Programmation « avancée » en Rust : modules, `unsafe`, mutabilité intérieure

Programmation avancée

Jacques-Henri Jourdan

15 mars 2024

Le rôle des modules en Rust

Le rôle du système de module en Rust :

- Organiser le code dans des espaces de noms.
  - Important pour la compilation séparée : chaque unité de compilation (*crate*) a un espace de nom séparé.
    Pas de conflit de noms entre les unités de compilation.
- Organiser l'accessibilité des objets.
  - Certains types ou fonctions sont à usage interne et ne doivent pas être utilisés à l'extérieur d'un module.
  - Ceci est crucial pour l'abstraction : on veut donner accès à des types et des fonctions, sans permettre au client de les utiliser directement.

Le rôle des modules en Rust

Le rôle du système de module en Rust :

- Organiser le code dans des espaces de noms.
  - Important pour la compilation séparée : chaque unité de compilation (*crate*) a un espace de nom séparé.
    Pas de conflit de noms entre les unités de compilation.
- Organiser l'accessibilité des objets.
  - Certains types ou fonctions sont à usage interne et ne doivent pas être utilisés à l'extérieur d'un module.
  - Ceci est crucial pour l'abstraction : on veut donner accès à des types et des fonctions, sans permettre au client de les utiliser directement.

Contrairement à OCaml, il n'y a pas de foncteur en Rust.
Les modules ne sont pas un mécanisme de programmation générique.
Ce sont les traits qui permettent la programmation générique en Rust !

Modules

Un module est un ensemble d'objets :

```
mod mon_module {
    struct T { u: i32 }
    fn f(x: i32) -> i32 { ... }
    ...
}
```

On peut définir des sous-modules :

```
mod m1 {
    fn f(x: i32) -> i32 { ... }
    mod m2 {
        ....
    }
    ....
}
```

Accéder au contenu d'un module

Depuis l'intérieur d'un module, on peut accéder à son contenu directement :

```
mod m {
  fn f1(x: i32) -> i32 { ... }
  fn f2(x: i32) -> i32 { f1(x) }
}
```

Mais depuis l'extérieur, il faut que l'objet soit déclaré public et qualifier le nom :

```
mod m {
  fn f1(x: i32) -> i32 { ... }
  pub fn f2(x: i32) -> i32 { ... }
}
fn f() {
  let a = m::f1(12);  // Erreur : f1 n'est pas public
  let b = m::f2(32);  // OK
}
```

Objets publics, privés

Par défaut, la visibilité d'un élément d'un module est privée.

Si nécessaire, il faut donc utiliser `pub` pour tous les objets nommés d'un module :

- fonctions, méthodes, constantes, traits, types, sous-modules...

Mais les instances de traits sont toujours publiques.

Par ailleurs, les champs des types `struct` ont aussi une visibilité.

- Par défaut : les champs ne sont accessibles que dans le module.
- Sinon, il faut utiliser `pub` : exemple `struct S { pub a: i32 }`.
- On ne peut construire un objet de type `struct` que si on peut accéder à tous les champs.
- Cela permet de créer des `types abstraits` : aucun champ public.

Imports

Comme vous le savez, on peut importer des objets publics d'autres modules :

```
mod m {
    pub fn g(x: i32) -> i32 { ... }
}

use m::g;

fn f() {
    let b = g(32);
}
```

Le nom g est ajouté au module courant, de manière privée.
I.e., on ne voit pas l'import depuis l'extérieur.

Note : lorsque l'on fait un import, on peut aussi renommer l'objet à la volée :

```
use m::g as h;
```

Exports

Mais on peut aussi ajouter un `pub` :

```
pub use m::g;
```

Cela permet de réexporter `m::g` en tant que membre public du module courant :

```
mod m1 {
    mod m2 {
        pub fn g(x: i32) -> i32 { ... }
    }

    pub use m2::g as h; // On réexporte m2::g en tant que h
}

use m1::h;  // On voit g comme un membre de m1, renommé en h.

fn f() {
  let b = h(32);
}
```

On peut aussi réexporter l'intégralité du contenu public d'un module :

```
pub use m::*
```

Modules et fichiers

En Rust, chaque fichier est un module. On peut « invoquer » un autre fichier en ne donnant pas le contenu d'un module :

```
mod mon_module;  // ou pub mod mon_module, si on veut que le sous-module soit public
```

Cela va chercher :

- un fichier `mon_module.rs`,
- ou un fichier `mod.rs` dans un dossier `mon_module/`,

et utiliser le contenu de ce fichier comme contenu du module `mon_module`.
(Un peu comme on fait `#include`, mais dans un sous-module dédié.)

Du coup, la hiérarchie des modules est *a priori* calquée sur la hiérarchie des fichiers dans un projet.

Exports et organisation d'un projet

À quoi servent les exports ?

Exports et organisation d'un projet

À quoi servent les exports ?
Cela donne de la flexibilité dans l'organisation d'un projet en modules et en fichiers.

A priori, les modules (i.e., les espaces de noms) suivent la même hiérarchie que les fichiers/dossiers.

Mais les exports permettent de modifier ce comportement si besoin.

Exports et organisation d'un projet

Exemple

Imaginons qu'on souhaite faire une bibliothèque de tris.
On veut une fonction `merge_sort` et une fonction `quick_sort`.

Mais comme elles sont compliquées, elles dépendent d'autres fonctions auxiliaires, que l'on souhaite organiser dans des modules (fichiers) séparés.

On n'expose pas cette organisation interne au client :

```
mod merge_sort {  // Sous-module privé : invisible par le client
    fn aux(...) -> ... { ... }
    pub fn sort(&mut [i32]) { ... }
}

mod quick_sort {
    fn aux(...) -> ... { ... }
    pub fn sort(&mut [i32]) { ... }
}

pub use merge_sort::sort as merge_sort;
pub use quick_sort::sort as quick_sort;
```

En pratique, on mettra les modules `merge_sort` et `quick_sort` dans des fichiers dédiés.

Unité de compilation

Exemple

En général, une unité de compilation est la partie d'un programme qui est compilée lors d'un seul appel au compilateur.

En Rust, il s'agit d'une crate, elle peut contenir plusieurs fichiers, mais correspond toujours à un et un seul module à la racine de la hiérarchie.
Une crate est en général soit une bibliothèque, soit un exécutable.

Exemple :

```
// Import du type HashMap dans le sous-module collections du crate std (bibliothèque standard)
use std::collections::HashMap;

// Import du type Bump du crate bumpalo (allocateur de votre projet)
use bumpalo::Bump;
```

Les mécanismes d'abstraction (visibilité+modules) peuvent être utilisés vis-à-vis à la frontière d'un crate, mais aussi en son sein.
I.e., certains sous-modules (« sous-bibliothèques ») peuvent cacher certains de leurs éléments au reste du crate.

1 Modules en Rust

2 Code `unsafe` en Rust
- Notion
- Encapsulation
- Comportements indéfinis
- `unsafe` et aliasing
- Exemple

3 Mutabilité intérieure
- `Cell<T>`
- `RefCell<T>`
- `Rc<T>`

Rappel : sûreté du typage

Rappel : le système de types de certains langages permet d'établir un théorème :

Sûreté du typage

Les programmes bien typés ne plantent pas. « *Well-typed programs don't go wrong.* »

Pour une certaine notion de « bien typés » et de « ne plantent pas ».

Rappel : sûreté du typage

Rappel : le système de types de certains langages permet d'établir un théorème :

Sûreté du typage

Les programmes bien typés ne plantent pas. « *Well-typed programs don't go wrong.* »

Pour une certaine notion de « bien typés » et de « ne plantent pas ».

Contre-exemple : C ou C++ n'ont pas la sûreté du typage.

Même un programme bien typé peut avoir un comportement indéfini.

Parfois, cela a des conséquences catastrophiques, parce qu'un comportement indéfini est imprévisible :

- Si vous êtes chanceux, le système d'exploitation le détecte et arrête le programme immédiatement (« erreur de segmentation »).
- Sinon, le programme continue, mais en faisant n'importe quoi.

Rappel : sûreté du typage

Rappel : le système de types de certains langages permet d'établir un théorème :

Sûreté du typage

Les programmes bien typés ne plantent pas. « *Well-typed programs don't go wrong.* »

Pour une certaine notion de « bien typés » et de « ne plantent pas ».

En OCaml, si un programme est bien typé (i.e., le compilateur l'accepte), alors on sait qu'il ne va pas avoir de comportement indéfini.

- Pas d'erreur de segmentation, e.g., en déréférençant un pointeur invalide.
- Il peut planter « proprement » en levant une exception, par exemple.

Rappel : sûreté du typage

Rappel : le système de types de certains langages permet d'établir un théorème :

Sûreté du typage

Les programmes bien typés ne plantent pas. « *Well-typed programs don't go wrong.* »

Pour une certaine notion de « bien typés » et de « ne plantent pas ».

En OCaml, si un programme est bien typé (i.e., le compilateur l'accepte), alors on sait qu'il ne va pas avoir de comportement indéfini.

- Pas d'erreur de segmentation, e.g., en déréférençant un pointeur invalide.
- Il peut planter « proprement » en levant une exception, par exemple.

Quoique, un programme OCaml qui utilise le module `Obj` ou la sérialisation peut provoquer un comportement indéfini...

unsafe

Malgré les abstractions gratuites, la contrainte de sûreté du langage Rust a des coûts :

- pas de boucle dans le graphe de la mémoire,
- possession partagée : restreinte aux emprunts,
- appel à des bibliothèques externes : discipline de possession pas forcément respectée,
- vérification des bornes pour les accès à `Vec`.

On peut utiliser du code Rust `unsafe` pour éviter ces coûts :

- Des fonctionnalités qui complètent Rust.
- Uniquement disponibles dans des blocs `unsafe` ou des fonctions `unsafe` :

```
unsafe fn(...) -> .. { .... }

fn (...) -> { ... unsafe { .... } ... }
```

Fonctionnalités `unsafe`

- Déréférencer des pointeurs bas-niveau (*raw pointers*).
- Appeler des fonctions `unsafe` :
  - Par exemple, `Vec::get_unchecked` et `Vec::get_unchecked_mut` permettent d'accéder au contenu d'un `Vec` sans vérifier les bornes.
- Implémenter des traits `unsafe`
  - Exemple : les traits `Send` et `Sync`, permettant d'exprimer des propriétés relatives à la concurrence, sont `unsafe`.
    (Nous reparlerons de ces traits dans le cours sur la concurrence.)
- Modifier des variables globales (en Rust, variables annotées `static`).
  - (Car cela viole la règle « mutation XOR alias ».)
- Utiliser des types `union`.
  - C'est une sorte de « type somme », mais sans discriminant. Nous n'en parlerons pas.

Peut-on dire que Rust a la sûreté du typage ?

Peut-on dire que Rust a la sûreté du typage ?

À première vue, non : avec `unsafe`, on peut écrire des programmes Rust qui ont un comportement indéfini.
Est-ce tout ?

Peut-on dire que Rust a la sûreté du typage ?

À première vue, non : avec `unsafe`, on peut écrire des programmes Rust qui ont un comportement indéfini.
Est-ce tout ?

On pourrait dire « tout programme écrit sans `unsafe` est sûr ».
Mais ça sert à rien : (presque) tous les programmes utilisent `unsafe` (exemple : `Vec` est implémenté avec `unsafe`).

Peut-on dire que Rust a la sûreté du typage ?

À première vue, non : avec `unsafe`, on peut écrire des programmes Rust qui ont un comportement indéfini.
Est-ce tout ?

On pourrait dire « tout programme écrit sans `unsafe` est sûr ».
Mais ça sert à rien : (presque) tous les programmes utilisent `unsafe` (exemple : `Vec` est implémenté avec `unsafe`).

La vraie réponse à cette question réside dans l'encapsulation.

Encapsulation du code `unsafe`

Lorsque le programmeur utilise `unsafe`, il a le choix entre :

- Marquer la fonction qu'il écrit comme `unsafe`.
  - Les utilisateurs devront eux-mêmes utiliser `unsafe` pour utiliser la fonction.
  - La bibliothèque doit documenter quel est le risque.
- S'assurer (« à la main ») qu'un utilisateur ne peut pas appeler sa fonction pour déclencher un comportement indéfini.
  - Le danger d'`unsafe` est alors « encapsulé ».
  - Le mécanisme de visibilité des modules de Rust est souvent crucial pour assurer cette encapsulation.

Encapsulation du code `unsafe`

Lorsque le programmeur utilise `unsafe`, il a le choix entre :

- Marquer la fonction qu'il écrit comme `unsafe`.
  - Les utilisateurs devront eux-mêmes utiliser `unsafe` pour utiliser la fonction.
  - La bibliothèque doit documenter quel est le risque.
- S'assurer (« à la main ») qu'un utilisateur ne peut pas appeler sa fonction pour déclencher un comportement indéfini.
  - Le danger d'`unsafe` est alors « encapsulé ».
  - Le mécanisme de visibilité des modules de Rust est souvent crucial pour assurer cette encapsulation.

En fait, c'est un mécanisme fréquent dans les langages dont le système de types est sûr (dont OCaml) :

- Certaines bibliothèques utilisent `Obj` pour faire quelque chose d'interdit par le système de types. Mais l'interface exposée à l'utilisateur est sûre.
- Exemple : analyseurs syntaxiques générés par Ocamlyacc et Menhir.

Mais, en Rust, le mot-clé `unsafe` impose une discipline autour de cette pratique.

Exemple : encapsulation du code `unsafe` dans `Vec`

Le module `std::vec` de la bibliothèque standard est écrit en utilisant `unsafe`.

La définition ressemble à :

```
pub struct Vec<T> {
    ptr: *mut T,  // Pointeur vers les éléments
    cap: usize,   // Taille de la zone allouée en mémoire
    len: usize,   // Nombre d'éléments
}
```

Les accès se font en déréférençant `ptr`.

Mais la plupart des fonctions l'API de `Vec` ne permet pas de déclencher de comportement indéfini (notamment grâce à la vérification des bornes).

Certaines fonctions sont tout de même `unsafe` : exemple `Vec::get_unchecked`, `Vec::get_unchecked_mut`.

Exemple : encapsulation du code `unsafe` dans `Vec`

Le module `std::vec` de la bibliothèque standard est écrit en utilisant `unsafe`.

La définition ressemble à :

```
pub struct Vec<T> {
    ptr: *mut T,  // Pointeur vers les éléments
    cap: usize,   // Taille de la zone allouée en mémoire
    len: usize,   // Nombre d'éléments
}
```

L'encapsulation repose sur le fait que le client n'a pas accès aux champs `ptr`, `cap` et `len`. Sinon, il pourrait déclencher un comportement indéfini en modifiant `len`, ce qui rend la vérification des bornes inefficace.

Exemple : encapsulation du code `unsafe` dans `Vec`

Le module `std::vec` de la bibliothèque standard est écrit en utilisant `unsafe`.

La définition ressemble à :

```
pub struct Vec<T> {
    ptr: *mut T,  // Pointeur vers les éléments
    cap: usize,   // Taille de la zone allouée en mémoire
    len: usize,   // Nombre d'éléments
}
```

L'encapsulation repose sur le fait que le client n'a pas accès aux champs `ptr`, `cap` et `len`. Sinon, il pourrait déclencher un comportement indéfini en modifiant `len`, ce qui rend la vérification des bornes inefficace.

Grâce à l'encapsulation, on peut dire que `Vec` a des invariants supplémentaires.

I.e., l'information `v: Vec<T>`, apporte plus que simplement `ptr: *mut T`, `cap: usize`, ...
On sait aussi des propriétés importantes sur `ptr`, `cap` et `len`.
Ces propriétés permettent de garantir la sûreté de `std::vec`.

Comportements indéfinis

Lorsqu'on utilise `unsafe`, le programme compilé peut avoir un comportement indéfini !

Il faut donc savoir dans quel cas un programme déclenche un comportement indéfini.
C'est parfois très subtile.

Rust fournit un interprète de référence expérimental, appelé Miri, qui détecte certains comportements indéfinis.

- Peut être utilisé sur du code concret pour tester.

Écrire du code `unsafe` devrait néanmoins être réservé aux experts.
Il y a un livre entier dédié à l'écriture de code `unsafe` : le *Rustonomicon*

Je ne vous enseignerai donc pas `unsafe`.
Nous allons plutôt essayer de comprendre pourquoi c'est subtil.

`unsafe` et aliasing

Utilisation habituelle du code `unsafe` : affaiblir les restrictions d'aliasing.

Les pointeurs bas-niveau (*raw pointers*) : `*mut T` et `*const T` :

- n'ont aucune restriction d'aliasing vérifiée statiquement,
- peuvent se convertir depuis/vers les emprunts (à la fois partagés et uniques),
- peuvent être utilisés pour casser la politique d'aliasing.

unsafe et aliasing

Utilisation habituelle du code `unsafe` : affaiblir les restrictions d'aliasing.

Les pointeurs bas-niveau (*raw pointers*) : `*mut T` et `*const T` :

- n'ont aucune restriction d'aliasing vérifiée statiquement,
- peuvent se convertir depuis/vers les emprunts (à la fois partagés et uniques),
- peuvent être utilisés pour casser la politique d'aliasing.

Mais le compilateur utilise des propriétés des emprunts pour faire des optimisations :

```
fn test_noalias(x: &mut i32, y: &mut i32) -> i32 {
    // x, y ne peuvent pas être alias : ce sont des emprunts uniques
    *x = 42;
    *y = 37;
    return *x; // doit renvoyer 42 -- optimisation possible
}
```

unsafe et aliasing

Utilisation habituelle du code `unsafe` : affaiblir les restrictions d'aliasing.

Les pointeurs bas-niveau (*raw pointers*) : `*mut T` et `*const T` :

- n'ont aucune restriction d'aliasing vérifiée statiquement,
- peuvent se convertir depuis/vers les emprunts (à la fois partagés et uniques),
- peuvent être utilisés pour casser la politique d'aliasing.

Mais le compilateur utilise des propriétés des emprunts pour faire des optimisations :

```
fn test_unique(x: &mut i32) -> i32 {
    *x = 42;
    // unknown_function ne peut pas avoir d'alias de x
    unknown_function();
    return *x; // doit renvoyer 42 -- optimisation possible
}
```

unsafe et aliasing

Utilisation habituelle du code `unsafe` : affaiblir les restrictions d'aliasing.

Les pointeurs bas-niveau (*raw pointers*) : `*mut T` et `*const T` :

- n'ont aucune restriction d'aliasing vérifiée statiquement,
- peuvent se convertir depuis/vers les emprunts (à la fois partagés et uniques),
- peuvent être utilisés pour casser la politique d'aliasing.

Mais le compilateur utilise des propriétés des emprunts pour faire des optimisations :

```
fn test_shared(x: &i32) -> i32 {
    let y = *x;
    // unknown_function ne peut pas avoir d'alias de x
    unknown_function();
    return *x + y; // peut être optimisé vers 2*y
}
```

unsafe et aliasing

Utilisation habituelle du code `unsafe` : affaiblir les restrictions d'aliasing.

Les pointeurs bas-niveau (*raw pointers*) : `*mut T` et `*const T` :

- n'ont aucune restriction d'aliasing vérifiée statiquement,
- peuvent se convertir depuis/vers les emprunts (à la fois partagés et uniques),
- peuvent être utilisés pour casser la politique d'aliasing.

Mais le compilateur utilise des propriétés des emprunts pour faire des optimisations :

Ces optimisations deviennent fausses si on fait n'importe quoi avec des pointeurs bas-niveau.

Comportements indéfinis et aliasing

Des règles sont nécessaires pour exprimer ce qu'on peut faire avec les pointeurs bas-niveau.

Ces règles sont un équilibre entre :

- la flexibilité pour le programmeur de code `unsafe` ;
- la possibilité pour le compilateur d'effectuer des optimisations.

Le choix de ces règles dans tous les cas reste, à ce jour, un problème ouvert. L'interprète Miri implémente deux ensembles de règles appelées *Tree Borrows* et *Stacked Borrows* :

- expérimental et imparfait,
- mais exécutable sur des tests concrets.

Comportements indéfinis et aliasing

Des règles sont nécessaires pour exprimer ce qu'on peut faire avec les pointeurs bas-niveau.

Ces règles sont un é
- la flexibilité pou
- la possibilité po

Le choix de ces règl
L'interprète Miri imp
Stacked Borrows :
- expérimental et imparfait,
- mais exécutable sur des tests concrets.

Si vous écrivez du code `unsafe`, vous devez suivre ce genre de règles.

Je vous l'avais dit : écrire du code `unsafe` correct est subtile...

Exemple d'utilisation de `unsafe`

Une file implémentée avec une liste chaînée

Disons que nous voulons implémenter une file (FIFO) avec une liste simplement chaînée.

Il faut un pointeur au début de la liste (pour `pop`) et à sa fin (pour `push`).

Les règles d'aliasing sont violées, nous devons utiliser du code `unsafe`.

Exemple d'utilisation de `unsafe` : `Queue<T>`

```
mod queue {
    pub struct Queue<T> {
        head: *mut Node<T>,
        tail: *mut Node<T>
    }
    struct Node<T> {
        elem: T,
        next: *mut Node<T>,
    }

    ...
}
use queue::*;


fn (q: Queue<i32> /* Allowed */) {
   ...
   q.head  /* Error */
   ...
   Queue { ... } /* Error */
   ...
   let x : Node<i32> /* Error */ = ... ;
   ...
}
```

On utilise les modules pour contrôler la visibilité.

Le type `Node` n'est pas annoté par `pub` : il est privé.

Le type `Queue` est visible à l'extérieur du module. Mais tous ses champs sont privés.

- À l'extérieur du module, `Queue` est un type abstrait !

Exemple d'utilisation de `unsafe` : `Queue<T>`

Corps des fonctions

```
impl<T> List<T> {
    pub fn new() -> Self {
        List { head: ptr::null_mut(), tail: ptr::null_mut() }
    }
    pub fn push(&mut self, elem: T) {
        unsafe {
            let new_tail = Box::into_raw(Box::new(Node {
                elem: elem,
                next: ptr::null_mut(),
            }));

            if !self.tail.is_null() {
                (*self.tail).next = new_tail;
            } else {
                self.head = new_tail;
            }

            self.tail = new_tail;
        }
    }
    pub fn pop(&mut self) -> Option<T> {
    ...
}
```

Exemple d'utilisation de `unsafe` : `Queue<T>`

Corps des fonctions

```
impl<T> List<T> {
    pub fn new() -> Self {
        List { head: ptr::null_mut(), tail: ptr::null_mut() }
    }
    pub fn push(&mut self, elem: T) {
        unsafe {
            let
                }));
            if !
            } el
            }
            self.tail = new_tail;
        }
    }
    pub fn pop(&mut self) -> Option<T> {
    ...
}
```

La difficulté, ici, est de s'assurer que le compilateur ne va pas utiliser des propriétés spécifiques aux emprunts pour faire des optimisations incorrectes.

Intuitivement, ici, c'est correct, car les emprunts sont toujours utilisés de manière « bien parenthésée ».

Exemple d'utilisation de `unsafe` : `Queue<T>`

Corps des fonctions

```
impl<T> List<T>
    pub fn new()
        List { h
    }
    pub fn push(
        unsafe {
            let
                }));
            if !
            } el
            }
            self
        }
    }
    pub fn pop(&
    ...
}
```

Mais c'est une pente glissante.
Par exemple, la définition

```
pub struct Queue<T> {
    head: Box<T>,
    tail: *mut Node<T>
}
struct Node<T> {
    elem: T,
    next: Box<T>,
}
```

aurait été incorrecte, car elle introduit un alias interdit entre `tail` et `head`...

Si vous voulez en savoir plus :

- Chapitre 6 de *Learn Rust With Entirely Too Many Linked Lists*.
  https://rust-unofficial.github.io/too-many-lists/fifth.html

Contourner les règles d'aliasing

Peut-on faire mieux que `unsafe` ?

D'une part, les règles d'aliasing de Rust sont strictes ;
D'autre part, le code `unsafe` semble trop subtile à écrire...

Peut-on faire mieux ?

Contourner les règles d'aliasing

Peut-on faire mieux que `unsafe` ?

D'une part, les règles d'aliasing de Rust sont strictes ;
D'autre part, le code `unsafe` semble trop subtile à écrire...

Peut-on faire mieux ?

On peut utiliser la mutabilité intérieure :

- des bibliothèques qui assouplissent les règles d'aliasing, en respectant la sûreté,
- écrites avec du code `unsafe`, mais encapsulée pour garder la sûreté !
- une caractéristique commune : mutation de la mémoire en utilisant un emprunt partagé, avec des restrictions appropriées.
  - (Utilisation d'annotations spéciales pour désactiver certaines optimisations.)

Une idée d'une API avec mutabilité intérieure ?

Cell<T>

```rust
pub struct Cell<T> { ... }

impl<T> Cell<T> {
    pub fn new(value: T) -> Cell<T> { ... }
    pub fn into_inner(self) -> T { ... }
    pub fn set(&self, val: T) { ... }
    pub fn replace(&self, val: T) -> T { ... }
    pub fn get(&self) -> T where T : Copy { ... }
}
```

Informellement, pourquoi est-ce sûr ?

Cell<T>

```rust
pub struct Cell<T> { ... }

impl<T> Cell<T> {
    pub fn new(value: T) -> Cell<T> { ... }
    pub fn into_inner(self) -> T { ... }
    pub fn set(&self, val: T) { ... }
    pub fn replace(&self, val: T) -> T { ... }
    pub fn get(&self) -> T where T : Copy { ... }
}
```

Informellement, pourquoi est-ce sûr ?

À partir de `&Cell<T>`, on ne peut jamais obtenir un emprunt (partagé ou mutable) du contenu. Ainsi, on ne peut pas casser les invariants d'aliasing des emprunts de `T`.

On peut seulement échanger des valeurs de type `T` ou en faire une copie, mais pas d'emprunt interne.

Cell<T>

```rust
pub struct Cell<T> { ... }

impl<T> Cell<T> {
    pub fn new(value: T) -> Cell<T> { ... }
    pub fn into_inner(self) -> T { ... }
    pub fn set(&self, val: T) { ... }
    pub fn replace(&self, val: T) -> T { ... }
    pub fn get(&self) -> T where T : Copy { ... }
}
```

Informellement, pourquoi est-ce sûr ?

À partir de `&Cell<T>`, on ne peut jamais obtenir un emprunt (partagé ou mutable) du contenu. Ainsi, on ne peut pas casser les invariants d'aliasing des emprunts de `T`.

On peut seulement échanger des valeurs de type `T` ou en faire une copie, mais pas d'emprunt interne.

Et qu'en est-il si on veut un emprunt interne ?

RefCell<T> API(1/2)

RefCell, RefMut

```
pub struct RefCell<T> { ... }
pub struct RefMut<'b, T> where T: 'b { ... }

impl<T> RefCell<T> {
    pub fn new(value: T) -> RefCell<T> { ... }
    pub fn into_inner(self) -> T { ... }

    /* Vérifie qu'il n'y a pas d'emprunt et marque comme emprunté de manière unique. */
    pub fn borrow_mut<'a>(&'a self) -> RefMut<'a, T> { ... }
}

/* Cette instance de DerefMut signifie que RefMut<'b, T> peut être utilisée comme &'b mut T*/
impl<'b, T> DerefMut for RefMut<'b, T> {
    fn deref_mut<'a>(&'a mut self) -> &'a mut T /* where 'b: 'a */ { ... }
}
/* Cette instance de Deref signifie que RefMut<'b, T> peut être utilisée comme &'b T */
impl<'b, T> Deref for RefMut<'b, T> {
    type Target = T
    fn deref<'a>(&'a self) -> &'a T /* where 'b: 'a */ { ... }
}

/* Destructeur. */
impl<'a, T> Drop for RefMut<'a, T> {
    /* Marque le RefCell comme non emprunté. */
    fn drop(&mut self) { ... }
}
```

RefCell<T> Exemple

```rust
fn use_refcell(x : &RefCell<Vec<i32>>) {
    {
        let mut v: RefMut<'_, Vec<i32>> = x.borrow_mut();
        v.push(42);  // v peut être utilisé comme un emprunt unique

        /* Panic: il y a déjà un emprunt unique. */
        /* let v2 = x.borrow_mut().push(16); */

        /* Implicite : v.drop(); */
    }

    /* RefMut est supprimé, je peux en créer un autre : */
    println!("{}", x.borrow_mut()[0]);
}
```

RefCell<T> API (2/2)

Ref

```
...

pub struct Ref<'b, T> where T: 'b { ... }

impl<T> RefCell<T> {
    ...
    /* Vérifie qu'il n'y a pas d'emprunt unique. Incrémente le compteur d'emprunts. */
    pub fn borrow<'a>(&'a self) -> Ref<'a, T> { ... }
}

/* Cette instance de Deref signifie que Ref<'b, T> peut être utilisée comme &'b T */
impl<'b, T> Deref for Ref<'b, T> {
    type Target = T
    fn deref<'a>(&'a self) -> &'a T /* where 'b: 'a */ { ... }
}

/* Destructeur. */
impl<'a, T> Drop for Ref<'a, T> {
    /* Décrémente le compteur d'emprunts. */
    fn drop(&mut self) { ... }
}
```

Sûreté

Pourquoi `RefCell` est sûr ?

Sûreté

Pourquoi `RefCell` est sûr ?

La règle d'aliasing (aliasing XOR mutation) est appliquée dynamiquement, grâce à un compteur interne.

On peut voir `RefCell` comme un verrou lecture/écriture (reader/writer lock) non concurrent.
(Voir le prochain cours sur la concurrence.)

Question subtile sur les durées de vie de l'API de `RefCell`

Dans la bibliothèque standard :

```
impl<'b, T> RefMut<'b, T> {
    pub fn map<U, F>(orig: RefMut<'b, T>, f: F) -> RefMut<'b, U>
        where F: FnOnce(&mut T) -> &mut U
    { ... }
}

impl<'b, T> Ref<'b, T> {
    pub fn map<U, F>(orig: Ref<'b, T>, f: F) -> Ref<'b, U>
        where F: FnOnce(&T) -> &U
    { ... }
}
```

Cela peut être utilisé par exemple pour transformer un `RefMut<'b, T>` en un `RefMut` vers un des champs de `T`.

Question subtile sur les durées de vie de l'API de `RefCell`

Dans la bibliothèque standard :

```
impl<'b, T> RefMut<'b, T> {
    pub fn map<U, F>(orig: RefMut<'b, T>, f: F) -> RefMut<'b, U>
        where F: FnOnce(&mut T) -> &mut U
    { ... }
}

impl<'b, T> Ref<'b, T> {
    pub fn map<U, F>(orig: Ref<'b, T>, f: F) -> Ref<'b, U>
        where F: FnOnce(&T) -> &U
    { ... }
}
```

Cela peut être utilisé par exemple pour transformer un `RefMut<'b, T>` en un `RefMut` vers un des champs de `T`.

Exercice : Quelles sont les durées de vue des emprunts utilisés dans les fermetures ? Donner des (contre-)exemples.

Rc<T>

Un pointeur vers T, avec comptage de références

```
struct Rc<T> { ... }

impl<T> Rc<T> {
    pub fn new(value: T) -> Rc<T> { ... }
}

/* Cette instance de Deref signifie que Rc<T> peut être utilisée comme &T */
impl<T> Deref for Rc<T> {
    type Target = T
    fn deref<'a>(&'a self) -> &'a T { ... }
}

impl<T> Clone for Rc<T> {
    /* Copie le pointeur, incrémenter le compte de références. */
    fn clone(&self) -> Rc<T> { ... }
}

impl<T> Drop for Rc<T> {
    /* Supprime le pointeur, décrémente le compte de référence et drop+désallocation
       récursive si le compte est nul. */
    fn drop(&mut self) { ... }
}
```

Rc<T>

Un pointeur vers T, avec comptage de références

```
struct Rc<T> { ... }

impl<T> Rc<T> {
    pub fn new(value: T) -> Rc<T> { ... }
}

/* Cette instanc
impl<T> Deref fo
    type Target
    fn deref<'a>
}

impl<T> Clone fo
    /* Copie le
    fn clone(&sel
}

impl<T> Drop for Rc<T> {
    /* Supprime le pointeur, décrémente le compte de référence et drop+désallocation
       récursive si le compte est nul. */
    fn drop(&mut self) { ... }
}
```

Typiquement utilisé pour implémenter des structures de données avec partage.
Exemple : des dictionnaires purement fonctionnels, BDDs...

Pourquoi dis-je que c'est de la mutabilité intérieure ?

Rc<T>

Un pointeur vers T, avec comptage de références

```
struct Rc<T> { ... }

impl<T> Rc<T> {
    pub fn new(value: T) -> Rc<T> { ... }
}

/* Cette instanc
impl<T> Deref fo
    type Target
    fn deref<'a>
}

impl<T> Clone fo
    /* Copie le
    fn clone(&sel
}

impl<T> Drop for Rc<T> {
    /* Supprime le pointeur, décrémente le compte de référence et drop+désallocation
       récursive si le compte est nul. */
    fn drop(&mut self) { ... }
}
```

Typiquement utilisé pour implémenter des structures de données avec partage.
Exemple : des dictionnaires purement fonctionnels, BDDs...

La mutabilité intérieure est limitée au compte de références.

Obtenir des références mutables

A priori, un `Rc<T>` peut être aliasé, donc on n'a pas d'instance de `DerefMut`.

Mais on a :

```
impl<T> Rc<T> {
    /* Vérifie que le compteur est égal à 1. */
    pub fn get_mut(this: &mut Rc<T>) -> Option<&mut T> { ... }

    /* ''Clone-on-write'': clone le contenu dans une nouvelle zone mémoire
       si le compte est différent de 1. */
     pub fn make_mut(this: &mut Rc<T>) -> &mut T
}
```

Bien sûr, cela empêche la mutation et l'aliasing.
Comment obtenir une fonctionnalité proche du type `ref` d'OCaml ?

Obtenir des références mutables

A priori, un `Rc<T>` peut être aliasé, donc on n'a pas d'instance de `DerefMut`.

Mais on a :

```
impl<T> Rc<T> {
    /* Vérifie que le compteur est égal à 1. */
    pub fn get_mut(this: &mut Rc<T>) -> Option<&mut T> { ... }

    /* ''Clone-on-write'': clone le contenu dans une nouvelle zone mémoire
       si le compte est différent de 1. */
     pub fn make_mut(this: &mut Rc<T>) -> &mut T
}
```

Bien sûr, cela empêche la mutation et l'aliasing.
Comment obtenir une fonctionnalité proche du type `ref` d'OCaml ?

Réponse : `Rc<RefCell<T>>`. C'est une association assez commune.

Une remarque sur les performances

Le comptage de références a une réputation de lenteur.
C'est parce qu'il nécessite en général un grand nombre de modifications du compte (ex : passage de paramètres, affectation d'une variable...).

En Rust, on peut mélanger les pointeurs `Rc` et les emprunts :

- Emprunts quand on parcourt une structure de données en lecture seule.
- Le compte n'est incrémenté que lorsqu'un pointeur à longue durée de vie est créé.

Cela donne un plus de contrôle, et de meilleures performances.