Pourquoi calculer en Rust ?

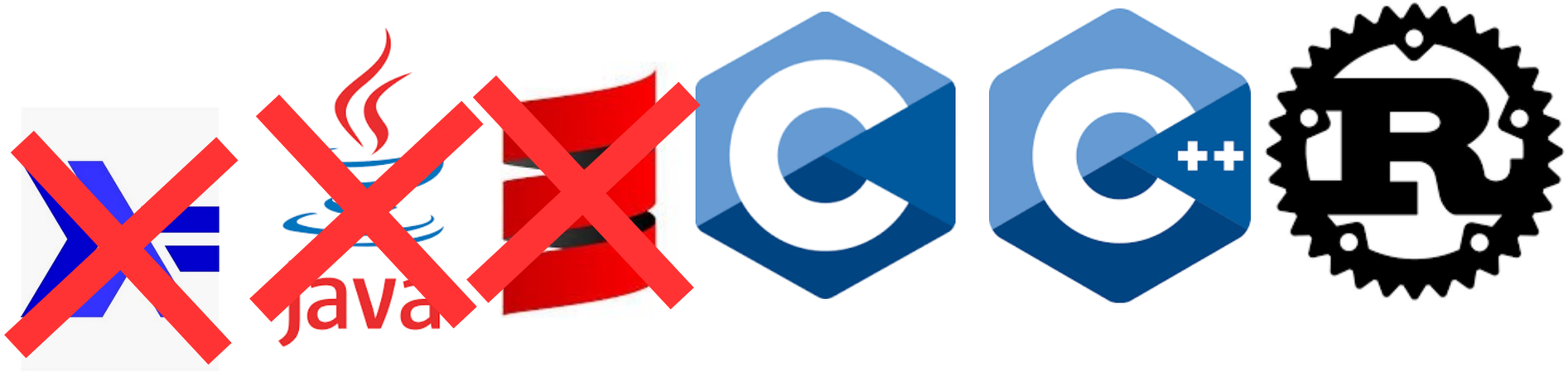
Hadrien Grasland

2024-02-29

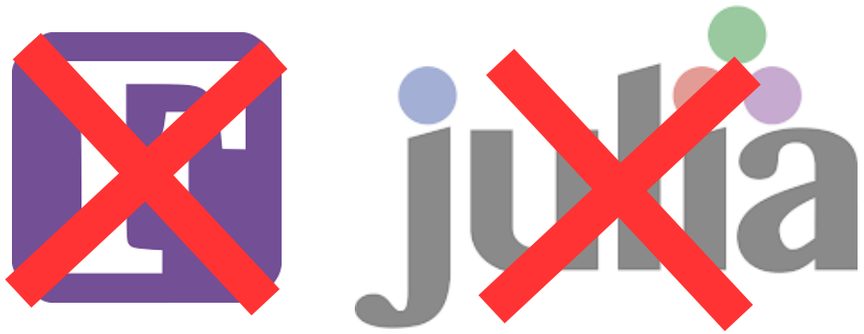
Mon langage de calcul idéal ?

Un langage généraliste

Facile d’optimiser les performances

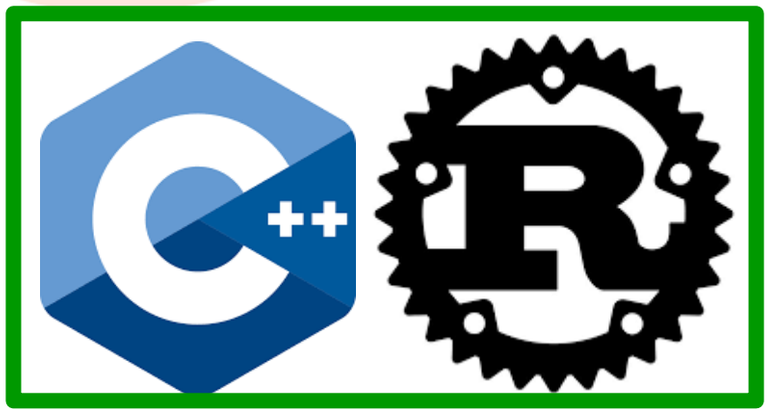


Bien équipé pour les gros projets*



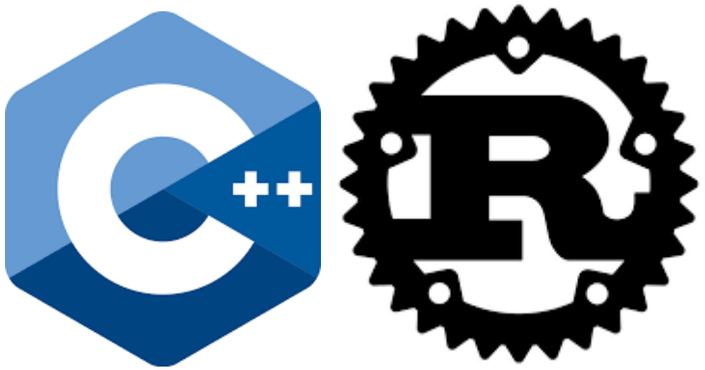
* Quand on a besoin d’encapsulation, généricité, polymorphisme, génération de code, API hiérarchisée…

Pas tant de choix au final…



Beaucoup de ressemblances

- Compilation AoT (normalement)
- Pas de ramasse-miettes
- Typage strict et explicite
- Contrôle bas niveau
- Métaprogrammation
- Abstractions riches, *zero-cost*
- Effort d’apprentissage important



- **Qu’est-ce qui est différent en Rust ?**

Comportement indéfini en C++

- Présumé impossible par le compilateur → Effet imprévisible
- **Arithmétique :** Overflow, shift > bits, -INT_MIN, *casts*…
- **Tableaux :** Accès hors bornes, invalidation d'itérateurs…
- **Pointeurs/références :** Nul, mal aligné, invalide, *strict aliasing*…
- **Mémoire non initialisée :** Lecture = comportement indéfini (attention aux destructeurs, affectations, exceptions…)
- **Boucles infinies** qui violent le théorème de Fermat
- **Multi-threading :** Accès concurrent aux données partagées
- Et bien d'autres → Inévitable dans un code non trivial…

Conséquence : Failles de sécurité*

- Taux de vulnérabilités liées à des erreurs mémoires :
  - 65 % dans Android (90 % des vulns media & bluetooth)
  - 65 % dans le noyau Linux (Ubuntu)
  - 66 % dans iOS, 72 % dans macOS
  - 70 % dans Chrome
  - 70 % dans les produits Microsoft
  - 74 % dans le traitement CSS de Firefox
- …et ce n'est pas le seul type de comportement indéfini !

* Peut-être qu'aujourd'hui vous pouvez vous permettre d'ignorer ce problème.
Mais qu'est-ce qui vous dit que demain votre code ne sera pas appelé par une visu web ?

Sûreté en Rust

- En Rust, hors blocs *unsafe*, **sûreté vs comportement indéfini** :
  - **Typage :** Les valeurs respectent les invariants de type
  - **Mémoire :** Références valides, mémoire initialisée
  - **Threads :** Pas d'accès concurrents non synchronisés
- En pratique, c'est un **bon compromis**
  - Besoin *d'unsafe* : preuve à la compilation parfois impossible, vérification à l'exécution parfois trop coûteuse
  - MAIS rarement requis au quotidien
  - Utilisation rare et facile à localiser → Code facile à auditer

Faiblesses de typage en C++

- **Comportements imprévus et erreurs incompréhensibles** résultant de l'interaction entre…
  - Conversions implicites (dont constructeurs non-explicites)
  - Surcharge de fonctions + arguments par défaut
  - Généricité via les *templates* + spécialisation
  - Méthodes virtuelles
- **Peu de vérifications de typage*** dans les *templates*
  - Equivalent à un langage *duck typé* comme Python

* Je reviens sur la question des concepts plus loin

Un programme C++ simple

```cpp
#include <algorithm>
#include <cstddef>
#include <concepts>
#include <iostream>
#include <vector>

template<std::floating_point T>
T median(const std::vector<T>& input) { /* ... */ }

int main()
{
    std::cout << median(std::vector<float>{ 1.2, 3.4, 5.6 }) << std::endl;
}
```

Sortie du compilateur

```
      |      ^~~~~~~~
/usr/include/c++/13/ostream:694:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   cannot convert ‘input’ (type ‘const std::vector<float>’) to type ‘const char8_t*’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~
/usr/include/c++/13/ostream:699:5: note: candidate: ‘template<class _Traits> std::basic_ostream<char, _Traits>& std::operator<<(basic_ostream<char, _Traits>&, const char16_t*)’ (deleted)
  699 |      operator<<(basic_ostream<char, _Traits>&, const char16_t*) = delete;
      |      ^~~~~~~~
/usr/include/c++/13/ostream:699:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   cannot convert ‘input’ (type ‘const std::vector<float>’) to type ‘const char16_t*’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~
/usr/include/c++/13/ostream:703:5: note: candidate: ‘template<class _Traits> std::basic_ostream<char, _Traits>& std::operator<<(basic_ostream<char, _Traits>&, const char32_t*)’ (deleted)
  703 |      operator<<(basic_ostream<char, _Traits>&, const char32_t*) = delete;
      |      ^~~~~~~~
/usr/include/c++/13/ostream:703:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   cannot convert ‘input’ (type ‘const std::vector<float>’) to type ‘const char32_t*’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~
/usr/include/c++/13/ostream:709:5: note: candidate: ‘template<class _Traits> std::basic_ostream<wchar_t, _Traits>& std::operator<<(basic_ostream<wchar_t, _Traits>&, const char8_t*)’ (deleted)
  709 |      operator<<(basic_ostream<wchar_t, _Traits>&, const char8_t*) = delete;
      |      ^~~~~~~~
/usr/include/c++/13/ostream:709:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   mismatched types ‘wchar_t’ and ‘char’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~
/usr/include/c++/13/ostream:714:5: note: candidate: ‘template<class _Traits> std::basic_ostream<wchar_t, _Traits>& std::operator<<(basic_ostream<wchar_t, _Traits>&, const char16_t*)’ (deleted)
  714 |      operator<<(basic_ostream<wchar_t, _Traits>&, const char16_t*) = delete;
      |      ^~~~~~~~
/usr/include/c++/13/ostream:714:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   mismatched types ‘wchar_t’ and ‘char’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~
/usr/include/c++/13/ostream:718:5: note: candidate: ‘template<class _Traits> std::basic_ostream<wchar_t, _Traits>& std::operator<<(basic_ostream<wchar_t, _Traits>&, const char32_t*)’ (deleted)
  718 |      operator<<(basic_ostream<wchar_t, _Traits>&, const char32_t*) = delete;
      |      ^~~~~~~~
/usr/include/c++/13/ostream:718:5: note:   template argument deduction/substitution failed:
concept.cpp:9:50: note:   mismatched types ‘wchar_t’ and ‘char’
    9 |      std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
      |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~
/usr/include/c++/13/ostream:801:5: note: candidate: ‘template<class _Ostream, class _Tp> _Ostream&& std::operator<<(_Ostream&&, const _Tp&)’
  801 |      operator<<(_Ostream&& __os, const _Tp& __x)
      |      ^~~~~~~~
/usr/include/c++/13/ostream:801:5: note:   template argument deduction/substitution failed:
/usr/include/c++/13/ostream: In substitution of ‘template<class _Ostream, class _Tp> _Ostream&& std::operator<<(_Ostream&&, const _Tp&) [with _Ostream = std::basic_ostream<char>&; _Tp = std::vector<float>]’:
concept.cpp:9:50:   required from ‘T median(const std::vector<T>&) [with T = float]’
concept.cpp:22:24:   required from here
/usr/include/c++/13/ostream:801:5: error: template constraint failure for ‘template<class _Os, class _Tp>  requires (__derived_from_ios_base<_Os>) && requires(_Os& __os, const _Tp& __t) {__os << __t;} using std::__rvalue_stream_insertion_t = _Os&&’
/usr/include/c++/13/ostream:801:5: note: constraints not satisfied
/usr/include/c++/13/ostream: In substitution of ‘template<class _Os, class _Tp>  requires (__derived_from_ios_base<_Os>) && requires(_Os& __os, const _Tp& __t) {__os << __t;} using std::__rvalue_stream_insertion_t = _Os&& [with _Os = std::basic_ostream<char>&; _Tp = std::vector<float>]’:
/usr/include/c++/13/ostream:801:5:   required by substitution of ‘template<class _Ostream, class _Tp> _Ostream&& std::operator<<(_Ostream&&, const _Tp&) [with _Ostream = std::basic_ostream<char>&; _Tp = std::vector<float>]’
concept.cpp:9:50:   required from ‘T median(const std::vector<T>&) [with T = float]’
concept.cpp:22:24:   required from here
/usr/include/c++/13/ostream:768:13:   required for the satisfaction of ‘__derived_from_ios_base<_Os>’ [with _Os = std::basic_ostream<char, std::char_traits<char> >&]
/usr/include/c++/13/ostream:768:39: note: the expression ‘is_class_v<_Tp> [with _Tp = std::basic_ostream<char, std::char_traits<char> >&]’ evaluated to ‘false’
  768 |      concept __derived_from_ios_base = is_class_v<_Tp>
      |                                        ^~~~~~~~~~~~~~~
hadrien@silent-graloufotron:~/Bureau/concept>
```

Où ~~est~~ sont les problèmes ?

Compile si pas instancié, malgré l'utilisation des concepts
(équivalent C++ du crash au runtime de Python, Julia...)

Illégal en C++
(pas d'alternative)

```cpp
template<std::floating_point T>
T median(const std::vector<T>& input) {
    std::cout << "DEBUG: Finding the median of " << input << "..." << std::endl;
    std::vector<T> sorted = input;
    std::sort(sorted.begin(), sorted.end());
    std::size_t midpoint = sorted.size() / 2;
    if (sorted.size() % 2 == 0) {
        return (sorted[midpoint] + sorted[midpoint + 1]) / 2.0;
    } else {
        return sorted[midpoint];
    }
}
```

UB si input.size() == 0

Aller retour float → double → float si T = float

Résultat douteux en présence de NaN
(UB probable avec un tri tierce partie)

Où ~~est~~ sont les problèmes ?

Je code en C++ depuis **~20 ans**

Mon code spontané reste rempli de ce genre de coquilles

Seule une infime partie est détectée par le compilateur (-Wall -Wextra)

Il me faut des heures de relectures, tests… pour en arriver à un résultat correct

Typage fort en Rust

- **Généricité contrainte** par les *traits** :
  - Le code générique spécifie de quoi il a besoin
  - Utiliser autre chose est une erreur de compilation
  - A comparer aux concepts C++20 : erreur silencieuse !
- Comportement **beaucoup plus prévisible**
  - Pas de conversion implicite
  - Le polymorphisme n'est possible que via les *traits*
  - L'utilisation de *traits* est explicite et sans ambiguïté

* Je détaille un peu plus loin.

Équivalent Rust du code précédent

```rust
use num_traits::Float;

fn median<T: Float>(input: &Vec<T>) -> T {
    println!("DEBUG: Finding the median of {input}...");
    let mut sorted = input.clone();
    sorted.sort_unstable();
    let midpoint = sorted.len() / 2;
    if sorted.len() % 2 == 0 {
        (sorted[midpoint] + sorted[midpoint + 1]) / 2.0
    } else {
        sorted[midpoint]
    }
}

fn main() {
    println!("{}", median(&vec![1.2, 3.4, 5.6]));
}
```

Sortie du compilateur : 3 erreurs

```
hadrien@silent-graloufotron:~/Bureau/concept/concept-rs> cargo check
    Checking concept-rs v0.1.0 (/home/hadrien/Bureau/concept/concept-rs)
error[E0277]: `Vec<T>` doesn't implement `std::fmt::Display`
 --> src/main.rs:4:44
  |
4 |     println!("DEBUG: Finding the median of {input}...");
  |                                            ^^^^^^^ `Vec<T>` cannot be formatted with the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for `Vec<T>`
  = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)

error[E0277]: the trait bound `T: Ord` is not satisfied
    --> src/main.rs:6:12
     |
6    |     sorted.sort_unstable();
     |            ^^^^^^^^^^^^^^ the trait `Ord` is not implemented for `T`
     |
note: required by a bound in `core::slice::<impl [T]>::sort_unstable`
    --> /home/hadrien/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/core/src/slice/mod.rs:2951:12
     |
2949 |     pub fn sort_unstable(&mut self)
     |            ------------- required by a bound in this associated function
2950 |     where
2951 |         T: Ord,
     |            ^^^ required by this bound in `core::slice::<impl [T]>::sort_unstable`
help: consider further restricting this bound
     |
3    | fn median<T: Float + std::cmp::Ord>(input: &Vec<T>) -> T {
     |                    +++++++++++++++

error[E0308]: mismatched types
 --> src/main.rs:9:53
  |
3 | fn median<T: Float>(input: &Vec<T>) -> T {
  |           - expected this type parameter
...
9 |         (sorted[midpoint] + sorted[midpoint + 1]) / 2.0
  |         ----------------------------------------   ^^^ expected type parameter `T`, found floating-point number
  |         |
  |         expected because this is `T`
  |
  = note: expected type parameter `T`
                     found type `{float}`

Some errors have detailed explanations: E0277, E0308.
For more information about an error, try `rustc --explain E0277`.
error: could not compile `concept-rs` (bin "concept-rs") due to 3 previous errors
hadrien@silent-graloufotron:~/Bureau/concept/concept-rs>
```

Erreur 1 : Affichage de Vec<T>

```
hadrien@silent-graloufotron:~/Bureau/concept/concept-rs> cargo check
    Checking concept-rs v0.1.0 (/home/hadrien/Bureau/concept/concept-rs)
error[E0277]: `Vec<T>` doesn't implement `std::fmt::Display`
 --> src/main.rs:4:44
  |
4 |     println!("DEBUG: Finding the median of {input}...");
  |                                            ^^^^^^^ `Vec<T>` cannot be formatted with the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for `Vec<T>`
  = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the macro `println`
```

- Signale le problème **même si jamais instancié**
- Propose une alternative : L'affichage de déboguage Debug {:?}

Erreur 2 : Tri de flottants

```
error[E0277]: the trait bound `T: Ord` is not satisfied
    --> src/main.rs:6:12
     |
6    |     sorted.sort_unstable();
     |            ^^^^^^^^^^^^^^ the trait `Ord` is not implemented for `T`
     |
note: required by a bound in `core::slice::<impl [T]>::sort_unstable`
    --> /home/hadrien/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/core/src/slice/mod.rs
     |
2949 |     pub fn sort_unstable(&mut self)
     |            ------------- required by a bound in this associated function
2950 |     where
2951 |         T: Ord,
     |            ^^^ required by this bound in `core::slice::<impl [T]>::sort_unstable`
help: consider further restricting this bound
     |
3    | fn median<T: Float + std::cmp::Ord>(input: &Vec<T>) -> T {
     |                    +++++++++++++++
```

- Refuse de trier des flottants par défaut (**NaN pas ordonné**)
  - Plusieurs moyens de faire l'assertion qu'il n'y a pas de NaN

Erreur 3 : Division par une litérale

```
error[E0308]: mismatched types
 --> src/main.rs:9:53
  |
3 | fn median<T: Float>(input: &Vec<T>) -> T {
  |           - expected this type parameter
...
9 |         (sorted[midpoint] + sorted[midpoint + 1]) / 2.0
  |         ----------------------------------------   ^^^ expected type parameter `T`, found floating-point number
  |         |
  |         expected because this is `T`
  |
  = note: expected type parameter `T`
                      found type `{float}`

Some errors have detailed explanations: E0277, E0308.
For more information about an error, try `rustc --explain E0277`.
error: could not compile `concept-rs` (bin "concept-rs") due to 3 previous errors
hadrien@silent-graloufotron:~/Bureau/concept/concept-rs>
```

- Pas de type/conversion implicite pour les litérales flottantes
- Pointe vers de l'aide complémentaire en fin de compilation

Reste une erreur non détectée*

- La version Rust est toujours **invalide pour un vecteur vide**
- Conduira à un crash déterministe (panic) à l'exécution
  - Pas de comportement indéfini comme en C++
- Est-ce la meilleure stratégie ?
  - Oui si on considère ça comme une erreur d'utilisation
    - Ne pas oublier de l'indiquer dans la documentation !
  - Sinon, on utilisera plutôt **Option<T>** : Some(T) ou None

* Eh non, Rust ne vous épargnera pas d'écrire des tests. Ils échoueront juste moins souvent.

Polymorphisme bancal en C++

- Deux mécanismes **complètement incompatibles**
  - Héritage et méthodes virtuelles
  - *Templates*, concepts et spécialisation
- Passer de l'un à l'autre est **très difficile**
  - Parfois requis : compromis coût compilation/exécution
  - Nécessite de tout réapprendre + réécrire beaucoup de code
- **Interopérabilité difficile** avec types extérieurs non modifiables

Traits en Rust*

- **Même formalisme** en polymorphisme statique & dynamique
  - Méthodes sans données, comme les interfaces Java

    + constantes et types associés

    + méthodes et types génériques

    + implémentations par défaut
  - Mais certaines choses ne sont supportées qu'en statique
- Implémentable pour un **objet extérieur** → Interopérabilité !

* Cette fonctionnalité est très fortement inspirée des *typeclasses* de Haskell

Exemple : SAXPY parallèle en C++17

- Utilisation de Intel TBB via std (*leaky abstraction* !)

```
std::transform(std::execution::par,
»          std::begin(tabX), std::end(tabX),
»          std::begin(tabY), std::begin(tabResult),
»          [=](float xi, float yi){ return a*xi + yi; });
```

- Niveau de bruit syntaxique insupportable*
- Attention, tabY doit être de la même taille, sinon UB
- Pensez à préallouer tabResult de la bonne taille, sinon UB
- S'adapte mal à du travail plus complexe (ergonomie + perf)

* Un peu réduit par les ranges C++20… quand ils fonctionneront correctement partout.
Mais le code utilisant les std::algorithm existants ne va pas se réécrire tout seul…

Exemple : SAXPY parallèle en Rust

- Utilisation de la bibliothèque rayon via un *extension trait*

```
use rayon::prelude::*;
let result = x.into_par_iter().zip(y)
              .map(|(x, y)| a * x + y)
              .collect::<Vec<_>>();
```

- Ressemble à std sans devoir y être intégré
- Syntaxe plus lisible, implème tout aussi efficace
- Facile de construire un tableau de résultats à la volée*
- Supporte des *pipelines* beaucoup plus complexes

* Pas beaucoup plus compliqué de réutiliser un tableau existant, sans risque d'UB.

Erreurs chaotiques en C++

- Historiquement, fort accent sur les **exceptions**
  - Lancement/capture très coûteux en performances
  - Très difficile d'écrire du code correct en cas d'*unwinding*
  - Découragé dans les destructeurs, mais pas d'alternative
- Hors de ce mécanisme, c'est **la jungle**
  - Valeurs spéciales ou int que personne ne lit, comme en C
  - Types exotiques spécifiques à chaque projet
  - Documentation souvent incomplète

Gestion d'erreurs en Rust

- Au quotidien, type variant **Result<T, E>**
  - Contient soit un résultat de type T, soit une erreur de type E*
  - Pour accéder au résultat, il faut gérer l'erreur éventuelle
- ***Panic* pour les erreurs fatales** (ex : echec d'une assertion)
  - Implémenté soit comme une exception, soit abort() direct
  - Capture possible, mais rare et non encouragée
- Fort **consensus communautaire** + erreurs bien documentées

* Exploite le support des types sommes de Rust (comme std::variant, mais utilisable)

Génération de code

- En C++, on a beaucoup de ***template metaprogramming***
  - Basé sur un bug de compilateur anobli (SFINAE)
  - Code réservé à quelques initiés, très difficile à maintenir
  - Très inefficace → Builds lentes, gourmandes en RAM
  - Sinon: Parsing de sorties de debug de compilateur, générateurs de code externes* comme ROOT…
- En Rust, outre les traits, on utilise beaucoup des **macros**
  - Parsing/génération de code intégré, opère sur l'AST
  - Compromis ergonomie/puissance mieux maîtrisé

* …avec ce que ça implique en termes de complexité du processus de construction.

Exemple : serde

- Générateur de code pour (dé)sérialisation ~universelle

```
1 use serde::{Deserialize, Serialize};
2
3 #[derive(Debug, Serialize, Deserialize)]
4 struct Record {
5     idx: u32,
6     data: f64,
7     comment: String,
8 }
```

N'a pas besoin d'être dans std

Permet la sérialisation JSON, CSV, Pickle…

Macro std pour générer un affichage de debug

Utilisation

```rust
fn main() {
    println!(
        "serialized: {:#?}\ndeserialized: {:#?}",
        serde_json::to_string(&Record {
            idx: 123,
            data: 4.56,
            comment: "Hello".into()
        }),
        serde_json::from_str::<Record>(
            r#"{ "idx": 42, "data": 6.55, "comment": "LOL" }"#
        ),
    );
}
```

```
serialized: Ok(
    "{\"idx\":123,\"data\":4.56,\"comment\":\"Hello\"}",
)
deserialized: Ok(
    Record {
        idx: 42,
        data: 6.55,
        comment: "LOL",
    },
)
```

Gestion des erreurs via Result

Compilation et dépendances

- En C++, on a **CMake** et les paquets des distributions Linux
  - Insatisfaction quasi-unanime vis à vis de ces outils
  - Conséquence : **Peu de réutilisation de code**
- En Rust, on a **cargo** en standard*
  - Combine gestion de compilation + dépendances
  - Simple pour un projet de petite taille, passe bien à l'échelle
  - Gestion des dépendances triviale → **Partage de code !**

* Parmi beaucoup d'autres outils aujourd'hui attendus : générateur de doc, tests…

Fragmentation de la communauté C++

- **Humainement**
  - Fracture entre les pratiquants C++98, 11/14/17 et 20/23
  - Enseignement déficient et en déclin → **Va en s'aggravant**
  - Les codes réels sont des mélanges de styles incohérents
- **Technologiquement**
  - N compilos incompatibles = une chance ?
  - En 2024, **support C++20 incomplet** sur compilos récents
  - Distributions RHEL, embarquées = **compilateurs antiques**…

Le prix de la compatibilité C/++

- C++ doit beaucoup à la compatibilité avec le C et lui-même
- Mais ça veut dire aussi vivre pour l'éternité avec…
  - Le **préprocesseur C**, ses macros, ses includes…
  - Des **types primitifs mal définis** : long (taille?), char (signe?)…
  - Les **litérales typées** (nécessité d'avoir 42ULL, 1.2f…)
  - Des **conversions implicites** piégeuses/coûteuses
  - **switch avec fallthrough**, affectation qui **copie par défaut**
  - **std::vector<bool>**, std::**numeric_limits::min** vs lowest…
  - …et tant d'autres choses dont Rust peut se libérer

Rust du présent = C++ du futur

- **C++17** vu de **Rust v1 (2015)**
  - **De nombreux rattrapages :**
    filesystem, any, string_view, byte, aligned_alloc
  - **Des cousins à peine utilisables :**
    optional, std::tuple, *structured bindings*, CTAD, std::variant
- Même tendance au réchauffé en **C++20** :
  - **Rattrapages :** Ranges, <=>, consteval, { .a }, format, span, endian, <bit>, barrier, latch, jthread, assume_aligned
  - **Alternatives ratées :** Coroutines, modules, concepts

En conclusion

- 2 bonnes raisons de lancer un projet C++ en 2024
  - Lié à un gros code C++ existant qu'on ne veut pas réécrire
  - Bibliothèques/outils C++ plus matures pour votre domaine
- Hors de ces situations, essayez Rust* !
  - Rare d'être bloqué par un écart de fonctionnalité langage
  - Ergonomie infiniment supérieure
  - Moins de debug → Plus de fonctionnalités & optimisations
  - Moins obscur et mieux supporté que C++2x

* Lien vers un cours FR orienté programmeurs C++, voir aussi The Rust Programming Language (débutant), Programming Rust (avancé, livre), Rust by Example (orienté examples d'utilisation)…

La suite au prochain épisode…

- 1ère partie aujourd'hui : Pourquoi calculer en Rust ?
- **2e partie le 18/04** : Comment calculer en Rust ?
  - Manipulations tabulaires
  - ILP et vectorisation
  - Parallélisation
  - GPU…

Merci de votre attention !

Comprendre le comportement indéfini

- Soit a, b et c trois entiers, on veut savoir si a*b == a*c
  - Naïvement, il faut 2 IMUL (coûteux) et un CMP
- Imaginez un compilateur qui sait simplifier en b == c
  - Elimine les deux IMUL coûteuses !
  - Comme toutes les optims, **pas toujours appliqué**
- Cette optim n'est OK que si a != 0 et il n'y a pas d'overflow
  - Sinon, elle sera source d'erreurs non déterministes
- -ffast-math = Une panoplie d'UB* en virgule flottante !

* Entre autres : NaN, +/-inf et -0 n'apparaissent ni en entrée ni en résultat d'une opération, pas d'exceptions IEEE-754, le mode d'arrondi est présumé être round-to-nearest...