L'engagement de **Rust** pour la fiabilité concerne aussi la gestion des erreurs. Les erreurs font partie de la vie des programmes informatiques, c'est pourquoi **Rust** propose des fonctionnalités pour gérer les situations dans lesquelles quelque chose dérape. Dans de nombreux cas, **Rust** exige que vous anticipiez les erreurs possibles et que vous preniez des dispositions avant de pouvoir compiler votre code. Cette exigence rend votre programme plus résiliant en s'assurant que vous détectiez et gérez les erreurs correctement avant même que vous ne déployez votre code en production !

__Rust__ classe les erreurs dans deux catégories principales : les erreurs récupérables et irrécupérables. Les erreurs récupérables, comme lorsque un fichier n'est pas trouvable, il est préférable de signaler le problème à l'utilisateur et de relancer l'opération. Les erreurs irrécupérables sont toujours des symptômes de bogues, comme essayer d'accéder à un élément en dehors de l'intervalle de données d'un tableau.

La plupart des langages de programmation ne font pas de distinction entre ces deux types d'erreurs et les gèrent de la même manière, en utilisant des fonctionnalités comme les exceptions. __Rust n'a pas d'exception__. À la place, il propose les types __Result<T, E>__ pour les erreurs récupérables, et la macro __panic!()__ qui arrête l'exécution quand le programme se heurte à des erreurs irrécupérables.

Les erreurs irrécupérables avec panic!

Parfois, des choses se passent mal dans votre code, et vous ne pouvez rien y faire. Pour ces cas-là, **Rust** possède la macro **panic!()**. Quand la macro **panic!()** s'exécute, votre programme va afficher un message d'erreur, dérouler et nettoyer la pile, et ensuite fermer le programme.

Cela se produit fréquemment lorsqu'un bogue a été détecté, et que le développeur ne sais pas comment gérer cette erreur. Essayons d'appeler panic!() dans un programme simple et visualisons le résultat en conséquence.

Panique à bord !

```
fn main() {
    panic!("Panique à bord!");
}
```

Résultat

```
thread 'main' panicked at 'Panique à bord!', src/main.rs:2:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

L'utilisation de __panic!()__ déclenche le message d'erreur présent dans les deux dernières lignes. La première ligne affiche notre message associé au panic et l'emplacement dans notre code source où se produit le panic : __src/main.rs:2:5__ indique que c'est la seconde ligne et cinquième caractère de notre fichier __src/main.rs__.

Utiliser le re-traçage de panic!()

Analysons maintenant un autre exemple pour voir ce qui se passe lors d'un appel de **panic!()** qui se produit dans une bibliothèque prédéfinie à cause d'un bug dans notre code plutôt qu'un appel à la macro directement.

Re-traçage

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

Résultat

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', src/main.rs:4:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Ici, nous essayons d'accéder au centième élément de notre vecteur, mais le vecteur possède seulement trois éléments. Dans ce cas, __Rust__ va paniquer. Utiliser __[ ]__ est censé retourner un élément, mais si vous lui donnez un indice invalide, __Rust__ ne pourra pas retourner un élément acceptable dans ce cas.

En __C++__, tenter de lire en dehors de la fin d'une structure de donnée suit un comportement non défini. Vous pourriez récupérer quelque chose à l'emplacement mémoire demandé qui pourrait correspondre à l'élément demandé de la structure de données, même si cette partie de la mémoire ne lui appartient pas.

Afin de protéger votre programme de ce genre de vulnérabilité, si vous essayez de lire un élément à un indice qui n'existe pas, Rust arrête immédiatement l'exécution du programme et refuse de continuer.

Des erreurs récupérables avec Result

La plupart des erreurs ne sont pas assez graves au point d'arrêter complètement le programme. Parfois, lorsque une fonction échoue, c'est pour une raison que vous pouvez facilement comprendre et agir en conséquence. Par exemple, si vous essayez d'ouvrir un fichier et que l'opération échoue, c'est certainement parce que le fichier n'existe pas, vous pourriez alors vouloir créer le fichier plutôt que d'arrêter prématurément le processus.

Le __T__ et __E__ sont des paramètres de type génériques. __T__ représente le type de valeur imbriquée dans la variante __Ok__ qui sera retournée en cas de succès, et __E__ représente le type d'erreur imbriquée dans la variante __Err__ qui sera retournée en cas d'échec. Comme __Result__ propose ces types de paramètres génériques, nous pouvons utiliser le type __Result__ et les fonctions

que la bibliothèque standard qui lui ont été associées pour différentes situations où la valeur de succès et la valeur d'erreur peuvent être différentes.

Énumération Result

```
enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

*À titre d'exemple je vous propose de faire à la fonction **File::open()** qui peut réussir et nous retourner un manipulateur de fichier qui peut nous permettre de le lire ou l'écrire. L'utilisation de cette fonction peut aussi échouer : par exemple, le fichier peut ne pas exister, ou nous n'avons pas le droit d'accéder au fichier.*

*La fonction **File::open()** doit avoir un moyen de nous dire si son utilisation a réussi ou échoué et en même temps nous fournir soit le manipulateur de fichier soit des informations sur l'erreur. C'est exactement ces informations que l'énumération **Result** se charge de nous transmettre.*

Ouverture d'un fichier

```
use std::fs::File;

fn main() {
    let f = match File::open("bienvenue.txt") {
       Ok(fichier) => fichier,
       Err(erreur) => panic!("Erreur lors de l'ouverture du fichier : {:?}", erreur),
    };
}
```

Résultat

```
thread 'main' panicked at 'Erreur lors de l'ouverture du fichier :
Os { code: 2, kind: NotFound, message: "No such file or directory" }', src/main.rs:6:24
```

*Remarquez que comme l'énumération **Option**, l'énumération **Result** et ses variantes ont été importés par l'étape préliminaire, donc vous n'avez pas besoin de préciser **Result::** devant les variantes **Ok** et **Err** dans les branches du **match**.*

*Ici nous indiquons à **Rust** que lorsque le résultat est **Ok**, il faut sortir la valeur **fichier** de la variante **Ok**, et nous assignons ensuite cette valeur à la variable **f**. Après le **match**, nous pourrons utiliser le manipulateur de fichier pour lire ou écrire.*

*L'autre branche du bloc **match** gère le cas où nous obtenons un **Err** à l'appel de **File::open()**. Dans cet exemple, nous avons choisi de faire appel à la macro **panic!()**. S'il n'existe pas de fichier « **bienvenue.txt** » dans notre répertoire actuel et que nous exécutons ce code, nous obtenons le résultat ci-dessus.*

Tester les différentes erreurs

Le code précédent nous explique le déroulement du programme, mais il n'est pas très fonctionnel. Ce que nous voudrions plutôt faire est de réagir différemment en fonction de différents cas d'erreurs : si **File::open()** échoue parce que le fichier n'existe pas, nous voulons créer le fichier et renvoyer le manipulateur de fichier pour ce nouveau fichier.

*Si **File::open()** échoue pour une toute autre raison, par exemple si nous n'avons pas l'autorisation d'ouvrir le fichier, nous voulons quand même que le code lance un **panic!()** afin d'être renseigné sur le problème de lecture.*

Création d'un fichier s'il n'existe pas

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = match File::open("bienvenue.txt") {
       Ok(fichier) => fichier,
       Err(erreur) => match erreur.kind() {
          ErrorKind::NotFound => match File::create("bienvenue.txt") {
             Ok(fc) => fc,
             Err(e) => panic!("Erreur lors de la création du fichier : {:?}", e),
          },
          autre_erreur => panic!("Erreur lors de l'ouverture du fichier : {:?}", autre_erreur),
       },
    };
}
```

*La valeur de retour de **File::open()** logée dans la variante **Err** est de type **io::Error**, qui est une structure fournie par la bibliothèque standard. Cette structure possède une méthode **kind()** que nous pouvons utiliser pour obtenir un retour de type **io::ErrorKind**.*

*L'énumération **io::ErrorKind** est fournie elle aussi par la bibliothèque standard qui embarque des variantes qui représentent différents types d'erreurs qui pourraient résulter d'une opération provenant du module **io**. La variante que nous voulons utiliser est **ErrorKind::NotFound**, qui nous informe que le fichier que nous essayons d'ouvrir n'existe pas encore. Donc nous utilisons **match** sur **f**, mais nous avons dans celle-ci une autre **match** sur **erreur.kind()**.*

*Nous souhaitons vérifier dans le **match** interne si la valeur de retour de **error.kind()** est la variante **NotFound** de l'énumération **ErrorKind**. Si c'est le cas, nous essayons de créer le fichier avec **File::create()**. Cependant, comme **File::create()** peut aussi échouer, nous avons besoin d'une seconde branche dans le **match** à l'intérieur.*

*Lorsque le fichier ne peut pas être créé, un message d'erreur différent est affiché. La seconde branche du **match** principal reste inchangé, donc le programme panique lorsqu'on rencontre une autre erreur que l'absence de fichier.*

Raccourci pour faire un Panic lors d'une erreur : unwrap et expect

L'utilisation de **match** fonctionne assez bien, mais elle peut être un peu verbeuse et ne communique pas forcément bien son intention. Le type **Result<T, R>** a de nombreuses méthodes qui lui ont été définies pour différents cas. Une de ces méthodes, qui s'appelle **unwrap()**, a été implémentée comme l'expression **match** que nous avons écrit précédemment.

*Si la valeur de **Result** est une variante de **Ok**, **unwrap()** retourne la valeur dans le **Ok**. Si le **Result** est une variante de **Err**, **unwrap()** appelle automatiquement la macro **panic!()** pour nous. Voici un exemple de **unwrap()** à l'action :*

Ouverture d'un fichier

```
use std::fs::File;

fn main() {
    let f = File::open("bienvenue.txt").unwrap();
}
```

Résultat

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value:
Os { code: 2, kind: NotFound, message: "No such file or directory" }', src/main.rs:4:41
```

*L'autre méthode, **expect()**, qui ressemble à **unwrap()**, nous donne la possibilité de définir le message d'erreur du **panic!()**. Utiliser **expect()** plutôt que **unwrap()** et lui fournir un bon message d'erreur permet de mieux exprimer le problème et faciliter la recherche de la source d'erreur. La syntaxe de **expect()** est la suivante :*

Ouverture d'un fichier

```
use std::fs::File;

fn main() {
    let f = File::open("bienvenue.txt").expect("Echec à l'ouverture de bienvenue.txt");
}
```

Résultat

```
thread 'main' panicked at 'Echec à l'ouverture de bienvenue.txt:
Os { code: 2, kind: NotFound, message: "No such file or directory" }', src/main.rs:4:41
```

*Nous utilisons **expect()** de la même manière que **unwrap()** : pour retourner le manipulateur de fichier ou appeler la macro **panic!()**. Le message d'erreur utilisé par **expect()** lors de son appel au **panic!()** sera le paramètre que nous avons passé à **expect()**, plutôt que le message par défaut de **panic!()** qu'utilise **unwrap()**.*

*Comme ce message d'erreur commence par le texte que nous avons précisé, « **Echec à l'ouverture de hello.txt** », ce sera plus facile de trouver où se situe ce message d'erreur dans le code. Si nous utilisons **unwrap()** à plusieurs endroits, cela peut prendre plus de temps de comprendre exactement quel **unwrap()** a causé la panique, car tous les appels aux **unwrap()** vont afficher le même message.*

Propager les Erreurs

Lorsque vous écrivez une fonction dont l'implémentation utilise quelque chose qui peut échouer, au lieu de gérer l'erreur dans cette fonction, vous pouvez retourner cette erreur au code qui l'appelle pour qu'il décide quoi faire. C'est ce que nous appelons propager l'erreur et donne ainsi plus de contrôle au code qui appelle la fonction, dans lequel il peut y avoir plus d'informations ou d'instructions pour traiter l'erreur que dans le contexte de votre code.

Par exemple, l'exemple ci-dessus montre une fonction qui lit le nom d'utilisateur à partir d'un fichier. Si ce fichier n'existe pas ou ne peut pas être lu, cette fonction retourne ces erreurs au code qui appelle cette fonction.

bienvenue.txt

```
Bonjour à tous !
```

Récupérer le contenu d'un fichier

```
use std::fs::File;
use std::io;
use std::io::Read;

fn lire_contenu_du_fichier(nomFichier: &str) -> Result<String, io::Error> {
    let mut fichier = match File::open(nomFichier) {
        Ok(fichier) => fichier,
        Err(erreur) => return Err(erreur)
    };
```

```
    let mut chaine = String::new();

    match fichier.read_to_string(&mut chaine) {
        Ok(_) => Ok(chaine),
        Err(erreur) => Err(erreur)
    }
}

fn main() {
    let nom = match lire_contenu_du_fichier("bienvenue.txt") {
        Ok(nom) => nom,
        Err(_) => panic!("Ce fichier n'existe pas ou n'est pas accessible.")
    };
    println!("{}", nom);
}
```

Résultat

```
Bonjour à tous !
```

Récupérer le contenu d'un fichier

```
...
fn main() {
    let nom = match lire_contenu_du_fichier("bonjour.txt") {
        Ok(nom) => nom,
        Err(_) => panic!("Ce fichier n'existe pas ou n'est pas accessible.")
    };
    println!("{}", nom);
}
```

Résultat

```
thread 'main' panicked at 'Ce fichier n'existe pas ou n'est pas accessible.'
```

Cette fonction peut être écrite de façon plus concise, mais nous avons décidé de commencer par faire un maximum de choses manuellement pour découvrir la gestion d'erreurs ; mais à la fin, nous verrons comment raccourcir le code.

*Commençons par regarder le type de retour de la fonction : **Result<String, io::Error>**. Cela signifie que la fonction retourne une valeur de type **Result<T, E>** où le paramètre générique **T** a été remplacé par le type **String** et le paramètre générique **E** a été remplacé par le type **io::Error**.*

*Si cette fonction réussit avec succès, le code qui appelle cette fonction va obtenir une valeur **Ok** qui contient un **String**, le nom d'utilisateur que cette fonction lit dans le fichier. Si cette fonction rencontre un problème, le code qui appelle cette fonction va obtenir une valeur **Err** qui contient une instance de **io::Error** qui donne plus d'informations sur la raison du problème.*

*Nous avons choisi **io::Error** comme type de retour de cette fonction parce qu'il se trouve que c'est le type d'erreur de retour pour toutes les opérations qui peuvent échouer que nous utilisons dans le corps de cette fonction : la fonction **File::open()** et la méthode **read_to_string()**.*

*Le corps de la fonction commence par appeler la fonction **File::open()**. Ensuite, nous gérons la valeur **Result** retourné, avec un **match**, mais, au lieu d'appeler **panic!()** dans le cas de **Err**, nous retournons prématurément le résultat de la fonction avec la valeur d'erreur de **File::open()** au code appelant avec la valeur d'erreur de cette fonction. Si **File::open()** réussit, nous enregistrons le manipulateur de fichier dans la variable **fichier** et nous continuons.*

*Ensuite, nous créons un nouveau **String** dans la variable **chaine** et nous appelons ensuite la méthode **read_to_string()** sur le manipulateur de fichier **fichier** pour extraire le contenu du fichier dans **chaine**. La méthode **read_to_string()** retourne aussi un **Result** car elle peut échouer, même si **File::open()** réussit.*

*Nous avons donc besoin d'un nouveau **match** pour gérer ce **Result** : si **read_to_string()** réussit, alors notre fonction a réussi, et nous retournons le nom d'utilisateur présent dans le contenu du fichier qui est maintenant intégré dans un **Ok**, lui-même stocké dans **chaine**.*

*Si **read_to_string()** échoue, nous retournons la valeur d'erreur de la même façon que nous avons retourné la valeur d'erreur dans le **match** qui gérait la valeur de retour de **File::open()**. Cependant, nous n'avons pas besoin d'écrire explicitement **return**, car c'est la dernière expression de la fonction.*

Grâce à ce mécanisme, c'est la fonction main() qui gère elle-même l'erreur possible ou récupère le contenu du bon fichier proposé en argument de cette fonction lire_contenu_du_fichier().

Un raccourci pour propager les erreurs : l'opérateur ?

Le **?** placé après une valeur **Result** est conçu pour fonctionner presque de la même manière que les expressions **match** que nous avons d'expérimenter. Si la valeur du **Result** est un **Ok**, la valeur dans **Ok** sera retournée par cette expression et le programme continuera. Si la valeur est une **Err**, l'erreur sera retournée par la fonction comme si nous avions utilisé le mot-clé **return** afin que la valeur d'erreur soit propagée au code appelant.

*À la fin de l'appel à **File::open()** le **?** retourne la valeur à l'intérieur d'un **Ok** associée à la variable **fichier**. Si une erreur se produit, l'opérateur **?** retourne prématurément la fonction et fournir une valeur **Err** au code appelant. La même chose se produira au **?** à la fin de l'appel à **read_to_string()**.*

Récupérer le contenu d'un fichier

```
use std::fs::File;
use std::io;
use std::io::Read;

fn lire_contenu_du_fichier(nomFichier: &str) -> Result<String, io::Error> {
    let mut fichier = File::open(nomFichier)?;
    let mut chaine = String::new();
    fichier.read_to_string(&mut chaine)?;
    Ok(chaine)
}

fn main() {
    let nom = match lire_contenu_du_fichier("bienvenue.txt") {
        Ok(nom) => nom,
        Err(_) => panic!("Ce fichier n'existe pas ou n'est pas accessible.")
    };
    println!("{}", nom);
}
```

Résultat

```
Bonjour à tous !
```

*L'opérateur **?** épargne de l'écriture de code et facilite l'implémentation de la fonction. Nous pouvons même encore plus réduire ce code en enchaînant immédiatement les appels aux méthodes après le **?** comme dans l'exemple qui suit :*

Récupérer le contenu d'un fichier

```
use std::fs::File;
use std::io;
use std::io::Read;

fn lire_contenu_du_fichier(nomFichier: &str) -> Result<String, io::Error> {
    let mut chaine = String::new();
    File::open(nomFichier)?.read_to_string(&mut chaine)?;
    Ok(chaine)
}

fn main() {
    let nom = match lire_contenu_du_fichier("bienvenue.txt") {
        Ok(nom) => nom,
        Err(_) => panic!("Ce fichier n'existe pas ou n'est pas accessible.")
    };
    println!("{}", nom);
}
```

Résultat

```
Bonjour à tous !
```

*Nous avons déplacé la création de la nouvelle **String** dans **chaine** au début de la fonction ; cette partie n'a pas changée. Au lieu de créer la variable **fichier**, nous enchaînons directement l'appel à **read_to_string()** sur le résultat de File::open("bienvenue.txt")?.*

*Nous conservons toujours le **?** à la fin de l'appel à **read_to_string()**, et nous retournons toujours une valeur **Ok** contenant le contenu du fichier dans **chaine** lorsque **File::open()** et **read_to_string()** réussissent toutes les deux plutôt que de retourner des erreurs.*

En parlant de différentes façons d'implémenter cette fonction, le code qui suit nous montre qu'il existe encore une autre façon d'écrire avec encore moins de code.

Récupérer le contenu d'un fichier

```
use std::fs;
use std::io;

fn lire_contenu_du_fichier(nomFichier: &str) -> Result<String, io::Error> {
    fs::read_to_string(nomFichier)
}

fn main() {
    let nom = match lire_contenu_du_fichier("bienvenue.txt") {
        Ok(nom) => nom,
        Err(_) => panic!("Ce fichier n'existe pas ou n'est pas accessible.")
    };
    println!("{}", nom);
}
```

Résultat

```
Bonjour à tous !
```

*Récupérer le contenu d'un fichier dans une chaîne **String** est une opération très courante, donc **Rust** fournir la fonction **fs::read_to_string()** assez pratique, qui ouvre le fichier, crée une nouvelle chaîne **String**, lit de contenu du fichier, envoie le contenu dans cette chaîne **String**, et la retourne.*

Contrôler la saisie clavier

Grâce à cette technique, nous pouvons contrôler la saisie clavier afin de vérifier qu'une valeur numérique est bien introduite, sachant que normalement la saisie renvoie systématiquement une chaîne de caractères. Reprenons le dernier projet élaboré pour calculer un nombre **factoriel** sachant que le nombre à calculer ne devra jamais dépasser la valeur **34**.

*À la fin de l'appel à **File::open()** le **?** retourne la valeur à l'intérieur d'un **Ok** associée à la variable **fichier**. Si une erreur se produit, l'opérateur **?** retourne prématurément la fonction et fournir une valeur **Err** au code appelant. La même chose se produira au **?** à la fin de l'appel à **read_to_string()**.*

Récupérer le contenu d'un fichier

```
use std::io::*;

fn saisie() -> u128 {
    loop {
        let mut saisie = String::new();
        print!("Factoriel de : ");
        stdout().flush().unwrap();
        stdin().read_line(&mut saisie).unwrap();
        match saisie.trim().parse() {
            Ok(x @ 0...34) => return x,
            Ok(_) => {
                println!("Votre nombre est trop grand !");
                continue;
            },
            Err(_) => {
                println!("Il faut une valeur numérique !");
                continue;
            }
        }
    }
}

fn main() {
    let nombre = saisie();
    let mut resultat: u128 = 1;
    for n in 2..nombre+1 {
        resultat *= n;
    }
    println!("{}! = {}", nombre, resultat);
}
```

Résultat

```
Factoriel de : fact
Il faut une valeur numérique !
Factoriel de : 35
Votre nombre est trop grand !
Factoriel de : 18
18! = 6402373705728000
```