Smart Pointers

Smart Pointers are references to data, so you can use it, without taking ownership of this data. But before talking about Smart Pointers, lets take a look at normal references.

Normal references

You can create references to variables. This gives code the ability to work with those variables, without taking the ownership.

Reference

TODO

Mutable Reference

TODO

Smart Pointers

Smart Pointers are often required, because they give you additional functionalities, where normal references do not fit the problem. They behave like a pointer, but add safety mechanisms to prevent wrong memory management.

Box

Variables are usually stored in the stack. But sometimes you need to allocate data on the heap memory. A good example is, when the size of the data struct can not be calculated on compile time. The Box smart pointer is basically a pointer to the struct on the heap.

```
// 5 is allocated in the heap,
// and "Pointer" is a smart pointer
// pointing on "5"
let pointer = Box::new(5);

// To use the value, dereference the pointer
println!("value = {}", *pointer);

// Implicit derefrencing may also work
// because Box implements the Deref trait
```

```
println!("value = {}", pointer);
```

Rc

The ownership of data is always bould to one place. One variable has the responsibility to clean up the memory used when done. but at some points mutiple places need to have ownership of those data.

Rc is a smart pointer with a **reference counter**, which allows to share ownership on multiple places. Each time another place needs the memory, the Rc pointer is copied and increases the reference counter. And every time the life time of a Rc ends, it decreases the reference counter. Eventually the counter becomes zero and the stored data will be decallocated.

Rc is used to share ownership in **single threaded** code and provides **read only access** to the data.

```
// create a Rc pointer
let magic_number = Rc::new(1337);

{
  // clone the pointer to create multiple references
  // and allow it to be used somewhere
  let magic_clone = Rc::clone(&magic_number);
  // Now the counter is at 2
}
// The lifetime of magic_clone ends and the
// counter is decreased to 1 again

println!("magic number = {}", *magic_number);
```

Arc

TODO

RefCell

TODO

Cell

TODO

Mutex

TODO

RwLock

TODO

Cow

A Cow pointer (Copy-on-read) points to borrowed read only data. But it allows to create a mutable, by cloning those data and creating a mutable pointer.

```
# Example from Rust docs

use std::borrow::Cow;

fn abs_all(input: &mut Cow<'_, [i32]>) {
    for i in 0..input.len() {
        let v = input[i];
        if v < 0 {
            // Clones into a vector if not already owned.
            input.to_mut()[i] = -v;
        }
    }
}

// No clone occurs because `input` doesn't need to be mutated.
let slice = [0, 1, 2];
let mut input = Cow::from(&slice[..]);
abs_all(&mut input);

// Clone occurs because `input` needs to be mutated.
let slice = [-1, 0, 1];
let mut input = Cow::from(&slice[..]);
abs_all(&mut input);

// No clone occurs because `input` is already owned.
let mut input = Cow::from(vec![-1, 0, 1]);
abs_all(&mut input);
```

Here, `input` initially points to `slice`, hence doesn't require as much space as a copy. For read-only operations it is efficient. But when `slice` contains a negative value which needs to be modified, `Cow` creates a copy when calling `to_mut`, which then can be safely modified.

This kind of pointer is useful, when you have larger data, which you want to use without modifying it, but you are **not sure whether you need to modify it or not**. When you don't need to modify the data, it saves space by borrowing the data instead.

Weak

TODO

Pin

TODO

Revision #4
Created 2026-02-07 22:25:13 UTC by Matthias
Updated 2026-03-09 15:03:58 UTC by Matthias