Nicolas Polomack

PRÉSENTATION DU LANGAGE

RUST

INSTALLATION

- Visitez https://rustup.rs
- Suivez les instructions du script d'installation (chemin à ajouter au PATH)
- Vous devriez maintenant avoir:
  - rustup: le gestionnaire de version du compilateur.
  - rustc: le compilateur (à la dernière version: actuellement 1.30.1).
  - cargo: le gestionnaire de packets et système de build.

CODE SAMPLES

- La totalité du code utilisé lors du talk est disponible sur: https://github.com/Hirevo/intro-to-rust-talk
- Lien pour cloner: git@github.com:Hirevo/intro-to-rust-talk.git
- Pour builder et lancer les binaires:
  - 1ère méthode: `$ cargo run --bin <nom du binaire>`
  - 2ème méthode:
    - `$ cargo build –bins`
    - puis: `$ ./target/debug/<nom du binaire>`
- Les noms de binaires sont listés dans le Cargo.toml et seront indiqués sur les slides.

SOMMAIRE

- Déclaration de mission et objectifs du langage
- Syntaxe, types et structures de contrôles du langage
- Gestion de mémoire: « ownership », emprunt, durées de vie et mutabilité
- Système de « trait »

MISSION ET OBJECTIFS

DÉCLARATION DE MISSION

Rust est un langage de programmation système ultra-rapide, qui prévient les erreurs de segmentation et garantit la sûreté entre threads.

LES OBJECTIFS

- Sécurité mémoire (garanties si le programme compile):
  - Pas d'erreurs de segmentation.
  - Pas de use-after-free, double-free, invalid-free ou de fuite mémoire.
  - Destruction des ressources déterministe.
  - Pas de garbage-collector (ou aucun autre type de runtime additionnel).
  - Pas de data-races (deux fils concurrents convoitant une ressource commune sans protection ou synchronisation).

LES OBJECTIFS

- Performance:
  - Zero-cost abstractions: tout comme C++
  - Sémantiques de "move" par défaut (au lieu de ceux de copie du C/C++)
  - Pas de runtime (pas de VM, pas de GC, pas de green threads, etc...)
  - Excellente FFIs (Foreign Function Interfaces, notamment avec le C)
  - Utilise LLVM en tant que back-end (profite des optimisations)

LES OBJECTIFS

- Productivité:
  - Elimine beaucoup de types d’erreurs (comme ceux liés aux durées de vie)
  - Rend la programmation système plus pratique via:
    - Meilleurs messages d’erreur (les meilleurs jamais vus)
    - Son gestionnaire de packets: ‘cargo'
  - Librairie standard mieux pensée et plus pratique

SYNTAXE ET MÉCANIQUES DU LANGAGE

TYPES PRIMITIFS (REF: 2-TYPES)

- Basiques: char, bool (un char fait 4 bytes, car il représente une valeur Unicode)
- Entiers: i8, i16, i32, i64, i128 et isize (comme ssize_t, taille variable)
- Non-signés: u8, u16, u32, u64, u128 et usize (comme size_t, taille variable)
- Flottants: f32 et f64
- Chaines de caractères: String et &str (similaire à std::string et std::string_view)
- Tableaux: Vec<T>, [T; N] et [T] (comme std::vector, std::array et std::span)
- Error handling: Option et Result (pas de try/catch ou d'exceptions)
- Autres: structs, enums, tuples, le type « () », le type « ! », références, pointeurs (unsafe)

VARIABLES & MUTABILITY (REF: 3-MUTABILITY)

- Par défaut, toute variable est constante:
  - let x = 5; // x est un i32 constant égal à 5
- Pour être mutable, il faut l'exprimer explicitement:
  - let mut x = 5; // x est un i32 mutable égal à 5
- Une variable peut être redéfinie dans le même scope:
  - let x = 5; let x = "toto"; // il y a 2 variables différentes nommées 'x'.
  - Le deuxième 'x' met de l'ombre sur le premier 'x' (cf: 'variable shadowing')

SÉMANTIQUES DE « MOVE » (REF: 4-MOVE_SEMANTICS)

- Par défaut, lorsqu'une valeur est utilisé lors d'un appel de fonction, elle est déplacée (« moved »):
- Si elle était dans une variable, la variable telle quelle devient inutilisable.
- Cela ouvre des portes à des optimisations (contre l'action de simple « copie »).
- Cette mécanique est au cœur du système de gestion de ressources de Rust.
- La copie d'une valeur est explicite (via 'x.clone()').
- Ceci ne s'applique pas pour les types primitifs (eg. integers, floats, bools, chars).

OWNERSHIP & BORROWING (REF: 5-REFERENCES)

- Chaque valeur a un seul et unique propriétaire.
- L'action de « move » une valeur équivaut à un transfert de propriété.
- Pour passer des valeurs sans transférer la propriété, on peut la « prêter »:
  - 'do_stuff(x);' prend la propriété de 'x'.
  - 'do_stuff(&x);' emprunte 'x' en lecture seule.
  - 'do_stuff(&mut x);' emprunte 'x' en lecture/écriture.
- Après la fin de l'appel de fonction, l'emprunt s'arrête et la valeur reste utilisable pour le propriétaire original.

OWNERSHIP & BORROWING (REF: 5-REFERENCES)

- Les pointeurs existent en Rust (pour pouvoir inter-opérer avec des fonctions C) mais tout, en Rust, est faisable sans pointeurs.
- Utiliser des pointeurs vous force à entrer dans la section « unsafe » du langage:
  - La fonction entière ou la section critique devra être annoté avec le mot-clé « unsafe ».
  - Le compilateur ne vous fera plus aucune garantie, ni dans le bloc « unsafe », ni dans tout ce qui y a touché de près ou de loin au bloc « unsafe ».
    - Excepté qu'il va en profiter pour insister à checker les pointeurs nuls comme jamais.

GARANTIES ET RÈGLES DU LANGAGE (REF: 6-ALIASING)

- Le compilateur permet d'établir des garanties quand à la validité et « santé mémoire » de votre code en imposant quelques règles:
  - Une valeur n'est mutable que si un seul block de code y a accès.
  - Une valeur mutable devient immutable tant qu'un emprunt immutable est en cours.
  - Un seul emprunt mutable ou plusieurs emprunts immutables, pas les deux.
  - Les durées de vies sont traquées par le compilateur pour assurer que ces règles tiennent.

STRUCTURES DE CONTRÔLE (REF: 7, 8 ET 9)

- Conditions: 'if' (pas de ternaires, car 'if' est une expression)
- Boucles: 'while', 'for in' et 'loop' ('loop' == 'while (true)')
  - Contrôles de boucle via 'continue' et 'break'
  - Les 'for in' fonctionnent à base d’itérateurs
- Pattern matching: 'if let', 'while let' et 'match'

PATTERN-MATCHING ET DESTRUCTURATION (REF: 10 ET 11)

- Les tuples, enums, structures et valeurs primaires peuvent être déstructurés:
  - Lors d'une déclaration (doit être exhaustif): `let (x, y) = v;`
  - Lors d'un `if let` ou `while let` (non-exhaustif): `if let Some(val) = result {`
  - Lors d'un `match` (doit être exhaustif)

GESTION DES RESSOURCES (REF: 12-MEMORY_MODEL)

- Rust ne requiert pas de « destruction manuel » des ressources (telle qu'une allocation mémoire, un fichier ouvert, une connexion en cours, etc…).
- La ressource est libérée (appelé un « drop ») immédiatement après que la variable soit perdue de vue par son propriétaire.
- La durée de vie d'une ressource peut donc être étendue ou raccourcie en la déplaçant dans un autre block (via un « move »).
- Le compilateur s'assure qu'aucun emprunt est utilisé après le « drop » d'une valeur (une erreur est émise le cas échéant, le fameux « use after move »).

GESTION DES ERREURS (REF: 13-ERRORS)

- En Rust, la manière idiomatique de gérer toute erreur est avec l'usage des enums « Option<T> » et « Result<T, E> »:
  - Option peut être un « Some(T) » ou un « None ».
  - Result peut être un « Ok(T) » ou un « Err(E) ».
- Result diffère d'Option en fournissant une explication ou un résultat alternatif à l'échec d'une opération.
- Option et Result possèdent des fonctions pour les enchaîner sans danger et remonter une erreur si une survient à n'importe quel moment dans la chaîne.

SYSTÈME DE TRAITS (REF: 14-TRAITS)

- Un « trait » est un ensemble de propriétés et méthodes décrivant une capacité commune à plusieurs types.
- Exemples de traits:
  - Opérateurs: Add, Sub, Mul, Div, Rem, Shl, Shr, …
  - Sémantiques: Clone, Copy
  - Affichage: Debug, Display
  - Itérables: Iterator, IntoIterator
  - Conversion: From<T>, Into<T>

SYSTÈME DE TRAITS (REF: 14-TRAITS)

- Invocable: Fn, FnOnce, FnMut
- Comparaison/égalité: PartialEq, Eq, PartialOrd, Ord
- Constructible par défaut: Default
- Destructible: Drop
- Thread-Safe: Send, Sync
- Utilisable en code retour du main: Termination
- IO: Read, BufRead, Write, BufWrite
- ...

PANICS (REF: 15-PANICS)

- En cas d'erreurs graves dont on ne peut pas recouvrer un fil d'exécution normal, un « panic » peut être levé.
- Un « panic » est un unwinding de la call stack, appelant tous les destructeurs et libérant toutes les ressources.
- Un « panic » est un évènement controlé dont le comportement est 100% bien défini, prédictible, sans danger et sans fuite.
- Il est possible de « catch » un « panic » via `std::panic::catch_unwind`.

BONUS – TYPES DE COLLECTIONS DE LA LIBRAIRIE STANDARD

- Séquence: Vec, VecDeque, LinkedList
- Clé-Valeur: HashMap, BTreeMap
- Set: HashSet, BTreeSet
- File de priorité: BinaryHeap

MERCI D'AVOIR SUIVI !
DES QUESTIONS ?