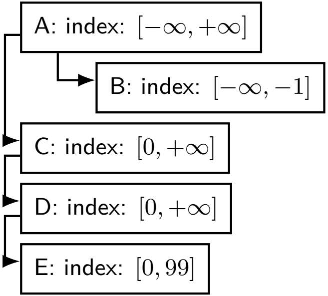
oversimplified array bounds (2)

```
int data[100];
int getItem() {
    int index = getFromUser();
    int functionToUse = getFromUser();
    // A
    if (index < 0) {
        // B
        return ERROR;
    }
    // C
    switch (functionToUse) {
        ...
        // unknown, hard to understand code
        // that doesn't change index
        // and might not even terminate
        ...
    }
    // D
    if (index > 100) {
        index = 99;
    }
```



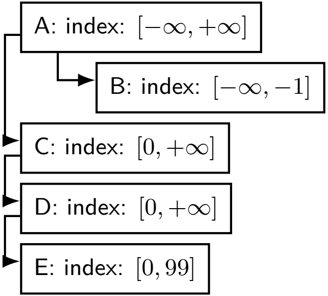
oversimplified array bounds (2)

```
int data[100];
int getItem() {
    int index = getFromUser();
    int functionToUse = getFromUser();
    // A
    if (index < 0) {
        // B
        return ERROR;
    }
    // C
    switch (functionToUse) {
        ...
        // unknown, hard to understand code
        // that doesn't change index
        // and might not even terminate
        ...
    }
    // D
    if (index > 100) {
        index = 99;
    }
```



oversimplified array bounds (2)

```
int data[100];
int getItem() {
    int index = getFromUser();
    int functionToUse = getFromUser();
    // A
    if (index < 0) {
        // B
        return ERROR;
    }
    // C
    switch (functionToUse) {
        ...
        // unknown, hard to understand code
        // that doesn't change index
        // and might not even terminate
        ...
    }
    // D
    if (index > 100) {
        index = 99;
    }
    // E
```

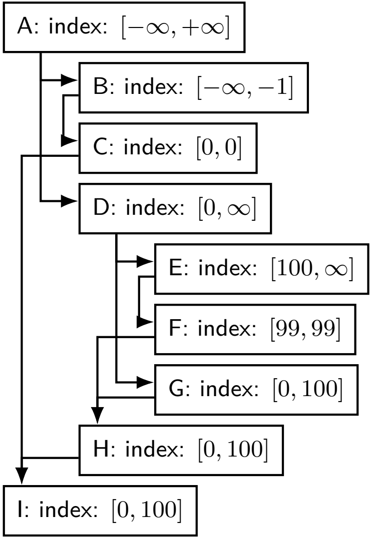


conserva

take unio

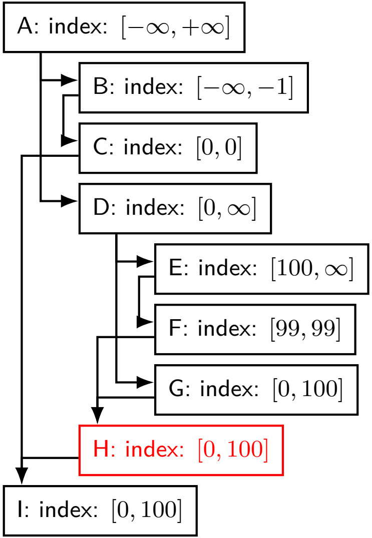
oversimplified array bounds (bug)

```
int data[100];
int getItem() {
    int index = getFromUser();
    // A
    if (index < 0) {
        // B
        index = 0;
        // C
    } else {
        // D
        if (index > 100) {
            // E
            index = 99;
            // F
        } else {
            // G
        }
        // H
    }
    // I
    return data[index];
}
```



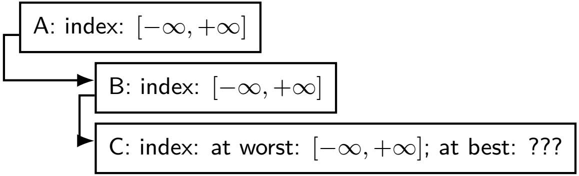
oversimplified array bounds (bug)

```
int data[100];
int getItem() {
    int index = getFromUser();
    // A
    if (index < 0) {
        // B
        index = 0;
        // C
    } else {
        // D
        if (index > 100) {
            // E
            index = 99;
            // F
        } else {
            // G
        }
        // H
    }
    // I
    return data[index];
}
```



too hard array bounds

```
int data[100];
int defaultIndex; // range: [0, 99]
int getItem() {
    int index = getFromUser();
    int useDefault = 0;
    // A
    if (index < 0 || index > 99) {
        useDefault = 1;
    }
    // B
    if (useDefault) {
        return data[defaultIndex];
    } else {
        // C
        return data[index];
    }
}
```



too hard array bounds

```
int data[100];
int defaultIndex; // range: [0, 99]
int getItem() {
    int index = getFromUser();
    int useDefault = 0;
    // A
    if (index < 0 || index > 99) {
        useDefault = 1;
    }
    // B
```

analysis too simple to handle this code "path-sensitivty"

```
        // C
        return data[index];
    }
}
```

A: index: $[-\infty, +\infty]$

B: index: $[-\infty, +\infty]$

C: index: at worst: $[-\infty, +\infty]$; at best: ???

diversion: Option<T>

Rust has no null

but has Option<T> type

if x is a variable of type T:

  Some(x)
  None (like Java null)

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
```