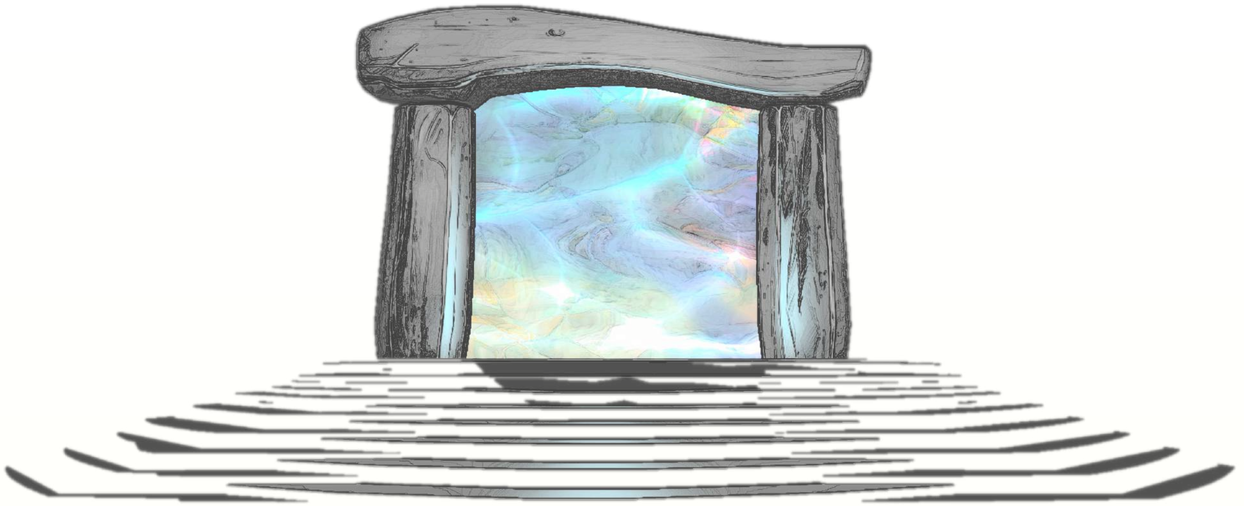


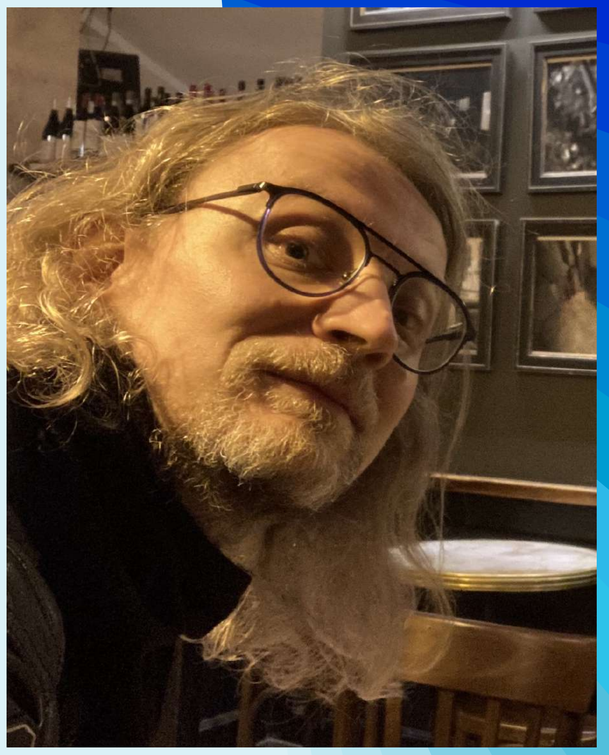
Découverte de **Rust**

avec Pascal HAVÉ ✉

About me

Pascal Havé

- Formateur et consultant:
  « Techniques de développement logiciel et Performances »
- *CTO on demand*
- Software Craftsman
- 20+ ans passés dans le calcul scientifique et le développement de logiciels HPC
- *Tech Lead Rust* @ Nuant / AllianceBlock
- Passionné par les structures intimes des choses
  (et ses implications dans le développement logiciel *clean*)
- Pense que le lemme de Yoneda définit un paradigme intéressant pour se représenter le monde.



https://haveneer.com
pascal@haveneer.com

Qu'est-ce que **Rust** ?

Rust

A systems programming language that runs blazingly fast, prevents almost all crashes, and eliminates data races

Sécurité **Vitesse** **Concurrence**

Rust

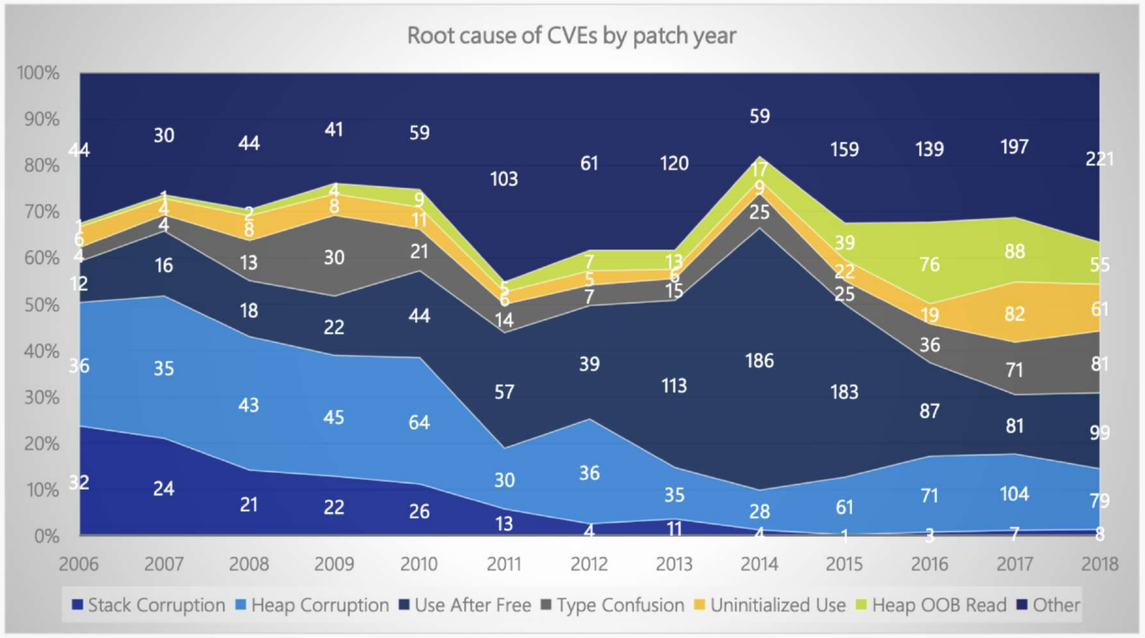
Rust: *a language empowering everyone to build reliable and efficient software.*

Performance **Fiabilité** **Productivité**

L'idée originelle

Pour un meilleur C++

Mozilla Firefox a été écrit en C++ pour des raisons de performances.
Il a ensuite souffert de problèmes de sécurité et de violation de l'intégrité de la mémoire.



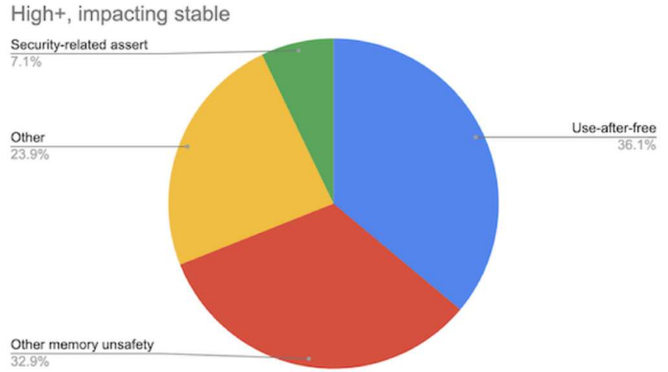
From Trends, challenge, and shifts in software vulnerability mitigation

Microsoft safety issues:
~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues.

The Chromium project finds that around 70% of our serious security bugs are memory safety problems. Our next major project is to prevent such bugs at source.

The problem

Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of *those* are use-after-free bugs.



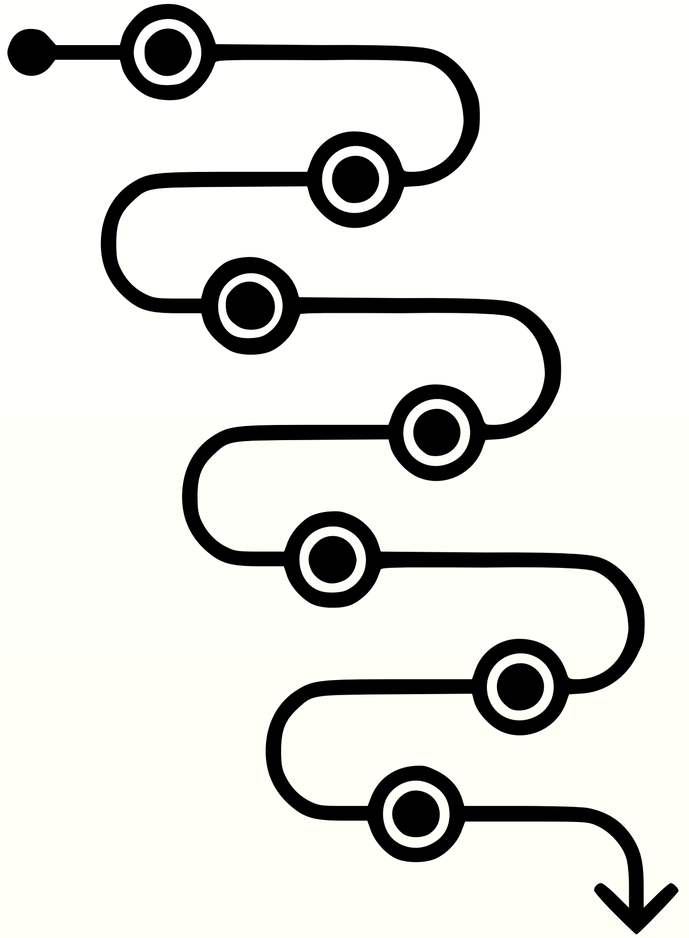
(Analysis based on 912 high or critical severity security bugs since 2015, affecting the Stable channel.)

From Memory safety in the Chromium Projects

Memory safety in the Chromium Projects

Page d'histoire

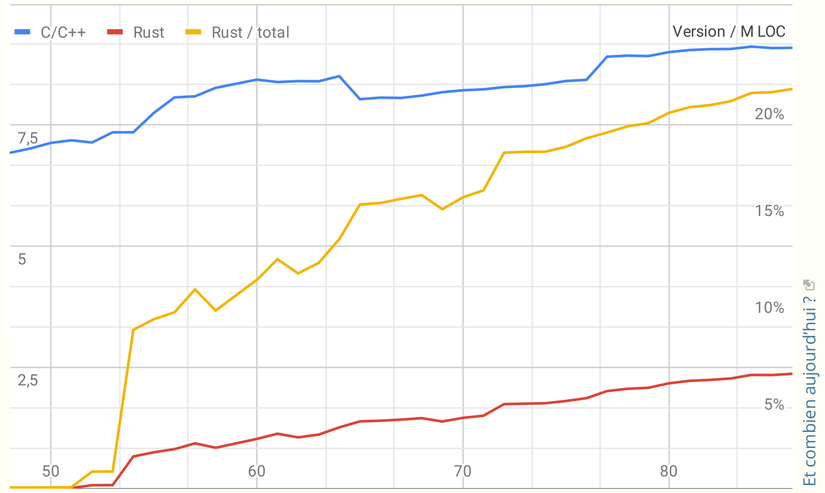
- En 2006, Graydon Hoare démarre le projet personnel d'un nouveau langage
- En 2009, Mozilla, son employeur commence à participer au projet avec les contributions (entre autres) de Brendan Eich*
- Le 7 juillet 2010, Mozilla annonce publiquement le projet
- En 2010, le premier compilateur écrit en OCaml est réécrit en Rust sur une base LLVM et en 2011 `rustc` s'auto-compile avec succès.
- En 2013, Samsung rejoint le projet
- Le 15 mai 2015, c'est la sortie de la première version stable, Rust 1.0 .
  Depuis 2015, une version stable est produite toutes les 6 semaines.
- L'édition 2018 (v1.31) marque la première version majeure révisée depuis l'édition 2015 (v1.0)
- En août 2020, en pleine crise de COVID19, Mozilla doit licencier une grosse partie de l'équipe Rust**.
- En février 2021, la fondation Rust est officiellement créée avec le soutien de AWS, Huawei, Google, Microsoft et Mozilla.
- L'édition 2021, sortie le 21 octobre 2021, est l'actuelle version majeure (LTS)



*: co-fondateur de Mozilla et créateur de Javascript
**: au cours d'un plan de licenciement de 250 employés soit 25% de ses effectifs mondiaux.

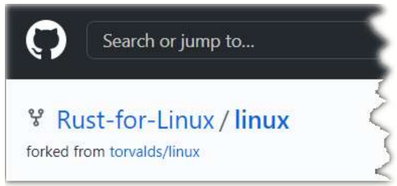
Rust en marche

- Dès 2016, Rust intègre le code de Firefox (48) en production.
  Le projet Oxidation continue de développer la part de Rust dans Firefox.
- Depuis 2016, Rust est le langage préféré dans le sondage StackOverflow
- Microsoft intègre Rust à sa stratégie de développement [1, 2, 3, 4]
- Amazon considère « Rust est un élément essentiel de notre stratégie à long terme »
- Google souhaite améliorer la sécurité de certains logiciels Open Source grâce à Rust [1, 2]
- En 2021, Rust est le langage le plus utilisé avec WebAssembly devant C++
- Au niveau de la recherche, Rust est aussi étudié en tant que réponse au problème de *Memory-Safety*.
  Le projet de recherche Microsoft Verona, inspiré de Rust, étudie l'extension du concept d'*ownership* étendu à un groupe d'objets (et non plus un seul).



Évolution de Rust dans le code de Firefox

Rust en marche

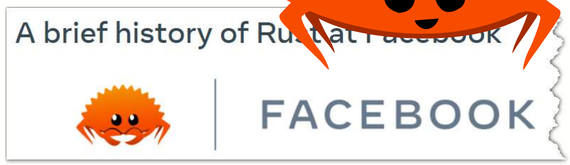


Le projet `Rust-for-Linux` ambitionne d'ajouter le support de Rust au noyau Linux

*" Overall, Rust is a language that has successfully leveraged **decades of experience from system programming languages as well as functional ones** [...]*
(source)

*" Rust is an intriguing language. It closely resembles C++ in many ways, hitting all the right notes [...]. [...] it also has the potential to **solve some of the most vexing issues that plague C++ projects** [...].*
(source)

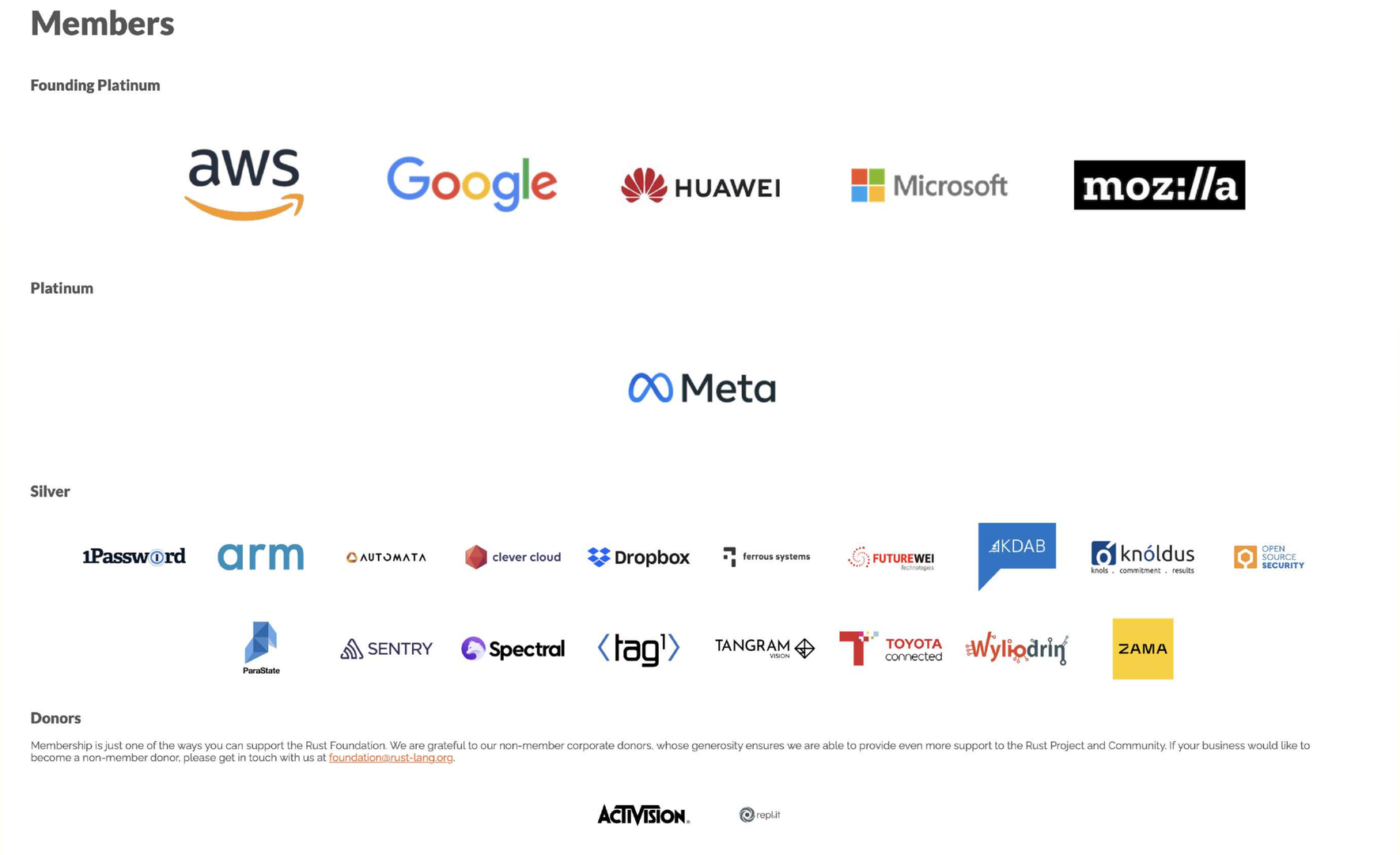
*" Rust provides memory **safety guarantees** [...] and runtime checks to ensure that memory accesses are valid. This safety is achieved while providing **equivalent performance to C and C++**.*
(source)



*" For developers, Rust offers the performance of older languages like C++ with a heavier focus on code safety. Today, there are **hundreds of developers** at Facebook writing **millions of lines of Rust** code.*
(source)

My name is Ferris

Une fondation solide

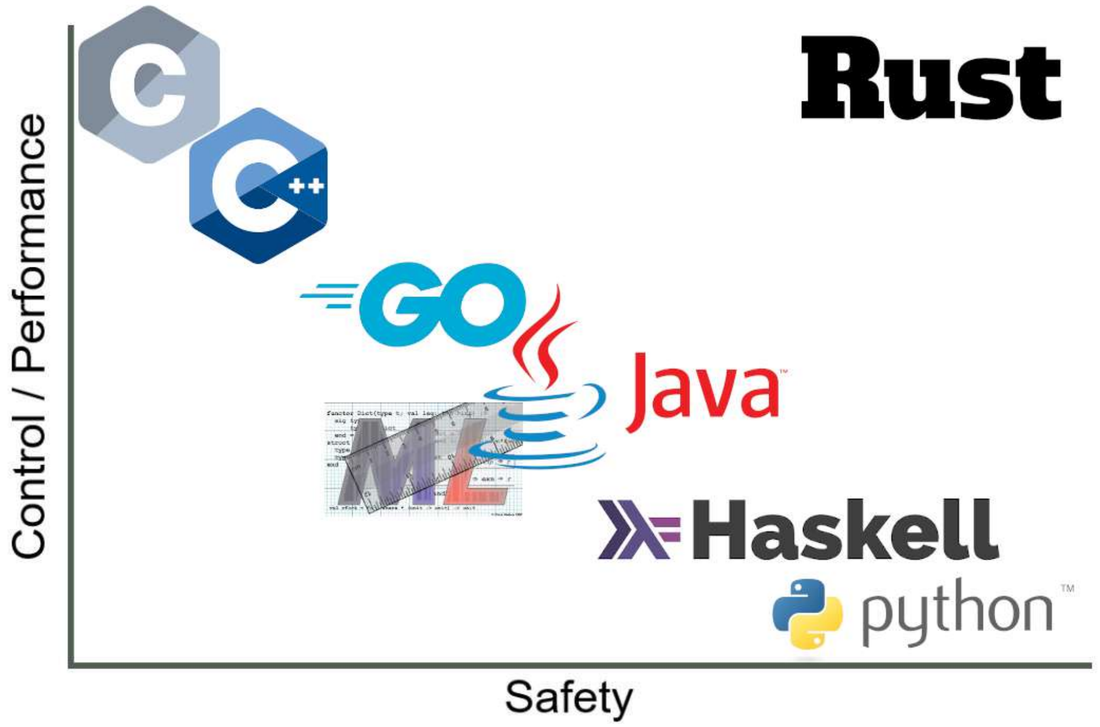


Depuis février 2021, la fondation Rust supporte le langage et son écosystème.

(et quelque 42 entreprises utilisant déjà Rust en production)

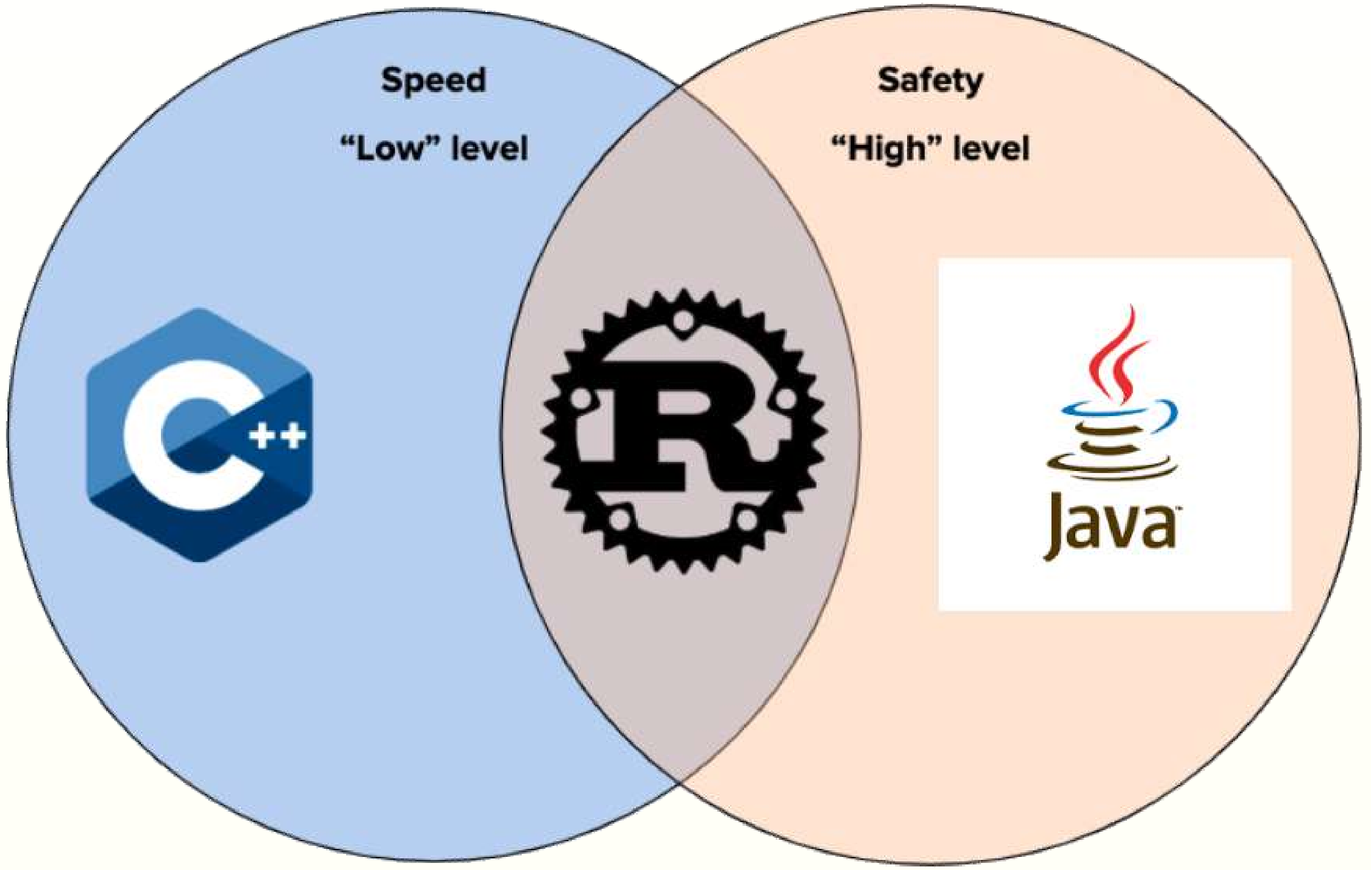
Le positionnement de Rust

Sécurité et Performances

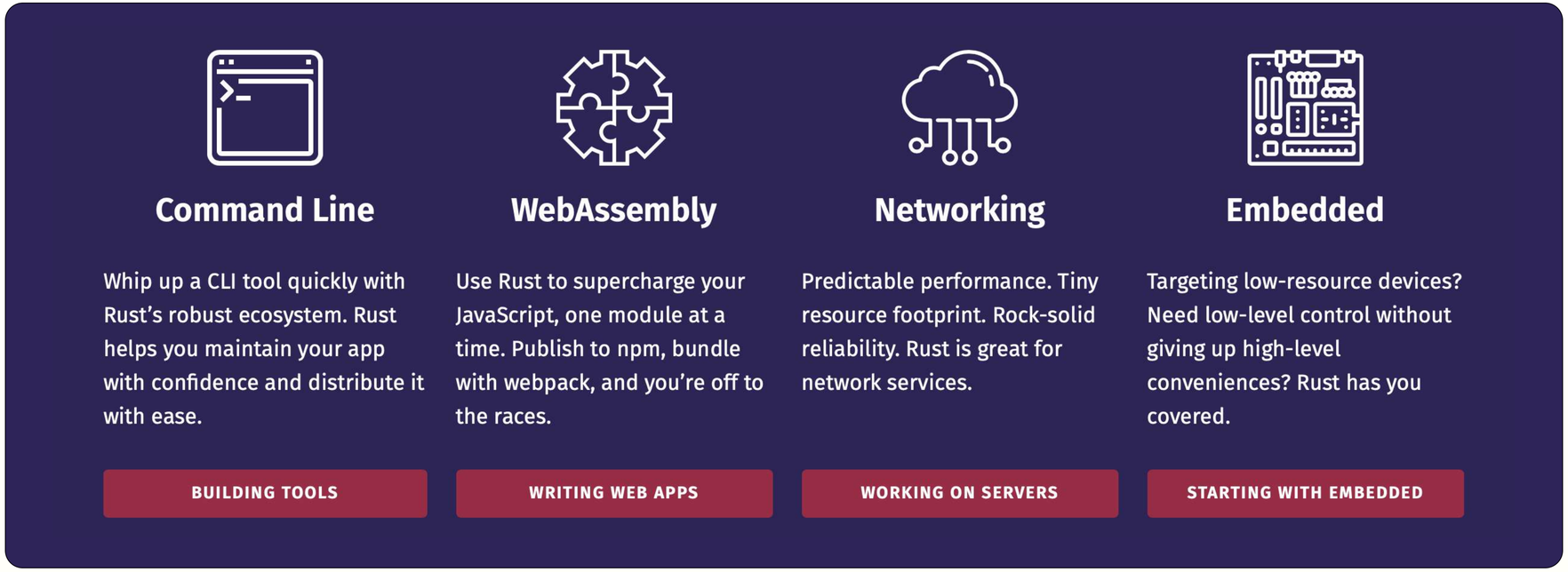


Le positionnement de Rust

Sécurité et Performances



Les cibles *officielles* de Rust



et le domaine scientifique commence à s'y intéresser...

- Why scientists are turning to Rust
- section `math` chez crates.io
- section `math` chez lib.rs

Les clefs de Rust

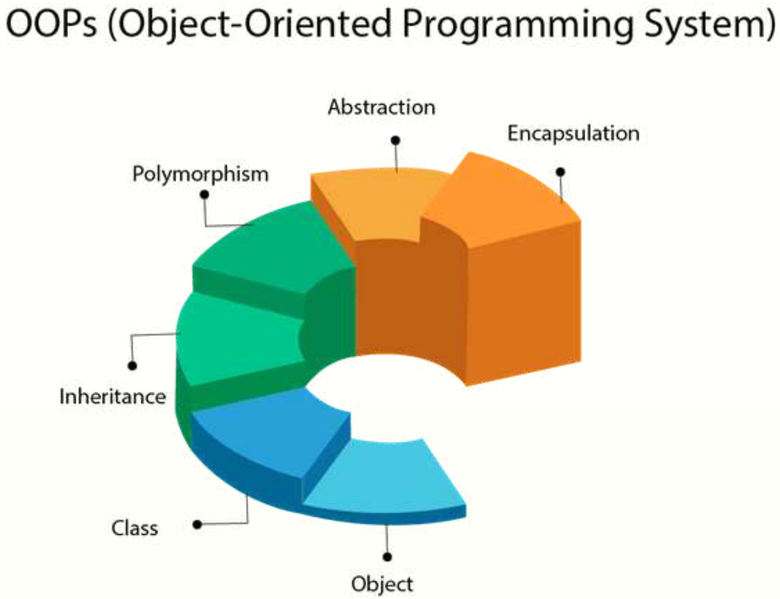
« Think different; don't think OO »

- D'après le *Gang of Four*, la programmation orientée objet se définit ainsi:

> *"Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations.*

  Alors **oui**, Rust est orienté objet (les données sont alors les `struct` et les `enum`, et les méthodes sont fournies par les `impl`)

- Si l'on considère l'encapsulation et en particulier la protection des données et des méthodes, alors **oui**, Rust est encore orienté objet (les données et attributs sont privées par défaut et peuvent être rendus accessibles par l'emploi explicite de `pub`).



« Think different; don't think OO »

- Si l'on considère l'héritage comme un élément indispensable de la POO au sens où un objet peut hériter des données et des comportements d'un parent (dans le but d'un partage de code), alors **non**, Rust n'est pas orienté objet.
  Rust ne permet pas l'héritage d'objets, mais peut proposer des approches alternatives:
  - Pour la réutilisation de code, il existe les `trait`s qui sont telles des interfaces pouvant porter un comportement par défaut (qui est ainsi la partie que l'on réutilise).
  - Pour l'emploi de l'héritage au niveau du système de type (*aka* le polymorphisme), tel d'un type enfant peut être utilisé à la place d'un type parent, Rust propose une autre forme de polymorphisme.
    Rust propose un polymorphisme basé sur la généricité (*bounded parametric polymorphism*) pouvant s'apparenter à un mélange de `template` et de `concept`(C++20) (plus strictes encore basé sur les `trait`s).

Certains emplois de l'héritage
peuvent introduire des problèmes.

code/cpp/src/liskov_violation.cpp

```cpp
//#region [Collapse all]

class Rectangle {};

class Square : public Rectangle {};

int main() {
  std::vector<Rectangle> v;
  v.push_back(Rectangle{2, 5});
  v.push_back(Square{3});

  for (auto &&x : v) {
    std::cout << x.area() << "\n";
  }
}
```

Autour du principe de Liskov (C++)

```java
String [] strings = new String [10];
Object [] objects = strings;  // alias String [] as Object []
objects[0] = new Date(); // Runtime ArrayStoreException!
```

Autour d'un problème de Variance (Java)
[exemple transposé de Wikipedia]

Rust, c'est

- De la sécurité
- De la performance
- Un écosystème avec de nombreux outils
- Le support de sponsors
- Une communauté

L'écosystème pour débuter

Rust *online*

Session du Rust Play Ground

Pour ceux voulant tester sans rien installer, il y a le

Rust Play Ground

On peut l'utiliser pour:

- tester des bouts de code (mono-fichier)
- tester différentes versions du compilateur (entre `stable`, `beta` et `nightly`)
- partager des *snippets* / *gists* de code
- vérifier votre code avec `clippy` ou d'autres outils (développement des macros, interpréteur d'analyse de code)
- analyser le code générer (différents niveaux dont ceux de LLVM)
  Un peu comme https://compiler-explorer.com

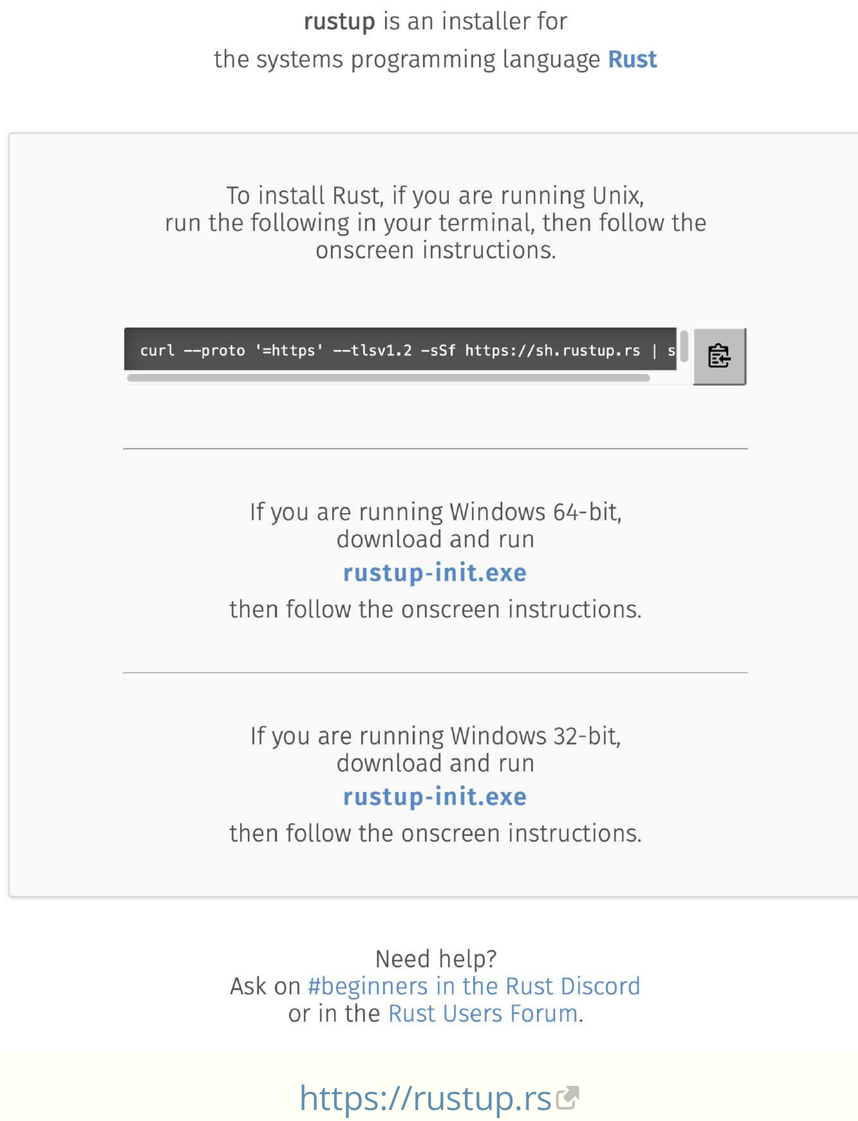
L'installation locale avec `rustup`

`rustup` c'est

- un site officiel https://rustup.rs qui permet en une commande d'installer un environnement **Rust** complet et prêt à l'emploi.
- la commande de mise à jour de tout votre environnement de développement

`rustup` s'occupe ainsi de tout installer

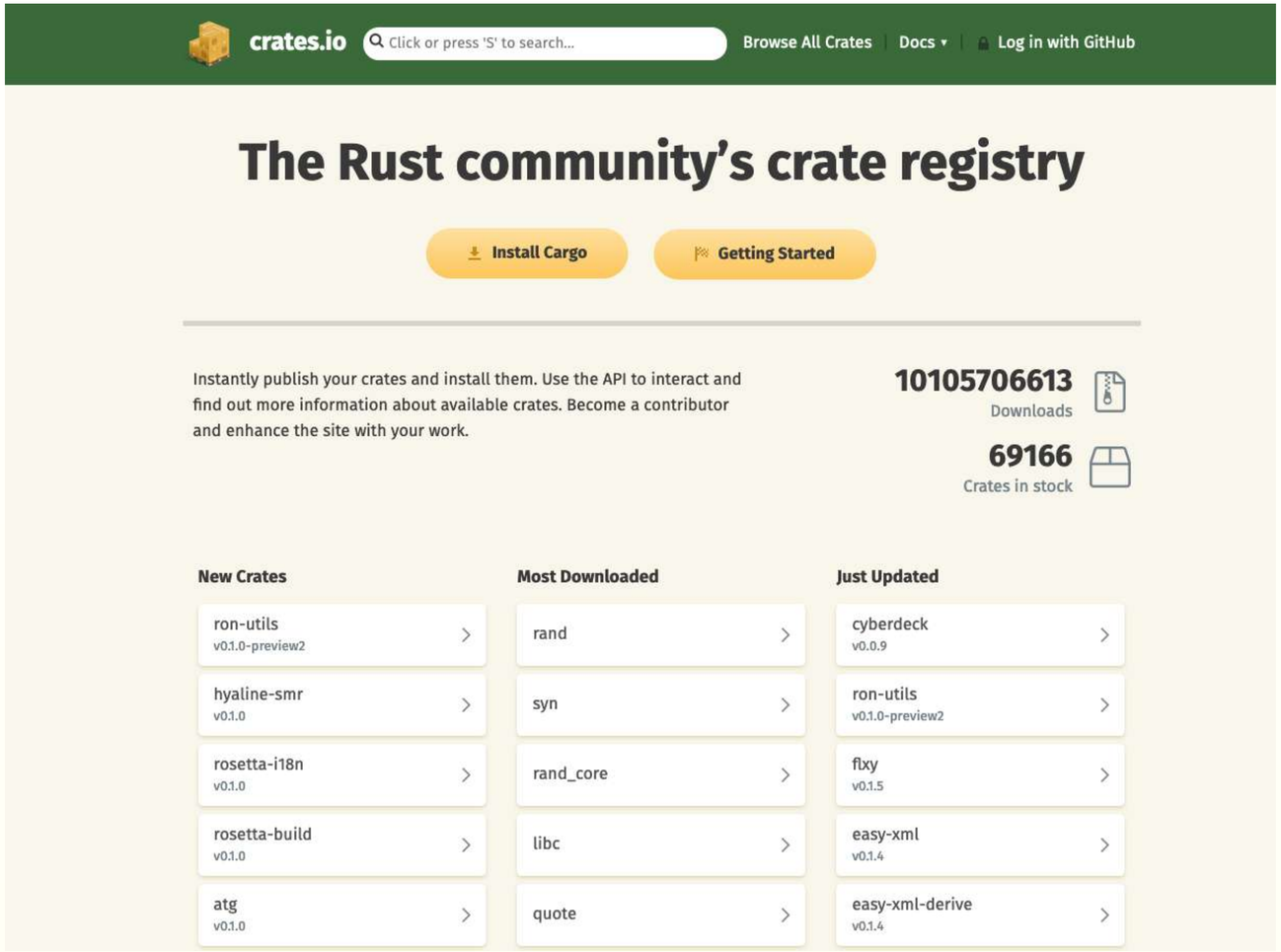
```
info: profile set to 'default'
info: default host triple is x86_64-unknown-linux-gnu
info: syncing channel updates for 'stable-x86_64-unknown-linux-gnu'
info: latest update on 2021-03-25, rust version 1.51.0 (2fd73fabe 2021-03-23)
info: downloading component 'cargo'      # le gestionnaire de central
info: downloading component 'clippy'     # l'analyse de code
info: downloading component 'rust-docs'  # la doc embarquée (en local)
info: downloading component 'rust-std'   # la bibliothèque standard
info: downloading component 'rustc'      # le (cross-)compilateur
info: downloading component 'rustfmt'    # le formatteur de code
...
info: default toolchain set to 'stable-x86_64-unknown-linux-gnu'
```



https://rustup.rs

Des caisses à la pelle

L'univers Rust se structure autour de *packages* (nommés *crates* en Rust), principalement centralisé sur https://crates.io



Choisissez votre environnement de développement

Rust adhère au Language Server Protocol
ce qui facilite beaucoup son intégration dans votre IDE préféré

First-class editor support

There's a Rust integration available for your editor of choice. Or you can build your own using the Rust Language Server.

VS Code	**Sublime Text 3**	**Atom**	**IntelliJ IDEA**
Eclipse	**Vim**	**Emacs**	**Geany**

Cliquez sur votre éditeur pour installer votre plugins Rust

Personnellement, j'ai choisi CLion avec le plugin Rust de JetBrains.

Fabriquer sans frontière

La cross-compilation devient (presque) sans effort.

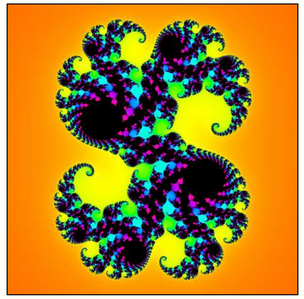
```
user@linux# rustup target add x86_64-pc-windows-gnu
info: downloading component 'rust-std' for 'x86_64-pc-windows-gnu'
info: installing component 'rust-std' for 'x86_64-pc-windows-gnu'
info: using up to 500.0 MiB of RAM to unpack components
user@linux# rustup toolchain install stable-x86_64-pc-windows-gnu
warning: toolchain 'stable-x86_64-pc-windows-gnu' may not be able to run on…
warning: If you meant to build software to target that platform, perhaps tr…
info: syncing channel updates for 'stable-x86_64-pc-windows-gnu'
info: latest update on 2021-07-29, rust version 1.54.0 (a178d0322 2021-07-2…
info: downloading component 'cargo'
…
user@linux# apt install mingw-w64 # MinGW toolchain required for linker
user@linux# cargo build --target x86_64-pc-windows-gnu --example julia
   Compiling cfg-if v1.0.0
   Compiling lazy_static v1.4.0
   …
   Compiling rs v0.1.0 (/data/rs)
   Finished dev [unoptimized + debuginfo] target(s) in 16.35s
```

```
user@windows# julia.exe
```

code/rs/examples/julia.rs

```rust
//! An example of generating julia fractals.
extern crate image;
extern crate num_complex;

use image::{ImageBuffer, Rgb};
use num_complex::Complex;

const MAX_ITER: i32 = 110;
const C: Complex<f32> = num_complex::Complex::new(0.285, 0.013);
//const C: Complex<f32> = num_complex::Complex::new(-0.9, 0.27015);

#[allow(dead_code)]
fn get_continuous_color(iteration_count: i32, z: &Complex<f32>) -> Rgb<u8> {…}

#[allow(dead_code)]
fn get_indexed_color(iteration_count: i32) -> Rgb<u8> {…}


fn main() {…}
```

Compilé sous Linux
puis directement exécuté sous Windows

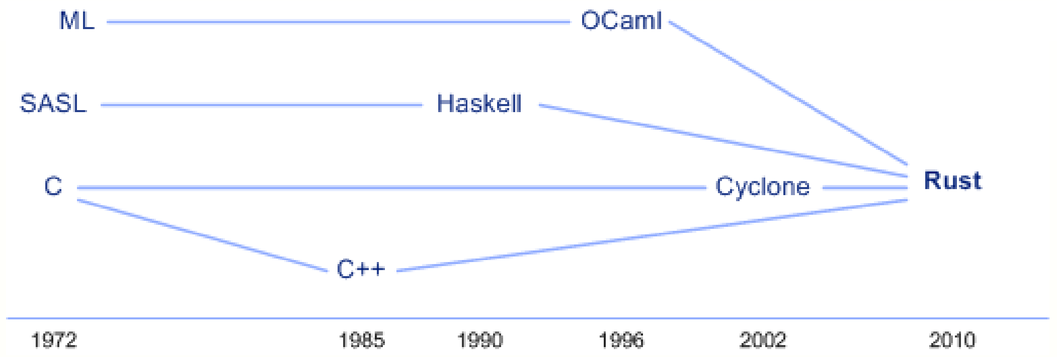


Les concepts clefs

La carte d'identité de Rust

Rust est un langage de programmation:

- **Impératif**: description des opérations en séquences d'instructions
  - **Structuré**: usage extensif de structures de contrôle de sélection, de répétition, de blocs de code et de sous-routines
- **Fonctionnel** (*impur*): description déclarative des calculs par l'évaluation de fonctions mathématiques, *i.e.* (a priori) sans mutation d'état ou effet de bord.
- **Générique**: polymorphisme par paramétrage de type
- **Concurrent** : traitement des calculs par morceaux et recouvrements de période de temps.



Influences historiques de Rust

NB: ne pas confondre *concurrence* et *parallélisme*;
Le parallélisme requière la présence de plusieurs processus de calcul sur la même période de temps.

La carte d'identité de Rust

Son système de typage est:

- **Statique**: le type est défini à la compilation
- **Fort***: le type capture des invariants qui permettent de garantir un usage conforme et évitant les conversions implicites.
- **Inféré**: le type est calculable par le compilateur.
  - basé sur le système de type de Hindley-Milner qui permet de déterminer un type en fonction de son utilisation [1]
- **Nominal**: les types sont définis par des déclarations explicites (et nommées)

*: beaucoup plus que C++; dans la veine de OCaml

code/rs/tests/hindler-milner.rs [slice 1:]

```rust
fn main() {
    let mut v = Vec::new(); // here, v is not yet fully typed
    let mut var;            // here, val is not yet fully typed

    // ... do things without using `var` and `v`

    v.push(3);  // now v is a vector of i32
    var = 3.14; // now var is a f64

    // v.push(3.14); // error: expected integer, found floating-point numb
    // var = 3; // error: expected floating-point number, found integer
}
```

Exemple de calcul de type en différé

Corollaire: la *rétro-propagation* dans le calcul de type imposera qu'une fonction doit expliciter son type de retour contrairement au `auto` de C++.

Un début de solution pour les problèmes de mémoire

code/cpp/src/memory-mistake.cpp

```cpp
//#region [Headers]
class Connection {};

void do_something(Connection &) {}

int main(int argc, char *argv[]) {
  Connection conn;

  if (argc > 1) {
    conn.connect(argv[1]);
  } else {
    conn.connect("Default value"s);
  }

  std::cout << "connection before = " << conn << std::endl;

  try {
    do_something(conn);
  } catch (std::exception &e) {
    std::cout << "An exception has been thrown\n";
    conn.dispose(); // may be an illegal free
  }

  std::cout << "connection after = " << conn << std::endl; // CWE416 ?
  conn.dispose();                                          // double free ?
}
```

Code C++ qui semble fonctionner mais... avec combien de bugs ?
Qui et quand trouvera-t-on ces bugs ?

Rust gère les données suivant ces règles:

- Toute valeur est associée à une variable qui en est son **propriétaire**
- Il ne peut y avoir qu'un seul propriétaire à la fois pour une valeur
- Quand la variable propriétaire sort du *scope*, la valeur est relâchée (*dropped*)

C'est une gestion

- explicite et prédictible, sans *garbage collector*
- Bonus: le compilateur a une excellente vue de l'*aliasing* de données.

Ces règles sont garanties par le compilateur.
(et non au *runtime* comme cela est le plus souvent)

La fiabilité de son système de type et de gestion des données (*ownership*, *lifetime*...) a été prouvée en 2018.

C'est la fin de bon nombre des Memory Leak, Use After Free et double free

Ownership[1/6] de main en main

code/rs/tests/ownership.rs [slice 4:10]

```rust
    let numbers = vec![1, 2, 3, 4, 5];

    let other_numbers = numbers;

    println!("{:?}", other_numbers);
    // numbers is freed here
```

code/rs/tests/ownership_failures/implicit_move.rs [slice 2:8]

```rust
    let numbers = vec![1, 2, 3, 4, 5];

    let other_numbers = numbers;

    println!("{:?}", other_numbers);
    println!("{:?}", numbers); //~ error: borrow of moved value: `numbers`
```

```
error[E0382]: borrow of moved value: `numbers`
  --> tests/ownership_failures/implicit_move.rs:10:26
   |
5  |         let numbers = vec![1, 2, 3, 4, 5];
   |             ------- move occurs because `numbers` has type `Vec<i32>`, which does not implement the `Copy` trait
6  |
7  |         let other_numbers = numbers;
   |                             ------- value moved here
...
10 |         println!("{:?}", numbers); //~ error: borrow of moved value: `numbers`
   |                          ^^^^^^^ value borrowed here after move
```

Ownership[2/6] de main en main

code/rs/tests/ownership.rs [slice 15:29]

```rust
        let numbers = vec![1, 2, 3, 4, 5];
        println!("{:?}", numbers);

        // Move ownership to other_numbers
        let other_numbers = numbers;
        println!("{:?}", other_numbers);

        // Now we cannot access numbers anymore because value was moved.
        // println!("{:?}", numbers); // error: does not COMPILE

        // Make a (deep) copy -> no move of ownership
        let cloned_numbers = other_numbers.clone();
        println!("clone = {:?}, source = {:?}", cloned_numbers, other_numbers);
        // Free numbers AND other_numbers vectors
```

Et souvent des erreurs très claires

En Rust, les données ne sont pas copiables ou même clonables par défaut.

- Pour permettre la copie, il faut mettre en place le mécanisme de *clone* et éventuellement celui de *copy* qui peut permettre une copie avec l'affectation sans appel explicite de `clone()`.
- Seules les données *triviales* implémentent par défaut ces traits de copie.

Ownership[3/6] de main en main

code/rs/tests/ownership.rs [slice 33:55]

```rust
fn move_and_functions() {
    let numbers = vec![1, 2, 3, 4, 5];
    consume(numbers); // Gives ownership to `consume`

    let produced_numbers = produce(); // Takes ownership
    println!("{:?}", produced_numbers);
    // produced_numbers gets out of scope -> free memory
}

fn consume(numbers: Vec<i32>) {
    let sum: i32 = numbers.iter().sum();
    println!("The sum is {}", sum);
    // numbers gets out of scope -> free memory
}

fn produce() -> Vec<i32> {
    let mut numbers: Vec<i32> = Vec::new();
    for i in 0..4 {
        numbers.push(i);
    }
    numbers // Gives ownership to caller : NO COPY
}
```

Sauf pour les données copiables (implémentant `Copy`), l'affectation ou le passage en argument correspondent par défaut à un `std::move` de C++11 avec un contrôle par le compilateur de non-réutilisation de la variable *source*.

C'est aussi vrai pour les retours de fonctions: **jamais de copie inutile**.

Le comportement par défaut est (le plus souvent) celui le plus performant tout en étant sécurisé (par le compilateur).

Ownership[4/6] de main en main

code/rs/tests/ownership.rs [slice 59:80]

```rust
fn borrow_and_functions() {
    let mut numbers = vec![1, 2, 3, 4, 5];

    println!(
        "The sum is {}", // Passes reference,
        borrow(&numbers)
    ); // keeps ownership
    println!(
        "The sum is {}", // Mutable reference,
        borrow_and_mut(&mut numbers)
    ); // keeps ownership

    println!("{:?}", numbers);
}

fn borrow(numbers: &Vec<i32>) -> i32 {
    // numbers is READ-ONLY, cannot be mutated
    // numbers.push(42);  // error: does NOT COMPILE
    let sum: i32 = numbers.iter().sum();
    sum
}
```

Et souvent des erreurs très claires

Créé par brgfx - fr.freepik.com

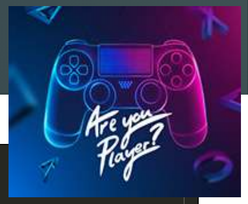
Même s'il n'y a **toujours** qu'un seul propriétaire, il est possible de prêter temporairement une donnée en lecture seule ou bien en lecture/écriture.

En Rust, l'emprunt (*borrow*)

- se fait sur toute la profondeur de la structure (si < 2021)
- soit (XOR) avec plusieurs accès en lecture seule simultanée
- soit (XOR) avec un unique accès en lecture/écriture

Ce comportement est toujours vérifié par le compilateur et en particulier en programmation concurrente !

Ownership[5/6] en toutes circonstances



code/cpp/src/in_async-mistake.cpp

```cpp
#include <chrono>
#include <future>
#include <iostream>
#include <numeric>
#include <vector>
using namespace std::chrono_literals;

auto add(std::vector<std::int32_t> &numbers) { numbers.push_back(42); }

auto sum(std::vector<std::int32_t> &numbers) -> std::int32_t {
  auto begin = std::begin(numbers);
  auto end = std::end(numbers);
  std::this_thread::sleep_for(200ms);
  return std::reduce(begin, end, 0, std::plus<>{});
}

int main() {
  std::vector<std::int32_t> numbers(100,1);
  auto sum_future = std::async(std::launch::async, sum, std::ref(numbers));
  std::this_thread::sleep_for(100ms);
  add(numbers);
  std::cout << "The sum is " << sum_future.get() << "\n";
}
```

Quel est résultat attendu ?

Suivant le langage, l'erreur corrompt la mémoire (ex: C++), est détectée au *runtime* (ex: C#) ou dès la compilation (ex: Rust).

code/rs/tests/ownership_failures/in_async.rs [slice 3:]

```rust
use futures::executor::block_on;
use std::time::Duration;

async fn async_main() {
    let mut numbers = vec![1; 100];
    let sum_future = sum(&numbers);
    std::thread::sleep(Duration::from_millis(100));
    add(&mut numbers); //~ cannot borrow `numbers` as mutable
    println!("The sum is {}", sum_future.await);
}

fn add(numbers: &mut Vec<i32>) {
    numbers.push(42);
}

async fn sum(numbers: &Vec<i32>) -> i32 {
    let iter = numbers.iter();
    std::thread::sleep(Duration::from_millis(200));
    iter.sum()
}

fn main() {
    block_on(async_main());
}
```

```
error[E0502]: cannot borrow `numbers` as mutable because it is also borrowed as immutable
  --> tests/ownership_failures/in_async.rs:11:9
   |
9  |     let sum_future = sum(&numbers);
   |                      -------- immutable borrow occurs here
10 |     std::thread::sleep(Duration::from_millis(100));
11 |     add(&mut numbers); //~ cannot borrow `numbers` as mutable
   |         ^^^^^^^^^^^^ mutable borrow occurs here
12 |     println!("The sum is {}", sum_future.await);
   |                               ---------- immutable borrow later used here
```

La sécurité du code avant tout

- ➕ Des règles claires
- ➕ Garanties par le compilateur
- ➕ Moins d'erreurs à l'exécution
- ➕ Moins de vulnérabilités potentielles
- ➖ Courbe d'apprentissage plus raide
- ➖ Il peut être parfois *nécessaire* passer dans le monde `unsafe` pour certaines opérations *fines*. (très rare sauf si en lien avec un langage *unsafe*).

- ➕ Potentiellement quelques optimisations en plus grâce à la gestion de l'*aliasing*

Tout est expression

Quelles différences entre ces deux séries d'échantillons ?

code/rs/tests/expression.rs [slice 18:20]

```rust
        let x = { 1; };
        println!("type of {{ 1; }} is {}", x.type_name());
```

code/rs/tests/expression.rs [slice 12:14]

```rust
        let x = { 1 };
        println!("type of {{ 1 }} is {}", x.type_name());
```

code/rs/tests/expression.rs [slice 32:34]

```rust
        let y = if test { "Hello"; } else { "World"; };
        println!("type of y is {}", y.type_name());
```

code/rs/tests/expression.rs [slice 25:27]

```rust
        let x = if test { "Hello" } else { "World" };
        println!("type of x is {}", x.type_name());
```

code/rs/tests/expression.rs [slice :-3]

```rust
pub trait AnyExt {…}

impl<T> AnyExt for T {…}

fn main() {…}
```

Tout bloc est une expression dont le retour est la dernière valeur sans `;` sinon le bloc est de type `()` (*unit*)

Pas besoin de *immediately invoked function expression*:

```
[&] { return x; }();
```

Les types utilisateurs[1/2]

Monas Hieroglyphica de John Dee

De quoi a-t-on besoin pour décrire *tous* les types de données ?

Pour répondre à cette question fondamentale,
Rust utilise une structure algébrique de types (ATD)

Un type somme

$$\sum$$

`enum { A, B, … }`

dont l'élément neutre `Void` est
`enum { }`
(ou `!` *aka never*)

$a + b = b + a$	:	`enum { A, B } =~ enum { B, A }`
$a \times b = b \times a$	:	`(A, B) =~ (B, A)`
$0 + x = x$	:	`enum { !, X } =~ enum { X }`
$0 \times x = 0$	:	`(!, X) =~ !`
$1 \times x = x$	:	`((), X) =~ X`

Et pour aller encore plus loin, il y a les GADTs [1, 2]
qui ne sont que partiellement disponibles dans Rust nightly.

Un type produit

$$\prod$$

`(A, B, … )`

dont l'élément neutre `Unit` est `()`

Les types utilisateurs[2/2]

Qu'a-t-on en **C++** moderne ?

- $\prod$: `struct` et `class` : même combat
- $\sum$: `enum` : uniquement des entiers
- $\sum$: `union` : n'est pas *type safe*
- $\sum$: `std::variant`[(C++17)] : pas facile à manipuler (besoin d'introspection le plus souvent par visiteurs)

Qu'a-t-on en **Rust** ?

- $\prod$: `struct`
- $\sum$: `enum`

Monas Hieroglyphica de John Dee

code/rs/tests/patterns.rs [slice 2:6]

```rust
enum Tree {
    Empty,
    Node { left: Box<Tree>, right: Box<Tree> },
}
```

Version plus complète d'un arbre avec un algorithme dans `code/rs/tests/tree.rs`

code/rs/tests/custom_types.rs [slice 5:23]

```rust
struct Person(&'static str);

struct Birthday {
    person: Person,
    date: Date<Utc>,
}

let _epoch = Birthday { person: Person("Unix Time"), date: Utc.ymd(1970, 1, 1) };

struct CardType(&'static str);

enum PaymentMethod {
    Cash,
    Cheque { number: u32 },
    Card(CardType),
}

let _payment = PaymentMethod::Card(CardType("Visa"));
```

Exemples de types personnalisés

Pattern matching & cie

Le principe n'est pas qu'un simple `switch (v) { case motif: }`

C'est bien plus de la destructuration de données *à la structured binding* de C++17 et largement plus étendue.

Il est ainsi possible:

- De déstructurer à la construction :

```
let Struct { a, b } = f();
let (b, a) = (a, b);
```

- De vérifier une déstructuration:

```
if let Ok(v) = function() { /* do something with v */ }
```

- De confronter une valeur à des expressions complexes:

```
match e {
    a @ Enum::A => println!("A = {:?}", a),
    b @ Enum::B => println!("B = {:?}", b),
    c @ Enum::C(0) => println!("C(0) = {:?}", c),
    Enum::C(x) if x == 1 => println!("C(x=1) = {:?}", e),
    x @ _ => println!("Other = {:?}", x),
}
```

code/rs/tests/patterns.rs

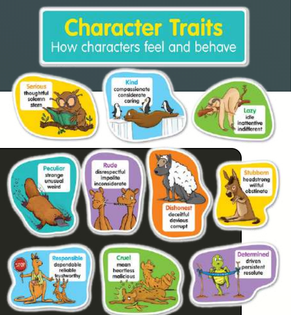
```
  1 #[test]
  2 fn trivial_match_enum() {
  3     enum Tree {
  4         Empty,
  5         Node { left: Box<Tree>, right: Box<Tree> },
  6     }
  7
  8     let tree = {
  9         Tree::Node {
 10             left: Box::new(Tree::Empty),
 11             right: Box::new(Tree::Node {
 12                 left: Box::new(Tree::Empty),
 13                 right: Box::new(Tree::Empty),
 14             })}
 15         };
 16
 17         match tree {
 18             Tree::Empty => { println!("Empty Tree"); }
 19             Tree::Node { left: _, right: _ } => { println!("Tree with nc
 20         }
 21     }
 22
 23     #[test]
 24     fn match_enum() {…}
 78
 79     #[test]
 80     fn match_struct() {…}
115
116     #[test]
117     fn pattern_multiple() {…}
136
137     #[test]
138     fn math_rules() {…}
152
153     #[test]
154     fn more_patterns() {…}
158
```

Ce n'est pas impossible à faire en C++17, mais cela demande quelques efforts supplémentaires pour combiner `std::variant`(C++17) et son visiteur.

code/cpp/src/variant_visitor.cpp

```cpp
//#region [Declarations]

struct Empty;
struct Node;

using Tree = std::variant<Empty, Node>;

struct Empty {};
struct Node {
  std::unique_ptr<Tree> left;
  std::unique_ptr<Tree> right;
};

//#region [overload trick]

int main() {
  Tree tree1{Empty{}};
  Tree tree2{Node{std::make_unique<Tree>(Empty{}),
                  std::make_unique<Tree>(Node{std::make_unique<Tree>(Empty{}),
                                              std::make_unique<Tree>(Empty{})})}};
  std::visit(
      [](const auto &t) {
        using type = std::decay_t<decltype(t)>;
        // clang-format off
        if constexpr (std::is_same_v<type, Empty>) { std::cout << "Empty\n"; }
        if constexpr (std::is_same_v<type, Node>) { std::cout << "Node\n"; }
        // clang-format on
      },
      tree1);

  std::visit(overload{[](const Empty &) { std::cout << "Empty\n"; },
                      [](const Node &) { std::cout << "Node\n"; },
                      [](auto x) { /* default case (useless here) */ }},
             tree2);
}
```

Traits 1/3



Les traits sont tels des contrats ou des interfaces qu'un type (type de base, `structure` ou `enumération`) peut implémenter.

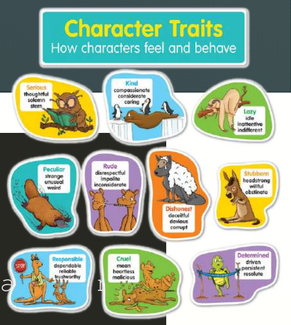
- Un trait porte autant sur des contraintes de fonctions requises que de types.
- Un trait peut avoir des comportements par défaut
- L'implémentation de traits se fait trait après trait après la définition du type

  (Il existe des règles limitant qui à la droit de définir un trait sur un type)

code/rs/tests/internal_monomorphisation.rs [slice :30]

```rust
trait Shape {
    fn area(&self) -> f64;
    fn name(&self) -> &'static str;
}

struct Square {
    side: f64,
}

struct Circle {
    radius: f64,
}

impl Shape for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
    fn name(&self) -> &'static str {
        "Square"
    }
}

impl Shape for Circle {
    fn area(&self) -> f64 {
        self.radius * self.radius * std::f64::consts::PI
    }
    fn name(&self) -> &'static str {
        "Circle"
    }
}
```

Traits 2/3



L'implémentation d'un trait est une contrainte sans aucun surcoût (ni en mémoire, ni en performance).
Il pourrait être (partiellement) comparé à un CRTP de C++.

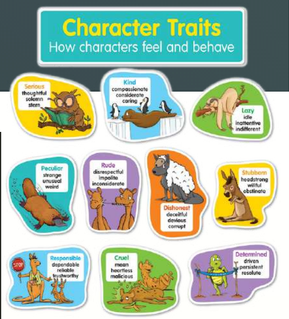
code/rs/tests/internal_monomorphisation.rs [slice :-3]

```rust
trait Shape {}

struct Square {}

struct Circle {}

impl Shape for Square {}

impl Shape for Circle {}

fn display_area(s: &impl Shape) -> String {}

fn main() {
    let v_impl: [Square; 2] = [Square { side: 2. }, Square { side: 1. }];
    println!("size_of(v_impl) = {}", std::mem::size_of_val(&v_impl));

    // Static dispatch
    let square = Square { side: 2. };
    println!("{}", display_area(&square));

    let circle = Circle { radius: 2. };
    println!("{}", display_area(&circle));
}
```

```
playground::display_area:
    ...
    movq        %rax, 64(%rsp)
    movq        %rdi, 192(%rsp)
    callq       <playground::Circle as playground::Sha
    movq        %rdx, 176(%rsp)
    movq        %rax, 168(%rsp)
    movq        48(%rsp), %rdi
    callq       <playground::Circle as playground::Shape>::area
    movsd       %xmm0, 184(%rsp)
    leaq        168(%rsp), %rax
    ...

playground::display_area:
    ...
    movq        %rax, 64(%rsp)
    movq        %rdi, 192(%rsp)
    callq       <playground::Square as playground::Shape>::name
    movq        %rdx, 176(%rsp)
    movq        %rax, 168(%rsp)
    movq        48(%rsp), %rdi
    callq       <playground::Square as playground::Shape>::area
    movsd       %xmm0, 184(%rsp)
    leaq        168(%rsp), %rax
    ...
```

Monomorphisation du code
avec spécialisation de la fonction pour chaque type.
(Rust Playground)

Traits 3/3



La même implémentation peut aussi être utilisée en dynamique au prix d'un dynamic dispatch.
Le surcoût mémoire se trouve alors dans des *fat pointers* (comme en Go et en D) et non dans l'objet (comme en C++).

code/rs/tests/internal_dynamic_dispatch.rs [slice :-3]

```rust
trait Shape {…}

struct Square {…}

struct Circle {…}

impl Shape for Square {…}

impl Shape for Circle {…}

fn display_area(s: &dyn Shape) -> String {…}

fn main() {
    let v_dyn: [&dyn Shape; 2] = [&Square { side: 2. }, &Circle { r
    println!("size_of(v_dyn) = {}", std::mem::size_of_val(&v_dyn));

    // Dynamic dispatch
    for &s in v_dyn.iter() {
        println!("{}", display_area(s));
    }
}
```

```
playground::display_area:
    ...
    movq        %rax, 72(%rsp)
    movq        %rdi, 200(%rsp)
    movq        %rdx, 208(%rsp)
    movq        32(%rdx), %rax
    callq       *%rax
    ...

playground::main:
    ...
    callq       core::slice::<impl [T]>::iter
    ...
    movq        %rsi, 320(%rsp)
    movq        %rdx, 328(%rsp)
    leaq        272(%rsp), %rdi
    callq       playground::display_area
    ...
```

Dispatch dynamique avec une indirection à chaque appel de la fonction.
(Rust Playground)

Différents usages, même code.

```
<playground::Square as playground::Shape>::area:
        pushq   %rax
        movq    %rdi, (%rsp)
        movsd   (%rdi), %xmm0
        mulsd   (%rdi), %xmm0
        popq    %rax
        retq

<playground::Circle as playground::Shape>::area:
        pushq   %rax
        movq    %rdi, (%rsp)
        movsd   (%rdi), %xmm0
        mulsd   (%rdi), %xmm0
        movsd   .LCPI58_0(%rip), %xmm1
        mulsd   %xmm1, %xmm0
        popq    %rax
        retq
```

Monomorphisation du code
avec spécialisation de la fonction pour chaque type.
(Rust Playground)

```
<playground::Square as playground::Shape>::area:
        pushq   %rax
        movq    %rdi, (%rsp)
        movsd   (%rdi), %xmm0
        mulsd   (%rdi), %xmm0
        popq    %rax
        retq

<playground::Circle as playground::Shape>::area:
        pushq   %rax
        movq    %rdi, (%rsp)
        movsd   (%rdi), %xmm0
        mulsd   (%rdi), %xmm0
        movsd   .LCPI70_0(%rip), %xmm1
        mulsd   %xmm1, %xmm0
        popq    %rax
        retq
```

Dispatch dynamique avec une indirection
à chaque appel de la fonction.
(Rust Playground)

Macros! 1/2

Les macros sont un puissant 💪 mécanisme qui permet de créer/transformer du code pendant la compilation.

- Elles sont elles-mêmes écrites en Rust
  (et accompagné de mécanismes d'introspection de l'AST)
  Elles sont construites dans une phase préalable de la compilation.
- Elles sont reconnaissables
  - par le symbole **!** qui suit leur nom
    ⚠ Ce ne sont pas des fonctions
  - par un bloc `#[macro(parameters)]` portant sur une entité
- Elles permettent:
  - Des simplifications d'écriture,
    d'éviter des répétitions (pour rester DRY);
  - De la génération de code; ex: décorateur de classes
  - De définir des DSL
  - Des contrôles ou des analyses de code (vérification de propriétés)

ℹ La commande `cargo expand` permet de visualiser le code généré par les macros.

```
println!("Hello {}", name);
assert_eq!(a,b);
let v = vec![2.0;3]

/* ..................................... */

#[route(GET, "/")]
fn index() { /* ... */ }

#[derive(Serialize, Deserialize)]
struct Struct {
    #[serde(rename = "Field")]
    field: i64,
}

/* ..................................... */

let sql = sql!(SELECT * FROM posts WHERE id=1);

write_html!(&mut out,
html[
    head[title["Macro demo"]]
    body[h1["Macros are the best!"]]
]);

/* ..................................... */

#[cfg(test)]
mod tests {
    #[test]
    fn test_function() {}
}
```

Macros!2/2

code/rs/tests/macro_simple_debug.rs [slice 26:-3]

```rust
// Declarative matching macros
macro_rules! debug {
    (msg:$msg:literal $($x:expr);+) => {{ // the order matters !
        println!("{:=^30}", $msg);
        $(
            println!("{:<10} = {}", std::stringify!($x),$x);
        )*
    }};
    ($($x:expr);+) => {{ // use ; separator (not ,)
        debug!(msg:"Debug Info" $($x);+);
    }};
}

fn main() {
    let var = 1;
    let something_else = "string";
    debug!(var);
    debug!(var; something_else);
    debug!(1+2; something_else);

    let str = MyStruct { field: 42 };
    debug!(msg:"Hello" var; str);

    println!("\nOk not moved : {}", str);
}

// Not copiable structure
#[derive(Debug)]
struct MyStruct {
    field: i8,
}

impl std::fmt::Display for MyStruct {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{:?}", &self)
    }
}
```

Et si vous souhaitez le niveau suivant:
code/rs/tests/macro_simple_debug.rs

code/rs/tests/macro_procedural.rs

```rust
extern crate procedural_macro; // cf produral_macro local c
use procedural_macro::make_answer;
use procedural_macro::CountFields;

make_answer!(); // create an answer1 function

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test1() {
        println!("Answer:{}", answer1());
    }

    #[derive(CountFields)]
    enum Any {
        A,
        _B,
        _C,
    }

    #[test]
    fn test2() {
        let a = Any::A {};

        println!("Field found:{}", a.count_field());
    }
}
```

```
Field found:3
Answer:42
```

Exemple d'utilisation d'une macro procédurale personnalisée

Les macros portent à la fois la généricité et l'adaptation ad-hoc de Rust.

Généricité[1/3]

La généricité en Rust serait-elle comme en C++ ?

code/cpp/src/generics_no_usage.cpp

```cpp
template <typename T> auto foo(T arg) {}

int main() {
  foo(1);
  foo(3.14);
  foo("hello");
}
```

code/rs/tests/generics_usage.rs [slice 0:7]

```rust
fn foo<T>(arg: T) {}

fn main() {
    foo(1);
    foo(3.14);
    foo("Hello");
}
```

Et si l'on va un peu plus loin...

code/cpp/src/generics_usage.cpp

```cpp
template <typename T> auto foo(T arg) -> T { return arg + 1; }

int main() {
  foo(1);
  foo(3.14);
  foo("hello");
}
```

code/rs/tests/failures/generics_bad_usage.rs [slice 0:-4]

```rust
fn foo<T>(arg: T) -> T { return arg + 1; }

fn main() {
    foo(1);
    foo(3.14);
    foo("Hello");
}
```

La généricité en C++ est une forme de *duck typing* statique.
Les contraintes ajoutées par les concepts de C++20 définissent un système de typage structurel (*i.e.* par des comportements).

La généricité en Rust doit **toujours** être bornée par un trait.
Le système de typage associé est alors nominal.

Généricité 2/3

Typage structurel VS nominal

code/cpp/src/generics_with_concepts.cpp [slice 12:-2]

```cpp
//#region [Collapse all]

template <typename T>
concept Stringable = requires(T a) {
  { a.stringify() } -> same_as<std::string>;
};

class Cat {
public:
  std::string stringify() { return "meow"; }
  void pet() {}
};

template <Stringable T> void f(T a) { a.pet(); }

int main() {
  f(Cat{});
}
```

code/rs/tests/failures/generics_with_unsatisfied_bounds.rs [slice 0:-3]

```rust
trait Stringable {
    fn stringify() -> String;
}

struct Cat {}

impl Cat {
    fn pet() {}
}

impl Stringable for Cat {
    fn stringify() -> String {
        "meow".to_string()
    }
}

fn f<T: Stringable>(a: T) {
    a.pet();  // error[E0599]: no method named `pet` found for type `T` in the cu
}

fn main() {
    f(Cat {});
}
```

Les contraintes de C++ sont optionnelles là où en Rust elles sont toujours obligatoires.

Il existe d'autres nuances telles que la forme purement additive du système de Rust (conjonction) et celui de C++ qui autorise aussi les disjonctions [1]

Généricité 3/3

code/rs/tests/generics_rustlings3.rs [slice :-5]

```rust
use std::fmt::{Display, Formatter};

pub struct ReportCard<T: Display> {
    pub grade: T,
    pub student_name: String,
    pub student_age: u8,
}

impl<T> ReportCard<T>
    where T: Display,
{
    pub fn print(&self) -> String {
        format!(
            "{} ({}) - achieved a grade of {}",
            &self.student_name, &self.student_age, &self.grade
        )
    }
}

impl<T> Display for ReportCard<T>
    where T: Display,
{
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}", self.print())?;
        Ok(())
    }
}

fn main() {}
```

code/rs/tests/tiny_vector_generics.rs [slice :17]

```rust
struct TinyVector<const SIZE: usize> {
    data: [f64; SIZE],
}

impl<const SIZE: usize> TinyVector<SIZE> {
    pub fn zero() -> Self {
        Self { data: [0f64; SIZE] }
    }
    pub fn constant(v: f64) -> Self {
        Self { data: [v; SIZE] }
    }
}

fn main() {
    let v1 = TinyVector::<4>::zero();
    let v2 = TinyVector::<4>::constant(1.);
}
```

Et il existe aussi une forme de spécialisation disponible dans les versions *unstable* du compilateur (*i.e.* en cours de stabilisation).
Ex: `code/rs/tests/unstable_specialization.rs`

Gestion des erreurs[1/2]

Sans sortir du chemin nominal

En Rust, dans l'esprit des langages fonctionnels, il n'y a pas d'exception.

- Les erreurs *récupérables* doivent être retournées en sortie de la fonction.
- Les erreurs *non récupérables* (aussi nommées *failures*) peuvent interrompre le code via `panic!(`*`message`*`)`.
  Notez qu'un `panic!` peut être généré pour lors d'une mauvaise gestion de cas limite:

```
unknown_result.unwrap();
```

L'usage veut que ce soit les types `Option<T>` et `Result<T, E>` qui soient recommandés

- `Option<T>` est relativement similaire à `std::optional<T>`(C++17) avec un cas `Some(`*`value`*`)` et un cas `None`.
- `Result<T, E>` se situe entre `std::variant<T, E>`(C++17) et `std::expected`(proposal) pour une valeur attendue de type `T` et une valeur d'erreur de type `E`.
  `Ok(`*`value`*`)` et `Err(`*`err`*`)` représentent respectivement le cas d'un succès de résultat et d'un retour d'erreur.

Gestion des erreurs 2/2

Sans sortir du chemin nominal

code/rs/tests/result.rs [slice :-4]

```rust
use std::{error, fmt};

#[derive(Debug)]
struct MyError {}

fn f1(n: Option<i64>) -> Result<i64, MyError> {}

fn f2(n: Option<i64>) -> Result<i64, MyError> {}

#[derive(Debug)]
enum MyError2 {}

impl fmt::Display for MyError2 {}

fn f3(n: Option<i64>) -> Result<i64, MyError2> {}

impl error::Error for MyError2 {}

// Runtime defined Error type
fn f4(n: Option<i64>) -> Result<i64, Box<dyn error::Error>> {}

fn show<E: std::fmt::Debug>(r: Result<i64, E>) {}

fn main() {}
```

Mais là où Rust simplifie radicalement la gestion des erreurs, c'est dans leur chainage *trivial*.

code/rs/tests/result_recover_error.rs [slice 22:30]

```rust
    fn read_and_validate(
        b: &mut dyn io::BufRead,
    ) -> Result<PositiveNonzeroInteger, Box<dyn error::Error>> {
        let mut line = String::new();
        b.read_line(&mut line)?; // Err here
        let num: i64 = line.trim().parse()?; // Err here
        Ok(PositiveNonzeroInteger::new(num)?) // Err here
    }
```

Remarquez les **?** un peu partout en fin de ligne.

`x?` est l'équivalent de

```rust
match x {
    Ok(v) => v,
    Err(e) => return Err(e),
}
```

On peut ainsi chaîner très simplement:

```rust
let x = obj.f()?.g()?.h()?;
```

Conversion

Les conversions en Rust doivent être soit **explicitées** soit codéesf avec le trait `From`.

- Il est ainsi possible de définir des conversions depuis n'importe quel type même un type de base.
- L'une des forces de cette forme de conversion est la synergie avec le système de type et le trait `Into`.

```
impl<T, U> Into<U> for T
    where U: From<T>,
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

`Into` est automatiquement implémenté que `From` l'est.

code/rs/tests/from_into.rs [slice :-4]

```rust
use std::convert::From;

struct Foo {
    str: &'static str,
    n: u16,
}

impl From<&'static str> for Foo {
    fn from(v: &'static str) -> Self {
        Foo { str: v, n: 0 }
    }
}

impl From<(&'static str, u16)> for Foo {…}

impl Foo {
    pub fn new<T>(v: T) -> Foo
        where T: Into<Foo> + std::fmt::Debug, // Into trait constraint
              Foo: From<T> {
        println!("{:?}", v);
        // let result = v.into();  // equivalent
        let result = Foo::from(v); //    forms
        result
    }
}

fn main() {
    let _f = Foo::from("Bob");
    let _f = Foo::from(("Mary", 16));
    let _f = Foo::new("Bob");
    let _f = Foo::new(("Mary", 16));
    let _f : Foo = ("Mary", 16).into();
}
```

Comparaisons avec C++

La surcharge

Soient des types `A`, `B`, `C` : comment définir une fonctionnalité `f` pour chacun de ces types ?

En **C++**:

- On peut définir `f` en la surchargeant pour chacun des types `A`, `B`, `C`
- On peut ajouter une méthode `f` à chacun des types `A`, `B`, `C` (éventuellement avec une interface pour mutualiser)
- On ne peut pas ajouter une méthode à un *type de base*.

En **Rust**:

- On ne peut pas surcharger les fonctions
- On peut implémenter un trait prédéfini pour n'importe quel type (même les types de base)

code/cpp/src/overload.cpp [slice 0:15]

```cpp
struct Interface {
  virtual void f() = 0;
};

struct A : Interface {
  void f() override {}
};

struct B : Interface  {
  void f() override {}
};

void f(A a) {}
void f(B b) {}
void f(double) {}
```

code/cpp/src/overload.cpp [slice 16:]

```cpp
int main() {
  A a;
  B b;

  a.f();
  f(a);

  b.f();
  f(b);

  f(3.14);
}
```

code/rs/tests/overload.rs [slice 2:21]

```rust
    trait Moon {
        fn moon(&self); // no default
    }

    struct A {}

    struct B {}

    impl Moon for A {
        fn moon(&self) { /* ... */ }
    }

    impl Moon for B {
        fn moon(&self) { /* ... */}
    }

    impl Moon for f64 {
        fn moon(&self) { /* ... */ }
    }
```

code/rs/tests/overload.rs [slice 23:36]

```rust
    fn main() {
        let a = A {};
        let b = B {};

        a.moon();
        A::moon(&a);

        b.moon();
        B::moon(&b);

        3.14.moon();
        f64::moon(&3.14);
    }
```

Comparaison avec C++ : le langage

C++

- ⊕ Grande proximité avec la machine: performances
- ⊕ *Zero-cost abstractions (si l'on fait bien attention)*
- ⊕ Expressivité : `auto`, opérateurs et surcharges...
- ⊕ Grande finesse d'attributs : `noexcept` (conditionnel), *rvalue reference*, capture des fonctions lambda...

- ⊖ Complexité (penser juste à l'initialisation d'une variable; *cf* Initialisation in modern C++ par Timur Doumler)
- ⊖ Peu de garde-fou; problème de sécurité à la charge du développeur
- ⊖ La qualité des messages d'erreur
- ⊖ Les comportements par défaut ne sont pas ceux *optimaux*: grosse *charge cognitive* pour le développeur

code/cpp/src/good_function.cpp [slice 4:]

```cpp
struct Object {};

[[nodiscard]] constexpr auto function(const Object &o) noexcept(noexcept(&Obje

int main() {}
```

Rust

- ⊕ Grande proximité avec la machine : performances
- ⊕ *Zero-cost abstractions*
- ⊕ Offre une grande protection dès la compilation
- ⊕ Les comportements par défaut sont transparents et *optimaux*
- ⊕ La qualité des messages d'erreur
  (et la clarté du code de `std::` par rapport à GCC)
- ⊖ Nommage parfois lourd
  (exprimant chaque variation de fonctionnalité)
- ⊖ Visibilité des mécanismes de borrowing / move (perte d'expressivité pour le numérique en particulier)
- ⊖ Abstractions minimales : complexité du langage;
  ex: différence des opérateurs sur types simples et complexes
- ⊖ Le typage très fort impose d'expliciter les conversions

Comparaison avec C++ : l'écosystème

C++

- plusieurs compilateurs : GCC, Clang, MSVC, NVidia/PGI, Intel…
- plusieurs chaines de compilation : Make, CMake, qmake, Meson, build2, Visual Studio…
- plusieurs distributions des *packages* : sources, conan, vcpkg, nuget, build2… (et la gestion du *Dependency Hell* est complexe)
- ÉNORMÉMENT de codes et de bibliothèques en C++
- Une grande et active communauté [Tendance Stackoverflow]
- de nombreux frameworks de tests
- De très nombreuses cibles supportées (déjà au moins 70 via GCC)

Rust

- 1 seul compilateur officiel: `rustc` (et en marge: mrustc et un front-end Rust pour GCC [github])
- Une chaîne de compilation officiel: `cargo` (intégrable aussi dans CMake)
- Gestion des packages à travers `cargo`: via dépôts git directs ou sites centraux tels que https://crates.io.
  (`cargo` propose une solution au *Dependency Hell*).
- Un adhésion forte à Rust et une belle croissance mais encore des manques (dont en HPC)
- un framework de tests intégré par défaut avec `cargo` (avec une doc compilable et testable)
- 19 cibles en Tier 1, 56 en Tier 2 et de nombreuses Tier 3 ([1])

Performances[1/3]

Image from Fuel Curve

Essentiellement Rust et C/C++ sont dans la même gamme def performances

- ⛔ Le compilateur de référence utilise un *backend* LLVM qui n'est pas toujours aussi optimisé que celui à base GCC.
  Certaines optimisations plus spécifiques n'ont pas encore été développées comme cela a pu être le cas pour C++.
- ⛔ Les chaînes de caractères en Rust sont encodées en UTF-8 ce qui pour certains usages *simples* peut parfois être *overkill*.
- ⛔ À cause de l'absence de conversion implicite, le type d'indexation de référence est le `usize` ce qui peut générer plus de pression sur les registres et la mémoire par rapport au conventionnel `int` de C/C++.
- ⛔ Comme il est facile d'embarquer des dépendances extérieures, il est aussi *trop* facile d'embarquer des bibliothèques non nécessaires ou redondantes (PS: jetez-y un œil avec `cargo tree`).
- 🟰 Les capacités d'*inlining* entre C++ et Rust sont comparables, tout l'effort étant renvoyé au compilateur.
- 🟰 La taille de l'exécutable *enfle* avec l'usage de *generics* tout comme avec les `template` en C++.
  La monomorphisation produit des algorithmes optimisés pour chaque cas d'usage.

Performances 2/3

Image from #automotocha9l

… avec certains potentiels non négligeables en plus:

- L'emploi plus intensif de la pile ou le suivi plus précis de l'*aliasing* en Rust permet aussi des optimisations po agressives (ex: moins d'indirections ou des types fondamentaux très optimisés: `option_optimization.rs` [playground])
- Rust se permet de réorganiser les champs de `struct` pour réaliser des optimisations.
  (sauf mention contraire du genre `#[repr(C)]` : `repr_struct.rs` [playground])
- Une réécriture de `lock(it); fill(it); call(it); find(it); view(it); code(it); jam(unlock(it));` ** à la sauce* fonctionnelle en `it.lock().fill().call().find().view().code().unlock().jam();` permet souvent de belles optimisations (comme ce que promettent les *ranges* de C++20).
- La *thread safety* est garantie par le compilateur autant dans votre code que dans celui des dépendances même si leurs auteurs n'y avaient pas prêté attention (sauf mention explicite de `unsafe`).
- Certaines bibliothèques telles que serde propose des fonctionnalités souvent inégalées en termes de performances.

PS: on peut toujours trouver des benchmarks (comme celui-ci) et en même temps il faut toujours garder un œil averti sur les techniques employées et le contexte d'utilisation.

Performances 3/3

Image from Fuel Curve

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

From https://thenewstack.io/which-programming-languages-use-the-least-electricity/
and http://greenlab.di.uminho.pt/wp-content/uploads/2017/10/sleFinal.pdf
2020 updated results : https://sites.google.com/view/energy-efficiency-languages/updated-functional-results-2020
Code source https://github.com/greensoftwarelab/Energy-Languages.

Normalized results for Energy, Time and Memory
A language efficiency comparison

Exemples

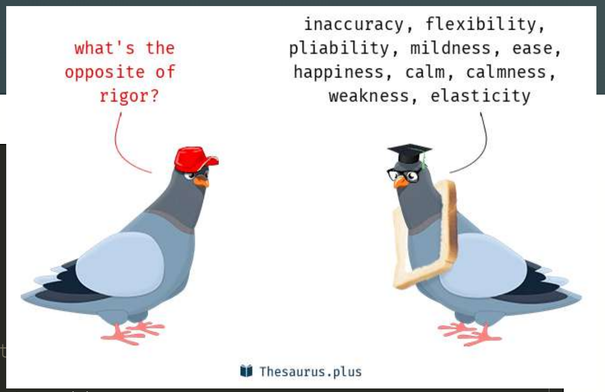
Rigueur ou rigidité ?

Soient deux nombres entiers x et y (disons de type `u8`).

Comment en faire la somme ?

$$x + y$$

Quels sont les risques et comment les prévenir ?



code/rs/tests/overflow_control.rs [slice 2:-3]

```rust
#![feature(bigint_helper_methods)]
#![feature(unchecked_math)]

fn main() {
    let x: u8 = 200;
    let y: u8 = 155;

    // x + y; // compile time error: this
    // x.add(y); // runtime error: attempt
    assert_eq!(x.checked_add(y), None);
    assert_eq!(x.overflowing_add(y), (99, true));
    assert_eq!(x.saturating_add(y), 255);
    assert_eq!(x.wrapping_add(y), 99);
    assert_eq!(unsafe { x.unchecked_add(y) }, 99); // unstable
    assert_eq!(x.carrying_add(y, true), (100, true)); // unstable
}
```

Le *style* fonctionnel

code/rs/benches/factorial.rs [slice 4:23]

```rust
fn imperative_factorial(mut n: u64) -> u64 {
    let mut r = 1;
    while n > 1 {
        r *= n;
        n-=1;
    }
    r
}

fn recursive_factorial(n: u64) -> u64 {
    match n {
        0 | 1 => 1,
        n => n * recursive_factorial(n - 1),
    }
}

fn fold_factorial(n: u64) -> u64 {
    (1..=n).fold(1, |acc, i| acc * i)
}
```

```
imperative_factorial    time:   [6.4344 ns 6.4391 ns 6.4450 ns]
recursive_factorial     time:   [6.4229 ns 6.4248 ns 6.4272 ns]
fold_factorial          time:   [4.0697 ns 4.0776 ns 4.0874 ns]
```

code/rs/tests/custom_iterator.rs [slice 94:117]

```rust
        // Implementation of this Unix command
        // </dev/urandom            |
        //     tr -dc 'a-fA-F0-9' |
        //     head -c15            |
        //     fold -w 3            |
        //     paste -sd-
        let result =
            RandomGenerator::new()
                .map(|x| (x % 256) as u8 as char)
                // .inspect(|x| println!("char: {}", x)) // show generated chars
                .filter(|x| x.is_ascii_hexdigit())
                .take(15)
                .subfold(3, String::new(), |acc, x| acc + &x.to_string())
                .fold(String::new(), |acc, s|
                    if acc.is_empty() {
                        s.to_string()
                    } else {
                        acc + "-" + &s.to_string()
                    })
            // .collect::<Vec<String>>().join("-") // Rust 1.56 stable with inter
            // .intersperse("-".into()).collect::<String>() // Rust nightly
            ;
        println!(">> {}", result);
```

```
>> F74-56A-7aA-6B8-6bB
```

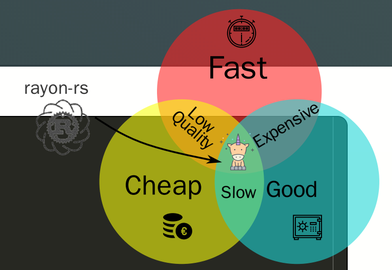
Le parallélisme de données pour 2x rien 1/2

Dans l'esprit des *ranges*(C++20) et de *std::execution::par*(C++17) de la STL parallèle de C++, rayon permet la mise en œuvre d'un parallélisme de données en quelques caractères : `par_`.

- Pratiquement sans rien faire
- Si une dépendance de données illicites est présente, elle sera détectée dès la compilation (par exemple un accumulateur fait *à la main*).
- Portable et *ThreadPool* paramétrable

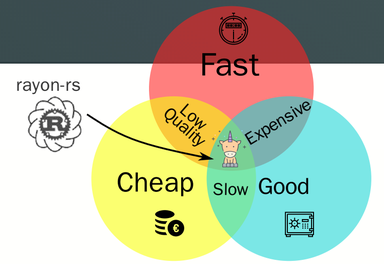
code/rs/tests/rayon.rs [slice 1:-2]

```rust
use rayon::prelude::*;

#[test]
fn test_stable_sort() {
    let mut v = [-5, 4, 1, -3, 2];
    v.par_sort();
    assert_eq!(v, [-5, -3, 1, 2, 4]);
}

#[test]
fn test_custom_unstable_sort() {
    let mut v = [-5i32, 4, 1, -3, 2];
    v.par_sort_unstable_by_key(|k| k.abs());
    assert_eq!(v, [1, 2, -3, 4, -5]);
}

#[test]
fn test_par_iter() {
    let sum = (1..=100)
        .into_par_iter() // simply parallel !
        .map(|n| n * n)
        .sum::<u32>();
    assert_eq!(sum, 338350);
}
```



Et aussi `code/rs/examples/quick_julia.rs`

Le parallélisme de données pour 2x rien 2/2



Le parallélisme de données, c'est simple et efficace comme `rayon`

code/rs/benches/sort.rs [slice 5:8]

```rust
    let mut rng = rand::thread_rng();
    let range = Uniform::new(0., 1.);
    let v: Vec<f64> = (0..1_000_000).map(|_| rng.sample(&range)).collect();
```

code/rs/benches/sort.rs [slice 11:13]

```rust
            let mut w = v.clone();
            w.sort_by(|a, b| a.partial_cmp(b).unwrap());
```

code/rs/benches/sort.rs [slice 24:26]

```rust
            let mut w = v.clone();
            black_box(w.par_sort_by(|a, b| a.partial_cmp(b).unwrap()));
```

```
sort                    time:   [88.847 ms 89.007 ms 89.173 ms]
par_sort                time:   [11.371 ms 11.426 ms 11.484 ms]
```

Doc et tests au cœur du code

code/rs/doclib/src/lib.rs [slice 20:-1]

```rust
/// The next function divides two numbers.
///
/// # Examples
///
/// ```
/// let result = doclib::div(10, 2);
/// assert_eq!(result, 5);
/// ```
/// # Arguments
///
/// * `a`, `b` - inputs
///
/// # Panics
///
/// The function panics if the second argument is zero.
///
/// ```rust,should_panic
/// // panics on division by zero
/// doclib::div(10, 0);
/// ```
pub fn div(a: i32, b: i32) -> i32 {
    if b == 0 {
        panic!("Divide-by-zero error");
    }

    a / b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test1() {
        assert_eq!(div(3, 2), 1);
    }

    #[test]
    #[should_panic]
    fn test2() { div(3, 0); }
}
```

Interopérabilité

Avec

grâce à

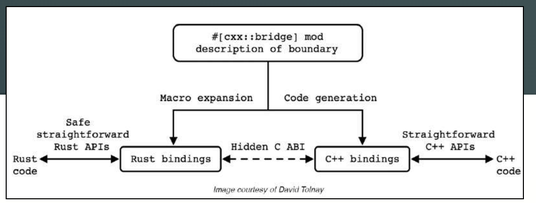
cxx

Avec

python

grâce à

PyO3



Ajouter une fonctionnalité sur `Iterator`

Objectifs de l'exemple:

- Ajouter un nouveau type itérable: `RandomGenerator`
- Ajouter une nouvelle opération du `Iterator`: `subfold`

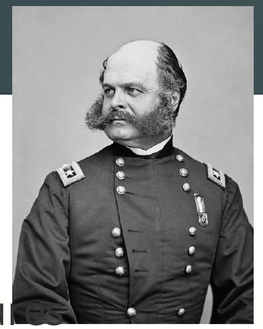
(À comparer avec les *Ranges* de C++20)

code/rs/tests/custom_iterator.rs [slice 100:113]

```rust
        let result =
            RandomGenerator::new()
                .map(|x| (x % 256) as u8 as char)
                // .inspect(|x| println!("char: {}", x)) // show generated chars
                .filter(|x| x.is_ascii_hexdigit())
                .take(15)
                .subfold(3, String::new(), |acc, x| acc + &x.to_string())
                .fold(String::new(), |acc, s|
                    if acc.is_empty() {
                        s.to_string()
                    } else {
                        acc + "-" + &s.to_string()
                    })
```

Sélection des meilleurs *crates*

Ma sélection


Ambrose Burnside

Outils

(pas forcément via crates.io)

- fd-find (`fd`) : un *turbo* find
- ripgrep (`rg`) : un *turbo* grep
- hyperfine : très bon outil de benchmark de commandes
- rustfmt (`cargo fmt`) : le formatteur standard de code
- clippy (`cargo clippy`) : l'équivalent de `clang-tidy` pour Rust.

Foreign Function Interface (FFI)

(via crates.io)

- cxx : binding vers et depuis C++≥11
- PyO3 : binding vers et depuis Python 3
- wasi / wasm-bindgen : l'entrée facile vers le WASM

Libs

(via crates.io)

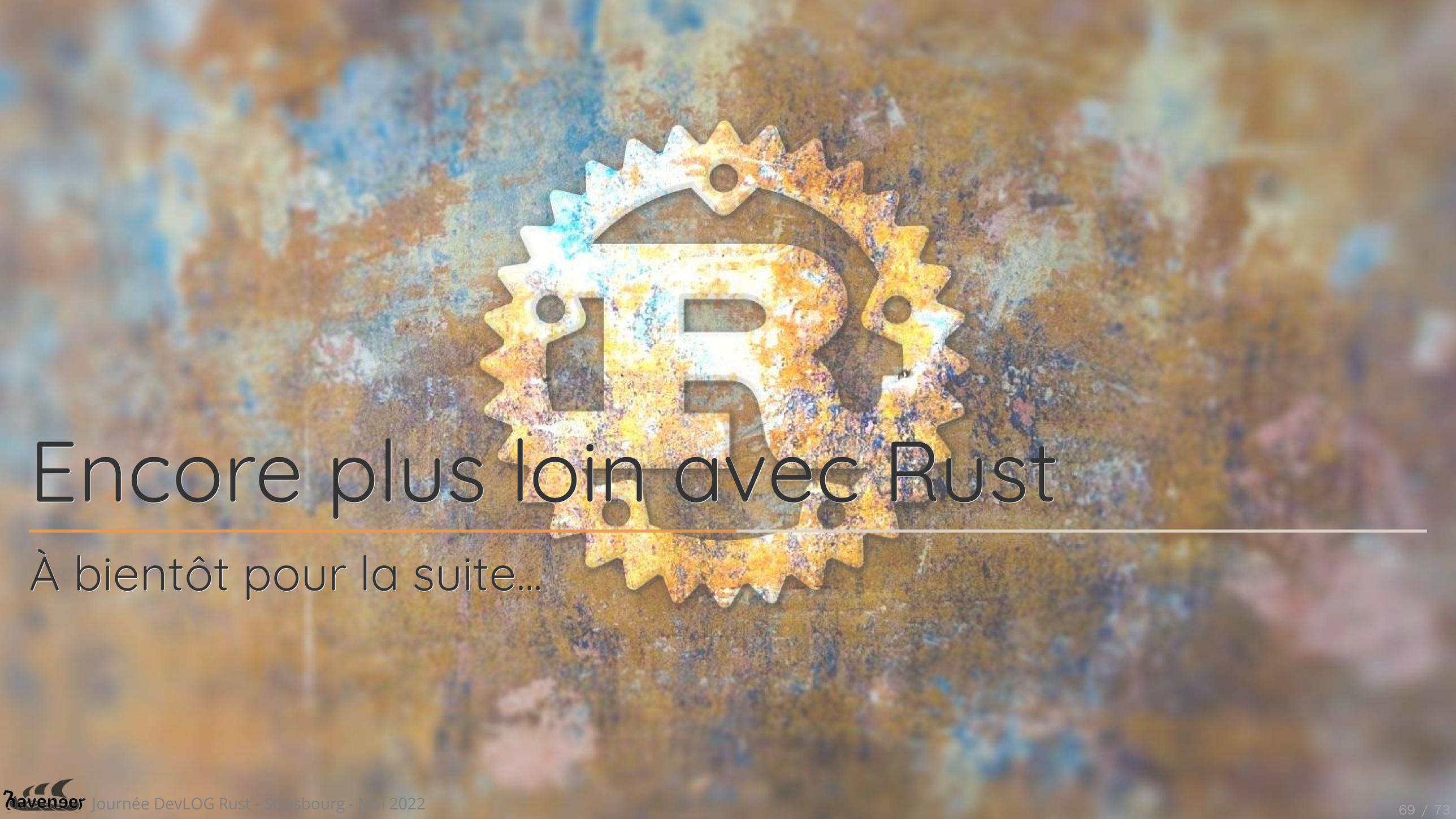
- serde : la **sér**ialisation / **dé**sérialisation de structures
- tokio : l'asynchronisme facile
- rayon : le parallélisme (de données) par vol de tâches
- crossbeam: outils pour la programmation concurrente
- clap : le parsing de la ligne de commandes
- rand : juste des nombres aléatoires (>100M downloads)
- criterion : micro-benchmarking *à la* Google Benchmark
- actix : un modèle d'acteurs performant (utilise tokio)
- log : des logs tout simplement
- nom : votre parseur pour données binaires ou textuelles
- itertools : vous reprendrez bien quelques algos en plus
- anyhow : outil pour définir un type d'erreur polymorphe
- thiserror : macro pour définir facilement le trait `std::error::Error`
- im : bibliothèque des structures immutables
- egui : un imgui à la sauce Rust

Quelques caisses pour le numérique

- Quelques pointeurs ici ou là
- ndarray: *n-dimensional container*
- nalgebra: algèbre linéaire pleine et creuse 🏦
- simba: algèbre SIMD 🏦 (en attendant std::simd)
- portable-simd: version de travail de std::simd 🔭
- num: collection de types numériques (big-int, complexe...)
- rsmpi: MPI bindings for Rust
- eigenvalues: *iterative algorithms for computing eigenvalues / eigenvectors*
- arrayfire: *high performance library for parallel computing (portable across CUDA, OpenCL and CPU devices)*
- Peroxide: *linear algebra, numerical analysis, statistics and machine learning tools with R, MATLAB, Python like macros*
- rustimization: *optimization library which includes L-BFGS-B and Conjugate Gradient algorithm*
- Rust-CUDA: CUDA for Rust
  (alternative to nvptx64-nvidia-cuda target [ABI])

Rust en calcul scientifique et HPC

- Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body (🏢2021 - Universidad Nacional de La Plata)
- Using Rust for Scientific Numerical applications (🧒💻2021 - Netherlands eScience Center)
- Why scientists are turning to Rust (🧒💻2020 - Nature)
- Rust programming language in the high-performance computing environment (🧒🎓2020 - ETHZ)
- Comparison of Multi-threading between C++ and Rust (OpenMP vs Rayon) (🧒🎓2019 - Carnegie Mellon University)
- Rust in HPC (🧒💻2018 - LANL)
- Evaluation of performance and productivity metrics of potential programming languages in the HPC environment (🧒🎓2015 - University of Hamburg)



Encore plus loin avec Rust

À bientôt pour la suite…

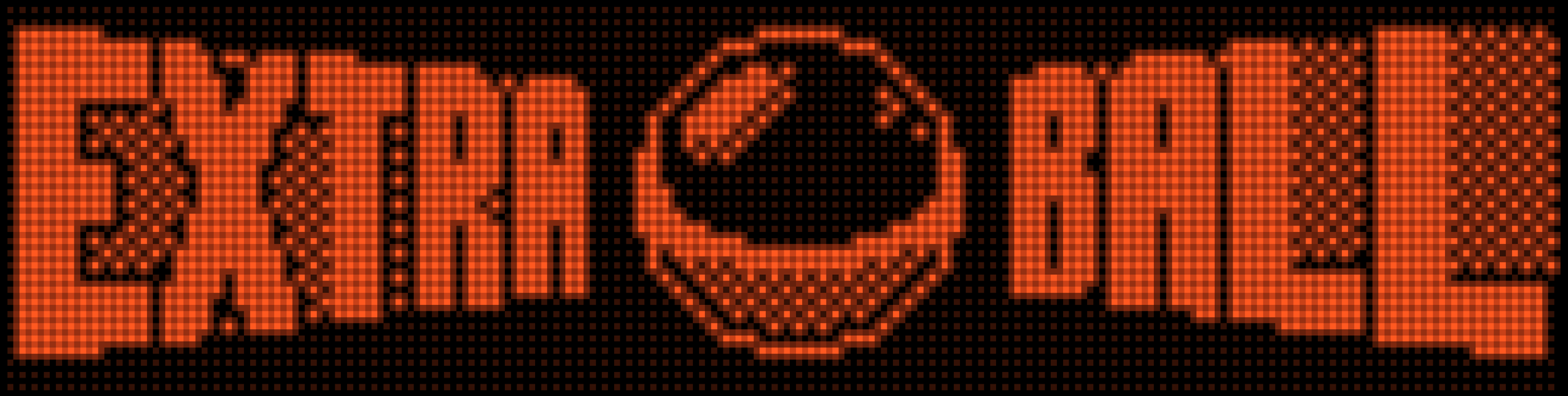
Non abordés

- Les fonctions `const` (comme `constexpr`(C++11)) [doc, demo folle]
- Le concept de *lifetime* du genre `&'static str`
- Les nombreux types de pointeurs
- Écrire du code `unsafe`
- La gestion de la visibilité (*crates*, `struct`, `mod`...)
- Les traits de type *markers* : `Copy`, `Send`, `Sized`, `Unpin`...
- La capture dans les lambdas
- Les types *Phantom*
- L'`async`hronisme

...


KEEP
CALM
AND
TRUST
IN RUST

S’il reste encore un peu de temps…



Pour finir

Finalement, plus ou moins *objet* que C++ ?

code/rs/tests/overload.rs [slice 2:5]

```rust
trait Moon {
    fn moon(&self); // no default behaviour
}
```

```rust
impl Moon for f64 {
    fn moon(&self) { /* ... */ }
}
```

```rust
    3.14.moon();
    f64::moon(&3.14);
```