Rust-101, Part 16: Unsafe Rust, Drop

As we saw, the rules Rust imposes to ensure memory safety can get us pretty far. A large amount of programming patterns can be written within safe Rust, and, more importantly, library code like iterators or threads can make use of the type system to ensure some level of correctness beyond basic memory safety.

However, there will still be programs that one cannot write in accordance with the borrow checker. And there will be cases where it may be possible to satisfy the compiler, but only at the cost of some run-time overhead, as we saw with `RefCell` - overhead which may not be acceptable. In such a situation, it is possible to use *unsafe* Rust: That's a part of the language that is *known* to open the gate to invalid pointer access and all other sorts of memory safety. Of course, `unsafe` also means "Here Be Dragons": You are on your own now.
The goal in these cases is to confine unsafety to the local module. Types like `Rc` and `Vec` are implemented using unsafe Rust, but *using* them as we did is (believed to be) perfectly safe.

Unsafe Code

As an example, let us write a doubly-linked list. Clearly, such a data-structure involves aliasing and mutation: Every node in the list is pointed to by its left and right neighbor, but still we will want to modify the nodes. We could now try some clever combination of `Rc` and `RefCell`, but this would end up being quite annoying - and it would incur some overhead. For a low-level data-structure like a doubly-linked list, it makes sense to implement an efficient version once, that is unsafe internally, but that can be used without any risk by safe client code.

As usually, we start by defining the types. Everything is parameterized by the type `T` of the data stored in the list. A node of the list consists of the data, and two node pointers for the predecessor and successor.

```
use std::ptr;
use std::mem;
use std::marker::PhantomData;
```

```
struct Node<T> {
    next: NodePtr<T>,
    prev: NodePtr<T>,
    data: T,
}
type NodePtr<T> = *mut Node<T>;
```

A node pointer is a *mutable raw pointer* to a node. Raw pointers (`*mut T` and `*const T`) are the Rust equivalent of pointers in C. Unlike references, they do not come with any guarantees: Raw pointers can be null, or they can point to garbage. They don't have a lifetime, either.

```
pub struct LinkedList<T> {
    first: NodePtr<T>,
    last:  NodePtr<T>,
    _marker: PhantomData<T>,
}
```

The linked list itself stores pointers to the first and the last node. In addition, we tell Rust that this type will own data of type `T`. The type `PhantomData<T>` does not actually store anything in memory - it has size zero. However, logically, Rust will consider a `T` to be present. In this case, Rust knows that data of type `T` may be dropped whenever a `LinkedList<T>` is dropped. Dropping has a lot of subtle checks to it, making sure that things can't go wrong. For this to work, Rust needs to know which types could potentially be dropped. In safe Rust, this can all be inferred automatically, but here, we just have a `*mut Node<T>`, and we need to tell Rust that we actually own such data and will drop it. (For more of the glory details, see this RFC.)

```
unsafe fn raw_into_box<T>(r: *mut T) -> Box<T> {
    mem::transmute(r)
}
```

Before we get to the actual linked-list methods, we write two short helper functions converting between mutable raw pointers, and boxed data. Both employ `mem::transmute`, which can convert anything to anything, by just re-interpreting the bytes. Clearly, that's an unsafe operation and must only be used with great care - or even better, not at all. Seriously. If at all possible, you should never use `transmute`.
We are making the assumption here that a `Box` and a raw pointer have the same representation in memory. In the future, Rust will provide such operations in the standard library, but the exact API is still being fleshed out.

```
fn box_into_raw<T>(b: Box<T>) -> *mut T {
    unsafe { mem::transmute(b) }
}
```

We declare `raw_into_box` to be an `unsafe` function, telling Rust that calling this function is not generally safe. This grants us the unsafe powers for the body of the function: We can dereference raw pointers, and - most importantly - we can call unsafe functions. (The other unsafe powers won't be relevant here. Go read The Rustonomicon if you want to learn all about this, but be warned - That Way Lies Madness.)
Here, the caller will have to ensure that `r` is a valid pointer, and that nobody else has a pointer to this data.

The case is slightly different for `box_into_raw`: Converting a `Box` to a raw pointer is always safe. It just drops some information. Hence we keep the function itself safe, and use an *unsafe block* within the function. This is an (unchecked) promise to the Rust compiler, saying that a safe invocation of `box_into_raw` cannot go wrong. We also have the unsafe powers in the unsafe block.

```
impl<T> LinkedList<T> {

    pub fn new() -> Self {
        LinkedList { first: ptr::null_mut(), last: ptr::null_mut(), _marker: PhantomData }
    }
```

A new linked list just contains null pointers. `PhantomData` is how we construct any `PhantomData<T>`.

```
    pub fn push_back(&mut self, t: T) {

        let new = Box::new( Node { data: t, next: ptr::null_mut(), prev: self.last } );
        let new = box_into_raw(new);
```

This function adds a new node to the end of the list.

Create the new node, and make it a raw pointer. Calling `box_into_raw` gives up ownership of the box, which is crucial: We don't want the memory that it points to to be deallocated!

```
        if self.last.is_null() {
            debug_assert!(self.first.is_null());

            self.first = new;
        } else {
            debug_assert!(!self.first.is_null());

            unsafe { (*self.last).next = new; }
        }
```

Update other pointers to this node.

```
        self.last = new;
    }
```

The list is currently empty, so we have to update the head pointer.

We have to update the `next` pointer of the tail node. Since Rust does not know that a raw pointer actually points to anything, dereferencing such a pointer is an unsafe operation. So this unsafe block promises that the pointer will actually be valid.

Make this the last node.

Exercise 16.1: Add some more operations to `LinkedList`: `pop_back`, `push_front` and `pop_front`. Add testcases for `push_back` and all of your functions. The `pop` functions should take `&mut self` and return `Option<T>`.

```
    pub fn iter_mut(&mut self) -> IterMut<T> {
        IterMut { next: self.first, _marker: PhantomData  }
    }
}
```

Next, we are going to provide an iterator. This function just creates an instance of `IterMut`, the iterator type which does the actual work.

What does the iterator need to store? Strictly speaking, all it needs is the pointer to the next node that it is going to visit. However, how do we make sure that this pointer remains valid? We have to get this right ourselves, as we left the safe realms of borrowing and ownership. Remember that the key ingredient for iterator safety was to tie the lifetime of the iterator to the lifetime of the reference used for `iter_mut`. We will thus give `IterMut` two parameters: A type parameter specifying the type of the data, and a lifetime parameter specifying for how long the list was borrowed to the iterator.

```
pub struct IterMut<'a, T> where T: 'a {
    next: NodePtr<T>,
    _marker: PhantomData<&'a mut LinkedList<T>>,
}
```

For Rust to accept the type, we have to add two more annotations. First of all, we have to ensure that the data in the list lives at least as long as the iterator: If you drop the `T: 'a`, Rust will tell you to add it back. And secondly, Rust will complain if `'a` is not actually used in the struct. It doesn't know what it is supposed to do with that lifetime. So we use `PhantomData` again to tell it that in terms of ownership, this type actually (uniquely) borrows a linked list. This has no operational effect, but it means that Rust can deduce the intent we had when adding that seemingly useless lifetime parameter.

```
impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {

        if self.next.is_null() {
            None
        } else {

            let next = unsafe { &mut *self.next };
            let ret = &mut next.data;
            self.next = next.next;
            Some(ret)
        }
    }
}
```

When implementing `Iterator` for `IterMut`, the fact that we have the lifetime `'a` around immediately pays of: We would not even be able to write down the type `Item` without that lifetime.

The actual iteration is straight-forward: Once we reached a null pointer, we are done.

Otherwise, we can convert the next pointer to a reference, get a reference to the data and update the iterator.

In `next` above, we made crucial use of the assumption that `self.next` is either null or a valid pointer. This only works because if someone tries to delete elements from a list during iteration, we know that the borrow checker will catch them: If they call `next`, the lifetime `'a` we artificially added to the iterator has to still be active, which means the mutable reference passed to `iter_mut` is still active, which means nobody can delete anything from the list. In other words, we make use of the expressive type system of Rust, decorating our own unsafe implementation with just enough information so that Rust can check *uses* of the linked- list. If the type system were weaker, we could not write a linked-list like the above with a safe interface!

Exercise 16.2: Add a method `iter` and a type `Iter` providing iteration for shared references. Add testcases for both kinds of iterators.

Drop

The linked list we wrote is already working quite nicely, but there is one problem: When the list is dropped, nobody bothers to deallocate the remaining nodes. Even worse, if `T` itself has a destructor that needs to clean up, it is not called for the element remaining in the list. We need to take care of that ourselves.

```
impl<T> Drop for LinkedList<T> {
```

In Rust, adding a destructor for a type is done by implementing the `Drop` trait. This is a very special trait. It can only be implemented for *nominal types*, i.e., you cannot implement `Drop` for `&mut T`. You also cannot restrict the type and lifetime parameters further than the type does - the `Drop` implementation has to apply to *all* instances of `LinkedList`.

```
    fn drop(&mut self) {
        let mut cur_ptr = self.first;
        while !cur_ptr.is_null() {
```

The destructor itself is a method which takes `self` in mutably borrowed form. It cannot own `self`, because then the destructor of `self` would be called at the end of the function, resulting in endless recursion.

```
            let cur = unsafe { raw_into_box(cur_ptr) };
            cur_ptr = cur.next;
            drop(cur);
        }
    }
}
```

In the destructor, we just iterate over the entire list, successively obtaining ownership (`Box`) of every node. When the box is dropped, it will call the destructor on `data` if necessary, and subsequently free the node on the heap. We call `drop` explicitly here just for documentation purposes.

The End

Congratulations! You completed Rust-101. This was the last part of the course. I hope you enjoyed it. If you have feedback or want to contribute yourself, please head to the Rust-101 website fur further information. The entire course is open-source (under CC-BY-SA 4.0).

If you want to do more, the examples you saw in this course provide lots of playground for coming up with your own little extensions here and there. The index contains some more links to additional resources you may find useful. With that, there's only one thing left to say: Happy Rust Hacking!