Designing a Safe Doubly Linked List with C++-like Iterators in Rust

Nikolaos Vathis

PhD Candidate, School of Electrical and Computer Engineering, National Technical University of Athens
& Software Engineer, Navarino S.A.

December 28, 2018

Who am I?

Who am I?

- Programming Language enthusiast, PhD candidate in Heuristic Algorithms, Software Engineer.

Who am I?

- Programming Language enthusiast, PhD candidate in Heuristic Algorithms, Software Engineer.
- My favourite languages are Rust, Ruby, F# and C++ (in order).

Who am I?

- Programming Language enthusiast, PhD candidate in Heuristic Algorithms, Software Engineer.
- My favourite languages are Rust, Ruby, F# and C++ (in order).
- If I had to describe myself with a single expression, that would be "Jack of all trades, master of none".

Why?

Why?

- Why Rust? My alternative was to serve C++ on the internet.

Why?

- Why Rust? My alternative was to serve C++ on the internet.
- Why linked lists? My PhD revolves around heuristic algorithms. I need the asymptotic behaviour of doubly linked lists.

Why?

- Why Rust? My alternative was to serve C++ on the internet.
- Why linked lists? My PhD revolves around heuristic algorithms. I need the asymptotic behaviour of doubly linked lists.
- But again, why Rust? Before changing PhD subject, I wanted to create a better C++. Someone noticed that my language looked a lot like Rust pre-1.0.0...The rest is history :)

Why is this even a presentation?

Why is this even a presentation?

- Never stop redesigning the wheel. Programming in itself is *not* a solved problem.

Why is this even a presentation?

- Never stop redesigning the wheel. Programming in itself is *not* a solved problem.
- Mutable doubly linked lists are *not* a solved problem. A datatype that *cannot* expose a safe interface is *not* a solution to *any* problem.

Why is this even a presentation?

- Never stop redesigning the wheel. Programming in itself is *not* a solved problem.
- Mutable doubly linked lists are *not* a solved problem. A datatype that *cannot* expose a safe interface is *not* a solution to *any* problem.
- Linked lists in Rust are a complete PITA. I urge you to read a book called “Learning Rust With Entirely Too Many Linked Lists” [3] at some point. Standing on the shoulders of giants and such :)

What is a linked list?

What is a linked list?

- `type 'a List = Nil | Cons of 'a * 'a List`
- `struct List { int x; List *next };`

What is a linked list?

- `type 'a List = Nil | Cons of 'a * 'a List`
- `struct List { int x; List *next };`
- `std::list<T>`

Rules of singly linked lists:

Rules of singly linked lists:

1. They are never the right datatype

Rules of singly linked lists:

1. They are never the right datatype
2. There are some exceptions

Rules of singly linked lists:

1. They are never the right datatype
2. There are some exceptions
3. What you are thinking right now is not an exception :)

Exception: Formal Proofs

```
let rec maxlist (lst: 'a list) =
    match lst with
    | [x] -> x
    | (x :: xs) -> max x (maxlist xs)
```

vs

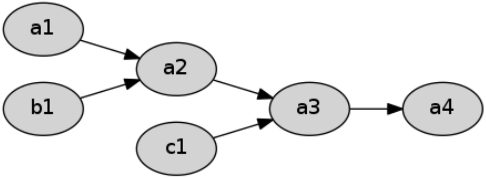
```
let maxarr (arr: 'a []) =
    let mutable ret = arr.[0]
    let mutable i = 1 (* 1 or 0? What is easier to prove? *)
    let l = Array.length arr

    while i < l do (* Does this even terminate? *)
        ret <- max ret arr.[i]
        i <- i + 1
        (* What is the invariant here? *)

    ret
```

Exception: Persistent Datatypes

You might need to generate many lists with a common suffix, and you want to abstract this away from you.



This makes sense only for *immutable* singly linked lists.

So, let's be honest. Singly linked lists are *useless* as general purpose datatypes and have no place in a language in which they need more than one line to be defined (i.e., a non-functional language).

What are the benefits of a doubly linked list, as opposed to singly linked lists?

What are the benefits of a doubly linked list, as opposed to singly linked lists?

- Bidirectional access

What are the benefits of a doubly linked list, as opposed to singly linked lists?

- Bidirectional access
- Can delete an element by having an iterator to the specific element (and not the previous)

What are the benefits of a doubly linked list, as opposed to singly linked lists?

- Bidirectional access
- Can delete an element by having an iterator to the specific element (and not the previous)
- If anyone wants to refute the previous bullet, the correct sentence would be "more intuitive mutable iterators and/or invalidation patterns"

What are the benefits of a doubly linked list, as opposed to singly linked lists?

- Bidirectional access
- Can delete an element by having an iterator to the specific element (and not the previous)
- If anyone wants to refute the previous bullet, the correct sentence would be "more intuitive mutable iterators and/or invalidation patterns"

When are doubly linked lists needed?

You want a dynamic datatype that has the magic property that it can grow on runtime.

You want a dynamic datatype that has the magic property that it can grow on runtime.

Ugh, just no. Please.

You want a dynamic datatype that has the magic property that it can grow on runtime.

Ugh, just no. Please.

- Choose a vector.

You want a dynamic datatype that has the magic property that it can grow on runtime.

Ugh, just no. Please.

- Choose a vector.
- You've seen the talk from Bjarne Stroustup [1], right?

You are writing a queue, therefore you cannot use a vector.

You are writing a queue, therefore you cannot use a vector.

I could tell you my opinion myself, or I could show you which datatype the STL uses for a queue...

You are writing a queue, therefore you cannot use a vector.

I could tell you my opinion myself, or I could show you which datatype the STL uses for a queue...

std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

You are writing a queue, therefore you cannot use a vector.

I could tell you my opinion myself, or I could show you which datatype the STL uses for a queue...

std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

Also, a fun fact is that Rust standard library defines deque using an underlying vector.

- You want to delete from/insert to the middle of the list. Watch the Bjarne Stroustrup talk [1], as that is $O(n)$

- You want to delete from/insert to the middle of the list. Watch the Bjarne Stroustrup talk [1], as that is $O(n)$
- The same as above, but you already have an iterator there. If you don't care about the order of elements, swap with the last element and then `pop_back`.

- You want to delete from/insert to the middle of the list. Watch the Bjarne Stroustrup talk [1], as that is $O(n)$
- The same as above, but you already have an iterator there. If you don't care about the order of elements, swap with the last element and then `pop_back`.
- The same as above, but you actually care about order. Well, congrats! You actually *need* to use a doubly linked list!

- You want to delete from/insert to the middle of the list. Watch the Bjarne Stroustrup talk [1], as that is $O(n)$
- The same as above, but you already have an iterator there. If you don't care about the order of elements, swap with the last element and then `pop_back`.
- The same as above, but you actually care about order. Well, congrats! You actually *need* to use a doubly linked list!
- There are other somewhat valid cases. The Rust standard library states frequent splitting / merging as another reason to use linked lists. Another somewhat valid case is iterator invalidation rules.

What to remember:

What to remember:

1. Use a doubly linked list if you need to delete from/insert to the middle of a container, while already having an iterator there, and order matters.

What to remember:

1. Use a doubly linked list if you need to delete from/insert to the middle of a container, while already having an iterator there, and order matters.
2. Always benchmark the datatypes you choose.

What to remember:

1. Use a doubly linked list if you need to delete from/insert to the middle of a container, while already having an iterator there, and order matters.
2. Always benchmark the datatypes you choose.
3. Doubly linked lists are niche datatypes and it is questionable whether they should be in a standard library.

What is Ownership?

When C++ passes something by value, it performs a deep copy.

When C++ passes something by value, it performs a deep copy.

```cpp
struct Foo { int field; /* + other fields */ };

void print(vector<Foo> vec)
{
    for (Foo elem : vec) // will copy each element
        cout << elem.field << endl;
}

int main()
{
    vector<Foo> x;
    // push several values to x;
    vector<Foo> y = x; // will copy whole vector
    print(y); // will copy whole vector to local variable
    return 0;
}
```

At least, if the Holy Trinity (destructor, copy constructor, copy assignment operator) is correctly implemented.

At least, if the Holy Trinity (destructor, copy constructor, copy assignment operator) is correctly implemented.

Of course, there are legitimated cases for a shallow copy. If the value is never used again, the programmer or the compiler might perform a move.

At least, if the Holy Trinity (destructor, copy constructor, copy assignment operator) is correctly implemented.

Of course, there are legitimated cases for a shallow copy. If the value is never used again, the programmer or the compiler might perform a move.

Of course, that depends on the correct implementation of the rule of three/five/zero.

When Rust passes something by value, it moves it. Or, on Rust terms, it passes the ownership. On the low level, it performs a shallow copy. On the high level, it invalidates the old object (with some exceptions to primitive types). So, this fails to compile:

When Rust passes something by value, it moves it. Or, on Rust terms, it passes the ownership. On the low level, it performs a shallow copy. On the high level, it invalidates the old object (with some exceptions to primitive types). So, this fails to compile:

```rust
fn print_vec_5(myvec: Vec<i32>)
{
    println!("{}", myvec[5]);
}

fn main()
{
    let myvec = vec![0,1,2,3,4,5,6];
    print_vec_5(myvec);
    println!("{}", myvec[5]);
}
```

One *wondrous* side effect is that it is the responsibility of the *caller* to perform an expensive deep copy. In addition it is done explicitly, so the runtime cost is obvious:

One *wondrous* side effect is that it is the responsibility of the *caller* to perform an expensive deep copy. In addition it is done explicitly, so the runtime cost is obvious:

```
fn ret_vec_5_plus_1(mut myvec: Vec<i32>) -> Vec<i32>
{
    myvec[5] += 1;
    myvec
}

fn main()
{
    let myvec = vec![0,1,2,3,4,5,6];
    let myvec2 = ret_vec_5_plus_1(myvec.clone());
    println!("{:?}", myvec);
    println!("{:?}", myvec2);
}
```

The simplest way to understand ownership is to ask the simple question “Who is responsible for the destruction of this object?”.

The simplest way to understand ownership is to ask the simple question “Who is responsible for the destruction of this object?”.

The answer usually is something of the below:

- A local variable
- A field of a struct
- An owning pointer `Box<T>`

The simplest way to understand ownership is to ask the simple question “Who is responsible for the destruction of this object?”.

The answer usually is something of the below:

- A local variable
- A field of a struct
- An owning pointer `Box<T>`

Finally, ownership is not Rust specific. It also exists in C++.

void QWidget::addAction(QAction **action*)

Appends the action *action* to this widget's list of actions.

All QWidgets have a list of QActions, however they can be represented graphically in many different ways. The default use of the QAction list (as returned by actions()) is to create a context QMenu.

A QWidget should only have one of each action and adding an action it already has will not cause the same action to be in the widget twice.

The ownership of *action* is not transferred to this QWidget.

See also removeAction(), insertAction(), actions(), and QMenu.

QWidget *QWidget::createWindowContainer(QWindow *window, QWidget *parent = nullptr, Qt::WindowFlags flags = ...) `[static]`

Creates a QWidget that makes it possible to embed *window* into a QWidget-based application.

The window container is created as a child of *parent* and with window flags *flags*.

Once the window has been embedded into the container, the container will control the window's geometry and visibility. Explicit calls to QWindow::setGeometry(), QWindow::show() or QWindow::hide() on an embedded window is not recommended.

The container takes over ownership of *window*. The window can be removed from the window container with a call to QWindow::setParent().

What is borrow checking?

When C++ passes something by reference, the reference can alias.

When C++ passes something by reference, the reference can alias.

When Rust passes something by reference (called borrowing), the (glorious) Borrow Checker ensures at compile time that:

- A reference cannot outlive its referent
- A mutable reference cannot be aliased

When C++ passes something by reference, the reference can alias.

When Rust passes something by reference (called borrowing), the (glorious) Borrow Checker ensures at compile time that:

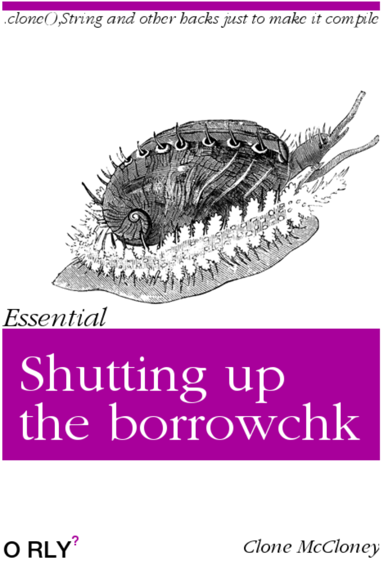
- A reference cannot outlive its referent
- A mutable reference cannot be aliased

This does not compile:

```rust
fn main()
{
    let mut myvec = vec![0,1,2,3,4,5,6];
    let r = &myvec[4];

    myvec.push(42);

    println!("{}", *r);
}
```


.clone(),String and other hacks just to make it compile
Essential
Shutting up the borrowchk
O RLY?
Clone McCloney

What to remember: Borrow checker ensures that each object can only be mutated by a single variable.

Let's talk about Rust standard LinkedList:

Let's talk about Rust standard LinkedList:

```
[-] pub fn split_off(&mut self, at: usize) -> LinkedList<T>
```

Splits the list into two at the given index. Returns everything after the given index, including the index.

This operation should compute in O(n) time.

Let's talk about Rust standard LinkedList:

```
[-] pub fn split_off(&mut self, at: usize) -> LinkedList<T>
```

Splits the list into two at the given index. Returns everything after the given index, including the index.

This operation should compute in O(n) time.

Oh come on! Give me a break! O(n) splitting‽ Adding only in the ends‽ At least it provided a valid reason to use two interrobangs!

Why is it so bad it hurts though? First and foremost, doubly linked lists as we know them *cannot* be sanely* expressed using safe Rust:

Why is it so bad it hurts though? First and foremost, doubly linked lists as we know them *cannot* be sanely* expressed using safe Rust:

- There is no sane* way to define the owner of each element.

Why is it so bad it hurts though? First and foremost, doubly linked lists as we know them *cannot* be sanely* expressed using safe Rust:

- There is no sane* way to define the owner of each element.
- There is no sane* way to define the next/prev references of each element.

Why is it so bad it hurts though? First and foremost, doubly linked lists as we know them *cannot* be sanely* expressed using safe Rust:

- There is no sane* way to define the owner of each element.
- There is no sane* way to define the next/prev references of each element.
- Even if we could, each element has at least one reference pointing to itself, so we cannot safely mutate it.

Why is it so bad it hurts though? First and foremost, doubly linked lists as we know them *cannot* be sanely* expressed using safe Rust:

- There is no sane* way to define the owner of each element.
- There is no sane* way to define the next/prev references of each element.
- Even if we could, each element has at least one reference pointing to itself, so we cannot safely mutate it.
- Finally, C++ lists are unsafe *by design*. C++ list iterators cannot be verified that do not violate memory, even on runtime. We yet have to discover a safe alternative for C++ iterators.

Why is it so bad it hurts though? First and foremost, doubly linked lists as we know them *cannot* be sanely* expressed using safe Rust:

- There is no sane* way to define the owner of each element.
- There is no sane* way to define the next/prev references of each element.
- Even if we could, each element has at least one reference pointing to itself, so we cannot safely mutate it.
- Finally, C++ lists are unsafe *by design*. C++ list iterators cannot be verified that do not violate memory, even on runtime. We yet have to discover a safe alternative for C++ iterators.

* Sane (noun): Without using `Rc<RefCell<_>>` or `Arc<Mutex<_>>`

As an alternative to standard library, many people suggest the linked-list crate [4]. This crate has something that is called a `Cursor`, which looks like a C++ iterator:

As an alternative to standard library, many people suggest the linked-list crate [4]. This crate has something that is called a `Cursor`, which looks like a C++ iterator:

```
fn reset  (&mut self)
fn next   (&mut self) -> Option<&mut T>
fn prev   (&mut self) -> Option<&mut T>
fn insert (&mut self, elem: T)
fn remove (&mut self) -> Option<T>
fn split  (&mut self) -> LinkedList<T>
fn splice (&mut self, other: &mut LinkedList<T>)
```

But still, we are halfway there. The method that gives as a `Cursor` of a linked_list::LinkedList is as follows (without lifetime elision):

But still, we are halfway there. The method that gives as a `Cursor` of a linked_list::LinkedList is as follows (without lifetime elision):

```
fn cursor(&'a mut self) -> Cursor<'a, T>
```

But still, we are halfway there. The method that gives as a `Cursor` of a linked_list::LinkedList is as follows (without lifetime elision):

```
fn cursor(&'a mut self) -> Cursor<'a, T>
```

Notice that this function borrows self (i.e. our list) as mut, and gives the `Cursor` the lifetime of the mutable reference. This means that, as long as the `Cursor` lives, it holds a mutable (i.e. exclusive) borrow on the LinkedList.

But still, we are halfway there. The method that gives as a `Cursor` of a linked_list::LinkedList is as follows (without lifetime elision):

```
fn cursor(&'a mut self) -> Cursor<'a, T>
```

Notice that this function borrows self (i.e. our list) as mut, and gives the `Cursor` the lifetime of the mutable reference. This means that, as long as the `Cursor` lives, it holds a mutable (i.e. exclusive) borrow on the LinkedList.

So, we cannot call this function on the same LinkedList a second time without first dropping the cursor. Therefore, we cannot do the first pass - keep best position - insert there magic stuff, because this requires keeping two cursors, the current and the best position.

Can we do better?

Can we do better?

"Simplicity does not precede complexity, but follows it"
Alan Perlis

First of all, let's tackle the linked list datatype. We obviously *cannot* have a full-fledged C++ linked list. We cannot have stray elements throughout the memory and expect to be able to validate on runtime if a pointer pointing to one of those elements is a valid pointer, or points to free memory.

First of all, let's tackle the linked list datatype. We obviously *cannot* have a full-fledged C++ linked list. We cannot have stray elements throughout the memory and expect to be able to validate on runtime if a pointer pointing to one of those elements is a valid pointer, or points to free memory.

So, after researching code found on the internet [2], let's use a Vec for the storage. Later, we will discuss the drawbacks.

Actually, the idea is so simple you will probably feel betrayed:

Actually, the idea is so simple you will probably feel betrayed:

- Each element has two *indices* pointing to the previous and next element.

Actually, the idea is so simple you will probably feel betrayed:

- Each element has two *indices* pointing to the previous and next element.
- On deletion from the middle, do not shift the elements. Instead, add the deleted index on a second "free memory" vector.

Actually, the idea is so simple you will probably feel betrayed:

- Each element has two *indices* pointing to the previous and next element.
- On deletion from the middle, do not shift the elements. Instead, add the deleted index on a second “free memory” vector.
- On insert, if the free memory has an element then insert there, else insert on the end.

$O(1)$ splitting and merging is lost. Can we gain it back?

$O(1)$ splitting and merging is lost. Can we gain it back?

Sure. If the lists that will get splitted and merged share the element Vec and the free memory Vec.

$O(1)$ splitting and merging is lost. Can we gain it back?

Sure. If the lists that will get splitted and merged share the element Vec and the free memory Vec.

Of course, this being Rust, our implementation will be different if we need these lists to be thread safe or not. (`Rc<RefCell<T>>` vs `Arc<Mutex<T>>` for the initiated)

$O(1)$ splitting and merging is lost. Can we gain it back?

Sure. If the lists that will get splitted and merged share the element Vec and the free memory Vec.

Of course, this being Rust, our implementation will be different if we need these lists to be thread safe or not. (`Rc<RefCell<T>>` vs `Arc<Mutex<T>>` for the initiated)

Remember, lists instead of vectors are *optimizations*, therefore a non-generalized custom implementation is warranted.

Did we lose anything else?

Did we lose anything else?

Yes. List deletion does not delete an element. No destructor is ever called!

Did we lose anything else?

Yes. List deletion does not delete an element. No destructor is ever called!

So, the verdict is, do not hold file handles or MutexGuards on a list. Hold indices to a vector if you must.

Did we lose anything else?

Yes. List deletion does not delete an element. No destructor is ever called!

So, the verdict is, do not hold file handles or MutexGuards on a list. Hold indices to a vector if you must.

Actually, in my opinion, lists make sense as *short lived* data structures that live while an algorithm runs, and should dissolve as soon as it finishes.

C++ iterators have an implicit hierarchy:

C++ iterators have an implicit hierarchy:

- All iterators can be incremented.

C++ iterators have an implicit hierarchy:

- All iterators can be incremented.
- Input iterators can be checked for (in)equality and dereferenced as rvalues.
- Output iterators can be dereferenced as lvalues.

C++ iterators have an implicit hierarchy:

- All iterators can be incremented.
- Input iterators can be checked for (in)equality and dereferenced as rvalues.
- Output iterators can be dereferenced as lvalues.
- Forward iterators are Input + Output iterators that can be dereferenced multiple times.
- Bidirectional iterators are Forward iterators that can also be decremented.

C++ iterators have an implicit hierarchy:

- All iterators can be incremented.
- Input iterators can be checked for (in)equality and dereferenced as rvalues.
- Output iterators can be dereferenced as lvalues.
- Forward iterators are Input + Output iterators that can be dereferenced multiple times.
- Bidirectional iterators are Forward iterators that can also be decremented.
- Random access iterators are Bidirectional iterators that can also be added, subtracted, checked for ordering and indexed.

When someone needs to hold persistent references to virtually anything in Rust, the answer always seems to be “use indices”. So, why cannot we generalize indices?

When someone needs to hold persistent references to virtually anything in Rust, the answer always seems to be “use indices”. So, why cannot we generalize indices?

```
pub trait ImmutableIndex<Idx>
{
    fn begin(&self) -> Idx;
    fn end(&self) -> Idx;
    fn valid(&self, &Idx) -> bool;
}

pub trait ForwardIndex<Idx>: ImmutableIndex<Idx>
{
    fn increment(&self, &mut Idx);
}
```

```rust
pub trait BackwardIndex<Idx>: ImmutableIndex<Idx>
{
    fn decrement(&self, &mut Idx);
}

pub trait BidirectionalIndex<Idx>:
    ForwardIndex<Idx> + BackwardIndex<Idx> {}

impl<Idx, T> BidirectionalIndex<Idx> for T
    where T: ForwardIndex<Idx> + BackwardIndex<Idx> {}
```

```rust
impl<T> ImmutableIndex<usize> for Vec<T>
{
    fn begin(&self) -> usize { 0 }
    fn end(&self) -> usize { self.len() - 1 }
    fn valid(&self, i: &usize) -> bool { *i < self.len() }
}

impl<T> ForwardIndex<usize> for Vec<T>
{
    fn increment(&self, i: &mut usize) { *i += 1; }
}

impl<T> BackwardIndex<usize> for Vec<T>
{
    fn decrement(&self, i: &mut usize) { *i = i.wrapping_sub(1); }
}
```

With these definitions, we can write generic code over collections:

With these definitions, we can write generic code over collections:

```
fn find_first_index_eq_to_0<Col, Idx>(col: &Col) -> Idx
    where Col: ForwardIndex<Idx> + Index<Idx, Output=i32>,
          Idx: Clone
{
    let mut curr = collection.begin();

    while collection.valid(&curr)
    {
        if collection[curr.clone()] == 0 {
            return curr;
        }
        collection.increment(&mut curr);
    }

    panic!("Did not find zero element!");
}
```

Can we do even more with this abstraction?

Can we do even more with this abstraction?

- We can guarantee that an index won't be used on the wrong collection. This can be done by embedding a GUID in each collection, which is passed to the Index when begin/end is called.

Can we do even more with this abstraction?

- We can guarantee that an index won't be used on the wrong collection. This can be done by embedding a GUID in each collection, which is passed to the Index when begin/end is called.
- We can guarantee that indexes that are tested for equality belong to the same collection, the same way.

Can we do even more with this abstraction?

- We can guarantee that an index won't be used on the wrong collection. This can be done by embedding a GUID in each collection, which is passed to the Index when begin/end is called.
- We can guarantee that indexes that are tested for equality belong to the same collection, the same way.
- We can guarantee that an index points to an element that has not been deleted. In order to do this, we need an increasing number (a timestamp if you like) that will be increased on each element deletion, and checked on each element access.

Can we do even more with this abstraction?

- We can guarantee that an index won't be used on the wrong collection. This can be done by embedding a GUID in each collection, which is passed to the Index when begin/end is called.
- We can guarantee that indexes that are tested for equality belong to the same collection, the same way.
- We can guarantee that an index points to an element that has not been deleted. In order to do this, we need an increasing number (a timestamp if you like) that will be increased on each element deletion, and checked on each element access.

Of course, the previous cannot be implemented for already defined containers. The orphan rule [5] is why we can't have nice things. The only way to overcome this is to have these Index Traits in the standard library.

Questions?

Bjarne stroustrup: Why you should avoid linked lists.
`https://www.youtube.com/watch?v=YQs6IC-vgmo`.

ixlist.
`https://bluss.github.io/ixlist/target/doc/ixlist/struct.List.html`.

Learning rust with entirely too many linked lists.
`https://cglab.ca/~abeinges/blah/too-many-lists/book/`.

linked-list on crates.io.
`https://crates.io/crates/linked-list`.

Rust book on traits (orphan rule).
`https://doc.rust-lang.org/book/ch10-02-traits.html`.