Rust

4SE02

Guillaume Duc Samuel Tardieu

Mars 2025

Définitions, concepts et historique

Introduction

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

A language empowering everyone to build reliable and efficient software

Promesses :

- Performance
  - Rapide et efficace (pas de ramasse miettes)
  - Peut faire tourner des services dont les performances sont critiques, tourner sur des systèmes embarqués et s'intégrer à d'autres langages
- Robustesse
  - Système de type riche et modèle de propriété (*ownership model*)
  - Garantissent la sûreté mémoire et thread (*memory-safety* et *thread-safety*)
  - Éliminent de nombreuses classes de bugs à la compilation
- Productivité
  - Documentation
  - Messages d'erreur informatifs du compilateur
  - Outils puissants

Bref historique

- Début en 2006 comme projet personnel de Graydon Hoare (employé de Mozilla)
- Sponsorisé par Mozilla depuis 2009
- Première version stable (1.0) le 15 mai 2015
- Très largement utilisé aujourd'hui, aussi bien par des acteurs établis que par de jeunes entreprises.

Création d'un nouveau projet

```
$ cargo new helloworld
     Created binary (application) `helloworld` package
$ cd helloworld
$ cargo run
   Compiling helloworld v0.1.0 (/tmp/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.86s
     Running `target/debug/helloworld`
Hello, world!
```

Pour travailler efficacement

Vous pouvez utiliser n'importe quel éditeur, mais nous conseillons Visual Studio Code avec l'extension `rust-analyzer`, qui permet de naviguer dans le code source.

Commentaires

- Commentaires internes au code source avec `//`
- Documentation de l'élément suivant avec `///` au format Markdown
- Documentation de l'élément englobant (module) avec `//!` en Markdown
- `rustdoc` ou `cargo doc` génèrent une documentation HTML à partir des sources

```rust
//! This module contains various algorithms
//! that can be used for pathfinding.

/// Compute shortest path using Dijkstra algorithm. Arguments:
/// - `graph`: the input graph
/// …
fn dijkstra(graph: &Graph, …) -> … {
    // Initialize the internal structures
    let mut seen HashSet<Node> = …; // Nodes already seen
    …
}
```

Types de base : entiers

Les types sont signés en complément à deux (`i`) ou non-signés (`u`) :

- 8 bits : `i8`, `u8`
- 16 bits : `i16`, `u16`
- 32 bits : `i32`, `u32`
- 64 bits : `i64`, `u64`
- 128 bits : `i128`, `u128`
- Taille correspondant à la taille d'un pointeur : `isize`, `usize`

Représentation littérale des entiers

- Par défaut : base 10
- Préfixe : `0x` (base 16), `0o` (base 8), `0b` (base 2)
- `_` peut servir de séparateur visuel (ignoré par le compilateur), exemples : `12_345`, `0x0123_4567`
- Suffixe : type explicite (le type par défaut dépend du contexte, si aucune contrainte : `i32`), exemples : `123u16`, `-42i64`

Types de base : flottants IEEE-754

- Simple précision : `f32`
- Double précision : `f64` (type par défaut des représentations littérales, exemple : `3.0`)

Autres types de base

- Booléens : `bool`, valeurs : `true` ou `false`
- Caractère Unicode : `char` (4 octets), exemples : `'A'`, `'é'`
- Type `unit ()`, qui n'a qu'une seule valeur `()`
  - Type retourné par une fonction que ne renvoie rien
  - ou par une expression terminée par un `;`
- Les types composés (tuples, tableaux et structures seront abordés ultérieurement)

Déclaration de variable

Définition d'une variable *locale* :

```
let x: i32 = 42;
```

Par défaut, les variables sont immuables (*immutable*), le code suivant ne compile donc pas :

```
let x: i32 = 42;
x = x + 1; // cannot assign twice to immutable variable
```

Pour rendre une variable mutable, il faut utiliser le mot clé `mut` (*mutable*) :

```
let mut x: i32 = 42;
x = x + 1; // x is now 43, could also be written as x += 1
```

Le type n'est en général pas nécessaire, le compilateur fait un très bon travail d'inférer le bon type :

```
let x = 42;
```

Variables, suite

Il n'est pas nécessaire de donner une valeur initiale tout de suite à la définition, y compris pour une variable immuable. Par contre, le compilateur détectera l'utilisation d'une variable non initialisée.

Une redéfinition d'une variable déjà présente dans le contexte masque la précédente. Exemple :

```
1  let x = 42;
2
3  let x = 12;
```

Fonctions

Syntaxe (simplifiée, sera complétée par la suite) :

```
fn nom_fonction([nom_arg: type_arg]*) [-> type_retour]? {
   // ...
}
```

La valeur de retour est la dernière expression évaluée mais peut être forcée (retour prématuré) avec `return` :

```
// Les fonctions add1 et add2 sont équivalentes
fn add1(a: i32, b: i32) -> i32 {
    a + b
}

fn add2(a: i32, b: i32) -> i32 {
    return a + b;
}
```

Types composés : tuples

```
let t: (i32, f64, u8) = (42, 3.5, 1);
```

Accès aux différents membres (par déstructuration) :

```
let (x, y, z) = t; // x = 42, y = 3.5, z = 1
```

Accès aux membres individuels :

```
let x = t.0; // 42
let y = t.1; // 3.5
let z = t.2; // 1
```

Les membres individuels sont modifiables (avec la syntaxe ci-dessus), si le tuple est déclaré `mut`.

```
let mut t: (i32, f64, u8) = (42, 3.5, 1);
t.0 = 12;
```

Types composés : tableaux

Déclaration :

```
let a: [u32; 4] = [1, 2, 3, 4];
```

Accès à un élément :

```
let x = a[0];
```

Lors de l'accès à un élément, Rust vérifie, à la compilation si c'est possible, sinon à l'exécution, que l'index est compris dans les bornes du tableau. Dans le cas contraire, Rust panique.

La taille des tableaux est fixe (ne change pas après la déclaration). Si vous voulez un tableau de taille dynamique, regardez les collections (`Vec` par exemple).

Initialisation concise :

```
let a: [u32; 4] = [0; 4]; // = [0, 0, 0, 0]
```

Types composés : structures

Déclaration d'une structure :

```
struct Coord {
    x: i32,
    y: i32,
    z: i32,
}
```

Instanciation :

```
let c1: Coord = Coord { x: 3, y: 4, z: 5 };
```

Accès aux membres :

```
let x = c1.x;
```

Types composés : structures (bloc impl)

On peut regrouper des fonctions concernant cette structure au sein d'un bloc `impl` :

```
1  impl Coord {
2      fn new(x: i32, y: i32, z: i32) -> Coord {
3          Coord { x, y, z }
4      }
5
6      fn manhattan_distance_from_orig(&self) -> i32 {
7          self.x.abs() + self.y.abs() + self.z.abs()
8      }
9  }
10
11 fn main() {
12     let c = Coord::new(1, 3, 5);
13     let d = c.manhattan_distance_from_orig();
14 }
```

On reviendra plus loin sur ces blocs `impl`.

Types composés : structures aux champs non nommées

On peut choisir de ne pas nommer les champs d'une structure. Elle ressemble alors à un tuple et ses éléments sont accessibles par `.0`, `.1`, etc.

```
struct PairOfIntegers(i32, i32);

fn main() {
    let pair = PairOfIntegers(17, 42);
    println!("Sum of fields: {}", pair.0 + pair.1);
}
```

Il est aussi possible de construire une structure sans champ, l'intérêt étant par exemple de pouvoir y associer un bloc `impl` :

```
struct A;
struct B {}
struct C();

…
let (a, b, c) = (A, B {}, C());
```

Note sur les types composés

Type produit

Chaque structure, tuple ou tableau est un “type produit”, car le nombre de valeurs que chaque type peut prendre est le produit du nombre de valeurs de ses composants.
Par exemple, un tuple `(i8, char)` peut prendre $2^8 \times 2^{32} = 2^{40}$ valeurs distinctes.

Types composés : énumérations

```rust
enum Vehicule {
    Plane,
    Boat,
    Land,
}
let car = Vehicule::Land;
```

Les variantes peuvent contenir des champs :

```rust
enum Vehicule {
    Plane { engines: u8, passengers: u32 },
    Boat,
    Land(u8),
}
let car = Vehicule::Land(4);
let a380 = Vehicule::Plane { engines: 4, passengers: 853 };
```

Notes sur les énumérations

Type somme

Une énumération en Rust est également appelée “type somme” car elle a autant de valeurs possibles que la somme des valeurs possibles de ses variantes, sans possibilité de confusion entre les variantes.
Par exemple l'énumération

```
1 enum E {
2   Something(i8),
3   SomethingElse(i8),
4   Nothing
5 }
```

peut prendre $256 + 256 + 1 = 513$ valeurs distinctes.

Raccourci d'utilisation

Les variantes d'une énumération sont préfixées par le nom de l'énumération (`Vehicule::Land(4)`). En utilisant `use Vehicule::*;` on rend visible directement les variantes (`Land(4)`).

Généricité

Pour les structures :

```
struct Coord<T> {
    x: T, y: T, z: T,
}

let c_i = Coord { x: 1, y: 2, z: 3 };
let c_f: Coord<f64> = Coord { x: 1.0, y: 2.0, z: 3.0 };
```

Pour les fonctions :

```
fn new_coord<T>(x: T, y: T, z: T) -> Coord<T> {
    Coord { x, y, z } // Coord::<T> { x, y, z }
}

let c = new_coord(1, 2, 3); // new_coord::<i32>(1, 2, 3);
```

Généricité (suite)

Pour le bloc implémentation :

```
1  struct Coord<T> {
2      x: T, y: T, z: T,
3  }
4
5  impl<T> Coord<T> {
6      fn new(x: T, y: T, z: T) -> Self {   // Type Self is Coord<T> in impl block
7          Coord { x, y, z }
8      }
9  }
```

Affichage : print !() et println !()

Les macros seront vues plus loin, mais celles-ci seront utiles :

- `print!()` et `println!()` (avec retour à la ligne) prennent une chaîne de formatage et des arguments
- la substitution de base est `{}`

```
1 print!("Hello, world!\n");
2 println!("Hello, world!");
3 println!("a = {a}, b = {b}, a+b = {}", a+b);
```

Bien entendu, ces macros sont bien plus puissantes que les exemples simples présentés ici.

Structures de contrôle : if

```rust
if answer == 42 {            // No need for () around condition
    println!("Correct");
} else {                     // {} are mandatory around then and else parts
    println!("Wrong");
}
```

Chaînage `else if`

```rust
if number < 0 {
    println!("Negative");
} else if number == 0 {
    println!("Zero");
} else {
    println!("Positive");
}
```

Structures de contrôle : if

`if` est une expression qui renvoie une valeur

```
let i = if condition { 2 } else { 3 };
```

Les deux branches du `if` doivent renvoyer le même type. Le code suivant ne compile donc pas :

```
let i = if condition { 2 } else { 3.0 };
// error[E0308]: `if` and `else` have incompatible types
```

Structures de contrôle : panic !()

`panic!([message…])` est une des macros (identifiables par le `!` final, vues plus loin) prédéfinies qui arrête brutalement le thread courant en cas d'erreur irrécupérable :

```
1  fn integer_slope(x0: i32, y0: i32, x1: i32, y1: i32) -> i32 {
2      if x0 == x1 {
3          panic!("unable to compute an infinite slope")
4      } else {
5          (y1-y0)/(x1-x0)
6      }
7  }
```

D'autres structures vues plus loin (`Option`, `Result`) permettent de gérer plus délicatement ces situations.

Quel est le type de retour de panic !() ?

Pourquoi ce code compile-t-il ?

```
1  fn integer_slope(x0: i32, y0: i32, x1: i32, y1: i32) -> i32 {
2      if x0 == x1 {
3          panic!("unable to compute an infinite slope")
4      } else {
5          (y1-y0)/(x1-x0)
6      }
7  }
```

Parce que le type de retour de `panic!()` est `!` (prononcé *never*). Une valeur de ce type est un calcul qui ne retourne jamais de résultat (erreur fatale ou boucle infinie). Il est donc compatible avec tout.

Une écriture plus compacte grâce à panic !()

Étant donné que `panic!()` retourne de manière prématurée (et brutale), on peut s'en servir comme d'une guarde :

```
1  fn integer_slope(x0: i32, y0: i32, x1: i32, y1: i32) -> i32 {
2      if x0 == x1 {
3          panic!("unable to compute an infinite slope");
4      }
5      (y1-y0)/(x1-x0)
6  }
```

Autre illustration de !

Ce code est correctement typé :

```rust
fn never_returns() -> ! {
    loop {}  // Infinite loop
}

let x: i32 = never_returns();
let y: Option<f64> = never_returns();
let z: [u8; 10] = never_returns();
```

En théorie des types, ! est un appelé un type *bottom*. Il se trouve en dessous de tous les types et peut être converti vers tous les types puisqu'il ne termine jamais normalement (il n'y a donc jamais rien à convertir).

Structures de contrôle : loop

`loop` est une boucle infinie (de type `!`) :

```
fn main() {
    loop { println!("SOS"); }
}
```

`break` permet d'interrompre la boucle la plus imbriquée (et de renvoyer une valeur) :

```
let res = loop {
    i = i + 1;
    if i == 10 {
        break i;
    }
}
```

Le type de la boucle est alors celui de la valeur renvoyée ou `()` sinon.

Structures de contrôle : loop

`continue` saute au début de la prochaine itération de la boucle la plus imbriquée :

```
loop {
    i = i + 1;
    if i == 10 {
        continue;
    }
}
```

Il est possible pour `break` et `continue` de spécifier de quelle boucle sortir ou continuer :

```
'out: loop {
    'in: loop {
        if i == 10 {
            continue 'out;
        }
    }
}
```

Type des expression break et continue

Le type des expressions `break` et `continue` est `!` (never). En effet, étant donné qu'elles provoquent un saut à un autre endroit du programme, elles ne retournent jamais rien.

Le code suivant est correctement typé :

```
1  let mut i = 0;
2  loop {
3      if i == 10 {
4          let a: f64 = break;
5      }
6      i += 1;
7  }
```

`a` ne recevra jamais d'affectation puisque l'évaluation de `break` sort de la boucle courante. `break` étant de type `!`, il peut donc être (jamais) affecté à `a` quelque soit le type de cette variable.

Structures de contrôle : while

Boucle tant que la condition est vraie :

```
1 while condition {
2     // ...
3 }
```

Le type de cette expression est `()`.

Structures de contrôle : for

`for` permet d'itérer sur les éléments d'une collection

```
let a = [1, 2, 3, 4, 5];
let mut sum = 0;
for i in a.iter() {
    sum = sum + i;
}
```

La collection passée après le mot clé `in` doit implémenter le trait `IntoIterator` (les traits seront traités ultérieurement). Le type de l'expression `for` est `()`.

```
let mut sum = 0;
for i in 0..10 { // 0..10 is [0,10[  - Use 0..=10 for [0,10]
    sum = sum + i;
}
```

Structures de contrôle : match

`match` permet d'unifier son argument lors d'un ensemble de clauses parcourues dans l'ordre, avec possibilités de guardes et de motif universel (`_`) :

```
1  enum Sign { Negative, Zero, Positive }
2
3  fn sign_of_x_plus_two(x: i32) -> Sign {
4      match x + 2 {
5          0          => Sign::Zero,
6          n if n > 0 => Sign::Positive,  // n is linked to x+2
7          _          => Sign::Negative,
8      }
9  }
```

Le type de l'expression `match` est celui retourné par ses branches (hormis les branches retournant le type `!` (never)).

Scopes et durée de vie

Tout objet en Rust a une période de vie déterminée lexicalement :

- de l'endroit de sa définition jusqu'à sa dernière utilisation directe ou indirecte (plus là dessus plus tard), au plus tard à la fin du scope où il est déclaré, ou
- infinie s'il s'agit d'un objet global (chaîne de caractère littérale par exemple)

```
1  {
2      let x = 3;       // x life starts here
3      {
4          let y = x + x;  // y life starts here
5          let z = 42;     // z life starts here
6          …
7      }  // y and z lives end no later than here
8  }  // x live ends no later than here
```

Références

Il est possible d'emprunter (*borrow*) une référence sur un objet existant à son propriétaire (*owner*) :

```
let mut x: i32 = 42;
let y: &i32 = &x;           // Read-only reference to x
let z: &mut i32 = &mut x;   // Read-write reference to x
```

La référence a une durée de vie qui ne peut excéder celle de l'objet référencé.

Une référence sur un objet non mutable ne peut être qu'en lecture seule :

```
let x: i32 = 42;
let y: &i32 = &x;           // Read-only reference to x
let z: &mut i32 = &mut x;   // error[E0596]: cannot borrow `x` as mutable,
                            // as it is not declared as mutable
```

Rust dispose d'un vérificateur statique (le *borrow checker*) qui s'assure que les références suivent des règles précises.

Prêt et emprunt

Les références empruntent temporairement l'objet référencé à son propriétaire et suivent des règles :

- `&mut x` est une référence sur `x` en lecture/écriture (*mutably borrowed*) ; tant que vous possédez cette référence, personne d'autre ne peut modifier la valeur de `x`
- `&x` est une référence sur `x` en lecture seule (*borrowed*) de type `&i32` ; tant que vous possédez une telle référence, vous savez que la valeur de `x` ne peut pas être modifiée

Ceci est vrai car le compilateur garantit que :

- Plusieurs `&x` peuvent exister en même temps, mais pas en même temps qu'un `&mut x` qui lui n'existe qu'en au plus un exemplaire à la fois
- `x` ne peut être modifié directement (ni devenir invalide) tant qu'un `&x` ou `&mut x` existe
- `&x` et `&mut x` ne peuvent pas être transmis à un autre thread, sauf cas prévu explicitement

Références et lifetimes

Quand on écrit `&i32`, on omet une partie du type, qui est en fait `&'a i32` où `'a` est une durée de vie (*lifetime*).

En fait, une référence est toujours assortie d'une durée de vie (*lifetime*) :

- `&'a i32` représente une référence sur un `i32` d'une durée de vie `'a`
- L'objet pointé par une référence `&'a i32` est nécessairement valide pendant toute la durée de vie `'a`
- Si une durée de vie `'b` est inclue dans une durée de vie `'a`, on peut stocker un `&'a i32` dans un objet `&'b i32`
- `'static` désigne une durée de vie infinie, pour des objets vivants pendant toute la durée du programme
- Une lifetime n'est pas forcément nommable (référence sur un objet par exemple), et sera donc implicite.

Déréférencement

L'opérateur * permet de désigner (ou "déréférencer") le contenu d'une référence :

```
fn inc(v: &mut i32) { *v += 1; }

let mut x = 41;
inc(&mut x);
println!("x = {x}");  // x = 42
```

Ici, une lifetime quelconque est inférée et s'adaptera au contexte, équivalent à :

```
fn inc<'a>(v: &'a mut i32) { *v += 1 }
```

Rust implémente également l'auto-déréférence. Si `s` est une référence sur une structure qui possède un champ `a`, au lieu d'écrire `(*s).a`, on pourra simplement écrire `s.a`.

Lifetimes : exemple en C

```c
#include <stdio.h>

typedef struct { int v; } s;

// Return pointer on inner integer
int *get_v_ptr(s *arg) {
  return &arg->v;
}

int main() {
  int *ptr;
  {
    s x = { 42 };
    printf("x.v = %d\n", x.v);
    ptr = get_v_ptr(&x);
  }
  printf("*ptr = %d\n", *ptr);   // This is incorrect (x no longer exists) but allowed
}
```

Lifetimes : exemple en Rust

```rust
struct S { v: i32 }

fn get_v_ptr(arg: &S) -> &i32 {  // fn get_v_ptr<'a>(arg: &'a S) -> &'a i32
    &arg.v   // Note the auto-dereference, this is equivalent to &(*arg).v
}

fn main() {
    let ptr: &i32;
    {
        let x = S { v: 42 };
        println!("x.v = {}", x.v);
        ptr = get_v_ptr(&x);
    }
    println!("*ptr = {}", *ptr);
}
```

ne compile pas :

```
error[E0597]: `x` does not live long enough --> t.rs:12:21
```

Lifetimes : exemple en Rust, aide du compilateur sur l'erreur

Le compilateur est même plus explicite sur l'explication de l'erreur :

```
error[E0597]: `x` does not live long enough
  --> t.rs:12:21
   |
12 |         ptr = get_v_ptr(&x);
   |                         ^^ borrowed value does not live long enough
13 |     }
   |     - `x` dropped here while still borrowed
14 |     println!("*ptr = {}", *ptr);
   |                           ---- borrow later used here

For more information about this error, try `rustc --explain E0597`.
```

Généricité et enumérations

On peut combiner généricité et types somme pour représenter l'existence facultative d'une valeur ou encore le résultat d'un calcul qui peut échouer :

```rust
/// Optionally contain a value of type `T`
enum Option<T> {
    Some(T),
    None,
}
use Option::{Some, None};   // Direct visibility over the variants

/// Contain a success value of type `S`, or a failure of type `F`
enum Result<S, F> {
    Ok(S),
    Err(F),
}
use Result::{Ok, Err};      // Direct visibility over the variants
```

Option : exemple d'utilisation

```rust
struct Person {
    first_name: String,
    last_name: String,
    birth_year: Option<u32>,
}

fn age_of(person: &Person, current_year: u32) -> Option<u32> {
    match person.birth_year {
        Some(y) => Some(current_year - y),
        None    => None
    }
}
```

On verra plus loin comment écrire plutôt :

```rust
fn age_of(person: &Person, current_year: u32) -> Option<u32> {
    person.birth_year.map(|y| current_year - y)
}
```

Implémentation de méthodes

Plutôt que de définir les fonctions comme `age_of` séparément du type `Person`, on peut les y attacher avec un bloc `impl` en utilisant `self` :

```
impl Person {
    fn age_of(&self, current_year: u32) -> Option<u32> {
        match self.birth_year {   // Here self is of type &Self, or &Person
            Some(y) => Some(current_year - y),
            None    => None
        }
    }
}
```

Le premier argument d'une fonction dans un bloc `impl` est traité comme une méthode s'il se réfère à

- `&self` : une référence en lecture seule sur un objet du type concerné
- `&mut self` : une référence en lecture-écriture sur un objet du type concerné
- `self` : l'objet lui-même dont la propriété est transférée à la fonction

Appel de méthodes

L'appel de méthode utilise une notation suffixée :

```
impl Person {
    fn age_of(&self, current_year: u32) -> Option<u32> {
        match self.birth_year {
            Some(y) => Some(current_year - y),
            None    => None
        }
    }
}

let p = Person { first_name: …, last_name: …, birth_year: Some(1977) };
match p.age_of(2020) {
    Some(age) => println!("This person is {age} year old"),
    None      => println!("I don't know the age"),
}
```

Retour à Option : extraction de la valeur

Le contenu d'une `Option` peut être extraite avec une fonction que nous appelerons `unwrap()` :

```
impl<T> Option<T> {
    fn unwrap(self) -> T {
        match self {
            Some(v) => v,
            None    => panic!("cannot unwrap from None"),
        }
    }
}
```

On notera que :

- `panic!()` est l'action appropriée à avoir quand l'option ne contient aucune valeur
- La méthode `unwrap` prend `self` (et non pas `&self`) et consomme l'objet qu'on lui passe : on échange l'objet `Option` contre son contenu ; on récupère l'original du contenu, pas une copie

Fonctions d'ordre supérieur

Une fonction est dite d'ordre supérieur lorsqu'elle peut prendre en paramètre d'autres fonctions. En Rust, un type fonction est défini comme :

```
fn (argtyp1, argtyp2, …) -> rettype
```

On peut utiliser ces types pour implémenter `map` qui applique une fonction sur la valeur contenue dans une `Option` quand elle existe et place le résultat dans une `Option` :

```
impl<T> Option<T> {
    /// Apply a function to an optional value if present and return a new Option
    /// containing the result. When applied to None, just return None.
    fn map<U>(self, f: fn (T) -> U) -> Option<U> {
        match self {
            Some(v) => Some(f(v)),
            None    => None
        }
    }
}
```

Fonctions anonymes (ou lambda-expressions)

Une fonction anonyme est définie par le nom (et le type éventuel) de ses arguments et l'expression permettant d'en calculer le résultat :

```
let f = |a, b| a+b;
println!("{}", f(10, 20));   // 30
```

Cela nous permet d'implémenter notre méthode `age_of` de manière plus concise :

```
impl Person {
    fn age_of(&self, current_year: u32) -> Option<u32> {
        self.birth_year.map(|y| current_year - y)
    }
}
```

Fonctions anonymes et fermeture transitive

Une fonction anonyme peut emprunter des références vers des valeurs externes (*closure*, ou fermeture transitive) :

```
1 let s = String::from("Hello");
2 let f = |n| s.chars().nth(n);              // Here, f borrows s
3 println!("{s} => {:?}", f(4));             // Hello => Some('o')
```

Les règles usuelles s'appliquent :

- La durée de vie de `f` ne peut pas s'étendre au delà de celle de `s`
- Même si `s` était mutable, la variable serait impossible à modifier tant qu'elle est prêtée à `f`

Le mot clé `move` permet de capturer les variables référencées plutôt que d'en prendre une référence. C'est inutile ici, mais cela s'écrirait :

```
1 let s = String::from("Hello");
2 let f = move |n| s.chars().nth(n);      // Here, s is transferred inside f
3 // s does not exist by itself anymore at this point and can no longer be used
```

Le type Result

Rappel :

```rust
/// Contain a success value of type `S`, or a failure of type `F`
enum Result<S, F> {
    Ok(S),
    Err(F),
}
use Result::{Ok, Err};      // Direct visibility over the variants
```

Il peut s'utiliser ainsi :

```rust
enum ErrorCode { DivisionByZeroError, OverflowError }

fn safe_divide(a: i32, b: i32) -> Result<i32, ErrorCode> {
    if b == 0 {
        return Err(ErrorCode::DivisionByZeroError);
    }
    Ok(a/b)
}
```

Result et chaînage

On peut enchaîner les calculs en s'arrêtant dès la première erreur grâce à `match` et `return` :

```
fn safe_divide_thrice(a: i32, b: i32, c: i32, d: i32) -> Result<i32, ErrorCode> {
    let r1 = match safe_divide(a, b) {
        Ok(r)  => r,
        Err(e) => return Err(e),
    };
    let r2 = match safe_divide(r1, c) {
        Ok(r)  => r,
        Err(e) => return Err(e),
    }
    safe_divide(r2, d)
}
```

La partie `match x { Ok(v) => v, Err(e) => return Err(e) }` peut être remplacée par l'opérateur `x?` de manière plus concise.

Result et chaînage avec ?

On peut enchaîner les calculs en s'arrêtant dès la première erreur grâce à ? qui retourne l'erreur s'il y en a une ou désencapsule le résultat en l'absence d'erreur :

```
fn safe_divide_thrice(a: i32, b: i32, c: i32, d: i32) -> Result<i32, ErrorCode> {
    let r1 = safe_divide(a, b)?;
    let r2 = safe_divide(r1, c)?;
    safe_divide(r2, d)
}
```

Notons qu'en Rust, on peut redéfinir une variable du nom d'une variable existante ; cette dernière disparaît. On écrira donc plutôt :

```
fn safe_divide_thrice(a: i32, b: i32, c: i32, d: i32) -> Result<i32, ErrorCode> {
    let r = safe_divide(a, b)?;
    let r = safe_divide(r, c)?;
    safe_divide(r, d)
}
```

Tests

```rust
#[test]
fn one_plus_one_equals_two() {
    assert!(1 + 1 == 2);
}
```

```
$ cargo test
running 1 test
test one_plus_one_equals_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Cargo test va exécuter toutes les fonctions ayant l'attribut `#[test]` et afficher les résultats. Si la fonction se termine, le test est considéré comme réussi, si elle panique, le test est considéré comme échoué.

Tests (exemple d'échec)

```rust
#[test]
fn one_plus_one_equals_three() {
    assert!(1 + 1 == 3);
}
```

```
$ cargo test
running 2 tests
test one_plus_one_equals_two ... ok
test one_plus_one_equals_three ... FAILED

failures:

---- one_plus_one_equals_three stdout ----
thread 'one_plus_one_equals_three' panicked at 'assertion failed: 1 + 1 == 3', src/main.rs:8:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

failures:
    one_plus_one_equals_three

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

Tests (macros)

- `assert!(cond, [message…])` panique si `cond` ne s'évalue pas à `true`
- `assert_eq!(a, b, [message…])` panique si on n'a pas `a == b` (*equal*)
- `assert_ne!(a, b, [message…])` panique si on n'a pas `a != b` (*not equal*)

Ces macros sont également utilisables en dehors des tests pour vérifier des conditions indispensables au bon fonctionnement du programme et qu'on préfère l'arrêter de manière fatale si ces conditions ne sont pas vérifiées.

On notera que des versions de développement existent (préfixées par `debug_`, par exemple `debug_assert!(cond, [message…])`) si on souhaite insérer des vérifications additionnelles uniquement en mode *debug* et les supprimer en mode *release* (lorsque l'on compile avec l'option `--release`).

Test (should_panic)

Si on veut tester qu'une fonction panique bien, on peut utiliser l'attribut `#[should_panic]`

```
1  #[test]
2  #[should_panic]
3  fn test_panic() { panic!(); }
4
5  #[test]
6  #[should_panic]
7  fn test_not_panic() { }
```

```
$ cargo test
running 2 tests
test test_not_panic ... FAILED
test test_panic ... ok

failures:

---- test_not_panic stdout ----
note: test did not panic as expected
```

Traits

Un *trait* décrit un comportement partagé par plusieurs types. Plus précisément, il liste les caractéristiques (fonctions, types associés, contraintes) que doivent implémenter les types implémentant ce *trait*.

Définition d'un trait :

```
1 trait CmpToZero {
2     fn is_zero(&self) -> bool;
3 }
```

Implémentation d'un trait par une structure :

```
1 struct Coord { x: i32, y: i32 }
2
3 impl CmpToZero for Coord {
4     fn is_zero(&self) -> bool {
5         (self.x == 0) && (self.y == 0)
6     }
7 }
```

Traits : implémentation par défaut

Un trait peut donner une implémentation par défaut pour une ou plusieurs fonctions :

```
1 trait CmpToZero {
2     fn is_zero(&self) -> bool;
3     fn is_not_zero(&self) -> bool {
4         !self.is_zero()
5     }
6 }
```

Cette implémentation par défaut peut être redéfinie lors de l'implémentation du trait sur un type.

Traits : paramètres des fonctions

Il est possible de spécifier qu'un argument d'une fonction doit implémenter un trait particulier :

```
1 fn check(val: &impl CmpToZero) {
2     if val.is_zero() { println!("Zero") } else { println!("Not zero") };
3 }
```

Ce code est équivalent à la notation :

```
1 fn check<T: CmpToZero>(val: &T) {
2     if val.is_zero() { println!("Zero") } else { println!("Not zero") };
3 }
```

Traits : paramètres des fonctions (suite)

Il est possible de spécifier qu'un argument implémente plusieurs traits :

```
fn f(t: &(impl Trait1 + Trait2), u: &(impl Trait3 + Trait4)) -> usize {}
```

Équivalent à :

```
fn f<T: Trait1 + Trait2, U: Trait3 + Trait4>(t: &T, u: &U) -> usize {}
```

Il est aussi possible d'écrire, pour simplifier la lecture de la signature de la fonction :

```
fn f<T, U>(t: &T, u: &U) -> usize
    where T: Trait1 + Trait2,
          U: Trait3 + Trait4
{}
```

Traits : paramètres des fonctions, &impl et &dyn

Lorsqu'on utilise `&impl Trait` dans un paramètre de fonction :

- La fonction est instanciée avec le type concret de l'argument passé (*monomorphization*).
- Un appel avec différents types de paramètres génère différentes copies de la fonction.
- Chaque copie, connaissant le type concret, n'a aucun surcoût d'aiguillage.

Il est aussi possible d'utiliser du polymorphisme avec `&dyn Trait`, auquel cas une seule copie de la fonction est générée et les appels aux fonctions associées au trait sont dynamiques :

```
1 fn f(t: &dyn Trait) -> usize { … }
```

Traits : retour de fonction

Il est possible de spécifier qu'une fonction renvoie un type implémentant un trait :

```
fn f(x: i32, y: i32) -> impl CmpToZero {
    Coord { x, y }
}
```

Attention, avec cette syntaxe, votre fonction ne peut renvoyer qu'un seul type. Si `Value` implémente aussi le trait `CmpToZero`, vous ne pouvez pas écrire :

```
fn g(x: i32, y: i32, c: bool) -> impl CmpToZero {
    if c {
        Coord { x, y }
    } else {
        Value { x }
    }
}
```

Traits standards

La bibliothèque standard de Rust définit un certain nombre de traits prédéfinis importants :

- Comparaison : `PartialEq`, `Eq`, `PartialOrd`, `Ord`
- Formatage : `Debug`, `Display`
- Opérations : `Add`, `Deref`, `Index`…
- Conversions : `From`, `Into`, `AsRef`, `AsMut`…
- Copie d'objet : `Copy`, `Clone`
- Multi-thread : `Send`, `Sync`

Certains de ces traits seront détaillés dans la suite du cours

Héritage de traits prédéfinis

Certains des traits prédéfinis viennent avec des macros permettant d'obtenir une implémentation (sous certaines conditions) "gratuite" de ce trait.

```
1  #[derive(Debug)]
2  struct Coord {
3      x: i32,
4      y: i32,
5  }
6
7  let c = Coord { x: 0, y: 0 };
8  println!("{c:?}");
```

```
Coord { x: 0, y: 0 }
```

Petite parenthèse sur l'affichage

Les marqueurs suivants (entre autres) peuvent être utilisés :

- `{}` : utilise le trait `Display`
- `{:?}` : utilise le trait `Debug`, qui peut être dérivé automatiquement
- `{:#?}` : utilise le trait `Debug` sous sa forme alternative (pretty-printing pour les structures par exemple)

```
1  let coord = Coord { x: 17, y: 42 };
2  println!("coord = {coord:?}");     // ou println!("coord = {:?}", coord);
3  println!("coord = {coord:#?}");     // ou println!("coord = {:#?}", coord);
```

```
coord = Coord { x: 17, y: 42 }
coord = Coord {
    x: 17,
    y: 42,
}
```

Clone

`Clone` indique que l'on peut dupliquer un objet explicitement (en appelant la méthode `clone`). Il est possible d'obtenir une implémentation par défaut de ce trait si tous les membres de la structure implémentent déjà ce trait. Dans ce cas, l'implémentation par défaut appelle la méthode `clone` sur chacun des membres de la structure.

```
1  #[derive(Clone)]
2  struct Coord {
3      x: i32,
4      y: i32,
5  }
6
7  let x = Coord { x: 0, y: 0 };
8  let y = x.clone();
```

Il est également possible d'implémenter soi-même `Clone` sur un type en définissant la méthode `clone`.

Copy

```
struct S { a: u32 };

let x = S { a: 42 };
let y = x;
// x can no longer be used (moved to y)
```

Il est possible de changer ce comportement en indiquant que la structure `S` utilise une sémantique de copie.

```
#[derive(Clone, Copy)]
struct S { a: u32 };

let x = S { a: 42 };
let y = x;
// x and y can be used (y is a copy of x)
```

Le trait `Copy` indique que l'on peut réaliser une copie bit-à-bit du contenu de la structure.

Différence entre Clone et Copy

`Copy` indique qu'il est possible de copier l'objet en copiant sa représentation mémoire. Cela n'est pas le cas pour certains types, par exemple un objet de type `Vec` contient l'adresse en mémoire d'où sont stockées ses données, ça n'aurait pas de sens de recopier l'information telle quelle car nous aurions alors deux objets qui pointent vers les mêmes données.

`Clone` permet d'implémenter l'opération `clone()` : dans l'exemple de `Vec`, on peut procéder à une duplication des données pour obtenir deux vecteurs identiques contenant des copies de leurs valeurs.

Quand un objet implémente `Copy`, on peut écrire l'implémentation de `Clone` à la main (même si on peut en général la faire dériver automatiquement) :

```
1  impl Clone for S {
2    fn clone(&self) -> S {
3      *self   // Possible because S is Copy
4    }
5  }
```

Retour sur les types : Vec

`Vec` est un type générique représentant un vecteur :

- Les objets contenus sont tous de même type
- Il est possible d'ajouter ou de retrancher des objets à la fin du vecteur
- Le vecteur croît lorsque cela est nécessaire (si sa capacité devient inférieur au nombre d'éléments contenus) en allouant une zone mémoire suffisante
- Un vecteur est initialisé avec la macro `vec![]`, un vecteur vide peut être créé avec `Vec::new()`

```
1 let mut v: Vec<i32> = vec![10, 20, 30];
2 v.push(42);
3 v.push(17);
4 println!("v = {v:?}");
```

```
v = [10, 20, 30, 42, 17]
```

Les tranches (slices)

Il est possible de représenter des blocs mémoire contenant un nombre variable de valeurs d'un type donné `T`. C'est une *slice* (une tranche) : `[T]`.

Les tableaux et les vecteurs peuvent être référencés comme des slices :

```
1  let v = vec![10, 20, 30, 40, 50, 60, 70];
2  let slice_v1: &[i32] = &v[..];            // Slice from vector
3  let slice_v2: &[i32] = &v[2..5];          // Slice of size 3 (v[2], v[3], v[4])
4
5  let slice_a: &[i32] = &[1, 2, 3];         // Slice from array
6
7  println!("v1 = {slice_v1:?}, v2 = {slice_v2:?}, a = {slice_a:?}");
```

```
v1 = [10, 20, 30, 40, 50, 60, 70], v2 = [30, 40, 50], a = [1, 2, 3]
```

Note sur les indices

On a vu la notation `v[2..5]` apparaître. Voici les différents types de *ranges* possibles en Rust et le nom de leur type :

```
let v = vec![10, 20, 30, 40, 50, 60, 70];
println!("v = {v:?}");
println!("v[1..4] = {:?}", &v[1..4]);       // Range
println!("v[1..=4] = {:?}", &v[1..=4]);     // RangeInclusive
println!("v[2..] = {:?}", &v[2..]);         // RangeFrom
println!("v[..3] = {:?}", &v[..3]);         // RangeTo
println!("v[..=3] = {:?}", &v[..=3]);       // RangeToInclusive
println!("v[..] = {:?}", &v[..]);           // RangeFull
```

```
v = [10, 20, 30, 40, 50, 60, 70]
v[1..4] = [20, 30, 40]
v[1..=4] = [20, 30, 40, 50]
v[2..] = [30, 40, 50, 60, 70]
v[..3] = [10, 20, 30]
v[..=3] = [10, 20, 30, 40]
v[..] = [10, 20, 30, 40, 50, 60, 70]
```

Résumons : tableaux, vecteurs, slices

- Tableau : nombre d'éléments fixe, les éléments ne bougent pas en mémoire
- Slice : nombre d'éléments dynamique mais non modifiable, les éléments ne bougent pas en mémoire
- Vecteur : nombre d'éléments variable, les éléments peuvent bouger en mémoire si la capacité (taille allouée) du vecteur devient insuffisante

```
1  fn square(s: &mut [i32]) {      // Take a slice as argument
2    for i in s { *i = *i * *i; }  // Square every element of the slice
3  }
4  let mut v = vec![10, 20, 30, 40, 50];
5  let mut a = [1, 2, 3, 4, 5];
6  square(&mut v[2..4]);
7  square(&mut a[1..4]);
8  println!("v = {v:?}, a = {a:?}");
```

```
v = [10, 20, 900, 1600, 50], a = [1, 4, 9, 16, 5]
```

Chaînes de caractères (1/4)

Rust a une approche plus saine que beaucoup d'autres langages des chaînes de caractères :

- Un caractère (type `char`) est un entier 32 bits représentant une valeur scalaire du standard « universel » Unicode.
- Une chaîne de caractères est une suite de caractères, chaque caractère étant codé au format UTF-8 (et donc de longueur variable).
- Une chaîne de caractères est représentée en mémoire par un ensemble d'octets (une slice `[u8]`) qui doivent constituer une suite de caractères en UTF-8 valide et le nombre d'octets.
- Le type `&str` est similaire à un `&[u8]`, donc un pointeur et une longueur en octets.

```
1 let name: &str = "world";       // The complete type is &'static str
2 println!("Hello, {name}!");
```

Chaînes de caractères (2/4)

On peut construire une chaîne à partir d'une suite d'octets :

```
let data = [0x41, 0x42, 0x43, 0x44, 0x20, 0xF0, 0x9F, 0x98, 0x80];
let s = std::str::from_utf8(&data[..]).unwrap();
println!("s = `{s}'");
println!("s.len() = {}", s.len());
println!("s.chars().count() = {}", s.chars().count());
println!("s.chars() = {:?}", s.chars().collect::<Vec<char>>());
println!("s.chars() = {:x?}", s.chars().map(|c| c as u32).collect::<Vec<_>>());
```

```
s = `ABCD 😀'
s.len() = 9
s.chars().count() = 6
s.chars() = ['A', 'B', 'C', 'D', ' ', '😀']
s.chars() = [0x41, 0x42, 0x43, 0x44, 0x20, 0x1f600]
```

Il y a six caractères (A, B, C, D, une espace, 😀) qui nécessitent 9 octets pour les représenter (le smiley, dont le code Unicode est 0x1f600, nécessite 4 octets en UTF-8).

Chaînes de caractères (3/4)

Il y a différents moyens de manipuler une chaîne de caractères :

- `&str` : contenu emprunté immuable de taille fixe (*borrowed*)
- `&mut str` : contenu emprunté modifiable de taille fixe (*mutably borrowed*)
- `String` : contenu modifiable (*owned*)

On peut passer de l'un à l'autre :

```
1  let s: String = String::from("hello 😀");  // Yes, Unicode directly in source file
2  let s: String = "hello 😀".to_owned();     // All those make a copy of the borrowed
3  let s: String = "hello 😀".to_string();    // string in order to own the new copy
4
5  let r: &str = s.as_str();                 // Lives no longer than s
```

La méthode `to_string()` est accessible ici car le type `&str` implémente le trait `Display`, et tous les types implémentant `Display` implémentent automatiquement le trait `ToString` qui ajoute la méthode `to_string()`.

Chaînes de caractères (4/4)

Un objet de type `String` est similaire à un `Vec` et contient un pointeur, une capacité et une longueur (en octets). Si la chaîne est modifiée et que la capacité n'est pas suffisante, elle peut être réallouée.

```
1  fn info_on(s: &String, step: usize) {
2      println!("{step}- s = `{s}', len = {}, capacity = {}, addr = {:#x}",
3          s.len(), s.capacity(), &raw const s.as_bytes()[0] as usize);
4  }
5
6  let mut s = String::from("Hello");      info_on(&s, 1);
7  s.push_str(", most wonderful world!"); info_on(&s, 2);
8  s.truncate(5);                         info_on(&s, 3);
9  s.push_str(", world!");                info_on(&s, 4);
```

```
1- s = `Hello', len = 5, capacity = 5, addr = 0x55769d931aa0
2- s = `Hello, most wonderful world!', len = 28, capacity = 28, addr = 0x55769d931b30
3- s = `Hello', len = 5, capacity = 28, addr = 0x55769d931b30
4- s = `Hello, world!', len = 13, capacity = 28, addr = 0x55769d931b30
```

Collections

La bibliothèque standard de Rust fournit un certain nombre de collections

- Séquences
  - `Vec`
  - `VecDeque`
  - `LinkedList`
- Dictionnaires (*map*)
  - `HashMap`
  - `BTreeMap` (trié sur les clés)
- Ensembles (*set*)
  - `HashSet`
  - `BTreeSet` (trié sur les éléments)
- Autre
  - `BinaryHeap`

Vec - Ajout d'éléments

- `push` : ajout d'un élément en fin

```rust
let mut v: Vec<i32> = vec![10, 20, 30];
v.push(42);
println!("v = {v:?}");
```

```
v = [10, 20, 30, 42]
```

- `insert` : position arbitraire (panique si la position indiquée est strictement supérieure à la taille actuelle)

```rust
let mut v: Vec<i32> = vec![10, 20, 30];
v.insert(1, 42);
println!("v = {v:?}");
```

```
v = [10, 42, 20, 30]
```

Vec - Ajout d'éléments

- append : ajout de tous les éléments d'un autre vecteur (qui devient vide)

```rust
let mut v1: Vec<i32> = vec![10, 20, 30];
let mut v2: Vec<i32> = vec![15, 25, 35];
v1.append(&mut v2);
println!("v1 = {v1:?}");
println!("v2 = {v2:?}");
```

```
v1 = [10, 20, 30, 15, 25, 35]
v2 = []
```

Vec - Suppression d'éléments

- `pop` : renvoie et supprime le dernier élément (s'il existe)

```
1  let mut v = vec![10];
2  let e1 = v.pop();
3  let e2 = v.pop();
4  println!("e1 = {e1:?}");
5  println!("e2 = {e2:?}");
```

```
e1 = Some(10)
e2 = None
```

- `remove` : supprime un élément à une position arbitraire (panique si l'index dépasse les limites du vecteur)

```
1  let mut v = vec![10, 20, 30];
2  v.remove(1);
3  println!("v = {v:?}");
```

```
v = [10, 30]
```

Vec - Suppression d'éléments

- `retain` : ne conserve dans le vecteur que les éléments qui valident une condition

```
1  let mut v = vec![1, 2, 3, 4];
2  v.retain(|&x| x % 2 == 0);
3  println!("v = {v:?}");
```

```
v = [2, 4]
```

- `clear` : supprime tous les éléments

Vec - Accès à un élément

- pop (vu précédemment)
- get et get_mut : renvoie une option vers une référence (éventuellement mutable) sur un élément ou une sous slice

```
1 let v: Vec<i32> = vec![1, 2, 3, 4];
2 let e1: Option<&i32> = v.get(1);
3 let e2: Option<&[i32]> = v.get(1..3);
4 println!("e1 = {e1:?}");
5 println!("e2 = {e2:?}");
```

```
e1 = Some(2)
e2 = Some([2, 3])
```

Ces fonctions proviennent de l'implémentation par Vec du trait Deref

Vec - Accès à un élément

- `Vec` implémente également le trait `Index` et `IndexMut`, ce qui permet d'écrire

```rust
let v: Vec<i32> = vec![1, 2, 3, 4];
let e1: i32 = v[1];    // It works because i32 implements the Copy trait
println!("e1 = {e1:?}");
```

```
e1 = 2
```

Attention, le programme paniquera si l'index est en dehors des limites du vecteur.

Itérateurs

Un itérateur implémente le trait `Iterator`. Ce trait contient :

- Un type `Item` qui représente le type des éléments renvoyés par l'itérateur
- Une fonction `fn next(&mut self) -> Option<Self::Item>` qui renvoie le prochain élément
- Une série de fonctions déjà implémentées (`map`, `filter`, `count`, `zip`…) mais qui peuvent être spécialisées pour des raisons d'efficacité.

Itérateurs

```rust
struct Counter { value: u8 }

impl Iterator for Counter {
    type Item = u8;

    fn next(&mut self) -> Option<Self::Item> {
        match self.value {
            255 => None,
            _ => { self.value += 1; Some(self.value) }
        }
    }
}

fn main() {
    let mut c = Counter { value: 0 };
    while let Some(v) = c.next() {
        println!("v = {v:?}");
    }
}
```

Itérateurs

```
v = 1
v = 2
// ...
v = 255
```

Au passage, on a introduit la construction `while let` :

```
while let Some(v) = c.next() {
    // ...
}
```

Il existe aussi `if let` :

```
if let Some(v) = c.next() {
    // ...
}
```

Itérateurs

Le trait `Iterator` fournit un ensemble de fonctions utiles : `count`, `min`, `max`, `nth`…

Par exemple (suite de l'exemple précédent) :

```
println!("count = {}", c.count());
```

```
count = 255
```

Il fournit également des fonctions qui transforment un itérateur en un autre (des adaptateurs). C'est par exemple le cas des fonctions `map`, `filter`, `enumerate`…

Note : on aurait aussi pu implémenter `count` plus efficacement en le spécialisant :

```
impl Iterator for Counter {
    …
    // Not necessary, but more efficient
    fn count(self) -> usize { 255 - self.value as usize }
}
```

Itérateurs

```rust
fn main() {
    let mut c = Counter { count: 0 };
    let mut i = c.map(|x| 2 * (x as u32));

    while let Some(v) = i.next() {
        println!("v = {v:?}");
    }
}
```

```
v = 2
v = 4
// ...
v = 512
```

Itérateurs

Par défaut, les itérateurs et les adaptateurs sont paresseux (*lazy*), il ne font rien tant que la fonction `next` n'est pas appelée (explicitement ou implicitement).

```
1  fn main() {
2      let mut c = Counter { count: 0 };
3      c.map(|x| println!("{x:?}"));
4  }
```

```
warning: unused `std::iter::Map` that must be used
   |
16 |     c.map(|x| println!("{x:?}"));
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(unused_must_use)]` on by default
   = note: iterators are lazy and do nothing unless consumed
```

Certaines méthodes comme `for_each` consomment explicitement l'itérateur :

```
1  c.for_each(|x| println!("{x:?}"));
```

Vec et les itérateurs

Un `Vec<T>` implémente trois fonctions retournant des itérateurs :

- `iter(&self)` qui renvoie des références vers les éléments du vecteur `&T` (implémentation du trait `Deref`)
- `iter_mut(&mut self)` qui renvoie des références mutables vers les éléments `&mut T` (implémentation du trait `Deref`)
- `into_iter(self)` qui renvoie les éléments eux-mêmes `T` (implémentation du trait `IntoIterator`)

```
1  let v = vec![1, 2, 3];
2  let mut i = v.into_iter();
3  while let Some(e) = i.next() {
4      println!("e = {e:?}");
5  }
```

```
e = 1
e = 2
e = 3
```

for in

```rust
let v = vec![1, 2, 3];

for e in v {
    println!("e = {e:?}");
}

// v does not exist anymore, it has been consumed by the implicit
// call to into_iter() caused by the for loop
```

```
e = 1
e = 2
e = 3
```

Dans cette construction, ce qui suit `in` doit implémenter le trait `IntoIterator`. La boucle commence par appeler la fonction `into_iter` pour récupérer l'itérateur, puis itère tant que la fonction `next` ne renvoie pas `None`.

Il est à noter que la bibliothèque standard fournit une implémentation du trait `IntoIterator` pour tous les itérateurs. On peut donc mettre directement un itérateur après le mot clé `in`.

Itérations et `&Vec`

`into_iter()` est disponible sur `&Vec` et `&mut Vec` et appelle respectivement `iter()` et `iter_mut()` sur l'objet référencé. On peut donc itérer sur une référence au vecteur, cela itérera en fait sur des références pointant sur les éléments. C'est un raccourci d'écriture pratique à utiliser.

```rust
let v = vec![1, 2, 3];

for e in &v {     // Equivalent to: for e in v.iter()
    println!("e = {:?}", *e);
}

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

```
e = 1
e = 2
e = 3
v = [1, 2, 3]
```

Exemple avec `&mut Vec`

```rust
let mut v = vec![1, 2, 3];

for e in &mut v {   // Equivalent to: for e in v.iter_mut()
    *e += 5;
}

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

```
v = [6, 7, 8]
```

Cet opérateur est également implémenté sur les références aux tableaux et aux slices.

Allocation dynamique : Box<T>

Box<T> est un pointeur intelligent (*smart pointer*) simple permettant d'allouer de la mémoire pour une instance de T sur le tas (*heap*).

```
1 {
2     let b: Box<u8> = Box::new(42);
3     let val = *b;
4 } // b goes out of the scope, the memory is freed
```

La mémoire allouée est automatiquement libérée à la fin de la durée de vie.

Allocation dynamique : Box<T>

Box peut, par exemple, être utile lorsque l'on veut avoir des types récursifs, exemple :

```
1 enum List {
2     Cons(i32, List),
3     Nil,
4 }
```

```
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |               ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List` representable
  |
2 |     Cons(i32, Box<List>),
  |               ^^^^    ^
```

Allocation dynamique : Box<T>

En effet, le compilateur doit connaître la taille du type `List`. Or ici, la définition de cette énumération est récursive donc ce n'est pas possible.

Comme le suggère le compilateur, on peut utiliser `Box` pour résoudre le problème :

```
1 enum List {
2     Cons(i32, Box<List>),
3     Nil,
4 }
```

`Box<List>` étant un pointeur, sa taille est connue et donc le compilateur peut connaître la taille de l'énumération `List`.

Allocation dynamique : `Box<T>`

Comme nous l'avons vu, il est possible d'accéder à la valeur pointée en déréférençant à l'aide de `*`, comme pour une référence. Ceci est rendu possible par l'implémentation du trait `Deref` par `Box<T>`.

Ce trait contient un type `Target` et une fonction `fn deref(&self) -> &Self::Target`.

`Box<T>` implémente `Deref` avec `Target = T`.

Lorsque l'on écrit :

```
1 let b: Box<u8> = Box::new(42);
2 let val = *b;
```

Le compilateur transforme le code en :

```
1 let b: Box<u8> = Box::new(42);
2 let val = *(b.deref());
```

Deref coercion

Le compilateur utilise implicitement `Deref` dans plusieurs cas.

Si `T` implémente `Deref<Target = U>` et que `x` est de type `T` :

- Si `T` n'est pas une référence ni un pointeur brut (voir plus loin), `*x` est équivalent à `*Deref::deref(&x)` dans un contexte non mutable
- Les valeurs du type `&T` peuvent être transformées (*coercion*) en valeurs du type `&U`
- `T` implémente implicitement toutes les méthodes immuables du type `U`

Lorsque l'on est dans un contexte mutable, `DerefMut` est utilisé à la place de `Deref` :

- Dans un contexte mutable, `*x` est équivalent à `*DerefMut::deref_mut(&mut x)`
- Les valeurs du type `&mut T` peuvent être transformées en valeurs du type `&mut U`

Deref coercion

Exemple : `String` implémente `Deref<Target=str>`, donc il est possible de passer une référence vers un `String` à une fonction attendant une référence vers un `str`.

```
1 fn takes_str(s: &str) { }
2 let s: String = String::from("Hello");
3 takes_str(&s);
```

De plus, `String` hérite de toutes les fonctions implémentées sur `str`.

Drop

`Box<T>` implémente également le trait `Drop`. Ce trait requiert une méthode `drop(&mut self)` qui sera appelée lorsque l'objet n'est plus utilisé (par exemple qu'il sort du scope courant). Attention, il n'est pas possible d'appeler directement cette fonction `drop`.

C'est ainsi que la mémoire allouée pour stocker l'instance de `T` est libérée automatiquement.

Il est possible de faire disparaître un objet avant la fin du scope en appelant explicitement la fonction `std::mem::drop`.

De même, on peut faire disparaître un objet **sans** appeler son destructeur `drop` grâce à la fonction `std::mem::forget`. Cette fonction est rarement utilisée et peut provoquer des fuites mémoire.

Rc<T>

Rc<T> est un pointeur avec compteur de référence. Il permet de partager la propriété d'une valeur de type T.

```
1  use std::rc::Rc;
2  struct S { }
3
4  fn main() {
5      let a = Rc::new(S {});
6      let b = Rc::clone(&a);
7      println!("Reference counter = {}", Rc::strong_count(&a));
8      {
9          let c = Rc::clone(&a);
10         println!("Reference counter = {}", Rc::strong_count(&a));
11     }
12     println!("Reference counter = {}", Rc::strong_count(&a));
13 }
```

```
Reference counter = 2
Reference counter = 3
Reference counter = 2
```

L'appel à `clone` permet d'obtenir un nouveau pointeur pointant vers le même objet. Le compteur de référence est incrémenté à chaque appel.

Le compteur est décrémenté lorsque les pointeurs terminent leur vie. Lorsque le compteur de référence atteint zéro, la mémoire occupée par l'objet est libérée.

Attention : vous ne pouvez stocker que des objets immuables à l'intérieur d'un `Rc`.

S'il existe un cycle entre deux pointeurs `Rc`, la mémoire ne sera jamais libérée. Pour "casser" le cycle, il existe la méthode `downgrade` qui permet d'obtenir un pointeur `Weak`, qui n'incrémente pas le compteur de référence. Ce pointeur `Weak` peut être utilisé pour récupérer la valeur de l'objet pointé en utilisant la méthode `upgrade` (qui renvoie une `Option<Rc<T>>` : la variante `None` est retournée si entre temps la mémoire a été libérée).

Rc<T>

```rust
use std::rc::{Rc, Weak};
struct S { }

fn main() {
    let a: Rc<S> = Rc::new(S {});
    let b: Weak<S> = Rc::downgrade(&a);
    println!("Reference counter = {}", Rc::strong_count(&a));
    let c: Option<Rc<S>> = b.upgrade();
    println!("Reference counter = {}", Rc::strong_count(&a));
}
```

```
Reference counter = 1
Reference counter = 2
```

Rc<T> et Arc<T>

Rc<T> est prévu pour une utilisation par un seul thread (la mise à jour du compteur n'est pas garantie être atomique).

Vous pouvez utiliser sync::Arc<T> si vous avez besoin d'une version thread-safe.

Interior mutability

Comme nous l'avons vu, sur un objet `T`, on peut avoir au plus :

- soit plusieurs référence immuables (`&T`) sur cet objet,
- soit une référence mutable (`&mut T`) sur cet objet.

et ces règles (*inherited mutability*) sont vérifiées statiquement à la compilation.

Dans certains cas, on pourrait souhaiter être capable de modifier un objet à travers une référence non mutable (par exemple pour pouvoir modifier des objets par l'intermédiaire d'un `Rc<T>`).

`Cell<T>` et `RefCell<T>` permettent (**dans un contexte mono-thread**), d'une façon contrôlée, de réaliser ce qui est appelé *interior mutability*.

RefCell<T>

```rust
use std::cell::RefCell;
struct S { v: u8, }

fn inc(c: &RefCell<S>) {
    let mut m = c.borrow_mut();
    m.v += 1;
}

fn main() {
    let cell = RefCell::new(S { v: 42 });
    inc(&cell);
    println!("v = {}", cell.borrow().v);
}
```

```
v = 43
```

RefCell<T>

- `borrow(&self) -> Ref<'_, T>` permet d'obtenir une référence non mutable sur la valeur à l'intérieur de `RefCell<T>` (plus précisément, elle renvoie un `Ref<T>` qui implémente `Deref<T>` donc se comporte comme une référence)
- `borrow_mut(&self) -> RefMut<'_, T>` permet d'obtenir une référence mutable sur la valeur (plus précisément un `RefMut<T>` qui implémente `Deref<T>` et `DerefMut<T>`)

Ces deux fonctions vérifient les règles concernant les références à l'exécution :

- `borrow` panique si une référence mutable vers l'objet existe déjà
- `borrow_mut` panique si une référence mutable ou non vers l'objet existe déjà

Note : les variantes `try_borrow` et `try_borrow_mut` retournent un `Result` au lieu de paniquer.

RefCell<T>

```rust
use std::cell::RefCell;
struct S { v: u8, }

fn inc(c: &RefCell<S>) {
    let mut m = c.borrow_mut();
    let n = c.borrow();
    m.v += 1;
}

fn main() {
    let cell = RefCell::new(S { v: 42 });
    inc(&cell);
    println!("v = {}", cell.borrow().v);
}
```

```
thread 'main' panicked at 'already mutably borrowed: BorrowError', src/main.rs:6:15
```

Cell<T>

Comme nous l'avons vu il est possible sur un `RefCell<T>` d'obtenir des références mutables et non mutables sur la valeur interne.

`Cell<T>` implémente différemment la mutabilité interne :

- `set(&self, v: T)` permet de remplacer la valeur contenue par une autre, l'ancienne valeur est perdue
- `replace(&self, v: T) -> T` permet de remplacer la valeur contenue par une autre et de retourner la propriété de l'ancienne valeur
- `get(&self) -> T` (si `T` implémente le trait `Copy`) retourne une copie de la valeur contenue

Vous étiez vous déjà demandé comment `Rc<T>` incrémentait ses compteurs de référence quand l'objet n'est accessible qu'en lecture ? Il utilise `Cell<usize>` pour stocker la valeur des compteurs.

Threads

```rust
use std::thread;
use std::time::Duration;

fn main() {
    // Create a separate thread that executes the closure
    let h = thread::spawn(|| {
        for i in 1..5 {
            println!("From the other thread: {i}");
            thread::sleep(Duration::from_millis(1));
        }
    });

    // Rest of the "main" thread
    for i in 1..5 {
        println!("From the main thread: {i}");
        thread::sleep(Duration::from_millis(1));
    }
    // Explicitely wait for the end of the other thread
    h.join().unwrap();
}
```

Threads

```
From the main thread: 1
From the other thread: 1
From the main thread: 2
From the other thread: 2
From the main thread: 3
From the other thread: 3
From the main thread: 4
From the other thread: 4
```

Threads : capture de l'environnement

```rust
use std::thread;
struct S { v: u8, }

fn main() {
    let s = S { v: 42 };

    let h = thread::spawn(|| {
        println!("v = {}", s.v);
    });

    h.join().unwrap();
}
```

Threads : capture de l'environnement

```
error[E0373]: closure may outlive the current function, but it borrows `s.v`,
 which is owned by the current function
 --> src/main.rs:7:27
  |
7 |     let h = thread::spawn(|| {
  |                           ^^ may outlive borrowed value `s.v`
8 |         println!("v = {}", s.v);
  |                            - `s.v` is borrowed here
note: function requires argument type to outlive `'static`
 --> src/main.rs:7:13
  |
7 |       let h = thread::spawn(|| {
  |  _____________^
8 | |         println!("v = {}", s.v);
9 | |     });
  | |_____^
help: to force the closure to take ownership of `s.v` (and any other referenced variables),
 use the `move` keyword
7 |     let h = thread::spawn(move || {
  |                           ^^^^^^^
```

Threads : capture de l'environnement

Comme on l'a vu précédemment avec les fonctions anonymes, la fonction passée à `spawn` emprunte une référence sur les objets externes. Ici le compilateur ne peut pas garantir que l'objet (appartenant à une fonction s'exécutant dans le thread principal) va vivre au moins aussi longtemps que la référence (utilisée dans le second thread).

Une solution est d'utiliser le mot clé `move` pour transférer la propriété des variables utilisées au thread :

```
1  fn main() {
2      let s = S { v: 42 };
3
4      let h = thread::spawn(move || {
5          println!("v = {}", s.v);
6      });
7
8      // s.v is no longer accessible here
9      h.join().unwrap();
10 }
```

Transfert de données entre thread

Il est possible de communiquer entre les threads par passage de message.

```rust
use std::sync::mpsc;
use std::thread;
struct S { v: u8, }

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let s = S { v: 42 };
        tx.send(s).unwrap();
    });

    let r = rx.recv().unwrap();
    println!("Received: {}", r.v);
}
```

```
Received: 42
```

Transfert de données entre thread

`mpsc::channel<T>() -> (Sender<T>, Receiver<T>)` crée un canal de communication entre deux (ou plus) threads. Le modèle implémenté est *Multiple Producer Single Consumer* (plusieurs émetteur, un seul récepteur).

La fonction `send(&self, t: T) -> Result<(), SendError<T>>` “consomme” l'objet envoyé (il n'est plus utilisable par le thread émetteur après l'appel à `send`).

Il est possible de créer plusieurs émetteurs en appelant `clone` sur l'émetteur (`Sender<T>`) créé par `channel`.

Mutex<T>

Un Mutex permet de garantir l'accès exclusif à une donnée partagée.

La fonction `fn new(t: T) -> Mutex<T>` crée un mutex, déverrouillé, contenant la donnée `t`.

Une fois dans le sémaphore (le sémaphore devient propriétaire de la donnée), la donnée n'est accessible que via les fonctions `lock` et `try_lock` :

La fonction `fn lock(&self) -> LockResult<MutexGuard<'_, T>>` bloque tant que le mutex est verrouillé. Dès qu'il est disponible, le mutex est verrouillé et la donnée contenue est accessible via le `MutexGuard` renvoyé (qui implémente `Deref` et `DerefMut`). Lorsque le `MutexGuard` retourné disparaît (fin du scope ou appel explicite à `drop`), le mutex est automatiquement déverrouillé. Cette fonction retourne la variante d'erreur si un autre thread a paniqué alors qu'elle avait verrouillé le mutex (le mutex est dit empoisonné).

Arc<T>

Un mutex seul n'est pas suffisant pour partager une donnée entre plusieurs threads. En effet, un seul thread peut posséder le mutex et le compilateur ne pourra pas vérifier (et donc interdira) que les références vers ce mutex, qui pourrait acquises par d'autres thread, vivent moins longtemps que le mutex.

Pour permettre la possession multiple du mutex, on peut utiliser `Arc<T>` qui est une variante thread-safe de `Rc<T>`.

Arc<T> : Exemple

```rust
use std::sync::{Arc, Mutex};
use std::thread;
struct S { v: u8, }

fn main() {
    let s = Arc::new(Mutex::new(S { v: 0 } ));
    let mut handles = vec![];
    for _ in 0..10 {
        let new_s = Arc::clone(&s);
        let handle = thread::spawn(move || {
            let mut st = new_s.lock().unwrap();
            st.v += 1;
        });
        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    println!("v: {}", s.lock().unwrap().v); // v = 10
}
```

Traits Sync / Send

Send

Le trait `Send` sur un type (qui est un *marker trait*, c'est-à-dire un trait "vide" qui ne sert qu'à indiquer qu'une propriété est vraie sur un type) indique que la propriété d'un objet de ce type peut être transférée d'un thread à un autre.
Par exemple `Rc<T>` n'implémente pas `Send` et donc le compilateur pourra détecter le fait que l'on passe la propriété d'un `Rc<T>` d'un thread à un autre (essayez de remplacer `Arc` par `Rc` dans l'exemple de la page précédente pour voir). `Arc<T>` est `Send` si `T` l'est.

Sync

Le trait `Sync` implémenté sur un type indique qu'il est possible de partager une référence vers un objet de ce type par plusieurs thread.

Traits `Sync` / `Send`

Ces deux traits sont automatiquement implémentés par le compilateur (il n'y a pas besoin d'utiliser `derive`) quand il juge que c'est approprié, c'est-à-dire quand tous les membres d'une structure implémentent ces traits. Presque tous les types prédéfinis implémentent `Send` et `Sync` sauf :

- les pointeurs bruts ne sont ni `Send` ni `Sync`
- `UnsafeCell`, `Cell` et `RefCell` ne sont pas `Sync`
- `Rc` n'est ni `Send` ni `Sync`

L'implémentation manuelle de ces traits est une opération *unsafe* (voir plus loin dans le cours).

Packages & crates

Un crate est un arbre de modules permettant de produire soit une application (*binary crate*) soit une bibliothèque (*library crate*). Il regroupe un ensemble cohérent de fonctionnalités au sein d'un même scope afin de pouvoir les réutiliser dans différents projets.

Un package regroupe un ou plusieurs crates. Il doit contenir au minimum un crate et au plus un crate bibliothèque. Il contient un fichier `Cargo.toml` indiquant comment construire ces crates.

En créant un projet avec :

```
$ cargo new project
     Created binary (application) `project` package
```

cargo crée un package nommé `project` avec un crate de type application. Si on avait ajouté `--lib`, ce crate aurait été de type bibliothèque.

Packages & crates

À l'intérieur de ce projet, s'il existe un fichier `src/main.rs`, il s'agira de la racine d'un *binary crate* portant le même nom que le package. De même, s'il existe un fichier `src/lib.rs`, il s'agira de la racine d'un *library crate* portant le même nom que le package.

Il est possible de créer plusieurs *binary crate* à l'intérieur d'un package en plaçant des fichiers Rust dans le répertoire `src/bin`. Chaque fichier sera considéré comme un *binary crate*.

Modules

Au sein d'un crate, il est possible de découper les fonctionnalités en modules.

```rust
// src/main.rs
mod a {
    mod ab {
        fn ab_f1() { }
    }
}
mod b {
    fn b_f() { }
}
```

On a défini la hiérarchie suivante :

```
  crate
- module a
-- module ab
--- fonction ab_f1
- module b
-- fonction b_f
```

Modules & chemins

Pour se référer à un élément (fonction, structure, etc.), il faut indiquer son chemin dans cette hiérarchie. Ce chemin peut être :

- Absolu par rapport au crate : `crate::a::ab::ab_f1`
- Relatif à la position actuelle. Exemple, depuis le module `a` :
  - `ab::ab_f1`
  - `self::ab::ab_f1`
  - `super::b::b_f`

Modules & visibilité

Par défaut, les membres d'un module (fonctions, structures, sous-modules, etc.) sont privés et ne sont visibles que des membres de ce module.

Pour les rendre visibles, il faut utiliser le mot clé `pub`.

```
1  mod a {
2      pub mod ab {
3          pub fn ab_f1() { }
4      }
5  }
6  mod b {
7      fn b_f() { crate::a::ab::ab_f1(); }
8  }
```

Modules & visibilité : cas des structures

```rust
mod a {
    pub struct S {
        pub v: u8,
        w: u8,
    }
    impl S {
        pub fn new(v: u8, w: u8) -> Self {
            S { v, w }
        }
    }
}
fn f() {
    let mut s = a::S::new(42, 43);
    s.v = 0;
    // s.w = 1 // Will not compile
}
```

Modules & visibilité : cas des énumérations

Si une énumération est déclarée `pub`, toutes ses variantes sont également publiques.

Modules & fichiers

Il est possible de répartir les modules dans différents fichiers.

Dans `src/main.rs` :

```
1 mod a;
```

indique que le contenu du module `a` est à charger soit depuis le fichier `src/a.rs`, soit depuis le fichier `src/a/mod.rs`. Ce fichier pourrait contenir :

```
1 pub mod ab {
2     pub fn ab_f1() { }
3 }
```

Modules & fichiers

De même, dans `src/a.rs`, il est possible de déclarer que le contenu du module `ab` se trouve dans son fichier propre en écrivant :

```
1 pub mod ab;
```

Le contenu du module `a::ab` sera alors lu depuis le fichier `src/a/ab.rs` ou `src/a/ab/mod.rs` :

```
1 pub fn ab_f1() { }
```

Où placer le fichier d'un module ?

Le choix est libre mais doit être unique :

- Si le module est indépendant, `nom-de-module.rs` est approprié
- Si le module est lui-même au sommet d'une hiérarchie, `nom-de-module/mod.rs` peut être préférable

Extrait venant du crate bibliothèque `pathfinding` :

```
directed/
   mod.rs
   astar.rs
   dijkstra.rs
undirected/
   mod.rs
   kruskal.rs
grid.rs
```

En cas d'utilisation de plusieurs chemins pour définir un module, `cargo` échouera lors de la compilation.

use

Le mot clé `use` permet d'importer des définitions dans le scope actuel :

```
1  mod a {
2      pub mod ab {
3          pub fn ab_f1() { }
4      }
5  }
6
7  fn f() { a::ab::ab_f1() }
```

```
1  mod a {
2      pub mod ab {
3          pub fn ab_f1() { }
4      }
5  }
6  use a::ab;
7  fn f() { ab::ab_f1() }
```

On aurait pu importer directement la fonction :

```
1  mod a {
2      pub mod ab {
3          pub fn ab_f1() { }
4      }
5  }
6  use a::ab::ab_f1;
7  fn f() { ab_f1() }
```

Idiomatiquement, on a tendant à importer les chemins complets vers les structures (pour pouvoir appeler directement la structure par son nom, sauf s'il y a un conflit, voir slide suivant). Pour les fonctions, on importe plutôt le module parent.

use et noms identiques

```
1  mod a { pub struct S { pub v: u8 } }
2  mod b { pub struct S { pub w: u8 } }
3
4  use a::S;
5  use b::S as BS;
6
7  fn f() { BS { w:42 }; }
```

as définit un alias, notamment pour éviter un conflit de nommage.

Restriction des autorisations de visibilité

Il est possible de qualifier `pub` pour limiter les permissions accordées sur des entités en cours de définition :

- `pub` : visible pour tous
- `pub(in path)` : `path` doit désigner un ancêtre de l'entité déclarée
- `pub(crate)` : visible dans la totalité du crate courant uniquement
- `pub(super)` : visible du module parent du module courant
- `pub(self)` : autoriser le module courant (inutile)

Il faut aussi prendre en compte les limitations suivantes :

- Il n'est pas possible de rendre visible (comme type d'un paramètre ou champ public d'une structure) des entités de visibilité plus restreinte
- Dans tous les cas, en plus des permissions qui autorisent l'accès sur l'entité elle-même, le chemin jusqu'à l'entité doit pouvoir être emprunté par l'utilisateur

Reexportation des entités importées

Avec `pub use` il est possible d'exporter depuis un module des entités importées d'un autre module :

```
mod implementation_details {
    pub mod api {
        …
    }

    …
}

// This reexports the "api" from the current module. This is allowed because
// the "api" module itself is "pub" and we can reach it through the immediately
// visible "implementation_details". Without this reexport, modules outside
// the current hierarchy could not access "implementation_details::api" because
// they cannot access "implementation_details" which is not public.
pub use implementation_details::api;
```

Les permissions accordées sur la réexportation ne doivent pas excéder celles accordées directement sur l'entité.

Programmation bas-niveau

Références et pointeurs bruts

Malgré les bénéfices apportées par les références, il est parfois souhaitable d'utiliser des pointeurs bruts :

- interfaçage avec d'autres langages de programmation
- programmation bas niveau pour créer des structures de plus haut niveau
- programmation système

Rust permet l'utilisation de pointeurs bruts. On perd par contre les garanties apportées par le langage pour les types classiques, à savoir

- pas de déréférencement possible d'une référence qui serait devenue invalide
- pas de déplacement d'un objet d'un thread à l'autre s'il ne supporte pas la migration
- pas d'accès concurrent à un objet ne le prévoyant pas explicitement

Les types pointeur en Rust

Il existe deux familles de pointeurs bruts (*raw pointers*) en Rust :

- `*const T` : pointeur en lecture seule sur un objet de type `T`
- `*mut T` : pointeur en lecture/écriture sur un objet de type `T`

Ces pointeurs ne sont pas assortis d'une durée de vie. Déréférencer de tels pointeurs est une opération non-sûre qui n'est permise qu'à travers un bloc `unsafe`, car :

- un pointeur peut être nul, contrairement à une référence ;
- un pointeur peut être mal aligné, contrairement à une référence ;
- un pointeur peut pointer vers un objet qui a été désalloué, contrairement à une référence ;
- un pointeur peut pointer vers un objet qui est en train d'être modifié, contrairement à une référence.

Obtention d'un pointeur brut

Pour obtenir un pointeur brut sur un objet `x` de type `T`, on utilisera :

- `&raw const x` pour obtenir un pointeur de type `*const T`
- `&raw mut x` pour obtenir un pointeur de type `*mut T`

On peut également convertir une référence `r` sur un objet de type `T` en pointeur brut avec :

- `&raw const *r` convertit `r` de `&T` en `*const T`
- `&raw mut *r` convertit `r` de `&mut T` en `*mut T`

La conversion peut également être implicite : un `&T` sera toujours accepté là où un `*const T` est attendu, et un `&mut T` pourra être utilisé au lieu d'un `*mut T`.

Conversion entre type de pointeurs

Comme en C, on peut convertir librement des pointeurs bruts :

- `p.cast_const()` transforme un pointeur `p` de `*mut T` en `*const T`
- `p.cast_mut()` transforme un pointeur `p` de `*const T` en `*mut T`
- `p.cast::<U>()` transforme un pointeur `p` de `*const T` en `*const U`, ou de `*mut T` en `*mut U`

unsafe

Le mot clé `unsafe` permet de déclarer qu'un bloc ou une fonction utilisent des constructions risquées au vu des garanties offertes habituellement par Rust.

```
1  /// Add the content designated by two pointers and return the result
2  pub fn add(a: *const u32, b: *const u32) -> u32 {
3    unsafe { *a + *b }
4  }
```

La fonction `add` donne une fausse impression de sécurité puisqu'elle peut être appelée en dehors d'un bloc `unsafe`. Il est possible de la marquer `unsafe` elle-même pour forcer ses utilisateurs à réaliser les risques encourus :

```
1  pub unsafe fn add(a: *const u32, b: *const u32) -> u32 {
2    unsafe { *a + *b }
3  }
```

Exemple : implémentation d'un type pile Stack

Nous pouvons implémenter notre propre type de pile `Stack` :

```
pub struct Stack<T> {
  storage: *mut T,
  size: usize,
}
```

- Rust dispose d'un type `NonNull<T>` permettant de représenter un pointeur équivalent à `*mut T` garanti être non nul, utilisons le
- Afin d'éviter de réallouer une zone plus grande à chaque modification de la pile, nous pouvons lui associer une capacité

```
use std::ptr::NonNull;

pub struct Stack<T> {
  storage: NonNull<T>,
  capacity: usize,
  size: usize,
}
```

Stack : allocation

```rust
use std::alloc::{self, Layout};
use std::ptr::NonNull;

impl<T> Stack<T> {
  /// Allocate a new stack
  pub fn new() -> Self { Self::with_capacity(16) }  // 16 is arbitrary

  /// Allocate a new stack with a given capacity, which must be greater than 0
  pub fn with_capacity(capacity: usize) -> Self {
    assert!(capacity > 0);
    let layout = Layout::array::<T>(capacity).unwrap();
    let storage = unsafe { alloc::alloc(layout).cast::<T>() };
    Stack {
      storage: NonNull::new(storage).unwrap(),
      capacity,
      size: 0,
    }
  }
}
```

Stack : stockage d'un nouvel élément

Il ne faut pas oublier de réallouer une nouvelle zone si la précédente est maintenant trop exigüe.

```
impl<T> Stack<T> {
  pub fn push(&mut self, x: T) {
    if self.size == self.capacity {
      // Reallocate a zone for storing twice the number of elements
      let layout = Layout::array::<T>(self.capacity).unwrap();
      self.capacity *= 2;
      let needed = Layout::array::<T>(self.capacity).unwrap().size();
      self.storage = NonNull::new(unsafe {
        alloc::realloc(self.storage.as_ptr().cast::<u8>(), layout, needed).cast::<T>()
      }).unwrap();
    }
    unsafe { self.storage.as_ptr().offset(self.size as isize).write(x) };
    self.size += 1;
  }
}
```

`(*mut T).write()` recopie l'élement bit à bit sans appeler son destructeur.

Stack : récupération du dernier élement empilé

Aucune réallocation n'est jamais nécessaire puisqu'on retire un élément de la pile ou on signale qu'elle n'en contient aucun :

```
impl<T> Stack<T> {
  pub fn pop(&mut self) -> Option<T> {
    if self.size > 0 {
      self.size -= 1;
      Some(unsafe { self.storage.as_ptr().offset(self.size as isize).read() })
    } else {
      None
    }
  }
}
```

`(*mut T).read()` renvoie une copie bit à bit de l'élément au sommet de la pile dont l'appelant devient propriétaire. Il y a donc une transmission de la propriété de l'appelant de `push()` à l'appelant ultérieur de `pop()`.

Stack : fonctions utilitaires

```rust
impl<T> Stack<T> {
  pub fn size(&self) -> usize { self.size as usize }

  pub fn capacity(&self) -> usize { self.capacity as usize }

  pub fn is_empty(&self) -> bool { self.size == 0 }

  pub fn clear(&mut self) {
    // The destructor of the popped elements will be called
    while let Some(_) = self.pop() {}
  }
}
```

Stack : désallocation

L'implémentation du trait `Drop` pour notre type permet de la désallouer si elle sort du scope courant afin d'éviter les fuites de mémoire. Il ne faut pas oublier d'appeler les destructeurs sur les objets stockés puisqu'ils sont dans une zone mémoire gérée à la main : cela se fait grâce à `self.clear()`.

```
1  use std::alloc::{self, Layout};
2
3  impl<T> Drop for Stack<T> {
4    fn drop(&mut self) {
5      self.clear();
6      unsafe {
7        let layout = Layout::array::<T>(self.capacity).unwrap();
8        alloc::dealloc(self.storage.as_ptr().cast::<u8>(), layout);
9      }
10   }
11 }
```

Stack : tests de base

À lancer avec `cargo test`.

```rust
#[test]
fn test_stack() {
  let mut stack = Stack::new();
  assert_eq!(stack.size(), 0);
  assert_eq!(stack.capacity(), 16);
  for i in 1..=100 { stack.push(i); }
  assert_eq!(stack.size(), 100);
  assert_eq!(stack.capacity(), 128);
  for i in 1..=100 { assert_eq!(stack.pop(), Some(101-i)); }
  assert_eq!(stack.size(), 0);
  assert_eq!(stack.capacity(), 128);
}
```

Stack : tests de drop

```rust
#[test]
fn test_drop_call() {
  use std::{cell::Cell, rc::Rc};
  struct S { pub x: Rc<Cell<usize>> }
  impl Drop for S {
    fn drop(&mut self) { self.x.set(self.x.get() + 1); }
  }
  let counter = Rc::new(Cell::new(0));
  {
    let mut s = Stack::new();
    for _ in 0..100 { s.push(S { x: Rc::clone(&counter) }); }
    assert_eq!(counter.get(), 0);
    // Check that drop() is not called by pop() but at end of _popped scope
    { let _popped = s.pop(); assert_eq!(counter.get(), 0); }
    assert_eq!(counter.get(), 1);
  }
  assert_eq!(counter.get(), 100);
}
```

Stack : tests de drop (suite)

Imaginons que dans `drop()` on ait omis l'appel à `self.clear()` :

```rust
use std::alloc::{self, Layout};

impl<T> Drop for Stack<T> {
  fn drop(&mut self) {
    // Missing call to self.clear(), objects are not popped
    unsafe {
      let layout = Layout::array::<T>(self.capacity).unwrap();
      alloc::dealloc(self.storage.as_ptr().cast::<u8>(), layout);
    }
  }
}
```

Le verdict serait sans appel :

```
thread 'test_drop_call' panicked at 'assertion failed: `(left == right)`
  left: `1`,
 right: `100`', src/lib.rs:107:3
```

Scopes et destructeurs

Quelle est la différence entre

1. `{ let _popped = s.pop(); assert_eq!(counter.get(), 0); }`
2. `{ let _ = s.pop(); assert_eq!(counter.get(), 1); }`
3. `{ s.pop(); assert_eq!(counter.get(), 1); }`

Dans les trois cas, la valeur de retour de `s.pop()` est ignorée car les variables commençant par _ sont des variables fictives.

Mais dans le premier cas, la valeur mise dans `_popped` est détruite à la fin du scope, alors que dans les deux autres elle est détruite immédiatement à la fin de l'instruction courante. Le test aurait donc échoué dans ces deux derniers cas car le compteur serait passé à 1 avant qu'on le teste.