Inteligentne wskaźniki i pamięć

21 listopada 2025

Instrukcja

Zadania ćwiczą użycie `Drop`, `Deref/DerefMut`, `Rc`, `Cell`, `RefCell`, `Weak`, `LazyCell`, `OnceCell` oraz `Cow`. Zadanie zaimplementuj jako aplikację, nie bibliotekę. Użyj wszystkich zdefiniowanych obiektów na przykładowe sposoby w main. Dopuszczalne jest użycie tylko biblioteki standardowej.

1. Zaimplementuj wszystkie elementy z sekcji *Funkcjonalność*.
2. Napraw wszystkie ostrzeżenia kompilatora.
3. Sprawdź, czy program przechodzi testy (`cargo test`).
4. Użyj `cargo clippy`, aby znaleźć typowe problemy i je poprawić.

Funkcjonalność

1. Zdefiniuj strukturę `AustroHungarianGreeter`.
   Zaimplementuj metodę `fn greet(&self) -> &'static str`, która zwraca kolejno, cyklicznie:

   - `"Es lebe der Kaiser!"`
   - `"Möge uns der Kaiser schützen!"`
   - `"Éljen Ferenc József császár!"`

   Kolejne wywołania mają rotować po tej trójce napisów. Użyj `Cell<T>`.

2. Dla typu z poprzedniego punktu zaimplementuj `Drop` tak, aby po wyjściu obiektu z zakresu wypisał komunikat:

   ```
   Ich habe N mal gegrüßt
   ```

   gdzie `N` to liczba wywołań `greet`.

3. Zdefiniuj typ enumeracyjny, który pozwala trzymać daną wartość na stosie lub na stercie:

   ```
   pub enum HeapOrStack<T> {
       Stack(T),
       Heap(Box<T>),
   }
   ```

   i zaimplementuj `Deref<Target = T>` oraz `DerefMut` tak, aby można było wywoływać metody `T` na `&HeapOrStack<T>` i modyfikować przez `&mut HeapOrStack<T>`.

4. Napisz funkcję:

```
pub fn canon_head<'a>(xs: &'a VecDeque<i32>)
    -> Option<Cow<'a, VecDeque<i32>>>;
```

Funkcja ma zwracać taką kolejkę, by jej pierwszy element (głowa) był nieparzysty. Jeżeli `xs` już spełnia warunek albo jest puste, zwróć `Cow::Borrowed` bez alokacji. W przeciwnym razie wykonaj rotację (przenosząc elementy z przodu na koniec). Jeśli wszystkie elementy `xs` są parzyste, to zwróć `None`.

5. Zdefiniuj typ:

```
pub struct CachedFile {
    cache: ???
}
```

Zaimplementuj metody:

```
pub fn new() -> Self;
// Jeżeli cache jest pusty, wczytaj plik z `path` i zapisz w `cache`.
// Jeżeli cache jest już ustawiony, zwróć jego zawartość.
pub fn get(&self, path: &Path) -> &str { ... }

// Jeżeli cache jest pusty, zwróć `None`
// Jeżeli cache jest już ustawiony, zwróć jego zawartość.
pub fn try_get(&self) -> Option<&str> { ... }
```

Użyj `std::fs::read_to_string` oraz `OnceCell`.

6. Zdefiniuj strukturę:

```
#[derive(Clone)]
pub struct SharedFile {
    file: ???
}
```

oraz metody

- `fn new(path: PathBuf) -> Self` tworzy obiekt, który leniwie wczyta zawartość pliku przy pierwszym dostępie.
- `fn get(&self) -> &str` zwraca referencję do treści pliku; wiele klonów `SharedFile` współdzieli ten sam bufor.

do implementacji użyj `Rc` i `LazyCell`.

7. Teraz dodamy możliwość tworzenia grafów skierowanych. Aby nie tworzyć cykli referencji graf będzie budowany najpierw poprzez drzewo z referencji typu `Rc`, a reszta krawędzi będzie tworzona przez referencje typu `Weak`. Zdefiniuj typ wierzchołka:

```
pub struct Vertex {
    pub out_edges_owned: Vec<Rc<RefCell<Vertex>>>,
    pub out_edges:       Vec<Weak<RefCell<Vertex>>>,
    pub data:            i32,
}
```

i zaimplementuj metody

```
// Tworzy pusty wierzchołek (data = 0, puste wektory).
pub fn new() -> Self { ... }

// Tworzy nowego sąsiada z data = 0,
// dodaje go do `out_edges_owned` i zwraca `Rc` na niego.
pub fn create_neighbour(&mut self) -> Rc<RefCell<Vertex>> { ... }

// Dodaje krawędź do istniejącego wierzchołka jako słaba referencja (`Weak`).
pub fn link_to(&mut self, other: &Rc<RefCell<Vertex>>) { ... }

// Zwraca wszystkich sąsiadów w postaci `Weak` (zarówno owned, jak i borrowed).
pub fn all_neighbours(&self) -> Vec<Weak<RefCell<Vertex>>> { ... }
```

Napisz również funkcję pomocniczą:

```
// Buduje cykl długości `n`: v0 -> v1 -> ... -> v{n-1} -> v0
// z danymi odpowiednio 0, 1, ..., n - 1
// Zadbaj o to, aby cykl nie powodował wycieków pamięci!
// (odpowiednio używaj `create_neighbour` i `link_to`)
pub fn cycle(n: usize) -> Rc<RefCell<Vertex>>;
```

Wymagania

- Kod kompiluje się ze stabilnym kompilatorem Rust.
- Brak ostrzeżeń kompilatora i `clippy` (w tym o nieużywanych elementach).
- Wszystkie testy przechodzą.

Ocena (3 pkt)

- 1 pkt: Praca w trakcie laboratorium.
- 1 pkt: Pełna funkcjonalność (implementacja zgodna z wymaganiami).
- 1 pkt: Prezentacja rozwiązania i odpowiedzi na pytania prowadzącego.