Ce qui fait le plus gros succès du langage **Rust**, c'est qu'il maîtrise parfaitement la gestion de la mémoire**. Rust** répond a deux besoins essentiels pour qu'un langage puisse être utilisé pour pratiquer la programmation système sûre :

> *C'est vous qui décidez de la durée de vie de chacune des valeurs du programme. **Rust** libère la mémoire et les ressources associées à une valeur sous votre entier contrôle.*
>
> *Vos programmes n'utiliserons jamais de pointeur sur un objet qui a été supprimé. C'est une faute fréquente en **C++** que d'utiliser un pointeur errant : si vous avez de la chance, le programme abandonne. Si vous n'en avez pas, le programme continue mais avec une faille de sécurité. **Rust** détecte ces erreurs au moment de la compilation.*
>
> *Le **C++** répond au premier de ces deux besoins puisqu'il suffit d'utiliser **free** ou **delete** sur un objet qui a été stocké dans la zone du **tas** de façon dynamique, quand vous le désirez. Mais cela se fait au détriment de la seconde attente : c'est à vous de garantir que vous n'allez plus ensuite utiliser un pointeur vers la valeur que vous venez de supprimer. Les expériences ne manquent pas pour montrer à quel point il est difficile de maintenir cette rigueur. L'utilisation erronée des pointeurs constitue le plus gros problème de sécurité système dans les langages **C** et **C++**.*

Comprendre la possession

La **possession** (**ownership**) est la fonctionnalité la plus remarquable de **Rust**, et elle permet de garantir la sécurité de la mémoire sans avoir besoin d'un ramasse-miettes (garbage collector comme Java). Par conséquent, il est important de comprendre comment la possession fonctionne en **Rust**. Durant cette étude, nous aborderons la **possession**, ainsi que d'autres fonctionnalités associées : **l'emprunt**, les **slices** et la façon dont **Rust** agence les données en mémoire.

> *Tous les programmes doivent gérer la façon dont ils utilisent la mémoire lorsqu'ils s'exécutent. Certains langages ont un ramasse-miettes qui scrute constamment la mémoire qui n'est plus utilisée pendant qu'il s'exécute ; dans d'autres langages, le développeur doit explicitement allouer et libérer la mémoire. **Rust** adopte une troisième approche : la mémoire est gérée avec un système de possession qui repose sur un jeu de règles que le compilateur vérifie au moment de la compilation. Aucune des fonctionnalités de possession ne ralentit votre programme à l'exécution.*
>
> *Toute valeur dans **Rust** utilise qu'un seul possédant qui détermine sa durée de vie. Lorsque le possédant est libéré, c'est-à-dire **largué** en terminologie **Rust** (**dropped**), la valeur possédée est **larguée** aussi. Cela vous permet de savoir quelle est la durée de vie par simple lecture du code source.*

> ***Dans cette étude, nous allons apprendre la possession avec plusieurs exemples qui se concentrent sur une structure de données très courante : les chaînes de caractères.***

La pile et le tas

Dans de nombreux langages, il n'est pas nécessaire de se préoccuper de la **pile** (stack) et du **tas** (heap). Mais dans un langage de programmation système comme **Rust**, qu'une donnée soit sur la **pile** ou sur le **tas** influe sur le comportement du langage et explique pourquoi nous devons faire certains choix. Nous décrirons plus loin dans ce chapitre comment la possession fonctionne vis-à-vis de la **pile** et du **tas**, voici donc une brève explication au préalable :

> *La **pile** et le **tas** sont tous les deux des emplacements de la mémoire qui sont à disposition de votre code lors de son exécution, mais sont organisés de façon différente. La **pile** enregistre les valeurs dans l'ordre qu'elle les reçoit et enlève les valeurs dans l'autre sens. C'est ce que nous appelons le principe **de dernier entré, premier sorti**. C'est comme une **pile** d'assiettes : quand vous ajoutez de nouvelles assiettes, vous les déposez sur le dessus de la **pile**, et quand vous avez besoin d'une assiette, vous en prenez une sur le dessus. Ajouter ou enlever des assiettes au milieu ou en bas ne serait pas aussi efficace ! Ajouter une donnée sur la **pile** se dit empiler et en retirer une se dit dépiler.*
>
> *Toutes les données stockées dans la **pile** doivent avoir une taille connue et fixe. Les **données avec une taille inconnue** au moment de la compilation ou une taille qui peut changer doivent plutôt être stockées sur le **tas**. Le **tas** est moins bien organisé : lorsque vous ajoutez des données sur le **tas**, vous demandez une certaine quantité d'espace mémoire. Le gestionnaire de mémoire va trouver un emplacement dans le **tas** qui est suffisamment grand, va le marquer comme étant en cours d'utilisation, et va retourner un **pointeur**, **qui est l'adresse de cet emplacement**. Cette procédure est appelée **allocation sur le tas**. L'ajout de valeurs sur la **pile** n'est pas considéré comme une allocation. Comme le pointeur a une taille connue et fixe, nous pouvons stocker ce **pointeur sur la pile**, mais quand nous voulons la **vraie valeur de donnée**, il faut suivre le pointeur.*

> ***Empiler sur la pile est plus rapide qu'allouer sur le tas car le gestionnaire ne va jamais avoir besoin de chercher un emplacement pour y stocker les nouvelles données ; il le fait toujours au sommet de la pile. En comparaison, allouer de la place sur le tas demande plus de travail, car le gestionnaire doit d'abord trouver un espace assez grand pour stocker les données et mettre à jour son suivi pour préparer la prochaine allocation.***

> *Quand notre code utilise une fonction, les valeurs passées à la fonction (incluant, potentiellement, des pointeurs de données sur le tas) et les variables locales à la fonction sont déposées sur la **pile**. Quand l'utilisation de la fonction est terminée, ces données sont retirées de la **pile**.*

Les règles de la possession, portée d'une variable

Tout d'abord, définissons les règles de la **possession**. Gardez à l'esprit ces règles pendant que nous travaillons sur des exemples qui les illustrent :

> - *Chaque valeur en **Rust** possède une variable qui est son propriétaire.*
> - *Il ne peut y avoir qu'un seul propriétaire à la fois.*
> - *Quand le propriétaire sort de la portée, la valeur est supprimée.*

Pour le premier exemple de **possession**, nous allons analyser la portée de certaines variables. Une portée est une zone dans un programme dans laquelle un élément est en vigueur. Imaginons que nous ayons la variable suivante :

Portée des variables

```
fn main() {
...
   {                    // s n'est pas encore connue ici, elle n'est pas encore déclarée
      let s = "hello"; // s est en vigueur à partir de ce point

                        // nous exploitons s ici
   }                    // cette portée est maintenant terminée, et s n'est plus en vigueur
}
```

*La variable **s** fait référence à un littéral de chaîne de caractères, où la valeur de la chaîne est codée en dur dans notre programme. La variable est en vigueur à partir du moment où elle est déclarée jusqu'à la fin de la portée actuelle. Autrement dit ici nous avons deux étapes importantes :*

- *Quand **s** rentre dans la portée, elle est en vigueur.*
- *Cela reste ainsi jusqu'à ce qu'elle sort de la portée.*

Pour le moment, la relation entre les portées et les conditions pour lesquelles les variables sont en vigueur sont similaires à d'autres langages de programmation. Maintenant, nous allons plus loin en exploitant plutôt le type String.

Le type String

La plupart des types que nous avons vus précédemment sont tous stockés sur la **pile** et sont retirés de la **pile** quand ils sortent de la portée, mais nous voulons expérimenter le stockage de données sur le **tas** et découvrir comment **Rust** sait quand il doit nettoyer ces données.

*Nous avons déjà rencontrés deux types de données qui utilisent le **tas** : les vecteurs **Vec<T>** et les chaînes de type **String**. Ces deux familles sont reconnaissables dans le fait qu'elles possèdent une capacité variable et qu'elles ont besoins de méthodes statiques comme **new()** et **from()**.*

Nous avons déjà vu les littéraux de chaînes de caractères, quand une valeur de chaîne est codée en dur dans notre programme. Les littéraux de chaînes sont pratiques, mais ils ne conviennent pas toujours à tous les cas où nous désirons utiliser du texte.

*Une des raisons est qu'ils sont **immuables**. Une autre raison est que nous ne connaissons pas forcément le contenu des chaînes de caractères quand nous écrivons notre code : par exemple, comment faire si nous voulons récupérer du texte saisi par l'utilisateur et l'enregistrer ?*

*Pour ces cas-ci, comme nous l'avons vu dans l'étude précédente, **Rust** possède un second type de chaîne de caractères, **String**. Ce type est alloué sur le **tas** et est ainsi capable de stocker une quantité de texte qui nous est inconnue au moment de la compilation. Vous pouvez créer une **String** à partir d'un littéral de chaîne de caractères en utilisant la méthode statique **from()**, avec l'opérateur **::** comme ceci :*

La chaîne String

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

*Ce type de chaîne de caractères peut être **mutable** :*

La chaîne String

```
fn main() {
   let mut s = String::from("Bonjour");
   s.push_str(", tout le monde!");         // Ajoute un littéral de chaîne dans un String
   println!("{}", s);                      // Affichage de `Bonjour, tout le monde!`
}
```

*Quelle est la différence ici ? Pourquoi **String** peut être **mutable**, et pourquoi les littéraux de chaînes ne peuvent pas l'être ? La différence se trouve dans la façon dont ces deux types de chaînes travaillent avec la mémoire.*

Mémoire et allocation

Dans le cas d'un littéral de chaîne de caractères, nous connaissons le contenu au moment de la compilation donc le texte est codé en dur directement dans l'exécutable final. Voilà pourquoi ces littéraux de chaînes de caractères sont performants et rapides. Mais ces caractéristiques viennent de leur **immuabilité**.

Malheureusement, nous ne pouvons pas accorder une grosse région de mémoire dans le binaire pour chaque morceau de texte qui n'a pas de taille connue au moment de la compilation et dont la taille pourrait changer pendant l'exécution de ce programme.

Avec le type **String**, pour nous permettre d'avoir un texte **mutable** et qui peut **s'agrandir**, nous devons allouer une quantité de mémoire sur le **tas**, inconnue au moment de la compilation, pour stocker le contenu.

Le tas est justement prévu pour stocker des variables dont la capacité évolue au cours du temps. C'est sa raison d'être. Cela signifie que :

• La mémoire doit être demandée auprès du gestionnaire de mémoire lors de l'exécution.

*• Nous avons besoin d'un moyen de rendre cette mémoire au gestionnaire lorsque nous aurons fini d'utiliser notre **String**.*

*Pour ce premier point : quand nous appelons **String::from()**, son implémentation demande la mémoire dont elle a besoin. C'est pratiquement toujours ainsi dans la majorité des langages de programmation.*

*Cependant, le deuxième point est différent. Dans des langages avec un **ramasse-miettes**, le système surveille et nettoie la mémoire qui n'est plus utilisée, sans que nous n'ayons à nous en préoccuper. Sans un **ramasse-miettes**, c'est de notre responsabilité d'identifier quand cette mémoire n'est plus utilisée et d'appeler du code pour explicitement la libérer, comme nous l'avons fait pour la demander auparavant.*

Historiquement, faire ceci correctement a toujours été une difficulté pour les développeurs. Si nous oublions de le faire, nous allons gaspiller de la mémoire. Si nous le faisons trop tôt, nous allons avoir une variable invalide. Si nous le faisons deux fois, cela produit aussi un bogue. Nous devons associer exactement un delete avec un free.

Rust prend un chemin différent : la mémoire est **automatiquement libérée** dès que la variable qui la possède **sort de la portée**. Voici une version de notre exemple de portée qui utilise une **String** plutôt qu'un littéral de chaîne de caractères :

La chaîne String

```
fn main() {
    {
        let s = String::from("hello");  // s est en vigueur à partir de ce point
                                        // nous exploitons s ici
    }                                   // cette portée est désormais terminée, et s n'est plus en vigueur maintenant
}
```

*Il existe un moment naturel où nous devons rendre la mémoire de notre chaîne **String** au gestionnaire : quand **s** sort de la portée. Quand une variable sort de la portée, **Rust** appelle une fonction spéciale pour nous. Cette fonction s'appelle **drop**, et c'est dans celle-ci que l'auteur de **String** a pu mettre le code pour libérer la mémoire. **Rust** appelle automatiquement **drop** à l'accolade fermante **}**.*

Cette façon de faire a un impact profond sur la façon dont le code Rust est écrit. Cela peut sembler simple dans notre cas, mais le comportement du code peut être surprenant dans des situations plus compliquées où nous désirons avoir plusieurs variables utilisant les mêmes données que nous avons affectées sur le tas. Examinons une de ces situations dès à présent.

Les interactions entre les variables et les données : le déplacement

Plusieurs variables peuvent interagir avec les mêmes données de différentes manières en **Rust**. Regardons un exemple avec un entier

Déclaration de variables immuables (constantes)

```
fn main() {
    let x = 5;
    let y = x;
}
```

*Nous pouvons probablement deviner ce que se passe : "Assigner la valeur **5** à **x** ; ensuite faire une copie de cette valeur de **x** et l'assigner à **y.**" Nous disposons maintenant de deux variables, **x** et **y,** et chacune vaut **5.** C'est effectivement ce qui se passe, car les entiers sont des valeurs simples avec une taille connue et fixée, et ces deux valeurs **5** sont stockées sur la pile.*

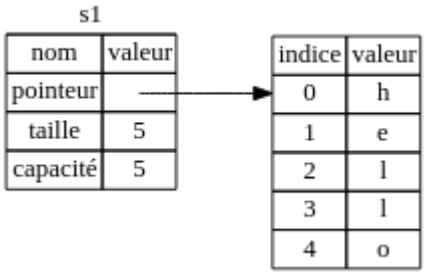
Essayons la même principe, mais cette fois-ci avec des types String :

Déclaration de variables String

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
}
```

*Cela ressemble beaucoup au code précédent, et nous pouvons supposer que cela fonctionne de la même manière que précédemment : ainsi, nous pensons que la seconde ligne va faire une copie de la valeur de **s1** et l'assigner à **s2**. Mais ce n'est pas tout à fait ce qu'il se passe.*

*Vous avez ci-contre la représentation en mémoire de la chaîne de type **String** qui contient la valeur **"hello"** assignée à **s1**.*



*La taille est la quantité de mémoire, en octets, que le contenu de la chaîne **String** utilise actuellement. La capacité est la quantité totale de mémoire, en octets, que la chaîne **String** a reçue du gestionnaire.*

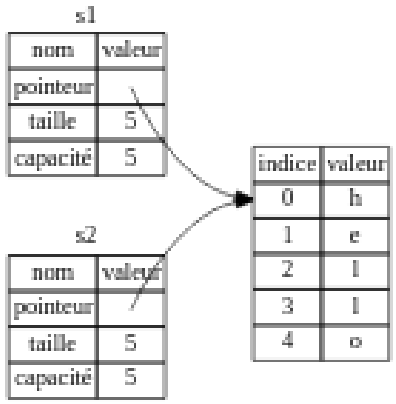
***Lorsque** nous assignons **s1** à **s2**, les données de la chaîne **String** sont copiées, ce qui veut dire que nous copions le pointeur, la taille et la capacité qui sont stockés sur la **pile**. Nous ne copions pas les données stockées sur le **tas** auxquelles le pointeur se réfère. Autrement dit, la représentation des données dans la mémoire ressemble à l'illustration de la page suivante :*

*Vous avez ci-contre la représentation en mémoire de la variable **s2** qui possède une copie du pointeur, de la taille et de la capacité de **s1**.*

*Si **Rust** avait aussi copié les données sur le **tas** l'opération **s2 = s1** pourrait potentiellement être **très coûteuse** en termes de performances d'exécution si les données sur le **tas** étaient relativement volumineuses.*

*Précédemment, nous avons dit que quand une variable sortait de la portée, **Rust** appelait automatiquement la fonction **drop** et nettoyait la mémoire sur le **tas** allouée pour cette variable. Mais l'illustration ci-dessus montre que les deux pointeurs de données pointent au même endroit. C'est un problème : quand **s2** et **s1** sortent de la portée, elles vont essayer toutes les deux de libérer la même mémoire.*

*C'est ce que nous appelons une erreur de **double libération** et c'est un des bogues de sécurité de mémoire que nous avons mentionnés précédemment. Libérer la mémoire deux fois peut mener à des corruptions de mémoire, ce qui peut potentiellement mener à des vulnérabilités de sécurité.*



Pour garantir la sécurité de la mémoire, il y a un autre petit détail qui se produit dans cette situation avec Rust. Plutôt qu'essayer de copier la mémoire allouée, Rust considère que s1 n'est plus en vigueur et donc, Rust n'a pas besoin de libérer quoi que ce soit lorsque s1 sort de la portée.

*Regardez ce qu'il se passe lorsque vous essayez d'utiliser **s1** après que **s2** ait été créée, cela ne va pas fonctionner :*

Déclaration de variables mutables

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    println!("{}, world!", s1);
}
```

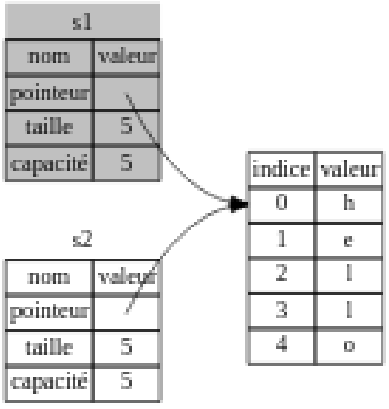
*Vous allez avoir une erreur comme celle ci-dessous, car **Rust** vous défend d'utiliser la référence qui n'est plus en vigueur :*

Résultat de la compilation

```
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
```

*Si vous avez déjà entendu parler de copie superficielle et de copie profonde en utilisant d'autres langages, l'idée de copier le pointeur, la taille et la capacité sans copier les données peut vous faire penser à de la copie superficielle. Mais comme **Rust** neutralise aussi la première variable, au lieu d'appeler cela une copie superficielle, on appelle cela un **déplacement**. Ici, nous pourrions dire que **s1 a été déplacé dans s2**. Donc ce qui se passe réellement est décrit par l'illustration ci-contre :*

*Cela résout notre problème ! Avec seulement **s2** en vigueur, quand elle sortira de la portée, elle seule va libérer la mémoire, et c'est tout. De plus, cela signifie que nous avons un nouveau choix de conception : **Rust** ne va jamais créer automatiquement de copie "profonde" de vos données. Par conséquent, toute copie automatique peut être considérée comme peu coûteuse en termes de performances d'exécution.*



Les interactions entre les variables et les données : le clonage

Si nous désirons faire une copie profonde des données sur le **tas** d'une chaîne **String** complète, et pas seulement les données associées sur la **pile**, nous pouvons utiliser une méthode commune à tous les classes dont les données sont stockées sur le **tas** et qui se nomme ***clone()***.

*Voici un exemple d'utilisation de la méthode **clone()** :*

Clonage de chaînes

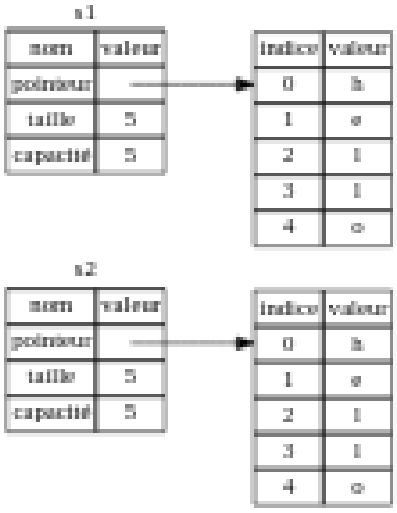
```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1.clone();

    println!("s1 = {}, s2 = {}", s1, s2);
}
```

Résultat

```
s1 = hello, s2 = hello
```

*Cela fonctionne très bien et c'est ainsi que vous pouvez reproduire le comportement décrit dans l'illustration ci-contre, où les données du **tas** sont copiées.*

*Lorsque vous voyez un appel à **clone()**, vous savez que ce code peut être coûteux en terme de performance.*



Données uniquement sur la pile : la copie

Il y a un autre détail dont nous n'avons pas encore parlé. Le code suivant utilise des entiers et correspond à priori au même comportement, sans la méthode **clone()** :

Copie d'entiers

```
fn main() {
    let x = 5;
    let y = x;

    println!("x = {}, y = {}", x, y);
}
```

Résultat

```
x = 5, y = 5
```

*Ce code semble contredire ce que nous venons d'apprendre dans le chapitre précédent : nous n'avons pas appelé la méthode **clone()**, **x** est toujours en vigueur et n'a pas été déplacé dans **y**.*

*La raison est que les types comme les **entiers** ont une **taille connue** au moment de la compilation et sont **entièrement stockés sur la pile**, donc la copie des vraies valeurs est rapide à faire. Cela signifie qu'il n'y a aucune raison de neutraliser **x** après avoir créé la variable **y**. En d'autres termes, il n'y a ici aucune différence entre la copie superficielle et profonde, donc appeler **clone()** ne ferait rien d'autre qu'une copie superficielle classique et nous pouvons nous en passer.*

***Rust** possède une annotation spéciale appelée le **trait Copy** que nous pouvons utiliser sur des types comme les **entiers** qui sont stockés sur la **pile** (nous verrons les **traits** dans une autre étude). Si un type implémente le **trait Copy**, l'ancienne variable sera toujours utilisable après avoir été affectée. Voici quelques types qui implémentent **Copy** :*

- *Tous les types **d'entiers**, comme **u32**.*
- *Le type **booléen**, **bool**, avec les valeurs **true** et **false**.*
- *Tous les types de **flottants**, comme **f64**.*
- *Le type de **caractère**, **char**.*

La possession et les fonctions

La syntaxe pour passer une valeur à une fonction est similaire à celle pour assigner une valeur à une variable. Passer une variable à une fonction (que nous étudierons dans une autre étude) va la déplacer ou la copier, comme l'assignation. Le code ci-dessous nous montrent où les variables rentrent et sortent de la portée :

Portée des variables

```
fn main() {
    let s = String::from("hello");  // s rentre dans la portée.

    prendre_possession(s);          // La valeur de s est déplacée dans la fonction…
                                    // … et n'est plus en vigueur à partir d'ici

    let x = 5;                      // x rentre dans la portée.

    creer_copie(x);                 // x va être déplacée dans la fonction,
                                    // mais i32 est Copy, donc nous pouvons utiliser x ensuite.

}                                   // Ici, x sort de la portée, puis ensuite s. Mais puisque la valeur de s a
                                    // été déplacée, il ne se passe rien de spécial.

fn prendre_possession(texte: String) {  // texte rentre dans la portée.
    println!("{}", texte);
}                                   // Ici, texte sort de la portée et `drop` est appelé. La mémoire est libérée.
fn creer_copie(entier: i32) {       // entier rentre dans la portée.
    println!("{}", entier);
}                                   // Ici, entier sort de la portée. Il ne se passe rien de spécial.
```

*Si nous essayons d'utiliser **s** après l'appel à **prendre_possession()**, **Rust** déclencherait une erreur à la compilation. Ces vérifications statiques nous protègent des erreurs. Essayez d'ajouter du code à la fonction **main()** qui utilise **s** et **x** pour découvrir lorsque vous pouvez les utiliser et lorsque les règles de la possession vous empêchent de le faire.*

Les valeurs de retour et les portées

Retourner des valeurs peut aussi transférer leur **possession**. La **possession** d'une variable suit toujours le même schéma à chaque fois : assigner une valeur à une autre variable la déplace. Quand une variable qui contient des données sur le

tas sort de la portée, la valeur sera nettoyée avec **drop** à moins que la donnée ait été déplacée pour être possédée par une autre variable.

Opérations sur des types différents

```
fn main() {
    let s1 = donne_possession();    // donne_possession() déplace sa valeur de retour dans s1

    let s2 = String::from("hello");  // s2 rentre dans la portée

    let s3 = prend_et_rend(s2);      // s2 est déplacée dans prend_et_rend(), qui elle aussi
                                     // déplace sa valeur de retour dans s3.
}                                    // Ici, s3 sort de la portée et est éliminée. s2 sort de la portée mais a été
                                     // déplacée donc il ne se passe rien. s1 sort aussi de la portée et est éliminée.

fn donne_possession() -> String {        // donne_possession() va déplace sa valeur de retour dans la
                                         // fonction qui l'appelle.

    let texte = String::from("hello");       // texte rentre dans la portée.
    texte                                // texte est retournée et est déplacée vers la fonction principale qui l'appelle.
}

// la fonction prend_et_rend() prend une chaîne de type String et la retourne.
fn prend_et_rend(texte: String) -> String {       // texte rentre dans la portée.

    texte                        // texte est retournée et déplacée vers la fonction principale qui l'appelle.
}
```

Il est un peu fastidieux de prendre la possession puis ensuite de retourner la possession à chaque fonction. Et qu'est-ce qu'il se passe si nous désirons qu'une fonction utilise une valeur, mais n'en prenne pas possession ?

*C'est assez pénible que tout ce que nous passons doit être retourné si nous voulons l'utiliser à nouveau, en plus de toutes les données qui découlent du corps de la fonction que nous voulons aussi récupérer. Il est possible de retourner plusieurs valeurs à l'aide d'un **tuple**, comme ceci :*

Retourner la possession des paramètres

```
fn main() {
    let s1 = String::from("hello");
    let (s2, taille) = calculer_taille(s1);
    println!("La taille de '{}' est {}.", s2, taille);
}

fn calculer_taille(s: String) -> (String, usize) {
    let taille = s.len();                                  // len() retourne la taille d'une chaîne de type String.
    (s, taille)
}
```

Résultat

```
La taille de 'hello' est 5.
```

*Mais c'est trop laborieux et beaucoup de travail pour un principe qui devrait être banal. Heureusement pour nous, **Rust** possède une fonctionnalité pour ce principe, c'est ce que nous appelons les **références** (découvertes dans l'étude précédente).*

Les références et l'emprunt

La difficulté avec le code précédent est que nous avons besoin de retourner la chaîne **String** au code appelant dans la fonction principale pour qu'il puisse continuer à utiliser cette chaîne **String** après l'appel de la fonctions ***calculer_taille()***, car elle a été déplacée à l'intérieur de cette fonction. Cette méthode me paraît très compliqué et fastidieuse à maîtriser pour un résultat somme toute assez simple et intuitif.

*Voyons comment modifier cette fonction **calculer_taille()** qui prend une **référence** à un objet en paramètre plutôt que de prendre possession de la valeur pour elle soit beaucoup plus intuitive à utiliser.*

Utiliser les références en paramètres de fonctions

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

    let long = calculer_taille(&s1);

    println!("La taille de '{}' est {}.", s1, long);
}

fn calculer_taille(s: &String) -> usize { // s est une référence à une chaîne String
    s.len()
}                                         // Ici, s sort de la portée. Mais comme elle ne prend pas possession de ce
                                          // à quoi elle fait référence, il ne se passe rien.
```

Résultat

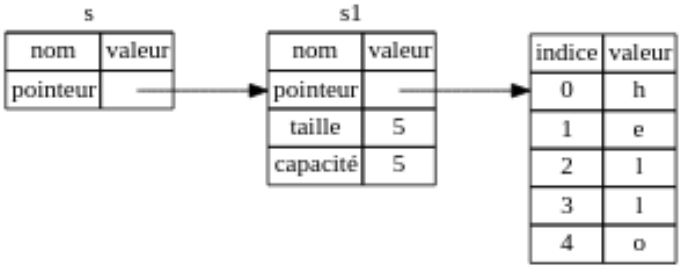
```
La taille de 'hello' est 5.
```

*Premièrement, nous pouvons bien sûr observer que tout le code des **tuples** dans la déclaration des variables et dans la valeur de retour de la fonction a été enlevé. Deuxièmement, remarquez que nous passons **&s1** à **calculer_taille()**, et que dans sa définition, nous utilisons **&String** plutôt que **String**.*

Ces esperluettes sont des références, et elles permettent de vous référer à une valeur sans en prendre possession.

*La syntaxe **&s1** nous permet de créer une **référence** qui se réfère à la valeur de **s1** sans en prendre la possession. Comme elle ne la possède pas, la valeur vers laquelle elle pointe ne sera pas libérée quand cette **référence** sortira de la portée.*

*De la même manière, la signature de la fonction utilise **&** pour indiquer que le type du paramètre **s** est une **référence**.*



*La portée dans laquelle la variable **s** est en vigueur est la même que toute portée d'un paramètre de fonction, mais nous ne libérons pas ce sur quoi cette **référence** pointe quand elle sort de la portée, car nous n'en prenons pas possession. Lorsque les fonctions ont des **références** en paramètres au lieu des valeurs réelles, nous n'avons pas besoin de retourner les valeurs pour les rendre, **car nous n'en avons jamais pris possession**.*

*Lorsque nous avons des **références** dans les paramètres d'une fonction, nous appelons cela **l'emprunt**. Comme dans la vie réelle, quand un objet appartient à quelqu'un, vous pouvez le lui **emprunter**. Et quand vous avez fini, vous devez le lui **rendre**. Que se passe-t-il lorsque nous essayons de modifier une valeur que nous **empruntons** ?*

Tentative de modification d'une valeur empruntée.

```
fn main() {
    let s = String::from("hello");
    changer(&s);
}

fn changer(texte: &String) {
    texte.push_str(", world");
}
```

Résultat

```
  |
7 | fn changer(texte: &String) {
  |                   ------- help: consider changing this to be a mutable reference: `&mut String`
8 |     texte.push_str(", world");
  |     ^^^^^ `texte` is a `&` reference, so the data it refers to cannot be borrowed as mutable
```

*Comme les variables sont **immuables par défaut**, les **références** le sont aussi. Nous ne sommes pas autorisés à modifier une variable lorsque nous avons une **référence** vers elle.*

Les références mutables

Nous pouvons résoudre l'erreur du code précédent avec une petite modification. D'abord, nous devons préciser que **s** est **mutable**. Ensuite, nous devons créer une **référence mutable** avec **&mut s** et accepter de prendre une **référence mutable** avec **texte**: **&mut String**.

Référence mutable

```
fn main() {
    let mut s = String::from("hello");
    changer(&mut s);
}

fn changer(texte: &mut String) {
    texte.push_str(", world");
}
```

*Mais les **références mutables** ont une grosse contrainte : vous ne pouvez avoir qu'une seule **référence mutable** pour chaque donnée dans chaque portée. Le code suivant va échouer :*

Références mutables

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

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

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

Résultat

```
4 |    let r1 = &mut s;
  |             ------ first mutable borrow occurs here
5 |    let r2 = &mut s;
  |             ^^^^^^ second mutable borrow occurs here
6 |
7 |    println!("{}, {}", r1, r2);
  |                        -- first borrow later used here
```

Cette contrainte autorise les mutations, mais de manière très contrôlée. C'est quelque chose que les nouveaux ont du mal à surmonter cette particularité, car la plupart des langages vous permettent de modifier les données quand vous le désirez.
*L'avantage d'avoir cette contrainte est que **Rust** peut empêcher les accès concurrents au moment de la compilation. Un accès concurrent est une situation qui se produit lorsque ces trois facteurs se combinent :*

- *Deux pointeurs ou plus accèdent à la même donnée au même moment.*
- *Au moins un des pointeurs est utilisé pour écrire dans cette donnée.*
- *Nous n'utilisons aucun mécanisme pour synchroniser l'accès aux données.*

*L'accès concurrent provoque des comportements indéfinis et rend difficile le diagnostic et la résolution de problèmes lorsque vous essayez de les reproduire au moment de l'exécution ; **Rust** évite ce problème parce qu'il ne va pas compiler du code avec des accès concurrents ! Une règle similaire existe pour combiner les **références immuables** et **mutables**. Ce code va mener à une erreur :*

Références immuables et mutables

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

    let r1 = &s;          // sans problème
    let r2 = &s;          // sans problème
    let r3 = &mut s;      // GROS PROBLÈME

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

Résultat

```
4 |    let r1 = &s; // sans problème
  |             -- immutable borrow occurs here
5 |    let r2 = &s; // sans problème
6 |    let r3 = &mut s; // GROS PROBLEME
  |             ^^^^^^ mutable borrow occurs here
7 |
8 |    println!("{}, {}, and {}", r1, r2, r3);
  |                                -- immutable borrow later used here$ cargo run
```

*Nous ne pouvons pas non plus avoir une référence **mutable** pendant que nous en avons une autre **immuable**. Les utilisateurs d'une référence **immuable** ne s'attendent pas à ce que sa valeur change soudainement ! Cependant, l'utilisation de **plusieurs références immuables** ne pose pas de problème, car simplement lire une donnée ne va pas affecter la lecture de la donnée par les autres.*
*Notez bien que la portée d'une **référence** commence dès qu'elle est introduite et se poursuit jusqu'au dernier endroit où cette référence est utilisée. Par exemple, le code suivant va se compiler car la dernière utilisation de la référence **immuable** est située avant l'introduction de la référence **mutable** :*

Références immuables et mutables

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

    let r1 = &s;                    // sans problème
    let r2 = &s;                    // sans problème
    println!("{} et {}", r1, r2);
                                    // r1 et r2 ne sont plus utilisés à partir d'ici

    let r3 = &mut s;                // sans problème
    println!("{}", r3);
}
```

*Les portées des références immuables **r1** et **r2** se terminent après le **println!()** où elles sont utilisées pour la dernière fois, c'est-à-dire avant que la référence mutable **r3** soit créée. Ces portées ne se chevauchent pas, donc ce code est autorisé.*
*Même si ces erreurs **d'emprunt** peuvent parfois être frustrantes, n'oubliez pas que le compilateur de **Rust** nous signale un bogue potentiel en avance (au moment de la compilation plutôt que l'exécution) et vous montre où se situe exactement le problème. Ainsi, vous n'avez pas à chercher pourquoi vos données ne correspondent pas à ce que vous pensiez qu'elles devraient être.*

Les références pendouillantes

Avec les langages qui utilisent les pointeurs, il est facile de créer par erreur un pointeur pendouillant (dangling pointer), qui est un pointeur qui pointe vers un emplacement mémoire qui a été donné à quelqu'un d'autre, en libérant la mémoire tout en conservant l'accès vers cette mémoire.

*En revanche, avec **Rust**, le compilateur garantit que les références ne seront jamais des **références pendouillantes** : si nous avons une **référence** vers une donnée, le compilateur va s'assurer que cette donnée ne va pas **sortir de la portée** avant que la **référence** vers cette donnée en soit elle-même sortie.*

*Essayons de créer une **référence pendouillante**, ce que **Rust** va empêcher avec une erreur au moment de la compilation :*

Référence pendouillante

```
fn main() {
    let reference_vers_rien = pendouille();
}

fn pendouille() -> &String {      // la fonction pendouille retourne une référence vers une String

    let s = String::from("hello");  // s est une nouvelle String
    &s                              // nous retournons une référence vers la String s
}                                   // Ici, s sort de la portée, et est libéré.
                                    // Sa mémoire disparaît. Attention, danger !
```

Résultat

```
5 | fn pendouille() -> &String {
  |                    ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from
    help: consider using the `'static` lifetime
  |
5 | fn pendouille() -> &'static String {
  |                     ^^^^^^
```

*Ce message d'erreur fait référence à une fonctionnalité que nous n'avons pas encore vue : **les durées de vie**. Si vous mettez de côté les parties qui parlent de durées de vie, le message explique pourquoi le code pose problème, qui peut se traduire par :*

Le type de retour de cette fonction contient une valeur empruntée, mais il n'existe plus aucune valeur qui peut être empruntée.

*Comme **s** est créé dans la fonction **pendouille()**, lorsque le code de **pendouille()** est terminé, la variable **s** est désallouée. Mais nous avons essayé de retourner une référence vers elle. Cela veut dire que cette référence va pointer vers une **String** invalide. Ce n'est pas bon ! **Rust** ne nous laisse pas faire cela. Ici la solution est de renvoyer la chaîne **String** directement :*

Références ne pendouille plus

```
fn main() {
    let string = ne_pendouille_pas();
}

fn ne_pendouille_pas() -> String {
    let s = String::from("hello");
    s
}
```

Cela fonctionne sans problème. La possession est transférée à la valeur de retour de la fonction, et rien n'est désalloué.

Les règles de référencement

R écapitulons ce que nous avons vu à propos des **références** : *À un instant donné, vous pouvez avoir*

- *Soit une **référence mutable**, soit un nombre quelconque de **références immuables**.*
- *Soit un nombre quelconque de **références immuables**.*
- *Les **références** doivent toujours être en vigueur..*

Le type slice

Un autre type de donnée qui ne prend pas possession est le type **slice**. Une **slice** vous permet d'obtenir une **référence** vers une **séquence continue** d'éléments d'une collection plutôt que **toute la collection**.

Voici un petit problème de programmation : écrire une fonction qui prend une chaîne de caractères et retourne le premier mot qu'elle trouve dans cette chaîne. Si la fonction ne trouve pas d'espace dans la chaîne, cela veut dire que la chaîne est en un seul mot, donc la chaîne en entier doit être retournée.

Fonction qui renvoi l'indice de la fin du premier mot

```
fn premier_mot(s: &String) -> usize {
    let octets = s.as_bytes();
    for (i, &element) in octets.iter().enumerate() {
        if element == b' ' {
            return i;
        }
    }
    s.len()
}
```

*Cette fonction, **premier_mot()**, prend un **&String** comme paramètre. Nous ne voulons pas en prendre possession, donc c'est ce qu'il nous faut. Mais que devons-nous retourner ? Nous n'avons aucun moyen de désigner une partie d'une chaîne de caractères. Cependant, nous pouvons retourner l'indice de la fin du mot.*

*Comme nous avons besoin de parcourir la chaîne **String** élément par élément et de vérifier si la valeur est un espace, nous convertissons notre **String** en un **tableau d'octets** en utilisant la méthode **as_bytes()**. Ensuite, nous créons un itérateur sur le tableau d'octets en utilisant la méthode **iter()**.*

*Nous aborderons plus en détail les itérateurs lors d'une prochaine étude. Pour le moment, sachez que **iter()** est une méthode qui retourne chaque élément d'une collection, et que **enumerate()** transforme le résultat de **iter()** pour retourner plutôt chaque élément à l'aide d'un **tuple**. Le premier élément du **tuple** retourné par **enumerate()** est l'indice, et le second élément est une **référence** vers l'élément. C'est un peu plus pratique que de calculer les indices par nous-mêmes.*

*Comme la méthode **enumerate()** retourne un **tuple**, nous pouvons utiliser des motifs pour déstructurer ce **tuple**, comme nous pourrions le faire n'importe où avec **Rust**. Donc dans la boucle **for**, nous précisons un motif qui indique que nous définissons **i** pour l'indice au sein du **tuple** et **&element** pour l'octet dans le **tuple**. Comme nous obtenons une **référence** vers l'élément avec **iter().enumerate()**, nous utilisons **&** dans le motif.*

*Au sein de la boucle **for**, nous recherchons l'octet qui représente l'espace en utilisant la syntaxe de littéral d'octet. Si nous trouvons un espace, nous retournons sa position. Sinon, nous retournons la taille de la chaîne en utilisant **s.len()**.*

*Nous avons maintenant une façon de trouver l'indice de la fin du premier mot dans la chaîne de caractères, mais il y a un problème. Nous retournons un **usize** tout seul, mais il n'a du sens que lorsqu'il est lié au **&String**. Autrement dit, comme il a une valeur séparée de la **String**, il n'y a pas de garantie qu'il restera toujours valide dans le futur. Imaginons le programme suivant :*

Exemple d'utilisation de la fonction premier_mot()

```
fn main() {
    let mut s = String::from("hello world");
    let mot = premier_mot(&s);  // la variable mot aura 5 comme valeur.
    s.clear();                  // ceci vide la chaîne String, elle vaut maintenant "".
                                // mot a toujours la valeur 5 ici, mais il n'y a plus de chaîne qui donne
                                // du sens à la valeur 5. mot est maintenant complètement invalide !
}
```

*Ce programme principal se compile sans aucune erreur et le ferait toujours si nous utilisions **mot** après avoir appelé **s.clear()**. Comme **mot** n'est pas du tout lié à **s**, **mot** contient toujours la valeur **5**. Nous pourrions utiliser cette valeur **5** avec la variable **s** pour essayer d'en extraire le premier mot, mais cela serait un bogue, car le contenu de **s** a changé depuis que nous avons enregistré **5** dans **mot**.*

*Se préoccuper en permanence que l'indice présent dans **mot** ne soit plus synchronisé avec les données présentes dans **s** est fastidieux et source d'erreur ! La gestion de ces indices est encore plus risquée si nous écrivons une fonction **second_mot()**. Sa signature ressemblerait à ceci :*

```
fn second_mot(s: &String) -> (usize, usize) {
```

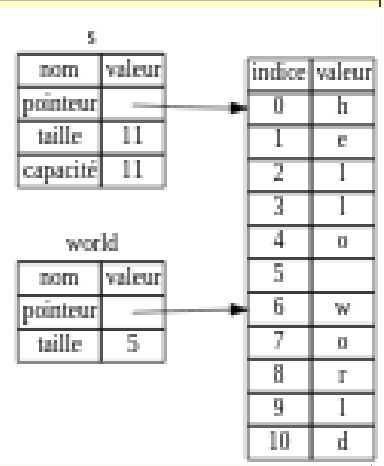
*Maintenant, nous avons un indice de début et un indice de fin, donc nous avons encore plus de valeurs qui sont calculées à partir d'une donnée dans un état donné, mais qui ne sont pas liées du tout à l'état de cette donnée. Nous avons maintenant trois variables isolées qui ont besoin d'être maintenues à jour. Heureusement, **Rust** a une solution pour ce problème : les **slices** de chaînes de caractères.*

Les slices de chaînes de caractères

Un **slice** de chaîne de caractères (ou **slice** de chaîne) est une **référence** à une partie d'une chaîne de caractères de type **String**, et ressemble à ceci :

Les slices de chaîne de caractères

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

    let hello = &s[0..5];
    let world = &s[6..11];
}
```



*Cela ressemble à une **référence** pour toute la chaîne **String**, mais avec la partie **[0..5]** en plus. Plutôt que d'être une **référence** vers toute la chaîne **String**, c'est une **référence** vers une **partie** de la chaîne **String**.*

*Nous pouvons créer des **slices** en utilisant un intervalle entre crochets en spécifiant **[indice_debut..indice_fin]**, où **indice_debut** est la position du premier octet de la **slice** et **indice_fin** est la position juste après le dernier octet de la **slice**.*

*En interne, la structure de données de la **slice** stocke la position de départ et la longueur de la slice, ce qui correspond à **indice_fin** moins **indice_debut**. Donc dans le cas de **let world = &s[6..11];**, **world** est une **slice** qui contient un pointeur vers le septième octet (en comptant à partir de 1) de **s** et une longueur de **5**.*

*Avec la syntaxe d'intervalle **..** de **Rust**, si vous voulez commencer au premier indice (zéro), vous pouvez ne rien mettre avant les deux points. Autrement dit, ces deux cas sont identiques :*

Les slices de chaîne de caractères

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

    let slice = &s[0..2];
    let slice = &s[..2];
}
```

De la même manière, si votre **slice** contient le dernier octet de la chaîne **String**, vous pouvez ne rien mettre à la fin. Cela veut dire que ces deux cas sont identiques :

Les slices de chaîne de caractères

```
fn main() {
    let s = String::from("hello");
    let taille = s.len();
    let slice = &s[3..taille];
    let slice = &s[3..];
}
```

Vous pouvez aussi ne mettre aucune limite pour créer une **slice** de toute la chaîne de caractères. Ces deux cas sont donc identiques :

Les slices de chaîne de caractères

```
fn main() {
    let s = String::from("hello");
    let taille = s.len();
    let slice = &s[0..taille];
    let slice = &s[..];
}
```

Maintenant que nous savons tout cela, essayons de réécrire **premier_mot()** pour qu'il retourne une **slice**. Le type pour les **slices** de chaînes de caractères s'écrit **&str** :

Retour sur la fonction premier_mot()

```
fn premier_mot(s: &String) -> &str {
    let octets = s.as_bytes();
    for (i, &element) in octets.iter().enumerate() {
        if element == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}
```

Nous récupérons l'indice de la fin du mot de la même façon que précédemment en cherchant la première occurrence d'un espace. Lorsque nous trouvons un espace, nous retournons une **slice** de chaîne en utilisant le début de la chaîne de caractères et l'indice de l'espace comme indices de début et de fin respectivement.

Désormais, quand nous appelons **premier_mot()**, nous récupérons une unique valeur qui est liée à la donnée de base. La valeur se compose d'une **référence** vers le point de départ de la **slice** et du nombre d'éléments dans la **slice**.

Nous disposons maintenant une API simple qui est bien plus difficile à mal utiliser, puisque le compilateur va s'assurer que les **références** dans la **String** seront toujours en vigueur. Vous souvenez-vous du bogue du programme précédent, lorsque nous avions un indice vers la fin du premier mot mais qu'ensuite nous avions vidé la chaîne de caractères et que notre indice n'était plus valide ?

Ce code était logiquement incorrect, mais ne montrait pas immédiatement une erreur. Les problèmes apparaîtront plus tard si nous essayons d'utiliser l'indice du premier mot avec une chaîne de caractères qui a été vidée. Les **slices** rendent ce bogue impossible et nous signalent bien plus tôt que nous avons un problème avec notre code. Utiliser la version avec la **slice** de **premier_mot()** va causer une erreur de compilation :

Référence pendouillante

```
fn main() {
    let mut s = String::from("hello world");
    let mot = premier_mot(&s);
    s.clear(); // Erreur !
    println!("Le premier mot est : {}", mot);
}
```

Résultat

```
16 |     let mot = premier_mot(&s);
   |                           -- immutable borrow occurs here
17 |
18 |     s.clear(); // Erreur !
   |     ^^^^^^^^^^ mutable borrow occurs here
19 |
```

```
20 |     println!("Le premier mot est : {}", mot);
   |                                          --- immutable borrow later used here
```

*Rappelons-nous que d'après les règles **d'emprunt**, si nous avons une **référence immuable** vers quelque chose, nous ne pouvons pas avoir une **référence mutable en même temps**. Étant donné que **clear()** a besoin de modifier la chaîne **String**, il a besoin d'une **référence mutable**. **Rust** interdit cette situation, et la compilation échoue. Non seulement **Rust** a simplifié l'utilisation de notre API, mais il a aussi éliminé une catégorie entière d'erreurs au moment de la compilation !*

Les littéraux de chaîne de caractères sont aussi des slices

Rappelez-vous lorsque nous avons appris que les littéraux de chaîne de caractères étaient enregistrés dans le binaire. Maintenant que nous connaissons les **slices**, nous pouvons désormais comprendre les littéraux de chaîne.

Les slices de chaîne de caractères

```
fn main() {
    let s = "Hello, world!";
}
```

*Ici, le type de **s** est un **&str** : c'est une **slice** qui pointe vers un endroit précis du binaire. C'est aussi la raison pour laquelle les littéraux de chaîne sont immuables ; **&str est une référence immuable**.*

Les slices de chaîne de caractères

```
fn premier_mot(s: &str) -> &str {
    let octets = s.as_bytes();
    for (i, &element) in octets.iter().enumerate() {
        if element == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}

fn main() {
    let ma_string = String::from("hello world");

    // premier_mot fonctionne avec les slices de `String`
    let mot = premier_mot(&ma_string[..]);
    let mon_litteral_de_chaine = "hello world";

    // premier_mot fonctionne avec les slices de littéraux de chaîne
    let mot = premier_mot(&mon_litteral_de_chaine[..]);

    // Comme les littéraux de chaîne *sont* déjà des slices de chaînes,
    // cela fonctionne aussi, sans la syntaxe de slice !
    let mot = premier_mot(mon_litteral_de_chaine);
}
```

*Savoir que nous pouvons utiliser des **slices** de **littéraux** et de **String** nous incite à apporter une petite amélioration à **premier_mot()**. Cela nous permet d'utiliser la même fonction sur les **&String** et aussi les **&str**.*

*Si nous avons une **slice** de **chaîne**, nous pouvons la passer en argument directement. Si nous avons une **String**, nous pouvons envoyer une **slice** de toute la **String**. Définir une fonction qui prend une **slice** de chaîne plutôt qu'une **référence** à une **String** rend notre API plus générique et plus utile sans perdre aucune fonctionnalité.*

Résumé

Les concepts de **possession**, **d'emprunt** et de **slices** garantissent la sécurité de la mémoire dans les programmes **Rust** au moment de la compilation. Le langage **Rust** vous donne le contrôle sur l'utilisation de la mémoire comme tous les autres langages de programmation système, mais le fait que celui qui possède des données nettoie automatiquement ces données quand il sort de la portée vous permet de ne pas avoir à écrire et déboguer du code en plus pour avoir cette fonctionnalité.

*La **possession** influe sur de nombreuses autres fonctionnalités de **Rust**, c'est pourquoi nous allons encore parler de ces concepts plus loin dans les différentes études.*