Advanced Programming Book

By Davide Andreolli

Gimme Double (2022/11 - 2023/01)

> Define the `Doublable` trait with a method `gimme_double` implement `Doublable` for `i32`, `gimme_double` returns a new `i32` that is twice self implement `Doublable` for `String`, `gimme_double` returns a new `String` that is self concatenated with self implement a function `printdouble` that takes a `Doublable` and prints the argument and its `gimme_double` using the `":?"` formatter it behaves as the example: *doubling 5 is 10 doubling "what" is "whatwhat"*.

```rust
use std::fmt::Debug;

trait Doublable {
    fn gimme_double(&self) -> Self;
}

impl Doublable for i32 {
fn gimme_double(&self) -> i32 {
        self * 2
    }
}

impl Doublable for String {
    fn gimme_double(&self) -> String {
        format!("{}{}", self, self)
    }
}

fn print_double<T: Doublable + Debug>(x: T) {
    println!("doubling {:?} is {:?}", x, x.gimme_double());
}
```

Gimme Next (2024/01)

> Define the `Nextable` trait with a method `gimme_next` implement `Nextable` for `i32`, `gimme_next` returns the optional successor of self implement `Nextable` for `char`, `gimme_next` returns the optional new `char` that is the next `char` (as a `u32` conversion) implement a function `printnext` that takes a `Nextable` and prints the argument and its `gimme_next` using the `":?"` formatter It behaves as the example: *next of 5 is Some(6) next of 's' is Some('t')*.

```rust
use std::fmt::Debug;

trait Nextable {
    fn gimme_next(&self) -> Option<Self>
    where Self: Sized;
}

impl Nextable for i32 {
    fn gimme_next(&self) -> Option<Self> {
        Some(*self + 1)
    }
}

impl Nextable for char {
    fn gimme_next(&self) -> Option<Self> {
        if *self == 'z' {
            None
```

```
        } else {
            Some((*self as u8 + 1) as char)
        }
    }
}

fn printnext(nextable: &(impl Nextable + Debug)) {
    println!("next of {:?} is {:?}", nextable, nextable.gimme_next())
}
```

Toggle (2024/01)

Implement a trait `Toggle` with a function `toggle(&mut self)`. Implement the Trait for:

- `bool`: were toggling (where false become true, and true become false
- `i32` where the number get negated (e.g. 11 -> -11; -44 -> 44)
- `String` where all letters gets their case inverted (e.g "Hello World!" -> "hELLO wORLD!". you can

Assume that the string contains only ascii characters.

Write a generic function `toggle_and_print` that take as input an immutable reference of an item, and prints (with new line at the end) with the debug formatter *"... toggled is ..."*

```
use std::fmt::Debug;

trait Toggle {
    fn toggle(&mut self);
}

impl Toggle for bool {
    fn toggle(&mut self) {
        *self = !*self;
    }
}

impl Toggle for i32 {
    fn toggle(&mut self) {
        *self = -*self;
    }
}

impl Toggle for String {
    fn toggle(&mut self) {
        *self = self.chars()
            .map(|c| {
                if c.is_ascii_uppercase() {
                    c.to_ascii_lowercase()
                }
                else if c.is_ascii_lowercase() {
                    c.to_ascii_uppercase()
                }
                else {
                    c
                }
            })
            .collect::<String>();
    }
}

fn toggle_and_print<T: Toggle + Debug + Clone>(value: &T) {
```

```
    let mut cloned = value.clone();
    cloned.toggle();
    println!("{:?} toggled is {:?}", value, cloned);
}
```

Wrapper for i32 odds (2022/11 - 2024/01)

Define a struct `Wrapper` that contains a field `v` of type `Vec<i32>` define an iterator for `Wrapper` to cycle over the elements of the vector the iterator will skip every other element, effectively accessing only those at odd index in the inner vector (the first element is at index 0)

```
#[derive(Clone)]
struct Wrapper {
    v: Vec<i32>
}

impl Iterator for Wrapper {

    type Item = i32;

    fn next(&mut self) -> Option<Self::Item> {
        if let Some(number) = self.v.clone().get(1) {
            self.v = self.v[2..].to_vec();
            Some(*number)
        }
        else {
            None
        }
    }
}

impl Wrapper {
    fn iter(&self) -> impl Iterator<Item = i32> {
        self.clone()
    }
}
```

Wrapper for string lengths (2023/01)

Define a struct `Wrapper` that contains a field `v` of type `Vec<String>` define an iterator for `Wrapper` to cycle over the elements of the vector instead of returning a pointer to the elements of `v`, the iterator returns a the length of the elements of `v`.

```
struct Wrapper {
    v: Vec<String>,
}

impl Iterator for Wrapper {
    type Item = usize;

    fn next(&mut self) -> Option<Self::Item> {
        if let Some(s) = self.v.clone().get(0) {
            self.v.remove(0);
            Some(s.len())
        }
        else {
            None
        }
    }
```

```rust
}

impl Wrapper {
    fn iter(&self) -> Iterator<Item = usize> {
        self.clone()
    }
}
```

Wrapper ConsIter (2024/01)

Write a struct `ConsIter` that has a field `iter` of type `Chars (std::str::Chars)`. Write a struct `Wrapper` that has a field `inner` of type `String`, write a method `iter` for `Wrapper` that returns a `ConsIter`. Implement `Iterator` for `ConsIter` that iterates over chars, it yields all the characters that are part of the ascii code and aren't vocals ("aeiou").

Hints: use is_ascii() to check if a char is actually ascii, use to_ascii_lowercase() for managing mixed-cased words.

```rust
use std::str::Chars;

struct ConsIter<'a> {
    iter: Chars<'a>,
}

struct Wrapper {
    inner: String,
}

impl Wrapper {
    fn iter(&self) -> ConsIter {
        ConsIter {
            iter: self.inner.chars(),
        }
    }
}

impl<'a> Iterator for ConsIter<'a> {
    type Item = char;

    fn next(&mut self) -> Option<Self::Item> {
        let mut result = None;
        while let Some(c) = self.iter.next() {
            if c.is_ascii() && !['a', 'e', 'i', 'o', 'u'].contains(&c.to_ascii_lowercase()) {
                result = Some(c);
                break;
            }
        }
        result
    }
}
```

BasicBox Sum (2022/11 - 2024/01)

Write a function `basicbox_sum` that takes a vector of Strings and returns a vector of Boxes of usizes the returned vector contains all the lengths of the input vector followed by a final element that sums all the previous lengths

```rust
fn basicbox_sum(v: Vec<String>) -> Vec<Box<usize>> {
    let mut result = v
```

```rust
        .iter()
        .map(|s| s.len())
        .map(|i| Box::new(i))
        .collect::<Vec<Box<usize>>>();

    result.push(Box::new(
        v.iter().map(|s| s.len()).reduce(|l, r| l + r).unwrap_or(0),
    ));

    result
}
```

BasicBox Inc (2023/01)

Write a function `basicbox_inc` that takes a vector of `Strings` and returns a vector of `Box` of `usize` the returned vector contains all the lengths of the input vector + 1

```rust
fn basicbox_sum(v: Vec<String>) -> Vec<Box<usize>> {
    v.iter()
        .map(|s| s.len() + 1)
        .map(|i| Box::new(i))
        .collect::<Vec<Box<usize>>>()
}
```

List with boxes (2022/11 - 2024/01)

Take the following `List` and `Node` structs define these functions and methods for `List`, each one defines how many points it yields

- [7] remove: takes a position `p:i32` where to remove the element from the list and it returns a `Result<(),String>` The function removes the node at position `p` or returns the string `"wrong position"` if the list has fewer than `p` elements. That is: removing from position 2 in `[10,20,30]` will return `[10,20]`. Removing from position 3 in `[10,20,30]` will return `Err("wrong position)` removing from position 0 in `[10,20,30]` will return `[20,30]`.
- [2] pop: removes the head of the list
- [2] pop_last: removes the last element of the list
- [4] get: takes a position `p` and returns an optional pointer to the pth T-typed element in the list (That is, a pointer to the element, not a pointer to the `Node`)

Note: the tests already include the code below, all you need to paste as the answer are the impl blocks and possible imports (use ...).

```rust
// Given code
#[derive(Debug)]
pub struct List<T> {
    head: Link<T>,
    len: i32,
}

type Link<T> = Option<Box<Node<T>>>;

#[derive(Debug)]
struct Node<T> {
    elem: T,
    next: Link<T>,
}

#[derive(Debug)]
pub struct Content {
    s: String,
```

```rust
    b: bool,
    i: i32,
}

impl Content {
    pub fn new_with(s: String, b: bool, i: i32) -> Content {
        return Content { s, b, i };
    }
}


// Exercise code
impl<T> List<T> {

    fn remove(&mut self, mut pos: i32) -> Result<(), String> {

        if pos < 0 || pos >= self.len {
            return Err("wrong position".to_string());
        }

        let mut current = &mut self.head;

        while pos > 0 && current.is_some() {
            current = &mut current.as_mut().unwrap().next;
            pos -= 1;
        }

        let next = current.as_mut().unwrap().next.take();
        *current = next;
        self.len -= 1;

        Ok(())
    }

    fn pop(&mut self) -> Result<(), String> {
        self.remove(0)
    }

    fn pop_last(&mut self) -> Result<(), String> {
        self.remove(self.len - 1)
    }

    fn get(&self, mut pos: i32) -> Option<&T> {
        if pos < 0 || pos >= self.len {
            return None;
        }

        let mut current = &self.head;

        while pos > 0 {
            current = &current.as_ref().unwrap().next;
            pos -= 1;
        }

        Some(&current.as_ref().unwrap().elem)
    }
}
```

Clock (2024/01)

Write a two structs: `MasterClock` and `SlaveClock` that both derive `Debug`. `MasterClock` keeps track of a number of clock cycle (in `usize`). The struct has:

- [1] a `new()` method that initialize it with clock at zero.
- [1] a `tick(&mut self)` method that increase the clock cycle by 1.
- [2] a `get_slave(&self)` method that return an object of type `SlaveClock` `SlaveClock` can be built only using the `MasterClock::get_slave(&self)` method, and has a method named [2] `get_clock(&self)` that returns the current clock (that automatically sinks with the master clock).

```rust
use std::{rc::Rc, cell::RefCell};

#[derive(Debug)]
struct MasterClock {
    count: Rc<RefCell<usize>>
}

#[derive(Debug)]
struct SlaveClock {
    count: Rc<RefCell<usize>>
}

impl MasterClock {
    fn new() -> Self {
        Self {
            count: Rc::new(RefCell::new(0))
        }
    }

    fn tick(&self) -> () {
        *self.count.borrow_mut() += 1;
    }

    fn get_slave(&self) -> SlaveClock {
        SlaveClock {
            count: self.count.clone()
        }
    }
}

impl SlaveClock {
    fn get_clock(&self) -> usize {
        *self.count.borrow()
    }
}
```

Sorted list (2023/01)

Take the following `List` and `Node` structs define these functions and methods for `List`, each one defines how many points it yields

- [1] `new`: returns an empty list
- [6] `add`: takes an element `e:T`. The function inserts the element e while keeping the list sorted. That is: adding 3 to list [] returns [3] adding 3 to list [0,4] returns [0,3,4] adding 3 to list [0,1] returns [0,1,3]
- [4] `get`: takes a position p and returns an optional pointer to the pth T-typed element in the list (That is, a pointer to the element, not a pointer to the Node)

The list must work on Content, add the code that allows this ([4] points). The comparison between different Content structs only compares their i field That is, {"what",false,2} < {"super",true,5} < {"",false,10}

Note: the tests already include the code below, all you need to paste as the answer are the impl blocks and possible imports (use ...).

```rust
// Given code
#[derive(Debug)]
pub struct List<T> {
    head: Link<T>,
    len: i32,
}

type Link<T> = Option<Box<Node<T>>>;

#[derive(Debug)]
struct Node<T> {
    elem: T,
    next: Link<T>,
}

#[derive(Debug)]
pub struct Content {
    s : String, b : bool, i : i32,
}

impl Content {
    pub fn new_with(s:String, b:bool, i:i32) -> Content {
        return Content{s,b,i};
    }
}

// Execise code
impl<T> List<T> {
    fn new() -> Self {
        List { head: None, len: 0 }
    }

    fn get(&self, mut pos: i32) -> Option<&T> {
        let mut head = self.head.as_ref();
        while pos > 0 && head.is_some() {
            head = head.unwrap().next.as_ref();
            pos -= 1;
        }
        if let Some(head) = head {
            Some(&head.elem)
        }
        else {
            None
        }
    }
}

impl<T: PartialOrd> List<T> {

    fn add(&mut self, elem: T) {

        let mut current = &mut self.head;

        while current.as_ref().map_or(false, |node| elem > node.elem) {
            current = &mut current.as_mut().unwrap().next;
```

```rust
        }

        let new_node = Box::new(Node {
            elem,
            next: current.take(),
        });

        *current = Some(new_node);
    }
}

impl PartialEq for Content {
    fn eq(&self, other: &Self) -> bool {
        self.i == other.i
    }
}

impl PartialOrd for Content {
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        self.i.partial_cmp(&other.i)
    }
}
```

Shared Communication (2024/01)

Create a struct `SharedCommunications` that derives Debug with the following methods: -[1] `new()->Self`: create a new communication object connected to no one, with no message inside. -[1] `new_form(other: &Self)->Self`: create a new communication object connected to other. -[2] `send(&mut self, message: String)->Result<(),()>`: try to send a message... if the structure already has a message inside, it returns an error. otherwise it memorize the message and return Ok. -[2] `receive(&mut self)->Option<String>`: if the structure has a message inside it returns it. otherwise returns None.

The struct implement a kind of blocking pipe, where message can be sent only if the previous message has been received. The object must be sharable between multiple owners using the `new_form` method.

```rust
use std::{cell::RefCell, rc::Rc};

struct SharedCommunication {
    message: Rc<RefCell<Option<String>>>,
}

impl SharedCommunication {
    fn new() -> Self {
        Self {
            message: Rc::new(RefCell::new(None)),
        }
    }

    fn new_form(other: &Self) -> Self {
        Self {
            message: other.message.clone(),
        }
    }

    fn send(&mut self, message: String) -> Result<(), ()> {
        if self.message.borrow().is_none() {
            *self.message.borrow_mut() = Some(message);
            Ok(())
        } else {
```

```
            Err(())
        }
    }

    fn receive(&mut self) -> Option<String> {
        self.message.borrow_mut().take()
    }
}
```

Graph and SameBool (2022/11 - 2023/01)

`SameBool` is a Trait. It has a method `samebool` that takes a `SameBool` and it returns a `bool`. `Content` is a struct with an `i32` and a `bool`. Two Contents can be compared (<, >, ==) by comparing their `i32` field ([2 points]). `Content` implements `SameBool`: the method of the trait returns whether self has the same bool as the parameter ([1] point). Define a `Graph` as a vector of `Node` whose elements are arbitrary `T` - add a function for creating an empty graph ([1] points). When `T` implements `SameBool` and `PartialOrd`, define function `add_node` that adds a `Node` to the graph with these connections:

- the added node gets as neighbour all nodes in the graph that are < than it
- the added node becomes a neighbour of all the nodes with the samebool ([6] points).

Note: the tests already include the code below, all you need to paste as the answer are the impl blocks and possible imports (use ...).

```
// Given code
type NodeRef<T> = Rc<RefCell<Node<T>>>;

struct Node<T> {
    inner_value: T,
    adjacent: Vec<NodeRef<T>>,
}

impl<T: Debug> Debug for Node<T>{
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        write!(f,"iv: {:?}, adj: {}", self.inner_value, self.adjacent.len())
    }
}

struct Graph<T> {
    nodes: Vec<NodeRef<T>>,
}

pub trait SameBool{
    fn samebool(&self, other:&Self)->bool;
}

#[derive(Debug)]
pub struct Content{
    pub i:i32,
    pub b:bool
}

// Exercise code
impl Content {
    pub fn new_with(i: i32, b: bool) -> Content {
        Content { i, b }
    }
}
```

```rust
impl SameBool for Content{
    fn samebool(&self, other: &Self) -> bool {
        self.b == other.b
    }
}

impl<T> Graph<T> {
    fn new() -> Self {
        Graph { nodes: Vec::new() }
    }
}

impl<T: SameBool + PartialOrd> Graph<T> {

    fn add_node(&mut self, value: T) -> () {
        let mut new_node = Node {
            inner_value: value,
            adjacent: Vec::new(),
        };

        self.nodes
            .iter()
            .filter(|n| n.borrow().inner_value < new_node.inner_value)
            .for_each(|n| new_node.adjacent.push(Rc::clone(n)));

        let new_node = Rc::new(RefCell::new(new_node));

        self.nodes
            .iter()
            .filter(|n| new_node.borrow().inner_value.samebool(&n.borrow().inner_value))
            .for_each(|n| n.borrow_mut().adjacent.push(Rc::clone(&new_node)));

        self.nodes.push(new_node);
    }
}

impl PartialEq for Content {
    fn eq(&self, other: &Self) -> bool {
        self.i == other.i
    }
}

impl PartialOrd for Content {
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        self.i.partial_cmp(&other.i)
    }
}
```

Tree PartialOrd (2023/01 - 2024/01)

Take the following `Tree`, `Node`, and `Content` structs define these functions/methods for `Tree`: `new` [1] : creates an empty tree. `add_node` [6]: takes a generic element `el` and adds a node to the tree whose content is `el` and such that nodes on the left have contents which are < smaller than the current node, nodes on the center have contents which are == to the current node, nodes on the right have contents which are > than the current node. `howmany_smaller` [4] : takes a generic element `el` and returns an `i32` telling how many nodes does the tree have that are < than `el`.

Implement `PartialOrd` for `Content` [4]: contents can be compared by comparing the len of their `String` fields.

Note: the tests already include the code below, all you need to paste as the answer are the impl blocks and possible imports (use ...).

```rust
use std::{cmp::Ordering, collections::VecDeque};

// Given code
#[derive(Debug)]
pub struct Content{
    pub i:i32,
    pub s:String
}

impl Content {
    pub fn new(i: i32, s: String) -> Content {
        Content { i, s }
    }
}

#[derive(Debug)]
struct Node<T> {
    elem: T,
    left: TreeLink<T>,
    center: TreeLink<T>,
    right: TreeLink<T>,
}

impl<T> Node<T> {
    pub fn new(elem:T) -> Node<T> {
        Node {
            elem,
            left:None,
            center:None,
            right:None
        }
    }
}

#[derive(Debug)]
pub struct Tree<T> {
    root: TreeLink<T>,
    size : i32,
}

type TreeLink<T> = Option<Box<Node<T>>>;

// Exercise code
impl<T> Tree<T> {

    fn new() -> Self {
        Self {
            root: None,
            size: 0
        }
    }
}

impl<T: PartialOrd> Tree<T> {
```

```
    fn add_node(&mut self, el: T) -> () {
        let mut current = &mut self.root;
        while let Some(node) = current {
            current = match el.partial_cmp(&node.elem).unwrap() {
                Ordering::Less => &mut node.left,
                Ordering::Equal => &mut node.center,
                Ordering::Greater => &mut node.right
            }
        }
        *current = Some(Box::new(Node::new(el)));
        self.size += 1;
    }

    fn howmany_smaller(&self, el: &T) -> i32 {
        if let Some(root) = self.root.as_ref() {
            let mut count = 0;
            let mut queue = VecDeque::new();
            queue.push_back(root.as_ref());

            while let Some(current) = queue.pop_front() {
                if &current.elem < el {
                    count += 1;
                }
                if let Some(node) = current.left.as_ref() {
                    queue.push_back(node);
                }
                if let Some(node) = current.center.as_ref() {
                    queue.push_back(node);
                }
                if let Some(node) = current.right.as_ref() {
                    queue.push_back(node);
                }
            }

            count
        }
        else {
            0
        }
    }
}

impl PartialEq for Content {
    fn eq(&self, other: &Self) -> bool {
        self.s.len() == other.s.len()
    }
}

impl PartialOrd for Content {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        self.s.len().partial_cmp(&other.s.len())
    }
}
```

Point (2024/01)

Create a module named `point_2d` with inside a struct named `Point` with two attributes: `x` and `y`, both f32 and public.

[1] Create a module named `point_3d` with inside a a struct named `Point`. with two attributes.

- `x_y` with type `Point` of the module `point_2d`
- `z` with type `f32`

Derive debug on both `Point`. Write module named `util`. Inside the module util, use the module system to rename point_2d's `Point` to Point2D and point_3d's `Point` to Point3D. be sure to make this aliases public. [2] Inside the module util write a public function named `_3d_to_2d`. The function takes the ownership of a Point3D and returns a point2D by removing the `z` component.

```
mod point_2d {

    #[derive(Debug, PartialEq)]
    pub struct Point {
        pub x: f32,
        pub y: f32,
    }
}

mod point_3d {
    use super::point_2d::Point as Point2d;

    #[derive(Debug)]
    pub struct Point {
        pub x_y: Point2d,
        pub z: f32,
    }
}

mod util {
    pub use super::point_2d::Point as Point2d;
    pub use super::point_3d::Point as Point3d;

    pub fn _3d_to_2d(point: Point3d) -> Point2d {
        Point2d {
            x: point.x_y.x,
            y: point.x_y.y,
        }
    }
}
```

Finance (2024/01)

Define a module `finance`. Inside it, define two public modules `wallet_1` and `wallet_2`.

- [1] Define a struct `Wallet` inside `wallet_1` with an attribute `euro` with type `f32`.
- [1] Define a struct `Wallet` inside `wallet_2` with an attribute `euro` with type `u32`, and an attribute `cents` with type `u8`

Derive Debug on both `Wallet`, and make all attributes public. Create two public alias inside `finance`:

- `Wallet1` for `wallet_1::Wallet`
- `Wallet2` for `wallet_2::Wallet`

[2] Define a public function `compare_wallet` in the module `finance` that takes two arguments: `first` with type `&Wallet1` and `second` with type `&Wallet2` the function returns true if `first` has more money that `second`, otherwise it returns false

```
mod finance {
    pub mod wallet_1 {

        #[derive(Debug)]
```

```rust
        pub struct Wallet {
            pub euro: f32
        }
    }

    pub mod wallet_2 {

        #[derive(Debug)]
        pub struct Wallet {
            pub euro: u32,
            pub cents: u8
        }
    }

    pub type Wallet1 = wallet_1::Wallet;
    pub type Wallet2 = wallet_2::Wallet;

    pub fn compare_wallet(first: &Wallet1, second: &Wallet2) -> bool {
        first.euro > (second.euro as f32) + (second.cents as f32) / 100.0
    }
}
```