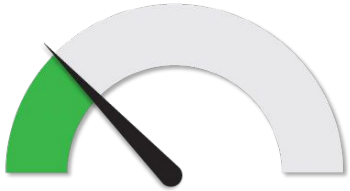
Cow in a Box & Friends

a tale of smart pointers

Rust Wrocław



mgr inż. Rafał Chabowski, 29.04.2020

About me

- Magister inżynier
- Software developer

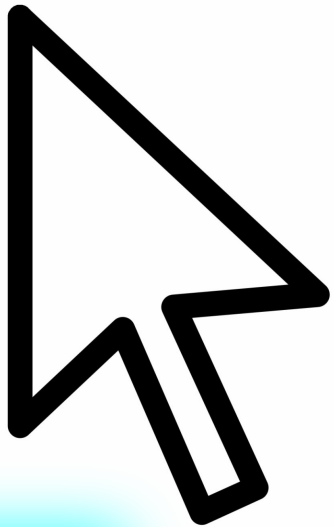
Agenda

- Pointer
- Smart Pointer
- Container
- Interior Mutability
- ~~Multithreading~~

Pointer

- Rust does have raw pointers...
  - both **const** and **mut**

```
fn main() {
    let x = 6;
    let px: *const i32 = &x;
    println!("{:?}", px);
}
```



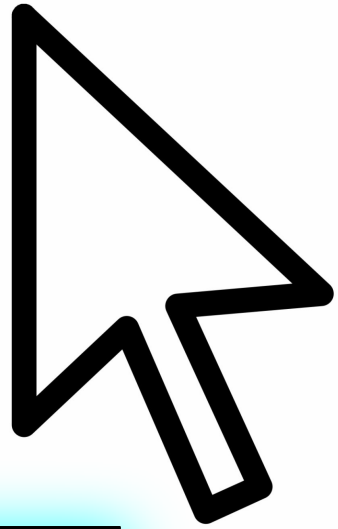
```
magister@inzynier:~/rust$ cargo run

0x7fff5277361c
```

Pointer

- ...but they cannot be **safely** dereferenced

```
fn main() {
    let x = 6;
    let px: *const i32 = &x;
    println!("{}", *px);
}
```



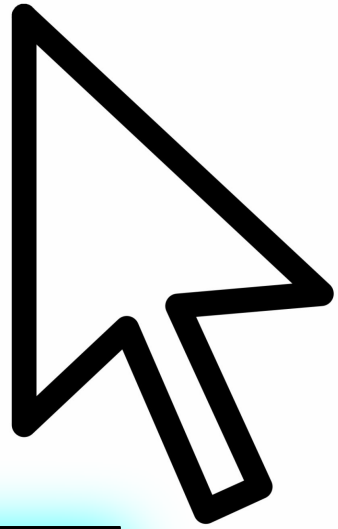
```
magister@inzynier:~/rust$ cargo run

|     println!("{}", *px);
|                    ^^^ dereference of raw pointer
```

Pointer

- **Unsafe** to the rescue!

```rust
let x = 6;
let px: *const i32 = &x;
unsafe {
    println!("{}", *px);
}
// -or just-
println!("{}", unsafe { *px });
```



```
magister@inzynier:~/rust$ cargo run

6
6
```

Smart Pointer

- It’s all about ownership
- Who owns a piece of data?
- Owner is obliged to release the memory:
  - safely
  - not prematurely
  - at all

Smart Pointers

- Box *(~T)*
- Rc *(@T)*
- Arc
- Weak
- Cow
- *...some more?*

Former form :)

Smart Pointers - Box

- Keeps exclusive ownership of the memory
- If you can't or doesn't want to keep a value on the stack
  - Its size is not know at compile time (like trait objects, for example)
  - It's too big
- Implements **Deref** trait
  - It shines in conjunction with **deref coercion**
- Mutations allowed

```rust
fn main() {
    let one = 1;
    let two = Box::new(2);

    let r_one = &one;
    println!("{} - {}", *r_one, *two);
}
```

```rust
let a = box Point
{           x: 10, y: 20
};
```

Former form :)

```
magister@inzynier:~/rust$ cargo run

1 - 2
```

Smart Pointers - Rc

- Shares the ownership of the memory
- Data must remain valid as long as there is at least one owner
- To be used in single-threaded cases only
  - The internal reference counter **is not** atomic
- Mutations **not** allowed (no single, clear owner)

```
let mut one = Box::new(1);
*one = 2;
println!("{}", *one);
```

```
~/rust$ cargo run

2
```

```
let mut one = Rc::new(1);
*one = 2;
println!("{}", *one);
```

```
~/rust$ cargo run
|     *one = 2;
|     ^^^^^^^^ cannot assign
```

Smart Pointers - Rc

```rust
struct Data {
    small: i32,
    huge: Rc<String>,
}

impl Data {
    pub fn new() -> Self { Data { small: 1,
            huge: Rc::new("Loooong".to_string(),),}}

    pub fn get_small(&self) -> i32 { self.small }
    pub fn get_huge(&self) -> Rc<String> {
          Rc::clone(&self.huge)}
}
```

```rust
fn main() {
    let db = Data::new();
    let (x, y, z) = (db.get_small(), db.get_small(),
                     db.get_small());
    println!("{} {} {}", x, y, z);
    let (x, y, z) = (db.get_huge(), db.get_huge(),
                     db.get_huge());
    println!("{} {} {}", x, y, z);
    println!("{} - {:?}", Rc::strong_count(&y),
                          Rc::ptr_eq(&x, &z));
    std::mem::drop(db);
    println!("{}", Rc::strong_count(&y));
}
```

```
magister@inzynier:~/rust$ cargo run
1 1 1
Loooong Loooong Loooong
4 - true
3
```

Smart Pointers - Rc

- Usage across threads

```
struct Data {}

fn main() {
    let data = Rc::new(Data {});
    thread::spawn(move || {
        let local = Rc::clone(&data);
    });
}
```

```
magister@inzynier:~/rust$ cargo build

|          `std::rc::Rc<Data>` cannot be sent between threads safely
```

Smart Pointers - Arc

- Usage across threads
- Now we're safe, but we need to explicitly indicate that we are ready to pay the cost of synchronization
- Contained **T** must be **Send** and **Sync**

```
struct Data {}

fn main() {
    let data = Arc::new(Data {});
    thread::spawn(move || {
        let local = Arc::clone(&data);
    });
}
```

```
magister@inzynier:~/rust$ cargo build

Finished dev [unoptimized + debuginfo] target(s) in 0.02s
```

Smart Pointers - Weak

- std::**rc**::Weak *and* std::**sync**::Weak
  - Result of downgrading **Rc** and **Arc** respectively
- They are smart in a way that they can “tell” if reference is still valid

```
struct Data {}
fn main() {
    let weak;
    {
        let data = Rc::new(Data {});
        weak = Rc::downgrade(&data);
    }
    let data = weak.upgrade();
    assert!(data.is_none());
}
```

```
struct Data {}
fn main() {
    let data = Rc::new(Data {});
    let weak;
    {
        weak = Rc::downgrade(&data);
    }
    let data = weak.upgrade();
    assert!(data.is_some());
}
```

Smart Pointers - Cow

- Clone-on-Write
- Provides immutable access to data
- Create clone of referenced data when **mutability** or **ownership** is required
- Imagine accessing a text file from the read-only network share
  - Anyone can seamlessly open it and read the text
  - But if someone wishes to make a correction, he needs to create a local copy
- The same principles apply to data stored in **Cow**
  - There's just a single instance of the (potentially huge) data
  - Until someone wants to mutate it (take ownership)
    - Then the clone is made

Smart Pointers - Cow

```rust
fn main() {
    let data = HugeData { number: 1 };

    let mut ptr1 = Cow::Borrowed(&data);
    println!("{}", ptr1.number);

    let mut ptr2 = ptr1.to_mut();
    ptr2.number = 2;
    println!("{}", ptr2.number);
}
```

```rust
impl Clone for HugeData {
    fn clone(&self) -> Self {
        println!("Well, here's your copy!");
        HugeData {
            number: self.number,
        }
    }
}
```

```
magister@inzynier:~/rust$ cargo run
1
Well, here's your copy!
2
```

Smart Pointers - Cow

```
fn foo() -> &'static str {
    let error_code = 17;
    match error_code {
        1 => "Not enough memory".into(),
        2 => "Too much memory".into(),
        _ => "Unknown".into(),
    }
}
```

- Returning static “strings” to avoid allocations
- Works great!

Smart Pointers - Cow

```
fn foo() -> &'static str {
    let error_code = 17;
    match error_code {
        1 => "Not enough memory".into(),
        2 => "Too much memory".into(),
        _ => format!("Unknown:
{}",                error_code).into(),
    }
}
```

- Works as long as we don't need to pass any additional details

```
magister@inzynier:~/rust$ cargo build
8 |         _ => format!("Unknown: {}", error_code).into(),
  |                                                 ^^^^ the trait
`std::convert::From<std::string::String>` is not implemented for `&str`
```

Smart Pointers - Cow

```
fn foo() -> String {
    let error_code = 17;
    match error_code {
        1 => "Not enough memory".into(),
        2 => "Too much memory".into(),
        _ => format!("Unknown:
{}",                error_code).into(),
    }
}
```

- So, just return an owned string
- Works, but not optimal performance-wise

```
magister@inzynier:~/rust$ cargo build
Finished dev [unoptimized + debuginfo] target(s) in 0.01s
```

Smart Pointers - Cow

```
fn foo() -> Cow<'static, str> {
    let error_code = 17;
    match error_code {
        1 => "Not enough memory".into(),
        2 => "Too much memory".into(),
        _ => format!("Unknown:
{}",                error_code).into(),
    }
}
```

- Now, owned string is returned only when really needed

```
magister@inzynier:~/rust$ cargo build
Finished dev [unoptimized + debuginfo] target(s) in 0.01s
```

Smart Pointers - Cow

```rust
impl<'a> From<&'a str> for Cow<'a, str> {
    #[inline]
    fn from(s: &'a str) -> Cow<'a, str> {
        Cow::Borrowed(s)
    }
}
```

```rust
impl<'a> From<String> for Cow<'a, str> {
    #[inline]
    fn from(s: String) -> Cow<'a, str> {
        Cow::Owned(s)
    }
}
```

- Cow will automatically pass on borrowed reference, until a Clone is needed
- Drop-in replacement in cases like that

Smart Pointers - Deref Coercion

- “*This rule is one of the only places in which Rust does an automatic conversion for you*” (for convenience)

If type **U** can be dereferenced to other type **T** then **&U** will automatically act as **&T**	If **Box<i32>** implements **Deref<Target=i32>** then **&Box** can be used in every place when **&i32** is expected. **&&&&&&&&Box** fits as well :-)

Smart Pointers - Deref Coercion

```rust
fn takes_ref(r: &i32) {
    println!("{}", r);
}

fn main() {
    let a = 7;
    takes_ref(&a);

    let b = Box::new(7);
    takes_ref(&b);
    takes_ref(&&&&&&&&&&&&&&&&b);

    takes_ref(&(*b)); // Manual deref
}
```

```
magister@inzynier:~/rust$ cargo run
7
7
7
7
```

Smart Pointers - Summary

- Choose your guarantees

	Box	Rc	Arc	Cow
Ownership	Single	Shared	Shared	Mixed
Mutability	✔	✘	✘	
Thread-safety	✔	✘	✔	

Inherited Mutability

- By design, Rust exhibits the **inherited** mutability
- Mutability is defined at binding-level (all or nothing approach)

```
struct S {
    a: i32,
    b: i32,
}
fn main() {
    let     s = S { a: 1, b: 2 };
    let mut ms = S { a: 1, b: 2 };
}
```

```
struct S {
    a: mut i32,
    b: const i32,
}
```

❌

Interior Mutability

- It might be required to mutate some members (mind the **mutable** keyword from C++)
- API should remain non-mutable
  - Do not force user to create mutable bindings just for the sake of some internal data manipulation
- Examples
  - Caching (Memoization)
  - Lazy-evaluation
  - Collecting performance statistics
  - Debugging
  - Increasing/decreasing reference counter in **Rc**

Interior Mutability - Toolbox

- UnsafeCell
  - The only core language feature to work around the mutability restrictions
  - Rust uses this basic tool as a foundation for more sturdy wrappers
  - A tool not to be used directly (!)
- Cell
  - A mutable memory location
  - Provides functionality corresponding to **mutable** class members in C++
  - Owns the memory location
- RefCell
  - Similar to Cell, but works on borrowed data
  - Provides access to **&mut** through **&self**
  - Never Sync

Interior Mutability - Cell

```rust
use std::cell::Cell;

struct S {
    a: i32,
    b: Cell<i32>,
}
fn main() {
    let s = S {
        a: 1,
        b: Cell::new(2),
    };
    s.b.set(3);
    println!("{}", s.b.get());
}
```

```
magister@inzynier:~/rust$ cargo run
3
```

Interior Mutability - Cell

- There is no borrow checking!

```
fn calculate_bonuses(hr_data: &Cell<i32>) {
    let salary = hr_data.get();
    hr_data.set(salary + 700);
}

fn main() {
    let hr_database = Cell::new(15_000);
    let salary = hr_database.get();
    calculate_bonuses(&hr_database);
    hr_database.set(salary);
    println!("{}", hr_database.get());
}
```

Prefer regular references or RefCell, so that Rust can protect you against such bugs.

```
~/rust$ cargo run
15000
```

Interior Mutability - RefCell

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

struct S {
    b: RefCell<i32>,
}

fn main() {
    let s = S { b: RefCell::new(1) };
    let mut b = s.b.borrow_mut();
    *b = 2;
    println!("{}", b);
}
```

```
magister@inzynier:~/rust$ cargo run
2
```

“Fixing” the issues with borrow checker :)

```rust
fn main() {
    let mut data = 23;
    let r = &data;
    println!("{}", r);
    if get_user_input() > 10 {
        let mr = &mut data;
        *mr = 24;
        println!("{}", mr);
    }

}
```

```rust
fn main() {
    let mut data = 23;
    let r = &data;

    if get_user_input() > 10 {
        let mr = &mut data;
        *mr = 24;
        println!("{}", mr);
    }
    println!("{}", r);
}
```

❌

“Fixing” the issues with borrow checker :)

```
error[E0502]: cannot borrow `data` as mutable because it is also borrowed as immutable
 --> src/main.rs:9:18
  |
7 |     let r = &data;
  |             ----- immutable borrow occurs here
8 |     if get_user_input() > 10 {
9 |         let mr = &mut data;
  |                  ^^^^^^^^^ mutable borrow occurs here
...
13 |     println!("{}", r);
  |                     - immutable borrow later used here

error: aborting due to previous error
```

The “unsafe” hammer

```
fn main() {
    unsafe {
        let mut data = 23;
        let r = &data;
        if get_user_input() > 10 {
            let mr = &mut data;
            *mr = 24;
            println!("{}", mr);
        }
        println!("{}", r);
    }
}
```

The “RefCell” hammer

```
fn main() {
    let data = RefCell::new(23);
    let r = data.borrow();
    if get_user_input() > 10 {
        let mut mr = data.borrow_mut();
        *mr = 24;
        println!("{}", mr);
    }
    println!("{}", r);
}
```

Fixed ✔

```
magister@inzynier:~/rust$ cargo run
23
```

The “RefCell” hammer

- It’s fixed, until the user input remains 10 or less...
- Otherwise...

```
~/rust$ cargo run
     Running `target/debug/rust`
thread 'main' panicked at 'already borrowed: BorrowMutError',
/rustc/b8cedc00407a4c56a3bda1ed605c6fc166655447/src/libcore/cell.rs:878:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

- **try_borrow()** family of functions can help

The “Rc<RefCell>” combo

- Rc - shared ownership
- RefCell - mutability...
- 🤔
- Not suitable in multi-threaded environment
  - **Arc**, **RwLock** or **Mutex**

```rust
fn main() {
    let data = Rc::new(RefCell::new(23));
    let r = data.clone();
    if get_user_input() > 10 {
        let mr = data.clone();
        let mut internal = mr.borrow_mut();
        *internal = 24;
    }
    println!("{}", *r.borrow());
}
```

```
magister@inzynier:~/rust$ cargo run
24
```

Summary

- Prefer inherited mutability (“cellless” code)
- Checklist before using interior mutability
  - Make sure you understand what you’re doing
    - **panic!** in runtime, even though safe, is still worse than compilation error
  - Rethink you approach, take 1, 2, … 10 steps back

THE END