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 (1)

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

simple Rust syntax (2)

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

simple Rust syntax (3)

```rust
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(number: i32) -> i32 {
    let mut r
    let mut i
    while ind
        resul
        index = index + 1;
    }
    return result;
}
```

“result” 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, z);
}
```

Rust example

```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

```
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

```
use std::io;

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

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

    io::std

    // 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 {
        total += value
    }
    re
}

fn foo
    le
    le
    //
```

```
  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
```

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

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?

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

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

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

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

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() {
    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
```

applying rules (1)

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)
```

```
int x = 42;          // (1)
int *p = &x;         // (2).
*p = 10;             // (3)'
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 (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

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);
```

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);
}
```

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

would be okay if giving away non-mutable reference

why compiler distinguishes mutable/non-mutable references

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

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

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

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

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

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 policies Rust supports

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

detect at runtime if used while already used

internally: destructor call when returned object goes out of scope

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

detect at runtime if used with count $= 0$

Mutex — with multicore, enforce one user at a time by waiting

...

other policies Rust supports

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

detect at runtime if used while already used

internally: destructor call when returned object goes out of scope

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

detect at runtime if used with count $= 0$

Mutex — with multicore, enforce one user at a time 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?

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
```