

WHICH LANGUAGE SHOULD WE STUDY IF OUR OBSESSION IS CONTROL (EXPLICIT AND IMPLICIT) AND PERFORMANCE?

Professor Ken Birman CS4414/5416 Lecture 3

TODAY'S MATERIAL WILL NOT APPEAR ON PRELIM EXAMS

In fact, as will become clear, Ken is not really fond of Rust.

But there is a White House report from 2023 urging companies to consider switching to this language, and some people like it

On prelims, we don't want you to just "echo" Ken's views, so this lecture won't be tested. You are responsible for all other lectures, including slides we covered very quickly or even skipped over.

FEATURES OF THE FASTEST WORD COUNTS

Multiple threads, including one for directory scan and file opening.

Avoid creating extra objects

Each thread has its own counting data structure: merge them at the end, using the special sort order we supplied.

Efficient ways of actually reading the data from the files, that match with what the O/S is good at (encourages prefetching)





It was in our list of candidate languages!

And is unquestionably a hot debate topic now, especially because the US government encouraged switching to Rust for improved program safety in a 2024 study.

Looks a lot like C++ and shares the same optimizing compiler backend as the Clang version of C++ 23.

IDEA MAP FOR TODAY

Many attacks exploit weak memory protection

C++ is built on C... and C is fundamentally unsafe

Rust idea

How well does it work?

What are the criticisms of Rust?

Today's entire lecture borrows heavily from a lecture opensourced by Prof. Wu-chang Feng at Portland State University, where they use Rust as their systems programming language



START AT THE END...

Obviously, this course uses C++ 23



In fact so are Rust, Java, Julia, many more



C++ IS DEEPLY CONNECTED TO C

The C++ optimizing compiler "transforms" your C++ code into a C program for compilation. It precomputes many things, yet the bottom line is that C++ reduces to "plain old C."

Thus, many weaknesses of C translate to vulnerabilities in C++

In recitation we learn about runtime tools that help compensate for this, such as Valgrind, the gdb debugger, gprof and more.

YET... C++ IS MORE THAN C

In fact the compiler literally uses different rules when compiling a .cpp file versus a .c file.

We consider C to be an unsafe language and we code entirely in C++ both in this course and professionally.

So when people assume that C++ literally inherits C's flaws, this is mostly for an unprofessional style of coding, not for C++ 23.

WHY C BECAME POPULAR (THE GOOD PARTS)

Efficient code especially in resource-constrained environments

Direct control over hardware such as network interfaces and GPUs

Performance over safety

- Memory managed in a controllable way that we will teach you
- No periodic garbage collection (pointers don't become invalidated)
- Most important to advanced programmers: total control

BUT...

Even with this help, both C and C++ make it very easy to make mistakes involving misuse of pointers.

C also has issues with type coercion (C++ fixes them!):

- Integer promotion/coercion errors (where the code specifies the size of an integer, but then uses it in a way inconsistent with the size)
- Unsigned vs. signed errors (in C, conversions back and forth are legal and won't even trigger a warning... C++ will warn)
- Integer casting errors (easy to misunderstand the rules)

AND... IN BOTH C AND C++...

Memory pointer errors are very easy to make

- Dereferencing a null pointer
- Buffer overflows, out-of-bound access (no array-bounds checking)
- > Format string errors in printf or std::cout
- Dynamic memory errors (Memory leaks, use-after-free (dangling pointer), double free of a pointer)

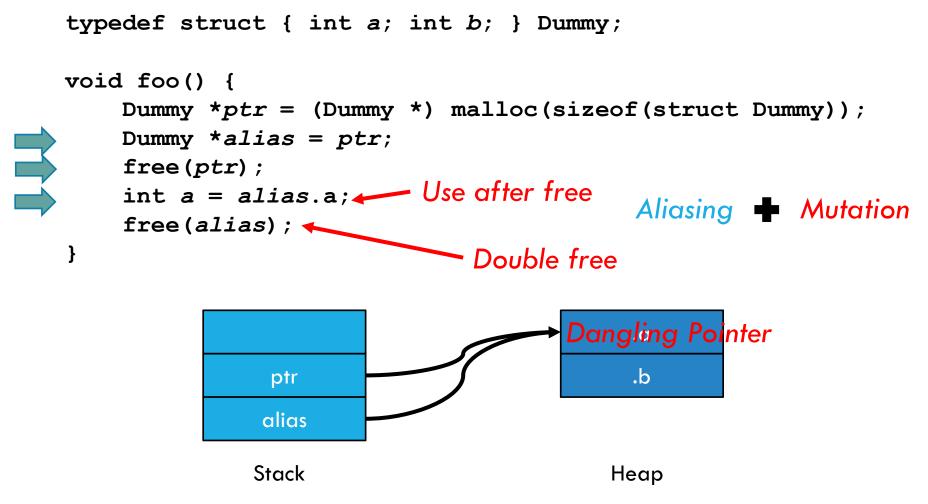
All of these can cause software crashes and security vulnerabilities.

EXAMPLE: C-STYLE POINTERS IN C OR C++

Lightweight, low-level control of memory

```
typedef struct { int a; int b; } Dummy;
                                        Precise memory layout
void foo() {
    Dummy *ptr = (Dummy *) malloc(sizeof(struct Dummy));
    ptr->a = 2048;
                                   Lightweight reference
    free (ptr);
                         Destruction
                                        a = 2048
                                           .b
               ptr
               Stack
                                          Heap
```

... ISSUE: ERRORS GO UNDETECTED



THESE ISSUES ARE ALL SOLVED BY MANAGED LANGUAGES, BUT THEY ARE OFTEN SLOW

Java, Python, Ruby, C#, Scala, Go...

All of them restrict direct access to memory with run-time management of memory via garbage collection. But...

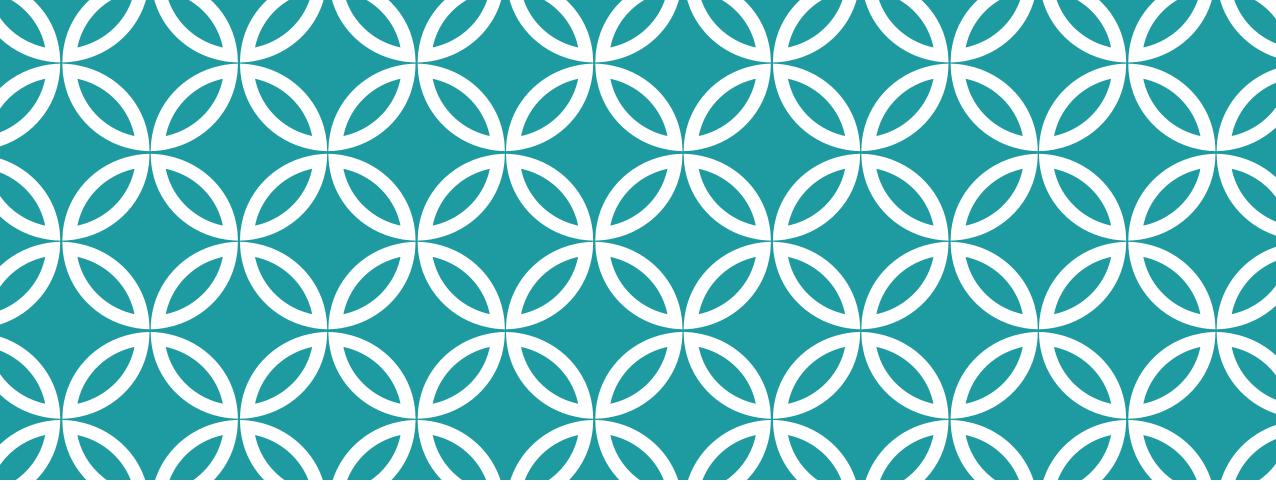
- > They pay a high overhead for tracking pointer use and array-index safety
- Performance can be unpredictable due to GC (bad for real-time systems)
- Limited concurrency (global interpreter lock typical)
- In some cases a VM is required (like for Python)
- Need more memory and CPU power (i.e. not bare-metal)

REQUIREMENTS FOR SYSTEM PROGRAMMING

The language must be fast and have minimal runtime overhead

Developer should be able to visualize every action the entire system is performing and gain control over everything.

We often <u>need</u> direct memory access, but wish it was memory-safe



Rust is named after <u>a fungus</u> that is robust, distributed, and parallel. It is also a subsequence of "robust".



RUST



RUST

From the official website (http://rust-lang.org):

Rust is a true system programming language.

- No runtime requirement (runs fast)
- Control over memory allocation/destruction.
- Guarantees memory safety

Created by Monzilla to address severe memory leakage and corruption bugs in Firefox. First stable release in 5/2015

RUST OVERVIEW

Performance, as with C or C++, similar look and feel as C++

Rust compiles to object code for bare-metal performance

Supports memory safety

- Programs cannot dereference pointers that have been freed
- Out-of-bound array accesses not allowed

Relatively low overhead

- Compiler checks to make sure rules for memory safety are followed
- Zero-cost abstraction in managing memory (i.e. no garbage collection)

RUST OVERVIEW

How is this done? Like C++, much occurs at compile time!

- Advanced type system
- Novel language features to prevent memory and pointer issues

But there is a cost

- Cognitive cost to programmers who must think more about rules for using memory and references as they program
- > Less control of the kind needed for ML and HPC programming

MORE ON THIS LAST POINT (1)?

Recall fast word-count. Sagar first turned every file into std::strings, but then discarded most of the strings. A wasteful design.

Ken had a layer that operated directly on data in character buffers. This uses pointers to ascii chars in arrays.

In Rust, pointer logic like what Ken did is much harder to implement because the compiler perceives it as unsafe. Inserting safety checks for every pointer dereference is similarly wasteful.

MORE ON THIS LAST POINT (2)?

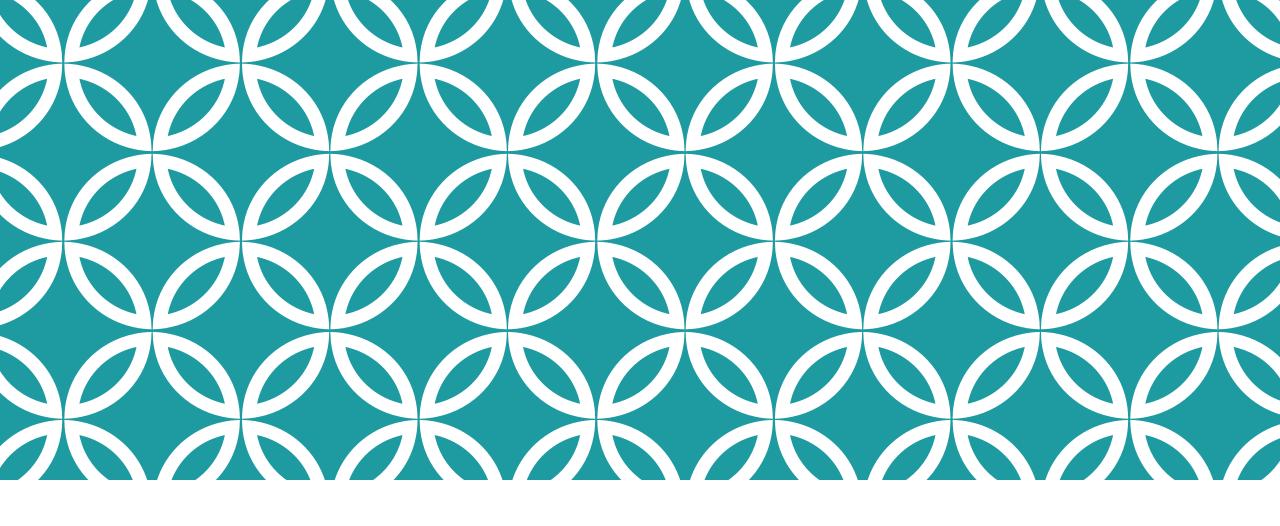
To maximize parallelism, we often need to control layout of data structures in memory **and** ensure that the alignment rules for data would be "visible" to the C++ compiler, and used templates + inlining + constexpr to preserve code elegance.

In Rust it can be hard or impossible to get the same kinds of guarantees. Rust lacks C++'s template "language".

RUST IS ALSO AT ODDS WITH DIRECT-MAPPED GPU MEMORY, DMA AND RDMA

Today's most important accelerators often are directly integrated into the memory system. This includes GPUs and direct memory transfer networking such as RDMA

All forms of direct memory transfers from network, disk or GPU are inherently unsafe, as are shared segments and page remapping: features we will learn about, and that advanced C++ developers often use.



RUST'S TYPE SYSTEM

RUST TYPES LOOK MUCH LIKE C/C++ TYPES

Primitive types

- bool
- char (4-byte unicode)
- > i8/i16/i32/i64/isize
- > u8/u16/u32/u64/usize
- F32/f64

Numeric types specified with width. The Unicode char default might surprise some C or C++ developers.

C TYPES HAVE SOME IDIOSYNCRASIES.

C "overloads" integers to get Booleans. Can create ambiguity: an integer isn't limited to just 0 or 1.

- > True, False, or Fail? 1, 0, -1? Misinterpretations lead to security issues
- Example: In the PHP is a widely used C library for web programming. In it, strcmp returns 0 for both equality *and* failure!

C++ offers a true Boolean type. If you use it, this can't occur. In fact C++ type checking is extremely powerful and strong.

C, C++ ARRAY TYPE

In Rust, arrays stored with their length [T; N]

>Allows for both compile-time and run-time checks on array access via []

```
C
void main() {
   int nums[8] = {1,2,3,4,5,6,7,8};
   for ( x = 0; x < 10; i++ )
      printf("%d\n",nums[i]);
}

Rust

int nums[8] = {1,2,3,4,5,6,7,8};
   for x in 0..10 {
      println!("{}",nums[x]);
   }
}</pre>
```

```
8
thread 'main' panicked at 'index out of bounds: the len is 8 but the index is 8', note: Run with `RUST_BACKTRACE=1` for a backtrace.

Program ended.
```

RUST AND BOUNDS CHECKING

But...

Checking bounds on every access adds overhead

```
fn main() {
    let nums = vec![1,2,3,4,5,6,7,8];
    for x in 0..10 {
        println!("{}",nums[x]);
    }
}
```

- Arrays typically accessed via more efficient iterators to allow compile time checking, avoid runtime overheads
 - ➤ Can use x86 loop instruction

```
1 fn main() {
2    let nums = vec![1,2,3,4,5,6,7,8];
3 for num in &nums {
4         println!("{}",num);
5    }
6 }
```

RUST IS VERY CAUTIOUS ABOUT COERSIONS

In C code, you can cast any integer to an unsigned integer of the same size, or back. C++ doesn't allow this... except when compiling C

-1 casts to 2147483648 (largest uint32). Is this what a developer intended? The C specification just says this is an "undefined" cast!

A European rocket once veered wildly off course, then exploded because one module used unsigned int, but another used signed int.

C has confusing implicit integer casts and promotion

```
-1 > 0U
2147483647U < -2147483648
```

Rust's type system prevents such comparisons

```
void main() {
    unsigned int a = 4294967295;
    int b = -1;
    if (a == b)
        printf("%u == %d\n",a,b);
}

mashimaro <~> 9:44AM % ./a.out
4294967295 == -1
```

Same or different?

```
void main() {
    char a=251;
    unsigned char b = 251;
    printf("a = %x\n", a);
    printf("b = %x\n", b);

if (a == b)
        printf("Same\n");
    else
        printf("Not Same\n");
}

mashimaro<> % ./a.out
    a = fffffffb
    b = fb
Not Same
```

```
fn main() {
    let a:i8 = 251;
    let b:u8 = 251;

    if a == b {
        println!("Same");
    } else {
        println!("Not Same");
    }
}

rustc 1.15.1 (021bd294c 2017-02-08)
error[E0308]: mismatched types
    --> <anon>:5:13
    |
5 | if a == b {
    error: aborting due to previous error
```

201 > 200?

```
#include <stdio.h>
void main() {
    unsigned int ui = 201;
    char c=200;
    if (ui > c)
        printf("ui(%d) > c(%d)\n",ui,c);
    else
        printf("ui(%d) < c(%d)\n",ui,c);
}

mashimaro <~> 12:50PM % ./a.out
ui(201) < c(-56)</pre>
```

```
fn main() {
    let ui:u32 = 201;
    let c:i8 = 200;
    if ui > c {
        println!("ui({})) > c({})",ui,c);
    } else {
        println!("ui({})) < c({})",ui,c)
    }
}

rustc 1.15.1 (021bd294c 2017-02-08)
error[E0308]: mismatched types
    --> <anon>:4:13
    |
4 | if ui > c {
```

In Rust, casting is allowed via the "as" keyword

- Follows similar rules as C
- ▶ But, warns problem before performing the promotion with sign extension.
 C++ does this too, but because C code can be pulled in so easily...

```
#include <stdio.h>
void main() {
  char c=128;
  unsigned int uc;
  uc = (unsigned int) c;
  printf("%x %u\n",uc, uc);
}

mashimaro <~> 1:24PM % ./a.out
ffffff80 4294967168
```

```
fn main() {
    let c:i8 = 128;
    let uc:u32 = c as u32;
    println!("uc = {}", uc);
}

rustc 1.15.1 (021bd294c 2017-02-08)
warning: literal out of range for i8, #
default
    --> <anon>:2:16
    |
2    | let c:i8 = 128;
uc = 4294967168
```

C has issues with unchecked underflow and overflow

> Silent wraparound in C caught by runtime check in Rust

```
void main() {
    unsigned int a = 4;
    a = a - 3;
    printf("%u\n",a-2);
}
mashimaro <~> 9:35AM % ./a.out
4294967295
```

```
fn main() {
   let mut a:u32 = 4;
   a = a - 3;
   println!("{}", a - 2);
}

rustc 1.15.1 (021bd294c 2017-02-08)
thread 'main' panicked at 'attempt to subtract with overflow',
   stack backtrace:
```

EXAMPLE: A FAMOUS C VULNERABILITY

DNS parser vulnerability discussed in B&O, Chapter 2

count read as byte, then count bytes concatenated to nameStr

```
char *indx;
int count;
char nameStr[MAX LEN]; //256
                                                          test.jim.com
memset(nameStr, '\0', sizeof(nameStr));
indx = (char *) (pkt + rr offset);
count = (char)*indx;
while (count) {
                                              What if count = 128?
  (char *)indx++;
                                              Sign extended then used in strncat
  strncat(nameStr, (char *)indx, count)
  indx += count;
                                                     Type mismatch in Rust
  count = (char)*indx;
  strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
nameStr[strlen(nameStr)-1] = ' \setminus 0';
          char *strncat(char *dest, const char *src, size t n);
```

ANOTHER C VULNERABILITY

2002 FreeBSD getpeername () bug (B&O Ch. 2)

- Kernel code to copy hostname into user buffer
 - >copy from kernel() call takes signed int for size from user
 - >memcpy call uses unsigned size_t
- ➤ What if adversary gives a length of "-1" for his buffer size?

```
#define KSIZE 1024
char kbuf[KSIZE]
void *memcpy(void *dest, void *src, size_t n);

int copy_from_kernel(void *user_dest, int maxlen){
    /* Attempt to set len=min(KSIZE, maxlen) */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

Unauthorized kernel memory copied out</pre>
```

RUST'S OWNERSHIP & BORROWING

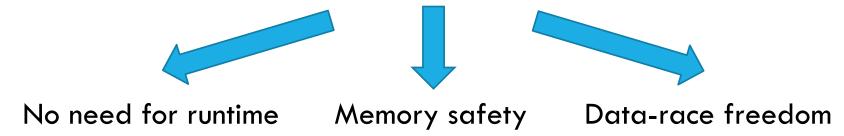


Compiler enforced:

Every resource has a unique owner.

Others can borrow the resource from its owner (e.g. create an alias) with restrictions

Owner cannot free or mutate its resource while it is borrowed.



OWNERSHIP AND LIFETIMES

There can be only one "owner" of an object at a time.

- > When the "owner" of the object goes out of scope, its data is automatically freed
- Can not access object beyond its lifetime (checked at compile-time)

Stack

```
struct Dummy { a: i32, b: i32 }
                               Memory allocation
fn foo() {
    let mut res = Box::new(Dummy {
                                     a: 0,
                                     b: 0
                                 });
    res.a = 2048;
          Resource owned by res is freed automatically
                           owns
                                        a = 2048
                                          .b = 0
               res
                                          Heap
```

ASSIGNMENT CHANGES OWNERSHIP

```
//#[derive(Clone)]
struct Point { x: i32, y: i32 }

fn main() {
    let a = Point { x: 1, y: 2};
    let b = a;

println!("{}, {}", a.x, a.y);
}
```

http://is.gd/pZKiBw

OWNERSHIP TRANSFERS IN FUNCTION CALLS

```
struct Dummy { a: i32, b: i32 }
fn foo() {
    let mut res = Box::new(Dummy {
                       a: 0,
                       b: 0
                   });
                                            Compiler Error. If you plan to
    take(res);
    println ("res.a = {}", res.a); 		 use res again, must employ "borrow"
                                            semantics, not "move" semantics!
             Ownership is moved from res to arg
fn take(arg: Box<Dummy>) {
      arg is out of scope and the resource is freed automatically
```

MUTABILITY: A STATIC FORM OF LOCKING

By default, Rust variables are immutable (read only)

Usage checked at compile time

mut is used to declare a resource as mutable.

```
1 fn main() {
2    let a: i32 = 0;
3    a = a + 1;
4    println!("{}" , a);
5 }
http://is.gd/OQDszP
```

error: aborting due to previous error

```
fn main() {
    let mut a: i32 = 0;
    a = a + 1;
    println!("{}" , a);
}

rustc 1.14.0 (e8a012324 2016-12-16)
1
Program ended.
```

BORROWING

You cannot borrow mutable reference from immutable object

Or mutate an object immutably borrowed

You cannot borrow more than one mutable reference (to support atomicity)

You can borrow an immutable reference many times

There cannot exist a mutable reference and an immutable one simultaneously (removes race conditions)

The lifetime of a borrowed reference should end before the lifetime of the owner object does (removes use after free)

BORROWING EXAMPLE (&)

You cannot borrow mutable reference from immutable object

BORROWING EXAMPLE (&)

You cannot mutate an object immutably borrowed

```
struct Dummy { a: i32, b: i32 }
fn foo() {
    let mut res = Box::new(Dummy{
                      a: 0,
                      b: 0
                  });
    take(&res);
    res.a = 2048;
        Resource is returned from arg to res
           Resource is immutably borrowed by arg from res
fn take(arg: &Box<Dummy>) { Compiler Error: Cannot mutate via
    arg.a = 2048; 
an immutable reference
```

BORROWING EXAMPLE (&MUT)

You cannot borrow more than one mutable reference

```
struct Dummy { a: i32, b: i32 }
                                               Mutation
fn foo() {
    let mut res = Box::new(Dummy\{a: 0, b: 0\});
                      Mutably borrowed by arg from res but returned when
    take(&mut res);
    res.a = 4096;
                       take completes.
    let borrower = &mut res; Multiple mutable borrowings
                                 are disallowed
             Returned from arg to res
fn take(arg: &mut Box<Dummy>) {
    arg.a = 2048;
```

IMMUTABLE, SHARED BORROWING (&)

You can borrow more than one immutable reference

• But, there cannot exist a mutable reference and an immutable one simultaneously

USE-AFTER FREE IN C OR C++

If these calls are far away from each other, this bug can be very hard to find.

RUST PREVENTS THIS!

The lifetime of a borrowed reference should end before the lifetime of the owner object does

USE AFTER FREE CAUGHT BY RUST AT COMPILE-TIME

Unique ownership, borrowing, and lifetime rules easily enforced

```
fn main() {
   // This binding lives in the main function
   let name = String::from("Hello world!");
   let mut name ref = &name;
       let newname = String::from("Goodbye!");
       name ref = \&newname;
   println!("name is {}", &name ref);
rustc 1.15.1 (021bd294c 2017-02-08)
error: `newname` does not live long enough
  --> <anon>:8:5
              name ref = \&newname;
                            ----- borrow occurs here
          println!("name is {}", &name ref);
10 | }
     - borrowed value needs to live until here
error: aborting due to previous error
```

DANGLING POINTER IN C

```
Famous scoping issues example (B&O Ch 3, Procedures)
  int* func(int x) {
    int n;
    int *np;
    int *np;
    n = x;
    np = &n;
    return np;
}

Local variable is allocated in stack,
    a temporal storage of function.

Reference returned, but variable now out
    of scope (dangling pointer)
```

What does **np** point to after function returns?

What happens if NP is dereferenced after being returned?

CAUGHT BY RUST AT COMPILE-TIME

Ownership/Borrowing rules ensure objects are not accessed beyond lifetime

http://is.gd/3MTsSC

SEEMS LIKE RUST WINS EVERY TIME! BUT IT ISN'T SO SIMPLE!



Rust

more control, more safety

RUST OWNERSHIP AND BORROWING CAN BE ANNOYING!

```
fn main() {
          let mut v = vec![];
                                        v is an owner of the vector
          v.push("Hello");
          let x = &v[0];
                                        By taking a reference to v[0], x borrows the vector from v
          v.push("world");
                                        now v cannot modify the vector
                                        because it lent the ownership to x
          println!("{}", x);
10
```

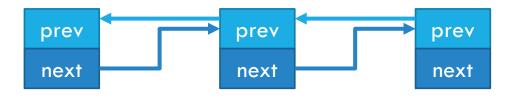
http://is.gd/dEamuS

CONCURRENCY & DATA-RACE FREEDOM

MUTABLY SHARING

Mutably sharing is *inevitable* in the real world.

Example: mutable doubly linked list



```
struct Node {
   prev: option<Box<Node>>,
   next: option<Box<Node>>
```

We require two mutable pointers to the middle mode. This is the essential feature of a list!

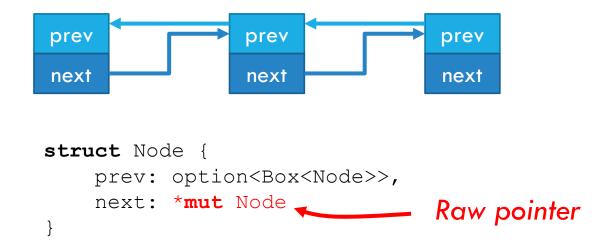
WHY DOES A DOUBLY-LINKED LIST NEED TWO MUTABLE POINTERS?

The problem is that when a node will be removed or added, there are two nodes that need to be updated.

- The prior node will need an updated next pointer.
- The next node needs an updated prior pointer.

So, each node's **next** neighbor has a mutable pointer back to it. And similarly for its **prior** neighbor. Thus: two mutable pointers to a single node! But Rust will disallow this.

PRE-2015 RUST SOLUTION: RAW POINTERS



Rust allows C-style pointers too.... With raw pointers, the compiler does NOT check memory safety of most operations involving that pointer.

If possible, operations wrt. raw pointers should be encapsulated in an unsafe {} syntactic structure.

RUST RAW POINTERS BREAK RUST'S NORMAL MUTABILITY RESTRICTIONS

MORE RECENT ADVICE

Check out this discussion on StackOverflow (very recent).

In summary, they recommend using std::collections, which has a double-linked list.

But there now is a way to create them using something called a Rust RefCell, combined with feature called a std::rc::Rc, which is a form of reference-counter maintained (safely) at runtime.

TALKING TO LIBRARIES: THE FOREIGN FUNCTION INTERFACE (FFI)

You can call code in libraries written in other languages, but the foreign functions are unsafe (e.g. libc calls)

```
extern {
    fn write(fd: i32, data: *const u8, len: u32) -> i32;
}

fn main() {
    let msg = b"Hello, world!\n";
    unsafe {
        write(1, &msg[0], msg.len());
    }
}
```

BIG DEAL?

It is, because libraries are really important.

In ML we rely on tools like LINPACK, MPI, etc. And most GEMM kernels for ML tasks mix C or C++ with GPU or host parallelism.

Even if recoded in Rust they would still be unsafe!

SO THIS TELLS US THAT RUST...

... often yields programs that are still unsafe! Partly due to hacking around shared memory, but often due to using non-Rust libraries

Rust isn't some sort of magic wand. It is more like a tool that we can use to protect ourselves against certain kinds of errors

Monzilla found it super effective! But some hard-core C++ developers who play with Rust find it annoying. And people focused on host parallelism may find the compiled code slow.

WHAT ABOUT SPEED? IS RUST FAST LIKE C++, OR SLOW LIKE PYTHON?

Used correctly, Rust code performs well – potentially, better than Java and certainly better than Python. Sometimes as well as C++

Rust can do most of what we do in C++, and often at the same speed.

But it lacks templates + constexpr + inlining: C++ magic speedup

RUST (CURRENTLY) IS WEAK ON COMPILE TIME OPTIMIZATIONS

With skilled use of templates and constexpr, a C++ program can be extensively precomputed, leaving only things that must occur at runtime.

Rust has generics but nothing analogous to the C++ template language, so SIMD coding isn't feasible. There are situations where Rust might be dramatically slower than C++ (despite using the same LLVM back end as Clang).

AN UNWINABLE DEBATE!

People who love Rust aren't going to switch back to C++

People who love C++ agree that Rust addresses many security issues (but at a cost). And they find ownership and mutability annoying, yet inadequate for even trivial data structures.

... the market adoption of Rust is good, but not overwhelming

BOTTOM LINE?

Rust genuinely is a powerful tool, but not trivial to use correctly, and limiting in important ways.

To get similar security in C++ requires systematic attention to risks, careful coding style, verification of logic, testing.

But because systems programming involves external libraries, GPUs and DMA, we sometimes have no other choice.

FURTHER READING

Haozhong Zhang "An Introduction to The Rust Programming Language"

Aaron Turon, The Rust Programming Language, Colloquium on Computer Systems Seminar Series (EE380), Stanford University, 2015.

Alex Crichton, Intro to the Rust programming language, http://people.mozilla.org/~acrichton/rust-talk-2014-12-10/

The Rust Programming Language, https://doc.rust-lang.org/stable/book/

Tim Chevalier, "Rust: A Friendly Introduction", 6/19/2013

RESOURCES

```
Rust website: <a href="http://rust-lang.org/">http://rust-lang.org/</a>
```

- Rust by example: http://rustbyexample.com/
- Guide: https://doc.rust-lang.org/stable/book/
- ➤ User forum: https://users.rust-lang.org/
- ➤ Book: https://doc.rust-lang.org/stable/book/academic-research.html

```
Speed of Rust versus C++*:
```

https://www.bairesdev.com/blog/when-speed-matters-comparing-rust-and-c/

WHAT DID WE LEARN TODAY?

The course really could have picked any of those languages

Rust is quite an interesting option: less risk of memory and concurrency issues (but forces unsafe code blocks in some cases).

We settled on C++ because it is elegant and strongly typed, while also making it easy to visualize how execution will behave.

BUT WHY DO WE NEED YOU TO LEARN C++?

Our course isn't a theory or ten mile high overview

Systems programming is about actually doing what we are teaching. Actually writing great code that explicitly and implicitly controls the hardware to achieve fantastic speed

... and yet is also correct, elegant, safe, and professional!

THEORY... VS. PRACTICE!



Cornell crypto club just made an <u>amazingly</u> profit.



This old car collector guy was on the losing side of the trade, and your friend, who has only driven a Prius, accepted this classic standard-shift Porche instead of cash.

Ready to jump in?

SELF-TEST

Do we really need to implement data structures by hand?

Can't we avoid these Rust criticisms by just relying entirely on libraries that do everything where unsafe code is needed?

SELF-TEST

Suppose you go all in on Rust



You build a big intelligent knowledge retrieval application using it. Now your company wants to brand it as the world's first totally safe intelligent document browser solution.

Would this be justified? What might a potential customer like a bank or a hospital ask, and how would you respond?