Les **clôtures** en **Rust** sont des **fonctions anonymes** qui peuvent être sauvegardées dans une variable ou qui peuvent être passées en argument d'autres fonctions. Il est possible de créer une **clôture** à un endroit du code et ensuite de l'appeler dans un contexte différent pour l'exécuter. Contrairement aux fonctions, les **clôtures** ont la possibilité de capturer les valeurs présentes dans le contexte où elles sont appelées. Nous allons montrer comment les fonctionnalités des **clôtures** permettent de réutiliser du code et suivre des comportements personnalisés.

> *Les fonction anonymes, appelées aussi **expressions lambda** sont bien connues dans la plupart des langages de programmation qui ont souvent été développée après coup. Nous allons voir comment utiliser les **clôtures** avec les méthodes de la librairie standard, comment une **clôture** sait capturer une variable dans sa portée, comment écrire des fonctions et des méthodes qui puissent accepter une **clôture** en paramètre et comment fermer une **clôture** pour réutilisation ultérieure en tant que fonction de rappel.*

Définitions des clôtures

Rust reconnaît les **clôtures**, en quelque sorte des valeurs à usage local en forme de fonctions légères. La syntaxe d'une **clôture** comporte une liste de paramètres (ou pas) entre **barres verticales** suivie d'une **expression**. Pour appeler une **clôture**, la syntaxe à appliquer est la même que celle de l'appel d'une fonction classique.

Premières clôtures

```
fn main() {
  let est_paire = |x| x%2==0;
  let est_impaire= |x| !est_paire(x);

  println!("13 est paire : {}", est_paire(13));
  println!("24 est paire : {}", est_paire(24));
  println!("13 est impaire : {}", est_impaire(13));
  println!("24 est impaire : {}", est_impaire(24));
}
```

Résultat

```
13 est paire : false
24 est paire : true
13 est impaire : true
24 est impaire : false
```

> ***Rust** déduit d'office le type des paramètres et de la valeur renvoyée. Vous pouvez les indiquer comme pour une fonction. Si vous spécifiez le type à renvoyer. Le corps de la **clôture** doit constituer un bloc afin de respecter la lisibilité syntaxique. Les **clôtures** constituent l'une des caractéristiques les plus élégantes du langage **Rust**.*

Premières clôtures

```
fn main() {
  let est_paire = |x: u32| ->bool { x%2==0 };
  let est_impaire= |x: u32| -> bool { !est_paire(x) };

  println!("13 est paire : {}", est_paire(13));
  println!("24 est paire : {}", est_paire(24));
  println!("13 est impaire : {}", est_impaire(13));
  println!("24 est impaire : {}", est_impaire(24));
}
```

Résultat

```
13 est paire : false
24 est paire : true
13 est impaire : true
24 est impaire : false
```

Clôture en argument de fonction

Rien n'empêche une fonction ou une méthode d'accepter une autre fonction en paramètre ou une **clôture**. Beaucoup de méthodes dans **Rust** sont implémentées de cette façon avec des comportement prédéfinis. C'est notamment le cas pour les méthodes de tri associé à la plupart des types prédéfinis.

> *Le fait de proposer des **clôtures** juste au moment de l'appel de la méthode de tri permet d'avoir une syntaxe concise. Le code est du coup relativement simple à comprendre. Dans l'exemple ci-dessous, nous utilisons la méthode **sort_by()** qui attend en argument l'algorithme de tri en prenant en compte deux éléments consécutifs de la collection.*

Tri décroissant

```
use std::cmp::Ordering;
fn main() {
   let entiers = [4, 8, 1, 10, 0, 45, 12, 7];
   let mut croissant = entiers.clone();
   let mut decroissant = entiers.clone();
```

```
    croissant.sort();
    decroissant.sort_by(|a, b|
       if a<b       { Ordering::Greater }
       else if a>b { Ordering::Less }
       else         { Ordering::Equal });

    println!("entiers : {:?}", entiers);
    println!("croissant : {:?}", croissant);
    println!("décroissant : {:?}", decroissant);
}
```

Résultat

```
entiers : [4, 8, 1, 10, 0, 45, 12, 7]
croissant : [0, 1, 4, 7, 8, 10, 12, 45]
décroissant : [45, 12, 10, 8, 7, 4, 1, 0]
```

*Les collections proposent une méthode **sort()** prédéfinie qui propose de replacer les valeurs dans l'ordre croissant. Si vous désirez avoir un ordre différent, vous utilisez alors la méthode **sort_by()** prévue à cet effet.*

*Ici, la **clôture** est relativement longue puisque nous passons par une suite de tests en collaboration de l'énumération prédéfinie **Ordering**. Nous pouvons être beaucoup plus concis en utilisant directement la méthode **cmp()**, qui permet de faire le même traitement, et qui est systématiquement intégré sur tous les types primitifs (prédéfinis).*

Tri décroissant

```
fn main() {
  let entiers = [4, 8, 1, 10, 0, 45, 12, 7];
  let mut croissant = entiers.clone();
  let mut decroissant = entiers.clone();

  croissant.sort();
  decroissant.sort_by(|a, b| b.cmp(a) );

  println!("entiers : {:?}", entiers);
  println!("croissant : {:?}", croissant);
  println!("décroissant : {:?}", decroissant);
}
```

*Au travers de cet exemple, nous voyons bien ici l'intérêt, la souplesse et la puissance des **clôtures**. L'aspect le plus important est certainement la concision, ce qui fait son principal atout.*

*Toujours dans la rubrique de la gestion des tris, nous pouvons utiliser cette fois-ci la méthode **sort_by_key()** d'une collection qui permet de sélectionner un champ particulier d'une structure si vous souhaitez classer l'ensemble des éléments suivant un critère spécifique.*

Tris spécifiques

```
#[derive(Debug)]
struct Personne<'a> {
  nom: &'a str,
  prenom: &'a str,
  age: i32
}

fn trier_nom(personnes: &mut Vec<Personne>) {
  personnes.sort_by_key(|personne| personne.nom);
}

fn trier_age(personnes: &mut Vec<Personne>) {
  personnes.sort_by_key(|personne| -personne.age);
}

fn afficher(titre: &str, personnes: &Vec<Personne>) {
  println!("{:.>60}", titre);
  for personne in personnes { println!("{:?}", personne); }
}

fn main() {
  let mut personnes= vec![
    Personne { nom: "BROSSE", prenom: "Adam", age: 18 },
    Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 },
    Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 },
    Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
  ];

  afficher("désordre", &personnes);
  trier_nom(&mut personnes);
  afficher("nom", &personnes);
  trier_age(&mut personnes);
```

```
  afficher("âge", &personnes);
}
```

Résultat

```
.....désordre
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
.....nom
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
.....âge
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
```

Capture de variables

Dans les exemples précédents, nous n'avons utilisé les **clôtures** uniquement comme des **fonctions anonymes** internes. Cependant, les **clôtures** possèdent une capacité supplémentaire que les fonctions n'ont pas : elles peuvent capturer leur environnement et accéder aux variables de la portée dans laquelle elles sont définies.

*Dans l'exemple qui suit, nous pouvons nous passer de la **clôture**. Toutefois, le code de la fonction **divisions()** est plus agréable à lire et à comprendre, notamment au moment de l'affichage.*

*Notre **clôture** utilise la variable **diviseur** qui appartient à la fonction englobante **divisions()**. La **clôture division** « **capture** » **diviseur**, ce qui est une des caractéristiques normales des **clôtures**.*

Capture d'une variable dans une clôture

```
fn divisions(nombres: &[u32], diviseur: u32) {
  let division = |dividende| (dividende, dividende/diviseur, dividende%diviseur);
  println!("diviseur = {}", diviseur);
  for nombre in nombres {
    println!("dividende, quotient, reste = {:?}", division(nombre));
  }
}

fn main() {
  divisions(&[15, 17, 21, 25], 3);
  divisions(&[15, 17, 21, 25], 7);
}
```

Résultat

```
diviseur = 3
dividende, quotient, reste = (15, 5, 0)
dividende, quotient, reste = (17, 5, 2)
dividende, quotient, reste = (21, 7, 0)
dividende, quotient, reste = (25, 8, 1)
diviseur = 7
dividende, quotient, reste = (15, 2, 1)
dividende, quotient, reste = (17, 2, 3)
dividende, quotient, reste = (21, 3, 0)
dividende, quotient, reste = (25, 3, 4)
```

*Reprenons l'exemple du chapitre précédent où la **clôture** est beaucoup plus justifiée et qui va nous permettre de faire un tri suivant le **nom** ou le **prénom** de la **personne**.*

Capture d'une variable dans une clôture

```
#[derive(Debug)]
struct Personne<'a> {
  nom: &'a str,
  prenom: &'a str,
  age: i32
}

enum Choix {Nom, Prenom}
use Choix::*;

fn trier(personnes: &mut Vec<Personne>, choix: Choix) {
  personnes.sort_by_key(|personne| match choix {
    Nom => personne.nom,
    Prenom => personne.prenom
```

```
  });
}

fn afficher(titre: &str, personnes: &Vec<Personne>) {
  println!("{:.>60}", titre);
  for personne in personnes { println!("{:?}", personne); }
}

fn main() {
  let mut personnes= vec![
    Personne { nom: "BROSSE", prenom: "Adam", age: 18 },
    Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 },
    Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 },
    Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
  ];

  afficher("désordre", &personnes);
  trier(&mut personnes, Nom);
  afficher("nom", &personnes);
  trier(&mut personnes, Prenom);
  afficher("prénom", &personnes);
}
```

Résultat

```
.....................................................................désordre
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
.......................................................................nom
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
....................................................................prénom
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
```

*Lorsque **Rust** crée la **clôture**, il va automatiquement emprunter une référence à **choix**, ce qui est logique puisque la **clôture** doit pouvoir accéder à cette variable. La suite est très simple : la **clôture** se plie aux règles concernant les **emprunts** et les **durées de vie**.*

*Les **clôtures** peuvent capturer les valeurs de leur environnement de trois façons différentes, qui correspondent directement aux trois façons dont une fonction peut prendre un paramètre : prendre **possession**, **emprunter** de manière **immuable** et **emprunter** de manière **mutable**. Ces moyens sont codés dans les trois traits **Fn** comme ceci :*

- ***FnOnce** consomme les variables qu'il capture à partir de sa portée, connu sous le nom de l'environnement de la **clôture**. Pour consommer les variables capturées, la **clôture** doit prendre **possession** de ces variables et les déplacer dans la **clôture** lorsqu'elle est définie. La partie **Once** du nom représente le fait que la **clôture** ne puisse pas prendre **possession** des mêmes variables plus d'une fois, donc **elle ne peut être appelée qu'une seule fois**.*
- ***FnMut** peut changer l'environnement car elle **emprunte** des valeurs de manière **mutable**.*
- ***Fn emprunte** des valeurs de l'environnement de manière **immuable**.*

*Lorsque nous créons une **clôture**, **Rust** déduit quel trait utiliser en se basant sur la façon dont la **clôture** utilise les valeurs de l'environnement. Toutes les **clôtures** implémentent **FnOne** car elles peuvent toute être appelées au moins une fois. Les **clôtures** qui ne déplacent pas les variables capturées implémentent également **FnMut**, et les **clôtures** qui n'ont pas besoin d'accès mutable aux variables capturées implémentent aussi **Fn**.*

*Dans le premier exemple de ce chapitre, la clôture **division emprunte diviseur** immuablement (donc **division** utilise le trait **Fn**) parce que le corps de la **clôture** ne fait que lire la valeur de **diviseur**. C'est également le cas de l'exemple précédent.*

*Dans l'écriture du **match** qui constitue la **clôture**, nous remarquons que nous ne pouvons pas choisir **l'âge** comme critère de tri puisque le type est différents des deux autres, **nom** et **prénom**. Avec l'inférence de type, la **clôture** indique le type associé à la clé, ici **&str** puisque c'est le premier choix proposé avec le tri par nom.*

*Si vous désirez avoir le choix de tous les critères possibles constituant la structure, la seule possibilité est de proposer une **clôture** pour chaque champ de la structure. Voici la modification qui prend en compte ces différents critères :*

Plusieurs clôtures

```
#[derive(Debug)]
struct Personne<'a> {
  nom: &'a str,
  prenom: &'a str,
  age: i32
}
```

```rust
enum Choix {Nom, Prenom, Age}
use Choix::*;

fn trier(personnes: &mut Vec<Personne>, choix: Choix) {
  match choix {
    Nom =>     personnes.sort_by_key(|personne| personne.nom),
    Prenom => personnes.sort_by_key(|personne| personne.prenom),
    Age =>     personnes.sort_by_key(|personne| -personne.age)
  }
}

fn afficher(titre: &str, personnes: &Vec<Personne>) {
  println!("{:.>60}", titre);
  for personne in personnes { println!("{:?}", personne); }
}

fn main() {
  let mut personnes= vec![
    Personne { nom: "BROSSE", prenom: "Adam", age: 18 },
    Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 },
    Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 },
    Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
  ];

  afficher("désordre", &personnes);
  trier(&mut personnes, Nom);
  afficher("nom", &personnes);
  trier(&mut personnes, Prenom);
  afficher("prénom", &personnes);
  trier(&mut personnes, Age);
  afficher("âge", &personnes);
}
```

Résultat

```
.....................................................................désordre
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
.......................................................................nom
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
.....................................................................prénom
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
.......................................................................âge
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
```

Capture qui dérobe

Si nous voulons forcer la **clôture** à prendre **possession** des valeurs qu'elle utilise dans l'environnement, nous pouvons utiliser le mot-clé **move** avant la liste des paramètres. Cette technique est notamment très utile lorsque vous passez une **clôture** à une nouvelle **tâche** pour déplacer les données afin qu'elles appartiennent à cette **nouvelle tâche**.

> *Nous aborderons la gestion **multi-tâches** lors d'une autre étude. Je vous invite juste à montrer les conséquences de l'utilisation du mot-clé **move** dans une **clôture**.*

Sans move

```rust
fn main() {
  let nombres = vec![15, 17, 21, 25];
  let verifie = |nombre| nombres.contains(nombre);
  let x= 5;
  println!("{:?} possède {} : {}", nombres, x, verifie(&x));
}
```

Résultat

```
[15, 17, 21, 25] possède 5 : false
```

Avec move

```
fn main() {
  let nombres = vec![15, 17, 21, 25];
  let verifie = move |nombre| nombres.contains(nombre);
  let x= 5;
  println!("{:?} possède {} : {}", nombres, x, verifie(&x));
}
```

Résultat

```
error[E0382]: borrow of moved value: `nombres`
 --> src/main.rs:5:37
  |
2 |   let nombres = vec![15, 17, 21, 25];
  |       ------- move occurs because `nombres` has type `Vec<i32>`, which does not implement the `Copy` trait
3 |   let verifie = move |nombre| nombres.contains(nombre);
  |                 ------------ ------- variable moved due to use in closure
  |                 |
  |                 value moved into closure here
4 |   let x= 5;
5 |   println!("{:?} possède {} : {}", nombres, x, verifie(&x));
  |                                    ^^^^^^^ value borrowed here after move
```

Type des fonctions et des clôtures

Les fonctions comme les **clôtures** peuvent être utilisées en tant que valeur, ce qui signifie qu'elles possèdent un type. Dans l'exemple ci-dessous, les fonctions **abscisse()** et **ordonnee()** suivent la même forme de signature avec un paramètre de type **&Point** et renvoient une valeur de type **i32**.

*Toutes les opérations que vous pouvez faire avec une valeur, vous pouvez les réaliser avec une fonction. Vous pouvez stocker une fonction dans une **variable** et utiliser la syntaxe de **Rust** pour calculer la valeur d'une fonction.*

*Dans l'exemple qui suit, nous avons créé la variable mutable **fn_cle** initialisée avec **abscisse** et qui change de valeur plus loin dans le code avec l'autre fonction **ordonnee**.*

Variable fonction

```
#[derive(Debug)]
struct Point { x: i32, y: i32 }

fn abscisse(point: &Point) -> i32 { point.x }
fn ordonnee(point: &Point) -> i32 { point.y }

fn main() {
  let mut fn_cle : fn(&Point) -> i32 = abscisse;
  let mut points= vec![
    Point { x: 2, y: -5 },
    Point { x: -1, y: 3 },
    Point { x: 4, y: 0 },
    Point { x: -2, y: -1 }
  ];
  println!("désordre = {:?}", &points);
  points.sort_by_key(fn_cle);
  println!("abscisse = {:?}", &points);
  fn_cle = ordonnee;
  points.sort_by_key(fn_cle);
  println!("ordonnée = {:?}", &points);
}
```

Résultat

```
désordre = [Point { x: 2, y: -5 }, Point { x: -1, y: 3 }, Point { x: 4, y: 0 }, Point { x: -2, y: -1 }]
abscisse = [Point { x: -2, y: -1 }, Point { x: -1, y: 3 }, Point { x: 2, y: -5 }, Point { x: 4, y: 0 }]
ordonnée = [Point { x: 2, y: -5 }, Point { x: -2, y: -1 }, Point { x: 4, y: 0 }, Point { x: -1, y: 3 }]
```

*Grâce à ce mécanisme, une structure peut posséder un champ de type fonction. Les types génériques tels que **Vec** peuvent accueillir toute une série de fonctions, à condition qu'elles soient toutes du même type **fn**.*

*Les valeurs des fonctions restent compactes : la valeur d'une **fn** se résume à l'adresse mémoire de son code exécutable, comme un pointeur sur fonction du **C++**.*

*Vu ces considérations là, rien n'empêche une fonction d'accepter une autre fonction en paramètre. Si vous connaissez les pointeurs sur fonction du **C++**, vous verrez que les valeurs de fonctions **Rust** représentent strictement la même chose.*

Fonction en paramètre d'une autre fonction

```
#[derive(Debug)]
struct Point { x: i32, y: i32 }
```

```rust
fn abscisse(point: &Point) -> i32 { point.x }
fn ordonnee(point: &Point) -> i32 { point.y }

fn afficher(titre: &str, points: &mut Vec<Point>, cle: fn(&Point) -> i32) {
  if titre!="désordre" {
    points.sort_by_key(cle);
  }
  println!("{:.>50}", titre);
  for point in points { println!("{:?}", point); }
}

fn main() {
  let mut points= vec![
    Point { x: 2, y: -5 },
    Point { x: -1, y: 3 },
    Point { x: 4, y: 0 },
    Point { x: -2, y: -1 }
  ];
  afficher("désordre", &mut points, abscisse);
  afficher("abscisse", &mut points, abscisse);
  afficher("ordonnée", &mut points, ordonnee);
}
```

Résultat

```
.....désordre
Point { x: 2, y: -5 }
Point { x: -1, y: 3 }
Point { x: 4, y: 0 }
Point { x: -2, y: -1 }
.....abscisse
Point { x: -2, y: -1 }
Point { x: -1, y: 3 }
Point { x: 2, y: -5 }
Point { x: 4, y: 0 }
.....ordonnée
Point { x: 2, y: -5 }
Point { x: -2, y: -1 }
Point { x: 4, y: 0 }
Point { x: -1, y: 3 }
```

*Lorsque une fonction attend une autre fonction en paramètre, l'idéal serait de pouvoir proposer des **clôtures** en lieu et place de la fonction attendue, puisque une **clôture** est aussi une fonction sauf qu'elle est anonyme.*

Fonction ou clôture en argument d'une autre fonction

```rust
#[derive(Debug)]
struct Point { x: i32, y: i32 }

fn abscisse(point: &Point) -> i32 { point.x }

fn afficher(titre: &str, points: &mut Vec<Point>, cle: fn(&Point) -> i32)
{
  if titre!="désordre" {
    points.sort_by_key(cle);
  }
  println!("{:.>50}", titre);
  for point in points { println!("{:?}", point); }
}

fn main() {
  let mut points= vec![
    Point { x: 2, y: -5 },
    Point { x: -1, y: 3 },
    Point { x: 4, y: 0 },
    Point { x: -2, y: -1 }
  ];
  afficher("désordre", &mut points, |_point| 0);
  afficher("abscisse", &mut points, abscisse);
  afficher("ordonnée", &mut points, |point| point.y);
}
```

Résultat

```
.....désordre
Point { x: 2, y: -5 }
Point { x: -1, y: 3 }
Point { x: 4, y: 0 }
Point { x: -2, y: -1 }
.....abscisse
Point { x: -2, y: -1 }
Point { x: -1, y: 3 }
Point { x: 2, y: -5 }
Point { x: 4, y: 0 }
.....ordonnée
Point { x: 2, y: -5 }
Point { x: -2, y: -1 }
Point { x: 4, y: 0 }
Point { x: -1, y: 3 }
```

*En reprenant l'exemple sur les structures de personnes, nous pouvons exécuter le même type de codage. J'en profite pour rajouter l'énumération **Option<T>** afin que la fonction d'affichage puisse éventuellement ne prendre aucune **clôture** (ou fonction) dans l'argument attendant une option de type fonction.*

Fonction ou clôture en argument d'une autre fonction

```
#[derive(Debug)]
struct Personne {
  nom: String,
  prenom: String,
  age: i32
}

fn afficher(titre: &str, personnes: &mut Vec<Personne>, choix: Option<fn(&Personne) -> i32>) {
  if let Some(n) = choix {
    personnes.sort_by_key(n)
  }
  println!("{:.>80}", titre);
  for personne in personnes { println!("{:?}", personne); }
}

fn vieux(personne: &Personne) -> i32 { -personne.age }

fn main() {
  let mut personnes= vec![
    Personne { nom: "BROSSE".to_string(), prenom: "Adam".to_string(), age: 18 },
    Personne { nom: "PÊCHEUR".to_string(), prenom: "Martin".to_string(), age: 23 },
    Personne { nom: "BORÉALE".to_string(), prenom: "Aurore".to_string(), age: 20 },
    Personne { nom: "TÉRIEUR".to_string(), prenom: "Alain".to_string(), age: 19 }
  ];

  afficher("Désordre", &mut personnes, None);
  afficher("Plus âgé", &mut personnes, Some(vieux));
  afficher("Plus jeune", &mut personnes, Some(|personne| personne.age));
}
```

Résultat

```
.....Désordre
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
.....Plus âgé
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
.....Plus jeune
Personne { nom: "BROSSE", prenom: "Adam", age: 18 }
Personne { nom: "TÉRIEUR", prenom: "Alain", age: 19 }
Personne { nom: "BORÉALE", prenom: "Aurore", age: 20 }
Personne { nom: "PÊCHEUR", prenom: "Martin", age: 23 }
```