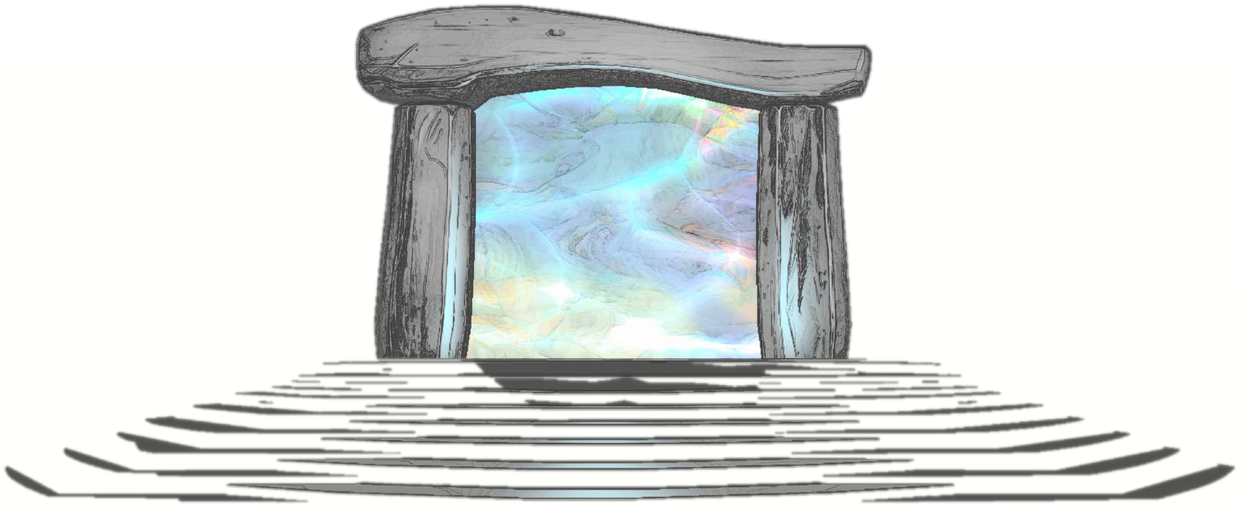


Café Calcul du 13 janvier 2022

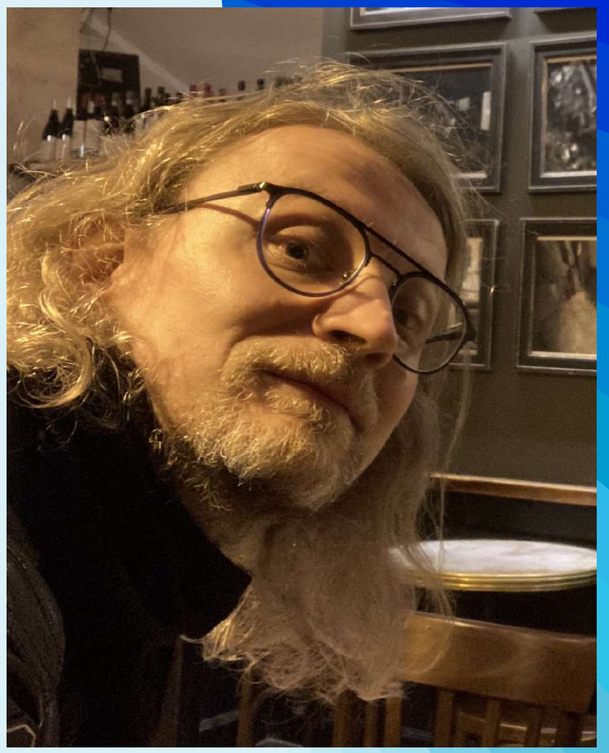
Découverte de **Rust**

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
- 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

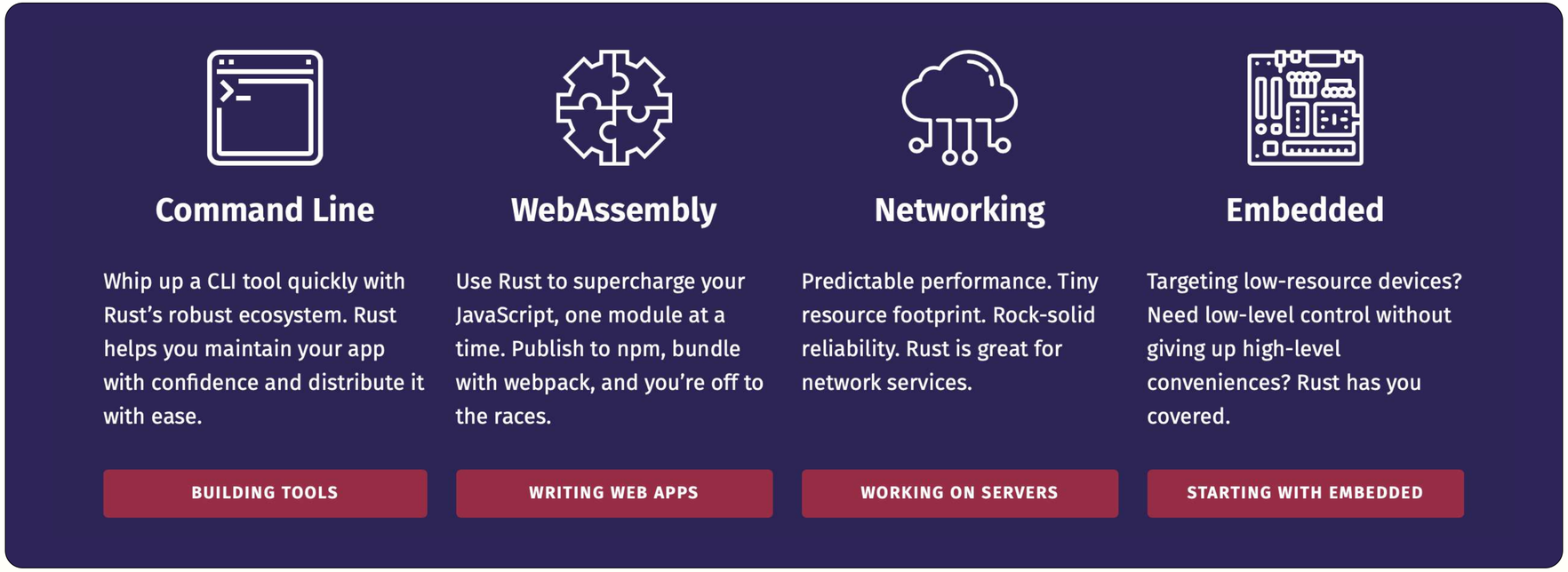
Sécurité **Vitesse** **Concurrence**

Rust

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

Performance **Fiabilité** **Productivité**

Les cibles *officielles* de Rust

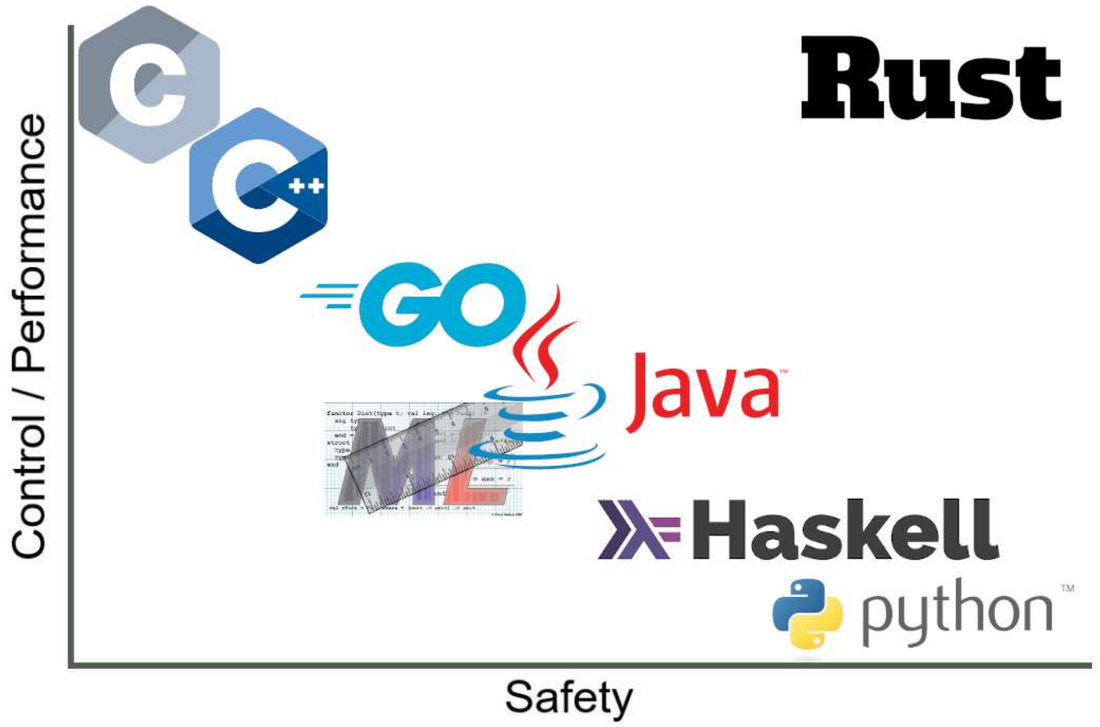


et le domaine scientifique commence à s'y intéresser...

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

Le positionnement de Rust

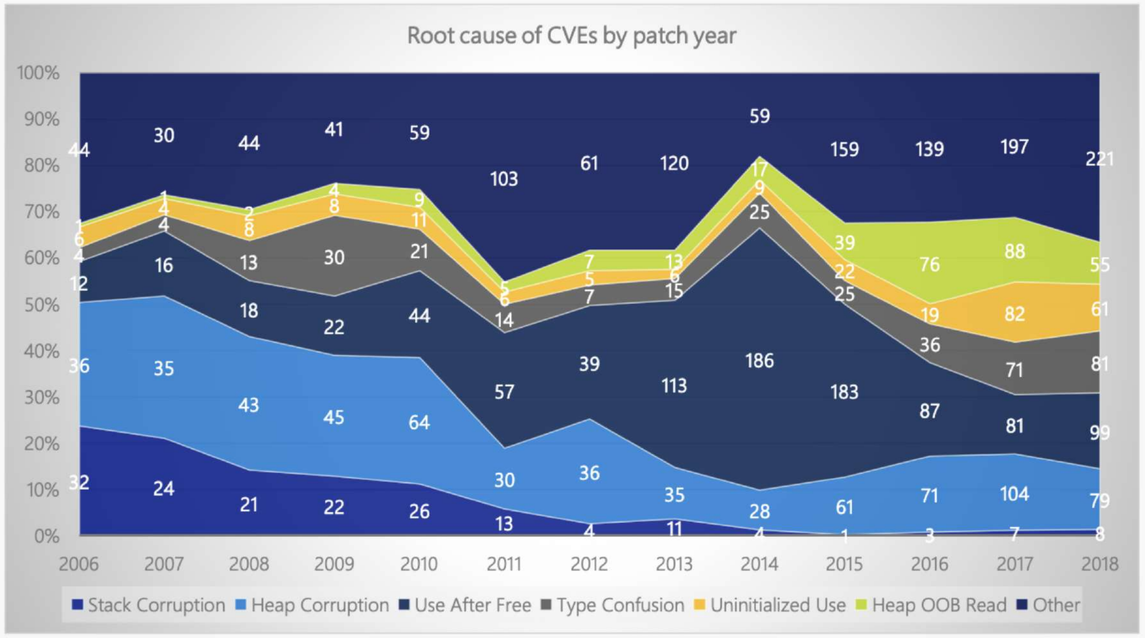
Sécurité et Performances



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.



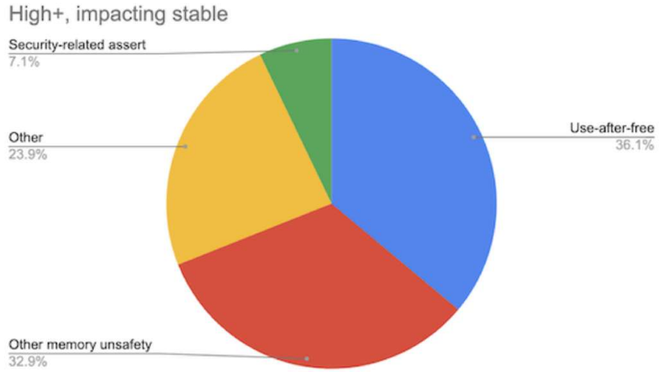
From Trends, challenge, and shifts in software vulnerability mitigation

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.)

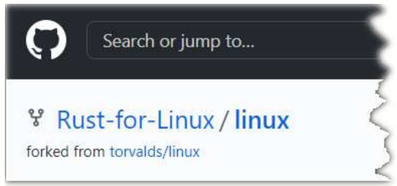
From Memory safety in the Chromium Projects

Memory safety in the Chromium Projects

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).

Rust en marche

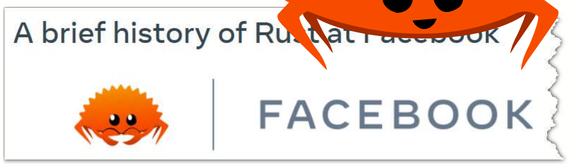


Le projet `Rust-for-Linux` ambitionne d'ajouter le support de Rust au noyau Linux

*" 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)

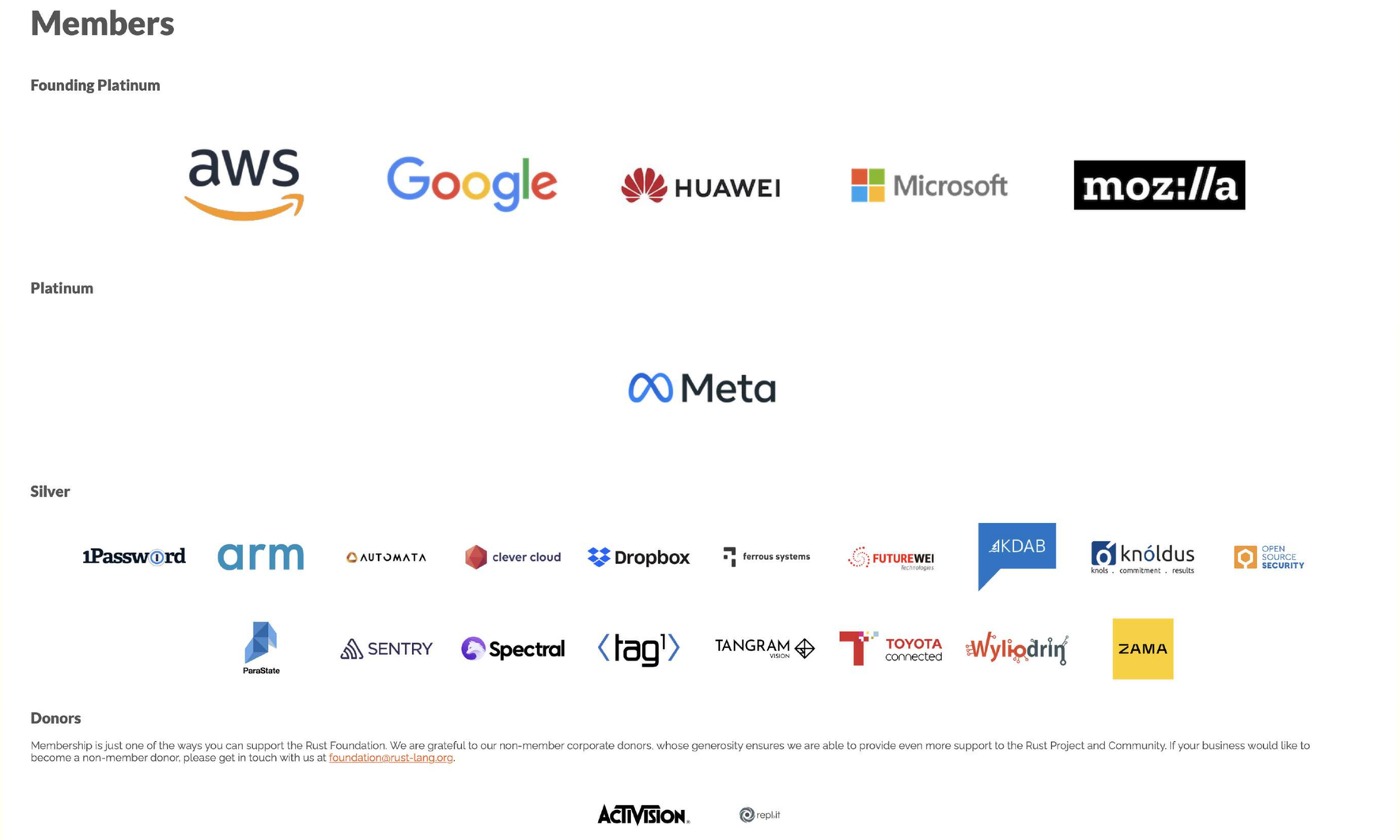
*" Rust provides memory **safety guarantees** [...] and runtime checks to ensure that memory accesses are valid. This safety is achieved while providing **equivalent performance to C and C++**.*
(source)



*" 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)

Rust, c'est

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

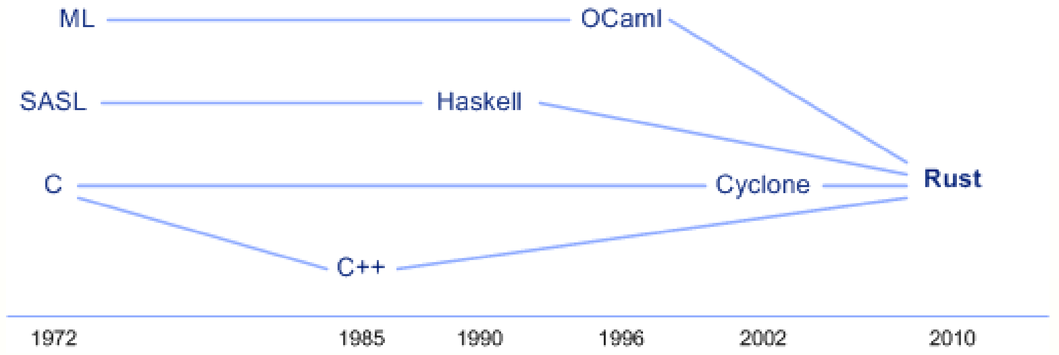
Les clefs du langage

La carte d'identité de Rust

Rust propose un modèle de programmation

- **Impératif** et **Structuré**
- **Fonctionnel** (*impur*)
- **Générique** (polymorphisme par paramétrage de type)
- **Concurrent**

- basé sur un système de typage **statique**, **fort*** et **inféré**
  *: beaucoup plus *fort* que C++; dans la veine de OCaml



Influences historiques de Rust

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

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

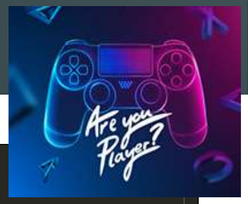
Même s'il n'y a **toujours** qu'un seul propriétaire, il est possible de prêter temporairement une donnée en lecture seule ou bien en lecture/écriture.

En Rust, l'emprunt (*borrow*)

- se fait sur toute la profondeur de la structure (< 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*

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]

Monas Hieroglyphica de John Dee

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

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

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);
```

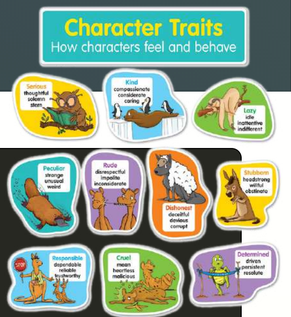
- De vérifier une déstructuration:

```
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),
}
```

Traits



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

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++.

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_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"
    }
}
```

Macros!

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)

```
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() {}
}
```

Généricité

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_no_usage.rs [slice 0:-4]

```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_usage.rs [slice 0:-7]

```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.

Comparaisons avec C++

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 (même si le principe RAII aide pas mal)
- ⊖ 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:]

```
 1  struct Object {···};
 6
 7  [[nodiscard]] constexpr auto function(const Object &o) noexcept(noexcept(&Obje
10
11  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 (pas de surcharge)
- ⊖ 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[1/2]

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[2/2]

Image from #automotocha9l

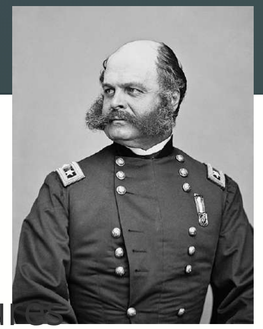
… avec certains potentiels non négligeables en plus:

- 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.

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)

Pour se lancer avec **Rust**

- Le site officiel (avec beaucoup de documentations)
- Rust Play Ground, un compilateur en ligne (comme compiler explorer)
- https://rustup.rs, le site officiel pour installer un l'environnement prêt à l'emploi.
- https://crates.io, le principal dépôts des *crates* (i.e. *packages*)

Merci pour votre attention