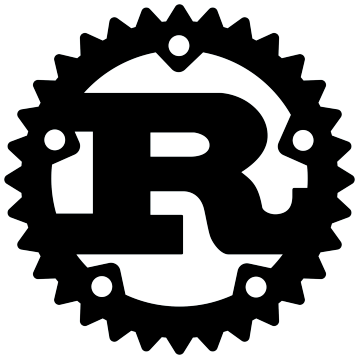
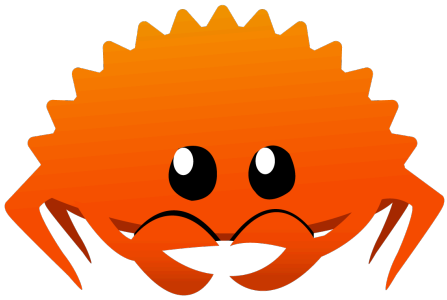
Rust





`https://rust-lang.org/`

1. Pourquoi rust ?

Toute première version en 2006, projet personnel de Graydon Hoare, chez Mozilla. Première version stable (1.0) en 2015.

Maintenant géré par la Rust Foundation, fondée et financée par Mozilla, Huawei, Google, AWS, Microsoft.

Commence à sérieusement être considéré comme une alternative à C/C++ : noyau Linux, composants windows, navigateurs, infrastructure web, etc.

Mais aussi dans des systèmes embarqués critiques : automobile, aviation, spatial,

Plusieurs agences gouvernementales US poussent aussi à l'adoption en masse du langage. En France, l'ANSSI (Agence Nationale de Sécurité des Systèmes d'Information) s'intéresse aussi de très près au langage.

Pourquoi ? Rust résoud enfin le dilemme entre langages de bas niveau (efficaces mais dangereux) et langages de haut niveau (plus sûrs mais moins performants).

C'est un langage de bas niveau, même s'il dispose de nombreuses abstractions qui permettent de garantir la sécurité des programmes sur plusieurs aspects.

Mais le langage est très complexe.

2. La mémoire

2.1. La mémoire statique

Ce sont des variables ou des constantes globales, définies directement dans le binaire de l'exécutable. Elles sont créées et accessibles (théoriquement) pour toute la durée de vie du programme.

Typiquement, ce seront soit les variables déclarées `static` en C/C++ (et quelques autres langages), soit les tableaux et chaînes de caractères littérales.

```
static int TAILLE = 100; // TAILLE est une variable statique

int main(int argc, char* argv[]) {
  printf("Entrez votre nom");
  char nom[TAILLE];
  fgets(nom, TAILLE, stdin);
  printf("Hello, %s\n", nom);
}
```

Dans le programme ci-dessus, sont statiques :

- la variable `TAILLE`
- la chaîne littérale `"Entrez votre nom"`
- la chaîne littérale `"Hello, %s\n"`

2.2. La pile (stack)

Sur la pile, on met les **données dont la taille est connue à la compilation.**

```
int main(int argc, char* argv[]) {
  if (argc == 1) {
    printf("pas d'argument passé\n");
  }
  int x = argc * 2;
  int y = argc * x + 2;
  f(); // Une autre fonction qui prend du temps à s'exécuter
  return 0;
}
```

Ici, quand `main` est appelée (au début du programme), l'OS réserver de la place pour 3 entiers (`argc`, `x` et `y`) et un pointeur (`argv`). Cet espace mémoire sera utilisé jusqu'à la fin du programme (enfin, jusqu'à la fin de `main`).

Quand une fonction est appelée, on lui alloue la place nécessaire sur la pile. L'espace est ensuite libéré automatiquement, aussitôt que la fonction se termine, et sera réutilisé lorsqu'une autre fonction sera appelée.

```
int main(int argc, char* argv[]) {
  // On utilise de l'espace pour argc, argv et x, défini plus bas.
  // Cet espace sera utilisé jusqu'à la fin du programme.
  f1(); // f1 va utiliser de l'espace sur la pile. Puis, une fois la fonction
terminée, cet espace sera libéré.
  int x = f2(); // f2 va réutiliser l'espace qui a été libéré par f1.
}
```

Évidemment, une fonction peut elle-même appeler une autre fonction (ou elle-même, récursivement), dans ce cas l'espace utilisé sur la pile augmente encore.

```
void f1() {
  int x = f2();
```

```
  f3(x);
}

int f2() {
  return fib(3);
}

int fib(int n) {
  if (n < 2) {
    return 1;
  } else {
    return fib(n-1) + fib(n-2);
  }
}

void f3(int x) {
  printf("%d\n", x);
}
```

Attention aux appels récursifs trop nombreux : si on appelle 1000 fois la même fonction récursivement, alors on allouera sur la pile 1000 fois l'espace nécessaire. Au bout d'un moment, un *stack overflow* (débordement de pile) se produit et le programme s'arrête aussitôt.

Quand on appelle une fonction, ses paramètres sont copiés sur la pile. Pareil pour le `return`: sa valeur est copiée sur la pile, on peut soit l'ignorer soit l'utiliser.

Et si on veut *modifier* une variable ? Alors il va falloir passer l'adresse de la variable en question.

```
int main(int argc, char* argv[]) {
  int a = 42;
  int b = 0;
  swap(&a, &b);
  printf("a=%d, b=%d\n", a, b);
}

void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Lorsqu'une fonction est appelée, on doit copier ses paramètres sur la pile lors de l'appel, pour que la fonction puisse savoir ce qu'elle doit lire. Pour les types simples (entier, booléen, etc.) c'est facile : on peut copier les données. Mais pour les types plus volumineux, ou dont la taille peut varier (les tableaux, typiquement), c'est différent : on ne vaut pas copier intégralement les données. On va plutôt donner l'adresse du début des données sur la pile, et le nombre d'élements.

```
const int TAILLE = 10;

int main(int argc, char* argv[]) {
  int tab[TAILLE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // La taille est connue à la
compilation
  opposer(tab, TAILLE);
}

void opposer(int *tab, int n) {
  for (int i = 0; i < n ; i++) {
```

```
    tab[i] = -tab[i];
  }
}
```

Cette fonction fonctionne pour toutes les tailles de tableau, et ne nécessite pas de copier toutes ses valeurs lors de l'appel.

2.2.1. Références invalides et durée de vie

Quand la fonction se termine, ses variables ne sont plus accessibles : attention aux références vers quelque chose qui est dans la pile !

Imaginons : plutôt que de modifier les données d'origine, je veux avoir une copie. Imaginons (pour simplifier) que je sais que les données sont de taille N.

```
int* oppose(int* tab) {
  int result[TAILLE];
  for (int i = 0; i < TAILLE; i++) {
    result[i] = -tab[i];
  }
  return result; // Oh non !
}
```

On a un problème de *durée de vie* : la référence vit plus longtemps que la variable référencée elle-même : la variable "meurt" à la sortie de la fonction, tandis que la référence lui survit. Ce n'est pas possible, seul l'inverse est possible :

La référence doit vivre aussi longtemps, ou moins longtemps que ce qui est référencé.

2.2.2. Avantages de la pile

Facile de s'y retrouver, pas de mémoire à allouer / désallouer à la main.

2.2.3. Inconvénients de la pile

On doit connaître à l'avance la taille des objets que l'on gère : quid des listes de taille dynamique ? Comment créer des objets à la volée, au moment où on en a besoin et pas avant ?

2.3. Le tas (heap)

On y met les données dont la taille n'est pas connue à la compilation. Par exemple, les listes dynamiques. Si j'ai une liste qui peut contenir 3 valeurs, ou 10, ou 1000, avec la pile, je suis embêté : je dois déterminer quelle est la taille maximale, et réserver tout l'espace nécessaire. Si je dis que la taille maximale est de 1000, alors je ne pourrai pas stocker plus de 1000 éléments dans ma liste, et si en réalité je n'en utilise que 3 ou 10, alors l'espace supplémentaire est perdu.

Avec le tas, je peux dire "j'ai besoin de x éléments", et alors seul l'espace nécessaire (ou un peu plus, pour des raisons pratiques) sera réservé.

Mais sa manipulation est plus compliquée : on doit allouer de la mémoire explicitement, le moment venu, en indiquant la quantité nécessaire. On récupère une référence vers la zone allouée. Une fois que la zone ne sera plus utilisée, il faudra trouver un moyen de la désallouer.

```
int *f() {
  int *valeurs = (int*) malloc(10*sizeof(int));
  ...
  return valeurs;
}

...
```

```c
int main(int argc, char* argv[]) {
  int *vals = f();
  ...
  free(vals);
}
```

Il faut penser à libérer la mémoire à un moment, sinon fuite mémoire.

```c
int f2() {
  int *l = f();
  ajouter_valeur(&l, 1);
  ajouter_valeur(&l, 2);
  ajouter_valeur(&l, 3);
  return moyenne(&l);
}
// Oops, je suis sorti de la fonction.
// l n'existe plus, mais je n'ai pas désalloué la mémoire.
// Elle est à tout jamais perdue !
// Chaque fois que j'appellerai f2(), je consommerai de la mémoire en plus.
// Si le programme tourne longtemps, il finira par planter, faute de mémoire
disponible.
```

Autre problème (plus grave encore) : il ne faut pas continuer à utiliser la mémoire qui a été libérée.

```c
int f3() {
  int *l = f();
  int *l2 = l; // Oh, qu'est-ce que je fais ?
  free(l);
  ajouter_valeur(&l2, 1); // oh non !
}
```

Ici, ce qui n'est pas clair, c'est : à qui appartenaient les données ? À `l` ? Ou à `l2` ? Sur des programmes complexes, c'est parfois très difficile à savoir.

Quand plusieurs références pointent vers la même ressource (*aliasing*), des tas de problèmes se posent :

- qui est responsable de la libération de cette zone ?
- qui a le droit d'écrire ou de lire les données ?
- comment s'assurer que deux threads n'écrivent pas en même temps ?

2.4. Le garbage collector

La gestion du tas est complexe et source de nombreux bugs et vulnérabilités graves. Solution : le GC (garbage collector).

Le GC s'occupe de surveiller quels sont les objets du tas qui ne sont plus référencés par personne, et de les désallouer automatiquement. En d'autres termes, on fait juste le `malloc`, il s'occupe lui-même du `free` !

2.4.1. Le compteur de références

Très simple dans le principe :
- quand on alloue, on indique aussi qu'il y a une référence qui pointe vers l'objet,
- quand une nouvelle référence souhaite pointer vers l'objet, on incrémente le compteur,
- quand une référence cesse de pointer vers l'objet, on décrémente le compteur,
- quand le compteur arrive à 0, on désalloue l'espace mémoire.

Avantages:

- déterministe (on sait à quel moment il est appelé : quand on référence ou déréférence une zone mémoire),
- efficace (c'est juste un compteur à incrémenter ou décrémenter),
- facile à implémenter.

Inconvénient principal : ne détecte pas les cycles.

Les versions de python jusqu'à récemment utilisaient un compteur de références.

```
l = [1, 2, 3] # une référence pointe sur cette liste qui vient d'être créée
l2 = l        # Deux références
l3 = l2       # Trois références
l = None      # Deux références. Le créateur ne la référence plus mais on s'en
fiche !
l3 = None     # Une référence
l2 = None     # Zéro référence : la liste [1, 2, 3] est désallouée ici
```

Ce genre de comportement serait difficile à écrire en C correct.

C'est très élégant pour désallouer une liste chaînée ou un arbre, ou même un DAG.

MAIS…

Quid d'une liste doublement chaînée ? D'un tourniquet ? D'un graphe ?

Malgré leur défaut, les compteurs de référence sont très utiles : simples, efficaces et déterministes.

En C++ : le `shared_pointer` utilise un compteur de références. En rust, on les utilisera aussi parfois, via le type `Rc<T>`.

2.4.2. Les GC plus complexes

Les langages de haut niveau (ceux qui gèrent automatiquement la mémoire, comme python, java, javascript, etc.) utilisent un GC plus évolué (récent pour python).

Ces GC sont très complexes (c'est un domaine de recherche en soi), ils gèrent parfaitement les cycles. Ils offrent beaucoup de garanties et résolvent les problèmes évoqués plus haut.

2.4.3. Inconvénients des GC hors compteurs de références

Ils sont **non-déterministes** : on ne sait pas quand ils s'exécuteront, on sait que la mémoire sera libérée à un moment, mais quand ? Pas forcément avant la fin de l'exécution du programme, en tout cas. Dans certains cas, s'ils s'exécutent trop rarement, cela veut dire que l'on peut se retrouver sans mémoire disponible alors que beaucoup de zones auraient pu être libérées. Mais s'ils s'exécutent trop souvent, alors ils consomment trop de temps CPU, et le programme passe plus de temps à gérer sa mémoire qu'à faire son travail.

"Stop the world" : de temps en temps, le programme principal et tous les threads sont mis en pause pour laisser le GC s'exécuter sans interférence. Dans les bons GC, ces pauses sont limitées dans le temps (par exemple : on garantit qu'elle ne durera pas plus de 100 ms d'affilée). Mais dans certains cas, c'est inacceptable : systèmes embarqués temps réel par exemple, où l'on a besoin de s'assurer que telle ou telle fonction ne prendra pas plus de x ms à s'exécuter. Ou bien, dans un jeu, la pause sera perceptible et nuira à l'expérience.

Temps CPU : le GC consomme des cycles CPU pour effectuer son travail. C'est une surcharge pour le programme, qui n'est pas toujours nécessaire et peut nuire aux performances.

Espace mémoire : le GC consomme de l'espace mémoire pour assurer son propre fonctionnement. Dans l'embarqué, où chaque kilooctet est compté, on ne peut pas toujours se le permettre.

C’est presque un programme à l’intérieur du programme, en fait.

2.5. Le beurre et l’argent du beurre

Et si on pouvait avoir les garanties offertes par les GC (gestion sans faille de la mémoire) sans en payer le coût ? On pourrait avoir des programmes efficaces, sans surcharge, et qui serait invulnérables à la plupart des failles coûteuses des programmes C/C++.

C’est une des promesses de rust.

Mais le langage est très complexe.

3. Mutabilité et concurrence

3.1. Alias

Une variable est (en général) une zone mémoire dont la valeur peut *varier* (sic) au cours de l'exécution du programme.

Il est possible de référencer une autre variable (ou une zone mémoire du tas) via une *référence* (sic) ou *pointeur*.

Plusieurs références peuvent donc référencer la même zone de mémoire. On parle d'aliasing : une zone mémoire a plusieurs noms (ou alias).

Problème : qui possède quoi ? Si b référence a, et que a est modifié, alors b est aussi modifié. Est-ce toujours acceptable ?

Si `l1` et `l2` sont des listes, et que j'ajoute un élément à `l1`, alors j'ai ajouté un élément à `l2`. Est-ce forcément ce que l'on voulait faire ?

Exemple :

J'ai la liste des notes que j'ai obtenues ce semestre. On part du principe que la liste n'est pas vide.

J'ai deux fonctions, `calculer_médiane` qui me donne la médiane d'une liste de notes, et `calculer_pente` qui me permet de savoir si les notes que j'obtiens sont en progrès ou non.

```
def calculer_médiane(notes: 'list[int]') -> int:
  notes.sort()
  return notes[int(len(notes)/2)]

# pente = Σ((xi - x̄)(yi - ȳ)) / Σ((xi - x̄)²)
def calculer_pente(notes: 'list[int]') -> float:
  n = len(notes)
  y = notes
  moyenne_x = (n-1) / 2
  moyenne_y = sum(y) / n
  numerateur = sum((x[i] - moyenne_x) * (y[i] - moyenne_y) for i in range(n))
  denominateur = sum((x[i] - moyenne_x) ** 2 for i in range(n))
  return numerateur / denominateur

notes = [17, 12, 16, 10, 8, 9, 5, 7, 3]
print(calculer_pente(notes))
print(calculer_médiane(notes))
print(calculer_pentes(notes)) # QUOI ?
```

Ici, `calculer_médiane` modifie son paramètre. Solutions ?

3.1.1. Cloner

À chaque fois, tout cloner (mais peut être coûteux) :

```
notes = [17, 12, 16, 10, 8, 9, 5, 7, 3]
print(calculer_pente(notes[:]))
print(calculer_médiane(notes[:]))
print(calculer_pentes(notes[:])) # Tellement de copies 😭
```

On fait beaucoup de copies, par sécurité. Certaines sont inutiles : `calculer_pente` ne modifie pas son paramètre. Mais comment le savoir à l'avance ?

À l'inverse, on pourrait dire que c'est la responsabilité de `calculer_médiane` de faire la copie. Mais le risque, c'est de faire la copie deux fois (une avant l'appel, une dans la fonction), par sécurité.

3.1.2. Structures non-mutables

Le problème vient de la mutabilité : si on choisit des structures non-mutables, le problème est résolu.

```
def calculer_médiane(notes: 'tuple[int]') -> int:
  notes = sorted(notes) # On ne peut pas utiliser sort() avec des tuples
  return notes[int(len(notes)/2)]

# pente = Σ((xi - x̄)(yi - ȳ)) / Σ((xi - x̄)²)
def calculer_pente(notes: 'tuple[int]') -> float:
  n = len(notes)
  x = tuple(range(n))
  y = notes
  moyenne_x = sum(x) / n
  moyenne_y = sum(y) / n
  numerateur = sum((x[i] - moyenne_x) * (y[i] - moyenne_y) for i in range(n))
  denominateur = sum((x[i] - moyenne_x) ** 2 for i in range(n))
  return numerateur / denominateur

notes = [7, 12, 10, 8, 15, 17, 13]
print(calculer_pente(notes))
print(calculer_médiane(notes))
print(calculer_pentes(notes))
```

Ici, on a la garantie que `calculer_médiane` ne modifiera *pas* son paramètre, étant donné qu'un tuple n'est pas mutable.

3.1.3. La solution ultime : la programmation fonctionnelle

En Haskell et dans les autres langages fonctionnels purs, on a la garantie que *toutes* les valeurs (listes, tuples, dictionnaires et autre structures de données) sont non-mutables. L'aliasing n'est donc jamais un problème.

Mais les langages fonctionnels purs sont très difficiles à manipuler pour certaines tâches habituellement triviales.

En outre, les langages fonctionnels poussent à faire énormément de copies de données et à créer énormément d'objets sur le tas. Ils nécessitent l'utilisation d'un GC.

Donc ce n'est pas si ultime que ça. En fait, on voudrait un moyen de dire "toi tu peux modifier tel paramètre, toi tu ne peux pas".

3.2. Parallélisme

Quand on a plusieurs threads qui s'exécutent en parallèle et manipulent en même temps les même données, les problèmes sont démultipliés. En outre, les comportements deviennent non-déterministes et les bugs d'autant plus difficiles à résoudre.

3.3. Le beurre, l'argent du beurre et le sourire de la crémière

Rust offre aussi des garanties dans ce domaine : le compilateur s'assure toujours qu'un même objet ne peut pas être accessible en écriture plusieurs fois en même temps, ni qu'une référence en lecture ne soit utilisée en même temps qu'une référence en écriture.

Sans être un langage fonctionnel, rust part du principe que, par défaut, les variables et références sont non-mutables : la mutabilité doit être l'exception, pas la norme.

4. Rust

Un langage qui permet d'assurer à la fois la fiabilité (et même plus) des langages de haut niveau, tout en étant aussi efficace et déterministe que les langages de bas niveau.

Le langage est complexe, mais il émerge peu à peu et commence à conquérir des domaines que l'on pensait chasse gardée de C : noyau linux, systèmes embarqués, navigateurs web et d'autres.

4.1. Hello, world!

```
fn main() {
  println!("Hello, world!"); // println! se termine par un point d'exclamation car
c'est une macro
}
```

```
fn main() {
  let name = "toto";
  println!("Hello, {}", name); // les {} représentent les paramètres. Le compilateur
s'assure que l'on en a le bon nombre.
  println!("Hello, {name}");   // on peut aussi faire comme avec les f-strings en
python, mais pour les variables uniquement, pas les expressions.
}
```

4.2. Variables

Définies à l'aide du mot-clé `let`. On peut les typer, mais en général le compilateur détermine automatiquement leur type (inférence de type). Elles sont non-mutables par défaut.

```
fn main() {
  let a = 3;
  let b = 2;
  let c = a + b;
  println!("{a} + {b} = {c}");
  c += 1; // Erreur de compilation
}
```

```
error[E0384]: cannot assign twice to immutable variable `c`
 --> src/main.rs:6:5
  |
4 |     let c = a + b;
  |         - first assignment to `c`
5 |     println!("{a} + {b} = {c}");
6 |     c += 1;
  |     ^^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
4 |     let mut c = a + b;
  |         +++
```

Notez que le compilateur nous aide beaucoup. Pour qu'une variable soit mutable, on doit la définir à l'aide de `let mut`.

4.3. Types de données

4.3.1. Les entiers

Comme on se destine à la programmation de bas niveau, il n'y a pas de type entier par défaut, mais uniquement des types définis en fonction de leur taille, signés ou non : `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, `u128`, `i128`. Le type inféré par défaut est `i32`. On rencontre aussi souvent `usize` qui est le type

pour représenter les tailles, notamment la taille d'un pointeur (64 bits sur ma machine). `isize` existe aussi, on le rencontre moins.

4.3.2. Les flottants

On utilise les types `f32` et `f64`. Le premier est plus compact, le second plus précis. En général on utilise `f64`, c'est celui inféré par défaut. Mais `f32` peut être pratique quand on manipule beaucoup de flottants (matrices par exemple, pour le machine learning).

4.3.3. Les booléens

Type `bool`, valeurs `true` et `false`. On ne peut pas interpréter un booléen comme un entier et *vice versa* (contrairement à python ou C).

4.3.4. Les caractères

On dispose du type `char`. Il est suffisamment grand pour contenir n'importe quel caractère encodé en UTF-8 (4 octets). On met les caractères entre quotes simples `'`.

```
let premier: char = 'a';
let dernier = 'z';
```

4.3.5. Aparté concernant les caractères et UTF-8

Autrefois, le monde informatique était dominé par les anglophones.

Il n'y a pas d'accent sur les lettres, un octet c'est largement suffisant pour représenter tous les caractères (ASCII). Une chaîne = un tableau d'octets, une chaîne de 23 octets = une chaîne de 23 caractères.

Avec Internet et le développement mondial de l'informatique, il a fallu intégrer les accents et tous les caractères non-latins, et pourquoi pas des symboles rigolos (emojis) puisqu'on a la place : standard Unicode.

Problème : il y a des centaines de milliers de caractères différents en fin de compte, il faut donc 4 octets pour les représenter tous 😱.

Deux solutions :

4.3.5.1. UTF-32

Tous les caractères font 4 octets.

Avantage : indexer une chaîne reste simple, le ième caractère de la chaîne est à la position `4*i`.

Inconvénient : les chaînes deviennent très volumineuses, et le standard n'est pas compatible avec ASCII, il faut donc passer par des phases de conversion inutiles.

4.3.5.2. UTF-8

On veut garder la compatibilité avec ASCII (toute chaîne ASCII est aussi une chaîne UTF-8) et optimiser l'utilisation de l'espace. On est prêt à perdre les facilités d'indexation : on stocke / affiche des chaînes bien plus souvent qu'on ne les indexe.

Solution : la taille des caractères est variable, les caractères qui sont dans ASCII continuent d'être représentés sur un octet, d'autres caractères nécessitent deux octets, ou bien trois, voire quatre.

Une chaîne en UTF-8 est en général bien plus compacte qu'une chaîne en UTF-32, par contre on perd la possibilité d'indexer une chaîne : où est le 12ème caractère d'une chaîne quand tous les caractères ne font pas la même taille ? Il faut parcourir la chaîne du début pour le trouver.

Conclusions :

– en UTF-8, une chaîne de caractère n'est **pas** un tableau de caractères,
– en UTF-8, tous les caractères ne font pas la même taille, et ils peuvent faire jusqu'à 4 octets.

En rust, les caractères individuels sont donc stockés sur 4 caractères, et un tableau de caractères sera plus volumineux que la chaîne équivalente.

Les chaînes de caractères sont un type étonnamment complexe, on les verra un peu plus tard.

4.3.6. Les tuples

Comme en haskell et, d'une certaine manière, en python, on dispose d'un type tuple, qui permet d'associer des valeurs de types différents.

```
fn main() {
  let t: (char, i32, f64) = ('💀', 17, 3.14);
  println!("{:?}", t); // On ne peut pas afficher un tuple facilement : il faut
utiliser le mode Debug
}
```

On accède aux éléments d'un tuple en utilisant la notation pointée et en indiquant l'indice désiré.

```
fn main() {
  let t = ('💀', 17, 3.14); // Inférence de type
  let pi = t.2;             // Inférence ici aussi
  println!("{}", pi);       // On *peut* afficher un flottant
}
```

4.3.7. Les tableaux

Ils sont déclarés sur la pilet et sont donc de taille fixe, et contiennent un certain nombre d'éléments d'un type donné.

```
let tab: [i32; 5] = [1, 2, 3, 4, 5]; // Type : tableau de cinq i32
let tab2 = ['a', 'b', 'c'];          // Type inféré : [char; 3]
let tab3: [i32; 10] = [0; 10];       // 10 fois la valeur 0.
```

Les tableaux sont faciles à manipuler (ils sont sur la pile) mais rigides (taille fixe et connue à la compilation : ils sont sur la pile). On verra plus tard comment manipuler des tableaux dynamiques, alloués sur le tas (vecteurs).

4.4. Les fonctions

On utilise le mot-clé `fn`. Les paramètres doivent être typés (pas d'inférence de type).

```
fn add_two (x: i32) -> i32 {
  return x + 2;
}

fn main() {
  let a = 5;
  let b = add_two(a);
  println!("{a} + 2 = {b}");
}
```

Astuce rust que l'on verra souvent : quand la dernière expression d'une fonction n'a pas de point-virgule, on considère qu'il y a un `return` implicite. Cette syntaxe vient des langages fonctionnels.

```
fn add_two(x: i32) -> i32 {
  x + 2
}
```

Si je mets le point-virgule sans le return ça ne marche pas :

```
fn add_two(x: i32) -> i32 {
  x + 2;
}

error[E0308]: mismatched types
 --> src/main.rs:1:23
  |
1 | fn add_two(x: i32) -> i32 {
  |    -------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
2 |     x + 2;
  |          - help: remove this semicolon to return this value
```

Notez que, une fois encore, le compilateur nous donne la solution.

Notez aussi au passage qu'on considère qu'une fonction qui ne renvoie pas de résultat renvoie le type unit `()`.

4.5. Les conditionnelles et les boucles

On utilise `if` un peu comme en C, sauf que la condition n'a pas besoin d'être entourée de parenthèses et les accolades sont obligatoires.

```
fn affiche_mention(note: i32) {
  if note < 10 {
    println!("échec");
  } else if note < 12 {
    println!("passable");
  } else if note < 14 {
    println!("assez bien");
  } else if note < 16 {
    println!("bien");
  } else {
    println!("très bien");
  }
}
```

On peut utiliser le `if` dans une expression, comme en programmation fonctionnelle à condition de ne pas mettre de `;` dedans.

J'ai une valeur entière, je veux la mettre à 0 si elle est négative.

```
let val2 = if val < 0 {
  0
} else {
  val
};

// ou

let val2 = if val < 0 { 0 } else { val };
```

C'est l'équivalent de l'opérateur `?:` en C :

```
int val2 = val < 0 ? 0 : val; // Syntaxe équivalente en C, ceci n'est pas du rust
```

Le `while` fonctionne comme en C, sans les parenthèses mais accolades obligatoires.

Le `for` itère sur un itérateur (sic) : `for elt in iter`.

Comme en python, donc.

Un itérateur est quelque chose qui produit une nouvelle valeur chaque fois qu'on lui demande, jusqu'à ce qu'il soit vide (éventuellement).

Afficher les nombres de 0 à 10:

```
for i in 0..10 { // 0..10 est un itérateur, c'est l'équivalent python de range(10)
  println!("{i}");
}

// Oh non ! J'ai oublié le 10

for i in 0..11 {
  println!("{i}");
}

for i in 0..=10 {
  println!("{i}");
}

// Maintenant, je veux faire la même chose mais dans l'autre sens

for i in (0..=10).rev() {
  println!("{i}");
}
```

5. Ownership (propriété)

Concept **fondamental** et innovant en rust.

On sait en permanence qui est propriétaire d'une ressource, et donc (notamment) qui est responsable de sa destruction le cas échéant.

C'est un concept unique à rust (en tout cas ils ont été les premiers à le développer dans un langage grand public) et assez difficile à comprendre totalement. On se bat assez souvent avec le *borrow checker*, outil qui, à la compilation, vérifie les règles draconiennes de rust. Il peut parfois refuser des programmes corrects mais dangereux.

Le *borrow checker* donne des garanties de sûreté mais permet aussi de faire certaines optimisations à la compilation, étant donné les garanties qu'il offre. Optimisations qui ne seraient pas possibles en C.

On emprunte (*borrow*) une variable en y faisant référence.

5.1. Les références

Quand on passe un paramètre entier, ou flottant, ou booléen, ou un caractère à une fonction, on peut copier directement sa valeur sur la pile : ces types ne prennent pas beaucoup de place en mémoire.

Mais pour des types plus gros (chaînes de caractères, tableaux, etc.) ce n'est pas vrai. Copier une chaîne de 10000 caractères prendrait du temps et occuperait inutilement de la place en mémoire. Pour ces types, on va donc plutôt utiliser une référence (passage par référence) / un pointeur.

On utilise le caractère & pour référencer une variable ainsi que pour définir les types référence. Quand on référence quelque chose, on l'emprunte (*borrow*).

Exemple : une fonction qui renvoie la sommee d'un tableau d'entiers.

```
// t est une référence vers un tableau de i32.
// NB : on ne connaît pas la taille de ce tableau.
fn sum_of(t: &[i32]) -> i32 {
  let mut sum = 0;
  for i in 0..t.len() {
    sum += t[i];
  };
  sum
}

fn main() {
  let t: [i32; 5] = [5, 1, 3, -2, 0];
  let s = sum_of(&t);
  println!("sum_of({:?}) = {s}", t);
}
```

`t` est un tableau litéral, il existera pendant toute la durée de vie de la fonction `main`. `t` *possède* les données.

`&t` est une référence vers `t`. On *emprunte* temporairement l'adresse de `t`. Ce n'est pas un problème, puisque la fonction se termine avant que `t` ne soit détruit.

En outre, ni `t` ni `&t` ne sont mutables, on n'aura donc pas de mauvaise surprise.

Revenons à notre calcul sur les notes.

```
fn calculer_médiane(notes: &[i32]) -> i32 {
  let mut notes2 = notes.to_vec(); // On crée un clone
  notes2.sort();
  notes2[notes2.len()/2]
```

```
}

fn calculer_pente(notes: &[i32]) -> f64 {
    let n = notes.len();
    let moyenne_x = (n - 1) as f64 / 2.0;
    let somme_y: i32 = sum_of(notes);
    let moyenne_y = somme_y as f64 / n as f64;
    let mut numerateur = 0.0;
    let mut denominateur = 0.0;
    for i in 0..n {
        let xi = i as f64;
        let yi = notes[i] as f64;
        let diff_x = xi - moyenne_x;
        let diff_y = yi - moyenne_y;
        numerateur += diff_x * diff_y;
        denominateur += diff_x * diff_x;
    }
    numerateur / denominateur
}

fn main() {
  let notes = [17, 12, 16, 10, 8, 9, 5, 7, 3];
  println!("{}", calculer_pente(&notes));
  println!("{}", calculer_médiane(&notes));
  println!("{}", calculer_pentes(&notes));
}
```

Ici, on sait que ni `calculer_pente` ni `calculer_médiane` ne peuvent modifier les données qui leur sont prêtées. D'ailleurs, comment le pourraient-ils: `notes` n'est pas mutable non plus.

On évite les copies inutiles, et on a les avantages de la programmation fonctionnelle seulement là où c'est pertinent.

Et si on voulait pouvoir modifier la liste ? Par exemple, on veut harmoniser les notes, on veut qu'elles soient toutes comprises entre 0 et 20.

```
fn harmo(note: i32) -> i32 {
  if note < 0 {
    0
  } else if note > 20 {
    20
  } else {
    note
  }
}

fn harmoniser(notes: &mut [i32]) {
  for i in 0..notes.len() {
    notes[i] = harmo(notes[i]);
  }
}

fn main() {
  let mut notes = [-1, 20, 25, 0]; // mut : on doit pouvoir modifier !
  harmoniser(&mut notes);          // On voit que notes peut être modifié
  println!("{}", calculer_pente(&notes));   // Pas de modif ici
```

```
  println!("{}", calculer_médiane(&notes)); // Ici non plus
}
```

Dans tous les cas ci-dessus, la propriété des données appartient à `notes`. Les fonctions ne font qu'emprunter ces données (*borrow*) via des références (mutables ou non). Ces références ne vivent pas plus longtemps (*lifetime*) que les données, donc le compilateur est satisfait.

Ici aussi c'est bon :

```
fn main() {
  let mut fibs: [i32; 20] = [1; 20]; // 20 fois la valeur 1
  for i in 2..fibs.len() {
    fibs[i] = fibs[i-1] + fibs[i-2];
  }
  for i in 0..fibs.len() {
    // let x = fibs[i]; On ferait ça en principe
    let r = &fibs[i]; // On prend une référence juste pour l'exemple
    println!("fib({i}) = {}", *r);
    println!("fib({i}) = {}", r); // C'est pareil : la déréférence est implicite
  }
}
```

La référence `r` ne vit que le temps de la boucle. Elle est libérée après. Sa durée de vie est donc plus courte que celle de `fibs` (qui vit toute la fonction). Donc c'est bon.

MAIS…

```
fn main() {
  let mut r: &i32;
  for i in 0..10 {
    r = &i;
  }
  println!("{}", *r);
}
```

```
error[E0597]: `i` does not live long enough
 --> src/main.rs:5:13
  |
4 |     for i in 1..10 {
  |         - binding `i` declared here
5 |         r = &i;
  |             ^^ borrowed value does not live long enough
6 |     }
  |     - `i` dropped here while still borrowed
7 |
8 |     println!("{}", *r);
  |                     - borrow later used here
```

Ça ne marchera pas : `i` vit moins longtemps que la référence `r`. Quelle pourrait être la valeur de `*r` à la sortie de la boucle ? 9 ? Pitié, on n'est pas chez python ici.

5.2. Les règles d'emprunt

On peut emprunter autant de fois que l'on veut de manière non-mutable une variable non-mutable :

```
let a = 42;
let r1 = &a;
let r2 = &a;
println!("{}", a + r1 + r2);
println!("{}", a + *r1 + *r2); // C'est pareil
```

Ça ne pose pas de problème, la variable est non-mutable, les références aussi, la vie est belle.

MAIS…

Quand une variable est empruntée, on ne peut plus modifier sa valeur (sinon, cela reviendrait à dire qu'une variable non-mutable voit son contenu muter 😱

```
let mut a = 42;
let r1 = &a;
println!("Jusqu'ici tout va bien : {r1}");
```

Mais

```
let mut a = 42;
let r1 = &a;
a += 1;
println!("Quelle est la valeur de r1 ? {r1}");

error[E0506]: cannot assign to `a` because it is borrowed
 --> src/main.rs:4:5
  |
3 |     let r1 = &a;
  |              -- `a` is borrowed here
4 |     a += 1;
  |     ^^^^^^ `a` is assigned to here but it was already borrowed
5 |     println!("Quelle est la valeur de r1 ? {r1}")
  |                                            -- borrow later used here
```

Ou encore :

```
fn main() {
    let mut a = 42;
    let r1 = &mut a;
    let r2 = &a;
    *r1 += 1;
    println!("{r2}")
}

error[E0502]: cannot borrow `a` as immutable because it is also borrowed as mutable
 --> src/main.rs:4:14
  |
3 |     let r1 = &mut a;
  |              ------ mutable borrow occurs here
4 |     let r2 = &a;
  |              ^^ immutable borrow occurs here
5 |     *r1 = 3;
  |     ------- mutable borrow later used here
```

En fait, la règle est la suivante : **quand on emprunte de manière mutable, personne d'autre ne peut emprunter en même temps**.

Cela permet d'éviter les *data races* et autres comportements imprévisibles (comme une référence non-mutable qui est soudainement mutée).

Ce sont des règles assez draconiennes, le compilateur nous embêtera souvent avec ça. Mais c'est pour la bonne cause !

6. Le tas et le type `Vec<T>`

Pour l'instant on sait créer des tableaux sur la pile, mais ils sont contraignants : on doit connaître leur taille à la compilation, et on ne peut pas ajouter d'éléments pour agrandir le tableau une fois qu'il est plein.

Heureusement, il est possible de créer des tableaux dynamiques (des *vecteurs*) via le type **`Vec<T>`**.

C'est un type générique : on peut l'instancier pour n'importe quel type, on peut par exemple créer un `Vec<i32>`.

```
let mut v = Vec::new();
v.push(42);
println!("{}", v.len()); // 1
println!("{:?}", v); // [42]
```

Un `Vec`, c'est trois valeurs : un pointeur vers le début des données, la taille actuelle, et la capacité. La capacité, c'est l'espace total qui a été réservé. Tant que la capacité n'est pas atteinte, c'est facile d'ajouter de nouvelles données. Une fois qu'elle est atteinte, il faut allouer une nouvelle zone mémoire. C'est simple pour l'utilisateur : tout est transparent.

NB : pour créer un `Vec`, il faut allouer de la mémoire sur le tas. Il faut donc aussi la désallouer à la fin ! Or, il n'y a pas de GC en rust. Mais ici c'est transparent : on part du principe que les données sont possédées par `v` (ownership), on sait donc que c'est `v` qui devra la désallouer au moment où il "mourra" (quand il sortira du scope). Le compilateur ajoute automatiquement pour nous l'appel à `free()` (en fait, en rust, l'appel à `drop()`) quand la variable qui possède les données sort du scope. C'est pour ça qu'il est primordial de savoir *qui* possède telle ou telle variable !

```
fn main() {
  let mut v1 = Vec::new();
  for i in 0..10 {
    let mut v2 = Vec::new();
    v1.push(i);
    v2.push(i);
    println!("{:?}", v1);
    println!("{:?}", v2);
    // Ici, à chaque itération, v2 meurt.
    // Appel implicite à la fin de l'itération:
    // free(v2);
    // En rust, free s'appelle drop, donc en fait on fait :
    // drop(v2);
  }
  // Ici, v1 meurt. Appel implicite :
  // drop(v1);
}
```

Ici, `v1` est créé au début de la fonction, il n'est pas retourné en fin de fonction, il sera donc désalloué en sortie. Quant à `v2`, il est créé au début de chaque itération de la boucle, il est détruit à la fin de l'itération, il sera donc désalloué à chaque fois à ce moment-là.

C'est très rigoureux et transparent pour l'utilisateur qui n'a pas à s'en occuper mais sait que les données seront *forcément* désallouées à ce moment précis. Ni avant, ni après.

Et si on veut désallouer plus tôt ? On peut, en appelant explicitement `drop`. Évidemment, une fois qu'on a dropped une variable, on n'y a plus accès.

```
fn main() {
  let mut v: Vec<i32> = Vec::new();
```

```
  drop(v);
  println!("{:?}", v);
}

error[E0382]: borrow of moved value: `v`
 --> src/main.rs:4:20
  |
2 |   let mut v: Vec<i32> = Vec::new();
  |       ----- move occurs because `v` has type `Vec<i32>`, which does not implement
the `Copy` trait
3 |   drop(v);
  |        - value moved here
4 |   println!("{:?}", v);
  |                    ^ value borrowed here after move
```

Je dis “évidemment”, mais ce n’est pas évident pour quelqu’un qui a l’habitude de C et qui ne s’étonnera pas que le programme suivant compile :

```
int *c = (int*) malloc(sizeof(int));
free(c);
printf("c est supprimé mais yolo: %d\n", *c);
```

Si on veut créer un `Vec` prérempli avec des valeurs prédéfinies, on peut utiliser la macro `vec!`.

```
let v = vec![1, 2, 3]; // Sur le tas
println!("{}", v.len()); // 3
println!("{:?}", v); // [1, 2, 3]
```

C’est un peu plus lourd que :

```
let tab = [1, 2, 3]; // Sur la pile
println!("{}", v.len());
println!("{:?}", v);
```

C’est plus lourd parce qu’à l’exécution, il faut allouer des données pour `v` sur le tas, et qu’elles seront désallouées au moment du `drop()`. En outre, un `Vec<T>` nécessite aussi d’avoir un entier pour la taille et un autre pour la capacité.

Alors quels intérêt ? Si `v` est mutable, alors on pourra lui ajouter des données, en supprimer, etc. Mais si on ne prévoit pas de faire ça, alors c’est inutile. En fait, la bonne pratique est la suivante :

On n’alloue sur le tas *que* si on ne peut pas déclarer sur la pile.

On peut référencer un `Vec` : dans ce cas, la référence sera semblable à une référence vers un tableau. On parle de *slice* (tranche) pour ce type `&[T]`.

```
fn afficher_len(t: &[i32]) {
  println!("La longueur de {:?} est {}", t, t.len());
}

fn main() {
  let v = vec![1, 2, 3];
  let tab = [1, 2, 3];
  afficher_len(&v);
  afficher_len(&tab);
}
```

Ici encore, utiliser un `Vec` ou un `[i32]` ne fait pas de différence. Comme la taille est connue et fixe dans les deux cas, il vaudrait mieux utiliser un tableau : c’est plus efficace (pas d’allocation / désallocation).

On peut les rendre mutables, évidemment :

```
fn opposer(t: &mut [i32]) {
  for i in 0..t.len() {
    t[i] = -t[i];
  }
}

fn main() {
  let mut v = vec![1, 2, 3];
  let mut tab = [4, 5, 6];
  opposer(&mut v);
  opposer(&mut tab);
  println!("{:?} {:?}", v, tab); // [-1, -2, -3] [-4, -5, -6]
}
```

Alors dans quels cas utiliser un `Vec` vu que c'est plus lourd à l'exécution ? Eh bien, déjà, on peut retourner un `Vec` en fin de fonction : on n'a pas besoin de connaître sa taille à l'avance.

```
fn jours_dans_mois(année: i32) -> [i32; 12] {
  if est_bisextile(année) {
    [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  } else {
    [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  }
}

fn jours_dans_mois_vec(année: i32) -> Vec<i32> {
  if est_bisextile(année) {
    vec![31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  } else {
    vec![31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  }
}
```

Ici, les deux fonctionnent, et utiliser un `[i32; 12]` est préférable : c'est plus efficace (même si on doit copier toutes les données chaque fois qu'on l'utilise dans une autre fonction). Bon, en vrai, la différence est négligeable.

Mais il serait impossible d'écrire la fonction suivante avec un tableau, puisque les tableaux ont des tailles fixes :

```
fn les_lundis_du_mois(mois: i32, année: i32) -> Vec<i32> {
  // Renvoie le numéro de tous les lundis d'un mois donnée
  // Il peut y en avoir 4 ou 5
}
```

Impossible d'écrire cette fonction avec un tableau : doit-on renvoyer un `[i32; 4]` ou un `[i32; 5]` ?

Mais est-ce qu'on ne pourrait pas renvoyer un `&[i32]` pour régler ce problème ? Non ! Car alors il y a un problème de durée de vie.

```
fn jours_dans_mois(année: i32) -> &[i32] {
  if est_bisextile(année) {
    &[31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  } else {
    &[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  }
}
```

```
fn jours_dans_mois_vec(année: i32) -> &[i32] {
  if est_bisextile(année) {
    &vec![31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  } else {
    &vec![31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  }
}

error[E0106]: missing lifetime specifier
 --> src/lib.rs:1:35
  |
1 | fn jours_dans_mois(année: i32) -> &[i32] {
  |                                   ^ 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, but this is uncommon unless you're
returning a borrowed value from a `const` or a `static`
  |
1 | fn jours_dans_mois(année: i32) -> &'static [i32] {
  |                                    +++++++
help: instead, you are more likely to want to change the argument to be borrowed...
  |
1 | fn jours_dans_mois(année: &i32) -> &[i32] {
  |                           +
help: ...or alternatively, you might want to return an owned value
  |
1 - fn jours_dans_mois(année: i32) -> &[i32] {
1 + fn jours_dans_mois(année: i32) -> Vec<i32> {
  |

error[E0106]: missing lifetime specifier
 --> src/lib.rs:9:39
  |
9 | fn jours_dans_mois_vec(année: i32) -> &[i32] {
  |                                       ^ 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, but this is uncommon unless you're
returning a borrowed value from a `const` or a `static`
  |
9 | fn jours_dans_mois_vec(année: i32) -> &'static [i32] {
  |                                        +++++++
help: instead, you are more likely to want to change the argument to be borrowed...
  |
9 | fn jours_dans_mois_vec(année: &i32) -> &[i32] {
  |                               +
help: ...or alternatively, you might want to return an owned value
  |
9 - fn jours_dans_mois_vec(année: i32) -> &[i32] {
9 + fn jours_dans_mois_vec(année: i32) -> Vec<i32> {
  |
```

C'est un problème de *lifetime* : on renvoie une référence vers une valeur qui est détruite à la fin de la fonction. N'oubliez pas le `drop()` automatique à la fin.

Mais le compilateur nous aide : quand on veut renvoyer un `&[i32]`, en fait, ce qu'on veut sans doute faire, dans 90% des cas, c'est renvoyer un `Vec<i32>`. Oui, cela coûte une allocation et plus tard une désallocation, mais souvent, on peut se le permettre.

6.1. Sémantique de mouvement

Copier un entier, un booléen, ou même un tableau "brut", on voit comment ça marche : on fait une simple copie bit à bit.

Mais copier une liste, un `Vec`, ça veut dire quoi ? Si je fais ça, qu'est-ce qui est censé se passer ?

```
let mut v1 = vec![1, 2, 3];
let mut v2 = v1;
v1.push(42);
println!("{}", v2.len());
```

Qu'est-ce que j'attends comme résultat ? En python ou en java, `v2` sera modifié par la modification de `v1`. Mais est-ce que c'est vraiment ce que je veux ? Pas toujours. Et surtout, dans ce cas : qui possède les données ? Qui sera responsable de leur destruction ? Ce n'est pas clair. Et comme ce n'est pas clair, ce sera refusé par le compilateur.

```
error[E0382]: borrow of moved value: `v1`
 --> src/main.rs:4:5
  |
2 |     let mut v1 = vec![1, 2, 3];
  |         ------ move occurs because `v1` has type `Vec<i32>`, which does not
implement the `Copy` trait
3 |     let mut v2 = v1;
  |                  -- value moved here
4 |     v1.push(42);
  |     ^^ value borrowed here after move
  |
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let mut v2 = v1.clone();
  |                    ++++++++
```

Intéressant ! Il nous dit que `Vec` n'implémente pas `Copy`, ça veut dire qu'on ne peut pas le copier (bit à bit), contrairement aux types basiques : la sémantique associée à la copie d'un pointeur n'est pas évidente.

Il dit aussi qu'on peut **cloner** le `Vec` si on veut une copie (c'est-à-dire, deux vecteurs identiques mais indépendants l'un de l'autre). Il rappelle que cela a un coût : il faut allouer à nouveau, copier toutes les données, utiliser deux fois plus de mémoire, et désallouer à la fin.

Et surtout, il dit que la valeur a été **déplacée (*move*)**. En fait, on a transféré la responsabilité des données à `v2` : on les lui a en quelque sorte offertes.

C'est `v2` maintenant qui possède les données, qui peut les utiliser comme il le souhaite, et qui les détruira lorsqu'il sortira du scope. Et `v1` maintenant n'est plus valide : on ne peut plus l'utiliser.

Ah oui, mais si je veux vraiment que `v1` et `v2` référencent la même liste sous-jacente, et potentiellement les modifier toutes les deux ? Alors là, c'est plus compliqué, on aura besoin d'utiliser des types plus avancés pour gérer à la fois la propriété partagée et la mutabilité partagée.

6.2. Les chaînes de caractères

Ce sont en quelque sorte des tableaux spécialisés, mais pas tout à fait (à cause d'utf-8). De la même manière qu'il y a deux types de tableaux (par exemple, `&[i32]` et `Vec<i32>`), il y a deux types de

chaînes de caractères : `&str` pour décrire les chaînes statiques ou les fragments de chaînes, et `String` pour les chaînes dynamiques que l'on veut pouvoir modifier.

```
let s: &str = "Hello, world";
let mut s2: String = s.to_string(); // On alloue et on copie
s2.push('!'); // On ajoute un caractère, peut-être une réallocation a été nécessaire
```

`String` est vraiment le pendant de `Vec<T>` et `&str` est le pendant de `&[T]`. Mais attention: ce n'est pas tout à fait un tableau de caractères, rappelez-vous : en UTF-8, tous les caractères ne font pas la même taille. Un caractère utf-8 va occuper en mémoire entre 1 et 4 octets. Un `char`, lui occupe 4 octets (pour pouvoir contenir n'importe quel caractère, y compris les plus gros).

C'est un peu pénible en pratique d'avoir deux types pour les chaînes de caractères. Souvent, on se voit obligé de passer de l'un à l'autre. On en arrive à écrire des choses bizarre du genre `"toto".to_string()` parce que `"toto"` est de type `&str` alors qu'on a besoin d'un `String`.

7. Les énumérations et les types somme

7.1. Types énumérés de base

On peut faire un type énuméré.

```
#[derive(Debug, Copy, Clone)]
enum Jour {
  Lundi,
  Mardi,
  Mercredi,
  Jeudi,
  Vendredi,
  Samedi,
  Dimanche,
}

fn main() {
  let j = Jour::Lundi;
  let j2 = j;
  println!("{:?}", j, j2);
}
```

C'est quoi ces `#[derive(Debug, Copy, Clone)]` au début ? Ce sont des directives qui indiquent que l'on peut copier bit à bit (faire (`j = j1` ici)), cloner (faire un `j.clone()` par exemple, ce qui, ici, reviendrait à faire une copie bit à bit), et afficher en mode Debug les valeurs de type `Jour`. Les types `Copy`, `Clone` et `Debug` sont des ***traits***, on verra plus tard de quoi il s'agit exactement.

Pour tester les valeurs d'une énumération, on utilisera le pattern matching. En gros, `match`, c'est l'équivalent de `switch` dans d'autres langages, mais un `switch` boosté aux hormones (on verra plus tard pourquoi).

```
fn est_weekend(j: Jour) -> bool {
  match j {
    Jour::Samedi => true,
    Jour::Dimanche => true,
    _ => false
  }
}
```

L'avantage du `match`, c'est qu'on doit tester *tous* les cas. Si on en oublie, le compilateur refusera le programme.

```
fn fonction_bizarre(j: Jour) -> i32 {
  match j {
    Jour::Lundi    => 16,
    Jour::Mercredi => 21,
    Jour::Jeudi    => 11,
    Jour::Vendredi => 17,
    Jour::Samedi   => 23,
    Jour::Dimanche => 36
  }
}
```

```
error[E0004]: non-exhaustive patterns: `Jour::Mardi` not covered
  --> src/lib.rs:13:9
   |
13 |   match j {
```

```
   |         ^ pattern `Jour::Mardi` not covered
   |
note: `Jour` defined here
  --> src/lib.rs:2:6
   |
 2 | enum Jour {
   |      ^^^^
 3 |   Lundi,
 4 |   Mardi,
   |   ----- not covered
   = note: the matched value is of type `Jour`
help: ensure that all possible cases are being handled by adding a match arm with a
wildcard pattern or an explicit pattern as shown
   |
19 ~     Jour::Dimanche => 36,
20 +     Jour::Mardi => todo!()
   |
```

Le compilateur nous dit que nous n'avons pas traité le cas `Mardi`, et nous suggère, faute de mieux, d'utiliser la macro `todo!()` qui paniquera à l'exécution si on l'atteint.

7.1.1. `if let`

Parfois, je ne veux traiter qu'une seule valeur possible de l'énumération : si la valeur est X, alors je fais quelque chose, dans tous les autres cas… eh bien, je n'ai rien de particulier à faire.

Par exemple, je veux mettre en place une sorte de rappel : le mardi, j'ai cours de Rust. Si on est mardi, je veux donc que mon programme me le rappelle. Et c'est tout. Dans les autres cas, je ne veux rien faire. En principe, je devrait faire ça.

```
let j = aujourdhui();
match j {
  Jour::Mardi => {
    println!("On est mardi : c'est jour de Rust !");
  },
  _ => {}
}
```

C'est un peu verbeux ! Moi je veux pouvoir dire "*si* on est Mardi, alors fais ceci", je n'ai pas envie de faire un gros `match`de 6 lignes.

Dans ce cas (fréquent), on va utiliser la notation `if let` (insérer ici un jeu de mot désopilant sur les tartiflettes) qui fait du pattern matching dans un `if` :

```
let j = aujourdhui();
if let Jour::Mardi = j {
  println!("On est mardi : c'est jour de Rust !");
}
```

7.2. Les types somme

Avec les enums, on peut aussi faire des types somme, c'est-à-dire des énumérations dont les branches sont associées à des valeurs.

Exemple : je veux identifier des gens dans mon application. Mais il y a trois façons de faire :

- un prénom et un nom,
- un pseudo,
- ou bien simplement être anonyme.

```rust
#[derive(Debug, Clone)]
enum Personne {
  PrenomNom(String, String),
  Pseudo(String),
  Anonyme
}

fn personne_to_string(p: &Personne) -> String {
    match p {
        Personne::PrenomNom(prenom, nom) => format!("{prenom} {nom}"),
        Personne::Pseudo(pseudo) => pseudo.clone(),
        Personne::Anonyme => "<anonyme>".to_string(),
    }
}

fn main() {
  let t: [Personne; 5] = [
    Personne::Anonyme,
    Personne::Pseudo("toto".to_string()),
    Personne::Pseudo("titi".to_string()),
    Personne::Anonyme,
    Personne::PrenomNom("Fabien".to_string(), "Delorme".to_string()),
  ];
  for i in 0..t.len() {
    println!("{}", personne_to_string(&t[i]));
  }
  for p in t {
    println!("{}", personne_to_string(&p));
  }
}
```

Note : ici, le type ne peut pas être `Copy` parce qu'on utilise des `String`, et les `String` ne sont pas `Copy` : ils sont alloués sur le tas.

Allez, faisons un `if let` pour bien enfoncer le clou. Je veux compter le nombre de personnes qui ont le prénom "Fabien" parmi tous mes utilisateurs :

```rust
fn compter_fabien(lst: &[Personne]) -> usize {
  let mut res = 0;
  for p in lst {
    if let Personne::PrenomNom(prenom, _) = p && prenom == "Fabien" {
      res += 1;
    }
  }
  res
}
```

7.3. Le type `Option<T>`

Je veux créer une liste d'utilisateurs. Je demande à l'utilisateur s'il souhaite ajouter un nouveau nom, puis de saisir ledit nom. Une fois qu'il a saisi tous les noms, je cherche lequel est le plus long, et je l'affiche. Comment faire ?

```rust
use std::io;

fn read_names() -> Vec<String> {
  let mut names: Vec<String> = Vec::new();
  loop {
```

```
    println!("Souhaitez-vous ajouter un nom ? (o/n)");
    let mut response = String::new();
    io::stdin().read_line(&mut response).expect("erreur de lecture");
    response = response.trim().to_string();
    if response == "o" {
      println!("Entrez le nom");
      let mut nom = String::new();
      io::stdin().read_line(&mut nom).expect("erreur de lecture");
      nom = nom.trim().to_string();
      names.push(nom);
    } else {
      break;
    }
  }
  names
}

fn nom_le_plus_long(names: &[String]) -> String {
  let mut max: &str = &names[0];
  let mut max_len: usize = max.len();
  for name in names {
    if name.len() > max_len {
      max = name;
      max_len = name.len();
    }
  }
  max.to_string()
}

fn main() {
  let names = read_names();
  println!("Le nom le plus long est {}", nom_le_plus_long(&names));
}
```

Pourquoi j'utilise `names: &[String]` et pas `names: Vec<String>` ? Si je faisais ça, alors ma fonction recevrait la propriété des données. Même si ça ne change rien ici, ce n'est pas ce que l'on veut faire.

```
fn nom_le_plus_long(names: Vec<String>) -> String {
  let mut max: &str = &names[0];
  let mut max_len: usize = max.len();
  for name in names {
    if name.len() > max_len {
      max = name;
      max_len = name.len();
    }
  }
  max.to_string()
  // Fin de fonction : le paramètre names sort du scope.
  // appel implicite :
  // drop(names);
}

fn main() {
  let names = read_names();
  println!("Le nom le plus long est {}", nom_le_plus_long(names));
}
```

Ici ça marche parce qu'on ne réutilise plus `names` par la suite, mais si on le voulait, on ne le pourrait pas !

Alors pourquoi ne pas fait `&Vec<String>` plutôt que `&[String]` ? Simplement parce que le second est plus générique et fonctionne aussi sur des références vers des tableaux brut.

Et est-ce que je pourrais renvoyer un `&str` ? Oui, mais ce n'est pas simple, parce qu'il faut réussir à convaincre le compilateur que la référence ne survit pas aux données d'origine. Mais ça c'est compliqué, on verra plus tard.

Revenons à nos moutons. Notre fonction a un problème : si je n'entre pas de nom, je vais avoir une erreur. Quel est le nom le plus long d'une liste vide ? Il n'y en a aucun (*none*).

On doit pouvoir dire que certaines fonctions peuvent ne pas retourner de résultat : soit elle ne retourne rien, soit elle retourne une valeur (laquelle ?)

Il existe un type pour ça : le type `Option<T>`. C'est l'équivalent du type `Maybe` en Haskell. Il a deux valeurs possibles :
- `None` : pas de valeur,
- `Some(x)` : la valeur `x`, où `x` est de type `T`.

```
fn nom_le_plus_long(names: &[String]) -> Option<String> {
  if names.len() == 0 {
    return None;
  }
  let mut max: &str = &names[0];
  let mut max_len: usize = 0;
  for name in names {
    if name.len() > max_len {
      max = name;
      max_len = name.len();
    }
  }
  Some(max.to_string())
  // On ne peut pas juste renvoyer max.to_string() : c'est une valeur de type String
pas Option<String>
}
```

Maintenant, la fonction renvoie un `Option<String>`, qui peut contenir soit `None` soit `Some(s)` où `s` est de type `String`. Pour récupérer cette valeur, ou traiter le cas `None`, on doit désormais faire du ***pattern matching***.

```
fn main() {
  let names = read_names();
  match nom_le_plus_long(&names) {
    None => {
      println!("Il n'y a pas de nom !");
    },
    Some(name) => { // On a récupéré notre nom, enfin !
      println!("Le nom le plus long est {}", name);
    }
  }
}
```

Le fait d'utiliser `Option<T>` nous oblige à traiter le cas `None`. On ne peut pas l'ignorer.

```
fn main() {
  let names = read_names();
```

```
  let name: String = nom_le_plus_long(&names);
  println!("Le nom le plus long est {}", name);
}

error[E0308]: mismatched types
  --> src/main.rs:23:22
   |
23 |   let name: String = nom_le_plus_long(&names);
   |             ------   ^^^^^^^^^^^^^^^^^^^^^^^^ expected `String`, found
`Option<String>`
   |             |
   |             expected due to this
   |
   = note: expected struct `String`
              found enum `Option<String>`
```

Et quand on fait un `match`, il est nécessaire de traiter tous les cas :

```
fn main() {
  let names = read_names();
  match nom_le_plus_long(&names){
    Some(name) => {
        println!("Le nom le plus long est {}", name);
    }
  }
}

error[E0004]: non-exhaustive patterns: `None` not covered
   --> src/main.rs:23:9
    |
 23 |   match nom_le_plus_long(&names){
    |         ^^^^^^^^^^^^^^^^^^^^^^^^ pattern `None` not covered
    |
```

C'est une très bonne pratique qui évite bien des bugs !

En C/C++, on utiliserait sans doute un pointeur, qui serait nul s'il n'y a pas de valeur. En (mauvais) Java, pareil, on renverrait sans doute `null`, et en python, `None`. Charge ensuite à l'utilisateur de bien penser à vérifier s'il a obtenu une valeur ou non. Et s'il ne le fait pas ? Alors on aura une `NullPointerException` ou quelque chose d'équivalent. Exception qui sera peut-être levée bien plus tard dans l'exécution du programme. Et si je mettais cette valeur nulle dans une liste qui s'attend à ce qu'aucune valeur ne soit nulle ? Et si on ne lisait pas le contenu de cette liste avant plusieurs heures ? Alors le bug mettrait des heures à être découvert.

C'est un comportement très coûteux (cf l'article "The billion dollar mistake"). Les pointeurs nuls sont considérés par certains comme un défaut majeur de conception dans un langage de programmation.

Il n'y a pas de pointeur nul en rust.

7.3.1. Décapsuler un `Option<T>` qui n'est pas vide

Parfois, on est sûr que la valeur de notre option est `Some(x)`. On sait que ça ne peut pas être `None`. Dans ce cas, on peut utiliser `unwrap()` pour récupérer plus facilement la valeur.

```
fn main() {
  let mut names = read_names();
  names.push("Toto".to_string());
  let name = nom_le_plus_long(&names).unwrap();
```

```
  println!("Le nom le plus long est {}", name);
}
```

Ici je sais qu'il y aura *forcément* un nom dans la liste. Je sais donc que le cas `None` ne peut pas se produire. Faire un `unwrap()` m'évite toute la machinerie du `match`, ici inutile.

Mais évidemment, attention à ne pas en abuser ! Si je tombe sur un `None` qaund je fais un `unwrap()` alors le programme va paniquer. Ce serait l'équivalent d'une `NullPointerException`, et on sait que c'est grave. On ne veut pas de ça.

7.4. Les erreurs

Il y a un type qui ressemble à `Option<T>` et que l'on utilise très souvent : le type `Result<T,U>`. Il est utilisé pour indiquer qu'une fonction peut renvoyer soit un résultat normal, soit une erreur (si elle échoue). Le premier type est le type de retour attendu, le second est un type d'erreur.

```
enum Result<T,U> {
  Ok(T),
  Err(U),
}

fn division_entière(a: i32, b: i32) -> Result<i32,String> {
  if b != 0 {
    Ok(a / b)
  } else {
    Err("division par zéro".to_string())
  }
}

fn division_entière_mieux(a: i32, b: i32) -> Result<(i32,i32),String> {
  if b != 0 {
    Ok((a / b, a % b))
  } else {
    Err("division par zéro".to_string())
  }
}

fn pgcd(a: i32, b: i32) -> Result<i32,String> {
  let mut a = a; // Shadowing pour pouvoir modifier a : on oublie le a d'origine
  let mut b = b; // Idem
  while a != 0 {
    if a < b {
      let tmp = b;
      b = a;
      a = tmp;
    }
    match division_entière_mieux(a, b) {
      Ok((_, n)) => {
        a = n;
      },
      Err(msg) => {
        return Err(msg);
      },
    }
  }
  Ok(b)
}
```

```
fn main() {
    match pgcd(15,0) {
        Ok(n) => {
            println!("Le PGCD est {}", n);
        },
        Err(msg) => {
            println!("impossible de calculer le PGCD: {}", msg);
        }
    }
}
```

Ici, je gère l'erreur chaque fois qu'elle peut arriver. Le fait que `division_entière_mieux` renvoie un `Result` m'oblige soit à traiter l'erreur localement, soit à la retourner à la fonction appelante. Je ne peux *pas* l'ignorer. Là encore, c'est une mesure de sécurité. En java / python, au mieux, on lèverait une exception et on laisserait le programme planter gentiment, parce que l'on aurait la flemme de gérer l'exception. Ou bien, on la laisserait remonter la pile d'appels et on la traiterait tardivement, et mal.

On a tendance à considérer que les erreurs sont exceptionnelles et ne font pas vraiment partie du programme, mais sont simplement un truc qui nous embête.

C'est faux, écrire un programme, c'est principalement gérer des erreurs potentielles. Dans certains programmes, notamment ceux qui utilisent des données transmises par l'utilisateur, l'essentiel du programme consiste à traiter des cas d'erreur. C'est une partie pénible et pas très fun, mais il faut le faire.

`Result` nous oblige donc à le faire. Mais la syntaxe est un peu lourde. Est-ce qu'on ne pourrait pas alléger un peu tout ça ? Très souvent, on veut juste renvoyer l'erreur à l'appelant quand elle survient. Comment faire ?

On peut utiliser la syntaxe `?`

```
fn pgcd(a: i32, b: i32) -> Result<i32,String> {
  let mut a = a; // Shadowing pour pouvoir modifier a : on oublie le a d'origine
  let mut b = b; // Idem
  while a != 0 {
    if a < b {
      let tmp = b;
      b = a;
      a = tmp;
    }
    (_, a) = division_entière_mieux(a, b)?;
  }
  Ok(b)
}
```

Le `?` est du sucre syntaxique qui simplifie énormément les écritures. Mais en arrière-plan, il fait la même chose que notre `match`.

L'avantage, c'est que l'on matérialise clairement les fonctions qui peuvent échouer, et les lignes au sein de ces fonctions qui peuvent y échouer.

On peut les chaîner. Imaginons qu'on veuille enchaîner plusieurs actions dont chacune peut échouer. On peut faire :

```
Utilisateur::new(name)?.ajouter_score(10)?.donner_les_droits_admin()?;
```

Ici on a trois méthodes qui peuvent échouer. Sans la syntaxe `?`, il faudrait imbriquer trois `match`.

Évidemment, on ne peut utiliser le `?` que dans les fonctions qui renvoient un `Result<T,U>`. Cela n'aurait pas de sens ailleurs.

7.4.1. `expect()`

Parfois, quand une erreur survient, on ne sait pas quoi faire, et on souhaite simplement paniquer.

Cela arrive aussi quand on est encore en phase de développement et que l'on ne souhaite pas sortir l'artillerie lourde tout de suite. Il ne faut donc pas en abuser.

```
fn pgcd(a: i32, b: i32) -> i32 {
  let mut a = a;
  let mut b = b;
  while a != 0 {
    if a < b {
      let tmp = b;
      b = a;
      a = tmp;
    }
    (_, a) = division_entière_mieux(a, b).expect("erreur de calcul du PGCD");
  }
  b
}

fn main() {
    println!("Le PGCD est {}", pgcd(15,0));
}
```

C'est plus léger, mais ce n'est à utiliser que dans les cas suivants :
- phase de prototypage, pas en prod,
- on ne sait pas quoi faire de cette erreur,
- erreur qui n'a que très peu de chances de se produire.

7.4.2. Autres choses que l'on peut faire avec des `Result<T,U>` (et aussi des `Option<T>`)

Parfois, on sait que l'appel ne peut pas échouer. On ne veut donc pas s'embêter avec le retour `Err(U)` : ça ne peut pas arriver. On va donc utiliser `unwrap()` pour récupérer la valeur attendue.

C'est la même chose que `expect()`, mais sans le message d'erreur associé. Comme `expect`, ce n'est à utiliser que lorsque l'on est sûr que la valeur est `Ok` ou `Some`, et que l'on considère qu'il y aurait forcément un bug quelque part dans le programme dans le cas contraire.

```
(_, a) = division_entière_mieux(a, b).unwrap();
```

Parfois, on veut simplement récupérer une valeur par défaut en cas d'erreur. Dans ce cas on utilise `unwrap_or`. Attention : parfois ça a un sens, parfois c'est incorrect. Dans notre cas, ce serait ridicule.

```
(_, a) = division_entière_mieux(a, b).unwrap_or((0, 0));
// Ridicule : le PGCD de (15, 0) n'est pas 0 : il est indéfini. C'est forcément une
erreur !
```

7.4.3. `panic!()`

Parfois, une erreur grave survient, tellement grave ou difficile à gérer que cela n'a pas de sens de poursuivre l'exécution du programme. Dans ce cas, on va paniquer. Cela veut dire que l'on va faire remonter une erreur que les fonctions n'ont pas à gérer. On le réserve aux cas vraiment exceptionnels : il n'y a pas grand-chose d'autre à faire que de quitter proprement le programme.

Mais la plupart du temps, on va utiliser des `Result<T,U>` pour gérer les erreurs. Oui, c'est pénible à gérer. C'est le prix à payer pour avoir des programmes robustes, corrects et efficaces.

8. Les structures de données

On peut rassembler des données dans une structure.

```
struct Joueur {
  nom: String,
  score: i32,
  vies: i32,
}

fn afficher_joueur(j: &Joueur) {
  println!("{} a {} points et {} vies", j.nom, j.score, j.vies);
}
```

Par défaut, on ne peut ni copier, ni afficher (en mode debug), ni cloner un struct.

Faisons des essais qui vont échouer.

```
fn main() {
  let j = Joueur {
    score: 0,
    vies : 3,
  }; // J'ai oublié de donner un nom
}
```

```
error[E0063]: missing field `name` in initializer of `Joueur`
 --> src/main.rs:9:13
  |
9 |     let j = Joueur {
  |             ^^^^^^ missing `name`
```

```
fn main() {
  let j = Joueur {
    score: 0,
    vies : 3,
    nom: "Toto", // "Toto" est de type &str, pas String
  };
}
```

```
error[E0308]: mismatched types
  --> src/main.rs:12:14
   |
12 |         nom: "Toto",
   |              ^^^^^^ expected `String`, found `&str`
   |
help: try using a conversion method
   |
12 |         nom: "Toto".to_string(),
   |                     ++++++++++++
```

```
fn main() {
  let j = Joueur {
    score: 0,
    vies: 3,
    nom: "Toto".to_string(),
  };
  println!("{:?}", j);
}
```

```
error[E0277]: `Joueur` doesn't implement `Debug`
  --> src/main.rs:14:22
   |
14 |     println!("{:?}", j);
   |                ----   ^ `Joueur` cannot be formatted using `{:?}` because it
doesn't implement `Debug`
   |                |
   |                required by this formatting parameter
   |
   = help: the trait `Debug` is not implemented for `Joueur`
   = note: add `#[derive(Debug)]` to `Joueur` or manually `impl Debug for Joueur`
help: consider annotating `Joueur` with `#[derive(Debug)]`
   |
 2 + #[derive(Debug)]
 3 | struct Joueur {
   |
```

Quoi ? C'est quoi cette histoire de `Debug` ?

En fait, pour pouvoir être affiché avec `{:?}`, un type doit implémenter le ***trait*** `Debug`. C'est quoi un trait ? On peut voir ça comme une interface Java. En gros, cela veut dire que le type concerné doit implémenter certaines méthodes.

On peut aller au plus simple et appeler le débugger par défaut. Pour cela, au-dessus de la définition du type, il faut ajouter les directives de compilation suivantes :

```
#[derive(Debug)]
struct Joueur {
  ...
```

Continuons à échouer.

```
fn main() {
  let j1 = Joueur {score: 0, vies: 3, nom: "Toto".to_string()};
  let j2 = j1;
  println!("{:?}", j1);
}

error[E0382]: borrow of moved value: `j1`
  --> src/main.rs:11:22
   |
 9 |     let j1 = Joueur { score: 0, vies: 3, nom: "Toto".to_string() };
   |         -- move occurs because `j1` has type `Joueur`, which does not implement
the `Copy` trait
10 |     let j2 = j1;
   |              -- value moved here
11 |     println!("{:?}", j1);
   |                      ^^ value borrowed here after move
   |
note: if `Joueur` implemented `Clone`, you could clone the value
```

Oh non ! En faisant `j2 = j1`, j'ai déplacé (move) `j1` dans `j2`, donc maintenant, `j1` n'est plus valide. Mais moi, je voulais faire un clone. C'est vrai ça, le compilateur m'a donné une idée.

```
let j2 = j1.clone();

error[E0599]: no method named `clone` found for struct `Joueur` in the current scope
  --> src/main.rs:10:17
```

```
   |
 2 | struct Joueur {
   | ------------- method `clone` not found for this struct
...
10 |     let j2 = j1.clone();
   |                 ^^^^^ method not found in `Joueur`
   |
   = help: items from traits can only be used if the trait is implemented and in
scope
   = note: the following trait defines an item `clone`, perhaps you need to implement
it:
           candidate #1: `Clone`
```

Même problème qu'avec `Debug` : pour pouvoir cloner un type, il doit implémenter `Clone` :

```
#[derive(Debug, Clone)]
struct Joueur {
  ...
```

Et là cela marchera.

8.1. Implémenter un type

On peut associer des méthodes à un type, et utiliser les grands principes de la programmation objet.

```
impl Joueur {
  fn new(nom: &str) -> Joueur {
    Joueur { nom: nom.to_string(), score: 0, vies: 3 }
  }

  fn get_score(&self) -> i32 {
    self.score
  }

  fn get_nom(&self) -> String {
    self.nom
  }

  fn est_vivant(&self) -> bool {
    self.vies > 0
  }

  fn perdre_vie(&mut self) {
    self.vies -= 1;
  }

  fn gagner_des_points(&mut self, score_en_plus: i32) {
    self.score += score_en_plus;
  }
}

fn main() {
  let mut j = Joueur::new("toto");
  let mut points = 1000;
  while j.est_vivant() {
    j.gagner_des_points(points);
    j.perdre_vie();
    points *= 2;
```

```
  }
  println!("{} est mort, mais il a valeureusement gagné {} points", j.get_nom(),
j.get_score());
}

error[E0507]: cannot move out of `self.nom` which is behind a shared reference
  --> src/main.rs:18:5
   |
18 |     self.nom
   |     ^^^^^^^^ move occurs because `self.nom` has type `String`, which does not
implement the `Copy` trait
   |
help: consider cloning the value if the performance cost is acceptable
   |
18 |     self.nom.clone()
   |              ++++++++
```

Bon, OK, je fais ce qu'il dit, mais c'est relou : à chaque fois que j'appelle `get_nom()`, je dois créer une nouvelle chaîne de caractères sur le tas !

Est-ce que je ne pourrais pas simplement emprunter la valeur du nom, via une référence ? Promis, je la rends après.

Oui, on peut : ici on est sûr que `self.nom` vivra aussi longtemps que `self`, donc il n'y aura pas de problème de durée de vie.

```
fn get_nom(&self) -> &str {
  &self.nom
}
```

Ici, on ne fait pas de copie inutile. Pas d'allocation ni de désallocation couteuse. C'est mieux ! Mais parfois on peut se heurter à des problèmes de durée de vie.

8.2. Les types génériques

Je veux faire une fonction `empty` qui prend en paramètre une référence vers un tableau et dit s'il est vide ou non.

```
fn empty(tab: &[i32]) -> bool {
  tab.len() == 0
}
```

Problème : cette fonction ne peut être utilisée que sur des `&[i32]`. J'aimerais aussi l'utiliser pour vérifier si mon tableau de `f64` est vide. Comment faire pour utiliser ma fonction sur n'importe quel type `T`? On va utiliser un type générique.

```
fn empty<T>(tab: &[T]) -> bool {
  tab.len() == 0
}

fn main() {
  println!("{}", empty(&[1, 2, 3])); // type &[i32]
  println!("{}", empty(&["toto"]));  // type &[&str]
  println!("{}", empty(&[]));        // Oui c'est vide, mais c'est quoi le type de
T ?
}

error[E0282]: type annotations needed
  --> src/main.rs:16:20
```

```
   |
16 |     println!("{}", empty(&[]));
   |                     ^^^^^ --- type must be known at this point
   |                     |
   |                     cannot infer type of the type parameter `T` declared on the
function `empty`
   |
help: consider specifying the generic argument
   |
16 |     println!("{}", empty::<T>(&[]));
   |                          +++++
```

On doit utiliser l'opérateur *turbofish* `::<` pour d'obscures raisons de compilation : ici `empty<T>` serait trop difficile à compiler, donc on doit écrire `empty::<T>`.

```
fn main() {
  println!("{}", empty(&[1, 2, 3])); // type &[i32]
  println!("{}", empty(&["toto"]));  // type &[&str]
  println!("{}", empty::<i32>(&[])); // Par exemple
  let tab: &[char] = &[]; // Ça marche aussi
  println!("{}", empty(tab));
  let tab2: Vec<bool> = vec![];
  println!("{}", empty(&tab2));
}
```

Mais parfois, certaines fonctions génériques ne peuvent s'appliquer qu'à certains types. Par exemple, on peut copier bit à bit des types simples (entiers, booléens, caractères, tableaux bruts, enum simples, etc.) mais pas les autres : ces types implémentent `Copy`.

```
fn head<T>(t: &[T]) -> Option<T> {
  if empty(t) {
    None
  } else {
    Some(t[0])
  }
}
```

```
error[E0508]: cannot move out of type `[T]`, a non-copy slice
 --> src/main.rs:9:14
  |
9 |         Some(t[0])
  |              ^^^^
  |              |
  |              cannot move out of here
  |              move occurs because `t[_]` has type `T`, which does not implement
the `Copy` trait
  |
help: if `T` implemented `Clone`, you could clone the value
 --> src/main.rs:5:9
  |
5 | fn head<T>(t: &[T]) -> Option<T> {
  |         ^ consider constraining this type parameter with `Clone`
...
9 |         Some(t[0])
  |              ---- you could clone this value
```

Ici on a deux suggestions: soit utiliser un type `Copy`, soit utiliser un type `Clone`. Les types `Clone` sont plus nombreux, mais ça implique de faire un `.clone()`, qui peut être couteux.

```
fn head<T: Copy>(tab: &[T]) -> Option<T> {
  if est_vide(tab) {
    None
  } else {
    Some(tab[0])
  }
}

fn main() {
  println!("{:?}", head(&[1, 2, 3]));
  println!("{:?}", head(&["toto"]));  // type &[&str]
  println!("{:?}", head::<i32>(&[])); // Par exemple
  let tab: &[char] = &[]; // Ça marche aussi
  println!("{:?}", head(tab));
  let tab2: Vec<bool> = vec![];
  println!("{:?}", head(&tab2));
  let tab_tab: [Vec<bool>;2] = [vec![], vec![true, false]];
  println!("{:?}", head(&tab_tab));
}
```

```
error[E0277]: the trait bound `Vec<bool>: Copy` is not satisfied
  --> src/main.rs:22:25
   |
22 |   println!("{:?}", head(&tab_tab));
   |                     ---- ^^^^^^^^ the trait `Copy` is not implemented for
`Vec<bool>`
   |                     |
   |                     required by a bound introduced by this call
   |
note: required by a bound in `head`
  --> src/main.rs:5:12
   |
 5 | fn head<T: Copy>(t: &[T]) -> Option<T> {
   |            ^^^^ required by this bound in `head`
```

Impossible d'appeler `head(&tab_tab)` car `&tab_tab` est de type `&[Vec<bool>]`, or `Vec<bool>` n'est pas `Copy`. Si on veut une fonction `head` qui marche avec tous les types clonables, il faut changer la définition :

```
fn head<T: Clone>(t: &[T]) -> Option<T> {
    if empty(t) {
        None
    } else {
        Some(t[0].clone())
    }
}
```

On fait un `.clone()`. Ce n'est pas gratuit, mais souvent c'est un coût acceptable pour simplifier l'écriture des programmes.

Comment pourrait-on faire pour des types qui ne seraient pas `Clone` ? Il faudrait utiliser des références, mais là, attention aux durées de vie.

```
fn head2<T>(t: &[T]) -> Option<&T> {
    if empty(t) {
        None
    } else {
        Some(&t[0])
```

```
    }
}

fn main() {
  let mut s1: String = String::new();
  let mut s2: &str = &"";
  {
    let mut s3: &str = &"";
    let tab = &["toto".to_string()];
    // On fait des .unwrap() parce qu'on est sûr qu'on aura des Some(T)
    s1 = head(tab).unwrap(); // On fait un .clone()
    s2 = head2(tab).unwrap();
    s3 = head2(tab).unwrap();
    println!("{}", s1);
    println!("{}", s2);
    println!("{}", s3);
  }
  println!("{}", s1);
  println!("{}", s2);
}
```

```
error[E0716]: temporary value dropped while borrowed
  --> src/main.rs:26:16
   |
26 |     let tab = &["toto".to_string()];
   |                ^^^^^^^^^^^^^^^^^^^^ creates a temporary value which is freed
while still in use
...
33 |   }
   |   - temporary value is freed at the end of this statement
34 |   println!("{}", s1);
35 |   println!("{}", s2);
   |                   -- borrow later used here
   |
```

Eh oui : les références, c'est génial, c'est beaucoup plus léger qu'un `.clone()`, mais on a rapidement des problèmes de durée de vie. En rust, le compilateur nous aide à détecter ces erreurs, en C/C++ on tombe sur une vulnérabilité difficile à détecter.

Moralité : à moins d'avoir besoin de performance à un endroit donné, le plus sage / simple la plupart du temps, c'est de commencer en faisant un clone.

8.2.1. Types génériques

On a fait des fonctions génériques, créons maintenant un type générique avec des méthodes associées.

Un dictionnaire, c'est un ensemble de paires clé/valeur. La clé est une chaîne de caractères, la valeur est n'importe quel type clonable. Une paire, c'est donc un tuple de deux éléments, une chaîne, et un `T` clonable.

```
type Pair<T: Clone> = (String, T);

struct Dict<T: Clone> (Vec<Pair<T>>);
```

Ici j'ai défini `Dict<T>` comme étant un tuple qui contient un seul élément.

```
impl<T: Clone> Dict<T> {
  fn new() -> Dict<T> {
```

```
    Dict(vec![])
  }

  fn len(&self) -> usize {
    self.0.len()
  }
}

fn main() {
  let d: Dict<String> = Dict::new();
  println!("d.len()={}", d.len());
}
```

Jusqu'ici, tout fonctionnerait même avec des type non `Clone`, évidemment. Continuons.

```
impl<T: Clone> Dict<T> {
  ...

  fn get(&self, key: &str) -> Option<T> {
      for p in &self.0 {
          if p.0 == key {
              return Some(p.1.clone());
          }
      }
      None
  }

  fn set(&mut self, key: &str, value: T) {
      for p in &mut self.0 {
          if p.0 == key {
              p.1 = value;
              return;
          }
      }
      self.0.push((key.to_string(), value));
  }
}

fn main() {
    let mut d: Dict<String> = Dict::new();
    d.set("toto", "titi".to_string());
    d.set("toto", "tata".to_string());
    println!("d.len()={}", d.len());
    println!("{:?}", d.get("toto"));
}
```

Beaucoup de subtilités : parfois on doit faire `.to_string()`, parfois non. Heureusement, le compilateur nous guide à chaque fois.

Maintenant, j'aimerais avoir un dictionnaire qui ne prend pas forcément des chaînes comme clés, mais n'importe quel type qui est comparable.

```
type Pair<K, T: Clone> = (K, T);

struct Dict<K, T: Clone> (Vec<Pair<K,T>>);

impl<K, T: Clone> Dict<K,T> {
  fn new() -> Dict<K,T> {
```

```
    Dict(vec![])
  }

  fn len(&self) -> usize {
      self.0.len()
  }

  fn get(&self, key: K) -> Option<T> {
      for p in &self.0 {
          if p.0 == key {
              return Some(p.1.clone());
          }
      }
      None
  }

  fn set(&mut self, key: K, value: T) {
      for p in &mut self.0 {
          if p.0 == key {
              p.1 = value;
              return;
          }
      }
      self.0.push((key, value));
  }
}

fn main() {
    let mut d: Dict<String, String> = Dict::new();
    d.set("toto".to_string(), "titi".to_string());
    d.set("toto".to_string(), "tata".to_string());
    println!("d.len()={}", d.len());
    println!("{:?}", d.get("toto".to_string()));
}

error[E0369]: binary operation `==` cannot be applied to type `K`
  --> src/main.rs:16:18
   |
16 |           if p.0 == key {
   |              --- ^^ --- K
   |              |
   |              K
   |
help: consider restricting type parameter `K` with trait `PartialEq`
   |
 5 | impl<K: std::cmp::PartialEq, T: Clone> Dict<K,T> {
   |       ++++++++++++++++++++++

error[E0369]: binary operation `==` cannot be applied to type `K`
  --> src/main.rs:25:18
   |
25 |           if p.0 == key {
   |              --- ^^ --- K
   |              |
   |              K
   |
```

```
help: consider restricting type parameter `K` with trait `PartialEq`
   |
 5 | impl<K: std::cmp::PartialEq, T: Clone> Dict<K,T> {
   |       ++++++++++++++++++++
```

La clé doit être comparable, on doit pouvoir faire un == dessus. Pour cela, il faut que le type implément le trait `PartialEq` :

```
type Pair<K: PartialEq, T: Clone> = (K, T);

struct Dict<K: PartialEq, T: Clone> (Vec<Pair<K,T>>);

impl<K: PartialEq, T: Clone> Dict<K,T> {
  ...
```

Et maintenant, on peut utiliser n'importe quel type `PartialEq` en tant que clé. Par exemple, `Vec<bool>` est `PartialEq`, on peut donc l'utiliser.

```
fn main() {
    let mut d: Dict<Vec<bool>, String> = Dict::new();
    d.set(vec![true,false], "titi".to_string());
    d.set(vec![true,false], "tata".to_string());
    println!("d.len()={}", d.len());
    println!("{:?}", d.get(vec![true]));
}
```

8.3. Les traits

C'est quoi ces histoires de traits ? En fait, un trait, c'est un ensemble de méthodes qu'un type s'engage à implémenter. C'est un peu l'équivalent des interfaces en java, mais en un peu plus puissant.

Par exemple, imaginons que je veuille définir un moyen d'avoir une valeur par défaut pour mon dictionnaire. La valeur par défaut, c'est simplement le dictionnaire vide. Je vais donc implémenter le trait `Default`, et pour ce faire, je vais avoir besoin d'implémenter une méthode `default()`.

```
impl<K: PartialEq, T: Clone> Default for Dict<K,T> {
  fn default() -> Self {
    Dict::new()
  }
}
```

Ici, `Self` veut dire "le type actuel". C'est une façon plus générique de dire "ma fonction renvoie une valeur du type actuel".

À quoi ça sert ? Ça peut servir avec le type `Option<T>`, par exemple. Si `T` est `Default`, alors on peut utiliser la méthode `unwrap_or_default()` :

```
fn main() {
  let opt1 = Some(1);
  let opt2: Option<i32> = None;
  println!("{}", opt1.unwrap());
  println!("{}", opt2.unwrap());
}

1
thread 'main' (79) panicked at src/main.rs:38:25:
called `Option::unwrap()` on a `None` value
```

J'ai appelé `unwrap()` sur un `None`. C'est une catastrophe, le programme plante, j'aurais dû gérer mon code mieux et faire un pattern matching. Je suis un mauvais programmeur.

```
fn main() {
    let opt1 = Some(1);
    let opt2: Option<i32> = None;
    println!("{}", opt1.unwrap_or_default());
    println!("{}", opt2.unwrap_or_default());
}
```

```
1
0
```

Ça marche ! Je lui ai dit de prendre la valeur par défaut en cas de None. Attention, ce n'est pas une façon correcte de gérer toutes les `Option` : parfois, `None` et `Some(0)` n'ont pas la même sémantique et on ne peut pas les ignorer. On l'a vu avec les PGCD plus haut.

La fonction `unwrap_or_default` marche pour les types qui sont `Default`, mais pas pour les autres.

```
struct Joueur {
    nom: String,
    score: i32,
    vies: i32,
}

fn main() {
    let opt: Option<Joueur> = None;
    println!("Le nom du joueur par défaut est {}", opt.unwrap_or_default().nom);
}
```

```
error[E0277]: the trait bound `Joueur: Default` is not satisfied
    --> src/main.rs:42:56
     |
  42 |     println!("Le nom du joueur par défaut est {}",
opt.unwrap_or_default().nom);
     |                                                        ^^^^^^^^^^^^^^^^^ the
trait `Default` is not implemented for `Joueur`
     |
note: required by a bound in `Option::<T>::unwrap_or_default`
    --> /playground/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/
src/rust/library/core/src/option.rs:1094:12
     |
1092 |     pub const fn unwrap_or_default(self) -> T
     |                   ----------------- required by a bound in this associated
function
1093 |     where
1094 |         T: [const] Default,
     |            ^^^^^^^^^^^^^^^ required by this bound in
`Option::<T>::unwrap_or_default`
```

Il faut donc que mon type `Joueur` implémente `Default`:

```
impl Default for Joueur {
    fn default() -> Self {
        Joueur {
            nom: "<anonyme>".to_string(),
            score: 0,
            vies: 3,
```

```
        }
    }
}
```

Pour certains traits très communs, il existe une implémentation par défaut du trait. Si on veut l'utiliser, on peut utiliser la directive de compilation `#derive`. On l'utilise souvent pour `Copy`, `Clone`, `Debug`, `Default` et d'autres.

```
#[derive(Debug, Default)]
struct Joueur {
    nom: String,
    score: i32,
    vies: i32,
}


fn main() {
    let opt: Option<Joueur> = None;
    println!("Le joueur par défaut est {:?}", opt.unwrap_or_default());
}
```

```
Le joueur par défaut est Joueur { nom: "", score: 0, vies: 0 }
```

Magnifique ! Les traits et la généricité sont complexes à maîtriser, mais ils ont l'avantage d'éviter de dupliquer du code, et cela en toute sécurité (le compilateur veille !) et sans coût supplémentaire à l'exécution : un type générique n'est pas plus lent ni ne consomme plus de mémoire qu'un type standard. Par contre, la compilation est un peu plus lente.

À propos de choses complexes…

9. Les *lifetimes*

Notion inventée par rust, indispensable quand on manipule des références de manière un peu complexe.

Quand on a une référence, on doit garder en tête combien de temps elle vit : elle ne peut pas vivre plus longtemps que la valeur référencée. Mais quand on jongle avec plusieurs références, qui référencent des valeurs qui n'ont pas forcément toutes la même durée de vie, comment ça se passe ?

```
fn la_plus_longue (s1: &str, s2: &str) -> &str {
  if s1.len() > s2.len() {
    s1
  } else {
    s2
  }
}

fn main() {
  println!("{}", la_plus_longue("toto", "titi"));
}
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:43
  |
1 | fn la_plus_longue (s1: &str, s2: &str) -> &str {
  |                        ----      ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature
does not say whether it is borrowed from `s1` or `s2`
help: consider introducing a named lifetime parameter
  |
1 | fn la_plus_longue<'a> (s1: &'a str, s2: &'a str) -> &'a str {
  |                  ++++       ++           ++          ++
```

Ici, le compilateur est perdu : quelle est la durée de vie de chacune des références ? Est-ce qu'il y en a une qui vit plus longtemps que les autres ?

Il va falloir étiqueter les références avec des tags de lifetime. Par exemple, `&'a str` signifie "une référence vers une chaîne, qui vivra un temps identifié comme 'a". Oui, c'est vague, c'est normal, on veut rester assez générique.

Ici, il va falloir que s1 et s2 vivent aussi longtemps l'une que l'autre, et le résultat vivra aussi longtemps que s1 et s2.

```
fn la_plus_longue<'a> (s1: &'a str, s2: &'a str) -> &'a str {
  if s1.len() > s2.len() {
    s1
  } else {
    s2
  }
}

fn main() {
  println!("{}", la_plus_longue("toto", "titi"));
}
```

```
titi
```

Et si mes deux paramètres ont des lifetime concrets différents ? Eh bien, `'a` sera le plus petit des deux.

```
fn main() {
    let s1 = "tototo".to_string();
    let res;
    {
        let s2 = "titi".to_string();
        res = la_plus_longue(&s1, &s2);
    }
    println!("{}", res);
}
```

```
error[E0597]: `s2` does not live long enough
  --> src/main.rs:14:35
   |
13 |         let s2 = "titi".to_string();
   |             -- binding `s2` declared here
14 |         res = la_plus_longue(&s1, &s2);
   |                                   ^^^ borrowed value does not live long enough
15 |     }
   |     - `s2` dropped here while still borrowed
16 |     println!("{}", res);
   |                    --- borrow later used here
```

Ici `'a` est la durée de vie de `s2`, pas celle de `s1`, car `s2` vit moins longtemps que `s1`. Ce qui veut dire que le résultat ne pourra pas vivre plus longtemps que `s2`, quoi qu'il arrive. C'est pour ça que le compilateur refuse notre code (alors que, techniquement, il est correct, mais le compilateur ne peut pas le savoir).

C'est important (et compliqué) quand on stocke des références dans des `struct`.

```
struct Joueur {
  nom: &str,
  score: i32,
  vies: i32,
}
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:2:10
  |
2 |     nom: &str,
  |          ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct Joueur<'a> {
2 ~     nom: &'a str,
  |
```

Eh oui : ici, on a besoin de savoir que la référence vivra… un certain temps, en tout cas au moins aussi longtemps que le `struct` en lui-même.

```
#[derive(Default,Debug,Clone)]
struct Joueur<'a> {
    nom: &'a str,
    score: i32,
    vies: i32,
```

```
}

fn main() {
  let mut j = Joueur::default();
  for i in 0..10 {
    j.score = i; // On fait des copies : aucun souci !
  }
  for _ in 0..10 {
    let nom = "toto".to_string();
    j.nom = &nom; // Erreur : nom ne vit pas assez longtemps !
  }
}

error[E0597]: `nom` does not live long enough
  --> src/main.rs:15:13
   |
14 |     let nom = "toto".to_string();
   |         --- binding `nom` declared here
15 |     j.nom = &nom; // Erreur : nom ne vit pas assez longtemps !
   |             ^^^^ borrowed value does not live long enough
16 |   }
   |   - `nom` dropped here while still borrowed
```

Évidemment, il n'y aura aucun souci ici :

```
fn main() {
  let mut j = Joueur::default();
  for i in 0..10 {
    j.score = i; // On fait des copies : aucun souci !
  }
  let nom = "toto".to_string();
  for _ in 0..10 {
    j.nom = &nom; // nom est défini au-dessus, il vit aussi longtemps que j, aucun
souci.
  }
}
```

9.1. Dictionnaire et refs

On refait notre type `Dict` mais avec des références cette fois. Ce sera plus efficace en temps CPU et en mémoire, mais attention aux subtilités des durées de vie.

```
type Pair<T> (&str, &T);

error[E0106]: missing lifetime specifier
 --> src/lib.rs:1:17
  |
1 | type Pair<T> = (&str, &T);
  |                 ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 | type Pair<'a, T> = (&'a str, &T);
  |           +++        ++

error[E0106]: missing lifetime specifier
 --> src/lib.rs:1:23
  |
```

```
1 | type Pair<T> = (&str, &T);
  |                       ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 | type Pair<'a, T> = (&str, &'a T);
  |           +++              ++
```

On a des refs dans un `struct`. On doit donc dire quel est le lifetime de chaque référence et du type en lui-même.

```
type Pair<'a,T> (&'a str, &'a T);
```

Ça c'est bien, mais ça veut dire que la durée de vie des deux éléments est égale à la plus petite durée de vie. Ça peut poser des problèmes subtils :

```
type Pair<'a, T> = (&'a str, &'a T);

fn get_key<'a, T>(p: Pair<'a, T>) -> &'a str {
    p.0
}

fn main() {
    let name: &str = &String::from("toto");
    let key;
    {
        let age = 42;
        let p = (name, &age);
        key = get_key(p);
        // p sort du scope et est détruit, age sort du scope;
    }
    println!("{}", key); // key vit aussi longtemps que age.
}
```

```
error[E0597]: `age` does not live long enough
  --> src/main.rs:12:24
   |
11 |         let age = 42;
   |             --- binding `age` declared here
12 |         let p = (name, &age);
   |                        ^^^^ borrowed value does not live long enough
...
15 |     }
   |     - `age` dropped here while still borrowed
16 |     println!("{}", key); // key vit aussi longtemps que age,
   |                    --- borrow later used here
```

Il n'y a pas de raison : en théorie, ce que référence `key` vit assez longtemps, mais on a dit au compilateur que `key` avait la même durée de vie que `age`, qui est sorti du scope.

```
type Pair<'a, 'b, T> = (&'a str, &'b T);

fn get_key<'a, 'b, T>(p: Pair<'a, 'b, T>) -> &'a str {
    p.0
}

fn main() {
    let name: &str = &String::from("toto");
```

```
    let key;
    {
        let age = 42;
        let p = (name, &age);
        key = get_key(p);
        // p sort du scope et est détruit, age sort du scope;
        // mais key est encore vivant, il a un lifetime différent.
    }
    println!("{}", key);
}
```

```
toto
```

Eh oui, c'est méga-chaud à bien comprendre, c'est sans doute l'aspect le plus difficile de rust. Mais c'est ce qui permet d'avoir à la fois :

- une sûreté du typage
- pas de garbage collector ni aucun autre dispositif coûteux en temps CPU / en RAM,
- un dispositif permettant de ne pas devoir cloner à tout va.

Quand on développe, pour éviter l'optimisation prématurée, il est conseillé d'utiliser des `.clone()` généreusement même si ce n'est pas optimal, pour éviter les problèmes de lifetime.

9.2. Le lifetime `'static`

Il y a des valeurs qui existent tout au long de la vie du programme. Ce sont les littéraux qui sont déclarés tels quels dans le programme.

Notamment, les chaînes de caractères litérales sont statiques.

```
fn main() {
  let r;
  {
    r = &"titi";  // "titi" est 'static
  }
  println!("{}", r);
}
```

```
titi
```

```
fn main() {
  let r: &str;
  {
    r = &("titi".to_string());
  }
  println!("{}", r);
}
```

```
  error[E0716]: temporary value dropped while borrowed
 --> src/main.rs:4:10
  |
4 |     r = &("titi".to_string());
  |          ^^^^^^^^^^^^^^^^^^^^- temporary value is freed at the end of this
statement
  |          |
  |          creates a temporary value which is freed while still in use
5 |   }
```

```
6 |   println!("{}", r);
  |                   - borrow later used here
  |
```

Ça veut dire qu'on peut contraindre notre paire pour que la clé soit forcément une chaîne statique.

```
type Pair<'a, T> = (&'static str, &'a T);

fn get_key<'a, T> (p: Pair<'a, T>) -> &'static str {
  p.key
}

fn main() {
  let key;
  {
    let name = "toto"; // &'static str
    let p = (name, 42);
    key = p.get_key();
  }
  println!("{}", key);
}

toto

type Pair<'a, T> = (&'static str, &'a T);

fn get_key<'a, T> (p: Pair<'a, T>) -> &'a str {
  p.key
}

fn main() {
  let key;
  {
    let name = "toto"; // &'static str
    let p = (name, 42);
    key = p.get_key();
  }
  println!("{}", key);
}

error[E0597]: `val` does not live long enough
  --> src/main.rs:12:20
   |
11 |     let val = 42;
   |         --- binding `val` declared here
12 |     let p = (name, &val);
   |                    ^^^^ borrowed value does not live long enough
13 |     key = get_key(p);
14 |   }
   |   - `val` dropped here while still borrowed
15 |   println!("{}", key);
   |                  --- borrow later used here
```

Dans le second cas ça ne marche pas car on a contraine le résultat de `get_key` à ne pas vivre plus longtemps que le `struct` en lui-même.

10. Le type Box<T>

10.1. La liste chaînée

Qu'est-ce qu'une liste chaînée ? C'est une structure composée de noeuds. Un noeud, c'est soit rien, soit une valeur et une référence vers une autre liste.

Quand on dit "soit/soit", il faut penser "enum" (type somme plus précisément).

```
enum List<T> {
  Vide,
  Noeud(T, List<T>),
}

error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:2:1
  |
2 | enum List<T> {
  | ^^^^^^^^^^^^
3 |     Vide,
4 |     Noeud(T, List<T>)
  |              ------- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle
  |
4 |     Noeud(T, Box<List<T>>)
  |              ++++       +
```

Eh oui, quelle est la taille de l'alternative `Noeud` ? Si une liste contient une liste, on a une structure de taille infinie.

Il faut donc utiliser une référence pour éviter la récursion. Le compilateur nous parle de `Box` et de `Rc`, ce sont des types de références avancés dont on parlera plus tard. Spoiler pour ceux à qui ça parle : ce sont les équivalent des types C++ `unique_ptr` et `shared_ptr`.

```
enum List<T> {
    Vide,
    Noeud(T, &List<T>)
}

error[E0106]: missing lifetime specifier
 --> src/main.rs:4:14
  |
4 |     Noeud(T, &List<T>)
  |              ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
2 ~ enum List<'a, T> {
3 |     Vide,
4 ~     Noeud(T, &'a List<T>)
  |
```

C'est mieux, mais on a besoin d'identifier les lifetime vu qu'on a une référence dans un enum.

```
enum List<'a,T> {
    Vide,
    Noeud(T, &'a List<'a, T>)
}
```

```
impl<'a,T> List<'a, T> {
    fn new() -> Self {
        List::Vide
    }

    fn len(&self) -> usize {
        match &self {
            List::Vide            => 0,
            List::Noeud(_, suite) => 1 + suite.len(),
        }
    }
}

fn main() {
    println!("{}", List::<i32>::new().len());
}
```

Pas mal ! Maintenant, ajoutons des données à la liste.

```
#[derive(Debug, Clone)]
enum List<'a,T: Clone> {
    Vide,
    Noeud(T, &'a List<'a, T>)
}

impl<'a,T: Clone> List<'a, T> {
    fn new() -> Self {
        List::Vide
    }

    fn len(&self) -> usize {
        match &self {
            List::Vide            => 0,
            List::Noeud(_, suite) => 1 + suite.len(),
        }
    }

    fn tete(&self) -> Option<T> {
        match &self {
            List::Vide          => None,
            List::Noeud(val, _) => Some(val.clone())
        }
    }
}

fn main() {
    let l: List<i32> = List::new();
    println!("{}", l.len());
    println!("{:?}", l.tete());
    let l2 = List::Noeud(42, &List::Vide);
    println!("{}", l2.len());
    println!("{:?}", l2.tete());
    let l3 = List::Noeud(0, &l2);
    println!("{}", l3.len());
    println!("{:?}", l3.tete());
    println!("{:?}", l3);
```

```
}

0
None
1
Some(42)
2
Some(0)
Noeud(0, Noeud(42, Vide))
```

Hey, c'est pas mal ! Mais j'aimerais bien une méthode `push` qui prend un élément, une liste, et renvoie la nouvelle liste.

```
fn push(&self, val: T) -> Self {
  List::Noeud(val, self)
}

error: lifetime may not live long enough
  --> src/main.rs:27:9
   |
 7 | impl<'a,T: Clone> List<'a, T> {
   |      -- lifetime `'a` defined here
...
26 |     fn push(&self, val: T) -> Self {
   |             - let's call the lifetime of this reference `'1`
27 |         List::Noeud(val, self)
   |         ^^^^^^^^^^^^^^^^^^^^^^ method was supposed to return data with lifetime
`'a` but it is returning data with lifetime `'1`
```

Bon là on n'y comprend rien, mais le problème, c'est qu'on utilise une référence vers `self`, or `self` est une référence locale. Sa durée de vie est donc celle de la fonction.

C'est vraiment le bazar : comment faire pour que la liste *possède* une référence vers `self` plutôt que d'en emprunter une ?

Parce que c'est cela qu'on essaie de résoudre :
- une référence (pour éviter la taille infinie)
- une référence que l'on possède (pour ne plus être embêtés par les lifetimes).

Pour cela, on va utiliser un `Box<T>`. Un `Box`, c'est une référence vers une valeur, que l'on possède (et pas que l'on emprunte). Il ne peut y avoir qu'une seule `Box` qui référence une valeur donnée. La valeur en question vit aussi longtemps que le `Box`et est détruite au moment où le `Box` sort du scope.

10.2. La mise en boîte avec `Box<T>`

Quand on fait un `Box<T>`, on alloue sur le tas de quoi stocker un objet de type `T`. Le `Box` en lui-même est de taille fixe, tandis que le `T` peut être de taille quelconque. Ça permet de stocker des structures récursives comme une liste chaînée, ou de faire du polymorphisme avec des traits (coucou la programmation objet).

```
fn main() {
  let b = Box::new(42); // On alloue sur le tas la valeur 42 et on récupère un
pointeur intelligent dessus
  println!("{}", b);
  println!("{}", *b); // Implicite
  // drop(b);
}
```

Magnifique. Qu'est-ce que je peux faire avec ce machin, à part le créer avec `new()` et le déréférencer avec *?

Je peux le prêter à quelqu'un, comme n'importe quelle référence :

```
fn main() {
    let a: &Box<i32>;
    {
        let b = Box::new(42);
        println!("{}", b);
        let c: &Box<i32> = &b;
        println!("{}", c); // OK : c vit aussi longtemps que b
        a = &b;
        // b est détruit :
        // drop(b);
    }
    println!("{}", a);  // Pas OK, évidemment
}
```

Je peux aussi en transférer la propriété à quelqu'un d'autre :

```
fn main() {
  let b: Box<i32> = Box::new(42);
  let b2: Box<i32> = b; // Maintenant c'est b2 qui possède les données : b est
invalide
  println!("{}", b); // Invalide !
  // drop(b2);
}
```

Transférer la propriété est très peu coûteux.

Un `Box<T>` permet donc d'avoir la propriété d'objets qui sont sur le tas plutôt que sur la pile.

10.2.1. Retour à la liste chaînée

Tout devient plus simple pour décrire notre liste chaînée :

```
#[derive(Debug, Clone)]
enum List<T: Clone> {
    Vide,
    Noeud(T, Box<List<T>>)
}

impl<T: Clone> List<T> {
    fn new() -> Self {
        List::Vide
    }

    fn len(&self) -> usize {
        match &self {
            List::Vide            => 0,
            List::Noeud(_, suite) => 1 + suite.len(),
        }
    }

    fn tete(&self) -> Option<T> {
        match &self {
            List::Vide          => None,
            List::Noeud(val, _) => Some(val.clone())
        }
```

```
    }

    fn push(&self, val: T) -> Self {
        List::Noeud(val, Box::new(self.clone()))
    }
}

fn main() {
    let l: List<i32> = List::new();
    println!("{}", l.len());
    println!("{:?}", l.tete());
    let l2 = List::Noeud(42, Box::new(List::Vide));
    println!("{}", l2.len());
    println!("{:?}", l2.tete());
    let l3 = List::Noeud(0, Box::new(l2));
    println!("{}", l3.len());
    println!("{:?}", l3.tete());
    println!("{:?}", l3);
}
```

```
0
None
1
Some(42)
2
Some(0)
Noeud(0, Noeud(42, Vide))
```

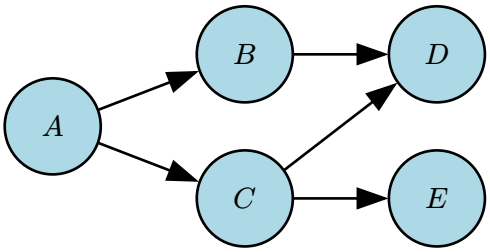
On n'est plus embêté par les lifetimes, et un `Box`, c'est relativement léger, un peu plus lourd qu'une référence brute. Ça implique quand même une allocation sur le tas, on ne va donc pas les utiliser partout.

En fait, on les utilise quand :
- on veut posséder les données (et pas seulement les emprunter),
- la taille des données n'est pas connue à la compilation,
- on veut stocker des objets très volumineux (la taille de la pile est très limitée),
- on veut faire du polymorphisme.

10.3. Complications…

Les `Box<T>` c'est cool, ça permet de faire des listes simplement chaînées et des arbres. Mais sorti de là, c'est plus compliqué…

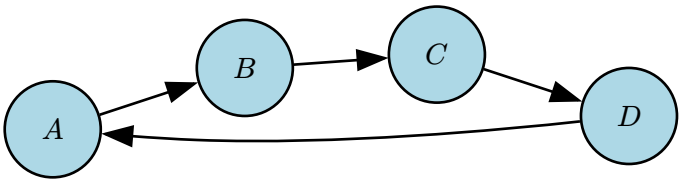
10.3.1. Les graphes acycliques dirigés



Ici, qui possède D ? B ou C ?

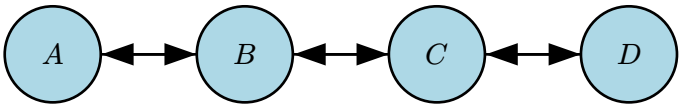
10.3.2. Les tourniquets et autres graphes cycliques

Je souhaite représenter un tourniquet. Un tourniquet, c'est une liste chaînée dans laquelle le dernier élément pointe vers le premier.



Ici, qui possède quoi ? Les élements se "possèdent" récursivement.

Ou imaginons une liste doublement chaînée : chaque élément pointe vers son suivant et son précédent.



Ici c'est pareil : B est possédé à la fois par A et par C, qui eux-mêmes possèdent B.

Peut-on utiliser des `Box` dans ces cas-là ?

Non ! Dans un `Box` il n'y a ni partage, ni récursivité. Dans ces cas-là, on doit utiliser des `Rc`.

11. Le type Rc<T>

C'est un compteur de références (Reference Counter). On en a déjà parlé : c'est un garbage collector minimaliste. Plusieurs variables peuvent référencer la même donnée. Chaque fois que cette donnée est référencée, un compteur est incrémenté. Chaque fois qu'une référence sort du scope, elle est détruite, et le compteur est décrémenté. Quand le compteur arrive à 0, la valeur référencée est détruite à son tour.

C'est très utile quand les valeurs sont référencées de façon très complexe et dynamique (exemple typique : un graphe d'objets). Dans ce cas, il est difficile de savoir qui possède quoi, qui doit détruire quoi, qui vit plus longtemps que quoi. C'est là qu'on va utiliser un `Rc`. C'est beaucoup plus lourd qu'une référence normale ou même une `Box`, mais c'est parfois incontournable.

Pour utiliser des `Rc`, on doit importer le module `std::rc`;

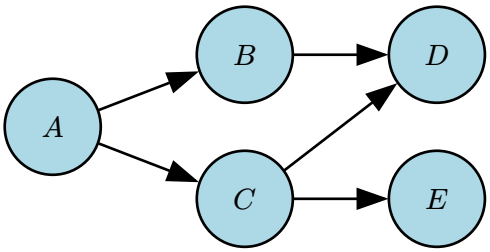
```
use std::rc::Rc;

fn main() {
    let x0;
    {
        let x1 = Rc::new(42); // Allocation sur le tas d'un entier
        println!("{}", Rc::strong_count(&x1)); // 1
        let x2 = Rc::clone(&x1);
        println!("{}", Rc::strong_count(&x1)); // 2
        {
            let x3 = Rc::clone(&x1);
            let x4 = Rc::clone(&x3);
            println!("{}", Rc::strong_count(&x1)); // 4
            // x3 et x4 sortent du scope: strong_count = 2
        }
        println!("{}", Rc::strong_count(&x1)); // 2
        x0 = Rc::clone(&x2);
        println!("{}", Rc::strong_count(&x2)); // 3
        // x1 (qui était le créateur !) et x2 sortent du scope: 1
    }
    println!("{}", Rc::strong_count(&x0)); // 1
    // x0 sort du scope, strong_count = 0, la zone sur le tas est libérée.
}
```

On a rarement besoin d'utiliser `strong_count`, ici c'est pour voir comment cela fonctionne.

11.0.1. Retour au DAG

Reprenons le graphe suivant :



Nous pouvons le représenter avec des `Rc` : chaque noeud possède un label et une liste (éventuellement vide) de noeuds fils. Un même noeud fils peut être référencé par plusieurs parents, donc, on ne peut pas utiliser de `Box`.

```rust
use std::rc::Rc;

struct Node {
    name: &'static str,
    children: Vec<Rc<Node>>,
}

fn main() {
    let d = Rc::new(Node {
        name: "D",
        children: vec![],
    });
    let e = Rc::new(Node {
        name: "E",
        children: vec![],
    });
    let b = Rc::new(Node {
        name: "B",
        children: vec![d.clone()],
    });
    let c = Rc::new(Node {
        name: "C",
        children: vec![d.clone(), e.clone()],
    });
    let a = Rc::new(Node {
        name: "A",
        children: vec![b.clone(), c.clone()],
    });
    for node in [&a, &b, &c, &d, &e] {
        println!("{}: {}", node.name, Rc::strong_count(node));
    }
}
```

```
A: 1
B: 2
C: 2
D: 3
E: 2
```

À la sortie de la fonction, tous les compteurs sont décrémentés étant donné que les variables sortent du scope. Comme A passe à 0, alors il est détruit, cd qui fait que B et C passent à 0, etc.

Et pour faire un cycle ? C'est un peu plus compliqué. Car les cycles posent problème aux compteurs de référence.

12. Lambdas, programmation fonctionnelle et itérateurs

L'outil de base en programmation fonctionnelle, ce sont les fonctions anonymes, c'est-à-dire les litéraux de type fonction (`fn`).

```
fn main() {
  let f = |n: i32| n * 2; // f double son paramètre n
  let a = 1;
  let b = f(a);
  let c = f(b);
  let mut d = f(c);
  for _ in 0..10 {
    d = f(d);
  }
  println!("{}", d);
}
```

```
8192
```

Ici, `f` est de type `fn (i32) -> i32`. On peut très bien imaginer un tableau qui contient des fonctions de ce type.

```
fn main() {
    let id = |n: i32| n;
    let double = |n: i32| n * 2;
    let incr = |n: i32| n + 1;
    let abs = |n: i32| if n < 0 { -n } else { n };

    let tab: [fn(i32) -> i32; 5] = [
        id,
        double,
        incr,
        abs,
        |n: i32| n / 2
    ];

    let mut a = 42;
    for f in tab {
        a = f(a);
        println!("{}", a);
    }
}
```

On parle de **lambdas** ou de **closures**. On n'est pas limité à la programmation fonctionnelle pure, on peut avoir une lambda qui modifie ses paramètres, si ces derniers sont `&mut` évidemment.

```
let f = |x: &mut i32, y: i32| x += y;
let mut a = 0;
for i in i..10 {
  f(&mut a, i); // On voit bien dans l'appel que f peut modifier a
}
println!("{}", a);
```

Les lambdas capturent leur environnement (c'est-à-dire les variables qui l'entourent), ce qui est pratique mais a de grosses conséquences en matière de typage.

```
let mut a = 0;
let mut incremente_a = |x: i32| a += x;
for i in 1..10 {
```

```
  incremente_a(i);
}
```

On peut avoir des lambdas qui n'ont pas de paramètre :

```
let mut a = 0;
let mut incremente_a = || a += a;
for i in 1..10 {
  incremente_a();
}
```

Beaucoup de choses sont faisables avec des lambdas, et le langage peut devenir très complexe quand on mêle lambdas, mutabilité, lifetime, capture de l'environnement et autres. On n'entrera pas trop dans les détails, mais on va voir à quoi cela sert concrètement.

12.1. Les itérateurs

Un itérateur, c'est quelque chose qui permet d'itérer sur des valeurs : l'itérateur fournit une nouvelle valeur à la demande, jusqu'à ce qu'il soit vide.

L'itérateur le plus simple, c'est "toutes les valeurs entières entre a et b", que l'on utilise tout le temps :

```
let tab = [0, 1, 2, 4, 8];
let it = 0..tab.len(); // Ça c'est un itérateur
for i in it {
  println!("{}", tab[i]);
}
```

On peut aussi récupérer un itérateur sur les collections (tableaux, vecteurs, mais aussi ensembles, hashmaps, etc.) en appelant la méthode `.iter()` :

```
let tab = [0, 1, 2, 4, 8];
for val in tab.iter() { // tab.iter() est un itérateur
  println!("{}", val);
}
```

Souvent, dans les boucles `for`, on peut omettre l'appel à `.iter()`, qui devient implicite :

```
let tab = [0, 1, 2, 4, 8];
for val in tab { // appel implicite de tab.iter()
  println!("{}", val);
}
```

Un itérateur, c'est simplement quelque chose qui implémente le trait `Iterator`. Le trait `Iterator` a une méthode intéressante :

```
fn next(&mut self) -> Option<T>;
```

C'est la fonction qui renvoie l'élément suivant, ou `None` lorsque la fin est atteinte. On peut créer nos propres itérateurs si on en a envie, sur nos propres types.

Quand on appelle `.next()`, on dit que l'on *consomme* l'itérateur.

Cela veut dire qu'un appel en apparence trivial comme :

```
let tab = [0, 1, 2, 4, 8];
for val in tab {
  println!("{}", val);
}
```

Se traduit en réalité par quelque chose comme :

```rust
let tab = [0, 1, 2, 4, 8];
let mut it = tab.iter();
while true { // On aurait aussi pu écrire "loop {"
  let next = it.next();
  match next {
    None => {
      break;
    },
    Some(val) => {
      println!("{}", val);
    }
  }
}
```

À part l'utiliser dans un `for`, que peut-on faire avec un itérateur ? Plein de choses, en réalité. Il y a énormément de méthodes associées. On peut par exemple remplacer notre boucle `for` par un appel à `for_each()` :

```rust
let tab = [0, 1, 2, 4, 8];
tab.iter().for_each(|val| println!("{}", val));
```

La méthode `for_each` prend une lambda en paramètre. Elle appelle la méthode `next()` et, si elle obtient `Som(val)`, appelle la lambda sur `val`. Elle recommence, jusqu'à ce qu'elle tombe sur `None`.

À quoi ça sert ? C'est plus compact qu'une boucle `for`, si l'on a un traitement simple à faire sur chaque élément, c'est souvent plus lisible.

On peut faire d'autres choses. Par exemple, calculer la somme des valeurs via la méthode `sum` :

```rust
let tab = [0, 1, 2, 4, 8];
let s: i32 = tab.iter().sum();
println!("{}", s);
```

Ici, j'ai dû indiquer le type de `s` car `sum()` renvoie quelque chose qui implémente le trait `Sum<T>`. Avec les itérateurs, on arrive à des niveaux de généricité (et de complexité d'écriture) tellement élevés qu'il faut souvent expliciter ce que l'on attend.

Certaines méthodes produisent d'autres itérateurs. C'est le cas des méthodes bien aimées des adeptes de la programmation fonctionnelle : `map` et `filter`.

La méthode `map` permet d'appliquer une fonction données à tous les éléments d'un itérateur et de renvoyer un nouvel itérateur avec les nouvelles valeurs :

```rust
let tab = [0, 1, 2, 4, 8];
let it = tab.iter().map(|x| x + 1);
// it est un nouvel itérateur. Il est paresseux.
// Ses valeurs ne seront calculées que si l'on appelle next()
for val in it {
  println!("{}", val);
}
```

Oui, les itérateurs sont **paresseux**. Cela veut dire que les valeurs associées ne sont calculées que lorsque c'est nécessaire. Cela permet notamment d'avoir des itérateurs infinis, l'équivalent des générateurs de python.

Évidemment, on peut aussi écrire :

```rust
let tab = [0, 1, 2, 4, 8];
for val in tab.iter().map(|x| x + 1) {
```

```
  println!("{}", val);
}
```

Ou encore, puisqu'un `for` peut être réécrit via `for_each()` :

```
let tab = [0, 1, 2, 4, 8];
tab.iter().map(|x| x + 1).for_each(|val| println!("{}", val));
```

On a chaîné les appels de méthodes. Ce n'est pas très lisible, en général on préfère faire comme suit :

```
let tab = [0, 1, 2, 4, 8];
tab.iter()
  .map(|x| x + 1)
  .for_each(|val| println!("{}", val));
```

On avait aussi vu la méthode `rev()` plus haut. Elle retourne un nouvel itérateur avec les données dans l'ordre inverse :

```
let tab = [0, 1, 2, 4, 8];
tab.iter()
  .map(|x| x + 1)
  .rev()
  .for_each(|val| println!("{}", val));
```

Souvent, on veut convertir l'itérateur en collection (en vec par exemple) :

```
let tab = [0, 1, 2, 4, 8];
let tab2: Vec<i32> = tab.iter()
  .map(|x| x + 1)
  .rev()
  .collect(); // C'est cet appel qui transforme l'itérateur en Vec<i32>
```

Ajoutons maintenant un filtre : je ne veux que des valeurs plus grandes que 3.

```
let tab = [0, 1, 2, 4, 8];
let tab2: Vec<i32> = tab.iter()
  .map(|x| x + 1)
  .filter(|x| *x > 3)
  .rev()
  .collect(); // C'est cet appel qui transforme l'itérateur en Vec<i32>
```

Attends, attends, c'est quoi cette déréférence là dans l'appel à `filter` ?

C'est (encore et toujours) une question de propriété. La fonction `map` transforme des données, elle va donc les consommer pour en produire une nouvelle. On attend de la fonction `filter`, au contraire, qu'elle observe les données, sans les modifier. On ajoute donc un niveau de référence supplémentaire.

Les règles à ce niveau sont assez complexes, d'autant que parfois le déréférencement est implicite.

Autre exemple avec des chaînes :

```
let tab = ["toto", "bob", "alice"];
let tab2: Vec<String> = tab.iter()
    .filter(|s| s.len() > 3)
    .map(|s| s.to_uppercase())
    .collect();
println!("{:?}", tab2);
```

On peut faire des choses complexes. J'ai des joueurs, je veux afficher le nom en majuscules de ceux qui sont encore en vie :

```rust
#[derive(Clone,Debug,Default)]
struct Joueur {
    nom: String,
    score: i32,
    vies: i32,
}

impl Joueur {
    fn new(nom: &str) -> Self {
        Joueur {
            nom: nom.to_string(),
            score: 0,
            vies: 3,
        }
    }

    fn incr_score(&mut self, val: i32) -> &Self {
        self.score += val;
        self
    }

    fn perdre_vie(&mut self) -> &Self {
        self.vies -= 1;
        self
    }

    fn est_vivant(&self) -> bool {
        self.vies > 0
    }
}

fn main() {
    let mut toto = Joueur::new("toto");
    let mut titi = Joueur::new("titi");
    let mut tutu = Joueur::new("tutu");
    while titi.est_vivant() {
        titi.perdre_vie();
    }
    let tab = vec![toto, titi, tutu];
    tab.iter()
        .filter(|j| j.est_vivant())
        .map(|j| j.nom.to_uppercase())
        .for_each(|nom| println!("{} est encore vivant !", nom));
}
```

Je veux maintenant donner 1000 points de plus à ceux qui sont encore vivants, puis leur retirer une vie. Je veux juste faire ça. Notez qu'il y a un changement par rapport à avant : cette fois, l'itérateur **modifie** les valeurs sur lesquelles on itère.

Désormais, `tab` doit être mutable, mais aussi on doit obtenir un itérateur qui mute ses valeurs (un itérateur… mutant ?) On utilise pour cela la méthode `iter_mut()` :

```rust
fn main() {
    let mut toto = Joueur::new("toto");
    let mut titi = Joueur::new("titi");
    let mut tutu = Joueur::new("tutu");
    while titi.est_vivant() {
```

```
        titi.perdre_vie();
    }
    toto.perdre_vie();
    toto.perdre_vie();
    let mut tab = vec![toto, titi, tutu];
    tab.iter_mut()
        .filter(|j| j.est_vivant())
        .map(|j| j.incr_score(1000))
        .for_each(|j| println!("{}", j.nom));
}
```

Ici, `tab` a été modifié, et les variables `toto`, `titi` et `tutu` ne peuvent désormais évidemment plus être utilisées : la propriété de leurs données a été transférée à `tab`.