

Chapitre 1. Les programmeurs de systèmes peuvent avoir de belles choses

Dans certains contextes, par exemple le contexte ciblé par Rust, être 10 fois plus rapide, voire 2 fois plus rapide que la concurrence, est une chose décisive. Il décide du sort d'un système sur le marché, autant qu'il le ferait sur le marché du matériel.

—[Graydon Hoare](#)

*Tous les ordinateurs sont maintenant parallèles...
La programmation parallèle **est** la programmation.*

—Michael McCool et al., *Programmation parallèle structurée*

Faillite de l'analyseur TrueType utilisée par un attaquant d'État-nation pour la surveillance ; tous les logiciels sont sensibles à la sécurité.

—[Andy Wingo](#)

Nous avons choisi d'ouvrir notre livre avec les trois citations ci-dessus pour une raison. Mais commençons par un mystère. Que fait le programme C suivant?

```
int main(int argc, char **argv) {  
    unsigned long a[1];  
    a[3] = 0x7ffff7b36cebUL;  
    return 0;  
}
```

Sur l'ordinateur portable de Jim ce matin, ce programme a imprimé:

undef: Error: .netrc file is readable by others.

undef: Remove password or make file unreadable by others.

Puis il s'est écrasé. Si vous l'essayez sur votre machine, il peut faire autre chose. Qu'est-ce qui se passe?

Le programme est défectueux. Le tableau n'a qu'un seul élément de long, donc l'utilisation est, selon la norme du langage de programmation C, *un comportement indéfini* : `a a[3]`

Comportement, lors de l'utilisation d'une construction de programme non portable ou erronée ou de données erronées, pour lesquelles la présente Norme internationale n'impose aucune exigence

Un comportement indéfini n'a pas seulement un résultat imprévisible : la norme autorise explicitement le programme *à faire n'importe quoi*. Dans notre cas, le stockage de cette valeur particulière dans le quatrième élément de ce tableau particulier corrompt la pile d'appels de fonction de sorte que le retour de la fonction, au lieu de quitter le programme gracieusement comme il se doit, saute au milieu du code de la bibliothèque C standard pour récupérer un mot de passe à partir d'un fichier dans le répertoire de base de l'utilisateur. Ça ne se passe pas bien. `main`

C et C++ ont des centaines de règles pour éviter un comportement indéfini. Ils sont surtout du bon sens: n'accédez pas à la mémoire que vous ne devriez pas, ne laissez pas les opérations arithmétiques déborder, ne divisez pas par zéro, etc. Mais le compilateur n'applique pas ces règles ; il n'a aucune obligation de détecter des violations, même flagrantes. En effet, le programme précédent compile sans erreurs ni avertissements. La responsabilité d'éviter un comportement indéfini incombe entièrement à vous, le programmeur.

Empiriquement parlant, nous, les programmeurs, n'avons pas un excellent bilan à cet égard. Alors qu'il était étudiant à l'Université de l'Utah,

le chercheur Peng Li a modifié les compilateurs C et C ++ pour que les programmes qu'ils traduisaient indiquent s'ils ont exécuté certaines formes de comportement indéfini. Il a constaté que presque tous les programmes le font, y compris ceux de projets très respectés qui maintiennent leur code à des normes élevées. Supposer que vous pouvez éviter un comportement indéfini en C et C ++, c'est comme supposer que vous pouvez gagner une partie d'échecs simplement parce que vous connaissez les règles.

Le message étrange occasionnel ou le crash peut être un problème de qualité, mais un comportement indéfini par inadvertance a également été une cause majeure de failles de sécurité depuis que le ver Morris de 1988 a utilisé une variante de la technique montrée précédemment pour se propager d'un ordinateur à un autre sur les débuts d'Internet.

C et C++ mettent donc les programmeurs dans une position délicate : ces langages sont les normes de l'industrie pour la programmation de systèmes, mais les exigences qu'ils imposent aux programmeurs garantissent pratiquement un flux constant de plantages et de problèmes de sécurité. Répondre à notre mystère soulève simplement une question plus importante : ne pouvons-nous pas faire mieux ?

Rust supporte la charge pour vous

Notre réponse est encadrée par nos trois citations d'ouverture. La troisième citation fait référence à des rapports selon lesquels Stuxnet, un ver informatique trouvé s'introduisant dans des équipements de contrôle industriel en 2010, a pris le contrôle des ordinateurs des victimes en utilisant, entre autres techniques, un comportement indéfini dans le code qui analysait les polices TrueType intégrées dans les documents de traitement de texte. Il y a fort à parier que les auteurs de ce code ne s'attendaient pas à ce qu'il soit utilisé de cette façon, illustrant que ce ne sont pas seulement les systèmes d'exploitation et les serveurs

qui doivent se soucier de la sécurité: tout logiciel qui pourrait gérer des données provenant d'une source non fiable pourrait être la cible d'un exploit.

Le langage Rust vous fait une promesse simple : si votre programme passe les vérifications du compilateur, il est exempt de comportement indéfini. Les pointeurs pendants, les doubles libres et les déréférencements de pointeur nul sont tous détectés au moment de la compilation. Les références de tableau sont sécurisées avec un mélange de vérifications au moment de la compilation et de l'exécution, de sorte qu'il n'y a pas de dépassements de tampon: l'équivalent Rust de notre malheureux programme C se ferme en toute sécurité avec un message d'erreur.

De plus, Rust vise à être à la fois *sûr* et *agréable à utiliser*. Afin de donner des garanties plus solides sur le comportement de votre programme, Rust impose plus de restrictions sur votre code que C et C ++, et ces restrictions nécessitent de la pratique et de l'expérience pour s'y habituer. Mais le langage dans son ensemble est flexible et expressif. Ceci est attesté par l'étendue du code écrit dans Rust et la gamme de domaines d'application auxquels il est appliqué.

D'après notre expérience, être capable de faire confiance à la langue pour attraper plus d'erreurs nous encourage à essayer des projets plus ambitieux. La modification de programmes volumineux et complexes est moins risquée lorsque vous savez que les problèmes de gestion de la mémoire et de validité du pointeur sont résolus. Et le débogage est beaucoup plus simple lorsque les conséquences potentielles d'un bogue n'incluent pas la corruption de parties non liées de votre programme.

Bien sûr, il y a encore beaucoup de bugs que Rust ne peut pas détecter. Mais dans la pratique, retirer un comportement indéfini de la table change considérablement le caractère du développement pour le mieux.

La programmation parallèle est apprivoisée

La concurrence d'accès est notoirement difficile à utiliser correctement en C et C++. Les développeurs se tournent généralement vers la concurrence uniquement lorsque le code monothread s'est avéré incapable d'atteindre les performances dont ils ont besoin. Mais la deuxième citation d'ouverture soutient que le parallélisme est trop important pour les machines modernes pour être traité comme une méthode de dernier recours.

Il s'avère que les mêmes restrictions qui garantissent la sécurité de la mémoire dans Rust garantissent également que les programmes Rust sont exempts de courses de données. Vous pouvez partager des données librement entre les threads, tant qu'elles ne changent pas. Les données qui changent ne sont accessibles qu'à l'aide de primitives de synchronisation. Tous les outils d'accès concurrentiel traditionnels sont disponibles : mutex, variables de condition, canaux, atomes, etc. Rust vérifie simplement que vous les utilisez correctement.

Cela fait de Rust un excellent langage pour exploiter les capacités des machines multicœurs modernes. L'écosystème Rust offre des bibliothèques qui vont au-delà des primitives d'accès concurrentiel habituelles et vous aident à répartir uniformément les charges complexes entre les pools de processeurs, à utiliser des mécanismes de synchronisation sans verrouillage tels que Lecture-Copie-Mise à jour, etc.

Et pourtant, la rouille est toujours rapide

Ceci, enfin, est notre première citation d'ouverture. Rust partage les ambitions que Bjarne Stroustrup articule pour C++ dans son article « Ab-

straction and the C++ Machine Model » :

En général, les implémentations C++ obéissent au principe zéro-surcharge : ce que vous n'utilisez pas, vous ne le payez pas. Et plus loin: ce que vous utilisez, vous ne pourriez pas mieux coder à la main.

La programmation des systèmes consiste souvent à pousser la machine à ses limites. Pour les jeux vidéo, toute la machine doit être consacrée à la création de la meilleure expérience pour le joueur. Pour les navigateurs Web, l'efficacité du navigateur fixe le plafond de ce que les auteurs de contenu peuvent faire. Dans les limites inhérentes à la machine, autant d'attention que possible sur la mémoire et le processeur doit être laissée au contenu lui-même. Le même principe s'applique aux systèmes d'exploitation : le noyau doit mettre les ressources de la machine à la disposition des programmes utilisateurs, et non les consommer lui-même.

Mais quand nous disons que Rust est « rapide », qu'est-ce que cela signifie vraiment? On peut écrire du code lent dans n'importe quel langage à usage général. Il serait plus précis de dire que, si vous êtes prêt à faire l'investissement pour concevoir votre programme afin de tirer le meilleur parti des capacités de la machine sous-jacente, Rust vous soutient dans cet effort. Le langage est conçu avec des valeurs par défaut efficaces et vous donne la possibilité de contrôler comment la mémoire est utilisée et comment l'attention du processeur est dépensée.

Rust facilite la collaboration

Nous avons caché une quatrième citation dans le titre de ce chapitre : « Les programmeurs de systèmes peuvent avoir de belles choses. » Cela fait référence à la prise en charge par Rust du partage et de la réutilisation du code.

Le gestionnaire de paquets et l'outil de construction de Rust, Cargo, facilitent l'utilisation des bibliothèques publiées par d'autres sur le référentiel public de paquets de Rust, le site [Web crates.io](https://crates.io). Il vous suffit d'ajouter le nom de la bibliothèque et le numéro de version requis à un fichier, et Cargo se charge de télécharger la bibliothèque, ainsi que toutes les autres bibliothèques qu'il utilise à son tour, et de lier le tout ensemble. Vous pouvez considérer Cargo comme la réponse de Rust à NPM ou RubyGems, en mettant l'accent sur la gestion des versions et les versions reproductibles. Il existe des bibliothèques Rust populaires fournissant tout, de la sérialisation standard aux clients et serveurs HTTP et aux API graphiques modernes.

Pour aller plus loin, le langage lui-même est également conçu pour prendre en charge la collaboration : les traits et les génériques de Rust vous permettent de créer des bibliothèques avec des interfaces flexibles afin qu'elles puissent servir dans de nombreux contextes différents. Et la bibliothèque standard de Rust fournit un ensemble de types fondamentaux qui établissent des conventions partagées pour des cas communs, ce qui rend différentes bibliothèques plus faciles à utiliser ensemble.

Le chapitre suivant vise à concrétiser les revendications générales que nous avons faites dans ce chapitre, avec une visite de plusieurs petits programmes Rust qui montrent les forces de la langue.

[Soutien](#) [Se déconnecter](#)