CS263: Rust vs C++

Subramaniyam Shankar
Rakshith Gopalakrishna

Introduction (Rust)

- Rust is developed by Mozilla.
- Rust is focused on providing performance and safety, especially safe concurrency and memory safety without garbage collection.
- Rust is very similar to C++, hence the comparison.
- Rust is the "most loved programming language" in the Stack Overflow developer survey since 2016.

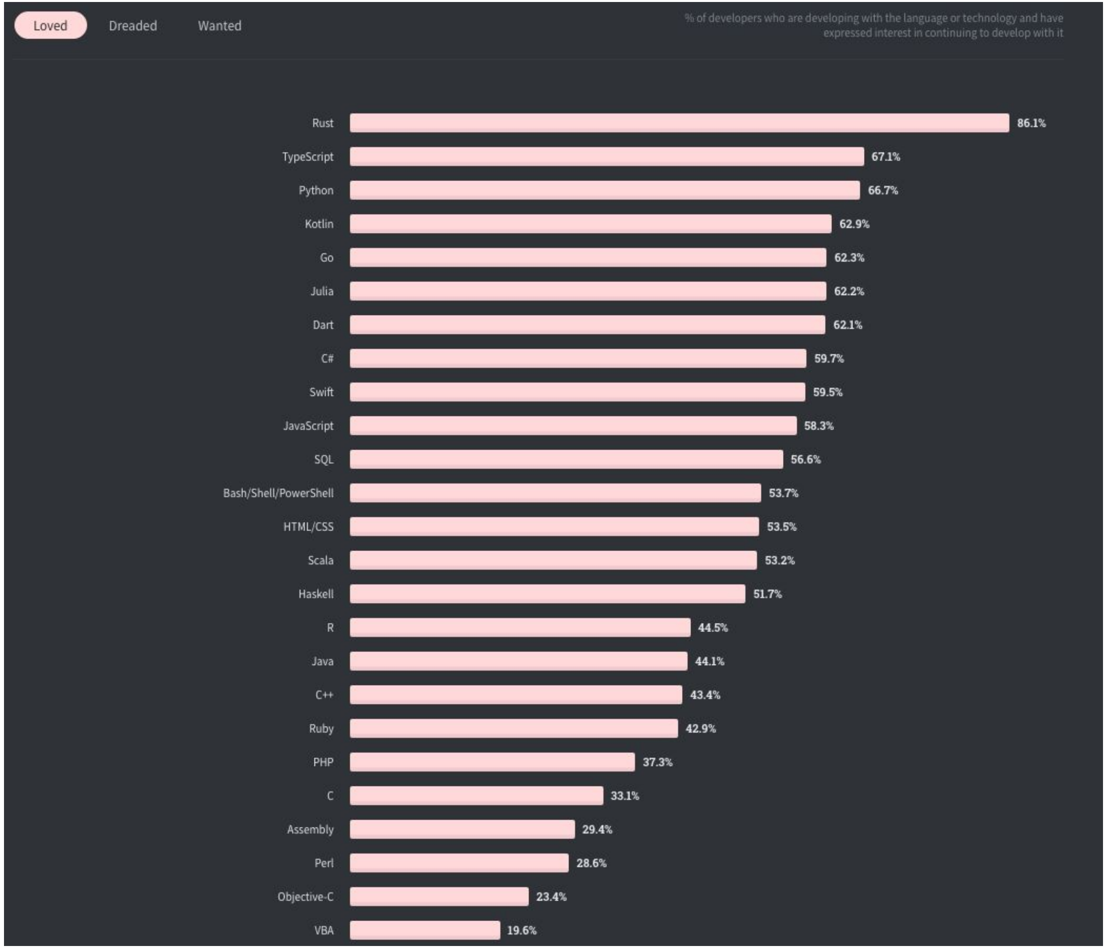
Introduction (Rust)

Rust: 86.1%

C++: 43.4%

Link:

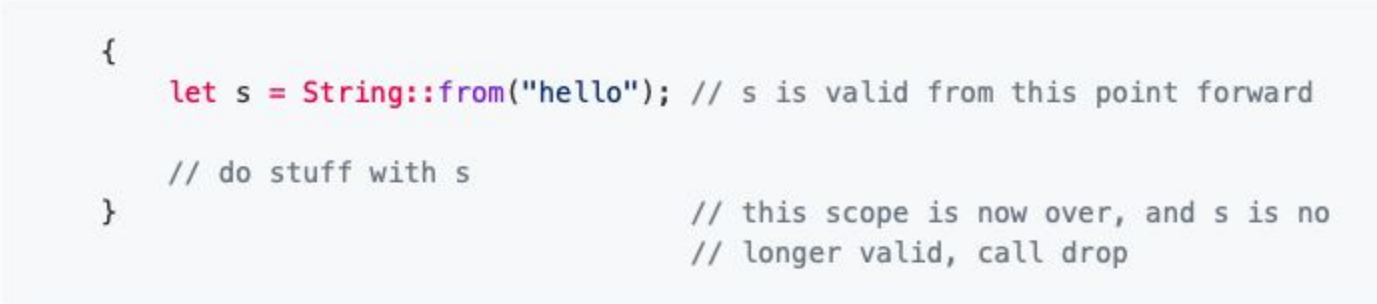
https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved



% of developers who are developing with the language or technology and have expressed interest in continuing to develop with it

Ownership

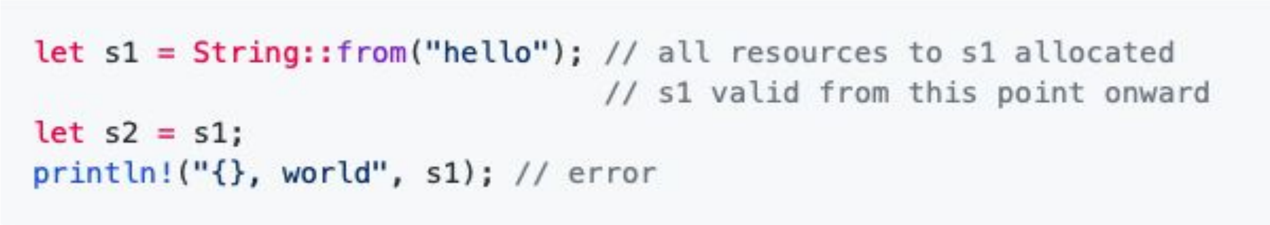
- Each value in Rust has a variable that’s called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.



```
{
    let s = String::from("hello"); // s is valid from this point forward

    // do stuff with s
}                                  // this scope is now over, and s is no
                                   // longer valid, call drop
```

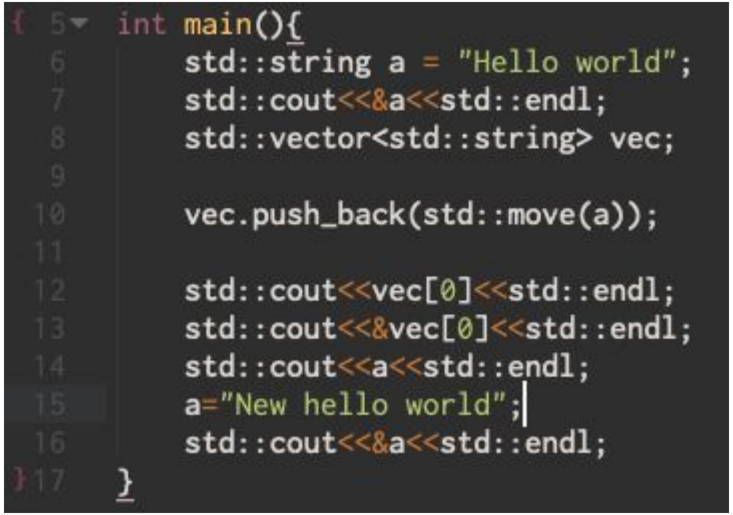
Ownership (Move semantics)

Rust:

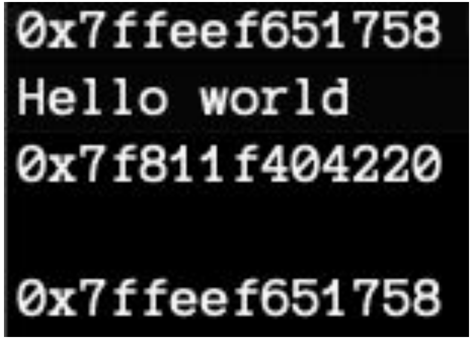


```rust
let s1 = String::from("hello"); // all resources to s1 allocated
                                // s1 valid from this point onward
let s2 = s1;
println!("{}, world", s1); // error
```

C++:



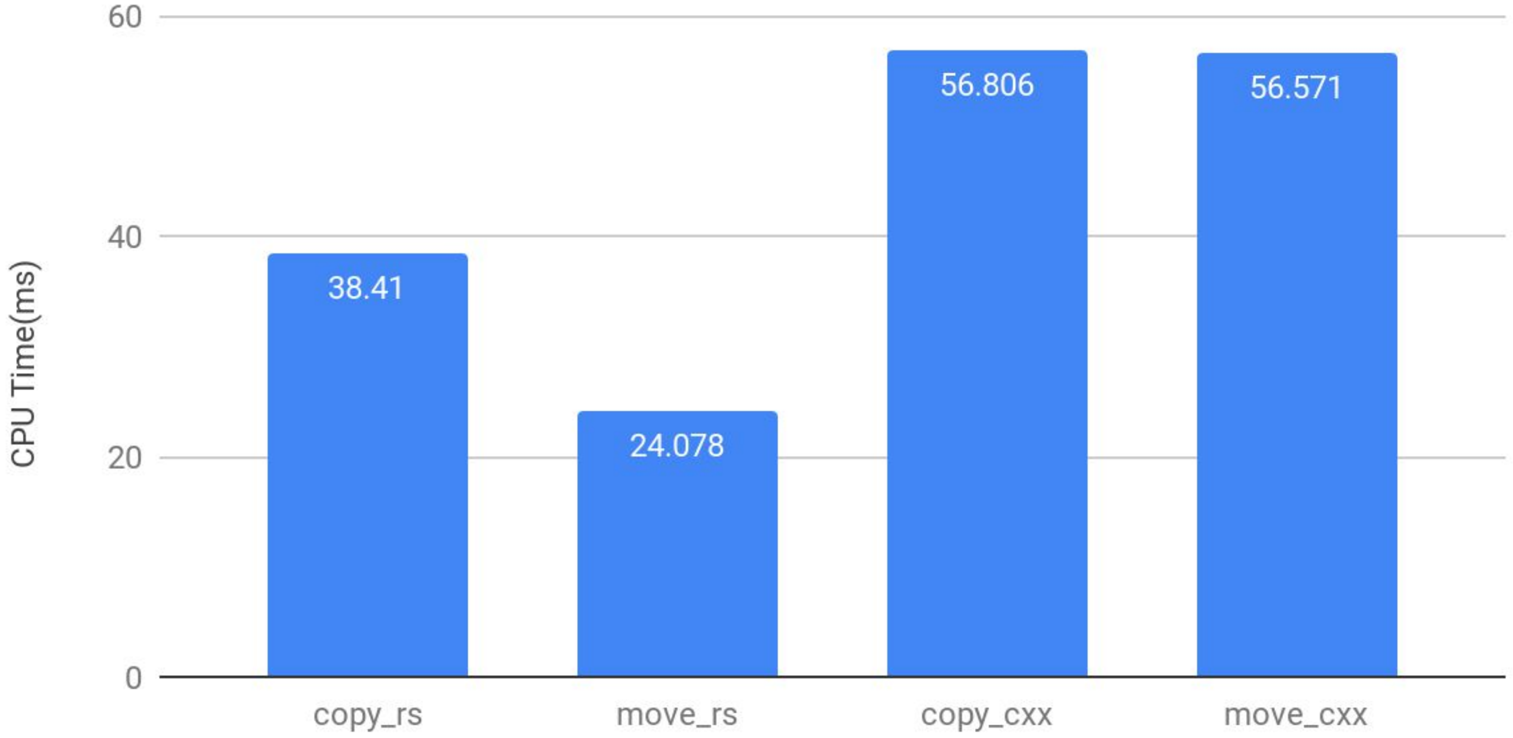
```cpp
int main(){
    std::string a = "Hello world";
    std::cout<<&a<<std::endl;
    std::vector<std::string> vec;

    vec.push_back(std::move(a));

    std::cout<<vec[0]<<std::endl;
    std::cout<<&vec[0]<<std::endl;
    std::cout<<a<<std::endl;
    a="New hello world";
    std::cout<<&a<<std::endl;
}
```



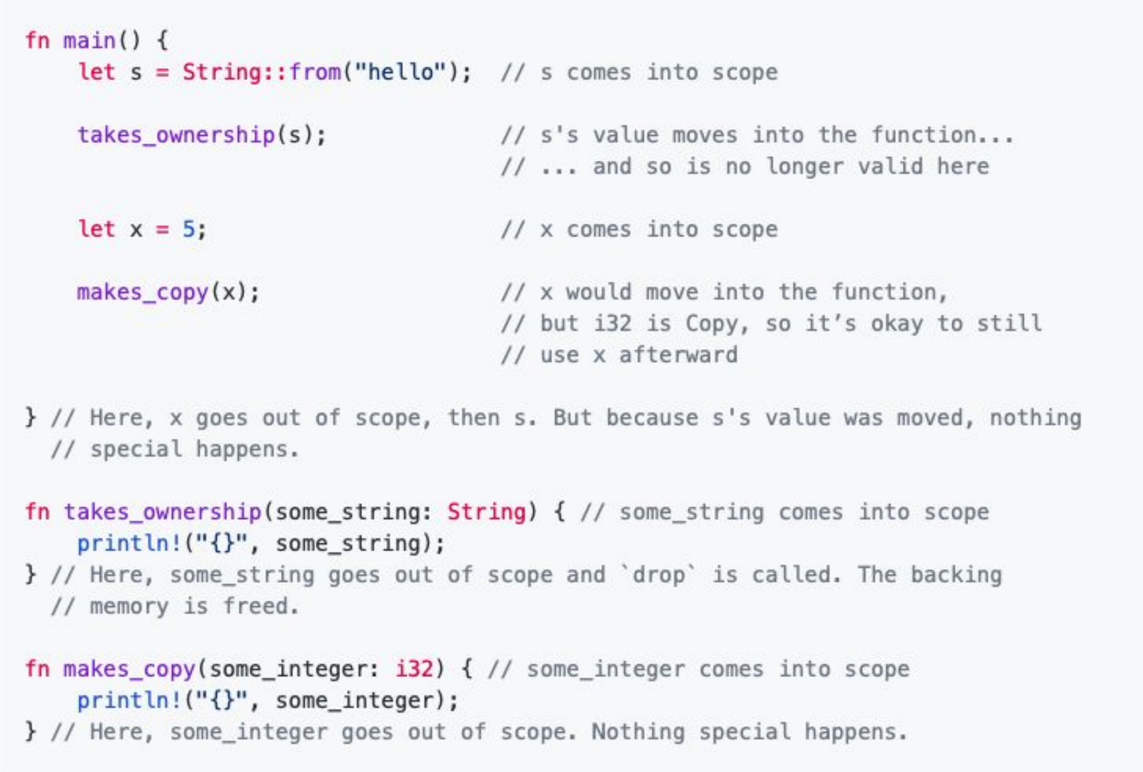
```
0x7ffeef651758
Hello world
0x7f811f404220

0x7ffeef651758
```

Vector move vs. copy

A vector of 10,000,000 integers was moved/copied.

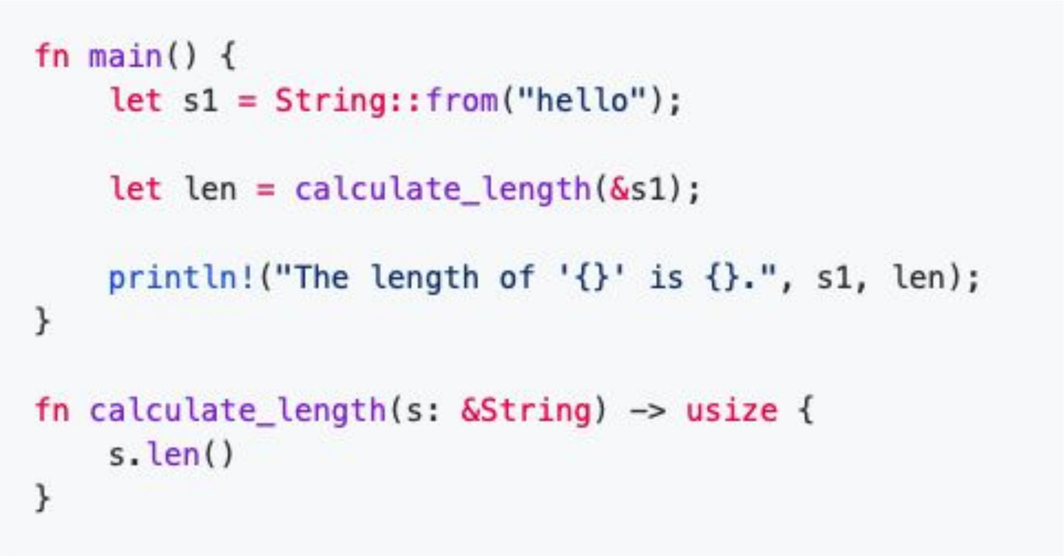


References in Rust



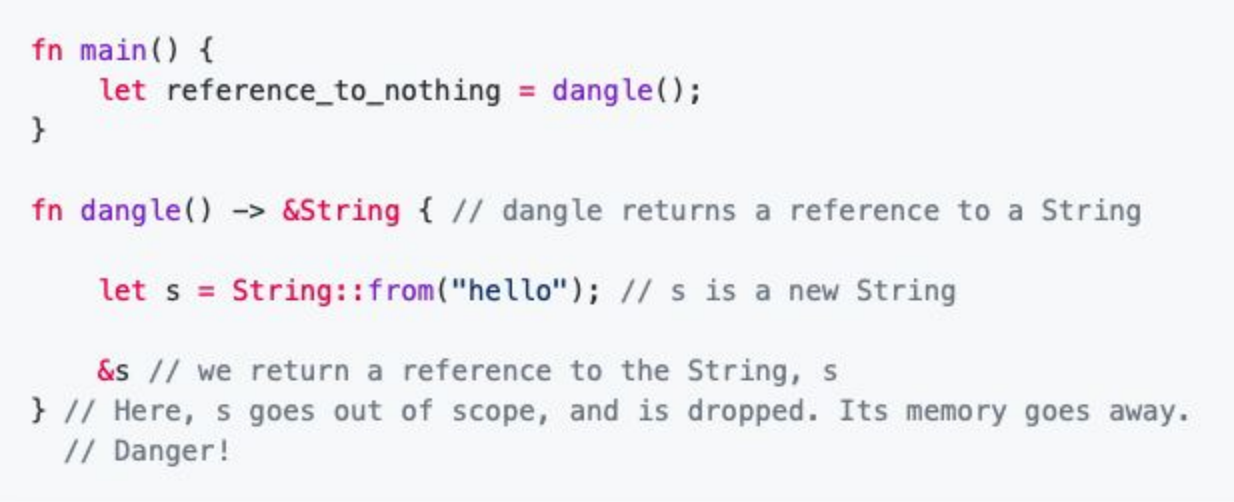
```rust
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it’s okay to still
                                    // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

References in Rust (contd.)



```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

References in Rust (contd.)

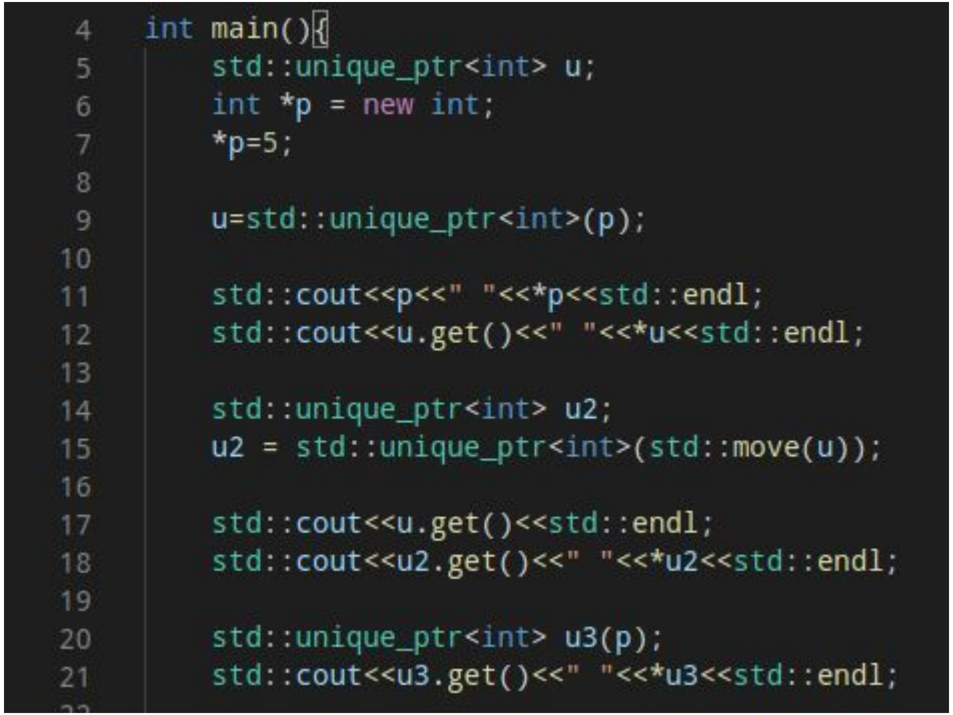


```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String { // dangle returns a reference to a String

    let s = String::from("hello"); // s is a new String

    &s // we return a reference to the String, s
} // Here, s goes out of scope, and is dropped. Its memory goes away.
  // Danger!
```

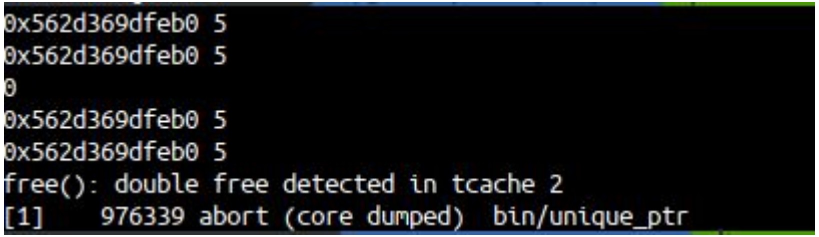
Invariants of Rust

- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid.

Smart pointers in C++: unique_ptr and Ownership

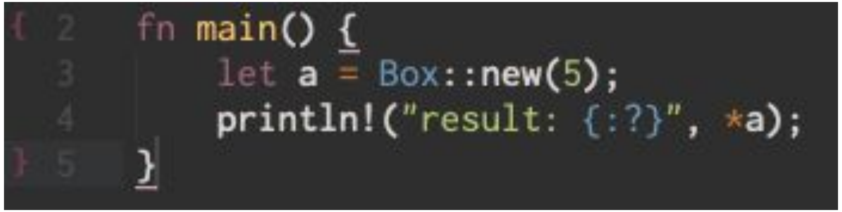


```cpp
int main(){
    std::unique_ptr<int> u;
    int *p = new int;
    *p=5;

    u=std::unique_ptr<int>(p);

    std::cout<<p<<" "<<*p<<std::endl;
    std::cout<<u.get()<<" "<<*u<<std::endl;

    std::unique_ptr<int> u2;
    u2 = std::unique_ptr<int>(std::move(u));

    std::cout<<u.get()<<std::endl;
    std::cout<<u2.get()<<" "<<*u2<<std::endl;

    std::unique_ptr<int> u3(p);
    std::cout<<u3.get()<<" "<<*u3<<std::endl;
```



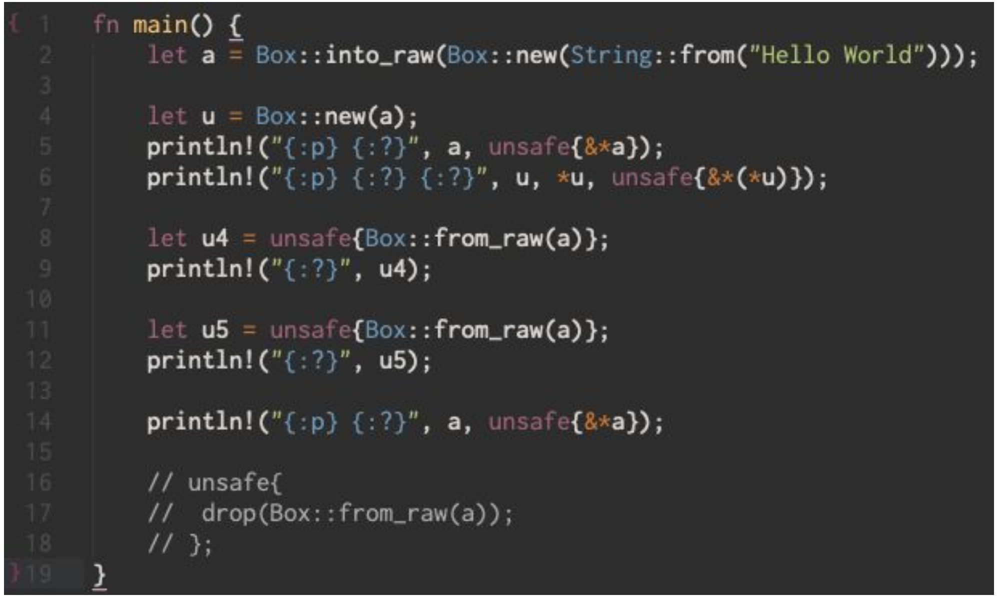
```
0x562d369dfeb0 5
0x562d369dfeb0 5
0
0x562d369dfeb0 5
0x562d369dfeb0 5
free(): double free detected in tcache 2
[1]    976339 abort (core dumped)  bin/unique_ptr
```

Smart pointer in Rust: Box pointer

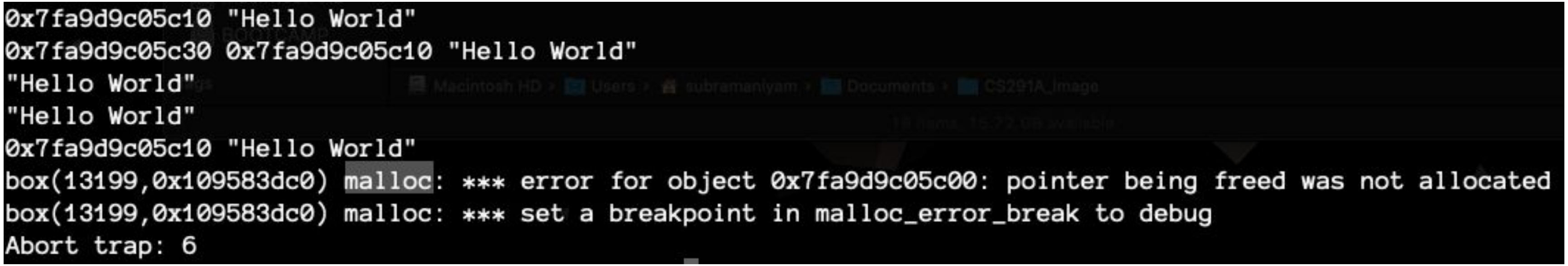


```
fn main() {
    let a = Box::new(5);
    println!("result: {:?}", *a);
}
```

Box pointer from Raw pointers (unsafe Rust)

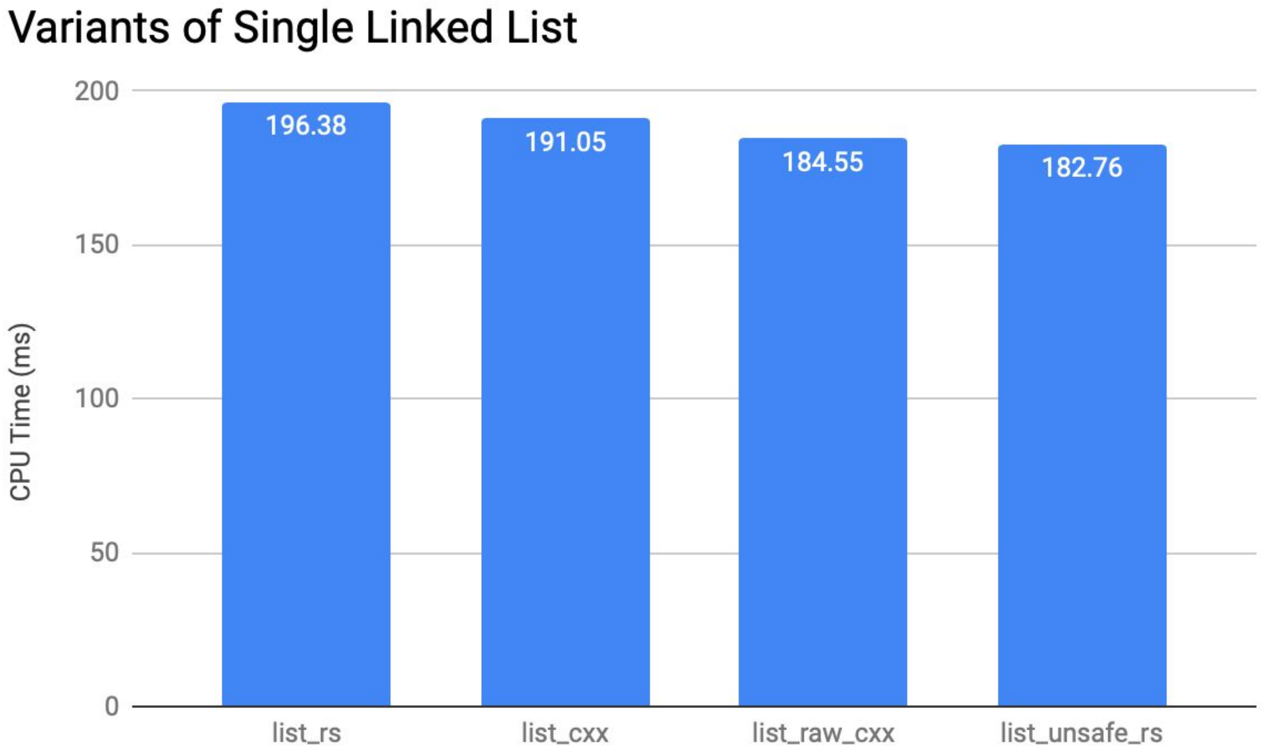


```rust
fn main() {
    let a = Box::into_raw(Box::new(String::from("Hello World")));

    let u = Box::new(a);
    println!("{:p} {:?}", a, unsafe{&*a});
    println!("{:p} {:?} {:?}", u, *u, unsafe{&*(*u)});

    let u4 = unsafe{Box::from_raw(a)};
    println!("{:?}", u4);

    let u5 = unsafe{Box::from_raw(a)};
    println!("{:?}", u5);

    println!("{:p} {:?}", a, unsafe{&*a});

    // unsafe{
    //  drop(Box::from_raw(a));
    // };
}
```

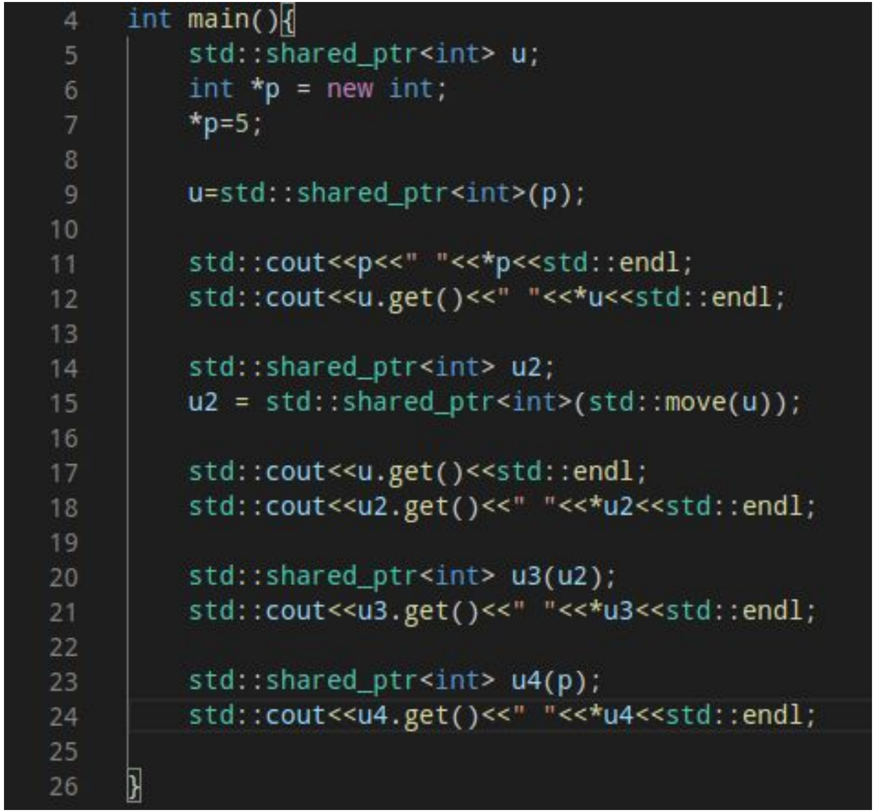


```
0x7fa9d9c05c10 "Hello World"
0x7fa9d9c05c30 0x7fa9d9c05c10 "Hello World"
"Hello World"
"Hello World"
0x7fa9d9c05c10 "Hello World"
box(13199,0x109583dc0) malloc: *** error for object 0x7fa9d9c05c00: pointer being freed was not allocated
box(13199,0x109583dc0) malloc: *** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

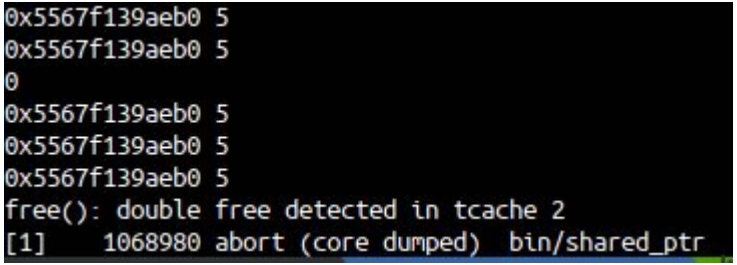
Linked List (Single)



Smart pointers in C++: shared_ptr

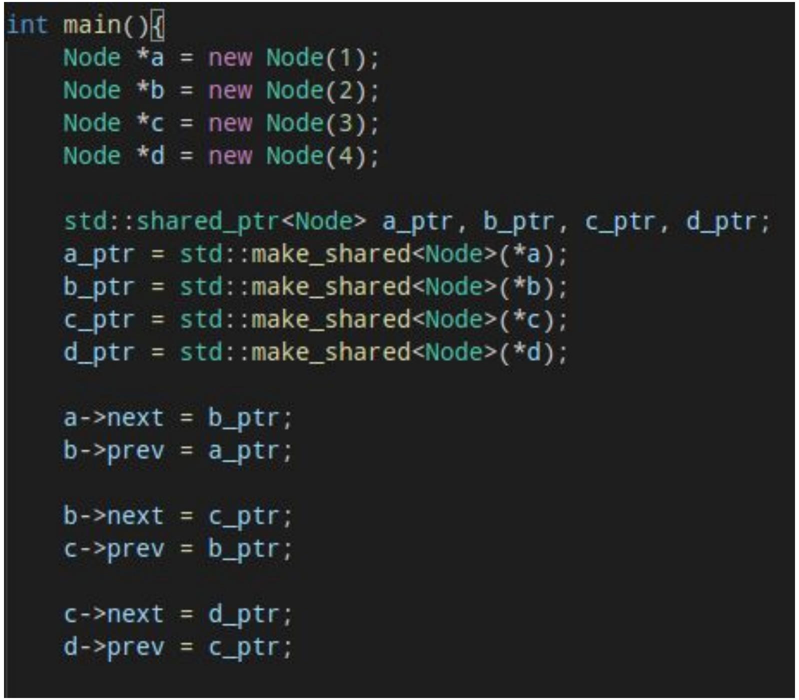


```cpp
int main(){
    std::shared_ptr<int> u;
    int *p = new int;
    *p=5;

    u=std::shared_ptr<int>(p);

    std::cout<<p<<" "<<*p<<std::endl;
    std::cout<<u.get()<<" "<<*u<<std::endl;

    std::shared_ptr<int> u2;
    u2 = std::shared_ptr<int>(std::move(u));

    std::cout<<u.get()<<std::endl;
    std::cout<<u2.get()<<" "<<*u2<<std::endl;

    std::shared_ptr<int> u3(u2);
    std::cout<<u3.get()<<" "<<*u3<<std::endl;

    std::shared_ptr<int> u4(p);
    std::cout<<u4.get()<<" "<<*u4<<std::endl;

}
```

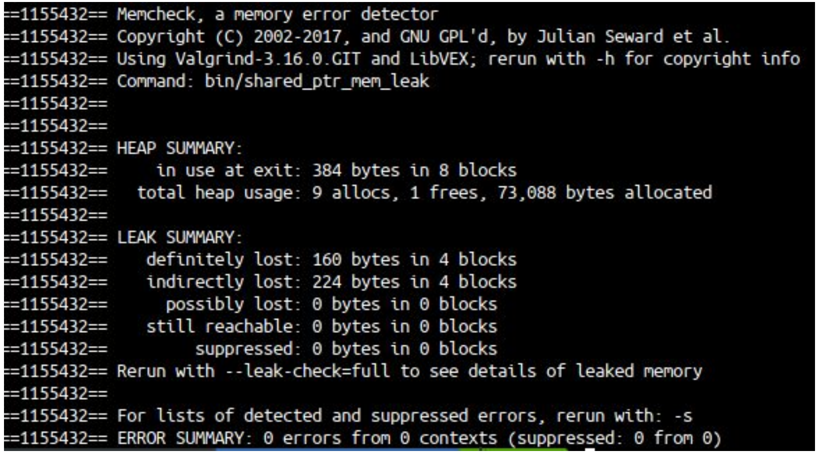


```
0x5567f139aeb0 5
0x5567f139aeb0 5
0
0x5567f139aeb0 5
0x5567f139aeb0 5
0x5567f139aeb0 5
free(): double free detected in tcache 2
[1]    1068980 abort (core dumped)  bin/shared_ptr
```

Memory leak with shared_ptr

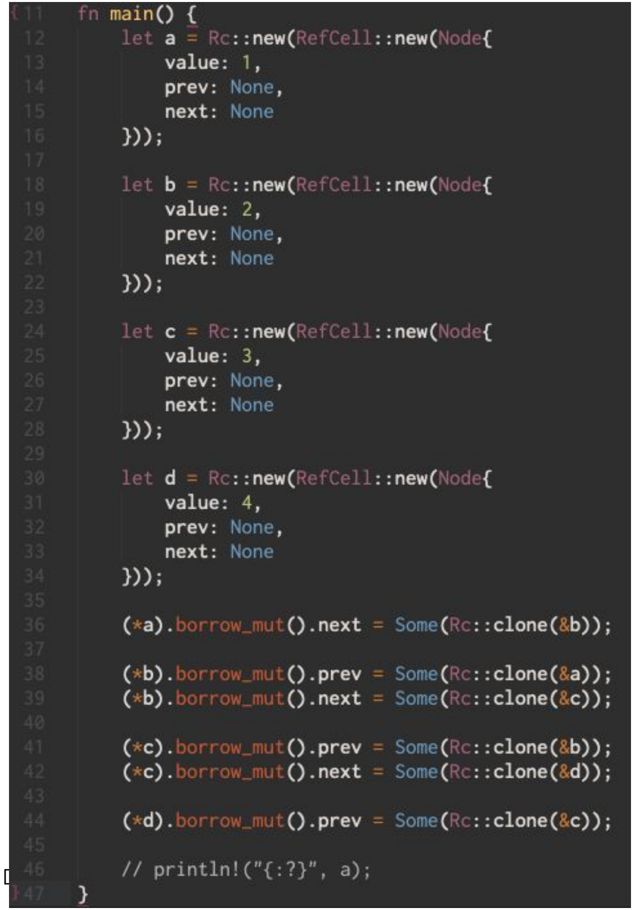


```
int main(){
    Node *a = new Node(1);
    Node *b = new Node(2);
    Node *c = new Node(3);
    Node *d = new Node(4);

    std::shared_ptr<Node> a_ptr, b_ptr, c_ptr, d_ptr;
    a_ptr = std::make_shared<Node>(*a);
    b_ptr = std::make_shared<Node>(*b);
    c_ptr = std::make_shared<Node>(*c);
    d_ptr = std::make_shared<Node>(*d);

    a->next = b_ptr;
    b->prev = a_ptr;

    b->next = c_ptr;
    c->prev = b_ptr;

    c->next = d_ptr;
    d->prev = c_ptr;
```

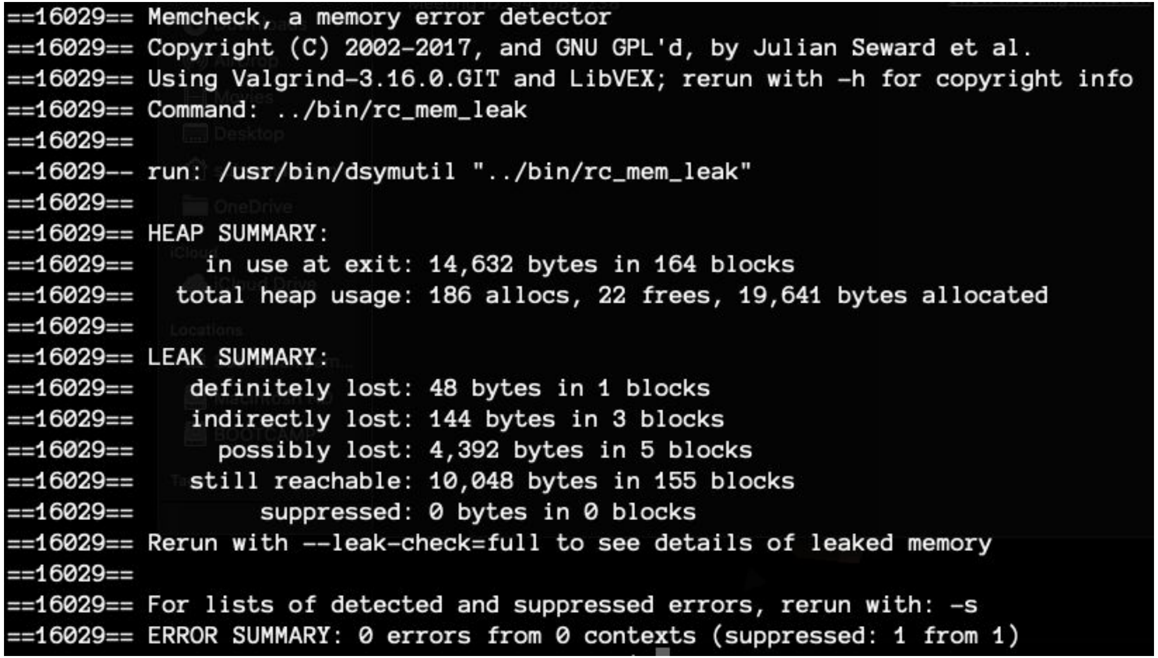


```
==1155432== Memcheck, a memory error detector
==1155432== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1155432== Using Valgrind-3.16.0.GIT and LibVEX; rerun with -h for copyright info
==1155432== Command: bin/shared_ptr_mem_leak
==1155432==
==1155432==
==1155432== HEAP SUMMARY:
==1155432==     in use at exit: 384 bytes in 8 blocks
==1155432==   total heap usage: 9 allocs, 1 frees, 73,088 bytes allocated
==1155432==
==1155432== LEAK SUMMARY:
==1155432==    definitely lost: 160 bytes in 4 blocks
==1155432==    indirectly lost: 224 bytes in 4 blocks
==1155432==      possibly lost: 0 bytes in 0 blocks
==1155432==    still reachable: 0 bytes in 0 blocks
==1155432==         suppressed: 0 bytes in 0 blocks
==1155432== Rerun with --leak-check=full to see details of leaked memory
==1155432==
==1155432== For lists of detected and suppressed errors, rerun with: -s
==1155432== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rc (Reference Counted) Smart Pointer: Rust

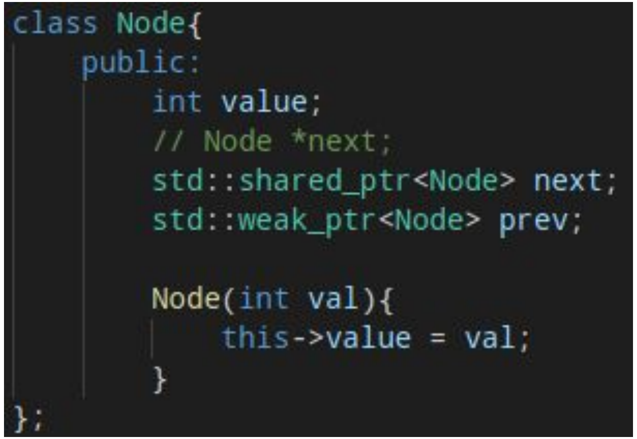


```rust
fn main() {
    let a = Rc::new(RefCell::new(Node{
        value: 1,
        prev: None,
        next: None
    }));

    let b = Rc::new(RefCell::new(Node{
        value: 2,
        prev: None,
        next: None
    }));

    let c = Rc::new(RefCell::new(Node{
        value: 3,
        prev: None,
        next: None
    }));

    let d = Rc::new(RefCell::new(Node{
        value: 4,
        prev: None,
        next: None
    }));

    (*a).borrow_mut().next = Some(Rc::clone(&b));

    (*b).borrow_mut().prev = Some(Rc::clone(&a));
    (*b).borrow_mut().next = Some(Rc::clone(&c));

    (*c).borrow_mut().prev = Some(Rc::clone(&b));
    (*c).borrow_mut().next = Some(Rc::clone(&d));

    (*d).borrow_mut().prev = Some(Rc::clone(&c));

    // println!("{:?}", a);
}
```

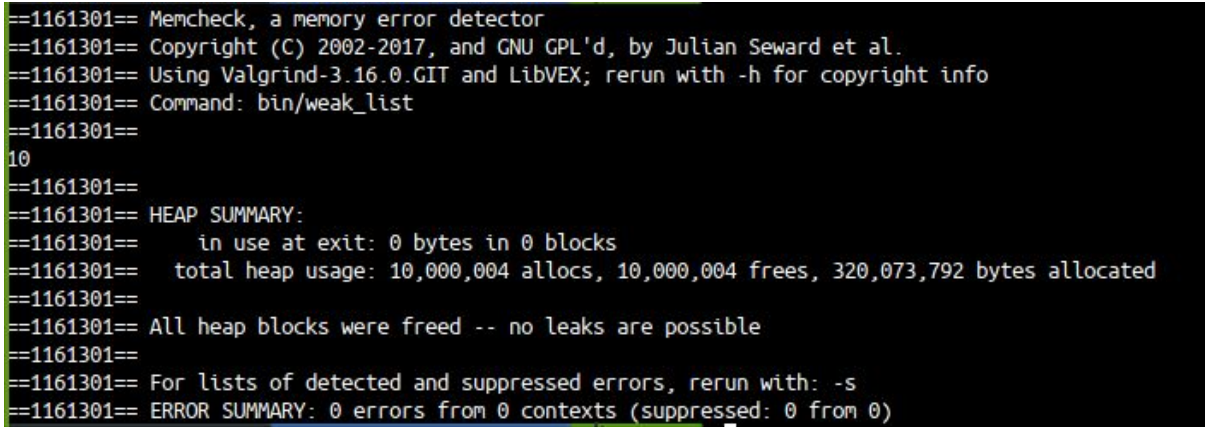


```
==16029== Memcheck, a memory error detector
==16029== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==16029== Using Valgrind-3.16.0.GIT and LibVEX; rerun with -h for copyright info
==16029== Command: ../bin/rc_mem_leak
==16029==
--16029-- run: /usr/bin/dsymutil "../bin/rc_mem_leak"
==16029==
==16029== HEAP SUMMARY:
==16029==     in use at exit: 14,632 bytes in 164 blocks
==16029==   total heap usage: 186 allocs, 22 frees, 19,641 bytes allocated
==16029==
==16029== LEAK SUMMARY:
==16029==    definitely lost: 48 bytes in 1 blocks
==16029==    indirectly lost: 144 bytes in 3 blocks
==16029==      possibly lost: 4,392 bytes in 5 blocks
==16029==    still reachable: 10,048 bytes in 155 blocks
==16029==         suppressed: 0 bytes in 0 blocks
==16029== Rerun with --leak-check=full to see details of leaked memory
==16029==
==16029== For lists of detected and suppressed errors, rerun with: -s
==16029== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 1 from 1)
```

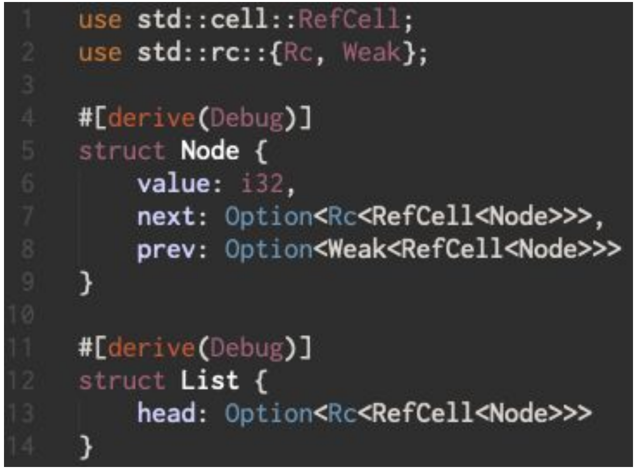
weak_ptr(C++)/Weak<T>(Rust) to the rescue



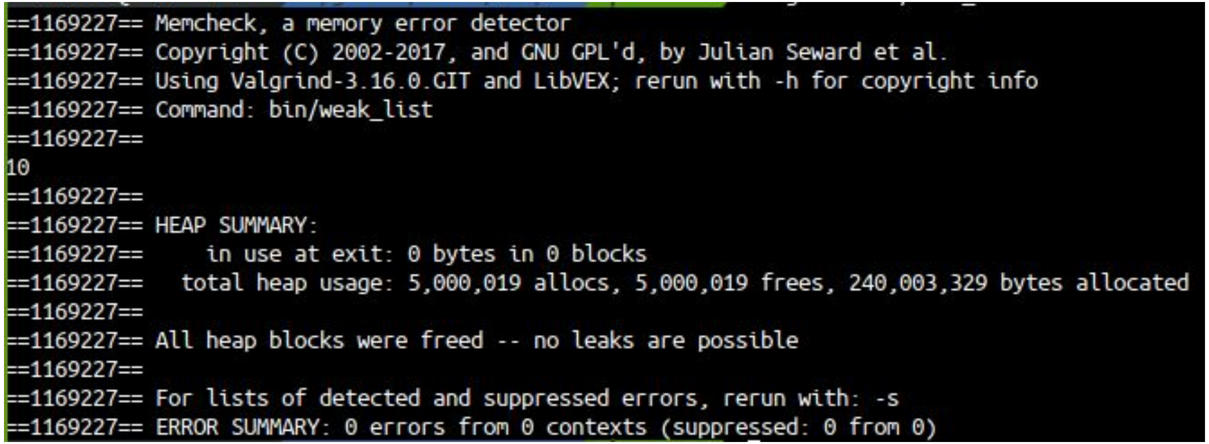
```
class Node{
    public:
        int value;
        // Node *next;
        std::shared_ptr<Node> next;
        std::weak_ptr<Node> prev;

        Node(int val){
            this->value = val;
        }
};
```



```
==1161301== Memcheck, a memory error detector
==1161301== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1161301== Using Valgrind-3.16.0.GIT and LibVEX; rerun with -h for copyright info
==1161301== Command: bin/weak_list
==1161301==
10
==1161301==
==1161301== HEAP SUMMARY:
==1161301==     in use at exit: 0 bytes in 0 blocks
==1161301==   total heap usage: 10,000,004 allocs, 10,000,004 frees, 320,073,792 bytes allocated
==1161301==
==1161301== All heap blocks were freed -- no leaks are possible
==1161301==
==1161301== For lists of detected and suppressed errors, rerun with: -s
==1161301== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

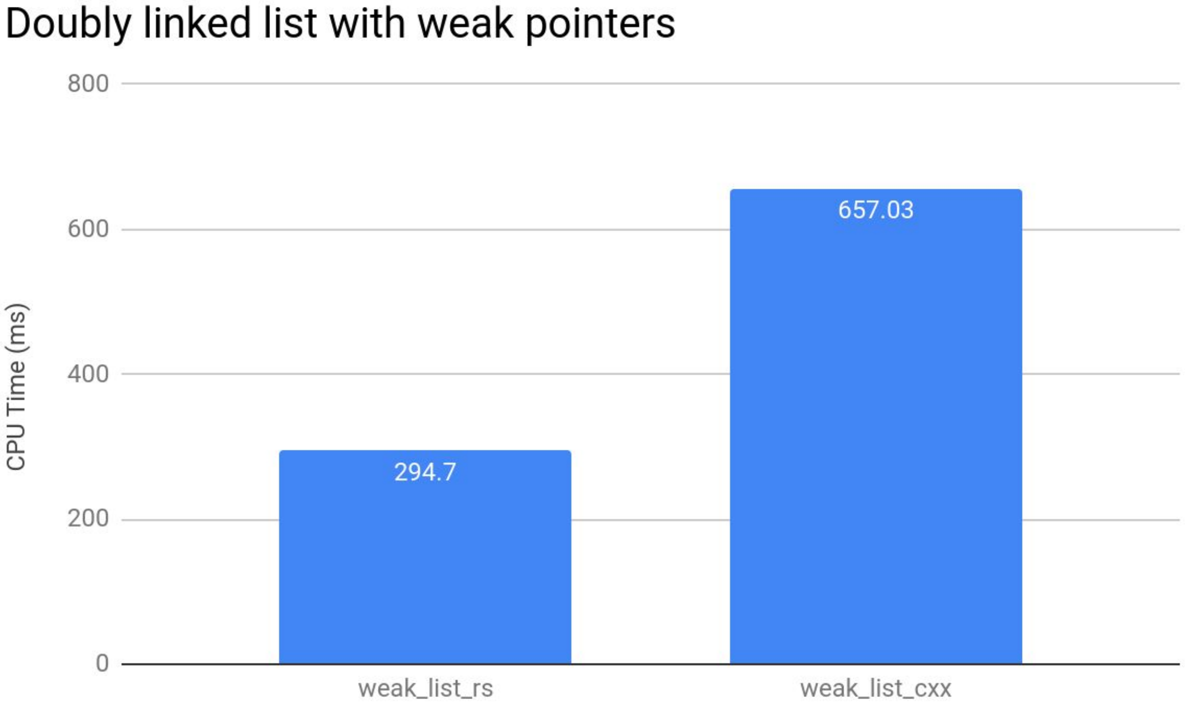


```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    next: Option<Rc<RefCell<Node>>>,
    prev: Option<Weak<RefCell<Node>>>
}

#[derive(Debug)]
struct List {
    head: Option<Rc<RefCell<Node>>>
}
```

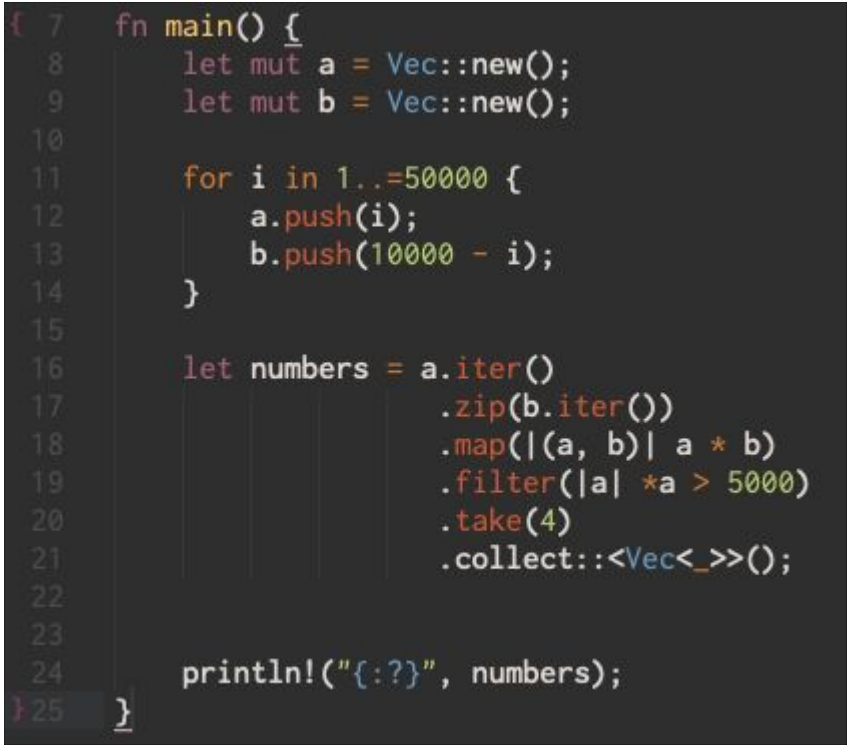


```
==1169227== Memcheck, a memory error detector
==1169227== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1169227== Using Valgrind-3.16.0.GIT and LibVEX; rerun with -h for copyright info
==1169227== Command: bin/weak_list
==1169227==
10
==1169227==
==1169227== HEAP SUMMARY:
==1169227==     in use at exit: 0 bytes in 0 blocks
==1169227==   total heap usage: 5,000,019 allocs, 5,000,019 frees, 240,003,329 bytes allocated
==1169227==
==1169227== All heap blocks were freed -- no leaks are possible
==1169227==
==1169227== For lists of detected and suppressed errors, rerun with: -s
==1169227== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

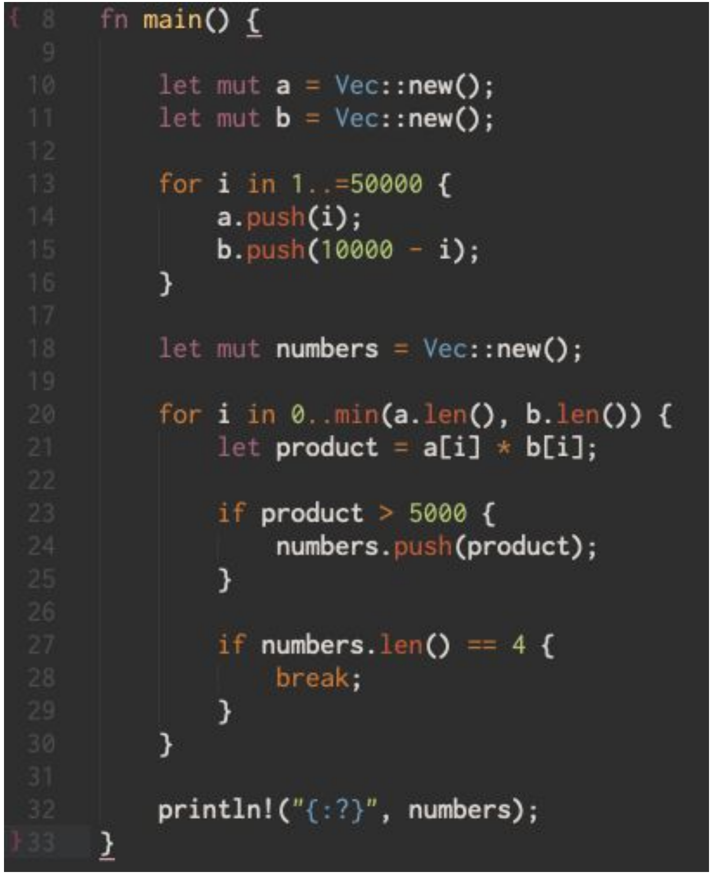
Weak_ptr to the rescue



Zero cost abstractions (iterators and closures)

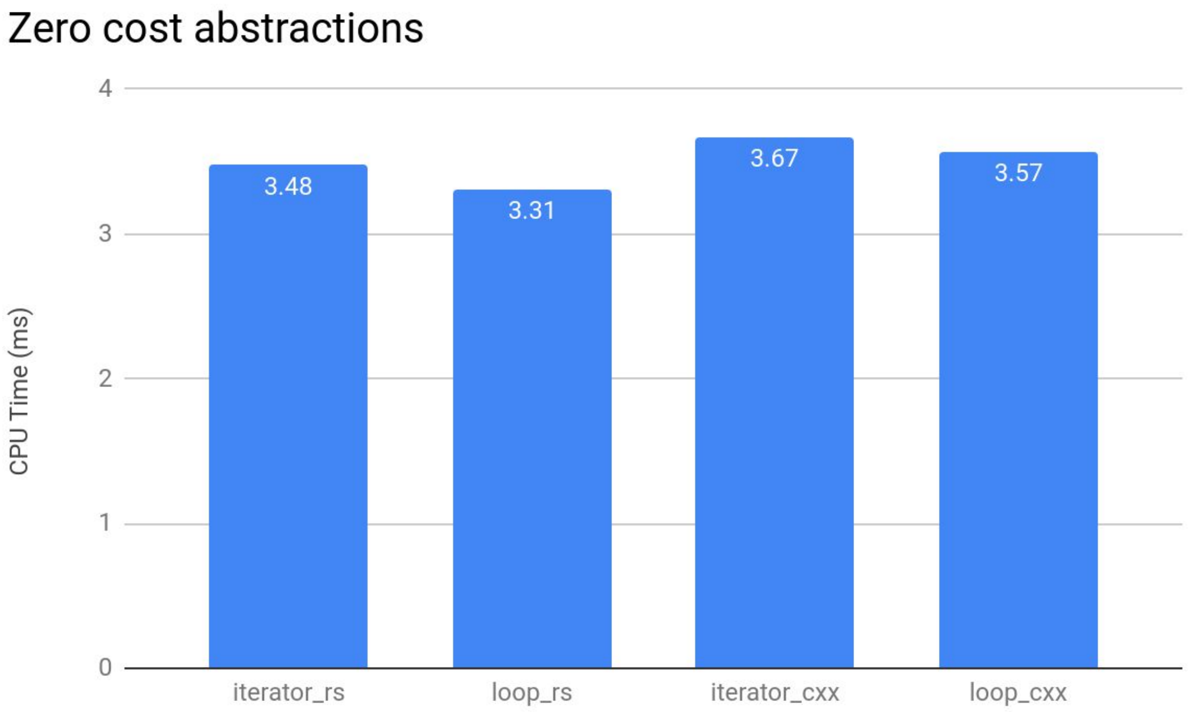


```
fn main() {
    let mut a = Vec::new();
    let mut b = Vec::new();

    for i in 1..=50000 {
        a.push(i);
        b.push(10000 - i);
    }

    let numbers = a.iter()
                   .zip(b.iter())
                   .map(|(a, b)| a * b)
                   .filter(|a| *a > 5000)
                   .take(4)
                   .collect::<Vec<_>>();


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

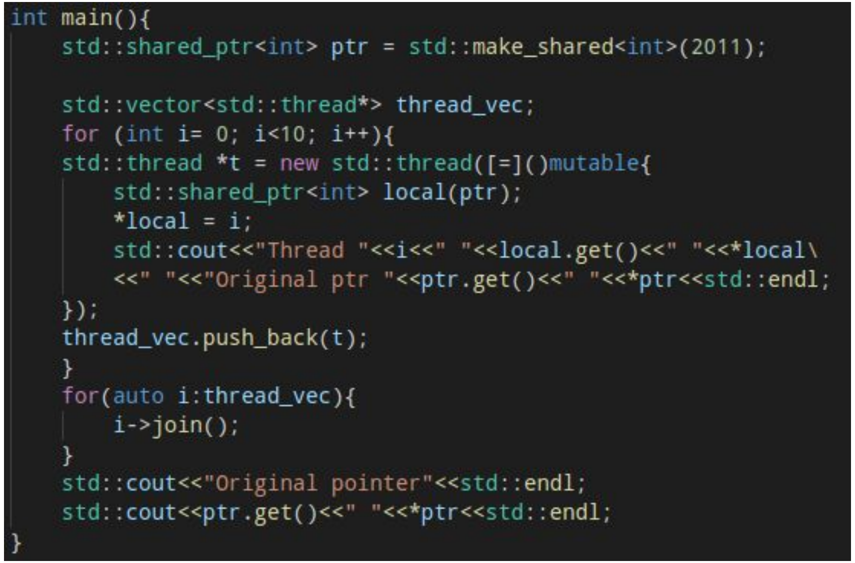


```
fn main() {

    let mut a = Vec::new();
    let mut b = Vec::new();

    for i in 1..=50000 {
        a.push(i);
        b.push(10000 - i);
    }

    let mut numbers = Vec::new();

    for i in 0..min(a.len(), b.len()) {
        let product = a[i] * b[i];

        if product > 5000 {
            numbers.push(product);
        }

        if numbers.len() == 4 {
            break;
        }
    }

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

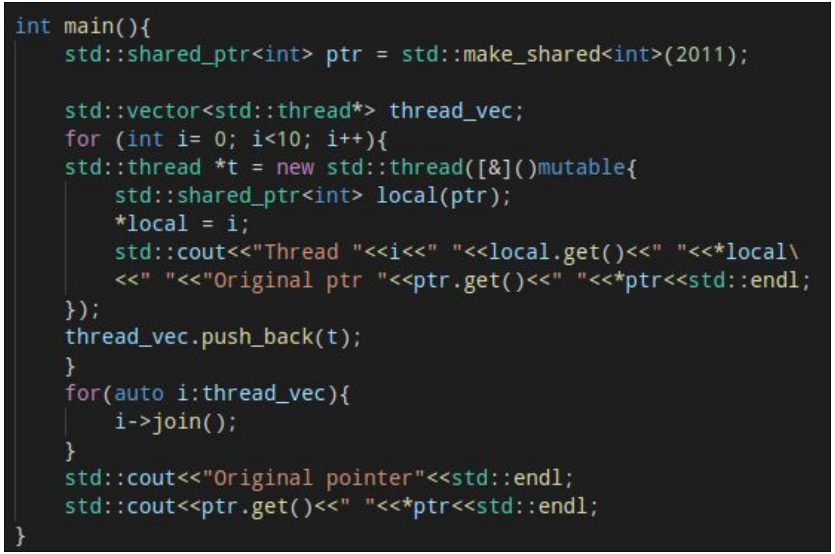
Zero cost abstractions



Concurrency: Sharing Same memory

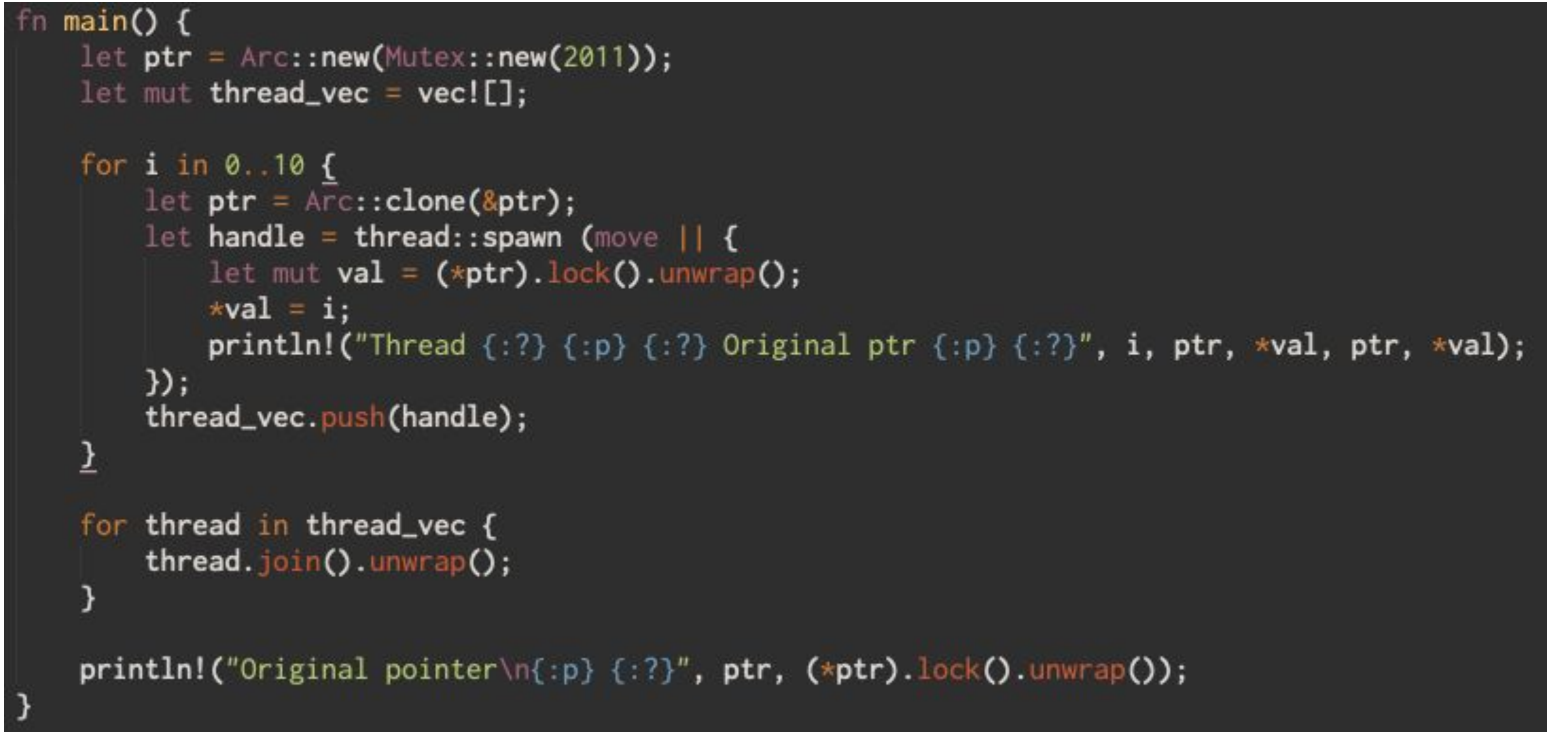


```
int main(){
    std::shared_ptr<int> ptr = std::make_shared<int>(2011);

    std::vector<std::thread*> thread_vec;
    for (int i= 0; i<10; i++){
    std::thread *t = new std::thread([=]()mutable{
        std::shared_ptr<int> local(ptr);
        *local = i;
        std::cout<<"Thread "<<i<<" "<<local.get()<<" "<<*local\
        <<" "<<"Original ptr "<<ptr.get()<<" "<<*ptr<<std::endl;
    });
    thread_vec.push_back(t);
    }
    for(auto i:thread_vec){
        i->join();
    }
    std::cout<<"Original pointer"<<std::endl;
    std::cout<<ptr.get()<<" "<<*ptr<<std::endl;
}
```



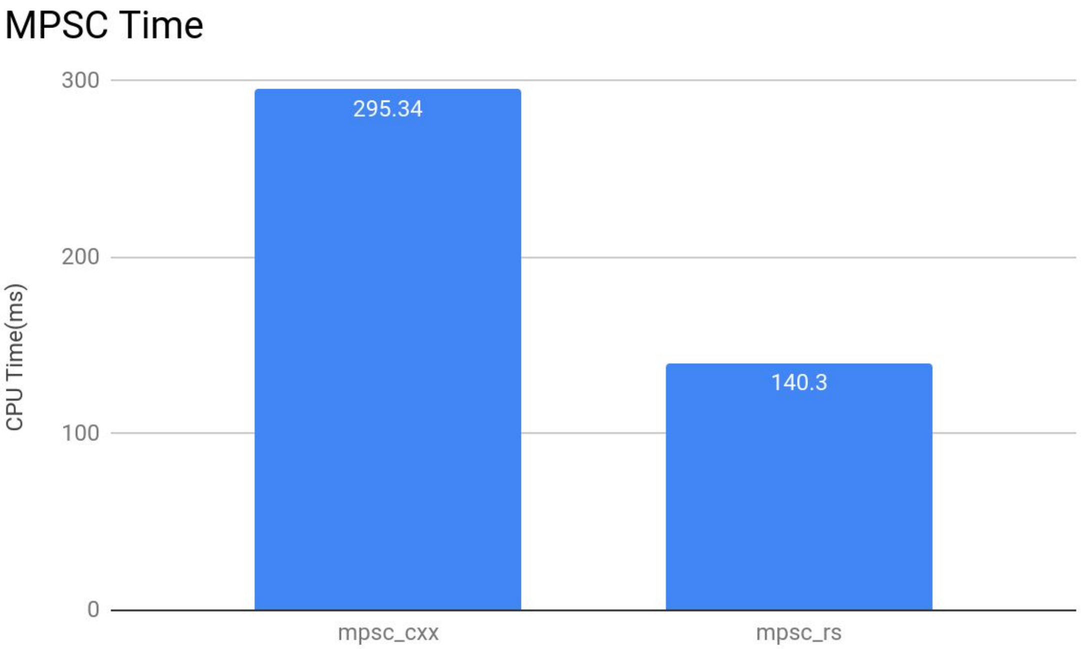
```
int main(){
    std::shared_ptr<int> ptr = std::make_shared<int>(2011);

    std::vector<std::thread*> thread_vec;
    for (int i= 0; i<10; i++){
    std::thread *t = new std::thread([&]()mutable{
        std::shared_ptr<int> local(ptr);
        *local = i;
        std::cout<<"Thread "<<i<<" "<<local.get()<<" "<<*local\
        <<" "<<"Original ptr "<<ptr.get()<<" "<<*ptr<<std::endl;
    });
    thread_vec.push_back(t);
    }
    for(auto i:thread_vec){
        i->join();
    }
    std::cout<<"Original pointer"<<std::endl;
    std::cout<<ptr.get()<<" "<<*ptr<<std::endl;
}
```

Concurrency: Sharing Same memory



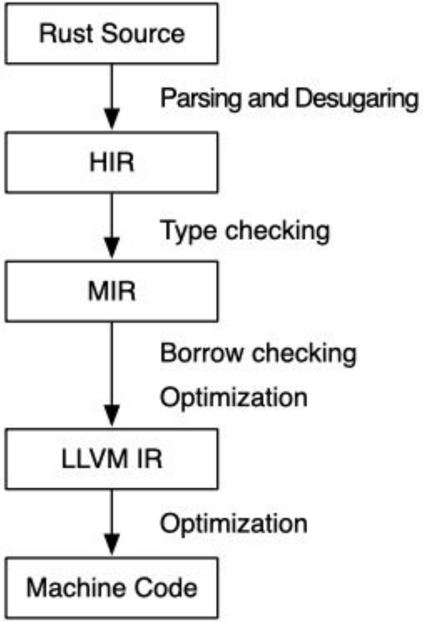
```
fn main() {
    let ptr = Arc::new(Mutex::new(2011));
    let mut thread_vec = vec![];

    for i in 0..10 {
        let ptr = Arc::clone(&ptr);
        let handle = thread::spawn (move || {
            let mut val = (*ptr).lock().unwrap();
            *val = i;
            println!("Thread {:?} {:p} {:?} Original ptr {:p} {:?}", i, ptr, *val, ptr, *val);
        });
        thread_vec.push(handle);
    }

    for thread in thread_vec {
        thread.join().unwrap();
    }

    println!("Original pointer\n{:p} {:?}", ptr, (*ptr).lock().unwrap());
}
```

Concurrency: Message Passing



Other Features that we explored

- Monomorphization of Generic types in both C++ and Rust by examining the assembly
- Dynamic Dispatch
- Index Loop unrolling



Questions?