Developpez.com
Club des développeurs

Les erreurs du compilateur Rust les plus couramment rencontrées dans RustRover

Par Vitaly Bragilevsky - Delphine Massenhove (traducteur)

Date de publication : 4 mars 2024

Pour réagir au contenu de cet article, un espace de dialogue vous est proposé sur le forum.
Commentez

Introduction ... 3
I - Identification des erreurs les plus courantes à partir des données d'utilisation de RustOver – Partie 1 ... 3
I-1 - Erreur fréquente n° 10 : E0412 (un nom de type utilisé n'est pas dans la portée) ... 3
I-2 - Erreur fréquente n° 9 : E0061 (un nombre non valide d'arguments a été passé lors de l'appel d'une fonction) ... 4
I-3 - Erreur fréquente n° 8 : E0282 (le compilateur n'a pas pu déduire un type et a demandé une annotation de type) ... 6
I-4 - Erreur fréquente n° 7 : E0432 (une importation n'a pas été résolue) ... 7
I-5 - Erreur fréquente n° 6 : E0382 (une variable a été utilisée après que son contenu a été déplacé) ... 7
I-6 - Résumé ... 8
II - Identification des erreurs les plus courantes à partir des données d'utilisation de RustOver – Partie 2 ... 8
II-1 - Erreur fréquente n° 5 : E0433 (utilisation d'une crate, d'un module ou d'un type non déclaré) ... 8
II-2 - Erreur fréquente n° 4 : E0425 (un nom non résolu a été utilisé) ... 9
II-3 - Erreur fréquente n° 3 : E0599 (une méthode est utilisée sur un type qui ne l'implémente pas) ... 10
II-4 - Erreur fréquente n° 2 : E0308 (le type attendu ne correspondait pas au type reçu) ... 11
II-5 - Erreur fréquente n° 1 : E0277 (vous avez essayé d'utiliser un type qui n'implémente pas de trait à un endroit où un trait est attendu) ... 12
II-6 - Observation des erreurs Rust en général ... 12
II-7 - Résumé ... 15

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Introduction

Le compilateur Rust est une créature pointilleuse et exigeante. S'il est mécontent du code source, il peut renvoyer plus de 400 erreurs différentes (sans compter celles qui sont ajoutées chaque mois !). Certaines de ces erreurs sont extrêmement rares, tandis que d'autres compliquent le travail des développeurs Rust au quotidien. Dans la première partie de l'article, nous examinons une partie des messages d'erreur du compilateur Rust que les développeurs rencontrent le plus fréquemment dans RustRover, l'IDE de JetBrains conçu pour la programmation en Rust, et nous vous donnons des conseils pour éviter ces erreurs. Commençons par préciser ce que nous entendons par « erreurs les plus couramment rencontrées ».

I - Identification des erreurs les plus courantes à partir des données d'utilisation de RustOver – Partie 1

Tout utilisateur de RustRover peut donner son accord pour que JetBrains dispose de **ses données d'utilisation anonymisées**. Plus le nombre d'utilisateurs acceptant de partager leurs données d'utilisation est important et plus ils utilisent RustRover, plus nous sommes en mesure de comprendre leur expérience et d'améliorer les fonctionnalités d'assistance au codage de l'IDE. Nous profitons donc de cette occasion pour remercier toutes les personnes qui partagent leurs données d'utilisation ! La confidentialité des données étant une priorité, l'IDE ne collecte que des informations très limitées, qui ne permettent en aucun cas de remonter à l'utilisateur. Ces données anonymes nous permettent aussi d'identifier les types de messages d'erreur générés le plus fréquemment.

Lorsqu'un utilisateur ayant donné son consentement lance la commande Cargo Build depuis l'IDE (par exemple, en déclenchant une configuration Run qui nécessite la compilation d'un projet) et que le compilateur Rust signale une erreur, nous enregistrons un **code d'erreur**. Cela n'inclut pas tous les problèmes qui surviennent pendant que les utilisateurs écrivent leur code, mais seulement ceux qui persistent après qu'ils ont créé leur projet. Les erreurs intermédiaires peuvent souvent être corrigées à l'aide des inspections et des correctifs rapides fournis par l'IDE.

Nous avons collecté les codes d'erreur des utilisateurs de RustRover et avons classé les erreurs en fonction de leur fréquence. Cet article porte sur les erreurs figurant de la 10e à la 6e place et le prochain révélera les cinq erreurs les plus fréquentes. Nous allons examiner les causes de ces erreurs en utilisant des exemples simples et voir quels sont les moyens de les corriger.

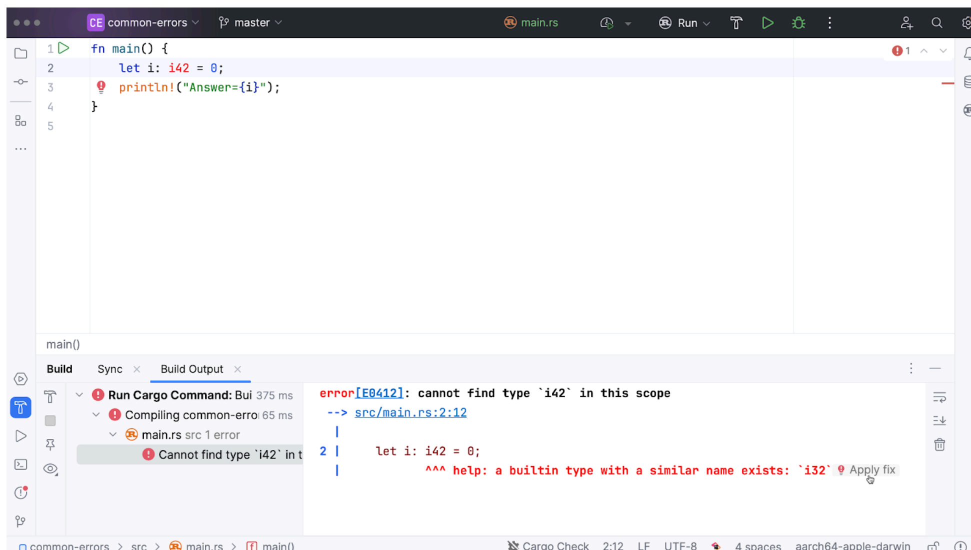
I-1 - Erreur fréquente n° 10 : E0412 (un nom de type utilisé n'est pas dans la portée)

Rust fait une distinction stricte entre les sites de déclaration de type et les utilisations de nom de type. Chaque nom de type (y compris les types génériques) doit être déclaré quelque part et être disponible dans la portée dans laquelle il est utilisé. Lorsque le compilateur rencontre une utilisation de nom de type et n'a pas d'information sur son site de déclaration, il émet l'erreur E0412. Environ 12 % des utilisateurs de RustRover ont déjà fait l'expérience de cette erreur.

Imaginons que vous ayez saisi `i42` au lieu de `i32`.

RustRover détecte ce problème et met en évidence le nom de type inconnu. Le compilateur donne plus de détails et propose un correctif qui peut être facilement appliqué en cliquant sur le bouton Apply fix dans la sortie du compilateur :

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.



Une erreur E0412 peut se produire dans d'autres situations, notamment :

- si l'on oublie de déclarer un type ;
- lorsqu'un type est importé dans la portée actuelle.

En cas d'introduction d'un nom de type générique qui rend le type inaccessible au compilateur.

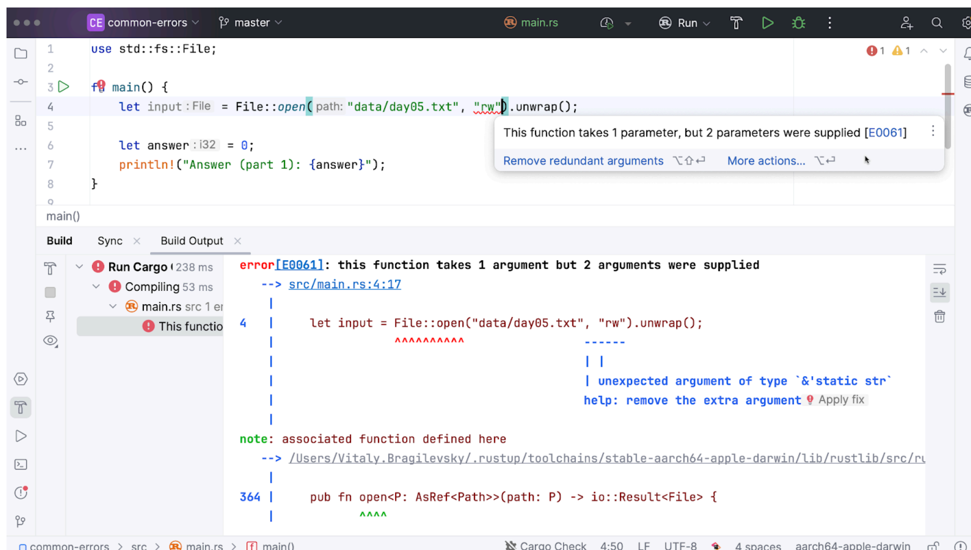
Pour résoudre le problème, vous pouvez fournir une déclaration de type (déclarer un *struct* ou introduire un nom de type générique correctement) ou amener le type dans une portée (via la clause `use`). **L'explication officielle de l'erreur E0412** donne plus d'exemples.

I-2 - Erreur fréquente n° 9 : E0061 (un nombre non valide d'arguments a été passé lors de l'appel d'une fonction)

Bien que RustRover connaisse cette erreur et fournisse plusieurs correctifs, 13 % des utilisateurs la laissent passer et omettent de la corriger avant la création de leurs projets.

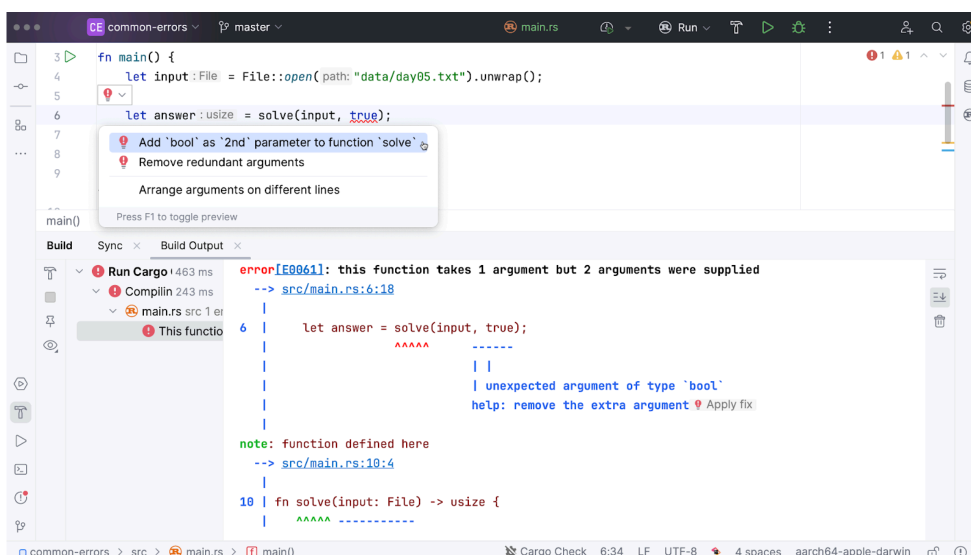
L'intitulé de cette erreur est assez explicite : une fonction est déclarée dans la portée actuelle ou importée depuis un autre endroit, et le site d'appel donne trop ou trop peu d'arguments. Voyons-en un exemple et comparons les suggestions de RustRover à celles du compilateur Rust :

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://jetbrains.developpez.com/tutoriel/analyse-erreurs-compilateur-rust-dans-rustrover/



Si l'on a l'habitude de programmer avec un autre langage, on pourrait être tenté de fournir un deuxième argument, en oubliant qu'en Rust, cette méthode n'en prend qu'un seul. RustRover et le compilateur Rust nous invitent donc à supprimer le second argument. Il est appréciable de bénéficier de cette suggestion de la part de l'IDE avant d'avoir créé le projet. Prêtez attention aux lignes de soulignement rouges ondulées dans votre code : généralement, elles sont là pour une bonne raison !

La situation devient plus intéressante si la fonction que nous appelons est définie dans notre propre code. Supposons que nous continuons de travailler sur le même exemple de code :



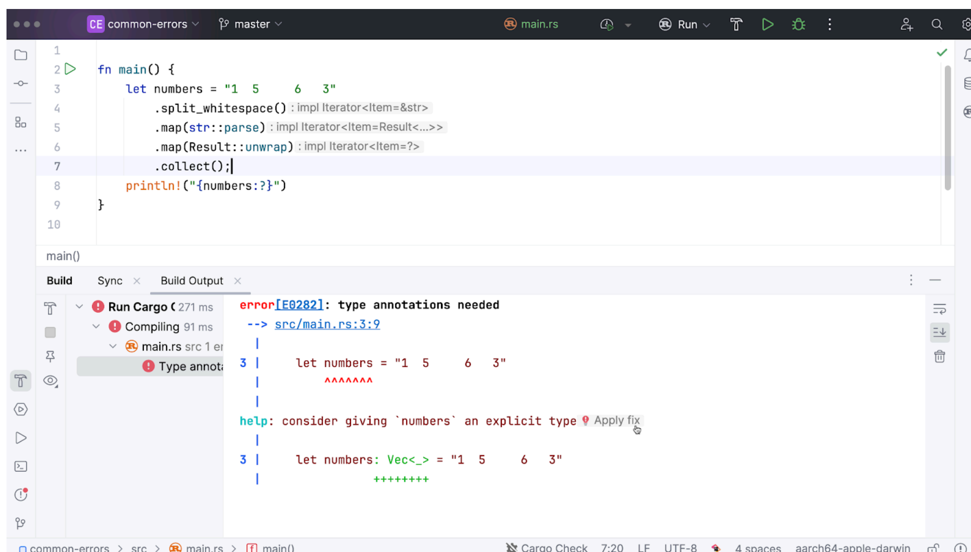
Dans ce cas, RustRover suggère comme première solution d'ajouter un paramètre à la fonction, ce qui peut faire sens. Le compilateur Rust insiste quant à lui pour le supprimer. Cette divergence a une explication logique. Le travail du compilateur consiste à s'assurer que le programme est correct, et le moyen le plus facile d'y parvenir consiste à éliminer l'argument supplémentaire sur le site de l'appel. En revanche, la mission de l'IDE est de vous accompagner vers la réalisation de votre objectif. Si vous avez saisi cet argument pour votre fonction, RustRover considère qu'il est probable que cela soit volontaire et tente donc de vous aider à terminer ce que vous avez commencé.

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://jetbrains.developpez.com/tutoriel/analyse-erreurs-compilateur-rust-dans-rustrover/

I-3 - Erreur fréquente n° 8 : E0282 (le compilateur n'a pas pu déduire un type et a demandé une annotation de type)

Parfois le compilateur est perdu : il ne trouve pas quel est le type requis pour une variable et ne peut que suggérer d'ajouter une annotation manuellement. Si vous avez déjà rencontré cette erreur, bienvenue au club : 13,5 % des utilisateurs de RustRover y ont déjà été confrontés.

La source principale des erreurs telles que E0282 est la généricité. De nombreuses fonctions de bibliothèques utilisent des paramètres de type générique, mais le compilateur doit instancier ces paramètres en type concret, ce qui est source de confusion pour lui. Prenons l'exemple suivant :



Ici nous devons collecter des nombres à partir d'une chaîne dans un conteneur. Malheureusement, le compilateur ne peut pas déterminer le type des nombres ou le genre de conteneur dont il s'agit.

Le compilateur suggère la spécification du type de conteneur en premier. Cependant, si nous appliquons ce correctif, nous obtiendrons de nouveau le même genre d'erreur, mais concernant `str::parse` cette fois. `collect` et `parse` sont des méthodes génériques, mais le compilateur a besoin de connaître le type précis afin de compiler le code qui les utilise. Veuillez noter que RustRover ne met pas les erreurs en évidence, car nous travaillons encore au perfectionnement de sa fonctionnalité de vérification de type.

Il existe plusieurs façons de corriger le problème, parce qu'une annotation de type peut être ajoutée à différents endroits. Nous pouvons spécifier le type concret du vecteur `numbers` :

`let numbers: let numbers: Vec = "1 5 6 3"`

Ou nous pouvons mentionner le même type lors de l'appel de `collect`:

Enfin, nous pouvons mentionner différents types à différents endroits :

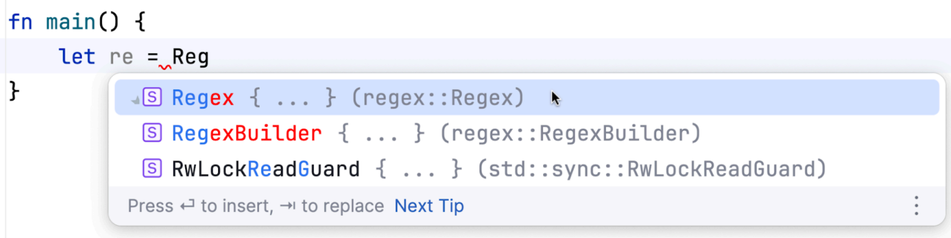
```
let numbers = GUILLEMET-KITOOODVP1  5      6    3GUILLEMET-KITOOODVP
    .split_whitespace()
    .map(str::parse::)
    .map(Result::unwrap)
    .collect::<Vec>();
```

Cette erreur est facile à corriger : il suffit de spécifier le type ou les types que vous voulez.

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://jetbrains.developpez.com/tutoriel/analyse-erreurs-compilateur-rust-dans-rustrover/

I-4 - Erreur fréquente n° 7 : E0432 (une importation n’a pas été résolue)

RustRover fournit de nombreuses fonctionnalités de saisie semi-automatique. Commençons par exemple en introduisant une expression régulière dans notre code :

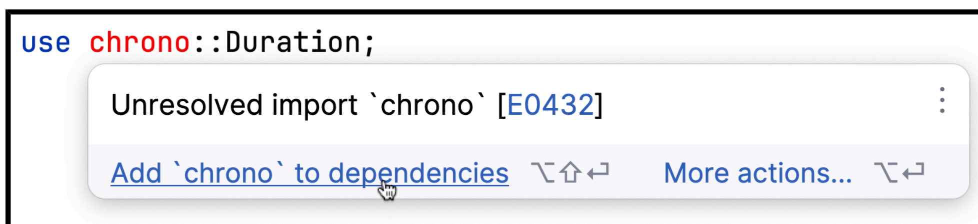


Si vous choisissez la première suggestion, deux autres choses vont se produire en plus de la saisie semi-automatique elle-même :

- une dépendance de la *crate* regex sera ajoutée à votre fichier *Cargo.toml ;*
- une clause *use* regex::Regex; sera ajoutée en haut du fichier.

Lorsque vous ajoutez des clauses d’utilisation comme cela, les importations sont écrites automatiquement et correctement. Mais il faut parfois écrire les importations manuellement et c’est là que l’erreur E0432 risque de se produire. 15,5 % des utilisateurs de RustRover la rencontrent occasionnellement, le plus souvent en cas de faute dans le nom d’une *crate* ou d’un module, de tentative d’importation de quelque chose d’inexistant ou de présence d’une clause use erronée dans le code après l’avoir copiée/collée à partir d’un autre endroit. La première suggestion est de toujours vérifier les dépendances et les noms.

Parfois, RustRover peut nous aider à éviter cette erreur. S’il a connaissance de la *crate* que nous tentons d’importer, il suggère l’ajout d’une dépendance lorsque vous collez du code depuis des sources externes ou fournit une assistance grâce aux correctifs rapides suivants :



L’ajout de la dépendance correspondante à *Cargo.toml* corrige l’erreur immédiatement. Une fois que la *crate* est disponible, il est plus sûr d’utiliser la saisie semi-automatique pour les autres composants du chemin dans les clauses use, afin d’éviter la survenance de nouveaux problèmes avec les noms. Rappelez-vous aussi que la disponibilité de certains noms peut dépendre des fonctionnalités de la *crate* qui sont activées.

Il peut également y avoir des problèmes avec les noms de chemin spéciaux tels que super ou crate, notamment parce qu’ils sont traités différemment dans les différentes versions de Rust. Vous trouverez plus d’informations à ce sujet dans l’**explication officielle**.

I-5 - Erreur fréquente n° 6 : E0382 (une variable a été utilisée après que son contenu a été déplacé)

Point intéressant : les problèmes liés à la possession ! 17 % des utilisateurs de RustRover ont déjà rencontré cette erreur. L’**explication officielle** est très détaillée et donne de nombreux exemples. Malheureusement, RustRover

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://jetbrains.developpez.com/tutoriel/analyse-erreurs-compilateur-rust-dans-rustrover/

n'est pas d'une grande aide dans ce cas. Si un **linter externe** est désactivé, le mécanisme interne de RustRover ne voit aucun problème dans le code suivant :

```
fn main() {
    let vec = vec![1, 2, 3, 4, 5];
    let mut sum = 0;
    for v in vec {
        sum += v;
    }
    println!(GUILLEMET-KITOOODVPSum of {vec:?} elements is {sum}GUILLEMET-KITOOODVP);
}
```

Ce code d'aspect anodin serait parfaitement légitime dans certains autres langages de programmation. Nous avons un vecteur et voulons calculer la somme de ses éléments. Si vous travaillez habituellement avec C et que vous ne connaissez pas les astuces de programmation liées aux itérateurs, vous écririez plutôt une boucle `for` traditionnelle à la place. Une fois les éléments ajoutés, il n'y aurait alors plus qu'à imprimer le vecteur et le résultat des calculs. Exact ?

Eh bien ce n'est pas le cas avec Rust à cause des règles de la possession propres au langage.

Le problème est que la source de données dans la boucle for est étendue à l'appel `into_iter()`, qui prend alors possession de l'ensemble du vecteur. Par conséquent, lorsque nous tenterons ensuite d'accéder aux éléments du vecteur dans `println!`, le compilateur indiquera qu'il a été déplacé.

Le correctif proposé par le navigateur est simple : itérer sur `&vec` pour éviter de le déplacer dans la boucle, puis l'emprunter.

En général, il est utile de conserver constamment une trace des règles de possession des valeurs. Le déplacement et l'emprunt de valeurs sont des concepts fondamentaux de Rust. Il est important pour les néophytes d'assimiler ces concepts dès le début.

I-6 - Résumé

Dans cette première partie, nous avons passé en revue certaines des erreurs du compilateur Rust les plus fréquentes en nous basant sur les données d'utilisation de RustRover. Dans la prochaine partie, nous verrons les 5 erreurs les plus fréquentes et tenterons de répondre à une question que se posent tous les développeurs Rust : « Quelle partie de Rust cause le plus de problèmes ? ».

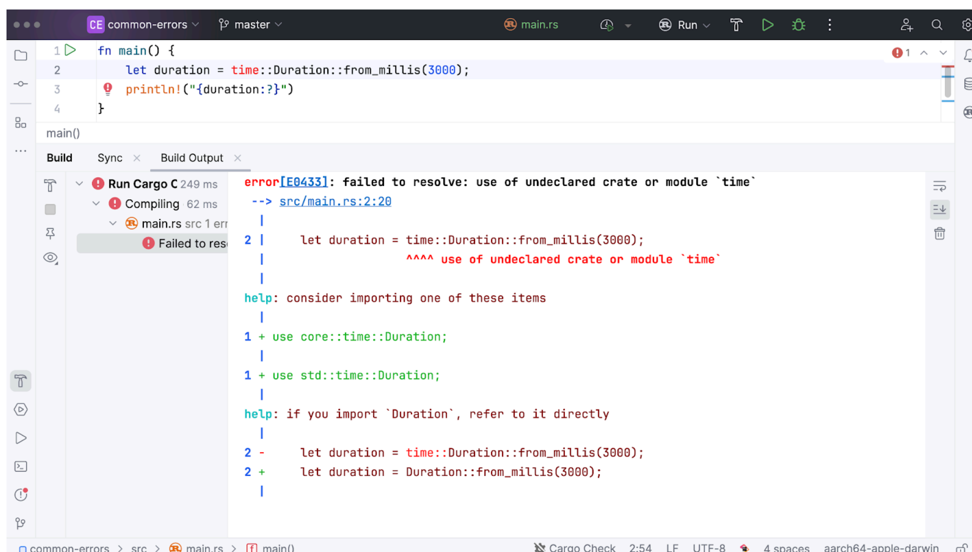
II - Identification des erreurs les plus courantes à partir des données d'utilisation de RustOver – Partie 2

Les données d'utilisation de RustRover nous ont permis d'identifier les 10 erreurs du compilateur Rust les plus fréquentes, que nous avons commencé à examiner dans **une première partie**. Dans cette seconde partie de l'article, nous poursuivons avec l'examen du top 5 de ces erreurs et abordons les aspects du langage Rust qui posent le plus de problèmes aux développeurs avec le compilateur.

II-1 - Erreur fréquente n° 5 : E0433 (utilisation d'une crate, d'un module ou d'un type non déclaré)

Cette erreur est semblable à l'erreur E0432 (une importation n'a pas été résolue), que nous avons vue dans l'article précédent. La seule différence ici est qu'un chemin avec un composant problématique est utilisé directement dans un nom, sans la clause `use`. 17,5 % des utilisateurs de RustRover ont rencontré cette erreur. Dans l'exemple de code ci-dessous, le compilateur ne comprend pas s'il s'agit d'une *crate* ou d'un module et nous le fait savoir :

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.



Cet exemple montre pourquoi les messages d'erreur de Rust sont autant appréciés. Même sans lire le message, le correctif à apporter est évident : vérifier les noms, ajouter les dépendances requises à *Cargo.toml* si nécessaire ou introduire les importations appropriées.

De son côté, RustRover permet de choisir les noms importés.

```
fn main() {
    let duration = time::Duration::from_millis(3000);
    println!("{duration
}
```

Import

Press F1 to toggle preview

```
fn main() {
    let duration = time::Duration::from_millis(3000);
    println!("{dura
}
```

Item to import

time (std) std

time (core) core

L'utilisation de la saisie semi-automatique de code permet d'éviter cette erreur.

II-2 - Erreur fréquente n° 4 : E0425 (un nom non résolu a été utilisé)

Poursuivons avec une autre erreur de la catégorie « non résolu ». 20,5 % des utilisateurs de RustRover ont déjà été dans une situation dans laquelle ils ont utilisé un nom non résolu, qui n'était syntaxiquement ni une *crate* ni un module. Voici ce que l'ont peut voir lorsque cette erreur survient :

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
fn main() {
    println!("{}", what_we_are_all_doing_here);
}
```

Cannot find value `what_we_are_all_doing_here` in this scope [E0425]

Évidemment, lorsque cela arrive, le contexte n'est pas toujours aussi simple que dans cet exemple.

Hélas, même l'excellent compilateur de Rust n'est pas d'une grande aide face à cette erreur. Plusieurs approches sont possibles ici : réviser le nom, fournir une définition ou introduire des importations correctes.

II-3 - Erreur fréquente n° 3 : E0599 (une méthode est utilisée sur un type qui ne l'implémente pas)

Au risque de gâcher la surprise, à partir de maintenant, toutes les erreurs porteront sur la vérification de type ! En effet, les trois erreurs les plus fréquentes sont toutes liées à une utilisation de type incorrecte. Concernant l'erreur n ° 3, 27,5 % des utilisateurs de RustRover ont tenté d'appeler une méthode sur une valeur de type qui n'implémente pas cette méthode. Examinons l'exemple suivant :

```
fn main() {
    let numbers: Vec = vec![4, 11, 15, 12];
    let sum: usize = numbers.sum();
    println!(GUILLEMET-KITOOODVPAnswer = {sum}GUILLEMET-KITOOODVP);
}
```

Tout semble correct, sauf qu'il n'y a pas de méthode `sum` pour `Vec`. Voici le contenu exact du message d'erreur :

`error[E0599]: `Vec` is not an iterator`

Dans ce cas, il faut appeler une méthode `iter` en premier, puis utiliser `sum` sur l'itérateur résultant.

RustRover dispose d'une fonctionnalité qui permet d'éviter ce genre d'erreurs : la saisie semi-automatique chaînée. Voici un exemple de ce que cela donne :

```
1  fn main() {
2      let numbers: Vec<usize> = vec![4, 11, 15, 12];
3      let sum: usize = numbers.sum
4      println!("Answer = {
5  }
6
```

iter().sum(self)   S
Press ^. to choose the s...Next Tip

Lorsque la suggestion de saisie semi-automatique est acceptée, les méthodes `iter` et `sum` sont ajoutées à la chaîne d'appels, ce qui corrige immédiatement le code.

En général, la saisie semi-automatique évite de recevoir un message d'erreur E0599. Si vous ne voyez pas de suggestion pour appeler la méthode dont vous avez besoin, il est possible que le problème réel provienne en réalité du fait que le type de l'appelé est incorrect.

Un autre aspect de cette méthode vaut la peine d'être mentionné : dans de nombreux contextes, vous avez le contrôle sur vos propres déclarations de type. Vous pouvez notamment appeler une méthode en premier et demander ensuite à RustRover de procéder au scaffolding de son implémentation en appuyant sur **Alt-Entrée** (⌥Option–**Retour)** et en exécutant l'action suggérée :

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
1 usage
struct Info {}

let info = Info {};
let outline: String = info.summary();
println!("{outline}");
```

Create method `summary`

Press F1 to toggle preview

Suite à cette action, RustRover génère le modèle d'implémentation de la méthode en prenant en compte les informations contextuelles concernant les types :

```
impl Info {
    pub(crate) fn summary(&self) -> String {
        todo!()
    }
}
```

L'exécution de ce code résulte en une erreur d'exécution, mais au moins le compilateur n'y trouve rien à redire.

II-4 - Erreur fréquente n° 2 : E0308 (le type attendu ne correspondait pas au type reçu)

Imaginez que vous utilisiez une expression de type **A** dans un contexte où une valeur de type **B** est attendue. Ici, la notion de « contexte » peut désigner plusieurs choses, comme un paramètre de fonction sur un site d'appel, une déclaration de variable ou une instruction/expression de contrôle du flux. 30 % des utilisateurs de RustRover ont déjà été dans cette situation.

D'un point de vue conceptuel, deux choix s'offrent à nous dans ce cas : modifier une valeur ou modifier le contexte. La modification d'une valeur peut impliquer le typage, le référencement/déréférencement, l'appel d'une méthode de transformation ou son remplacement par quelque chose d'autre. On ne peut pas toujours agir sur le contexte, mais il est parfois possible de modifier le type attendu pour qu'il corresponde au type reçu.

RustRover prend en charge les deux modes opératoires en fournissant plusieurs correctifs rapides, par exemple :

```
fn solve(input: usize) -> usize {
    todo!()
}

let input: &str = "100";
let answer: usize = solve(input);
```

Change type of parameter `input` in function solve to `&str`
Convert to usize using `FromStr` trait

Press F1 to toggle preview

```
14 let answer = solve(input.parse().unwrap());
```

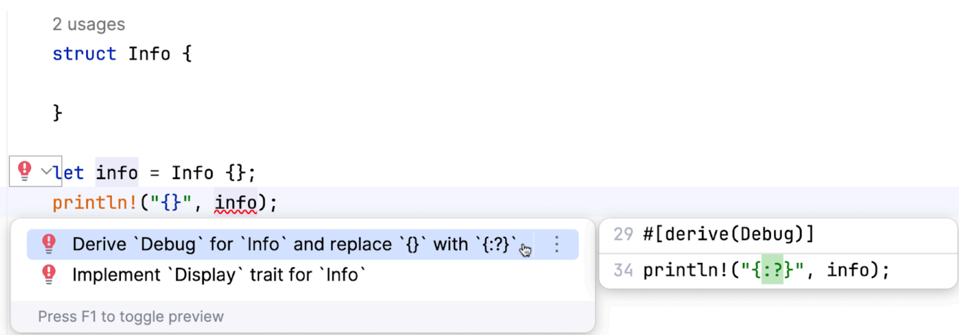
Notez la deuxième suggestion dans la liste : dans ce cas, RustRover a suggéré une transformation assez sophistiquée d'une valeur qui convient au compilateur. Suivre la première suggestion corrigerait le contexte, à savoir la définition de la fonction.

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

II-5 - Erreur fréquente n° 1 : E0277 (vous avez essayé d'utiliser un type qui n'implémente pas de trait à un endroit où un trait est attendu)

Enfin, nous arrivons à la 1re place du classement ! L'erreur du compilateur Rust la plus fréquemment rencontrée dans RustRover est E0277. 32 % des utilisateurs de RustRover ont fait l'expérience de cette situation cauchemardesque mêlant types et traits. L'**explication officielle** fait un bon travail en fournissant des exemples et des explications des correctifs possibles. Ici, nous allons observer le comportement de RustRover.

Voici ma fonctionnalité de RustRover préférée lorsqu'il s'agit de ce genre d'erreurs :



Le compilateur renvoie le message d'erreur suivant :

error[E0277]: `Info` doesn't implement `std::fmt::Display`

Bien sûr, l'implémentation du trait Display est un choix possible, et RustRover peut aider pour le scaffolding d'une implémentation. Mais dans ce cas, je préfère appliquer la première suggestion, qui comprend deux étapes simultanées :

- dérivation d'une implémentation pour le trait Debug.
- modification de la chaîne de format pour appeler le formatage du trait Debug.

Malheureusement, dans de nombreux autres cas, RustRover ne parvient pas à découvrir et à mettre en évidence cette erreur par lui-même. Comme mentionné dans l'article précédent, sa fonctionnalité de vérification de type n'est pas encore assez puissante, mais nous travaillons activement à son amélioration.

II-6 - Observation des erreurs Rust en général

Récemment, nous avons **fait un sondage** auprès des membres de la communauté afin de savoir quel est l'aspect du langage causant le plus d'erreurs dans leur code, et voici les réponses que nous avons obtenues :

Types and traits	17.9%
Ownership and lifetimes	**61.6%**
Macros	15.2%
Other (comment in thread)	5.4%

112 votes · Final results

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://jetbrains.developpez.com/tutoriel/analyse-erreurs-compilateur-rust-dans-rustrover/

Afin de comparer les réponses de la communauté à nos données, nous avons examiné les 25 erreurs de compilateur Rust les plus fréquentes et les avons classées en cinq grandes catégories :

- types et traits
- propriétés et durées de vie
- macros
- noms non résolus ou éléments inexistants
- autres

Voici les résultats (numéros de code d'erreur classés par ordre croissant) :

Code de l'erreur	Description	Catégorie
E0061	Un nombre non valide d'arguments a été passé lors de l'appel d'une fonction.	Autre
E0063	Aucun champ de structure ou de variante d'énumération de type structure n'a été fourni.	non résolu/non existant
E0106	Cette erreur indique que l'un des types ne comporte pas de durée de vie.	propriété et durées de vie
E0107	Un nombre incorrect d'arguments génériques a été fourni.	types et traits
E0277	Vous avez essayé d'utiliser un type qui n'implémente pas un trait à un endroit où un trait est attendu.	types et traits
E0282	Le compilateur n'a pas pu déduire un type et a demandé une annotation de type.	types et traits
E0283	Une implémentation ne peut pas être choisie de façon claire en raison du manque d'informations.	types et traits
E0308	Le type attendu ne correspond pas au type reçu.	types et traits
E0369	Une opération binaire a été tentée sur un type qui ne la prend pas en charge.	types et traits
E0382	Une variable a été utilisée après que son contenu a été déplacé.	propriété et durées de vie
E0412	Un nom de type utilisé n'est pas dans la portée.	non résolu/non existant
E0423	Un identifiant a été utilisé comme un nom de fonction ou une valeur était attendue et l'identifiant existe, mais appartient	non résolu/non existant

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

	à un espace de nom différent.	
E0425	Un nom non résolu a été utilisé.	non résolu/non existant
E0432	Une importation n'a pas été résolue.	non résolu/non existant
E0433	Une *crate*, un module ou un type non déclaré a été utilisé.	non résolu/non existant
E0502	Une variable déjà empruntée comme immuable a été empruntée comme mutable.	propriété et durées de vie
E0507	Une valeur empruntée a été déplacée.	propriété et durées de vie
E0515	Une référence à une variable locale a été renvoyée.	propriété et durées de vie
E0596	Cette erreur se produit parce que vous avez tenté d'emprunter de façon mutable une variable non mutable.	propriété et durées de vie
E0597	Cette erreur se produit, car une valeur a été supprimée alors qu'elle était encore empruntée.	propriété et durées de vie
E0599	Cette erreur se produit lorsqu'une méthode est utilisée pour un type qui ne l'implémente pas.	types et traits
E0609	Tentative d'accès à un champ non existant dans une structure.	non résolu/non existant
E0614	Tentative de déférencement d'une variable qui ne peut pas être déférencée.	Autre
E0658	Une fonctionnalité instable a été utilisée.	Autre
E0716	Une valeur temporaire est supprimée alors qu'un emprunt est toujours actif.	propriété et durées de vie

Malheureusement, les données dont nous disposons ne donnent pas beaucoup d'informations sur les macros. Nous ne pouvons ni détecter de façon fiable les problèmes d'expansion de macros, ni identifier d'autres erreurs résultant d'expansions de macros réussies. Cela montre que nous pourrions appliquer une catégorisation plus fine aux données que nous collectons.

En faisant abstraction des macros, aucune des 25 erreurs les plus fréquentes ne se détache véritablement, il ne semble donc pas qu'il y ait un aspect de Rust en particulier qui pose davantage problème aux développeurs :

- types et traits – 7 erreurs
- propriété et durées de vie – 8 erreurs
- noms non résolus ou éléments inexistants – 7 erreurs

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://jetbrains.developpez.com/tutoriel/analyse-erreurs-compilateur-rust-dans-rustrover/

- autres – 3 erreurs

II-7 - Résumé

Dans ce second article, nous avons passé en revue les cinq erreurs du compilateur Rust les plus fréquemment rencontrées dans RustRover et les correctifs possibles. Trois d'entre elles se rapportent aux types et traits, ce dont on peut déduire que ces éléments ont un rôle significatif dans l'obtention d'un code Rust de meilleure qualité. En examinant des données plus générales relatives aux 25 erreurs les plus fréquentes, on voit aussi que bien d'autres aspects de Rust peuvent contribuer aux erreurs que nous rencontrons souvent.

Copyright ® 2024 Vitaly Bragilevsky. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
https://jetbrains.developpez.com/tutoriel/analyse-erreurs-compilateur-rust-dans-rustrover/