# 2_References_fr

May 2, 2023

## 1 Références et emprunt (borrowing)

Les **références** (esperluettes) permettent d'utiliser une variable sans en prendre la propriété.

[ ]:
```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
main();
```

> **Note:** L'opération inverse est le **déréférencement** (dereferencing), qui se fait par l'intermédiaire de l'opérateur `*`.

Quand on utilise des références comme paramètres de fonction on dit qu'on fait un emprunt (**borrowing**). Comme dans la vraie vie, si une personne possède quelque chose, nous devons l'emprunter pour l'utiliser. Et lorsqu'un on a fini de s'en servir, on doit le lui rendre.

Tout comme les variables, les références sont immuables par défaut. On n'a pas le droit de modifier quelque chose qu'on a obtenu par référencement.

[ ]:
```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
main();
```

### 1.1 Références mutables

[ ]:
```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
main();
```

On ne peut avoir qu'**une seule référence mutable** sur une variable, dans un contexte donné. Le code suivant est incorrect :

[ ]:
```
fn main() {
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

    println!("{}, {}", r1, r2);
}
main();
```

L'avantage de cette restriction est d'éviter les courses de données (*data races*) au moment de la compilation. Une course de données apparaît quand :

- Deux ou plusieurs pointeurs essayent d'accéder à la même donnée simultanément.
- Au moins un des pointeurs est utilisé pour écrire/modifier la donnée.
- Il n'y a pas de mécanisme pour synchroniser l'accès à la donnée.

On ne peut avoir une référence immuable et une référence mutable en même temps sur une variable. Les utilisateurs d'une référence immuable s'attendent que les valeurs ne changent pas en cours de route ! Le code suivant est incorrect :

[ ]:
```
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // ok
    let r2 = &s; // ok
    let r3 = &mut s; // GROS PROBLEME!

    println!("{}, {}, and {}", r1, r2, r3);
}
main();
```

### 1.2 Pointeur suspendu (Dangling pointer)

Dans les langages à pointeurs, il est facile de générer, par erreur, des pointeurs suspendus (un pointeur qui référence une zone de mémoire qui ne lui appartient plus). Cela peut arriver si on libère une zone de mémoire sans supprimer le pointeur qui la référence. En Rust, cela ne peut pas arriver. Le compilateur garantit qu'aucune référence n'est suspendue. Ainsi, une donnée référencée ne peut pas expirée (devenir out of scope) avant sa référence.

Exemple de pointeur suspendu :

[ ]:
```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s // on retourne une référence au String, s
} // Ici, s sort du contexte (expire) et est supprimée. Sa zone de mémoire est␣
  ↪libérée.
  // Danger!

main();
```

La solution ici est de retourner directement le String :

[ ]:
```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

## 2 Validation des références par la durée de vie (lifetime)

Chaque référence a une durée de vie (**lifetime**) représentée par le contexte dans lequel elle est valide. La plupart du temps, les durées de vie sont inférées et implicites. Le compilateur Rust dispose d'un borrow checker qui compare les scopes (contextes) pour déterminer si tous les emprunts sont valides.

[ ]:
```
// le code suivant ne compile pas, parce que la durée de vie de x ('b) est plus␣
  ↪courte
// que la durée de vie de r ('a)
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
```

```
                            //          |
    println!("r: {}", r); //          |
}
```

[ ]:
```
// ce code compile
fn main() {
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}
main();
```

### 2.1 Annotation des durées de vie

De la même manière que l'on doit annoter les types quand plusieurs types sont possibles, on doit également annoter les durées de vie pour éviter les ambiguïtés. Les annotations des durées de vie décrivent les relations qui existent entre les durées de vie de plusieurs références. Ces annotations n'ont aucun impact sur les durées de vie elles-mêmes.

En Rust il faut annoter les relations avec des paramètres lifetime génériques afin de s'assurer que les références utilisées à l'exécution soient valides.

```
&i32        // une référence
&'a i32     // une référence avec une durée de vie explicite
&'a mut i32 // une référence mutable avec une durée de vie explicite
```

### 2.2 Annotations lifetime dans la signature d'une function

[ ]:
```
/* pour la durée de vie 'a, la fonction prend 2 paramètres,
   les deux étant des slices de chaîne de caractères avec une durée de vie
   au moins égale à la durée de vie 'a
*/
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let string1 = String::from("long string is long");
    let string2 = String::from("xyz");
    let result = longest(&string1, &string2);
    println!("The longest string is: \"{}\"", result);
```

```
}
main();
```

Quand une fonction utilise des références vers (ou provenant) du code externe, il devient impossible pour le compilateur de déduire les durées de vie des paramètres ou de retourner des valeurs. Les durées de vie peuvent varier à chaque appel de la fonction. On doit donc annoter manuellement les durées de vie.

Quand on passe des références concrètes à `longest`, la durée de vie `'a` est égale à l'intersection entre le scope de `x` et celui de `y`. Autrement dit, la durée de vie générique `'a` sera égale à la durée de vie la plus courte des paramètres `x` et `y`.

### 2.3 Annotations lifetime dans les définitions des struct

Les structs peuvent contenir des références à condition d'ajouter une annotation lifetime à chaque référence dans la définition du struct.

[ ]:
```
fn main() {
    #[derive(Debug)]
    // une instance de ImportantExcerpt ne peut pas avoir une durée de vie plus␣
 ↪grande que la référence contenue
    // dans le champs 'part'
    struct ImportantExcerpt<'a> {
        part: &'a str,
    }

    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.')
        .next()
        .expect("Could not find a '.'");
    let i = ImportantExcerpt { part: first_sentence };
    println!("{:#?}", i);
}
main();
```

### 2.4 Annotations lifetime dans les définitions des methodes

Il est obligatoire de : * déclarer le paramètre lifetime (ici 'a) après le mot clé `impl` * utiliser le paramètre lifetime après le nom du type

Il n'est pas nécessaire d'annoter la durée de vie de la référence vers self.

[ ]:
```
fn main() {
    struct ImportantExcerpt<'a> {
        part: &'a str,
    }
    impl<'a> ImportantExcerpt<'a> {
        fn announce_and_return_part(&self, announcement: &str) -> &str {
            println!("Attention please: {}", announcement);
```

```
            self.part
        }
    }
}
```

### 2.5 Durée de vie statique

Les durées de vie statiques permettent de référencer des variables statiques ou des données qui ne changent pas. La durée de vie statique est égale à la durée du programme.

Toutes les chaînes de littéraux ont une durée de vie statique qui peut être annotée ainsi :

[ ]:
```
let s: &'static str = "I have a static lifetime.";
```

## 3 Exercices

**Exercice 1:** Identifier les erreurs dans le code et corriger-les.

[ ]:
```
fn main() {
    let mut x = 5;

    let y = &mut x;

    *y += 1;

    println!("{}", x);
    // résultat attendu : le code affiche 6
    println!("{}", y);
}
main();
```

**Solution:**

[ ]:
```
fn main() {
    let mut x = 5;

    {
        let y = &mut x; // -+ &mut borrow starts here
        *y += 1;        //  |
    }                   // -+ ... and ends here

    println!("{}", x);  // <- try to borrow x here
}
main();
```