Rust programming

Course – 10

Gavrilut Dragos

rev 7

Agenda for today

1. Unsafe blocks
2. NonNull pointers
3. Interior Mutability
4. Reference Count

Unsafe Rust

Unsafe Rust

Unsafe Rust is a mechanism that one can use to enable a set of features / behavior of the program that have the potential to trigger a problem / bug in your program if not treated correctly. For this type of behavior, Rust uses a special keyword: **unsafe** that can be used to declare a block or a function where the safety rules don't apply.

Cases where unsafe can be used:

- To work with regular pointers (just like C/C++)
- To modify mutable global variables
- To run a method from an external module that was not compiled with Rust (e.g. a module compiled in C/C++, a system API, etc)
- To implement an unsafe trait
- To access a field of a union (similar to the concept of union from C/C++)

Unsafe Rust

A pointer in rust can be defined in the following way:

- *const <type> → for constant pointers (equivalent to const <type>* from C/C++)
- *mut <type> → for non-constant (mutable) pointers (equivalent to <type>* from C/C++)

In Rust, pointers are not limited by any ownership / borrowing rules. As such, they are considered unsafe. Let's analyze the following code:

Rust

```rust
fn main() {
    let x = 10;
    let y = &x as *const i32;
    println!("{}", *y );
}
```

Error

```
error[E0133]: dereference of raw pointer is unsafe and requires unsafe function or block
 --> src\main.rs:4:20
  |
4 |     println!("{}", *y );
  |                    ^^ dereference of raw pointer
  |
  = note: raw pointers may be null, dangling or unaligned; they can violate aliasing rules
and cause data races: all of
these are undefined behavior
```

The main problem in this case is that we try to dereference a pointer (read the value from that pointer) and this is considered unsafe in Rust.

Unsafe Rust

A pointer in rust can be defined in the following way:

- *const <type> → for constant pointers (equivalent to const <type>* from C/C++)
- *mut <type> → for non-constant (mutable) pointers (equivalent to <type>* from C/C++)

In Rust, pointers are not limited by any ownership / borrowing rules. As such, they are considered unsafe. Let's analyze the following code:

Rust

```rust
fn main() {
    let x = 10;
    let y = &x as *const i32;
    println!("{}", unsafe { *y } );
}
```

Rust

```rust
fn main() {
    let x = 10;
    let y = &x as *const i32;
    unsafe { println!("{}", *y ); }
}
```

Output

```
10
```

The solution in this case is to use the **unsafe** keyword to enable reading a value from a pointer.

Unsafe Rust

One can use pointers to avoid various safety protocols:

1. ***Convert a const pointer to a mutable pointer***

Rust

```
fn main() {
    let x = 10;
    unsafe {
        let y = (&x as *const i32) as *mut i32;
        *y = 11;
    };
    println!("{}",x);
}
```

Output
11

In this example, even if "x" is defined as immutable, we can create a mutable pointer towards it and change its value. In reality, this technique is quite dangerous as "x" might be located on a read-only memory page and trying to write something at that location might crash the program.

Unsafe Rust

One can use pointers to avoid various safety protocols:

2. ***Accessing a memory allocated to a type with a pointer of another type***

Rust

```
fn main() {
    let x = 10;
    unsafe {
        let y = (&x as *const i32) as *mut i8;
        *y = 11;
    };
    println!("{}",x);
}
```

Output
11

In this example, “x” is a pointer of type i32, but we are going to access its memory via “y” (a pointer of type i8). This is considered to be unsafe as you can write data outside allocated memory space/breaking invariants.

Unsafe Rust

One can use pointers to avoid various safety protocols:

3. ***Apply pointer arithmetic's***

Rust

```
fn main() {
    let x = 10;
    unsafe {
        let y = (&x as *const i32) as *mut i8;
        *(y.add(1)) = 11;
    };
    println!("{}",x);
}
```

Output
2826

We can also apply pointer arithmetic's via specialized functions such as .add(...) or .sub(...). In our case, we will set the next 8 bytes of "x" to value 11, making the total value of "x" to be 10 + 11 * 256 = 2826. The main risk here is that using pointer arithmetic's might move a pointer to an unallocated memory space.

Unsafe Rust

One can use pointers to avoid various safety protocols:

4. ***Create a variable that points to a hardcoded memory location***

Rust

```
fn main() {
    let x = unsafe { &*(0x123458 as *const i32) };
    println!("{}",x);
}
```

Runtime Error

```
error: process didn't exit successfully:
`target\debug\rust_tester.exe` (exit code: 0xc0000005,
STATUS_ACCESS_VIOLATION)
```

In this case "x" is a immutable reference (**&i32**) that points to an invalid address (0x123458).

As a result, when trying to read the value of "x" a runtime error (crash) will happen. Keep in mind that these error rely on the fact that on most system there should be no memory allocated around 0x123458. However, this is an undefined behavior (UB) as it is theoretically possible to find a memory page allocated at that memory with a read right and in this case the program might not crash.

Unsafe Rust

One can use pointers to avoid various safety protocols:

5. ***Cast between different structures / types***

Rust

```
#[repr(C)]
struct IP { values: [u8; 4], }
fn main() {
    let i = IP {
        values: [127, 0, 0, 1],
    };
    let n = unsafe { *(((&i) as *const IP) as *const u32)};
    println!("{}", n);
}
```

Output

16777343

C++ (equivalent code)

```
struct IP {
    unsigned char value[4];
};

void main() {
    IP i = {127,0,0,1};
    unsigned int n = *(unsigned int*)&i;
    std::cout << n;
}
```

In this case, we convert the address of structure IP to a pointer of type *const u32 and then we read the value from there. This means that the value read (on little endian) architecture will be: $127 + 0 \times 2^{8} + 0 \times 2^{16} + 1 \times 2^{24} = 16777343$

Unsafe Rust

One can use pointers to avoid various safety protocols:

5. ***Cast between different structures / types***

Rust

```rust
#[repr(C)]
struct IP { va
fn main() {
    let i = IP {
        values: [127, 0, 0, 1],
    };
    let n = unsafe { *(((&i) as *const IP) as *const u32)};
    println!("{}", n);
}
```

Notice **#[repr(C)]** : This attribute tells rust to represent the data in this structure just like **C language** does. This is important because a side effect of this attribute is that it does not allow Rust to change the order of the members of the structure or to align them in a different way.

```cpp
};

void main() {
    IP i = {127,0,0,1};
    unsigned int n = *(unsigned int*)&i;
    std::cout << n;
}
```

In this case, we convert the address of structure IP to a pointer of type *const u32 and then we read the value from there. This means that the value read (on little endian) architecture will be: $127 + 0 \times 2^8 + 0 \times 2^{16} + 1 \times 2^{24} = 16777343$

Unsafe Rust

One can use pointers to avoid various safety protocols:

6. ***Evade ownership rules***

Rust

```
fn main() {
    let mut s = String::from("ABC");
    let mut s2 = unsafe { String::from_raw_parts((&mut s).as_mut_ptr(),s.len(),s.capacity()) };
    s2.clear();
    s2.push_str("123");
    println!("{},{}", s,s2);
}
```

So, let's see what happens in this case:

Unsafe Rust

One can use pointers to avoid various safety protocols:

6. *Evade ownership rules*

Rust

```
fn main() {
    let mut s = String::from("ABC");
    let mut s2 = unsafe { String::from_raw_parts((&mut s).as_mut_ptr(),s.len(),s.capacity()) };
    s2.clear();
    s2.push_str("123");
    println!("{},{}", s,s2);
}
```

So, let's see what happens in this case:

1. We create s2 from a raw pointer → this translates that both s as s2 use the same memory

Unsafe Rust

One can use pointers to avoid various safety protocols:

6. *Evade ownership rules*

Rust

```
fn main() {
    let mut s = String::from("ABC");
    let mut s2 = unsafe { String::from_raw_parts((&mut s).as_mut_ptr(),s.len(),s.capacity()) };
    s2.clear();
    s2.push_str("123");
    println!("{},{}", s,s2);
}
```

So, let's see what happens in this case:

1. We create s2 from a raw pointer → this translates that both s as s2 use the same memory
2. We clear the memory allocated for s2 (and we replace it with another string "123"). Notice that the new string has the same size as the previous one → this way we make sure that the pointer will not change, and the values from s will remain valid (length & capacity)

Unsafe Rust

One can use pointers to avoid various safety protocols:

6. *Evade ownership rules*

Rust

```
fn main() {
    let mut s = String::from("ABC");
    let mut s2 = unsafe { String::from_raw_parts((&mut s).as_mut_ptr(),s.len(),s.capacity()) };
    s2.clear();
    s2.push_str("123");
    println!("{},{}", s,s2);
}
```

Output

123, 123

So, let's see what happens in this case:

1. We create s2 from a raw pointer → this translates that both s as s2 use the same memory
2. We clear the memory allocated for s2 (and we replace it with another string "123"). Notice that the new string has the same size as the previous one → this way we make sure that the pointer will not change, and the values from s will remain valid (length & capacity)
3. Print both s and s2 (as they have the same pointer → they will have the same value: 123)

Unsafe Rust

One can use pointers to avoid various safety protocols:

6. *Evade ownership rules*

Rust

```
fn main() {
    let mut s = String::from("ABC");
    let mut s2 = unsafe { String::from_raw_parts((&mut s).as_mut_ptr(),s.len(),s.capacity()) };
    s2.clear();
    s2.push_str("123");
    println!("{},{}", s,s2);
}
```

Output

123, 123

Runtime Error

```
error: process didn't exit successfully:
`target\debug\rust_tester.exe` (exit code: 0xc0000374,
STATUS_HEAP_CORRUPTION)
```

So, let's see what happens in this case:

1. We create s2 from a raw pointer → this translates that both s as s2 use the same memory
2. We clear the memory allocated for s2 (and we replace it with another string "123"). Notice that the new string has the same size as the previous one → this way we make sure that the pointer will not change, and the values from s will remain valid (length & capacity)
3. Print both s and s2 (as they have the same pointer → they will have the same value: 123)
4. Deallocate memory for s and s2 (since they have the same pointer, a runtime error will occur)

Unsafe Rust

One can use pointers to avoid various safety protocols:

7. ***Evade borrowing rules***

Rust

```rust
#[derive(Debug)]
struct MyData {
    text: String,
    value: i32,
}
fn main() {
    let i = MyData {
        text: "abc".to_string(),
        value: 5,
    };
    let ref_i = &i;
    let ref_mut_i = unsafe { &mut *((&i) as (*const MyData) as (*mut MyData)) };
    ref_mut_i.value = 10;
    ref_mut_i.text.push_str("123");
    println!("{:?},{:?}", ref_i, ref_mut_i);
}
```

Output

```
MyData { text: "abc123", value: 10 },MyData { text: "abc123", value: 10 }
```

Unsafe Rust

One can use pointers to avoid various safety protocols:

7. *Evade borrowing rules*

Rust

```
#[derive(Debug)]
struct MyData {
    text: String,
    value: i32,
}
fn main() {
    let i = MyData {
        text: "abc".to_string(),
        value: 5,
    };
    let ref_i = &i;
    let ref_mut_i = unsafe { &mut *((&i) as (*const MyData) as (*mut MyData)) };
    ref_mut_i.value = 10;
    ref_mut_i.text.push_str("123");
    println!("{:?},{:?}", ref_i, ref_mut_i);
}
```

Output

```
MyData { text: "abc123", value: 10 },MyData { text: "abc123", value: 10 }
```

As stated in previous course, borrowing rules forbit the usage/existence of an immutable and mutable reference at the same time

Unsafe Rust

One can use pointers to avoid various safety protocols:

7. *Evade borrowing rules*

However, using **unsafe** we can evade those rules by converting a reference to a pointer and then back to a reference. We can even avoid the fact that i is ***immutable***. Its also important to highlight this is an undefined behavior (UB) as for immutable data Rust might decide to optimize them and not store them on the stack resulting in an invalid pointer if trying to obtain one.

value: 10 }

```
}
fn main() {
    let i = MyData {
        text: "abc".to_string(),
        value: 5,
    };
    let ref_i = &i;
    let ref_mut_i = unsafe { &mut *((&i) as (*const MyData) as (*mut MyData)) };
    ref_mut_i.value = 10;
    ref_mut_i.text.push_str("123");
    println!("{:?},{:?}", ref_i, ref_mut_i);
}
```

Unsafe Rust

One can use pointers to avoid various safety protocols:

7. ***Evade borrowing rules***

We can even create generic function that creates a mutable reference:

Rust

```
fn get_ref_mut<T>(ref_to_T: &T) -> &mut T {
    unsafe {
        let ptr = ref_to_T as *const T;
        let mut_ptr = ptr as *mut T;
        return &mut *mut_ptr;
    }
}
fn main() {
    let s = String::from("abc");
    let ref_mut_1 = get_ref_mut(&s);
    let ref_mut_2 = get_ref_mut(&s);
    ref_mut_1.push('1');
    ref_mut_2.push('2');
    println!("{},{}", ref_mut_1, ref_mut_2);
}
```

Output

```
abc12,abc12
```

Unsafe Rust

One can use pointers to avoid various safety protocols:

7. ***Evade borrowing rules***

We can even create generic function that creates a mutable reference:

Rust

```
fn get_ref_mut<T>(ref_to_T: &T) -> &mut T {
    unsafe {
        let ptr = ref_to_T as *const T;
        let mut_ptr = ptr as *mut T;
        return &mut *mut_ptr;
    }
}
fn main() {
    let s = String::from("abc");
    let ref_mut_1 = get_ref_mut(&s);
    let ref_mut_2 = get_ref_mut(&s);
    ref_mut_1.push('1');
    ref_mut_2.push('2');
    println!("{},{}", ref_mut_1, ref_mut_2);
}
```

Output

Step 1: we obtain a ***const pointer*** from any immutable reference

Unsafe Rust

One can use pointers to avoid various safety protocols:

7. ***Evade borrowing rules***

We can even create generic function that creates a mutable reference:

Rust

```
fn get_ref_mut<T>(ref_to_T: &T) -> &mut T {
    unsafe {
        let ptr = ref_to_T as *const T;
        let mut_ptr = ptr as *mut T;
        return &mut *mut_ptr;
    }
}
fn main() {
    let s = String::from("abc");
    let ref_mut_1 = get_ref_mut(&s);
    let ref_mut_2 = get_ref_mut(&s);
    ref_mut_1.push('1');
    ref_mut_2.push('2');
    println!("{},{}", ref_mut_1, ref_mut_2);
}
```

Output

Step 2: As there are no restriction for any pointer operations, we can convert the constant pointer (obtained in the previous step) into a mutable pointer.

Unsafe Rust

One can use pointers to avoid various safety protocols:

7. ***Evade borrowing rules***

We can even create generic function that creates a mutable reference:

Rust

```
fn get_ref_mut<T>(ref_to_T: &T) -> &mut T {
    unsafe {
        let ptr = ref_to_T as *const T;
        let mut_ptr = ptr as *mut T;
        return &mut *mut_ptr;
    }
}
fn main() {
    let s = String::from("abc");
    let ref_mut_1 = get_ref_mut(&s);
    let ref_mut_2 = get_ref_mut(&s);
    ref_mut_1.push('1');
    ref_mut_2.push('2');
    println!("{},{}", ref_mut_1, ref_mut_2);
}
```

Output

Step 3: Reconvert the mutable pointer obtained in the previous step into a mutable reference. Since we have been using pointer operations, the borrow checker mechanism will not link the new mutable reference to the object resulting in the ability to create multiple mutable references.

Unsafe Rust

One can use pointers to avoid various safety protocols:

8. *Create unsafe methods / functions*

Rust

```rust
fn get_ref_mut<T>(ref_to_T: &T) -> &mut T {
    unsafe {
        let ptr = ref_to_T as *const T;
        let mut_ptr = ptr as *mut T;
        return &mut *mut_ptr;
    }
}
fn main() {
    let s = String::from("abc");
    let ref_mut_1 = get_ref_mut(&s);
    let ref_mut_2 = get_ref_mut(&s);
    ref_mut_1.push('1');
    ref_mut_2.push('2');
    println!("{},{}", ref_mut_1, ref_mut_2);
}
```

In this case you can see that the entire content of function get_ref_mut is unsafe. Furthermore, there are cases where a function/method is unsafe. The main problem here is that one might look at the code from **main** function and *consider that it is safe* (pretty much assume that the whatever **get_ref_mut** function returns is safe).

To avoid these cases, some functions / method can be marked as **unsafe** and as such they can not be used normally (without unsafe block) in Rust.

Unsafe Rust

One can use pointers to avoid various safety protocols:

8. ***Create unsafe methods / functions***

Rust

```
fn get_ref_mut<T>(ref_to_T: &T) -> &mut T {
    unsafe {
        let ptr = ref_to_T as *const T;
        let mut_ptr = ptr as *mut T;
        return &mut *mut_ptr;
    }
}
fn main() {
    let s = String::from("abc");
    let ref_mut_1 = get_ref_mut(&s);
    let ref_mut_2 = get_ref_mut(&s);
    ref_mut_1.push('1');
    ref_mut_2.push('2');
    println!("{},{}", ref_mut_1, ref_mut_2);
}
```

Rust (unsafe function)

```
unsafe fn get_ref_mut<T>(ref_to_T: &T) -> &mut T {
    let ptr = ref_to_T as *const T;
    let mut_ptr = ptr as *mut T;
    return &mut *mut_ptr;
}


fn main() {
    let s = String::from("abc");
    let ref_mut_1 = unsafe { get_ref_mut(&s) };
    let ref_mut_2 = unsafe { get_ref_mut(&s) };
    ref_mut_1.push('1');
    ref_mut_2.push('2');
    println!("{},{}", ref_mut_1, ref_mut_2);
}
```

Unsafe Rust

One can use pointers to avoid various safety protocols:

8. ***Create unsafe methods / functions***

Rust

```
fn get_ref_mut<T>(ref_to_T: &T) -> &mut T {
    unsafe {
        let ptr = ref_to_T as *const T;
        let mut_ptr = ptr as *mut T;
        return &mut *mut_ptr;
    }
}
fn main() {
    let s = String:
    let ref_mut_1 =
    let ref_mut_2 =
    ref_mut_1.push('1');
    ref_mut_2.push('2');
    println!("{},{}", ref_mut_1, ref_mut_2);
}
```

Rust (unsafe function)

```
unsafe fn get_ref_mut<T>(ref_to_T: &T) -> &mut T {
    let ptr = ref_to_T as *const T;
    let mut_ptr = ptr as *mut T;
    return &mut *mut_ptr;

_ref_mut(&s) };
_ref_mut(&s) };
    ref_mut_1.push('1');
    ref_mut_2.push('2');
    println!("{},{}", ref_mut_1, ref_mut_2);
}
```

Notice that if **unsafe** is used on front of a method/function, you don't need to use it in a block inside the function/method. ***Keep in mind that in edition 2024 this behavior might change.***

Unsafe Rust

One can use pointers to avoid various safety protocols:

8. ***Create unsafe methods / functions***

Rust

Notice that we can no longer use get_ref_mut method as it is (in a safe mode). Instead, we have to use it within an **unsafe** block. This underlines the idea that the output of that function is unsafe and has to be interpreted as such.

```rust
fn get_ref_mut<T>(ref_to_T: &
    unsafe {
        let ptr = ref_to_T a
        let mut_ptr = ptr as
        return &mut *mut_ptr;
    }
}
fn main() {
    let s = String::from("abc");
    let ref_mut_1 = get_ref_mut(&s);
    let ref_mut_2 = get_ref_mut(&s);
    ref_mut_1.push('1');
    ref_mut_2.push('2');
    println!("{},{}", ref_mut_1, ref_mut_2);
}
```

```rust
&mut T {
}
fn main() {
    let s = String::from("abc");
    let ref_mut_1 = unsafe { get_ref_mut(&s) };
    let ref_mut_2 = unsafe { get_ref_mut(&s) };
    ref_mut_1.push('1');
    ref_mut_2.push('2');
    println!("{},{}", ref_mut_1, ref_mut_2);
}
```

Unsafe Rust

One can use pointers to avoid various safety protocols:

8. ***Create unsafe methods / functions***

Rust

```rust
fn f() {}
unsafe fn g() {}

fn main() {
    let ptr: fn() = f;
    let ptr: fn() = g;
}
```

Compile Error

```
error[E0308]: mismatched types
 --> src\main.rs:6:21
  |
6 |     let ptr: fn() = g;
  |              ----   ^ expected normal fn, found unsafe fn
  |              |
  |              expected due to this
```

Rust (unsafe function)

```rust
fn f() {}
unsafe fn g() {}
fn main() {
    let ptr: unsafe fn() = g;
    let ptr: unsafe fn() = f;
    println!("OK");
}
```

Output
OK

It is also important to notice that an unsafe function can not be converted to a function reference. However, any function (safe or unsafe) can be converted to an unsafe function reference

Unsafe Rust

One can use pointers to avoid various safety protocols:

9. *Unsafe traits*

Rust

```
trait Foo {
    unsafe fn foo(&self);
}
struct A {}
impl Foo for A {
    unsafe fn foo(&self) {
        println!("foo");
    }
}
fn main() {
    let a = A{};
    unsafe { a.foo(); }
}
```

Output

foo

Notice that we need to call a.foo() within an **unsafe** block (this is because foo function was declared ***unsafe***).

But what if we know that the code we are going to add, ***adheres to Rust safety principals*** but for is in order to write it we need to use **unsafe block** ?

Unsafe Rust

One can use pointers to avoid various safety protocols:

9. ***Unsafe traits***

The solution in this case is to use an unsafe trait.

Rust

```
unsafe trait Foo
{
    fn foo(&self);
}
struct A {}
unsafe impl Foo for A
{
    fn foo(&self) {
        println!("foo");
    }
}
fn main() {
    let a = A{};
    a.foo();
}
```

Output
foo

Unsafe Rust

One can use pointers to avoid various safety protocols:

10. ***Modify/change global variables***

Rust

```rust
static mut global_x: i32 = 0;
fn main() {
    for _ in 0..3 {
        unsafe {
            global_x = global_x + 1;
            println!("{}",global_x);
        }
    }
}
```

Output

```
1
2
3
```

Global variable can only be immutable (this avoid various problems that could appear if multiple threads access and modify the same global variable). However, using unsafe, this behavior can be avoided like in this snippet.

Unsafe Rust

One can use pointers to avoid various safety protocols:

11. *Access APIs / ABIs*

Rust

```
extern "system" {
    fn GetTickCount() -> u32;
}
fn get_tick_count() -> u32 {
    unsafe { GetTickCount() }
}
fn main() {
    println!("Current tick: {}",get_tick_count());
}
```

Output

Current tick: 448103187

Rust can not guarantee the safeness of an API (including the ones of the operating system). For example, in this example, we try to access the method GetTickCount (available in Windows). We can get a reference to that method (via **extern** keyword), but if we want to use it , we can only use it using ***unsafe*** keyword.

Unsafe Rust

One can use pointers to avoid various safety protocols:

12. *Access the fields of a union*

Rust

```
union FloatToInt {
    int: u32,
    float: f32
}
fn main() {
    let mut x = FloatToInt { float: 8.625 };
    unsafe { println!("{} <-> {:X}",x.float, x.int); }
    x.int = 0x3FA00000;
    unsafe { println!("{} <-> {:X}",x.float, x.int); }
}
```

Output

```
8.625 <-> 410A0000
1.25 <-> 3FA00000
```

Rust support unions (just like C/C++). The only difference being that accessing union fields might result in unsafe operations (for example if one field is a heap allocated object and another one is a value). As such, accessing a value (e.g. for reading) must be done in an unsafe block

Unsafe Rust

Finally, a word of advice:

- Using unsafe is **NOT** recommended (if you want to code in Rust, you should first try to solve a problem using the safe functions)
- When using unsafe (and especially when using pointers) make sure that you consider all scenarios where the memory where a pointer points to might be moved, changed or deallocated (the memory management in this case become the programmer's job)

NonNull pointer

NonNull pointer

One potential security issue with raw pointers is that they can have a special value (called NULL) that implies an invalid memory address that, if read or write will produce a crash.

As such, Rust has a special wrapper around a raw pointer (called NonNull<T>) that guarantees that the inner pointer will never be NULL (even if the pointer is dereferenced). This features allows some optimizations for compounds such as Option<NonNull<T>> where the impossible value (NULL) is used as a discriminant (NULL means **None** in this case and any other value is associated with **Some**).

Rust (non_null.rs)

```
pub struct NonNull<T: ?Sized> {
    pointer: *const T,
}
```

NonNull pointer

Let’s see an example:

Rust

```rust
use std::ptr::NonNull;

fn main() {
    println!("{}",std::mem::size_of::<NonNull<i32>>());
    println!("{}",std::mem::size_of::<Option<NonNull<i32>>>());
    println!("{}",std::mem::size_of::<*const i32>());
    println!("{}",std::mem::size_of::<Option<*const i32>>());
}
```

Output

```
8
8
8
16
```

In this case, we can see that a NonNull wrapper and a raw pointer have the same size in memory (8 bytes for a 64 architecture), but an Option<NonNull<…>> is smaller (only 8 bytes) than an Option<*const …>. This is because in case of NonNull, the ***Option*** enum can use the NULL value as a discriminant to reflect the None option. In case of raw pointers, it has to add an additional member for the discriminant → hence the size of 16 bytes.

NonNull pointer

NonNull implements the following traits:

- Copy
- Clone
- Debug
- Eq
- PartialEq
- Ord
- PartialOrd
- Hash
- !Synk (meaning you can not transfer it between multiple threads – multithread safety)
- From (with different parameters).

NonNull pointer

Keep in mind that NonNull does not allocate memory on creation (you need a valid pointer to create a NonNull wrapper). To create a NonNull use one of the following:

Method	Usage
`fn new(ptr: *mut T)->Option<NonNull>`	Check ptr, if null returns None, otherwise returns Some
`unsafe fn new_unchecked(ptr: *mut T)->NonNull`	Sets the inner value to ptr, without checking it. Use this only when you are sure that ptr is not null. Because of this, this function is unsafe.
`fn dangling()->NonNull`	Creates an initialized (not NULL) pointer an invalid memory address. Don't try to read/write its value as it will result in a UB.

or one of the following From<T> implementations:

- From<&T>
- From<&mut T>

NonNull pointer

Notice that NonNull::dangling() method creates a NonNull (non-initialized pointer). ***One question here is why this method is NOT unsafe ?*** The reason for this is that you can not access the inner raw pointer without an unsafe block, and as such it is no problem when using NonNull::dangling() method. It is important to mention that NonNull::dangling() method will never create an inner pointer equal to Null.

This method is useful for:

- Lazy initialization cases (such as for a Vector)
- Structures where you need to initialize a pointer later , but after initialization you know that that pointer will never be Null

NonNull pointer

NonNull wrapper has the following (stable) methods (there are also several unstable method that will not be discuss here).

Method	Usage
`fn as_ptr(self) -> *mut T`	Returns the inner pointer, while consuming the NonNull object
`unsafe fn as_ref(&self) -> &T`	Returns an immutable reference to the object where the inner pointer points to. This method is unsafe (meaning you can only call it from within an unsafe block)
`unsafe fn as_mut(&mut self) -> &mut T`	Returns an mutable reference to the object where the inner pointer points to. This method is unsafe (meaning you can only call it from within an unsafe block)
`fn cast<U>(self) -> NonNull<U>`	Converts current NonNull pointer from type "T" to type "U"
`unsafe fn add(self, delta: usize) -> NonNull`	Performs a pointer addition over the inner pointer and returns a new NonNull wrapper, consuming the original one.

NonNull pointer

In the next example we create a NonNull structure over i32 and change its value via as_mut() method:

Rust

```rust
use std::ptr::NonNull;

fn main() {
    let mut y = 10;
    let x = NonNull::new(&mut y as *mut i32);
    if let Some(mut p_x) = x {
        unsafe { *p_x.as_mut() = 20 };
    }
    println!("{}",y);
}
```

Output

```
20
```

Rust

```rust
use std::ptr::NonNull;

fn main() {
    let mut y = 10;
    let mut x = NonNull::from(&mut y);
    unsafe { *x.as_mut() = 20 };
    println!("{}",y);
}
```

Output

```
20
```

Notice that in both cases, we still need to use an **unsafe block** to access the value where the inner pointer points to.

NonNull pointer

NonNull::dangling() to create a valid NonNull wrapper. Its purpose is to create an object that will be later initialized.

Rust

```rust
use std::ptr::NonNull;

fn main() {
    let mut x = NonNull::<i32>::dangling();
    unsafe { *x.as_mut() = 20 };
}
```

Error (runtime)

```
error: process didn't exit successfully:
`target\debug\rust_tester.exe` (exit code:
0xc0000005, STATUS_ACCESS_VIOLATION)
```

In this example, "x" will have an inner pointer (non-null, correctly align) that points to a possible invalid memory address. The result of accessing or modifying the data from the inner pointer is an undefined behavior (in most cases, it will translate into a crash during the runtime execution of the code).

NonNull pointer

NonNull::dangling() can however be used to create a structure where you only need to initialize a NonNull member later. These scenarios are useful when you know that the NonNull pointer will be valid from the moment of its initialization and until the end of its lifetime. It's also up to the programmer to make sure that between initialization of the structure and the actual initialization of the NonNull wrapper, the inner pointer will not be accessed.

Rust

```rust
use std::ptr::NonNull;
struct MyData { ptr: NonNull<i32> }
fn main() {
    let mut y = 10;
    let mut m = MyData { ptr: NonNull::dangling() };
    // do some operations that don't affect "m.ptr" in any way
    m.ptr = NonNull::from(&mut y); // lazy initialization of m.ptr
    unsafe { *m.ptr.as_mut() = 20; };
    println!("{}",y);
}
```

Output

```
20
```

NonNull pointer

However, if a data member of a structure can be either NULL or a valid pointer, it is best to use it with an Option.

Rust

```rust
use std::ptr::NonNull;
struct MyData {
    ptr: Option<NonNull<i32>>,
}
fn main() {
    let mut y = 10;
    let mut m = MyData { ptr: None };
    // now we initialize the MyData.ptr
    m.ptr = Some(NonNull::from(&mut y));
    if let Some(p_x) = m.ptr.as_mut() {
        unsafe {
            *p_x.as_mut() = 20;
        };
    }
    // later on, we can nullify the MyData.ptr
    m.ptr = None;
    println!("{}", y);
}
```

Output

```
20
```

NonNull pointer

Let's analyze the following example (on little-endian architecture):

Rust

```
use std::ptr::NonNull;

fn main() {
    let mut y = 1024u32;
    let p_y = NonNull::from(&mut y);
    let mut p_byte_y: NonNull<u8> = p_y.cast();
    unsafe {
        *p_byte_y.as_mut() = 1;
    }
    println!("{}", y);
}
```

Output

```
1025
```

"y" is initialized with 1024 ➔ meaning the layout of "y" in memory for LE architecture is:

0	4	0	0

=> $y = 0 + \mathbf{4 \times 2^8} + 0 \times 2^{16} + 0 \times 2^{24} = 1024$

"p_byte_y" is a u8 pointer that points towards the first byte of "y". Setting the value 1 at that byte means a change the memory layout of "y" for LE as follows:

1	4	0	0

that implies that $y = \mathbf{1} + \mathbf{4 \times 2^8} + 0 \times 2^{16} + 0 \times 2^{24} = 1025$

Interior Mutability

Interior Mutability

Sometimes, you need to create an object that even if from the outside does not have to change, internally you have to change a state/data members. In Rust this ability is called **interior mutability** .

Interior mutability is often used for cases where normal ownership and borrowing rules can not be applied due to the nature of the algorithm that is being used, such as:

- Graphs
- Double linked list
- Trees (if a child needs to keep a handle towards its parent)

Interior Mutability

Let's consider the following scenario:

- We want to create a pseudo-random object that retrieves random values between 0 and a maximum number
- One simple algorithm will be to start with a fix seed
- Then whenever a new number is requested, we use the following algorithm:

```
seed ← seed * 22695477 + 1
return seed % maximum_value
```

- Notice that on any step we need to change the seed , so that the next time we will generate another number.

This algorithm can be written in various ways using either unsafe or cell. A cell in Rust is a type of wrapper around a given type that provides interior mutability.

Interior Mutability

Before we start discussing about implementation scenarios, let's set up some evaluation criteria that we can use:

1. **Multi thread protection** → means that there is no scenario where accessing a resource from multiple threads can lead to an undefined behavior
2. **Use of unsafe block** → means that the program must use "unsafe" when using a specific approach
3. **Works with references** → means that a described scenario can be utilized with references or not
4. **Global variable protections** → means that there is a protection against changing a global variable from multiple threads

Interior Mutability

1. ***Write this just like a regular Rust program***

Rust

```rust
mod random {
    pub struct Random {
        seed: u32,
    }
    impl Random {
        pub const fn new() -> Self { Self { seed: 1 } }
        pub fn get_value(&mut self, max_value: u32) -> u32 {
            self.seed = self.seed.overflowing_mul(22695477u32).0 + 1u32;
            return self.seed % max_value;
        }
    }
}
fn main() {
    let mut r = random::Random::new();
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

Output

```
8
5
0
```

Interior Mutability

1. ***Write this just like a regular Rust program***

Rust

```
mod random {
    pub struct Random {
        seed: u32,
    }
    impl Random {
        pub const fn new() -> Self { Self { seed: 1 } }
        pub fn get_value(&mut self, max_value: u32) -> u32 {
            self.seed = self.seed.overflowing_mul(22695477u32).0 + 1u32;
            return self.seed % max_value;
        }
    }
}
fn main() {
    let mut r = random::Random::new();
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

Output

```
8
5
0
```

Notice that we need to create the variable r as ***mutable*** if we want to call the method get_value that receives a mutable reference to self.

Interior Mutability

1. ***Write this just like a regular Rust program***

Rust

Output

```
mod random {
    pub struct Random {
        seed: u32,
    }
    impl Random {
        pub const fn new() -> Self { Self { seed: 1 } }
        pub fn get_value(&mut self, max_value: u32) -> u32 {
            self.seed = self.seed.overflowing_mul(22695477u32).0 + 1u32;
            return self.seed % max_value;
        }
    }
}
fn main() {
    let mut r = random::Random::new();
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

We have also made the data member seed ***private*** (it can only be access from within the module random) and it is used by methods new(…) and get_value(…) that are public.

Interior Mutability

1. ***Write this just like a regular Rust program***

Rust

```
mod random {
    pub struct Random {
        seed: u32,
    }
    impl Random {
        pub const fn new() -> Self { Self { seed: 1 } }
        pub fn get_value(&mut self, max_value: u32) -> u32 {
            self.seed = self.seed.overflowing_mul(22695477u32).0 + 1u32;
            return self.seed % max_value;
        }
    }
}
fn main() {
    let mut r = random::Random::new();
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

Output

We have also made the data member seed ***private*** (it can only be access from within the module random) and it is used by methods new(…) and get_value(…) that are public.

Furthermore, if we look at variable r from outside of module random, we will see no public field (just some methods). So, in fact, there is no visible need to make it ***mutable***

Interior Mutability

So … thinking about the previous algorithm, how can we make variable r immutable, and still generate (use the same algorithm) to generate a pseudo-random number (just like in the next snippet).

Rust (desired behavior)

```
mod random { … }
fn main() {
    let r = random::Random::new();
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

Output
8
5
9

So … how can we rewrite module random so that we get this behavior ?
(the solution to this requirement is called ***interior mutability***)

Interior Mutability

2. *Use unsafe to change the mutability of data member seed*

Rust

```rust
mod random {
    pub struct Random {
        seed: u32,
    }
    impl Random {
        pub const fn new() -> Self {
            Self { seed: 1u32 }
        }
        pub fn get_value(&self, max_value: u32) -> u32 {
            let new_seed = self.seed.overflowing_mul(22695477u32).0  + 1u32 ;
            unsafe {
                let p_to_me = (&self.seed as *const u32) as *mut u32;
                *p_to_me = new_seed;
            }
            return new_seed % max_value ;
        }
    }
}
```

Output

```
8
5
0
```

Interior Mutability

2. Use unsafe to change the mutability of data member seed

Rust

```
mod ra
    pu

    }
    im

            Self { seed:         }
        }
        pub fn get_value(&self, max_value: u32) -> u32 {
            let new_seed = self.seed.overflowing_mul(22695477u32).0  + 1u32 ;
            unsafe {
                let p_to_me = (&self.seed as *const u32) as *mut u32;
                *p_to_me = new_seed;
            }
            return new_seed % max_value ;
        }
    }
}
```

Notice that method get_value receives an ***<u>immutable reference to self.</u>*** This means that we do not have to create a mutable variable in order to use this method.

Variable r is immutable

```
fn main() {
    let r = random::Random::new();
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

Interior Mutability

2. ***Use unsafe to change the mutability of data member seed***

Rust

```
mod random {
    pub struct Random {
        seed: u32,
    }
    impl Random {
        pub const fn n
            Self { see
        }
        pub fn get_value(&self, max_value: u32) -> u32 {
            let new_seed = self.seed.overflowing_mul(22695477u32).0  + 1u32 ;
            unsafe {
                let p_to_me = (&self.seed as *const u32) as *mut u32;
                *p_to_me = new_seed;
            }
            return new_seed % max_value ;
        }
    }
}
```

We use an unsafe block to access the pointer to data member seed and change it.

Output

```
8
5
0
```

Interior Mutability

So .. is this a good approach (using unsafe), in terms of Rust safety principles ?

Let's evaluate a couple of cases:

A. **Dangling / invalid pointer**

Assuming we create multiple references to the same variable, is there a possibility of accessing an invalid memory address through a pointer ? The answer to this question is **NO**.

Rust

```
pub fn get_value(&self, max_value: u32) -> u32 {
    …
    unsafe {
        let p_to_me = (&self.seed as *const u32) as *mut u32;
        …
     }
     …
}
```

You will notice that p_to_me pointer only exists within the context of method get_value(…). Method get_value **has a valid self reference**, and as such, the p_to_me pointer will always be **valid**. As a result, there is no scenario where this pointer might point to an invalid memory address

Interior Mutability

So .. is this a good approach (using unsafe), in terms of Rust safety principles ?

Let's evaluate a couple of cases:

B. **Single thread soundness**
Assuming we run our code in a single thread scenario, are the results consistent. The answer is **YES** → the results are not only consistent, but deterministic for the current pseudo-random code generator.

C. **Multi thread soundness**
In this case there is a possibility that two threads might call the get_value method at the same time. If this is the case, the result for each thread will be undetermined. However , we should still mention that no runtime crash will happen as even if we access the same memory from multiple threads, its still the same memory so we will not end up with an invalid pointer.

Interior Mutability

So .. is this a good approach (using unsafe), in terms of Rust safety principles ?

Let's evaluate a couple of cases:

D. <u>Code optimizations</u>

An immutable variable might be optimized by Rust. One special case is if that variable is declared as a global variable. In such cases, Rust might decide to allocate space for that variable into a non-writeable page (for example in a section like .rdata in case of PE executable for Windows). This could be problematic when trying to write data through a pointer.

Rust

```rust
unsafe {
    let p_to_me = (&self.seed a
    *p_to_me = new_seed;
}
```

If p_to_me points to a memory located in a read-only page (such as .rdata section) this operation will crash the application with a **memory access violation error**.

Interior Mutability

So .. is this a good approach (using unsafe), in terms of Rust safety principles ?

Let's evaluate a couple of cases:

D. **Code optimizations**

Let's consider the following scenario:

Rust

```rust
mod random { … }

static r: random::Random = random::Random::new();

fn main() {
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

Error (runtime)

```
error: process didn't exit successfully:
`target\debug\rust_tester.exe` (exit code: 0xc0000005,
STATUS_ACCESS_VIOLATION)
```

This is not a **deterministic** result (future version of Rust might behave differently). It was obtained by compiling the code with debug with the following rust compiler: rustc 1.71.0 (8ede3aae2 2023-07-12)

Interior Mutability

So .. is this a good approach (using unsafe), in terms of Rust safety principles ?

Overview:

	Use of unsafe block	Multi thread protection	Global Variable protection	Working with references
Raw pointers	Yes	No	No	Yes (unsafe)

As a result, the approach is correct (safe and sound), but only for a single thread scenario.

So … the next question is → can we enforce a code like the similar one to run only on a single thread case ?

Interior Mutability

2. ***Use UnsafeCell***

Rust has a special structure (called UnsafeCell) defined in the following way:

Rust (from cell.rs)

```rust
#[lang = "unsafe_cell"]
#[stable(feature = "rust1", since = "1.0.0")]
#[repr(transparent)]
pub struct UnsafeCell<T: ?Sized> {
    value: T,
}
```

with a method .get(...) defined in the following way:

Rust (from cell.rs)

```rust
pub const fn get(&self) -> *mut T {
    // We can just cast the pointer from `UnsafeCell<T>` to `T` because of
    // #[repr(transparent)]. This exploits std's special status, there is
    // no guarantee for user code that this will work in future versions of the compiler!
    self as *const UnsafeCell<T> as *const T as *mut T
}
```

Interior Mutability

2. ***Use UnsafeCell***

Rust has a special structure (called UnsafeCell) defined in the following way:

Rust (from cell.rs)

```
#[lang = "unsafe_cell"]
#[stable(feature = "rust1", si
#[repr(transparent)]
pub struct UnsafeCell<T: ?Sized> {
    value: T,
}
```

This actually tells the compiler to treat this in a special way. For some std types, there is a need for the compiler to behave in a different way (e.g. add some extra checks). It’s worth mention that this attributes are considered internal and should not be used outside their purpose (e.g. with another structure).

with a method .get(…) defined in the following way:

Rust (from cell.rs)

```
pub const fn get(&self) -> *mut T {
    // We can just cast the pointer from `UnsafeCell<T>` to `T` because of
    // #[repr(transparent)]. This exploits std's special status, there is
    // no guarantee for user code that this will work in future versions of the compiler!
    self as *const UnsafeCell<T> as *const T as *mut T
}
```

Interior Mutability

2. *Use UnsafeCell*

Rust has a special structure (called UnsafeCell) defined in the following way:

Rust (from cell.rs)

```
#[lang = "unsafe_cell"]
#[stable(feature = "rust1", since = "1.0.0")]
#[repr(transparent)]
pub struct UnsafeCell<T:
    value: T,
}
```

with a method .get(..

Notice that get() method receives an immutable reference to self *(&self)* and it uses the same cast mechanism to get mutable pointer to the object of type T it contains.

Rust (from cell.rs)

```
pub const fn get(&self) -> *mut T {
    // We can just cast the pointer from `UnsafeCell<T>` to `T` because of
    // #[repr(transparent)]. This exploits std's special status, there is
    // no guarantee for user code that this will work in future versions of the compiler!
    self as *const UnsafeCell<T> as *const T as *mut T
}
```

Interior Mutability

2. *Use UnsafeCell*

Rust

```rust
mod random {
    pub struct Random { seed: UnsafeCell<u32> }
    impl Random {
        pub const fn new() -> Self { Self { seed: UnsafeCell::new(1) } }
        pub fn get_value(&self, max_value: u32) -> u32 {
            let seed = self.seed.get();
            unsafe {
                *seed = (*seed).overflowing_mul(22695477u32).0 + 1u32;
                return *seed % max_value;
            }
        }
    }
}
fn main() {
    let mut r = random::Random::new();
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

Output

```
8
5
0
```

Interior Mutability

2. *Use UnsafeCell*

Rust

```rust
mod random {
    pub struct Random { seed: UnsafeCell<u32> }
    impl Random {
        pub const fn new() -> Self { Self { seed: UnsafeCell::new(1) } }
        pub fn get_value(&self, max_value: u32) -> u32 {
            let seed = self.seed.get();
            unsafe {
                *seed = (*seed).overflowing_mul(22695477u32).0 + 1u32;
                return *seed % max_value;
            }
        }
    }
}
fn main() {
    let mut r = random::Random::new();
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

Output
8
5
0

Notice that we still have to use an **unsafe** block (the only difference is that we don't need to case the pointer ourselves). Furthermore, the seed data member need to be defined as an ***UnsafeCell<u32>***

Interior Mutability

2. *Use UnsafeCell*

Now let's see if we create a global variable using this code base on **UnsafeCell** and we test it to see the effects of a possible optimization what happens).

Rust

```rust
mod random { … }

static r: random::Random = random::Random::new();

fn main() {
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

Error (runtime)

```
error[E0277]: `UnsafeCell<u32>` cannot be shared between threads
safely
  --> src\main.rs:98:11
   |
98 | static r: random::Random = random::Random::new();
   |           ^^^^^^^^^^^^^^ `UnsafeCell<u32>` cannot be shared
   |                          between threads safely
   |
   = help: within `Random`, the trait `Sync` is not implemented
for `UnsafeCell<u32>`
```

Notice that the behavior is different than us using a raw pointer (in the sense that we can not create a static variable using UnsafeCell<u32>). This error and other checks are likely to be employed by the compiler due to the special trait that was added to UnsafeCell : **!Sync**

Interior Mutability

Let’s see an overview of using UnsafeCell for our previous problem:

Overview:

	Use of unsafe block	Multi thread protection	Global Variable protection	Working with references
UnsafeCell	Yes	Yes (Will not compile)	Yes (Will not compile)	Yes (unsafe)

As a result, this approach is better than the previous one. Furthermore, since ***UnsafeCell*** is part of the standard in Rust, we should expect that even if some things change in terms of raw pointer casting in Rust, ***UnsafeCell*** will maintain its functionality.

However, we still have to use an unsafe block → can we do something about this ?

Interior Mutability

3. ***Use Cell***

On top of UnsafeCell, Rust has another structure called Cell defined as follows:

Rust (from cell.rs)

```
pub struct Cell<T: ?Sized> {
    value: UnsafeCell<T>,
}
```

with two methods .get(...) and .set(...) defined in the following way:

Rust (from cell.rs)

```
pub fn get(&self) -> T {
    // SAFETY: This can cause data races if called from a separate thread,
    // but `Cell` is `!Sync` so this won't happen.
    unsafe { *self.value.get() }
}

pub fn set(&self, val: T) {
    let old = self.replace(val);
    drop(old);
}
```

```
pub fn replace(&self, val: T) -> T {
  // SAFETY: This can cause data races if called from a separate
  // thread, but `Cell` is `!Sync` so this won't happen.
  mem::replace(unsafe { &mut *self.value.get() }, val)
}
```

Interior Mutability

3. ***Use Cell***

The Cell<T> struct in Rust implements the following traits:

- Copy trait
- Clone trait
- PartialEq
- PartialOrd
- Eq
- Ord
- !Synk ➔ meaning that a Cell<T> can not be used in a multi-thread scenario

Since Cell<T> implements Copy and Clone, an object of this type is used with data types that implement Copy / Clone. Furthermore, method get(…) returns an object and not a reference (transferring the ownership). Similar, set(…) receives an object (thus transferring the ownership) and not a reference.

Interior Mutability

3. *Use Cell*

Let's see how Cell get(…) method works.

Rust (example 1)

```rust
fn main() {
    let s: Cell<u32> = Cell::new(10);
    let obj = s.get();
    println!("{}",obj);
}
```

Output

```
10
```

Rust (example 2)

```rust
fn main() {
    let s: Cell<String> = Cell::new(String::from("abc"));
    let obj = s.get();
    println!("{}",obj);
}
```

Error

```
Error[E0599]: the method `get` exists for struct `Cell<String>`,
but its trait bounds were not satisfied
   --> src\main.rs:112:17
    |
112 |     let obj = s.get();
    |                 ^^^
    |
365 | pub struct String {
    | ----------------- doesn't satisfy `String: Copy`
    |
    = note: the following trait bounds were not satisfied:
            `String: Copy`
```

In the first example, we use Cell with an u32, and as such, method .get(…) can copy the value.
In the second case, we use Cell with String, and method .get(…) is not available as String does not have the Copy trait.

Interior Mutability

3. *Use Cell*

Rust

```
mod random {
    pub struct Random { seed: Cell<u32> }
    impl Random {
        pub const fn new() -> Self { Self {seed: Cell::new(1) } }
        pub fn get_value(&self, max_value: u32) -> u32 {
            let mut seed = self.seed.get();
            seed = seed.overflowing_mul(22695477u32).0 + 1u32;
            self.seed.set(seed);
            return seed % max_value;
        }
    }
}
fn main() {
    let mut r = random::Random::new();
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

Output

```
8
5
0
```

Interior Mutability

3. *Use Cell*

Rust

```rust
mod random {
    pub struct Random { seed: Cell<u32> }
    impl Random {
        pub const fn new() -> Self { Self {seed: Cell::new(1) } }
        pub fn get_value(&self, max_value: u32) -> u32 {
            let mut seed = self.seed.get();
            seed = seed.overflowing_mul(22695477u32).0 + 1u32;
            self.seed.set(seed);
            return seed % max_value;
        }
    }
}
fn main()
    let m
    for _
        p
    }
}
```

Output

```
8
5
0
```

Notice that we do not need any unsafe block. Instead, we can transfer ownership from the Cell and then back into so that we can update the value.

Interior Mutability

Let’s see an overview of using Cell for our previous problem:

Overview:

	Use of unsafe block	Multi thread protection	Global Variable protection	Working with references
Cell	No	Yes (Will not compile)	Yes (Will not compile)	No (Will not compile)

As a result, this approach is better than the previous one. We don’t need to use unsafe block, but this code works only types that have Copy/Clone trait.

So .. What if we want to do the same thing, but for types that don’t support Copy trait. To do this, we would need to work with references !

Interior Mutability

4. *Use RefCell*

On top of Cell and UnsafeCell, Rust has a structure called RefCell:

Rust (from cell.rs)

```
pub struct RefCell<T: ?Sized> {
    borrow: Cell<BorrowFlag>,
    value: UnsafeCell<T>,
}
```

The idea on top of RefCell is that it enforces the ownership & borrowing rules of Rust at runtime (more exactly, it panics if one of those rules are being broken.

This is done via using a ***BorrowFlag*** (an **isize**) where Rust keeps count on how many immutable reference are and if there is one mutable reference. Based on these information it can enforce the ownership & borrowing rules at runtime, as follows:

- Value 0 ➔ not borrow at all (immutable / mutable)
- Values between 1 and isize::MAX ➔ count of immutable references
- Negative value ➔ an mutable reference exists

Interior Mutability

4. ***Use RefCell***

Method	Usage
`fn borrow_mut(&self) -> RefMut<T>`	Returns a mutable reference to an object
`fn borrow(&self) -> Ref<T>`	Returns an immutable reference to an object

Each one of this methods have a special logic in place:

- **borrow_mut ➔** if at least one mutable or immutable reference was made then a panic is thrown. Otherwise, an internal flag that marks that an immutable reference was made will be set and then a reference is returned
- **borrow ➔** if at least one mutable reference was made then a panic is thrown. Otherwise, an internal flag that

This is why the return value of those methods are objects of type RefMut and Ref, that when dropped will update a flag in a RefCell that stores the active immutable/mutable count.

Interior Mutability

4. *Use RefCell*

The logic of RefCell is that it upholds the safety (ownership & borrowing) rules that Rust enforces, only during execution (runtime) and not at compile time.

Let's analyze some examples:

Rust (example 1)

```
fn main() {
    let x = RefCell::new(1);
    let y = x.borrow_mut();
    println!("{}",y);
}
```

Output

```
1
```

Rust (example 2)

```
fn main() {
    let x = RefCell::new(1);
    let y = x.borrow_mut();
    let z = x.borrow();
}
```

Error (Runtime panic)

```
thread 'main' panicked at 'already
mutably borrowed: BorrowError'
```

Rust (example 3)

```
fn main() {
    let x = RefCell::new(1);
    let y = x.borrow();
    let z = x.borrow();
    println!("{},{}",y,z);
}
```

Output

```
1,1
```

Rust (example 4)

```
fn main() {
    let x = RefCell::new(1);
    let y = x.borrow_mut();
    let z = x.borrow_mut();
    println!("{},{}",y,z);
}
```

Error (Runtime panic)

```
thread 'main' panicked at 'already
mutably borrowed: BorrowError'
```

Interior Mutability

4. *Use RefCell*

Rust

```
mod random {
    pub struct Random { seed: RefCell<u32> }
    impl Random {
        pub const fn new() -> Self { Self {seed: RefCell::new(1) } }
        pub fn get_value(&self, max_value: u32) -> u32 {
            let mut seed = self.seed.borrow_mut();
            *seed = (*seed).overflowing_mul(22695477u32).0 + 1u32;
            return (*seed) % max_value;
        }
    }
}
fn main() {
    let mut r = random::Random::new();
    for _ in 0..3 {
        println!("{}", r.get_value(10));
    }
}
```

Output

```
8
5
0
```

Interior Mutability

Overall status:

Method	Raw pointers	UnsafeCell	Cell	RefCell
Use of unsafe block	**Yes**	**Yes**	**No**	**No**
Multi thread protection	**No**	**Won't compile**	**Won't compile**	**Won't compile**
Global variable protection	**Unsafe**	**Won't compile**	**Won't compile**	**Won't compile**
Working with ref	**Unsafe**	**Unsafe**	**Won't compile**	**Safe**

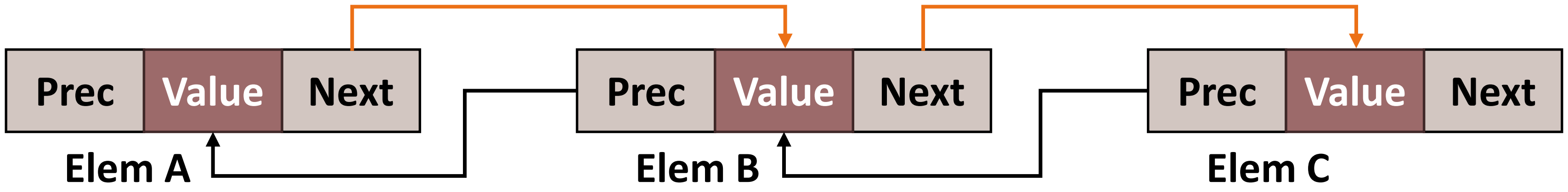
As a general observation, use:

- Cell<T> if you work with basic types / data with Copy traits
- RefCell<T> for Mutable data

Reference Count

Reference Count

The way Rust safety measures are designed, certain type of algorithms are hard (complicated) to write. Let's take for example a double linked list:



This type of construct is relatively simple to create in languages like C/C++ (as we just need some pointers to the next and previous elements). However, in Rust we need to think about this problem in a different way.

For example, both elements A and C have either an ownership or a reference to element B (through Next and Prec links). But the ownership rules state that there **could be only one owner for a memory zone** (meaning that we will need to think about this problem in a different way).

Reference Count

Let's explore some ideas for solving this problem in Rust:

C/C++

```
struct Node {
    Node * next;
    Node * prec;
    int value;
};
Node* add(Node* left, int value)
{
    Node * n = new Node();
    n->value = value;
    if (left->next) {
        left->next->prec = n;
    }
    n->next = left->next;
    n->prec = left;
    left->next = n;
    return n;
}
```

1. Try to write Node struct using Option<Node>

Rust

```
struct Node {
    next: Option<Node>,
    prec: Option<Node>,
    value: i32
}
```

Not really an option, as Node structure will have an infinite size. The recomandation is to use a Box

Error

```
error[E0072]: recursive type `Node` has infinite size
 --> src\main.rs:1:1
  |
1 | struct Node {
  | ^^^^^^^^^^^
2 |     next: Option<Node>,
  |                  ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle
  |
2 |     next: Option<Box<Node>>,
  |                  ++++    +
```

Reference Count

Let’s explore some ideas for solving this problem in Rust:

C/C++

```c
struct Node {
    Node * next;
    Node * prec;
    int value;
};
Node* add(Node* left, int value)
{
    Node * n = new Node();
    n->value = value;
    if (left->next) {
        left->next->prec = n;
    }
    n->next = left->next;
    n->prec = left;
    left->next = n;
    return n;
}
```

2. Try to write Node struct using Option<Box<Node>>

Rust

```rust
struct Node {
    next: Option<Box<Node>>,
    prec: Option<Box<Node>>,
    value: i32
}

fn add(left: &mut Box<Node>, value: i32) -> &Node {
    let mut n = Box::new(Node{next: None, prec: None, value: value});
    if let Some(left_next) = left.next.as_mut() {
        left_next.prec = Some(n);
    }
    n.next = left.next;
    n.prec = Some(*left);
    return &n;
}
```

Reference Count

Let's explore some ideas for solving this problem in Rust:

C/C++

```
struct Node {
    Node * next;
    Node * prec;
    int value;
};
Node* add(Node* left, i
{
    Node * n = new Node
    n->value = value;

    return n;
}
```

Error

```
error[E0382]: assign to part of moved value: `*n`
  --> src\main.rs:12:5
   |
8  |     let mut n = Box::new(Node{next: None, prec: None, value: value});
   |         ----- move occurs because `n` has type `Box<Node>`, which does not implement the `Copy` trait
9  |     if let Some(left_next) = left.next.as_mut() {
10 |         left_next.prec = Some(n);
   |                               - value moved here
11 |     }
12 |     n.next = left.next;
   |     ^^^^^^ value partially assigned here after move
```

```
fn add(left: &mut Box<Node>, value: i32) -> &Node {
    let mut n = Box::new(Node{next: None, prec: None, value: value});
    if let Some(left_next) = left.next.as_mut() {
        left_next.prec = Some(n);
    }
    n.next = left.next;
    n.prec = Some(*left);
    return &n;
}
```

This can not work, as from the moment we link left_next to our current object ("n"), we transfer the ownership and as such we can not use our object ("n") anymore.

Reference Count

Let's explore some ideas for solving this problem in Rust:

C/C++

```
struct Node {
    Node * next;
    Node * prec;
    int value;
};
Node* add(Node* left, int value)
{
    Node * n = new Node();
    n->value = value;
    if (left->next) {
        left->next->prec = n;
    }
    n->next = left->next;
    n->prec = left;
    left->next = n;
    return n;
}
```

3. Try to write Node struct using references

Rust

```
struct Node<'a> {
    next: Option<&'a mut Node<'a>>,
    prec: Option<&'a mut Node<'a>>,
    value: i32
}

fn add<'a>(left: &'a mut Node<'a>, value: i32) -> &'a mut Node<'a> {
    let mut n = Node{next: None, prec: None, value: value};
    if let Some(left_next) = left.next.as_mut() {
        left_next.prec = Some(&mut n);
    }
    n.next = left.next;
    n.prec = Some(left);
    return &mut n;
}
```

Reference Count

Let's explore some ideas for solving

C/C++

```
struct Node {
    Node * next;
    Node * prec;
    int value;
};
Node* add(Node* left, int value)
{
    Node * n = new Node();
    n->value = value;
    ...
    left->next = n;
    return n;
}
```

3. Try

Rus

```
str
...
}
fn
    let mut n = Node{next: None, prec: None, value: value};
    if let Some(left_next) = left.next.as_mut() {
        left_next.prec = Some(&mut n);
    }
    n.next = left.next;
    n.prec = Some(left);
    return &mut n;
}
```

Error

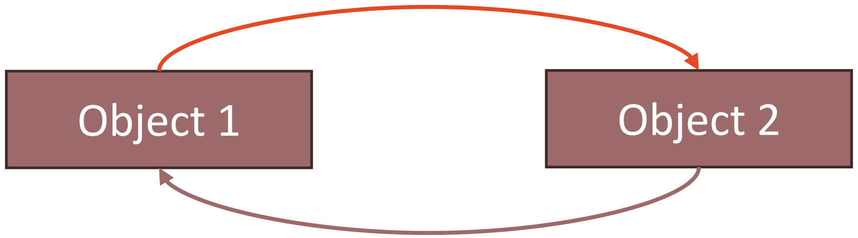
```
error[E0506]: cannot assign to `n.next` because it is borrowed
  --> src\main.rs:12:5
   |
7  | fn add<'a>(left: &'a mut Node<'a>, value: i32) -> &'a mut Node<'a> {
   |         -- lifetime `'a` defined here
...
10 |         left_next.prec = Some(&mut n);
   |         -----------------------------
   |         |                     |
   |         |                     `n.next` is borrowed here
   |         assignment requires that `n` is borrowed for `'a`
11 |     }
12 |     n.next = left.next;
   |     ^^^^^^^^^^^^^^^^^^ `n.next` is assigned to here but it was already borrowed
```

This is a similar case, when we transfer a mutable reference from n to left_next.prec, we can not access/use the same mutable reference again.

Reference Count

So … what are our options for this problem ? (Because any C-like similar solution ultimately will try to break the ownership rules and as such it will not compile).

More generically, any problem where two objects have reference one to another (just like in the following image) is hard to design in Rust due to ownerships rules.



As the two links can not be build at the same time (we must first build one of the objects and then the other, and as such for the first object, the link to the next one can not exist). The solution is to use an Option so that we can link the objects later

Reference Count

Let’s review a couple of solutions for Object1 and Object2.

1. Object1 has ownership over Object2 and Object2 has ownership over Object1

Rust

```
struct Object1 {
    link: Object2
}
struct Object2 {
    link: Object1
}
```

Error

```
error[E0072]: recursive types `Object1` and `Object2` have infinite size
 --> src\main.rs:1:1
  |
1 | struct Object1 {
  | ^^^^^^^^^^^^^^
2 |     link: Object2
  |           ------- recursive without indirection
3 | }
4 | struct Object2 {
  | ^^^^^^^^^^^^^^
5 |     link: Object1
  |           ------- recursive without indirection
```

This will never work as both Object1 and Object2 have infinite size (Rust can not compute their Size)

Reference Count

Let's review a couple of solutions for Object1 and Object2.

2. Object1 has ownership over Object2 (via Box) and vice versa

Rust

```rust
struct Object1 {
    link: Option<Box<Object2>>,
}
struct Object2 {
    link: Option<Box<Object1>>,
}
fn main() {
    let mut o1 = Box::new(Object1 { link: None });
    let mut o2 = Box::new(Object2 { link: None });
    o1.link = Some(o2);
    o2.link = Some(o1);
}
```

Reference Count

Let's review a couple of solutions for Object1 and Object2.

2. Object1 has ownership over Object2 (via Box) and vice versa

Rust

```
struct Object1 {
    link: Option<Box<Object2>>,
}
```

Once o2 is linked to o1 (o1.link = Some(o2)), variable o2 no longer has ownership and as such the next line is invalid.

```
    let mut o1 = Box::new(Object1 { link: None });
    let mut o2 = Box::new(Object2 { link: None });
    o1.link = Some(o2);
    o2.link = Some(o1);
}
```

Error

```
error[E0382]: assign to part of moved value: `*o2`
  --> src\main.rs:11:5
   |
9  |     let mut o2 = Box::new(Object2 { link: None });
   |         ------ move occurs because `o2` has type `Box<Object2>`,
   |                which does not implement the `Copy` trait
10 |     o1.link = Some(o2);
   |                    -- value moved here
11 |     o2.link = Some(o1);
   |     ^^^^^^^ value partially assigned here after move
```

Reference Count

Let’s review a couple of solutions for Object1 and Object2.

3. Object1 has a reference to Object2 and Object2 has a reference to Object1

Rust

```
struct Object1<'a> {
    link: Option<&'a Object2<'a>>,
}
struct Object2<'a> {
    link: Option<&'a Object1<'a>>,
}
fn main() {
    let mut o1 = Object1 { link: None };
    let mut o2 = Object2 { link: None };
    o1.link = Some(&o2);
    o2.link = Some(&o1);
}
```

Reference Count

Let's review a couple of solutions for Object1 and Object2.

3. Object1 has a reference to Object2 and Object2 has a reference to Object1

When the following line runs : `o1.link = Some(&o2);` an immutable reference to o2 is obtained. However, when the second line `o2.link = Some(&o1);` happens, in order to evaluate o2.link a mutable reference to **o2** is required. This is not possible as an immutable reference to **o2** already exists.

```rust
    let mut o1 = Object1 { link: None };
    let mut o2 = Object2 { link: None };
    o1.link = Some(&o2);
    o2.link = Some(&o1);
}
```

Error

```
error[E0506]: cannot assign to `o2.link` because it is borrowed
  --> src\main.rs:24:5
   |
23 |     o1.link = Some(&o2);
   |                    --- `o2.link` is borrowed here
24 |     o2.link = Some(&o1);
   |     ^^^^^^^^^^^^^^^^^^^
   |     |
   |     `o2.link` is assigned to here but it was already borrowed
   |     borrow later used here
```

Reference Count

Let’s review a couple of solutions for Object1 and Object2.

4. Object1 has a RefCell to Object2 and Object2 has a RefCell to Object1

Rust

```rust
use std::cell::RefCell;

struct Object1<'a> {
    link: Option<RefCell<&'a Object2<'a>>>
}
struct Object2<'a> {
    link: Option<RefCell<&'a Object1<'a>>>,
}
fn main() {
    let mut o1 = Object1 { link: None};
    let mut o2 = Object2 { link: None};
    o1.link = Some(RefCell::new(&o2));
    o2.link = Some(RefCell::new(&o1));
}
```

Reference Count

Let’s review a couple of solutions for Object1 and Object2.

4. Object1 has a RefCell to Object2 and Object2 has a RefCell to Object1

Rust

```
use std::cell::RefCell;

struct Object1<'a> {
    link: Option<RefCell<&'a Object2<'a>>>
```

Similar scenario as with the previous one. Since we use an immutable reference to an object (&o2) to link o1, we can not create a mutable reference (towards o2) while an immutable one already exists.

```
    let mut o1 = Object1 { link: None};
    let mut o2 = Object2 { link: None};
    o1.link = Some(RefCell::new(&o2));
    o2.link = Some(RefCell::new(&o1));
}
```

Error

```
error[E0506]: cannot assign to `o2.link` because it is borrowed
  --> src\main.rs:26:5
   |
25 |     o1.link = Some(RefCell::new(&o2));
   |                                 --- `o2.link` is borrowed here
26 |     o2.link = Some(RefCell::new(&o1));
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |     |
   |     `o2.link` is assigned to here but it was already borrowed
   |     borrow later used here
```

Reference Count

Let’s review a couple of solutions for Object1 and Object2.

5. Object1 has ownership over Object2 and Object2 has a reference to Object1

Rust

```rust
struct Object1<'a> {
    link: Object2<'a>
}
struct Object2<'a> {
    link: Option<&'a Object1<'a>>,
}
fn main() {
    let mut o1 = Object1 { link: Object2{ link: None} };
    o1.link.link = Some(&o1);
}
```

Reference Count

Let’s review a couple of solutions for Object1 and Object2.

5. Object1 has ownership over Object2 and Object2 has a reference to Object1

Rust

```
struct Object1<'a> {
```

Similar to the previous case. In order to change o1, we will need a mutable reference towards it. Since this exists, we can not obtain an immutable reference towards o1 at the same time (to create Some(&o1))

```
fn main() {
    let mut o1 = Object1 { link: Object2{ link: None} });
    o1.link.link = Some(&o1);
}
```

Error

```
error[E0506]: cannot assign to `o1.link.link` because it is borrowed
  --> src\main.rs:22:5
   |
22 |     o1.link.link = Some(&o1);
   |     ^^^^^^^^^^^^^^^^^^^^^---^
   |     |                    |
   |     |                    `o1.link.link` is borrowed here
   |     `o1.link.link` is assigned to here but it was already borrowed
   |     borrow later used here
```

Reference Count

Let's review a couple of solutions for Object1 and Object2.

6. Use unsafe: Object1 has ownership over Object2 and Object2 has a reference to Object1 (bent the ownership rules by making a copy of the reference to Object1)

Rust

```rust
struct Object1<'a> {
    link: Object2<'a>
}
struct Object2<'a> {
    link: Option<&'a Object1<'a>>,
}
fn main() {
    let mut o1 = Object1 { link: Object2{ link: None} };
    let ref_o1 = &o1;
    let copy_of_ref_to_o1 = unsafe { &*(ref_o1 as *const Object1) };
    o1.link.link = Some(copy_of_ref_to_o1);
    println!("OK");
}
```

Output
OK

Reference Count

Let's review a couple of solutions for Object1 and Object2.

7. Use raw pointers (just like in C/C++)

Rust

```
struct Object1 {
    link: *const Object2
}
struct Object2 {
    link: *const Object1
}
fn main() {
    let mut o1 = Object1 { link: std::ptr::null()};
    let mut o2 = Object2 { link: std::ptr::null()};
    o1.link = &o2 as *const Object2;
    o2.link = &o1 as *const Object1;
    println!("OK");
}
```

Output
OK

Reference Count

Let's review a couple of solutions for Object1 and Object2.

8. Use NonNull wrapper

Rust

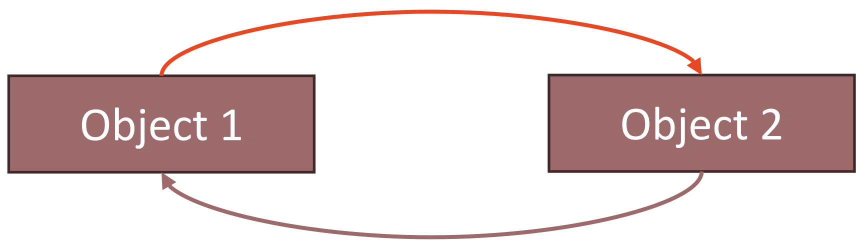
```rust
use std::ptr::NonNull;

struct Object1 {
    link: Option<NonNull<Object2>>
}
struct Object2 {
    link: Option<NonNull<Object1>>,
}
fn main() {
    let mut o1 = Object1 { link: None};
    let mut o2 = Object2 { link: None};
    o1.link = Some(NonNull::from(&o2));
    o2.link = Some(NonNull::from(&o1));
    println!("OK");
}
```

Output
OK

Reference Count

In all of the previous 3 solutions (6, 7 and 8) that worked (compiled) all we did was to transfer the responsibility of code safety to the programmer (meaning that the programmer has to be careful on stuff like dangling pointers, deallocations, etc).



For example, Rust compiler keeps tracks of all object lifetime (in this case it will try to make sure that none of the Object 1 or Object 2 will outlive the other one) – meaning that you can not destroy one without destroying the other one. On the other hand, using the previous 3 solutions (6, 7 and 8) , this check will have to be made by the programmer.

Reference Count

The solution for this problems is a special template(generic) called ***Rc*** (<u>R</u>eference <u>C</u>ount):

Rc (rc.rs)

```
pub struct Rc<T: ?Sized> {
    ptr: NonNull<RcBox<T>>,
    phantom: PhantomData<RcBox<T>>,
}
```

RcBox (rc.rs)

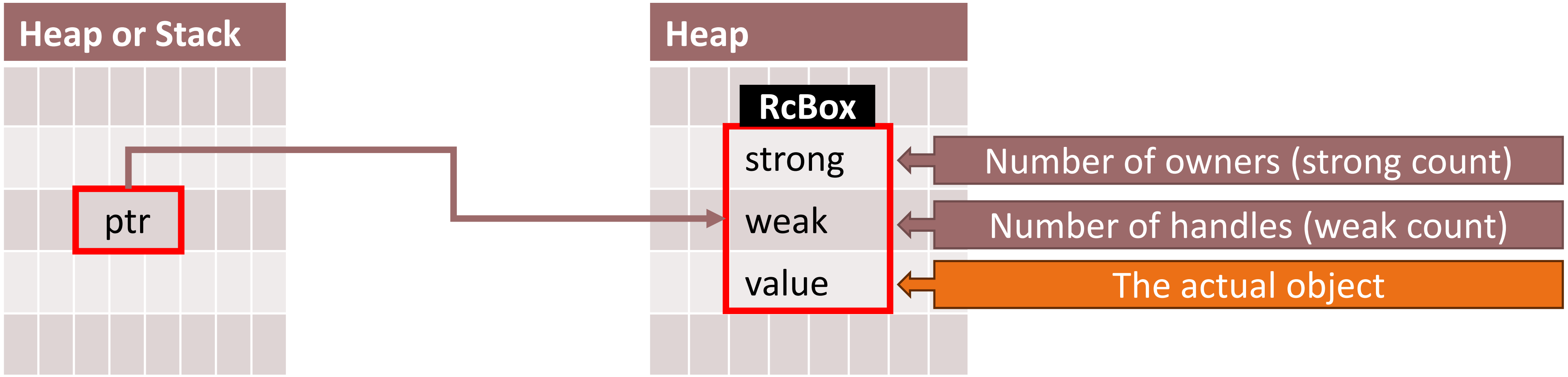
```
struct RcBox<T: ?Sized> {
    strong: Cell<usize>,
    weak: Cell<usize>,
    value: T,
}
```

Rc are heap allocated objects that maintain two counts:

- **A strong count** → how many objects own (have a reference) towards the current object. An object will never be destroyed as long as this count is bigger than 0
- **A weak count** → this works more like a handle (meaning that you refer something, bu you don't have ownership over it or more precisely you can not control its lifetime) object might be destroyed (but not deallocated) event if this count is bigger thar

Reference Count

Let's see the memory layout for a Rc:



The size of a Rc<T> will be 4 or 8 (size of a pointer, depending on architecture).

The actual size a Rc<T> in memory is: sizeof(ptr) + sizeof(strong) + sizeof(weak) + sizeof(T). Since ***ptr***, ***strong*** and ***week*** have the same size(4 or 8), then the actual size of a Rc<T> is minimum 12 or 24 + sizeof(T) (depending on alignament)

Reference Count

Let's see how a Rc works:

1. To add a new owner over the data use .clone() or Rc::clone(…) methods. This will increase the strong count, and create a new Rc<T> object with the same ptr as the original one
2. To create a weak reference (a handle – Weak<T>) use Rc::downgrade(…) method. This will increase the weak count.
3. Whenever a Rc<T> lifetime has ended, the strong count is decreased. If the strong count reaches 0, the destructor for object T is called.
4. Whenever a Weak<T> lifetime has ended, the weak count is decreased.
5. When both strong and weak counts reach 0, the **RcBox** is actually deallocated from memory.

Reference Count

Let's see an example – but first lets prepare a struct:

Rust

```rust
use std::rc::{Rc,Weak};

struct MyString {
    text: String,
}
impl MyString {
    fn new(text: &str) -> Self {
        Self {text: String::from(text) }
    }
}
impl Drop for MyString {
    fn drop(&mut self) {
        println!("Dropping MyString");
    }
}
fn print_stats(name: &str, obj: &Rc<MyString>) {
    println!("{} -> [S={},W={}]",name, Rc::strong_count(obj), Rc::weak_count(obj));
}
```

Reference Count

Let's see an example – but first lets prepare a struct:

Rust

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

struct MyString {
    text: String,
}
impl MyString {
    fn new(text: &str) -> Self {
        Self {text: String::from(text) }
    }
}
impl Drop for MyString {
    fn drop(&mut self) {
        println!("Dropping MyString");
    }
}
fn print_stats(name: &str, obj: &Rc<MyString>) {
    println!("{} -> [S={},W={}]",name, Rc::strong_count(obj), Rc::weak_count(obj));
}
```

Our struct contains a String (so that we have another heap allocation).

Reference Count

Let's see an example – but first lets prepare a struct:

Rust

```rust
use std::rc::{Rc,Weak};

struct MyString {
    text: String,
}
impl MyString {
    fn new(text: &str) -> Self {
        Self {text: String::from(text) }
    }
}
impl Drop for MyString {
    fn drop(&mut self) {
        println!("Dropping MyString");
    }
}
fn print_stats(name: &str, obj: &Rc<MyString>) {
    println!("{} -> [S={},W={}]",name, Rc::strong_count(obj), Rc::weak_count(obj));
}
```

We will also implement the Drop trait (so that we will have a notification when this object is dropped)

Reference Count

Let’s see an example – but first lets prepare a struct:

Rust

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

struct MyString {
    text: String,
}
impl MyString {
    fn new(text: &str) -> Self {
        Self {text: String::from(text) }
    }
}
impl Drop for
    fn drop(&
        print
    }
}
fn print_stats(name: &str, obj: &Rc<MyString>) {
    println!("{} -> [S={},W={}]",name, Rc::strong_count(obj), Rc::weak_count(obj));
}
```

Finally, the print_stats functions receives a Rc<MyString> reference and prints the number of strong and weak counts

Reference Count

Let's see an example (main code):

Rust

```
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
        print_stats("Owner-1",&owner_1);
        print_stats("Owner-2",&owner_2);
        w = Rc::downgrade(&owner_1);
        print_stats("Status",&owner_1);
        if let Some(owner_3) = w.upgrade() {
            println!("I have a new owner: {}",&owner_3.text);
            print_stats("Owner-3",&owner_3);
        }
        println!("--- destroy owners ---");
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```

Stack

Heap

Output

Reference Count

Let’s see an example (main code):

Rust

```
fn main() {
    let mut w = Weak::new();
    {
        let owner_     Rc::new(MyString::new("ABC"));

                                             wner_3.text);

        println!("--- destroy owners ---");
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```

This creates a weak reference with and invalid pointer address (meaning it has a pointer towards a `Rc<MyString>` but it is invalid (non-null)). This is useful if you want to create a weak reference and initialized it later.

```
pub struct Weak<T: ?Sized> {
    ptr: NonNull<RcBox<T>>,
}
```

Stack

w.ptr = ???

Heap

Output

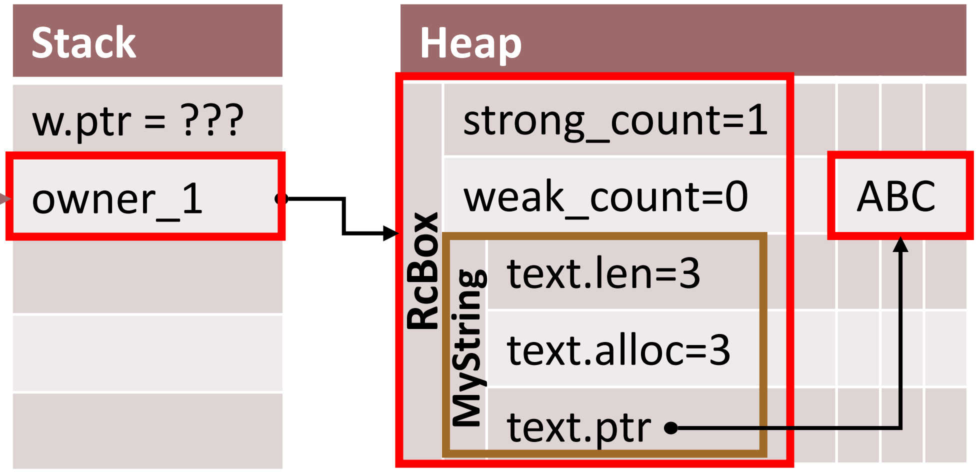
Reference Count

Let’s see an example (main code):

Rust

```
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single o   er",&owner_1);
        let owner_2 = owner_1   lone();
        pr
        pr
        w =
        pr
        if
            println!("I have a new owner: {}",&owner_3.text);
            print_stats("Owner-3",&owner_3);
        }
        println!("--- destroy owners ---");
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```

Rc<T> objects are created on the heap. Upon allocation we will have one strong count (as we have only one owner) an no weak counts.



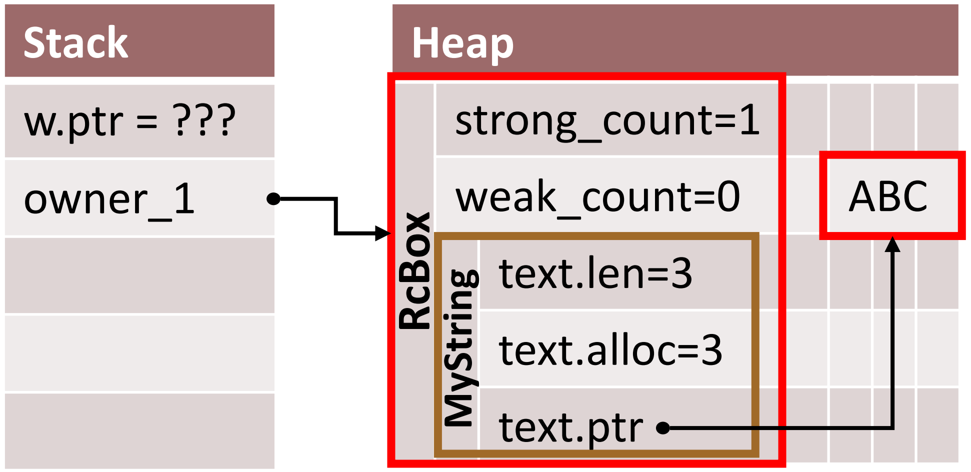
Output

Reference Count

Let's see an example (main code):

Rust

```rust
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
        print_stats("Owner-1",&owner_1);
        print_stats("Owner-2",&owner_2);
        w = Rc::downgrade(&owner_1);
        print_stats("Status",&owner_1);
        if let Some(owner_3) = w.upgrade() {
            println!("I have a new owner: {}",&owner_3.text);
            print_stats("Owner-3",&owner_3);
        }
        println!("--- destroy owners ---");
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```



Output

```
Single owner -> [S=1,W=0]
```

Reference Count

Let's see an example (main code):

Rust

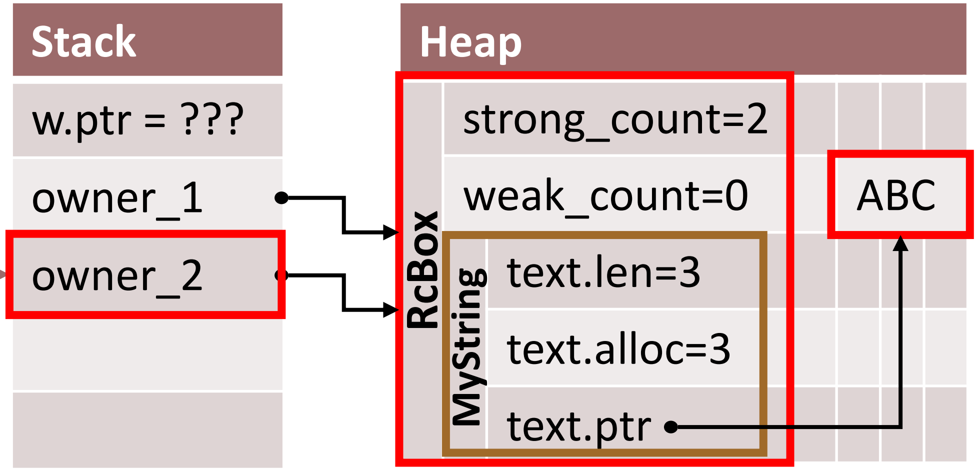
```
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
        print_stats("Own   -1",&owner_1);
        print_stats("Own   -2",&owner_2);

                                         ,&owner_3.text);

        }
    }
    if
        println!("Unable to gain ownership !");
    }
}
```

What .clone() method does is to copy the pointer to the existing ***RcBox*** and increase the strong count. In reality we would have only one object (***RcBox***).

Alternatively we can also use: Rc::clone(&owner_1)



Output

Single owner -> [S=1,W=0]

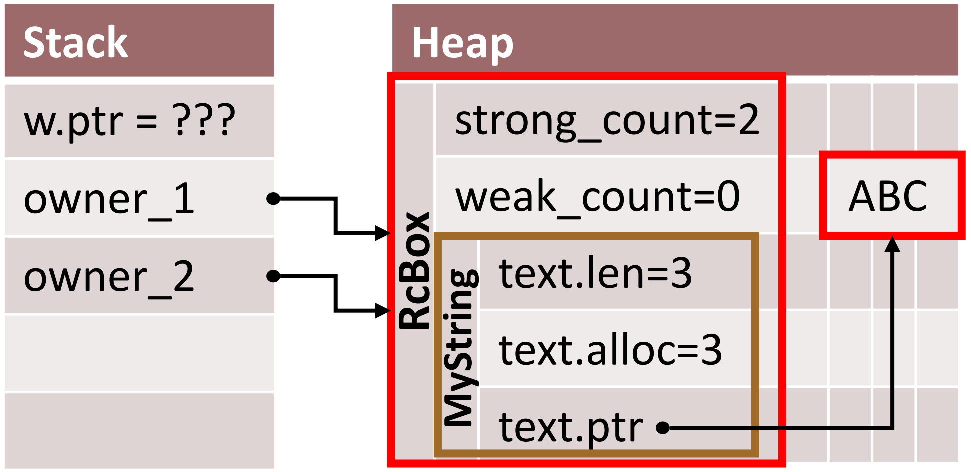
Reference Count

Let’s see an example (main code):

Rust

```
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
        print_stats("Owner-1",&owner_1);
        print_stats("Owner-2",&owner_2);
        w = Rc::downgra   (&owner_1);
        print_stats("St   us",&owner_1);
                                      ",&owner_3.text);
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```

Notice that we get the same strong and weak count for both owner_1 and owner_2. This is because both of them points to the same RcBox



Output

```
Single owner -> [S=1,W=0]
Owner-1 -> [S=2,W=0]
Owner-2 -> [S=2,W=0]
```

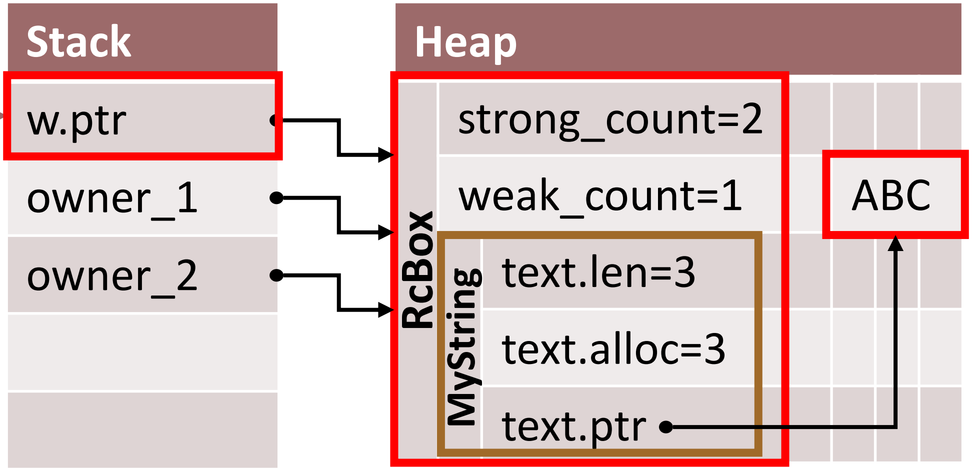
Reference Count

Let’s see an example (main code):

Rust

```
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
        print_stats("Owner-1",&owner_1);
        print_stats("Owner-2",&owner_2);
        w = Rc::downgrade(&owner_1);
        print_stats("Status",&owner_1);
        if let Some(owner_3) = w.upgrade() {
            println!("... have a new owner: {}", &owner_3.text);
                                                              ;
}
```

This is how we create a weak reference towards owner_1. In reality, just the pointer is being copied and as such, we will have an object that points to the same RcBox as owner_1 and owner_2. The weak count is also increased by 1.



Output

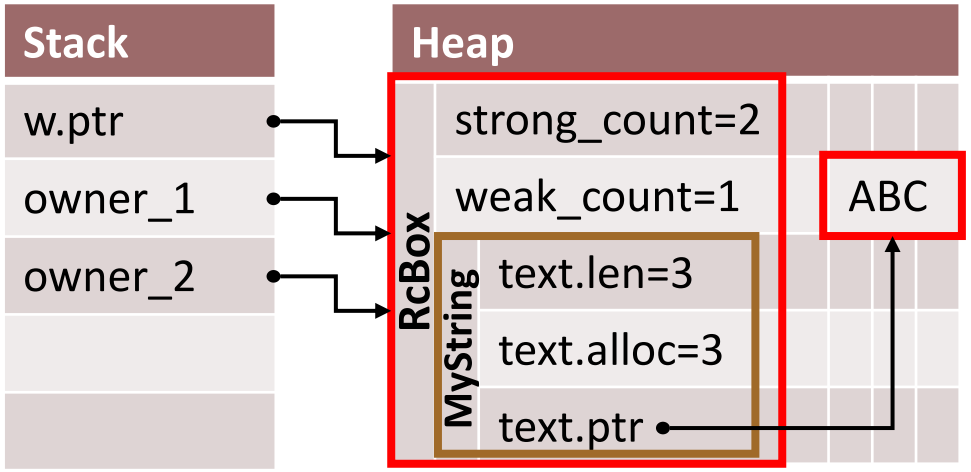
```
Single owner -> [S=1,W=0]
Owner-1 -> [S=2,W=0]
Owner-2 -> [S=2,W=0]
```

Reference Count

Let's see an example (main code):

Rust

```
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
        print_stats("Owner-1",&owner_1);
        print_stats("Owner-2",&owner_2);
        w = Rc::downgrade(&owner_1);
        print_stats("Status",&owner_1);
        if let Some(owner_3) = w.upgrade() {
            println!("I have a new owner: {}",&owner_3.text);
            print_stats("Owner-3",&owner_3);
        }
        println!("--- destroy owners ---");
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```



Output

```
Single owner -> [S=1,W=0]
Owner-1 -> [S=2,W=0]
Owner-2 -> [S=2,W=0]
Status -> [S=2,W=1]
```

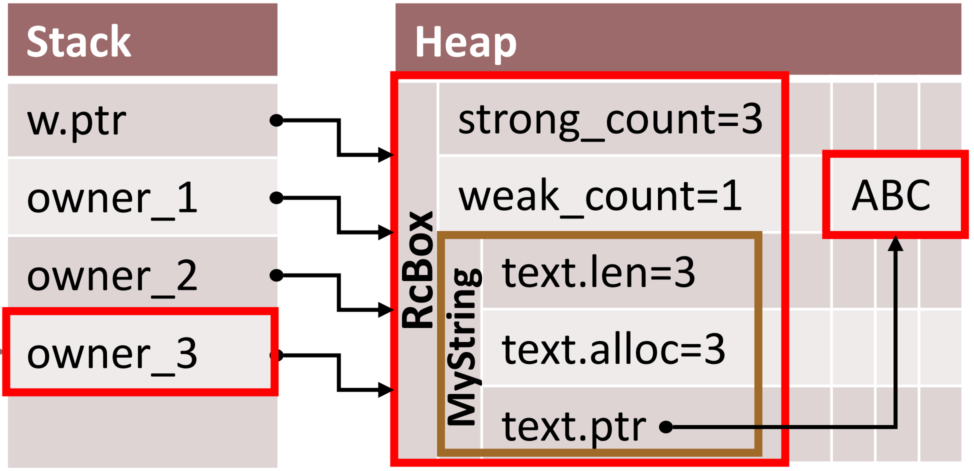
Reference Count

Let's see an example (main code):

Rust

Method .upgrade() from a week reference, checks to see if the strong count is bigger than 0 and if it does, it creates a new Rc<T> and increases the strong count. If the strong reference is 0, it will return None. This means that a Weak<T> can exists even if all Rc<T> were destroyed (only that in that moment the .ugrade() method will return None).

```
        print_stats("Status",&owner_1);
        if let Some(owner_3) = w.upgrade() {
            println!("I have a new owner: {}",&owner_3.text);
            print_stats("Owner-3",&owner_3);
        }
        println!("--- destroy owners ---");
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```



Output

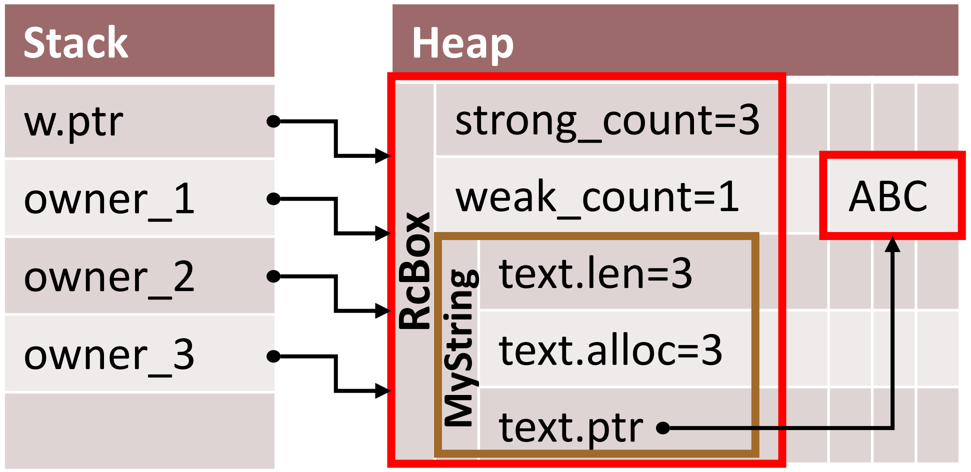
```
Single owner -> [S=1,W=0]
Owner-1 -> [S=2,W=0]
Owner-2 -> [S=2,W=0]
Status -> [S=2,W=1]
```

Reference Count

Let's see an example (main code):

Rust

```rust
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
        print_stats("Owner-1",&owner_1);
        print_stats("Owner-2",&owner_2);
        w = Rc::downgrade(&owner_1);
        print_stats("Status",&owner_1);
        if let Some(owner_3) = w.upgrade() {
            println!("I have a new owner: {}",&owner_3.text);
            print_stats("Owner-3",&owner_3);
        }
        println!("--- destroy owners ---");
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```



Output

```
Single owner -> [S=1,W=0]
Owner-1 -> [S=2,W=0]
Owner-2 -> [S=2,W=0]
Status -> [S=2,W=1]
I have a new owner: ABC
Owner-3 -> [S=3,W=1]
```

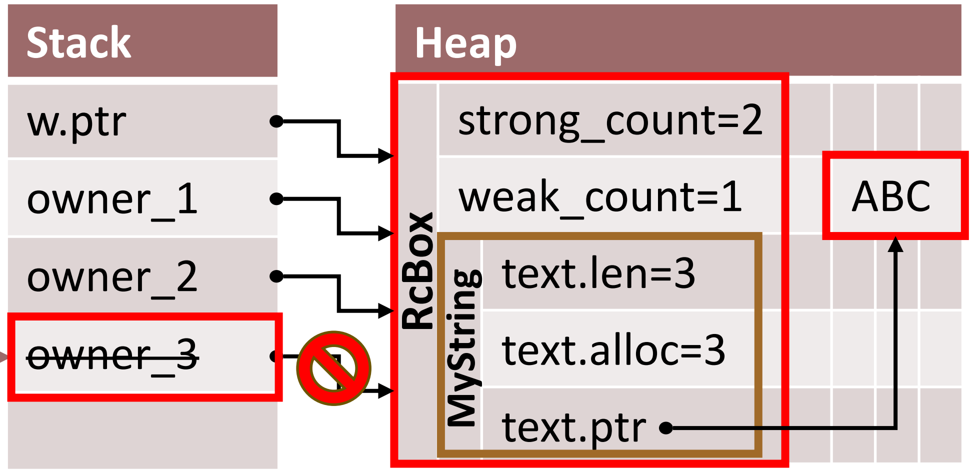
Reference Count

Let's see an example (main code):

Rust

```
fn main() {
    let mut w = Weak::new();
                  ::new(MyString::new("ABC"));
                  gle owner",&owner_1);
                  ner_1.clone();
                  er-1",&owner_1);
                  er-2",&owner_2);
                  e(&owner_1);
                  tus",&owner_1);
                  r_3) = w.upgrade() {
                  have a new owner: {}",&owner_3.text);
            print_stats("Owner-3",&owner_3);
        }
        println!("--- destroy owners ---");
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```

When the if let statement ends, owner_3 is dropped. When this happens, the strong count is decreased by 1. Since the new value (2) is still bigger than 0, this is everything that happens in this point.



Output

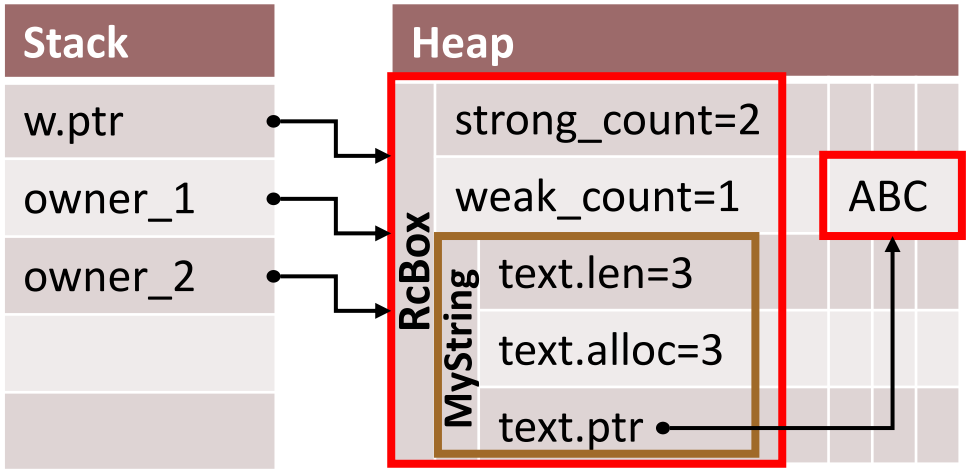
Single owner -> [S=1,W=0]
Owner-1 -> [S=2,W=0]
Owner-2 -> [S=2,W=0]
Status -> [S=2,W=1]
I have a new owner: ABC
Owner-3 -> [S=3,W=1]

Reference Count

Let's see an example (main code):

Rust

```rust
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
        print_stats("Owner-1",&owner_1);
        print_stats("Owner-2",&owner_2);
        w = Rc::downgrade(&owner_1);
        print_stats("Status",&owner_1);
        if let Some(owner_3) = w.upgrade() {
            println!("I have a new owner: {}",&owner_3.text);
            print_stats("Owner-3",&owner_3);
        }
        println!("--- destroy owners ---");
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```



Output

```
Single owner -> [S=1,W=0]
Owner-1 -> [S=2,W=0]
Owner-2 -> [S=2,W=0]
Status -> [S=2,W=1]
I have a new owner: ABC
Owner-3 -> [S=3,W=1]
--- destroy owners ---
```

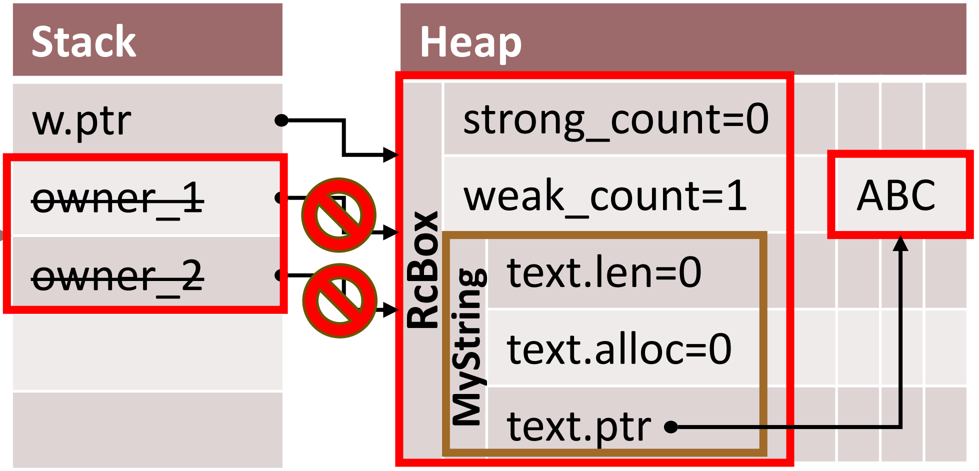
Reference Count

Let's see an example (main code):

Rust

```
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
        print_stats("Owner-1",&owner_1);
              ("Owner-2",&owner_2);
             ngrade(&owner_1);
              ("Status",&owner_1);
              (owner_3) = w.upgrade() {
              !("I have a new owner: {}",&owner_3.text);
              tats("Owner-3",&owner_3);
        }
        println!("--- destroy owners ---");
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```

When the inner block {..} ends, both owner_1 and owner_2 are dropped and the strong count for the Rc object reaches 0



Output

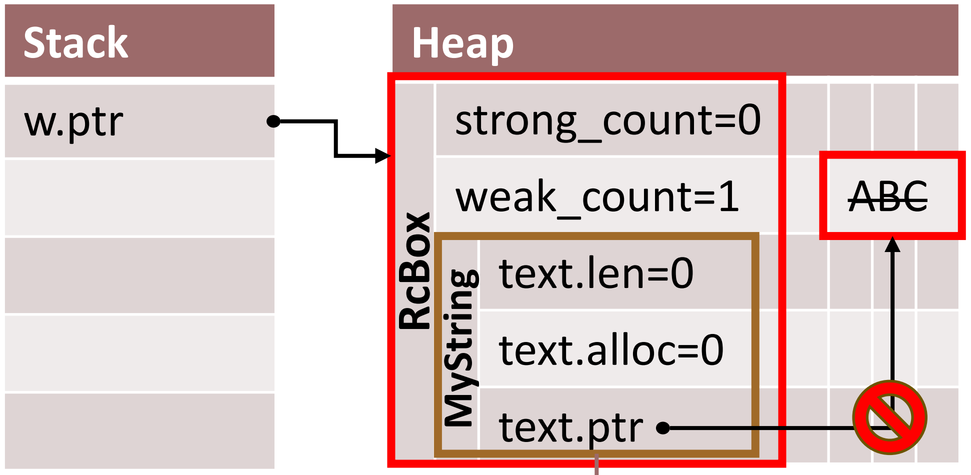
```
Single owner -> [S=1,W=0]
Owner-1 -> [S=2,W=0]
Owner-2 -> [S=2,W=0]
Status -> [S=2,W=1]
I have a new owner: ABC
Owner-3 -> [S=3,W=1]
--- destroy owners ---
```

Reference Count

Let’s see an example (main code):

Rust

```
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
        print_stats("Owner-1",&owner_1);
        print_stats("Owner-2",&owner_2);
        w = Rc::downgrade(&owner_1);
        print_stats("Status",&owner_1);
...
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```



Since the strong count is 0, there is no need to keep the MyString object (so Rust will call its destructor). This means that the pointer of the string within the ***MyString*** object will no longer be valid. However, the ***RcBox*** is not destroyed as the weak count is not 0 yet. As a result, MyObject still exists in memory (only that the text.ptr is invalid). We will also see the .drop() method called as a result of calling the destructor.

```
Owner-3 -> [S=3,W=1]
--- destroy owners ---
Dropping MyString
```

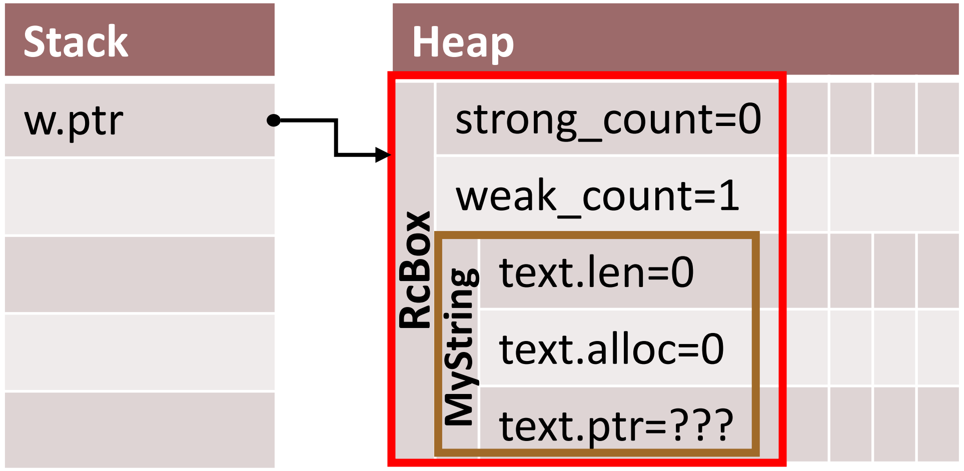
Reference Count

Let’s see an example (main code):

Rust

```
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
                                                      er_3.text);
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```

At this point we can safety call the .upgrade() method because the ***RcBox*** was not destroyed. However, since the strong count is 0, the result will be None (keep in mind that the MyString object was dropped and we can not provide a valid reference to it at this point).



Output

```
Single owner -> [S=1,W=0]
Owner-1 -> [S=2,W=0]
Owner-2 -> [S=2,W=0]
Status -> [S=2,W=1]
I have a new owner: ABC
Owner-3 -> [S=3,W=1]
--- destroy owners ---
Dropping MyString
Unable to gain ownership !
```

Reference Count

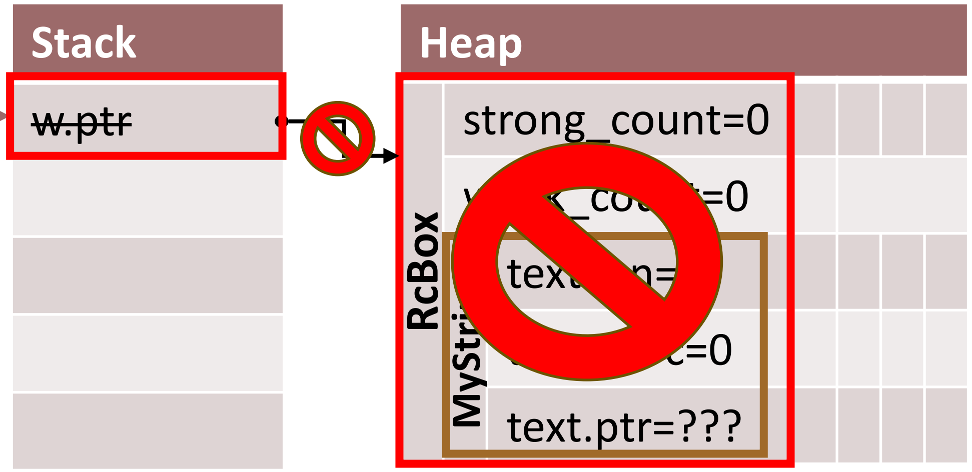
Let’s see an example (main code):

Rust

```
fn main() {
    let mut w = Weak::new();
    {
        let owner_1 = Rc::new(MyString::new("ABC"));
        print_stats("Single owner",&owner_1);
        let owner_2 = owner_1.clone();
                                   ",&owner_1);
                                   ",&owner_2);
                                   wner_1);
                                   ,&owner_1);
                                   .upgrade() {
                                   a new owner: {}",&owner_3.text);
                                   er-3",&owner_3);

                                  destroy owners ---");
    }
    if w.upgrade().is_none() {
        println!("Unable to gain ownership !");
    }
}
```

When the program ends, w is being dropped and with it the number of weak references reaches 0. At this point both weak and strong counts are 0 and Rust decides to drop the ***RcBox*** object.



Output

```
Single owner -> [S=1,W=0]
Owner-1 -> [S=2,W=0]
Owner-2 -> [S=2,W=0]
Status -> [S=2,W=1]
I have a new owner: ABC
Owner-3 -> [S=3,W=1]
--- destroy owners ---
Dropping MyString
Unable to gain ownership !
```

Reference Count

Let’s overview some of the methods from **Rc**:

Method (Rc<T>)	**Usage**
`fn new(value: T) -> Rc<T>`	Use to construct a new Rc object.
`fn new_cyclic<F>(data_fn: F) -> Rc<T>`	Use to construct a cyclic Rc object
`fn downgrade(this: &Self) -> Weak<T>`	Creates a new Weak<T> from an existing Rc<T>.
`fn weak_count(this: &Self) -> usize`	Returns the number of weak counts
`fn strong_count(this: &Self) -> usize`	Returns the number of weak counts
`fn get_mut(this: &mut Self) -> Option<&mut T>`	Returns a mutable reference only if strong count is 1 and weak count if 0

Reference Count

Let’s overview some of the methods from **Weak**:

Method (Weak<T>)	Usage
`fn new() -> Weak<T>`	Use to construct a new Weak object with an invalid pointer to an RcBox (unlinked). To create a Weak object that points to an RcBox use the **.downgrade(…)** method from Rc<T>
`fn upgrade(&self) -> Option<Rc<T>>`	Creates a Rc<T> from an Weak<T> only if the strong count of the RcBox where Weak<T> points to is bigger than 0.
`fn weak_count(this: &Self) -> usize`	Returns the number of weak counts
`fn strong_count(this: &Self) -> usize`	Returns the number of weak counts

Reference Count

Observations:

- Notice that some of these methods does not receive a self object but a parameter named this of type &Self. This means that these methods can be accessed via Rc:<method> and not directly through the object. This allows avoiding a confusion of having the same method defined in the object as well. Keep in mind the ***Rc<T>*** implements ***Deref*** trait meaning that you could access the methods of the T type directly from a ***Rc<T>*** object.
- **Rc** and **Weak** are subject to further optimizations. Currently, if the number of strong count is 0, .weak_count() method will return 0 even if the number of weak count is bigger than 0

Reference Count

Implemented traits for **Rc<T>** and **Weak<T>**

Trait	Rc<T>	Weak<T>
Clone	Increments strong count	Increments weak count
Drop	Decrements strong count. If strong count is 0 calls the destructor of type T. If both strong and weak counts are 0, it deallocates the Rc<T> from memory.	Decrements weak count. If both strong and weak counts are 0, it deallocates the Rc<T> from memory.
Deref	Provides access to the T object	-
Eq , PartialEq Ord, PartialOrd	Provides comparation of Rc<T> based on type T	-
AsRef, Borrow	Provides a direct immutable reference to the inner object of type T from an Rc<T>	-
Default	Creates a new Rc<T> if T supports a default value.	Calls Weak::new() – creates a new Weak<T> with an invalid pointer to an RcBox object.

Reference Count

Observations:

- There is no trait that allows to access the inner object of type ***T*** from a Weak<T>. This is done on purpose, as there is a possibility that a Weak<T> might exists, but the actual inner object of type T doesn't (e.g. if the strong count has reached 0). As such, you can only access the inner object via .upgrade(...) method that checks first to see if the inner object exist. In a way, we can say that Weak<T> behaves like a handle (you have him, but in order to access the data you need to check its validity every time).

- Rc<T> implements Deref, AsRef and Borrow but ***no trait that allows mutable access to data*** (such as DerefMut, AsMut or BorrowMut). This means that Rc<T> (as it is) can be used to read data but not to modify it !

Reference Count

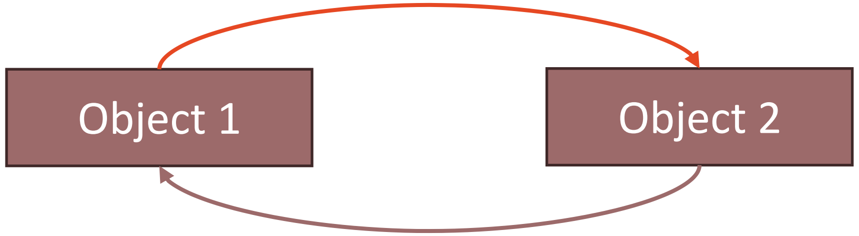
Let's try to solve the last problem but using Rc this time. One way of **trying to write this** will be as follows:

Rust

```
use std::rc::Rc;

struct Object1 { link: Option<Rc<Object2>> }
struct Object2 { link: Option<Rc<Object1>> }

fn main() {
    let mut o1 = Rc::new(Object1 { link: None });
    let mut o2 = Rc::new(Object2 { link: None });
    o1.link = Some(o2);
    o2.link = Some(o1);
}
```



Reference Count

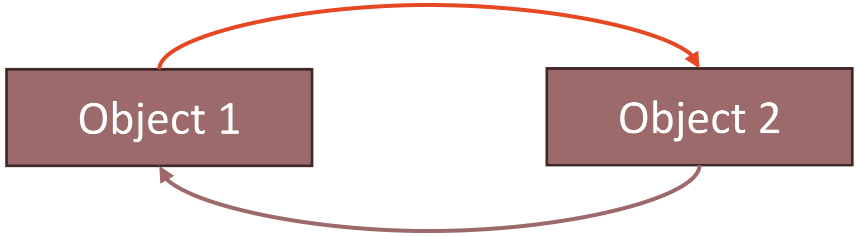
Let’s try to solve the last problem but using Rc this time. One way of **<u>trying to write this</u>** will be as follows:

Rust

```rust
use std::rc::Rc;

struct Object1 { link: Option<Rc<Object2>> }
struct Object2 { link: Option<Rc<Object1>> }

fn main() {
    let mut o1 = Rc::new(Object1 { link: None });
    let mut o2 = Rc::new(Object2 { link: None });
    o1.link = Some(o2);
    o2.link = Some(o1);
}
```



Error

```
error[E0594]: cannot assign to data in an `Rc`
  --> src\main.rs:13:5
   |
13 |     o1.link = Some(o2);
   |     ^^^^^^^ cannot assign
   |
   = help: trait `DerefMut` is required to modify through a dereference,
but it is not implemented for `Rc<Object1>`
```

Reference Count

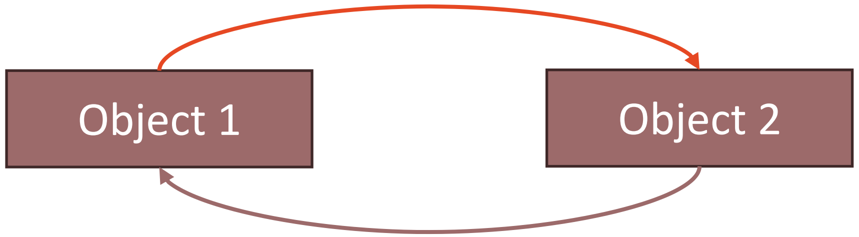
Let's try to solve the last problem but using Rc this time. One way of **trying to write this** will be as follows:

Rust

```rust
use std::rc::Rc;

struct Object1 { link: Option<Rc<Object2>> }
struct Object2 { link: Option<Rc<Object1>> }

fn main() {
    let mut o1 = Rc::new(Object1 { link: None });
    let mut o2 = Rc::new(Object2 { link: None });
    o1.link = Some(o2);
    o2.link = Some(o1);
}
```



Error

```
error[E0382]: borrow of moved value: `o2`
   --> src\main.rs:14:5
    |
12  |     let mut o2 = Rc::new(Object2 { link: None });
    |         ------ move occurs because `o2` has type `Rc<Object2>`,
    |                which does not implement the `Copy` trait
13  |     o1.link = Some(o2);
    |                    -- value moved here
14  |     o2.link = Some(o1);
    |     ^^^^^^^ value borrowed here after move
    |
    = note: borrow occurs due to deref coercion to `Object2`
```

Reference Count

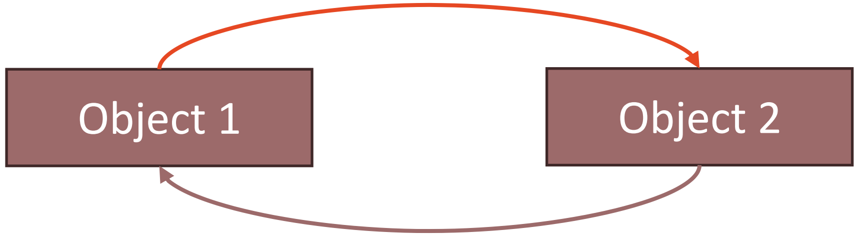
Since that solution did not work, let's try creating an Rc object directly when initializing Object1 or Object2

Rust

```
use std::rc::Rc;

struct Object1 { link: Rc<Object2> }
struct Object2 { link: Option<Rc<Object1>> }

fn main() {
    let mut o1 = Rc::new(Object1 {
        link: Rc::new(Object2 { link: None }),
    });
    o1.link.link = Some(o1);
}
```



Reference Count

Since that solution did not work, let’s try creating an Rc object directly when initializing Object1 or Object2

Rust

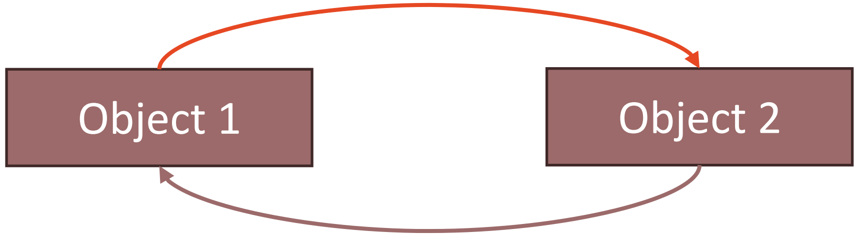
```
use std::rc::Rc;

struct Object1 { link: Rc<Object2> }
struct Object2 { link: Optio Rc<Object1>> }

fn ma
    l

    });
    o1.link.link = Some(o1);
}
```

Notice that since we constructing the data member link directly, we will no longer need the Option template for a lazy/late initialization.



Reference Count

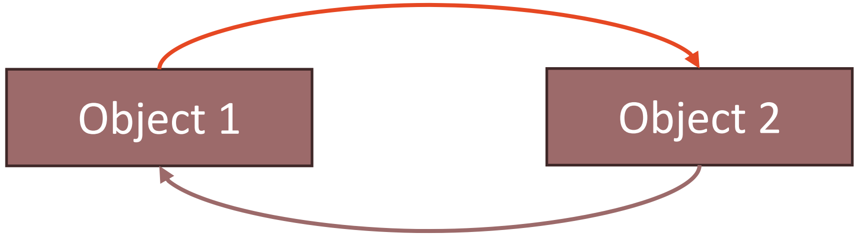
Since that solution did not work, let's try creating an Rc object directly when initializing Object1 or Object2

Rust

```rust
use std::rc::Rc;

struct Object1 { link: Rc<Object2> }
struct Object2 { link: Option<Rc<Object1>> }

fn main() {
    let mut o1 = Rc::new(Object1 {
        link: Rc::new(Object2 { link: None }),
    });
    o1.link.link = Some(o1);
}
```

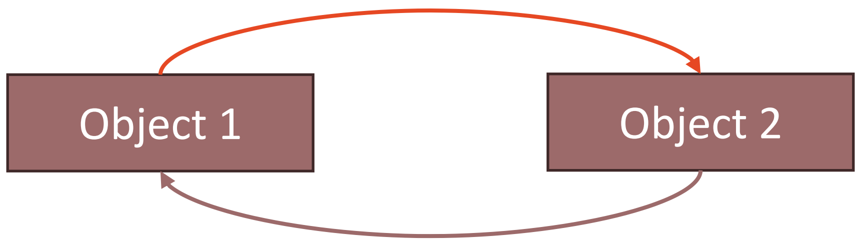


Error

```
error[E0382]: borrow of moved value: `o1`
   --> src\main.rs:14:5
    |
11  |     let mut o1 = Rc::new(Object1 {
    |         ------ move occurs because `o1` has type `Rc<Object1>`,
    |                which does not implement the `Copy` trait
...
14  |     o1.link.link = Some(o1);
    |     ^^^^^^^             -- value moved here
    |     |
    |     value borrowed here after move
```

Reference Count

So why this is not working ? The main problem is that if we have a cycle (like in our case where ***Object1*** owns through a Rc ***Object2*** and ***Object2*** owns through a Rc ***Object1***), those object will never be deallocated (the strong count will always be 2) and we will end up with memory leaks.



The solution in this case is to make one link of type Rc and the other one of type Weak.

Reference Count

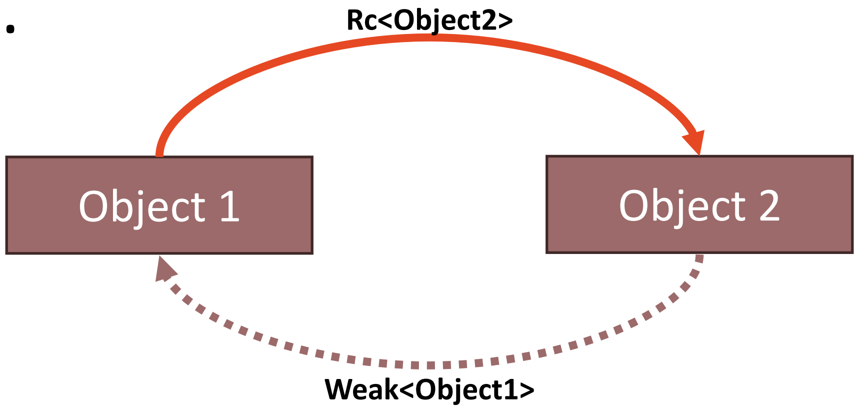
In this case we will try to have a Rc<T> link from Object1 to Object2 and a Weak<T> link from Object2 to Object 1.

Rust

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

struct Object1 {
    link: Rc<Object2>
}
struct Object2 {
    link: Weak<Object1>
}

fn main() {
    let mut o1 = Rc::new(Object1 {
        link: Rc::new(Object2 { link: Weak::new() }),
    });
    o1.link.link = Rc::downgrade(&o1);
}
```



Reference Count

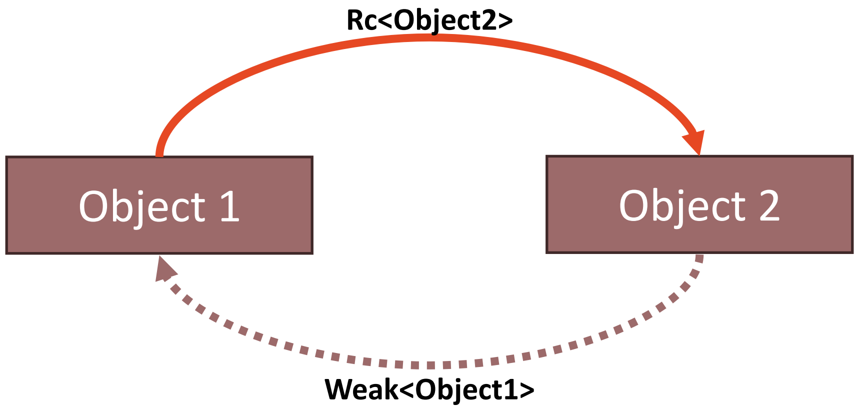
However, this solution will not work as an ***Rc<T>*** object does not implement DerefMut trait.

Rust

```rust
use std::rc::{Rc, Weak};

struct Object1 {
    link: Rc<Object2>
}
struct Object2 {
    link: Weak<Object1>
}

fn main() {
    let mut o1 = Rc::new(Object1 {
        link: Rc::new(Object2 { link: Weak::new() }),
    });
    o1.link.link = Rc::downgrade(&o1);
}
```



Error

```
error[E0594]: cannot assign to data in an `Rc`
  --> src\main.rs:15:5
   |
15 |     o1.link.link = Rc::downgrade(&o1);
   |     ^^^^^^^^^^^^ cannot assign
   |
   = help: trait `DerefMut` is required to modify through a dereference,
   |       but it is not implemented for `Rc<Object2>`
```

Reference Count

The actual solution is to use the .new_cyclic() method from Rc<T>. This method is defined as follows:

```
fn new_cyclic<F>(data_fn: F) -> Rc<T> where F: FnOnce(&Weak<T>) -> T
```

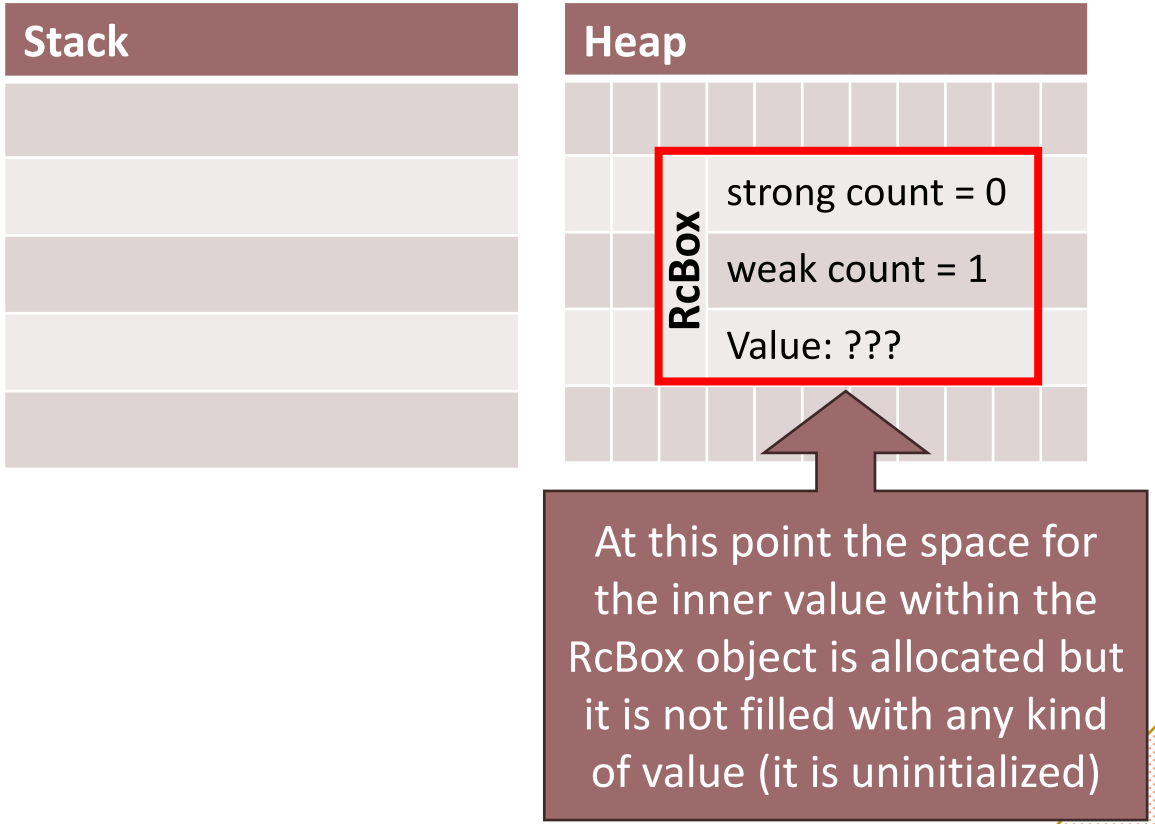
This method performs the following steps:

1. It allocates a new ***RcBox<T>*** data with strong count = 0 and weak count = 1
2. It creates a ***Weak<T>*** object over that ***RcBox<T>***. Since strong count is 0, any call to .upgrate() method will return ***None*** and as such there is no possibility to access the inner object of type ***T*** from the ***RcBox<T>***
3. It calls data_fn callback with the Weak object obtained in step 1, and gets an object of type T
4. It copies the memory from the returned object into the value field of the ***RcBox<T>***
5. It increments the strong count to 1
6. It returns a new ***Rc<T>*** based on the constructed ***RcBox<T>***

Reference Count

Let's see a graphical representation of how .new_cyclic() works:

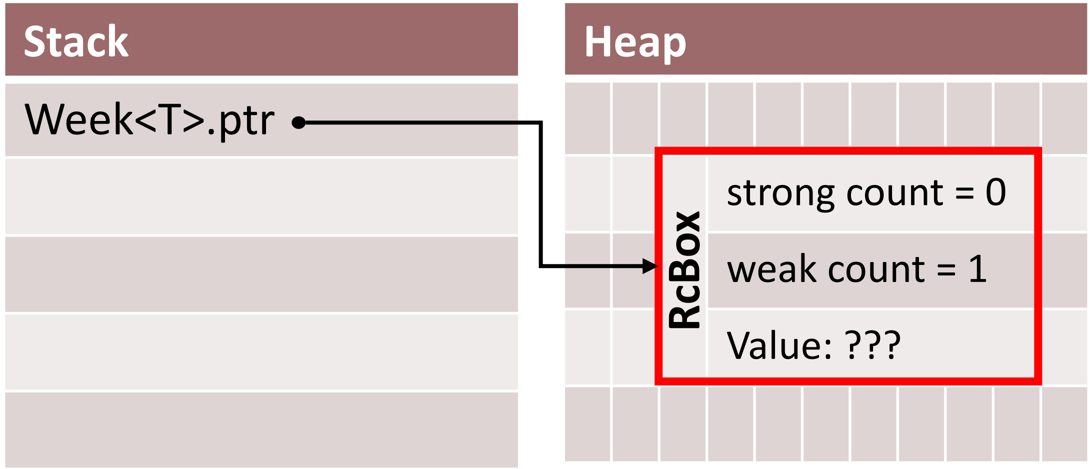
1. A new RcBox is create on the heap (strong count = 0, weak count = 1)



Reference Count

Let’s see a graphical representation of how .new_cyclic() works:

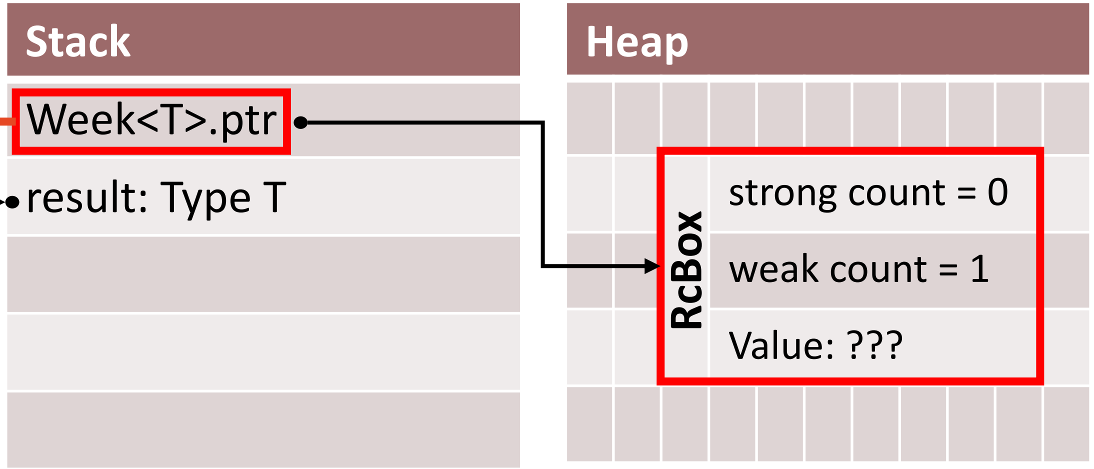
1. A new RcBox is create on the heap (strong count = 0, weak count = 1)
2. Creates a Weak<T> based on the RcBox created on step 1



Reference Count

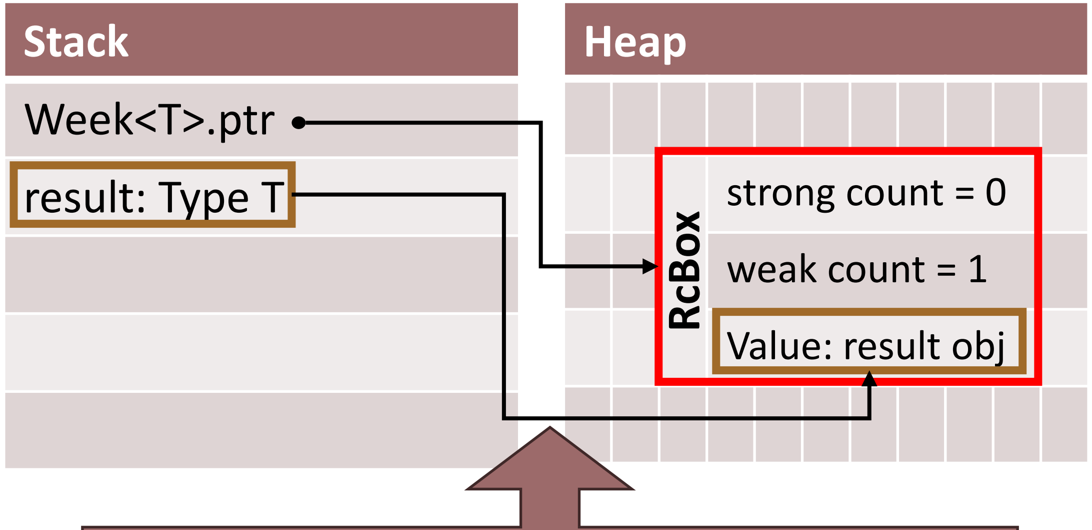
Let's see a graphical representation of how .new_cyclic() works:

1. A new RcBox is create on the heap (strong count = 0, weak count = 1)
2. Creates a Weak<T> based on the RcBox created on step 1
3. A call to data_fn(&Week<T>) is performed. The result of this call will be an object of type T that will be stored on the stack. This object can use the Week<T> object internally for its data member initialization.



Let’s see a graphical representation of how .new_cyclic() works:

1. A new RcBox is create on the heap (strong count = 0, weak count = 1)
2. Creates a Weak<T> based on the RcBox created on step 1
3. A call to data_fn(&Week<T>) is performed. The result of this call will be an object of type T that will be stored on the stack. This object can use the Week<T> object internally for its data member initialization.
4. The resulted object is being copied into the RcBox

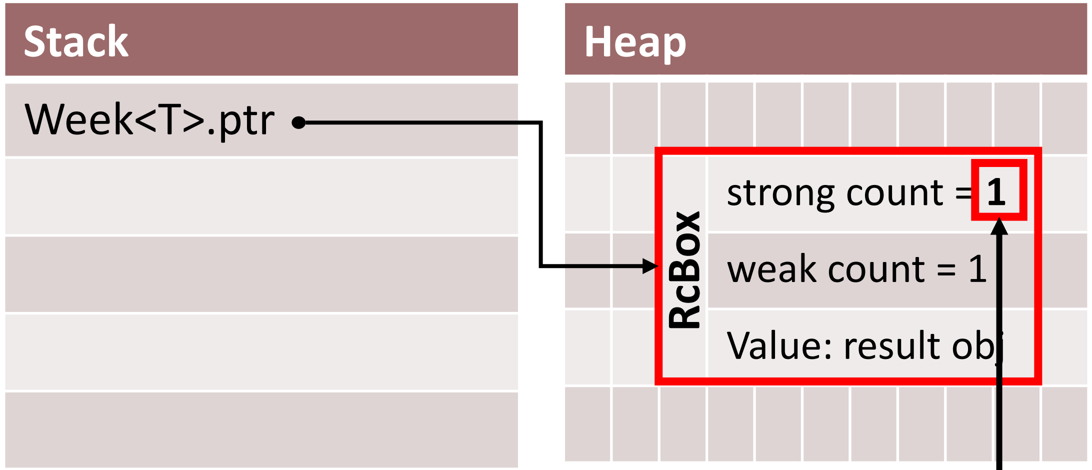


A bit-wise copy is performed at this point. From a semantic point of view, we move the resulted object of type T into the RcBox, meaning that after this operation, the resulted object will no longer be available.

Reference Count

Let’s see a graphical representation of how `.new_cyclic()` works:

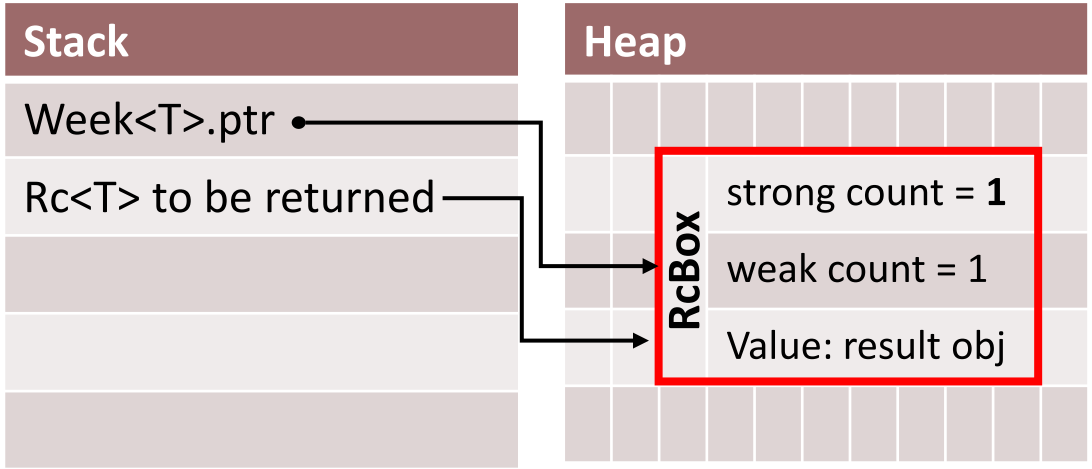
1. A new RcBox is create on the heap (strong count = 0, weak count = 1)
2. Creates a Weak<T> based on the RcBox created on step 1
3. A call to `data_fn( &Week<T> )` is performed. The result of this call will be an object of type T that will be stored on the stack. This object can use the Week<T> object internally for its data member initialization.
4. The resulted object is being copied into the RcBox
5. We increment the strong count (as we will return an Rc<T> object)



Reference Count

Let's see a graphical representation of how .new_cyclic() works:

1. A new RcBox is create on the heap (strong count = 0, weak count = 1)
2. Creates a Weak<T> based on the RcBox created on step 1
3. A call to data_fn(&Week<T>) is performed. The result of this call will be an object of type T that will be stored on the stack. This object can use the Week<T> object internally for its data member initialization.
4. The resulted object is being copied into the RcBox
5. We increment the strong count (as we will return an Rc<T> object)
6. We create a new Rc<T> over the existing RcBox and return it.



Reference Count

While this method is good enough to instantiate an such a construct, in reality there will be a need to modify data members that belong to an Rc<T>/Weak<T> smart pointer. Let's analyze the following code:

Rust

```rust
use std::rc::{Rc, Weak};
struct Object1 {
    link: Rc<Object2>,
    value: i32,
}
struct Object2 {
    link: Weak<Object1>,
    value: i32
}
fn main() {
    let o1 = Rc::new_cyclic(|me| {
        return Object1 { link: Rc::new(Object2 { link: me.clone(), value:0 }), value:0 };
    });
    println!("O1={}, O2={}",o1.value, o1.link.value);
}
```

Output

O1=0, O2=0

Reference Count

What if we want to change the field .value from both Object1 and Object2 after we construct them ?

Rust

```
use std::rc::{Rc, Weak};
struct Object1 {
    link: Rc<Object2>,
    value: i32,
}
    ...ic(|me| {
        return Object1 { link: Rc::new(Objec
    });
    o1.value = 10;
    o1.link.value = 20;
    println!("O1={}, O2={}",o1.value, o1.link.value);
}
```

Rc<T> does not implement ***DerefMut*** and as such, none of the data member of type T can me modified directly.

Error

```
error[E0594]: cannot assign to data in an `Rc`
  --> src\main.rs:19:5
   |
19 |     o1.value = 10;
   |     ^^^^^^^^^^^^^ cannot assign
   |
   = help: trait `DerefMut` is required to modify through a dereference,
           but it is not implemented for `Rc<Object1>`

error[E0594]: cannot assign to data in an `Rc`
  --> src\main.rs:20:5
   |
20 |     o1.link.value = 20;
   |     ^^^^^^^^^^^^^^^^^^ cannot assign
   |
   = help: trait `DerefMut` is required to modify through a dereference,
           but it is not implemented for `Rc<Object2>`
```

Reference Count

The solution is to use **interior mutability** through a RefCell:

Rust

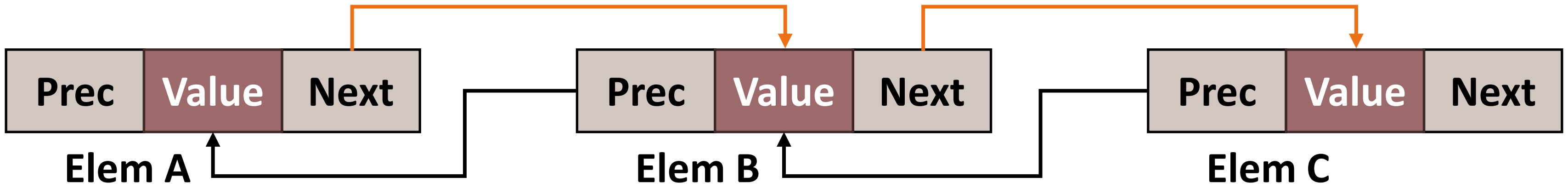
```
use std::{rc::{Rc, Weak}, cell::RefCell};
struct Object1 {
    link: Rc<RefCell<Object2>>,
    value: i32,
}
struct Object2 {
    link: Weak<RefCell<Object1>>,
    value: i32
}
fn main() {
    let o1 = Rc::new_cyclic(|me| {
        return RefCell::new(Object1 {
            link: Rc::new(RefCell::new(Object2 { link: me.clone(), value:0 })),
            value:0
        });
    });
    (*((*o1).borrow_mut())).value = 10;
    (*(*((*((*o1).borrow_mut())).link)).borrow_mut()).value = 20;
    println!("O1={}, O2={}",(*((*o1).borrow())).value, (*(*((*((*o1).borrow())).link)).borrow()).value);
}
```

Output

O1=10, O2=20

Reference Count

So … lets come back to the original problem (a double linked list) and discuss some options in Rust:



Options:

1. Use raw pointers or **NonNull** wrapper (similar like with C/C++)
2. Use a **Rc** combined with **Weak** and **CellRef** for interior mutability
3. Design the double linked list in a different way (keep all of the elements in a vector and store the **index** for the next and previous elements).
4. Use a **handle-like** system (keep all of the elements in a vector, but refer to each element through a handle that makes sures that you can not access an element just by using an index)

Reference Count

Each one of these 4 solutions will be embedded in a struct called DoubleLinkedList, and for each one of them we will design:

- **Node structure**
- An **initialization method** .new(…)
- A .sum_all(…) method (for this example we will assume that each node has a numerical value attached to it). The .sum_all(…) method iterates from the first to the last method (by using the .next data member from each node to go to the next one) and sums up all values.
- A .add(…) method that adds an element at the end of the double linked list and updates the .next and .previous data members
- **Additional methods** (if needed – e.g. for access an element)

Reference Count

1. Using **NonNull** (Node & DoubleLinkedList structures)

Rust (Node structure)

```
pub struct Node {
    pub next: Option<NonNull<Node>>,
    pub prec: Option<NonNull<Node>>,
    pub value: u64,
}
```

Rust (DoubleLinkedList structure)

```
pub struct DoubleLinkedList {
    pub head: *mut Node,
    pub tail: *mut Node,
}
```

Notice that we use a raw pointer in a DoubleLinkedList structure (similar to a C/C++ implementation). At the same time, .next and .prec are ***Option<NonNull<Node>>***, meaning that instead of using **null** to indicate that there is next or previous link, we can simply use None variant from Option.

Reference Count

1. Using **NonNull** (DoubleLinkedList implementation: .new() & .add() methods)

Rust (Node structure)

```
pub struct Node {

}
```

Rust (DoubleLinkedList structure)

Rust (DoubleLinkedList structure)

```
impl DoubleLinkedList {

    pub fn new(_capacity: usize) -> Self {
        let start_node = Box::into_raw(Box::new(Node { next: None, prec: None, value: 0 }));
        Self { head: start_node, tail: start_node }
    }

    pub fn add(&mut self, value: u64) {
        let new_node = Box::into_raw(Box::new(Node { next: None, prec: None, value }));
        unsafe {
            (*self.tail).next = Some(NonNull::new_unchecked(new_node));
            (*new_node).prec = Some(NonNull::new_unchecked(self.tail));
        }
        self.tail = new_node;
    }
    ......................................
}
```

Reference Count

1. Using **NonNull** (DoubleLinkedList implementation: .sum_all() method)

Rust (Node structure)

```
pub struct Node {
}
```

Rust (DoubleLinkedList structure)

Rust (DoubleLinkedList structure)

```
imp
}
```

Rust (DoubleLinkedList structure)

```
impl DoubleLinkedList {
    pub fn new(_capacity: usize) -> Self {…}
    pub fn add(&mut self, value: u64) {…}
    pub fn sum_all(&self) -> u64 {
        let mut sum = 0;
        let mut current = self.head;
        loop {
            unsafe {
                sum += (*current).value;
                if let Some(next) = (*current).next { current = next.as_ptr(); } else { break; }
            }
        }
        sum
    }
}
```

Reference Count

2. Using **Rc** (Node & DoubleLinkedList structures)

Rust (Node structure)

```
pub struct Node {
    pub next: Option<Rc<RefCell<Node>>>,
    pub prec: Option<Weak<RefCell<Node>>>,
    pub value: u64,
}
```

Rust (DoubleLinkedList structure)

```
pub struct DoubleLinkedList {
    pub head: Rc<RefCell<Node>>,
    pub tail: Rc<RefCell<Node>>,
}
```

In this case we will use a Rc<…> for the ***.next*** field and a Weak<…> for the ***.prec*** field. The double linked list structure will store a Rc<…> for the head and for the tail (first and last elements in the list).

Reference Count

2. Using **Rc** (DoubleLinkedList implementation: .new() & .add() methods)

Rust (Node structure)

Rust (DoubleLinkedList structure)

Rust (DoubleLinkedList structure)

```rust
impl DoubleLinkedList {

    pub fn new(_capacity: usize) -> Self {
        let start_node = Rc::new(RefCell::new(Node {next: None, prec: None, value: 0 }));
        Self { head: start_node.clone(), tail: start_node.clone() }
    }
    pub fn add(&mut self, value: u64) {
        let new_node = Rc::new(RefCell::new(Node {
            next: None,
            prec: Some(Rc::downgrade(&self.tail)),
            value,
        }));
        (*self.tail.borrow_mut()).next = Some(new_node.clone());
        self.tail = new_node;
    }
    ......................................
}
```

Reference Count

2. Using **Rc** (DoubleLinkedList implementation: .sum_all() method)

Rust (Node structure)

Rust (DoubleLinkedList structure)

Rust (DoubleLinkedList structure)

Rust (DoubleLinkedList structure)

```
impl DoubleLinkedList {
    ………………
    pub fn sum_all(&self) -> u64 {
        let mut sum = 0;
        let mut current = self.head.clone();
        loop {
            let next_node;
            {
                let tmp = current.borrow_mut();
                sum += (*tmp).value;
                next_node = if let Some(next) = (*tmp).next.as_ref() { Some(next.clone()) } else { None };
            }
            if next_node.is_none() { break; }
            current = next_node.unwrap();
        }
        sum
    }
}
```

Reference Count

3. Using **index** in a vector (Node & DoubleLinkedList structures)

Rust (Node structure)

```rust
pub struct Node {
    pub next: usize,
    pub prec: usize,
    pub value: u64,
}
```

Rust (DoubleLinkedList structure)

```rust
pub struct DoubleLinkedList {
    data: Vec<Option<Node>>,
    pub head: usize,
    pub tail: usize,
}
```

In this case we store all elements in a vector (.data member from the DoubleLinkedList structure). Each element will have the index of next and previous element from the vector.

A special value defined in the following way: `const INVALID_INDEX: usize = usize::MAX;` will be used to mark an index as invalid (meaning it does not point to another element in the vector).

Reference Count

3. Using **index** in a vector (DoubleLinkedList implementation: .new() & .add() methods)

Rust (Node structure)

Rust (DoubleLinkedList structure)

Rust (DoubleLinkedList structure)

```
impl DoubleLinkedList {
    pub fn new(capacity: usize) -> Self {
        let mut me = Self { data: Vec::with_capacity(capacity), head: 0, tail: 0 };
        me.data.push(Some(Node { next: INVALID_INDEX, prec: INVALID_INDEX, value: 0 }));
        me
    }
    pub fn add(&mut self, value: u64) {
        let new_node = Node { next: INVALID_INDEX, prec: self.tail, value };
        self.data.push(Some(new_node));
        let last_index = self.data.len() - 1;
        if let Some(previous_tail) = self.get_node_mut(self.tail) {
            previous_tail.next = last_index;
        }
        self.tail = last_index;
    }
    ..........................................
}
```

Reference Count

3. Using **index** in a vector (other methods)

Rust (Node structure)

Rust (DoubleLinkedList structure)

Rust (DoubleLinked

```
impl DoubleLinkedL
    pub fn new(cap
        let mut me
        me.data.pu                                                                          }));
        me
    }
    pub fn add(&mu
        let new_node
        self.data.push(Some(new_node));
        let last_index = self.data.len() - 1;
        if let Some(previous_tail) = self.get_node_mut(self.tail) {
            previous_tail.next = last_index;
        }
        self.tail = last_index;
    }
    ......................................
}
```

```
fn get_node_mut(&mut self, index: usize) -> Option<&mut Node>
{
    if index < self.data.len() {
        self.data[index].as_mut()
    } else {
        None
    }
}
```

Reference Count

3. Using **index** in a vector (DoubleLinkedList implementation: .sum_all() method)

Rust (Node structure)

Rust (DoubleLinkedList structure)

Rust (DoubleLinkedList structure)

Rust (DoubleLinkedList structure)

```rust
impl DoubleLinkedList {
    pub fn new(_capacity: usize) -> Self {…}
    pub fn add(&mut self, value: u64) {…}
    pub fn get_node(&self, index: usize) -> Option<&Node> {…}
    pub fn get_node_mut(&mut self, index: usize) -> Option<&Node> {…}
    pub fn sum_all(&self) -> u64 {
        let mut sum = 0;
        let mut current = self.head;
        while let Some(node) = self.get_node(current) {
            sum += node.value;
            current = node.next;
        }
        sum
    }
}
```

Reference Count

3. Using **index** in a vector (other methods)

Rust (Node structure)

Rust (DoubleLinkedList structure)

Rust (DoubleLinkedList structure)

Rust (DoubleLin

```
fn get_node(&self, index: usize) -> Option<&Node> {
    if index < self.data.len() {
        self.data[index].as_ref()
    } else {
        None
    }
}
```

```
impl DoubleLink
    pub fn new(
    pub fn add(
    pub fn get_
    pub fn get_
    pub fn sum_
        let mut sum = 0;
        let mut current = self.head;
        while let Some(node) = self.get_node(current) {
            sum += node.value;
            current = node.next;
        }
        sum
    }
}
```

Reference Count

4. Using a **handle** (Node , DoubleLinkedList and Handle structures)

Rust (Node structure)

```
pub struct Node {
    pub next: Handle,
    pub prec: Handle,
    pub value: u64,
    pub unique_id: u32,
}
```

Rust (DoubleLinkedList structure)

```
pub struct DoubleLinkedList {
    data: Vec<Option<Node>>,
    pub head: Handle,
    pub tail: Handle,
}
```

Rust (Handle structure)

```
pub struct Handle {
    index: u32,
    unique_id: u32,
}
```

A handle contains two elements: an index into a vector and a unique identifier. The second one is being used to make sure that when you ask for an element, you know exactly what element you are referring to. In case of the index approach, one could just manufacture a valid index and would have access to that object. In this case, you can not just have the index of that object from the vector, you also need the unique id (that is private).

Reference Count

4. Using a **handle** (DoubleLinkedList implementation: .new() method)

Rust (Node structure) *Rust (DoubleLinkedList structure)* *Rust (Handle structure)*

Rust (DoubleLinkedList structure)

```
impl DoubleLinkedList {
    pub fn new(capacity: usize) -> Self {
        let first_element_handle = Handle::new(0);
        let mut me = Self {
            data: Vec::with_capacity(capacity),
            head: first_element_handle,
            tail: first_element_handle,
        };
        me.data.push(Some(Node {
            next: Handle::INVALID,
            prec: Handle::INVALID,
            value: 0,
            unique_id: first_element_handle.unique_id,
        }));
        me
    }
}
```

Reference Count

4. Using a **handle** (DoubleLinkedList implementation: .new() method)

Rust (Node structure)

```
pub struct Node {
}
```

Rust (DoubleLinkedList structure)

Rust (Handle structure)

Rust (DoubleLinkedList structure)

```
impl DoubleLinkedList {
    pub fn new(capacity: usize) -> Self {
        let first_element_handle = Handle::new(0);
        let mut me = Self {
            data: Vec::with_capacity(capacity),
            head: first_element_handle,
            tail: first_element_handle,
        };
        me.data.push(Some(Node {
            next: Handle::INVALID,
            prec: Handle::INVALID,
            value: 0,
            unique_id: first_element_handle.unique_id,
        }));
        me
    }
}
```

A special value that could never be a valid handle
(e.g: index = 0xFFFFFFFF, and unique_id = 0xFFFFFFFF)

Reference Count

4. Using a **handle** (DoubleLinkedList implementation: .add() method)

Rust (Node structure) | *Rust (DoubleLinkedList structure)* | *Rust (Handle structure)*

Rust (DoubleLinkedList structure)

```rust
impl DoubleLinkedList {
    pub fn add(&mut self, value: u64) {
        let new_elem_index = self.data.len();
        let new_node_handle = Handle::new(new_elem_index as u32);
        let new_node = Node {
            next: Handle::INVALID,
            prec: self.tail,
            value,
            unique_id: new_node_handle.unique_id,
        };
        self.data.push(Some(new_node));
        if let Some(previous_tail) = self.get_node_mut(self.tail) {
            previous_tail.next = new_node_handle;
        }
        self.tail = new_node_handle;
    }
}
```

Reference Count

4. Using a **handle** (DoubleLinkedList implementation: .add() method)

Rust (Node structu

Rust (DoubleLin

```rust
fn get_node_mut(&mut self, handle: Handle) -> Option<&mut Node> {
    let index = handle.index as usize;
    if index < self.data.len() {
        if let Some(obj) = &self.data[index] {
            if obj.unique_id == handle.unique_id {
                self.data[index].as_mut()
            } else { None }
        } else { None }
    } else { None }
}
```

```rust
impl DoubleLink
    pub fn add(
        let new
        let new
        let new
            nex
            pre
            val
            unique_id: new_node_handle.unique_id
        };
        self.data.push(Some(new_node));
        if let Some(previous_tail) = self.get_node_mut(self.tail) {
            previous_tail.next = new_node_handle;
        }
        self.tail = new_node_handle;
    }
}
```

Reference Count

4. Using a **handle** (DoubleLinkedList implementation: .sum_all() method)

Rust (Node structure)

Rust (DoubleLinkedList structure)

Rust (Handle structure)

Rust (DoubleLinkedList structure)

Rust (DoubleLinkedList structure)

```rust
impl DoubleLinkedList {
    pub fn new(_capacity: usize) -> Self {…}
    pub fn add(&mut self, value: u64) {…}
    pub fn get_node(&self, handle: Handle) -> Option<&Node> {…}
    pub fn get_node_mut(&mut self, handle: Handle) -> Option<&Node> {…}
    pub fn sum_all(&self) -> u64 {
        let mut sum = 0;
        let mut current = self.head;
        while let Some(node) = self.get_node(current) {
            sum += node.value;
            current = node.next;
        }
        sum
    }
}
```

Reference Count

For all of the 4 existing scenarios, we will compute the time required for:

1. Create a 10 million nodes double linked list
2. Iterate through all 10 millions of nodes and sums up the values from each nodes

All test were performed 10 times and times were recorded. The tests were performed using a release version of the previous snippets of code.

The test was performed on Window 11, over a laptop with the following configuration: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz.

A list of all 4 scenarios (and several others) can be found on the following github repository: https://github.com/gdt050579/double_linked_list_test-rs

Reference Count

Times (milliseconds) for creating a double linked list:

	#1 (ms)	#2 (ms)	#3 (ms)	#4 (ms)	#5 (ms)	#6 (ms)	#7 (ms)	#8 (ms)	#9 (ms)	#10 (ms)	Average
NonNull	502	496	534	511	493	508	510	500	503	499	**505.6**
Rc	612	624	611	611	601	606	628	605	692	708	**629.8**
Index	55	54	55	55	56	54	54	56	74	71	**58.4**
Handle	117	109	108	107	107	108	108	108	173	154	**119.9**

Times (milliseconds) for iterating a double linked list:

	#1 (ms)	#2 (ms)	#3 (ms)	#4 (ms)	#5 (ms)	#6 (ms)	#7 (ms)	#8 (ms)	#9 (ms)	#10 (ms)	Average
NonNull	62	62	63	62	61	70	62	63	63	65	**63.3**
Rc	130	133	134	132	134	134	132	134	152	149	**136.4**
Index	30	31	30	30	33	30	29	31	37	35	**31.6**
Handle	37	36	36	38	36	37	36	37	45	41	**37.9**

Reference Count

The overall results look as follows:

	Average creation time (ms)	Average iteration time (ms)	Memory size (MiB)
NonNull	**505.6**	**63.3**	**228.88**
Rc	**629.8**	**136.4**	**457.76**
Index	**58.4**	**31.6**	**305.18**
Handle	**119.9**	**37.9**	**381.47**

Some observations:

- **Index** and **Handle** based solutions rely on allocation of a large continuous memory. This means that creation time will be lower and as most CPUs cache memory, iteration will be faster
- **Rc** solution seems to be generally slower and requires more memory than any other solution
- **NonNull** (even if similar cu C/C++) will be slower than **Index** and **Handle** based solution due to the fact memory is not necessarily cached in their case (as allocation will probably not be a continuous block).

Reference Count

When dealing with large lists and recursive ownership (either via a Rc<T> or Box<T> object) there is one other aspect we need to take into consideration. Let's take a close look Rc<T> implementation for a double linked list:

Rust (DoubleLinkedList structure)

```rust
pub struct Node {
    pub next: Option<Rc<RefCell<Node>>>,
    pub prec: Option<Weak<RefCell<Node>>>,
    pub value: u64,
}
pub struct DoubleLinkedList {
    pub head: Rc<RefCell<Node>>,
    pub tail: Rc<RefCell<Node>>,
}
impl DoubleLinkedList {
    pub fn new(_capacity: usize) -> Self {…}
    pub fn add(&mut self, value: u64) {…}
    pub fn sum_all(&self) -> u64 {…}
}
```

Rust (main)

```rust
fn main() {
    let mut d = DoubleLinkedList::new(1_000_000);
    for i in 0..1_000_000 {
        d.add(i);
    }
    println!("OK");
}
```

Error (Runtime)

```
OK

thread 'main' has overflowed its stack
error: process didn't exit successfully:
`target\debug\double_linked_list_test-rs.exe` (exit code: 0xc00000fd,
STATUS_STACK_OVERFLOW)
```

Reference Count

When dealing with large lists and recursive ownership (either via a Rc<T> or Box<T> object) there is one other aspect we need to take into consideration. Let's take a close look Rc<T> implementation for a double linked list:

Rust (DoubleLinkedList structure)

```
pub struct Node {
    pub next: Option<Rc<RefCell<Node>>>,
    pub prec: Option<Weak<RefCell<Node>>>,
    pub value: u64,
}
pub struct DoubleLinkedList {
    pub head: Rc<RefCell<Node>>
}
    pub fn sum_all(&self) -> u64 {…}
}
```

Rust (main)

```
fn main() {
    let mut d = DoubleLinkedList::new(1_000_000);
    for i in 0..1_000_000 {
        d.add(i);
    }
    println!("OK");
}
```

Error (Runtime)

```
OK

thread 'main' has overflowed its stack
error: process didn't exit successfully:
`target\debug\double_linked_list_test-rs.exe` (exit code: 0xc00000fd,
STATUS_STACK_OVERFLOW
```

So … why the runtime error ?

It looks like the code runs without any problem, it prints OK to the screen, and when it finishes it crashes with a **stack overflow** error.

Reference Count

When an object of type Rc<T> or Box<T> is being dropped, Rust recursively drops any other objects that the current object owns.
For example, if we have a `let t = Box::new(MyString{…});` , where ***MyString*** is a structure that contains a String object, when t is being dropped, the String object contained in the ***MyString*** structure will be dropped as well.

Rust does this by calling the destructor for each data member of structure ***MyString***. While this is perfectly fine, having a list, a tree or graph where the ownership of one node expends to multiple children may create a stack overflow as when those children have to be dropped a recursive call to drop all of them will be called.

Reference Count

The solution for these cases is to implement a custom Drop that does the same thing but instead of doing it recursively, it does it iteratively.

Rust

```rust
pub struct Node {
    pub next: Option<Rc<RefCell<Node>>>,
    pub prec: Option<Weak<RefCell<Node>>>,
    pub value: u64,
}
pub struct DoubleLinkedList {
    pub head: Rc<RefCell<Node>>,
    pub tail: Rc<RefCell<Node>>,
}
impl DoubleLinkedList {
    pub fn new(_capacity: usize) -> Self {…}
    pub fn add(&mut self, value: u64) {…}
    pub fn sum_all(&self) -> u64 {…}
}
```

Rust (Custom drop)

```rust
impl Drop for Node {
    fn drop(self: &mut Node) {
        loop {
            let Some(next) = self.next.take() else {
                return;
            };
            let next = next.borrow_mut().next.take();
            self.next = next;
        }
    }
}
```

Notice the usage of method .take() from an ***Option***. This method return the value and replace it with a None. Since the value is then dropped, this technique drops all children starting with one Node iteratively.

Q & A