Chapitre 2. Un tour de Rust

Rust présente aux auteurs d'un livre comme celui-ci un défi: ce qui donne à la langue son caractère n'est pas une caractéristique spécifique et étonnante que nous pouvons montrer sur la première page, mais plutôt la façon dont toutes ses parties sont conçues pour fonctionner ensemble en douceur au service des objectifs que nous avons exposés dans le dernier chapitre: programmation de systèmes sûrs et performants. Chaque partie de la langue est mieux justifiée dans le contexte de tout le reste.

Ainsi, plutôt que d'aborder une fonctionnalité de langue à la fois, nous avons préparé une visite de quelques programmes petits mais complets, dont chacun introduit d'autres fonctionnalités de la langue, dans le contexte:

- En guise d'échauffement, nous avons un programme qui effectue un calcul simple sur ses arguments de ligne de commande, avec des tests unitaires. Cela montre les types de base de Rust et introduit des *traits*.
- Ensuite, nous construisons un serveur Web. Nous utiliserons une bibliothèque tierce pour gérer les détails de HTTP et introduire la gestion des chaînes, les fermetures et la gestion des erreurs.
- Notre troisième programme trace une belle fractale, répartissant le calcul sur plusieurs threads pour la vitesse. Cela inclut un exemple de fonction générique, illustre comment gérer quelque chose comme un tampon de pixels et montre la prise en charge de la concurrence par Rust.
- Enfin, nous montrons un outil de ligne de commande robuste qui traite les fichiers à l'aide d'expressions régulières. Cela présente les fonctionnalités de la bibliothèque standard Rust pour travailler avec des fichiers et la bibliothèque d'expressions régulières tierce la plus couramment utilisée.

La promesse de Rust d'empêcher les comportements indéfinis avec un impact minimal sur les performances influence la conception de chaque partie du système, des structures de données standard telles que les vecteurs et les chaînes à la façon dont les programmes Rust utilisent des bibliothèques tierces. Les détails de la façon dont cela est géré sont couverts tout au long du livre. Mais pour l'instant, nous voulons vous montrer que Rust est un langage capable et agréable à utiliser.

Tout d'abord, bien sûr, vous devez installer Rust sur votre ordinateur.

rustup et Cargo

La meilleure façon d'installer Rust est d'utiliser . Allez dans *https://rustup.rs* et suivez les instructions qui s'y trouvent. `rustup`

Vous pouvez également accéder au site Web de Rust pour obtenir des packages prédéfinis pour Linux, macOS et Windows. Rust est également inclus dans certaines distributions de système d'exploitation. Nous préférons parce que c'est un outil pour gérer les installations Rust, comme RVM pour Ruby ou NVM pour Node. Par exemple, lorsqu'une nouvelle version de Rust est publiée, vous pourrez effectuer une mise à niveau sans aucun clic en tapant . `rustup rustup update`

Dans tous les cas, une fois l'installation terminée, vous devriez avoir trois nouvelles commandes disponibles sur votre ligne de commande:

```
$ cargo --version
cargo 1.49.0 (d00d64df9 2020-12-05)
$ rustc --version
rustc 1.49.0 (e1884a8e3 2020-12-29)
$ rustdoc --version
rustdoc 1.49.0 (e1884a8e3 2020-12-29)
```

Ici, l'invite de commande est la; sur Windows, ce serait ou quelque chose de similaire. Dans cette transcription, nous exécutons les trois commandes que nous avons installées, en demandant à chacune de signaler de quelle version il s'agit. Prendre chaque commande à tour de rôle : `$ C:\>`

- `cargo` est le gestionnaire de compilation, le gestionnaire de paquets et l'outil général de Rust. Vous pouvez utiliser Cargo pour démarrer un nouveau projet, générer et exécuter votre programme et gérer toutes les bibliothèques externes dont dépend votre code.
- `rustc` est le compilateur Rust. Habituellement, nous laissons Cargo invoquer le compilateur pour nous, mais parfois il est utile de l'exécuter directement.
- `rustdoc` est l'outil de documentation Rust. Si vous écrivez de la documentation dans des commentaires de la forme appropriée dans le code source de votre programme, vous pouvez créer du HTML bien formaté à partir d'eux. Comme , nous laissons généralement Cargo courir pour nous. `rustdoc rustc rustdoc`

Pour plus de commodité, Cargo peut créer un nouveau package Rust pour nous, avec des métadonnées standard organisées de manière appropriée :

```
$ cargo new hello
     Created binary (application) `hello` package
```

Cette commande crée un nouveau répertoire de package nommé *hello*, prêt à générer un exécutable de ligne de commande.

En regardant à l'intérieur du répertoire de niveau supérieur du package :

```
$ cd hello
$ ls -la
total 24
drwxrwxr-x.  4 jimb jimb 4096 Sep 22 21:09 .
drwx------. 62 jimb jimb 4096 Sep 22 21:09 ..
drwxrwxr-x.  6 jimb jimb 4096 Sep 22 21:09 .git
-rw-rw-r--.  1 jimb jimb    7 Sep 22 21:09 .gitignore
-rw-rw-r--.  1 jimb jimb   88 Sep 22 21:09 Cargo.toml
drwxrwxr-x.  2 jimb jimb 4096 Sep 22 21:09 src
```

Nous pouvons voir que Cargo a créé un fichier *Cargo.toml* pour contenir les métadonnées du package. Pour le moment, ce fichier ne contient pas grand-chose:

```
[package]
name = "hello"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Si jamais notre programme acquiert des dépendances avec d'autres bibliothèques, nous pouvons les enregistrer dans ce fichier, et Cargo se chargera de télécharger, de construire et de mettre à jour ces bibliothèques pour nous. Nous couvrirons le fichier *Cargo.toml* en détail au chapitre 8.

Cargo a configuré notre package pour une utilisation avec le système de contrôle de version, en créant un sous-répertoire de métadonnées *.git* et un fichier *.gitignore*. Vous pouvez dire à Cargo de sauter cette étape en passant à sur la ligne de commande. `git --vcs none cargo new`

Le sous-répertoire *src* contient le code Rust réel :

```
$ cd src
$ ls -l
total 4
-rw-rw-r--. 1 jimb jimb 45 Sep 22 21:09 main.rs
```

Il semble que Cargo ait commencé à écrire le programme en notre nom. Le fichier *main.rs* contient le texte suivant :

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

Dans Rust, vous n'avez même pas besoin d'écrire votre propre programme « Hello, World! » Et c'est l'étendue du passe-partout pour un nouveau programme Rust: deux fichiers, totalisant treize lignes.

Nous pouvons appeler la commande à partir de n'importe quel répertoire du package pour construire et exécuter notre programme: `cargo run`

```
$ cargo run
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s
     Running `/home/jimb/rust/hello/target/debug/hello`
Hello, world!
```

Ici, Cargo a appelé le compilateur Rust, puis exécute l'exécutable qu'il a produit. Cargo place l'exécutable dans le sous-répertoire *cible* en haut du package : `rustc`

```
$ ls -l ../target/debug
total 580
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 build
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 deps
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 examples
-rwxrwxr-x. 1 jimb jimb 576632 Sep 22 21:37 hello
-rw-rw-r--. 1 jimb jimb    198 Sep 22 21:37 hello.d
drwxrwxr-x. 2 jimb jimb     68 Sep 22 21:37 incremental
$ ../target/debug/hello
Hello, world!
```

Lorsque nous avons terminé, Cargo peut nettoyer les fichiers générés pour nous:

```
$ cargo clean
$ ../target/debug/hello
bash: ../target/debug/hello: No such file or directory
```

Fonctions de rouille

La syntaxe de Rust est délibérément peu originale. Si vous êtes familier avec C, C ++, Java ou JavaScript, vous pouvez probablement trouver votre chemin à travers la structure générale d'un programme Rust. Voici une fonction qui calcule le plus grand diviseur commun de deux entiers, en utilisant l'algorithme d'Euclide. Vous pouvez ajouter ce code à la fin de *src/main.rs* :

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
```

```
                let t = m;
                m = n;
                n = t;
            }
            m = m % n;
        }
        n
    }
```

Le mot-clé (prononcé « fun ») introduit une fonction. Ici, nous définissons une fonction nommée , qui prend deux paramètres et , dont chacun est de type , un entier 64 bits non signé. Le jeton précède le type de retour : notre fonction renvoie une valeur. L'indentation à quatre espaces est de style Rust standard. `fn gcd n m u64 -> u64`

Les noms de type entier de machine de Rust reflètent leur taille et leur signature : est un entier 32 bits signé ; est un entier 8 bits non signé (utilisé pour les valeurs « octet »), et ainsi de suite. Les types et contiennent des entiers signés et non signés de la taille d'un pointeur, 32 bits de long sur les plates-formes 32 bits et 64 bits de long sur les plates-formes 64 bits. Rust a également deux types à virgule flottante, et , qui sont les types à virgule flottante IEEE simple et double précision, comme et en C et C ++. `i32 u8 isize usize f32 f64 float double`

Par défaut, une fois qu'une variable est initialisée, sa valeur ne peut pas être modifiée, mais placer le mot-clé (prononcé « mute », abréviation de *mutable*) avant les paramètres et permet à notre corps de fonction de leur attribuer. En pratique, la plupart des variables ne sont pas affectées à; le mot-clé sur ceux qui le font peut être un indice utile lors de la lecture du code. `mut n m mut`

Le corps de la fonction commence par un appel à la macro, vérifiant qu'aucun des deux arguments n'est nul. Le caractère marque cela comme un appel de macro, et non comme un appel de fonction. Comme la macro en C et C++, Rust vérifie que son argument est vrai et, s'il ne l'est pas, met fin au programme avec un message utile comprenant l'emplacement source de la vérification défaillante ; ce genre de résiliation brutale s'appelle une *panique*. Contrairement à C et C++, dans lesquels les assertions peuvent être ignorées, Rust vérifie toujours les assertions, quelle que soit la façon dont le programme a été compilé. Il existe également une macro, dont les assertions sont ignorées lorsque le programme est compilé pour la vitesse. `assert! ! assert assert! debug_assert!`

Le cœur de notre fonction est une boucle contenant une instruction et une affectation. Contrairement à C et C++, Rust n'exige pas de parenthèses autour des expressions conditionnelles, mais il nécessite des accolades autour des instructions qu'ils contrôlent. `while if`

Une instruction déclare une variable locale, comme dans notre fonction. Nous n'avons pas besoin d'écrire le type de 's, tant que Rust peut le déduire de la façon dont la variable est utilisée. Dans notre fonction, le seul type qui fonctionne pour est , matching et . Rust ne déduit que des types dans les corps de fonction: vous devez écrire les types de paramètres de fonction et les valeurs de retour, comme nous l'avons fait auparavant. Si nous voulions épeler le type de ' nous pourrions écrire
: `let t t t u64 m n t`

```
    let t: u64 = m;
```

Rust a une instruction, mais la fonction n'en a pas besoin. Si un corps de fonction se termine par une expression qui *n'est pas* suivie d'un point-virgule, il s'agit de la valeur de retour de la fonction. En fait, tout bloc entouré d'accolades peut fonctionner comme une expression. Par exemple, il s'agit d'une expression qui imprime un message, puis donne comme valeur : `return gcd x.cos()`

```
    {
        println!("evaluating cos x");
        x.cos()
    }
```

Il est typique dans Rust d'utiliser cette forme pour établir la valeur de la fonction lorsque le contrôle « tombe de la fin » de la fonction, et d'utiliser des instructions uniquement pour les premiers retours explicites du milieu d'une fonction. `return`

Écriture et exécution de tests unitaires

Rust dispose d'un support simple pour les tests intégrés dans le langage. Pour tester notre fonction, nous pouvons ajouter ce code à la fin de *src/main.rs* : `gcd`

```
    #[test]
    fn test_gcd() {
        assert_eq!(gcd(14, 15), 1);

        assert_eq!(gcd(2 * 3 * 5 * 11 * 17,
                       3 * 7 * 11 * 13 * 19),
                   3 * 11);
    }
```

Ici, nous définissons une fonction nommée , qui appelle et vérifie qu'elle renvoie des valeurs correctes. Le haut de la définition marque comme une fonction de test, à sauter dans les compilations normales, mais inclus et appelé automatiquement si nous exécutons notre programme avec la

commande. Nous pouvons avoir des fonctions de test dispersées dans notre arborescence source, placées à côté du code qu'elles exercent, et nous les rassemblerons automatiquement et les exécuterons toutes. `test_gcd gcd #[test] test_gcd cargo test cargo test`

Le marqueur est un exemple *d'attribut*. Les attributs sont un système ouvert permettant de marquer des fonctions et d'autres déclarations avec des informations supplémentaires, telles que des attributs en C++ et C#, ou des annotations en Java. Ils sont utilisés pour contrôler les avertissements du compilateur et les vérifications de style de code, inclure le code conditionnellement (comme en C et C ++), indiquer à Rust comment interagir avec le code écrit dans d'autres langages, etc. Nous verrons d'autres exemples d'attributs au fur et à mesure. `#[test] #ifdef`

Avec nos définitions ajoutées au package *hello* que nous avons créé au début du chapitre, et notre répertoire actuel quelque part dans la sous-arborescence du package, nous pouvons exécuter les tests comme suit: `gcd test_gcd`

```
$ cargo test
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished test [unoptimized + debuginfo] target(s) in 0.35s
     Running unittests (/home/jimb/rust/hello/target/debug/deps/hello-2375..

running 1 test
test test_gcd ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Gestion des arguments de ligne de commande

Pour que notre programme prenne une série de nombres comme arguments de ligne de commande et imprime leur plus grand diviseur commun, nous pouvons remplacer la fonction dans *src/main.rs* par ce qui suit: `main`

```
use std::str::FromStr;
use std::env;

fn main() {
    let mut numbers = Vec::new();

    for arg in env::args().skip(1) {
        numbers.push(u64::from_str(&arg)
                     .expect("error parsing argument"));
    }
```

```
        if numbers.len() == 0 {
            eprintln!("Usage: gcd NUMBER ...");
            std::process::exit(1);
        }

        let mut d = numbers[0];
        for m in &numbers[1..] {
            d = gcd(d, *m);
        }

        println!("The greatest common divisor of {:?} is {}",
                 numbers, d);
    }
```

Il s'agit d'un gros bloc de code, alors prenons-le pièce par pièce:

```
use std::str::FromStr;
use std::env;
```

La première déclaration introduit le *trait* de bibliothèque standard dans la portée. Un trait est un ensemble de méthodes que les types peuvent implémenter. Tout type qui implémente le trait possède une méthode qui tente d'analyser une valeur de ce type à partir d'une chaîne. Le type implémente , et nous allons appeler pour analyser nos arguments de ligne de commande. Bien que nous n'utilisions jamais le nom ailleurs dans le programme, un trait doit être dans la portée afin d'utiliser ses méthodes. Nous couvrirons les traits en détail dans le chapitre 11. `use FromStr FromStr from_str u64 FromStr u64::from_str FromStr`

La deuxième déclaration apporte le module, qui fournit plusieurs fonctions et types utiles pour interagir avec l'environnement d'exécution, y compris la fonction, qui nous donne accès aux arguments de ligne de commande du programme. `use std::env args`

Passons maintenant à la fonction du programme : `main`

```
fn main() {
```

Notre fonction ne renvoie pas de valeur, nous pouvons donc simplement omettre le type et return qui suivrait normalement la liste des paramètres. `main ->`

```
let mut numbers = Vec::new();
```

Nous déclarons une variable locale mutable et l'initialisons sur un vecteur vide. est le type de vecteur cultivable de Rust, analogue à C++, une liste Python ou un tableau JavaScript. Même si les vecteurs sont conçus pour être cultivés et rétrécis dynamiquement, nous devons toujours mar-

quer la variable pour Rust pour nous permettre de pousser les nombres à la fin de celle-ci. `numbers Vec std::vector mut`

Le type de est , un vecteur de valeurs, mais comme auparavant, nous n'avons pas besoin de l'écrire. Rust va le déduire pour nous, en partie parce que ce que nous poussons sur le vecteur sont des valeurs, mais aussi parce que nous passons les éléments du vecteur à , qui n'accepte que des valeurs. `numbers Vec<u64> u64 u64 gcd u64`

```
for arg in env::args().skip(1) {
```

Ici, nous utilisons une boucle pour traiter nos arguments de ligne de commande, en définissant la variable à chaque argument à tour de rôle et en évaluant le corps de la boucle. `for arg`

La fonction du module renvoie un *itérateur*, une valeur qui produit chaque argument à la demande et indique quand nous avons terminé. Les itérateurs sont omniprésents dans Rust; la bibliothèque standard comprend d'autres itérateurs qui produisent les éléments d'un vecteur, les lignes d'un fichier, les messages reçus sur un canal de communication et presque tout ce qui a du sens à boucler. Les itérateurs de Rust sont très efficaces : le compilateur est généralement capable de les traduire dans le même code qu'une boucle manuscrite. Nous montrerons comment cela fonctionne et donnerons des exemples au chapitre 15. `std::env args`

Au-delà de leur utilisation avec des boucles, les itérateurs incluent une large sélection de méthodes que vous pouvez utiliser directement. Par exemple, la première valeur produite par l'itérateur renvoyé par est toujours le nom du programme en cours d'exécution. Nous voulons ignorer cela, alors nous appelons la méthode de l'itérateur pour produire un nouvel itérateur qui omet cette première valeur. `for args skip`

```
numbers.push(u64::from_str(&arg)
             .expect("error parsing argument"));
```

Ici, nous appelons pour tenter d'analyser notre argument de ligne de commande en tant qu'entier 64 bits non signé. Plutôt qu'une méthode que nous appelons sur une valeur que nous avons à portée de main, est une fonction associée au type, semblable à une méthode statique en C ++ ou Java. La fonction ne renvoie pas une valeur directe, mais plutôt une valeur qui indique si l'analyse a réussi ou échoué. Une valeur est l'une des deux variantes suivantes
: `u64::from_str arg u64 u64::from_str u64 from_str u64 Result t Result`

- Valeur écrite , indiquant que l'analyse a réussi et est la valeur produite `Ok(v) v`

- Valeur écrite , indiquant que l'analyse a échoué et est une valeur d'erreur expliquant pourquoi `Err(e) e`

Les fonctions qui font tout ce qui pourrait échouer, comme effectuer une entrée ou une sortie ou interagir avec le système d'exploitation, peuvent renvoyer des types dont les variantes portent des résultats réussis (le nombre d'octets transférés, le fichier ouvert, etc.) et dont les variantes portent un code d'erreur indiquant ce qui s'est mal passé. Contrairement à la plupart des langues modernes, Rust n'a pas d'exceptions: toutes les erreurs sont traitées en utilisant l'un ou l'autre ou la panique, comme indiqué dans le chapitre 7. `Result Ok Err Result`

Nous utilisons la méthode de ' pour vérifier le succès de notre analyse. Si le résultat est un , imprime un message qui inclut une description du programme et le quitte immédiatement. Cependant, si le résultat est , se renvoie simplement lui-même, que nous sommes finalement capables de pousser à la fin de notre vecteur de
nombres. `Result expect Err(e) expect e Ok(v) expect v`

```
if numbers.len() == 0 {
    eprintln!("Usage: gcd NUMBER ...");
    std::process::exit(1);
}
```

Il n'y a pas de plus grand diviseur commun d'un ensemble vide de nombres, nous vérifions donc que notre vecteur a au moins un élément et quittons le programme avec une erreur si ce n'est pas le cas. Nous utilisons la macro pour écrire notre message d'erreur dans le flux de sortie d'erreur standard. `eprintln!`

```
let mut d = numbers[0];
for m in &numbers[1..] {
    d = gcd(d, *m);
}
```

Cette boucle utilise comme valeur d'exécution, la mettant à jour pour rester le plus grand diviseur commun de tous les nombres que nous avons traités jusqu'à présent. Comme précédemment, nous devons marquer comme mutable afin que nous puissions l'affecter dans la
boucle. `d d`

La boucle a deux bits surprenants. Tout d'abord, nous avons écrit ; à quoi sert l'opérateur? Deuxièmement, nous avons écrit ; qu'est-ce que c'est? Ces deux détails sont complémentaires l'un de l'autre. `for for m in &numbers[1..] & gcd(d, *m) * *m`

Jusqu'à présent, notre code n'a fonctionné que sur des valeurs simples comme des entiers qui tiennent dans des blocs de mémoire de taille fixe.

Mais maintenant, nous sommes sur le point d'itérer sur un vecteur, qui pourrait être de n'importe quelle taille, peut-être très grand. Rust est prudent lorsqu'il manipule de telles valeurs : il veut laisser au programmeur le contrôle de la consommation de mémoire, en précisant combien de temps chaque valeur vit, tout en veillant à ce que la mémoire soit libérée rapidement lorsqu'elle n'est plus nécessaire.

Donc, lorsque nous itérons, nous voulons dire à Rust que *la propriété* du vecteur devrait rester avec ; nous empruntons *simplement ses éléments* pour la boucle. L'opérateur dans emprunte une *référence* aux éléments du vecteur à partir de la seconde. La boucle itére sur les éléments référencés, permettant d'emprunter chaque élément successivement. L'opérateur dans *les déréférencements* , donnant la valeur à laquelle il se réfère; c'est le prochain que nous voulons passer à . Enfin, puisque possède le vecteur, Rust le libère automatiquement lorsqu'il sort du champ d'application à la fin de
`.numbers & &numbers[1..] for m * *m m u64 gcd numbers number s main`

Les règles de propriété et de références de Rust sont essentielles à la gestion de la mémoire et à la concurrence d'accès sécurisée de Rust ; nous en discutons en détail au chapitre 4 et son compagnon, le chapitre 5. Vous devrez être à l'aise avec ces règles pour être à l'aise dans Rust, mais pour cette visite d'introduction, tout ce que vous devez savoir est qu'il emprunte une référence à , et c'est la valeur à laquelle la référence se réfère. `&x x *r r`

Poursuivant notre promenade à travers le programme:

```
    println!("The greatest common divisor of {:?} is {}",
             numbers, d);
```

Après avoir itéré sur les éléments de , le programme imprime les résultats dans le flux de sortie standard. La macro prend une chaîne de modèle, remplace les versions formatées des arguments restants par les formulaires tels qu'ils apparaissent dans la chaîne de modèle et écrit le résultat dans le flux de sortie standard. `numbers println! {...}`

Contrairement à C et C++, qui nécessitent de retourner zéro si le programme s'est terminé avec succès, ou un état de sortie non nul si quelque chose s'est mal passé, Rust suppose que si le programme revient du tout, le programme s'est terminé avec succès. Ce n'est qu'en appelant explicitement des fonctions comme ou pouvons-nous provoquer l'arrêt du programme avec un code d'état
d'erreur. `main main expect std::process::exit`

La commande nous permet de passer des arguments à notre programme, afin que nous puissions essayer notre gestion de ligne de

commande: `cargo run`

```
$ cargo run 42 56
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
     Running `/home/jimb/rust/hello/target/debug/hello 42 56`
The greatest common divisor of [42, 56] is 14
$ cargo run 799459 28823 27347
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello 799459 28823 27347`
The greatest common divisor of [799459, 28823, 27347] is 41
$ cargo run 83
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello 83`
The greatest common divisor of [83] is 83
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `/home/jimb/rust/hello/target/debug/hello`
Usage: gcd NUMBER ...
```

Nous avons utilisé quelques fonctionnalités de la bibliothèque standard de Rust dans cette section. Si vous êtes curieux de savoir ce qui est disponible, nous vous encourageons fortement à essayer la documentation en ligne de Rust. Il dispose d'une fonction de recherche en direct qui facilite l'exploration et inclut même des liens vers le code source. La commande installe automatiquement une copie sur votre ordinateur lorsque vous installez Rust lui-même. Vous pouvez afficher la documentation de la bibliothèque standard sur le site Web de Rust ou dans votre navigateur avec la commande : `rustup`

```
$ rustup doc --std
```

Servir des pages sur le Web

L'une des forces de Rust est la collection de paquets de bibliothèque disponibles gratuitement publiés sur le site Web crates.io. La commande permet à votre code d'utiliser facilement un package crates.io : il téléchargera la bonne version du package, le construira et le mettra à jour comme demandé. Un paquet Rust, qu'il s'agisse d'une bibliothèque ou d'un exécutable, est appelé une *caisse*; Cargo et crates.io tirent tous deux leurs noms de ce terme. `cargo`

Pour montrer comment cela fonctionne, nous allons mettre en place un serveur Web simple en utilisant la caisse du framework Web, la caisse de sérialisation et diverses autres caisses dont elles dépendent. Comme le montre la figure 2-1, notre site Web demandera à l'utilisateur deux nombres et calculera son plus grand diviseur commun. `actix-web serde`

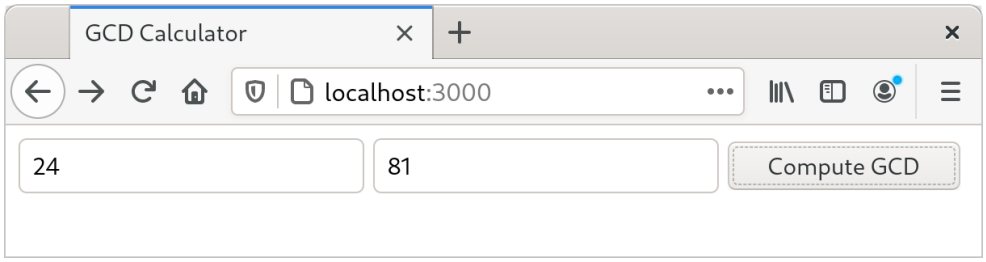

Figure 2-1. Page Web offrant le calcul de GCD

Tout d'abord, nous demanderons à Cargo de créer un nouveau package pour nous, nommé : `actix-gcd`

```
$ cargo new actix-gcd
     Created binary (application) `actix-gcd` package
$ cd actix-gcd
```

Ensuite, nous allons modifier le fichier *Cargo.toml* de notre nouveau projet pour répertorier les paquets que nous voulons utiliser; son contenu devrait être le suivant:

```
[package]
name = "actix-gcd"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
actix-web = "1.0.8"
serde = { version = "1.0", features = ["derive"] }
```

Chaque ligne de la section *cargo.toml* donne le nom d'une caisse sur crates.io, et la version de cette caisse que nous aimerions utiliser. Dans ce cas, nous voulons la version de la caisse et la version de la caisse. Il se peut qu'il existe des versions de ces caisses sur crates.io plus récentes que celles montrées ici, mais en nommant les versions spécifiques contre lesquelles nous avons testé ce code, nous pouvons nous assurer que le code continuera à se compiler même si de nouvelles versions des paquets sont publiées. Nous aborderons la gestion des versions plus en détail au chapitre 8. `[dependencies] 1.0.8 actix-web 1.0 serde`

Les caisses peuvent avoir des fonctionnalités optionnelles: des parties de l'interface ou de l'implémentation dont tous les utilisateurs n'ont pas besoin, mais qui ont néanmoins du sens à inclure dans cette caisse. La caisse offre un moyen merveilleusement laconique de gérer les données des formulaires Web, mais selon la documentation de , elle n'est disponible que si nous sélectionnons la fonctionnalité de la caisse, nous l'avons donc demandée dans notre fichier *Cargo.toml* comme indiqué. `serde serde derive`

Notez que nous n'avons qu'à nommer les caisses que nous utiliserons directement; prend soin d'apporter toutes les autres caisses dont ils ont besoin à leur tour. `cargo`

Pour notre première itération, nous garderons le serveur Web simple: il ne servira que la page qui invite l'utilisateur à calculer des nombres. Dans *actix-gcd/src/main.rs*, nous allons placer le texte suivant :

```
use actix_web::{web, App, HttpResponse, HttpServer};

fn main() {
    let server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(get_index))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to address")
        .run().expect("error running server");
}

fn get_index() -> HttpResponse {
    HttpResponse::Ok()
        .content_type("text/html")
        .body(
            r#"
                <title>GCD Calculator</title>
                <form action="/gcd" method="post">
                <input type="text" name="n"/>
                <input type="text" name="m"/>
                <button type="submit">Compute GCD</button>
                </form>
            "#,
        )
}
```

Nous commençons par une déclaration pour rendre certaines des définitions de la caisse plus faciles à obtenir. Lorsque nous écrivons , chacun des noms énumérés entre parenthèses devient directement utilisable dans notre code; au lieu d'avoir à épeler le nom complet chaque fois que nous l'utilisons, nous pouvons simplement nous y référer comme . (Nous irons à la caisse dans un instant.) `use actix-web use actix_web::{...} actix_web::HttpResponse HttpResponse serde`

Notre fonction est simple : elle appelle à créer un serveur qui répond aux demandes pour un seul chemin, ; imprime un message nous rappelant comment nous y connecter; , puis définit l'écoute sur le port TCP 3000 sur l'ordinateur local. `main HttpServer::new "/"`

L'argument auquel nous passons est l'expression de *fermeture* de Rust . Une fermeture est une valeur qui peut être appelée comme s'il s'agissait d'une fonction. Cette fermeture ne prend aucun argument, mais si c'était le cas, leurs noms apparaîtraient entre les barres verticales. Le est le corps de la fermeture. Lorsque nous démarrons notre serveur, Actix démarre un pool de threads pour gérer les demandes entrantes. Chaque thread appelle notre fermeture pour obtenir une nouvelle copie de la valeur qui lui indique comment acheminer et gérer les demandes. `HttpServer::new || { App::new() ... } || { ... } App`

La fermeture appelle à créer un nouveau, vide, puis appelle sa méthode pour ajouter un itinéraire unique pour le chemin d'accès . Le gestionnaire fourni pour cet itinéraire, , traite les requêtes HTTP en appelant la fonction . La méthode renvoie la même chose qu'elle a été appelée, maintenant améliorée avec le nouvel itinéraire. Comme il n'y a pas de point-virgule à la fin du corps de la fermeture, la valeur de retour de la fermeture est prête à être utilisée par le fil. `App::new App route "/" web::get().to(get_index) GET get_index route App App HttpServer`

La fonction génère une valeur représentant la réponse à une requête HTTP. représente un état HTTP, indiquant que la requête a réussi. Nous appelons ses méthodes et méthodes pour remplir les détails de la réponse; chaque appel renvoie celui auquel il a été appliqué, avec les modifications apportées. Enfin, la valeur renvoyée de sert de valeur de retour à . `get_index HttpResponse GET / HttpResponse::Ok() 200 OK content_type body HttpResponse body get_index`

Étant donné que le texte de réponse contient beaucoup de guillemets doubles, nous l'écrivons en utilisant la syntaxe Rust « chaîne brute »: la lettre , zéro ou plusieurs marques de hachage (c'est-à-dire le caractère), un guillemet double, puis le contenu de la chaîne, terminé par un autre guillemet double suivi du même nombre de marques de hachage. Tout caractère peut apparaître dans une chaîne brute sans être échappé, y compris les guillemets doubles ; en fait, aucune séquence d'échappement comme celle-ci n'est reconnue. Nous pouvons toujours nous assurer que la chaîne se termine là où nous le voulons en utilisant plus de marques de hachage autour des guillemets que jamais n'apparaissent dans le texte. `r # \"`

Après avoir écrit *main.rs*, nous pouvons utiliser la commande pour faire tout ce qui est nécessaire pour le mettre en marche: récupérer les caisses nécessaires, les compiler, construire notre propre programme, tout relier ensemble et le démarrer: `cargo run`

```
$ cargo run
    Updating crates.io index
 Downloading crates ...
  Downloaded serde v1.0.100
  Downloaded actix-web v1.0.8
  Downloaded serde_derive v1.0.100
...
   Compiling serde_json v1.0.40
   Compiling actix-router v0.1.5
   Compiling actix-http v0.2.10
   Compiling awc v0.2.7
   Compiling actix-web v1.0.8
   Compiling gcd v0.1.0 (/home/jimb/rust/actix-gcd)
    Finished dev [unoptimized + debuginfo] target(s) in 1m 24s
     Running `/home/jimb/rust/actix-gcd/target/debug/actix-gcd`
Serving on http://localhost:3000...
```

À ce stade, nous pouvons visiter l'URL donnée dans notre navigateur et voir la page illustrée plus haut dans la figure 2-1.

Malheureusement, cliquer sur Calculer GCD ne fait rien, si ce n'est naviguer dans notre navigateur vers une page vierge. Corrigeons cela ensuite, en ajoutant un autre itinéraire à notre pour gérer la demande à partir de notre formulaire. `App POST`

Il est enfin temps d'utiliser la caisse que nous avons répertoriée dans notre fichier *Cargo.toml*: elle fournit un outil pratique qui nous aidera à traiter les données du formulaire. Tout d'abord, nous devrons ajouter la directive suivante en haut de *src/main.rs* : `serde use`

```
use serde::Deserialize;
```

Les programmeurs Rust rassemblent généralement toutes leurs déclarations vers le haut du fichier, mais ce n'est pas strictement nécessaire : Rust permet aux déclarations de se produire dans n'importe quel ordre, tant qu'elles apparaissent au niveau d'imbrication approprié. `use`

Ensuite, définissons un type de structure Rust qui représente les valeurs que nous attendons de notre formulaire :

```
#[derive(Deserialize)]
struct GcdParameters {
    n: u64,
    m: u64,
}
```

Cela définit un nouveau type nommé qui a deux champs, et , dont chacun est un —le type d'argument attendu par notre fonction. `GcdParameters n m u64 gcd`

L'annotation au-dessus de la définition est un attribut, comme l'attribut que nous avons utilisé précédemment pour marquer les fonctions de test. Placer un attribut au-dessus d'une définition de type indique à la caisse d'examiner le type lorsque le programme est compilé et de générer automatiquement du code pour analyser une valeur de ce type à partir de données dans le format utilisé par les formulaires HTML pour les demandes. En fait, cet attribut est suffisant pour vous permettre d'analyser une valeur à partir de presque tous les types de données structurées : JSON, YAML, TOML ou l'un des nombreux autres formats textuels et binaires. La caisse fournit également un attribut qui génère du code pour faire l'inverse, en prenant des valeurs Rust et en les écrivant dans un format structuré. `struct #[test] #[derive(Deserialize)] serde POST GcdParameters serde Serialize`

Avec cette définition en place, nous pouvons écrire notre fonction de gestionnaire assez facilement:

```
fn post_gcd(form: web::Form<GcdParameters>) -> HttpResponse {
    if form.n == 0 || form.m == 0 {
        return HttpResponse::BadRequest()
            .content_type("text/html")
            .body("Computing the GCD with zero is boring.");
    }

    let response =
        format!("The greatest common divisor of the numbers {} and {} \
                 is <b>{}</b>\n",
                form.n, form.m, gcd(form.n, form.m));

    HttpResponse::Ok()
        .content_type("text/html")
        .body(response)
}
```

Pour qu'une fonction serve de gestionnaire de requêtes Actix, ses arguments doivent tous avoir des types qu'Actix sait extraire d'une requête HTTP. Notre fonction prend un argument, , dont le type est . Actix sait comment extraire une valeur de n'importe quel type à partir d'une requête HTTP si, et seulement si, `T` peut être désérialisé à partir de données de formulaire HTML. Depuis que nous avons placé l'attribut sur notre définition de type, Actix peut le désérialiser à partir des données de formulaire, de sorte que les gestionnaires de demandes peuvent s'attendre à une valeur en tant que paramètre. Ces relations entre les types et les fonctions sont toutes élaborées au moment de la compilation; Si vous écrivez une fonction de gestionnaire avec un type d'argument qu'Actix ne sait pas gérer, le compilateur Rust vous informe immédiatement de votre erreur. `post_gcd form web::Form<GcdParameters> web::Form<T> P`

```
OST #
[derive(Deserialize)] GcdParameters web::Form<GcdParameters
>
```

En regardant à l'intérieur, la fonction renvoie d'abord une erreur HTTP si l'un des paramètres est nul, car notre fonction paniquera s'ils le sont. Ensuite, il construit une réponse à la demande à l'aide de la macro. La macro est comme la macro, sauf qu'au lieu d'écrire le texte dans la sortie standard, elle le renvoie sous forme de chaîne. Une fois qu'il a obtenu le texte de la réponse, l'enveloppe dans une réponse HTTP, définit son type de contenu et le renvoie pour être remis à l'expéditeur. `post_gcd 400 BAD REQUEST gcd format! format! println! post_gcd 200 OK`

Nous devons également nous inscrire en tant que gestionnaire du formulaire. Nous allons remplacer notre fonction par cette version `: post_gcd main`

```
fn main() {
    let server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(get_index))
            .route("/gcd", web::post().to(post_gcd))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000").expect("error binding server to address")
        .run().expect("error running server");
}
```

Le seul changement ici est que nous avons ajouté un autre appel à , établissant comme gestionnaire pour le chemin `. route web::post().to(post_gcd) "/gcd"`

La dernière pièce restante est la fonction que nous avons écrite précédemment, qui va dans le fichier *actix-gcd / src / main.rs*. Avec cela en place, vous pouvez interrompre tous les serveurs que vous avez peut-être laissés en cours d'exécution et reconstruire et redémarrer le programme: `gcd`

```
$ cargo run
   Compiling actix-gcd v0.1.0 (/home/jimb/rust/actix-gcd)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/actix-gcd`
Serving on http://localhost:3000...
```

Cette fois, en visitant *http://localhost:3000*, en entrant des chiffres et en cliquant sur le bouton Calculer GCD, vous devriez réellement voir des résultats (Figure 2-2).

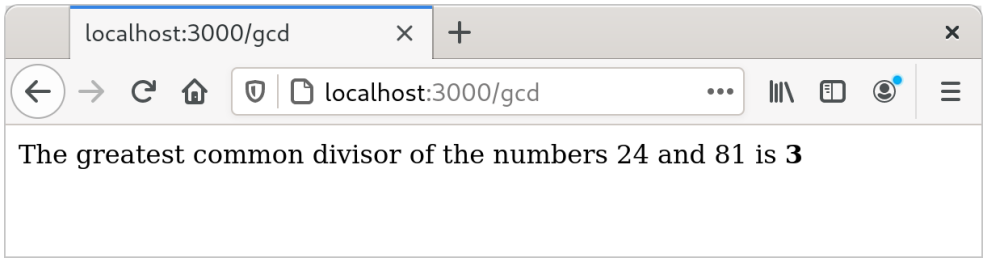

Figure 2-2. Page Web montrant les résultats du calcul gcD

Concurrence

L'une des grandes forces de Rust est sa prise en charge de la programmation simultanée. Les mêmes règles qui garantissent que les programmes Rust sont exempts d'erreurs de mémoire garantissent également que les threads ne peuvent partager la mémoire que de manière à éviter les courses de données. Par exemple:

- Si vous utilisez un mutex pour coordonner les threads qui apportent des modifications à une structure de données partagée, Rust s'assure que vous ne pouvez pas accéder aux données sauf lorsque vous maintenez le verrou enfoncé et libère automatiquement le verrou lorsque vous avez terminé. En C et C++, la relation entre un mutex et les données qu'il protège est laissée aux commentaires.
- Si vous souhaitez partager des données en lecture seule entre plusieurs threads, Rust garantit que vous ne pouvez pas modifier les données accidentellement. En C et C ++, le système de type peut aider à cela, mais il est facile de se tromper.
- Si vous transférez la propriété d'une structure de données d'un thread à un autre, Rust s'assure que vous avez effectivement renoncé à tout accès à celle-ci. En C et C++, c'est à vous de vérifier que rien sur le thread d'envoi ne touchera plus jamais les données. Si vous ne le faites pas correctement, les effets peuvent dépendre de ce qui se trouve dans le cache du processeur et du nombre d'écritures en mémoire que vous avez effectuées récemment. Non pas que nous soyons amers.

Dans cette section, nous vous guiderons tout au long du processus d'écriture de votre deuxième programme multithread.

Vous avez déjà écrit votre premier : le framework Web Actix que vous avez utilisé pour implémenter le serveur Greatest Common Divisor utilise un pool de threads pour exécuter les fonctions de gestionnaire de requêtes. Si le serveur reçoit des requêtes simultanées, il peut exécuter les fonctions et dans plusieurs threads à la fois. Cela peut être un peu choquant, car nous n'avions certainement pas la concurrence à l'esprit lorsque nous avons écrit ces fonctions. Mais Rust garantit que cela est sûr à faire, peu importe l'élaboration de votre serveur: si votre programme compile, il est exempt de courses de données. Toutes les fonctions Rust sont thread-safe. `get_form post_gcd`

Le programme de cette section trace l'ensemble de Mandelbrot, une fractale produite en itérant une fonction simple sur des nombres complexes. Tracer l'ensemble de Mandelbrot est souvent appelé un algorithme *parallèle embarrassant*, parce que le modèle de communication entre les fils est si simple; nous couvrirons des modèles plus complexes dans le chapitre 19, mais cette tâche démontre certains des éléments essentiels.

Pour commencer, nous allons créer un nouveau projet Rust :

```
$ cargo new mandelbrot
     Created binary (application) `mandelbrot` package
$ cd mandelbrot
```

Tout le code ira dans *mandelbrot / src / main.rs*, et nous ajouterons quelques dépendances à *mandelbrot / Cargo.toml*.

Avant d'entrer dans l'implémentation simultanée de Mandelbrot, nous devons décrire le calcul que nous allons effectuer.

Qu'est-ce que l'ensemble Mandelbrot est réellement

Lors de la lecture de code, il est utile d'avoir une idée concrète de ce qu'il essaie de faire, alors faisons une courte excursion dans les mathématiques pures. Nous allons commencer par un cas simple, puis ajouter des détails compliqués jusqu'à ce que nous arrivions au calcul au cœur de l'ensemble mandelbrot.

Voici une boucle infinie, écrite en utilisant la syntaxe dédiée de Rust pour cela, une déclaration: `loop`

```
fn square_loop(mut x: f64) {
    loop {
        x = x * x;
    }
}
```

Dans la vraie vie, Rust peut voir qu'il n'est jamais utilisé pour quoi que ce soit et pourrait donc ne pas se donner la peine de calculer sa valeur. Mais pour le moment, supposons que le code s'exécute comme écrit. Qu'advient-il de la valeur de ? La quadrature de tout nombre inférieur à 1 le rend plus petit, de sorte qu'il se rapproche de zéro; la quadrature 1 donne 1; la quadrature d'un nombre supérieur à 1 le rend plus grand, de sorte qu'il se rapproche de l'infini; et la quadrature d'un nombre négatif le rend positif, après quoi il se comporte comme l'un des cas précédents (Figure 2-3). `x x`

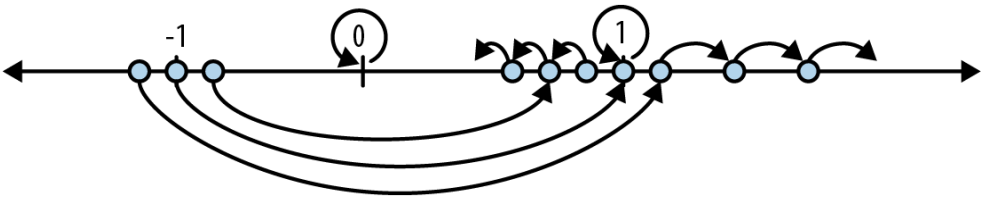

Figure 2-3. Effets de la quadrature répétée d'un certain nombre

Ainsi, selon la valeur à laquelle vous passez, reste à zéro ou à un, s'approche de zéro ou s'approche de l'infini. `square_loop x`

Considérons maintenant une boucle légèrement différente:

```
fn square_add_loop(c: f64) {
    let mut x = 0.;
    loop {
        x = x * x + c;
    }
}
```

Cette fois, commence à zéro, et nous modifions sa progression dans chaque itération en l'ajoutant après l'avoir quadragéné. Cela rend plus difficile de voir comment les tarifs, mais certaines expérimentations montrent que si est supérieur à 0,25 ou inférieur à -2,0, alors finit par devenir infiniment grand; sinon, il reste quelque part dans le voisinage de zéro. `x c x c x`

La ride suivante : au lieu d'utiliser des valeurs, considérez la même boucle en utilisant des nombres complexes. La caisse sur crates.io fournit un type de nombre complexe que nous pouvons utiliser, nous devons donc ajouter une ligne pour à la section dans le fichier *Cargo.toml* de notre programme. Voici l'intégralité du fichier, jusqu'à présent (nous en ajouterons d'autres plus tard) : `f64 num num [dependencies]`

```
[package]
name = "mandelbrot"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
num = "0.4"
```

Maintenant, nous pouvons écrire l'avant-dernière version de notre boucle:

```
use num::Complex;

fn complex_square_add_loop(c: Complex<f64>) {
    let mut z = Complex { re: 0.0, im: 0.0 };
```

```
        loop {
            z = z * z + c;
        }
    }
```

Il est traditionnel à utiliser pour les nombres complexes, nous avons donc renommé notre variable en boucle. L'expression est la façon dont nous écrivons un zéro complexe en utilisant le type de caisse. est un type de structure Rust (ou *struct*), défini comme ceci : `z Complex { re: 0.0, im: 0.0 } num Complex Complex`

```
    struct Complex<T> {
        /// Real portion of the complex number
        re: T,

        /// Imaginary portion of the complex number
        im: T,
    }
```

Le code précédent définit une structure nommée , avec deux champs et . est une structure *générique* : vous pouvez lire le nom après le type comme « pour n'importe quel type ». Par exemple, est un nombre complexe dont les champs sont des valeurs, utiliserait des flottants 32 bits, etc. Compte tenu de cette définition, une expression like produit une valeur avec son champ initialisé à 0,24 et son champ initialisé à 0,3. `Complex re im Complex <T> T Complex<f64> re im f64 Complex<f32> Complex { re: 0.24, im: 0.3 } Complex re im`

La caisse s'arrange pour que , et d'autres opérateurs arithmétiques travaillent sur des valeurs, de sorte que le reste de la fonction fonctionne comme la version précédente, sauf qu'elle fonctionne sur des points du plan complexe, pas seulement sur des points le long de la ligne de nombre réel. Nous vous expliquerons comment vous pouvez faire en sorte que les opérateurs de Rust travaillent avec vos propres types au chapitre 12. `num * + Complex`

Enfin, nous avons atteint la destination de notre excursion en mathématiques pures. L'ensemble de Mandelbrot est défini comme l'ensemble des nombres complexes pour lesquels il ne s'envole pas vers l'infini. Notre boucle de quadrature simple d'origine était assez prévisible : tout nombre supérieur à 1 ou inférieur à –1 s'envole. Jeter un dans chaque itération rend le comportement un peu plus difficile à anticiper: comme nous l'avons dit plus tôt, des valeurs supérieures à 0,25 ou inférieures à -2 provoquent un vol. Mais étendre le jeu à des nombres complexes produit des motifs vraiment bizarres et beaux, qui sont ce que nous voulons tracer. `c z + c c z`

Étant donné qu'un nombre complexe a à la fois des composantes réelles et imaginaires et , nous les traiterons comme les coordonnées d'un point sur le plan cartésien, et colorerons le point en noir si est dans l'ensemble de Mandelbrot, ou une couleur plus claire sinon. Ainsi, pour chaque pixel de notre image, nous devons exécuter la boucle précédente sur le point correspondant sur le plan complexe, voir s'il s'échappe à l'infini ou orbite autour de l'origine pour toujours, et le colorier en conséquence. `c c.re c.im x y c`

La boucle infinie prend un certain temps à courir, mais il y a deux astuces pour les impatients. Tout d'abord, si nous renonçons à exécuter la boucle pour toujours et essayons simplement un nombre limité d'itérations, il s'avère que nous obtenons toujours une approximation décente de l'ensemble. Le nombre d'itérations dont nous avons besoin dépend de la précision avec laquelle nous voulons tracer la frontière. Deuxièmement, il a été démontré que, si jamais le cercle de rayon 2 est centré sur l'origine, il finira certainement par voler infiniment loin de l'origine. Voici donc la version finale de notre boucle, et le cœur de notre programme : `z`

```
use num::Complex;

/// Try to determine if `c` is in the Mandelbrot set, using at most `limit`
/// iterations to decide.
///
/// If `c` is not a member, return `Some(i)`, where `i` is the number of
/// iterations it took for `c` to leave the circle of radius 2 centered on tl
/// origin. If `c` seems to be a member (more precisely, if we reached the
/// iteration limit without being able to prove that `c` is not a member),
/// return `None`.
fn escape_time(c: Complex<f64>, limit: usize) -> Option<usize> {
    let mut z = Complex { re: 0.0, im: 0.0 };
    for i in 0..limit {
        if z.norm_sqr() > 4.0 {
            return Some(i);
        }
        z = z * z + c;
    }

    None
}
```

Cette fonction prend le nombre complexe que nous voulons tester pour l'appartenance à l'ensemble de Mandelbrot et une limite sur le nombre d'itérations à essayer avant d'abandonner et de déclarer être probablement membre. `c c`

La valeur renvoyée de la fonction est un fichier . La bibliothèque standard de Rust définit le type comme suit : `Option<usize> Option`

```
enum Option<T> {
    None,
    Some(T),
}
```

`Option` est un *type énuméré*, souvent appelé *enum*, car sa définition énumère plusieurs variantes qu'une valeur de ce type pourrait être : pour tout type , une valeur de type est soit , où est une valeur de type , soit , indiquant qu'aucune valeur n'est disponible. Comme le type dont nous avons parlé précédemment, est un type générique: vous pouvez utiliser pour représenter une valeur facultative de n'importe quel type que vous aimez. `T Option<T> Some(v) v T None T Complex Option Option<T > T`

Dans notre cas, renvoie un pour indiquer si se trouve dans l'ensemble de Mandelbrot – et si ce n'est pas le cas, combien de temps nous avons dû itérer pour le savoir. Si n'est pas dans l'ensemble, renvoie , où est le numéro de l'itération à laquelle a quitté le cercle de rayon 2. Sinon, est apparemment dans l'ensemble, et renvoie `.escape_time Option<usize> c c escape_time Some(i) i z c esc ape_time None`

```
    for i in 0..limit {
```

Les exemples précédents montraient des boucles itérant sur des arguments de ligne de commande et des éléments vectoriels ; cette boucle itére simplement sur la plage d'entiers commençant par et jusqu'à (mais n'incluant pas) . `for for 0 limit`

L'appel de méthode renvoie le carré de la distance de 'de l'origine. Pour décider si a quitté le cercle de rayon 2, au lieu de calculer une racine carrée, nous comparons simplement la distance au carré avec 4,0, ce qui est plus rapide. `z.norm_sqr() z z`

Vous avez peut-être remarqué que nous utilisons pour marquer les lignes de commentaire au-dessus de la définition de la fonction; les commentaires ci-dessus les membres de la structure commencent également par. Il s'agit *de commentaires sur la documentation*; l'utilitaire sait comment les analyser, ainsi que le code qu'ils décrivent, et produire une documentation en ligne. La documentation de la bibliothèque standard de Rust est écrite sous cette forme. Nous décrivons en détail les commentaires de documentation au chapitre 8. `/// Complex /// rustdoc`

Le reste du programme consiste à décider quelle partie de l'ensemble tracer à quelle résolution et à répartir le travail sur plusieurs threads pour accélérer le calcul.

Analyse des arguments de ligne de commande de paire

Le programme prend plusieurs arguments de ligne de commande contrôlant la résolution de l'image que nous allons écrire et la partie de l'ensemble Mandelbrot que l'image montre. Étant donné que ces arguments de ligne de commande suivent tous une forme commune, voici une fonction pour les analyser :

```
use std::str::FromStr;

/// Parse the string `s` as a coordinate pair, like `"400x600"` or `"1.0,0.5"
///
/// Specifically, `s` should have the form <left><sep><right>, where <sep> is
/// the character given by the `separator` argument, and <left> and <right> a
/// both strings that can be parsed by `T::from_str`. `separator` must be an
/// ASCII character.
///
/// If `s` has the proper form, return `Some<(x, y)>`. If it doesn't parse
/// correctly, return `None`.
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
    match s.find(separator) {
        None => None,
        Some(index) => {
            match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
                (Ok(l), Ok(r)) => Some((l, r)),
                _ => None
            }
        }
    }
}

#[test]
fn test_parse_pair() {
    assert_eq!(parse_pair::<i32>("",        ','), None);
    assert_eq!(parse_pair::<i32>("10,",     ','), None);
    assert_eq!(parse_pair::<i32>(",10",     ','), None);
    assert_eq!(parse_pair::<i32>("10,20",   ','), Some((10, 20)));
    assert_eq!(parse_pair::<i32>("10,20xy", ','), None);
    assert_eq!(parse_pair::<f64>("0.5x",    'x'), None);
    assert_eq!(parse_pair::<f64>("0.5x1.5", 'x'), Some((0.5, 1.5)));
}
```

La définition de est une *fonction générique* : `parse_pair`

```
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
```

Vous pouvez lire la clause à haute voix comme suit : « Pour tout type qui implémente le trait... Cela nous permet effectivement de définir toute une famille de fonctions à la fois : c'est une fonction qui analyse des paires de

valeurs, analyse des paires de valeurs en virgule flottante, etc. Cela ressemble beaucoup à un modèle de fonction en C ++. Un programmeur Rust appellerait un *paramètre de type* de . Lorsque vous utilisez une fonction générique, Rust sera souvent en mesure de déduire des paramètres de type pour vous, et vous n'aurez pas besoin de les écrire comme nous l'avons fait dans le code de test. `<T: FromStr> T FromStr parse_pair::<i32> i32 parse_pair:: <f64> T parse_pair`

Notre type de retour est : soit ou une valeur , où est un tuple de deux valeurs, toutes deux de type . La fonction n'utilise pas d'instruction return explicite, de sorte que sa valeur return est la valeur de la dernière (et unique) expression de son corps : `Option<(T, T)> None Some((v1, v2)) (v1, v2) T parse_pair`

```
match s.find(separator) {
    None => None,
    Some(index) => {
        ...
    }
}
```

La méthode type recherche dans la chaîne un caractère qui correspond à . Si renvoie , ce qui signifie que le caractère séparateur ne se produit pas dans la chaîne, l'expression entière est évaluée à , indiquant que l'analyse a échoué. Sinon, nous prenons pour être la position du séparateur dans la chaîne. `String find separator find None match None index`

```
match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
    (Ok(l), Ok(r)) => Some((l, r)),
    _ => None
}
```

Cela commence à montrer la puissance de l'expression. L'argument de la correspondance est cette expression de tuple : `match`

```
(T::from_str(&s[..index]), T::from_str(&s[index + 1..]))
```

Les expressions et sont des tranches de la chaîne, précédant et suivant le séparateur. La fonction associée au paramètre type prend chacun d'entre eux et tente de les analyser comme une valeur de type , produisant un tuple de résultats. Voici ce à quoi nous nous opposons : `&s[..index] &s[index + 1..] T from_str T`

```
(Ok(l), Ok(r)) => Some((l, r)),
```

Ce modèle ne correspond que si les deux éléments du tuple sont des variantes du type, ce qui indique que les deux analyses ont réussi. Si c'est le cas, est la valeur de l'expression de correspondance et donc la valeur de retour de la fonction. `Ok Result Some((l, r))`

```
    _ => None
```

Le modèle générique correspond à n'importe quoi et ignore sa valeur. Si nous atteignons ce point, alors a échoué, nous évaluons donc à , fournissant à nouveau la valeur de retour de la fonction. `_ parse_pair None`

Maintenant que nous avons , il est facile d'écrire une fonction pour analyser une paire de coordonnées à virgule flottante et les renvoyer sous forme de valeur : `parse_pair Complex<f64>`

```
/// Parse a pair of floating-point numbers separated by a comma as a complex
/// number.
fn parse_complex(s: &str) -> Option<Complex<f64>> {
    match parse_pair(s, ',') {
        Some((re, im)) => Some(Complex { re, im }),
        None => None
    }
}

#[test]
fn test_parse_complex() {
    assert_eq!(parse_complex("1.25,-0.0625"),
               Some(Complex { re: 1.25, im: -0.0625 }));
    assert_eq!(parse_complex(",-0.0625"), None);
}
```

La fonction appelle , crée une valeur si les coordonnées ont été analysées avec succès et transmet les échecs à son appelant. `parse_complex parse_pair Complex`

Si vous lisiez attentivement, vous avez peut-être remarqué que nous avons utilisé une notation abrégée pour construire la valeur. Il est courant d'initialiser les champs d'une struct avec des variables du même nom, donc plutôt que de vous forcer à écrire, Rust vous permet simplement d'écrire . Ceci est calqué sur des notations similaires en JavaScript et Haskell. `Complex Complex { re: re, im: im } Complex { re, im }`

Mappage des pixels aux nombres complexes

Le programme doit travailler dans deux espaces de coordonnées connexes: chaque pixel de l'image de sortie correspond à un point sur le plan complexe. La relation entre ces deux espaces dépend de la partie de l'ensemble de Mandelbrot que nous allons tracer et de la résolution de l'im-

age demandée, telle que déterminée par les arguments de ligne de commande. La fonction suivante convertit *l'espace d'image en espace de nombres complexes* :

```
/// Given the row and column of a pixel in the output image, return the
/// corresponding point on the complex plane.
///
/// `bounds` is a pair giving the width and height of the image in pixels.
/// `pixel` is a (column, row) pair indicating a particular pixel in that ima
/// The `upper_left` and `lower_right` parameters are points on the complex
/// plane designating the area our image covers.
fn pixel_to_point(bounds: (usize, usize),
                  pixel: (usize, usize),
                  upper_left: Complex<f64>,
                  lower_right: Complex<f64>)
    -> Complex<f64>
{
    let (width, height) = (lower_right.re - upper_left.re,
                           upper_left.im - lower_right.im);
    Complex {
        re: upper_left.re + pixel.0 as f64 * width  / bounds.0 as f64,
        im: upper_left.im - pixel.1 as f64 * height / bounds.1 as f64
        // Why subtraction here? pixel.1 increases as we go down,
        // but the imaginary component increases as we go up.
    }
}

#[test]
fn test_pixel_to_point() {
    assert_eq!(pixel_to_point((100, 200), (25, 175),
                              Complex { re: -1.0, im:  1.0 },
                              Complex { re:  1.0, im: -1.0 }),
               Complex { re: -0.5, im: -0.75 });
}
```

La figure 2-4 illustre le calcul effectué. `pixel_to_point`

Le code de est simplement un calcul, nous ne l'expliquerons donc pas en détail. Cependant, il y a quelques points à souligner. Les expressions de cette forme font référence à des éléments de tuple : `pixel_to_point`

```
pixel.0
```

Il s'agit du premier élément du tuple . `pixel`

```
pixel.0 as f64
```

C'est la syntaxe de Rust pour une conversion de type : cela convertit en valeur. Contrairement à C et C++, Rust refuse généralement de convertir implicitement entre les types numériques; vous devez écrire les conver-

sions dont vous avez besoin. Cela peut être fastidieux, mais être explicite sur les conversions qui se produisent et quand est étonnamment utile. Les conversions implicites d'entiers semblent assez innocentes, mais historiquement, elles ont été une source fréquente de bogues et de failles de sécurité dans le code C et C ++ du monde réel. `pixel.0 f64`

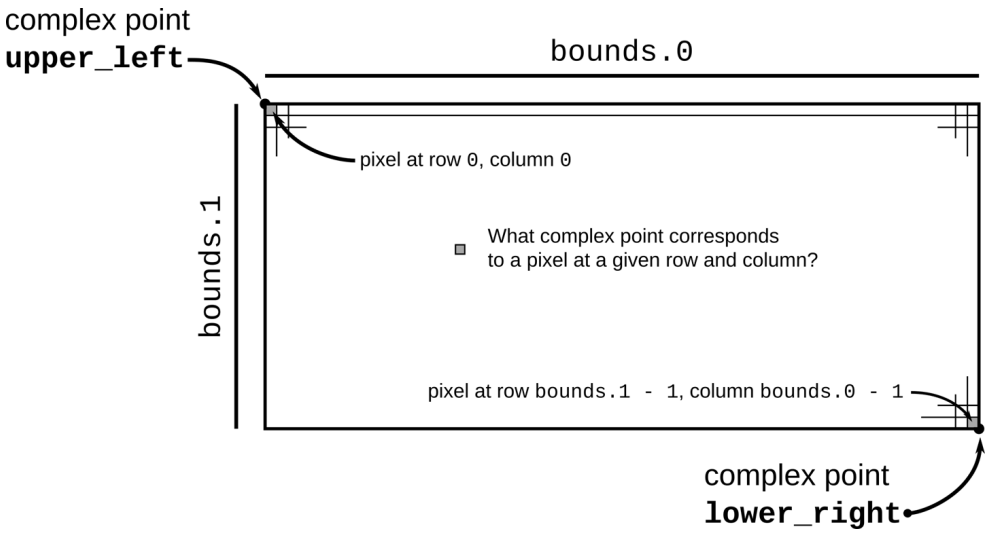

Figure 2-4. La relation entre le plan complexe et les pixels de l'image

Tracé de l'ensemble

Pour tracer l'ensemble de Mandelbrot, pour chaque pixel de l'image, il suffit d'appliquer au point correspondant sur le plan complexe, et de colorier le pixel en fonction du résultat: `escape_time`

```
/// Render a rectangle of the Mandelbrot set into a buffer of pixels.
///
/// The `bounds` argument gives the width and height of the buffer `pixels`,
/// which holds one grayscale pixel per byte. The `upper_left` and `lower_ri
/// arguments specify points on the complex plane corresponding to the upper-
/// left and lower-right corners of the pixel buffer.
fn render(pixels: &mut [u8],
          bounds: (usize, usize),
          upper_left: Complex<f64>,
          lower_right: Complex<f64>)
{
    assert!(pixels.len() == bounds.0 * bounds.1);

    for row in 0..bounds.1 {
        for column in 0..bounds.0 {
            let point = pixel_to_point(bounds, (column, row),
                                       upper_left, lower_right);
            pixels[row * bounds.0 + column] =
                match escape_time(point, 255) {
                    None => 0,
                    Some(count) => 255 - count as u8
                };
        }
    }
}
```

Tout cela devrait sembler assez familier à ce stade.

```
    pixels[row * bounds.0 + column] =
        match escape_time(point, 255) {
            None => 0,
            Some(count) => 255 - count as u8
        };
```

Si dit que cela appartient à l'ensemble, colore le pixel correspondant en noir (). Sinon, attribue des couleurs plus sombres aux nombres qui ont mis plus de temps à s'échapper du
cercle. `escape_time point render 0 render`

Écriture de fichiers image

La caisse fournit des fonctions de lecture et d'écriture d'une grande variété de formats d'image, ainsi que certaines fonctions de manipulation d'image de base. En particulier, il comprend un encodeur pour le format de fichier image PNG, que ce programme utilise pour enregistrer les résultats finaux du calcul. Pour utiliser , ajoutez la ligne suivante à la section de *Cargo.toml* : `image image [dependencies]`

```
    image = "0.13.0"
```

Avec cela en place, nous pouvons écrire:

```
    use image::ColorType;
    use image::png::PNGEncoder;
    use std::fs::File;

    /// Write the buffer `pixels`, whose dimensions are given by `bounds`, to the
    /// file named `filename`.
    fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
        -> Result<(), std::io::Error>
    {
        let output = File::create(filename)?;

        let encoder = PNGEncoder::new(output);
        encoder.encode(pixels,
                       bounds.0 as u32, bounds.1 as u32,
                       ColorType::Gray(8))?;

        Ok(())
    }
```

Le fonctionnement de cette fonction est assez simple: il ouvre un fichier et essaie d'y écrire l'image. Nous passons à l'encodeur les données de pixels réelles de , ainsi que sa largeur et sa hauteur à partir de , puis un argument final qui dit comment interpréter les octets dans : la valeur indique

que chaque octet est une valeur en niveaux de gris de huit bits. `pixels bounds pixels ColorType::Gray(8)`

Tout cela est simple. Ce qui est intéressant à propos de cette fonction, c'est la façon dont elle fait face lorsque quelque chose ne va pas. Si nous rencontrons une erreur, nous devons la signaler à notre appelant. Comme nous l'avons mentionné précédemment, les fonctions faillibles dans Rust doivent renvoyer une valeur, qui est soit sur le succès, où est la valeur réussie, soit sur l'échec, où est un code d'erreur. Alors, quels sont les types de succès et d'erreurs de ? `Result Ok(s) s Err(e) e write_image`

Quand tout se passe bien, notre fonction n'a aucune valeur utile à retourner; il a écrit tout ce qui était intéressant pour le dossier. Donc, son type de succès est le type *d'unité* , ainsi appelé parce qu'il n'a qu'une seule valeur, également écrite . Le type d'unité est similaire à C et C ++. `write_image () () void`

Lorsqu'une erreur se produit, c'est parce que vous n'avez pas pu créer le fichier ou que vous n'avez pas pu y écrire l'image ; l'opération d'E/S a renvoyé un code d'erreur. Le type de retour de est , tandis que celui de est , donc les deux partagent le même type d'erreur, . Il est logique que notre fonction fasse de même. Dans les deux cas, l'échec devrait entraîner un retour immédiat, en transmettant la valeur décrivant ce qui s'est mal passé. `File::create encoder.encode File::create Result<std::fs::File, std::io::Error> encoder.encode Result<(), std::io::Error> std::io::Error write_image std::io::Error`

Donc, pour gérer correctement le résultat de , nous devons sur sa valeur de retour, comme ceci: `File::create match`

```
let output = match File::create(filename) {
    Ok(f) => f,
    Err(e) => {
        return Err(e);
    }
};
```

Sur le succès, que soit le porté dans la valeur. En cas d'échec, transmettez l'erreur à notre propre appelant. `output File Ok`

Ce type d'énoncé est un modèle si courant dans Rust que le langage fournit l'opérateur comme raccourci pour l'ensemble. Ainsi, plutôt que d'écrire cette logique explicitement chaque fois que nous tentons quelque chose qui pourrait échouer, vous pouvez utiliser l'instruction équivalente et beaucoup plus lisible suivante: `match ?`

```
let output = File::create(filename)?;
```

En cas d'échec, l'opérateur renvoie à partir de , en transmettant l'erreur. Sinon, maintient le fichier . `File::create ?`

`write_image output File`

NOTE

C'est une erreur courante du débutant d'essayer d'utiliser dans la fonction. Cependant, comme elle-même ne renvoie pas de valeur, cela ne fonctionnera pas; au lieu de cela, vous devez utiliser une instruction ou l'une des méthodes abrégées telles que et . Il y a aussi la possibilité de simplement changer pour retourner un , que nous couvrirons plus tard. `?`

`main main match unwrap expect main Result`

Un programme Mandelbrot simultané

Toutes les pièces sont en place, et nous pouvons vous montrer la fonction, où nous pouvons mettre la concurrence au travail pour nous. Tout d'abord, une version non récurrente pour plus de simplicité : `main`

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    if args.len() != 5 {
        eprintln!("Usage: {} FILE PIXELS UPPERLEFT LOWERRIGHT",
                  args[0]);
        eprintln!("Example: {} mandel.png 1000x750 -1.20,0.35 -1,0.20",
                  args[0]);
        std::process::exit(1);
    }

    let bounds = parse_pair(&args[2], 'x')
        .expect("error parsing image dimensions");
    let upper_left = parse_complex(&args[3])
        .expect("error parsing upper left corner point");
    let lower_right = parse_complex(&args[4])
        .expect("error parsing lower right corner point");

    let mut pixels = vec![0; bounds.0 * bounds.1];

    render(&mut pixels, bounds, upper_left, lower_right);

    write_image(&args[1], &pixels, bounds)
        .expect("error writing PNG file");
}
```

Après avoir rassemblé les arguments de ligne de commande dans un vecteur de s, nous analysons chacun d'eux, puis commençons les calculs. `String`

```
let mut pixels = vec![0; bounds.0 * bounds.1];
```

Un appel de macro crée un élément vectoriel long dont les éléments sont initialisés à , de sorte que le code précédent crée un vecteur de zéros dont la longueur est , où est la résolution de l'image analysée à partir de la ligne de commande. Nous utiliserons ce vecteur comme un tableau rectangulaire de valeurs de pixels en niveaux de gris d'un octet, comme illustré à la figure 2-5. `vec![v; n] n v bounds.0 * bounds.1 bounds`

La ligne d'intérêt suivante est la suivante:

```
render(&mut pixels, bounds, upper_left, lower_right);
```

Cela appelle la fonction pour calculer réellement l'image. L'expression emprunte une référence mutable à notre tampon de pixels, permettant de le remplir avec des valeurs d'échelles de gris calculées, même si reste le propriétaire du vecteur. Les arguments restants passent les dimensions de l'image et le rectangle du plan complexe que nous avons choisi de tracer. `render &mut pixels render pixels`

```
write_image(&args[1], &pixels, bounds)
    .expect("error writing PNG file");
```

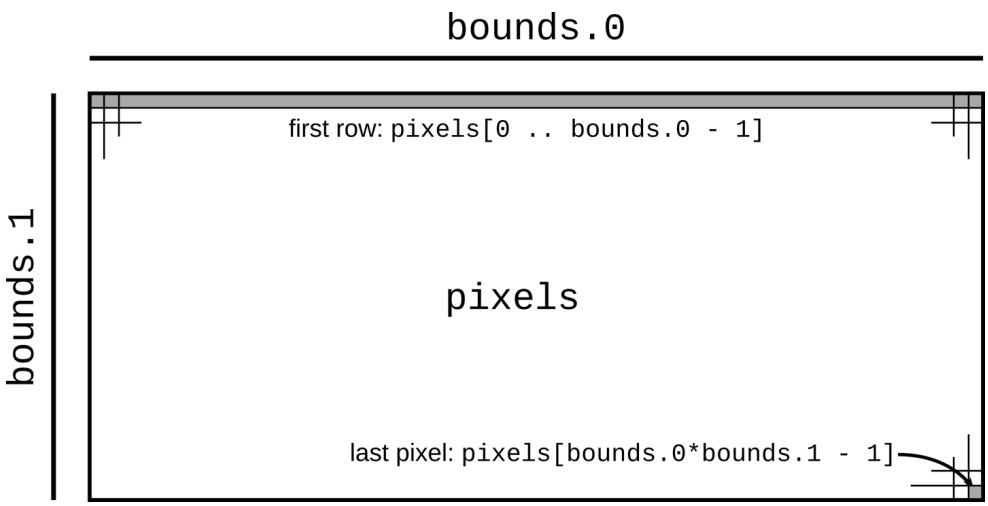

Figure 2-5. Utilisation d'un vecteur comme tableau rectangulaire de pixels

Enfin, nous écrivons le tampon de pixels sur le disque sous forme de fichier PNG. Dans ce cas, nous passons une référence partagée (non modifiable) au tampon, car il ne devrait pas être nécessaire de modifier le contenu du tampon. `write_image`

À ce stade, nous pouvons construire et exécuter le programme en mode release, ce qui permet de nombreuses optimisations puissantes du compilateur, et après plusieurs secondes, il écrira une belle image dans le fichier *mandel.png*:

```
$ cargo build --release
    Updating crates.io index
```

```
    Compiling autocfg v1.0.1
    ...
    Compiling image v0.13.0
    Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
     Finished release [optimized] target(s) in 25.36s
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20
real    0m4.678s
user    0m4.661s
sys     0m0.008s
```

Cette commande doit créer un fichier appelé *mandel.png*, que vous pouvez afficher avec le programme de visualisation d'images de votre système ou dans un navigateur Web. Si tout s'est bien passé, cela devrait ressembler à la figure 2-6.

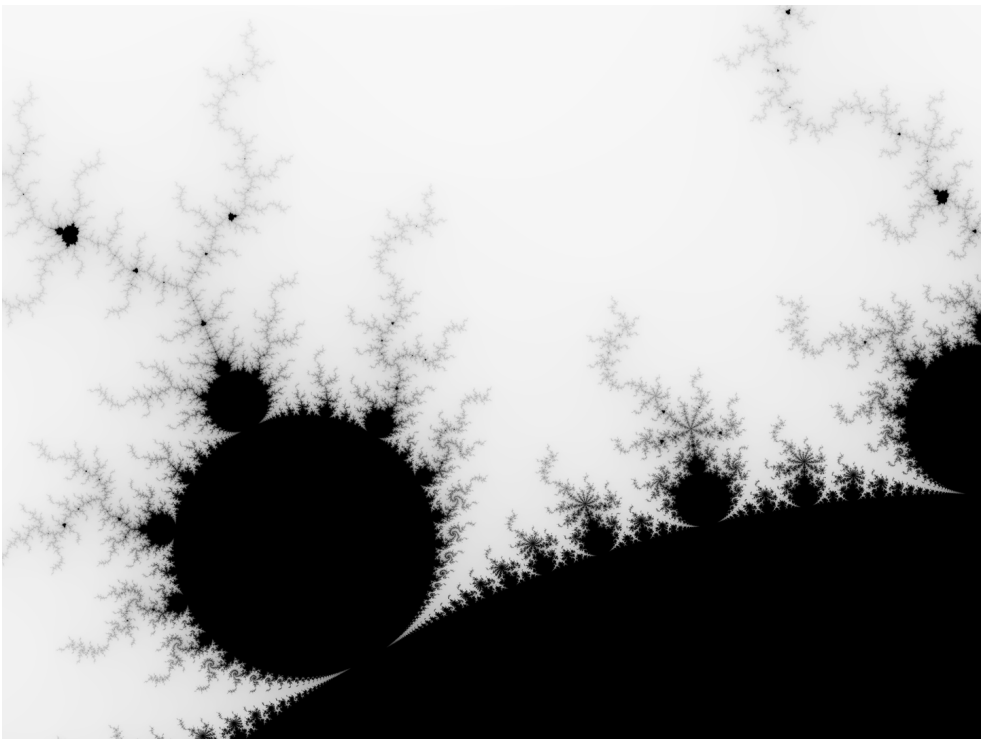

Figure 2-6. Résultats du programme parallèle Mandelbrot

Dans la transcription précédente, nous avons utilisé le programme Unix pour analyser la durée d'exécution du programme: il a fallu environ cinq secondes au total pour exécuter le calcul de Mandelbrot sur chaque pixel de l'image. Mais presque toutes les machines modernes ont plusieurs cœurs de processeur, et ce programme n'en utilisait qu'un. Si nous pouvions répartir le travail sur toutes les ressources informatiques que la machine a à offrir, nous devrions être en mesure de compléter l'image beaucoup plus rapidement. `time`

À cette fin, nous allons diviser l'image en sections, une par processeur, et laisser chaque processeur colorer les pixels qui lui sont attribués. Pour plus de simplicité, nous allons le diviser en bandes horizontales, comme illustré à la figure 2-7. Lorsque tous les processeurs ont terminé, nous pouvons écrire les pixels sur le disque.

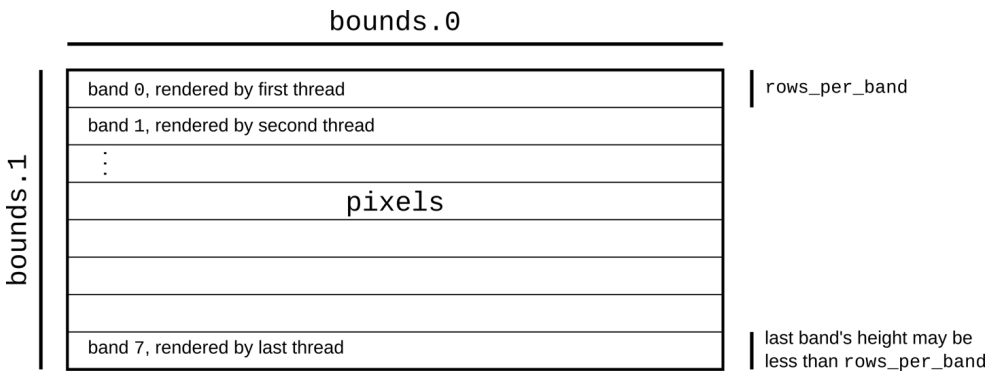

Figure 2-7. Diviser le tampon de pixels en bandes pour un rendu parallèle

La caisse fournit un certain nombre d'installations de concurrence précieuses, y compris une fonction *de thread étendue* qui fait exactement ce dont nous avons besoin ici. Pour l'utiliser, nous devons ajouter la ligne suivante à notre fichier *Cargo.toml*: `crossbeam`

```
crossbeam = "0.8"
```

Ensuite, nous devons supprimer l'appel de ligne unique et le remplacer par ce qui suit: `render`

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;

{
    let bands: Vec<&mut [u8]> =
        pixels.chunks_mut(rows_per_band * bounds.0).collect();
    crossbeam::scope(|spawner| {
        for (i, band) in bands.into_iter().enumerate() {
            let top = rows_per_band * i;
            let height = band.len() / bounds.0;
            let band_bounds = (bounds.0, height);
            let band_upper_left =
                pixel_to_point(bounds, (0, top), upper_left, lower_right);
            let band_lower_right =
                pixel_to_point(bounds, (bounds.0, top + height),
                               upper_left, lower_right);

            spawner.spawn(move |_| {
                render(band, band_bounds, band_upper_left, band_lower_right)
            });
        }
    }).unwrap();
}
```

Décomposer cela de la manière habituelle:

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;
```

Ici, nous décidons d'utiliser huit fils.[1] Ensuite, nous calculons le nombre de lignes de pixels que chaque bande devrait avoir. Nous arrondissons le nombre de rangées vers le haut pour nous assurer que les bandes couvrent toute l'image, même si la hauteur n'est pas un multiple de `. threads`

```
    let bands: Vec<&mut [u8]> =
        pixels.chunks_mut(rows_per_band * bounds.0).collect();
```

Ici, nous divisons le tampon de pixels en bandes. La méthode du tampon renvoie un itérateur produisant des tranches mutables et non superposées du tampon, chacune contenant des pixels, c'est-à-dire des lignes complètes de pixels. La dernière tranche qui produit peut contenir moins de lignes, mais chaque ligne contiendra le même nombre de pixels. Enfin, la méthode de l'itérateur construit un vecteur contenant ces tranches mutables et non superposées. `chunks_mut rows_per_band * bounds.0 rows_per_band chunks_mut collect`

Maintenant, nous pouvons mettre la bibliothèque au travail: `crossbeam`

```
    crossbeam::scope(|spawner| {
        ...
    }).unwrap();
```

L'argument est une fermeture de Rust qui attend un seul argument, . Notez que, contrairement aux fonctions déclarées avec , nous n'avons pas besoin de déclarer les types d'arguments d'une fermeture ; La rouille les déduira, ainsi que son type de retour. Dans ce cas, appelle la fermeture, en passant comme argument une valeur que la fermeture peut utiliser pour créer de nouveaux threads. La fonction attend que tous ces threads terminent l'exécution avant de se retourner. Ce comportement permet à Rust d'être sûr que ces threads n'accéderont pas à leurs parties après qu'il soit sorti de la portée, et nous permet d'être sûr que lorsque les retours, le calcul de l'image est terminé. Si tout se passe bien, retourne , mais si l'un des fils que nous avons engendrés a paniqué, il renvoie un fichier . Nous faisons appel à cela pour que, dans ce cas, nous paniquions aussi, et l'utilisateur recevra un rapport. `|spawner| { ... } spawner fn crossbeam::scope spawner crossbeam::scope pixels crossbeam::scope crossbeam::scope Ok(()) Err unwrap Result`

```
    for (i, band) in bands.into_iter().enumerate() {
```

Ici, nous itérons sur les bandes du tampon de pixels. L'itérateur donne à chaque itération du corps de boucle la propriété exclusive d'une bande, garantissant qu'un seul thread peut y écrire à la fois. Nous expliquons comment cela fonctionne en détail au chapitre 5. Ensuite, l'adaptateur

produit des tuples appairant chaque élément vectoriel avec son index. `into_iter()` `enumerate`

```
    let top = rows_per_band * i;
    let height = band.len() / bounds.0;
    let band_bounds = (bounds.0, height);
    let band_upper_left =
        pixel_to_point(bounds, (0, top), upper_left, lower_right);
    let band_lower_right =
        pixel_to_point(bounds, (bounds.0, top + height),
                       upper_left, lower_right);
```

Compte tenu de l'index et de la taille réelle de la bande (rappelons que la dernière peut être plus courte que les autres), nous pouvons produire un cadre de sélection du type requis, mais qui se réfère uniquement à cette bande du tampon, pas à l'image entière. De même, nous réutilisons la fonction du moteur de rendu pour trouver où les coins supérieur gauche et inférieur droit de la bande tombent sur le plan complexe. `render` `pixel_to_point`

```
    spawner.spawn(move |_| {
        render(band, band_bounds, band_upper_left, band_lower_right);
    });
```

Enfin, nous créons un thread, exécutant la fermeture . Le mot-clé à l'avant indique que cette fermeture s'approprie les variables qu'elle utilise ; en particulier, seule la fermeture peut utiliser la tranche mutable . La liste d'arguments signifie que la fermeture prend un argument, qu'elle n'utilise pas (un autre spawner pour créer des threads imbriqués). `move |_| { ... }` `move band |_|`

Comme nous l'avons mentionné précédemment, l'appel garantit que tous les threads sont terminés avant son retour, ce qui signifie qu'il est sûr d'enregistrer l'image dans un fichier, ce qui est notre prochaine action. `crossbeam::scope`

Exécution du traceur Mandelbrot

Nous avons utilisé plusieurs caisses externes dans ce programme : pour l'arithmétique des nombres complexes, pour l'écriture de fichiers PNG et pour les primitives de création de threads étendus. Voici le fichier *Cargo.toml* final incluant toutes ces dépendances : `num` `image` `crossbeam`

```
[package]
name = "mandelbrot"
version = "0.1.0"
edition = "2021"
```

```
[dependencies]
num = "0.4"
image = "0.13"
crossbeam = "0.8"
```

Avec cela en place, nous pouvons construire et exécuter le programme:

```
$ cargo build --release
    Updating crates.io index
   Compiling crossbeam-queue v0.3.2
   Compiling crossbeam v0.8.1
   Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
    Finished release [optimized] target(s) in #.## secs
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20
real    0m1.436s
user    0m4.922s
sys     0m0.011s
```

Ici, nous avons utilisé à nouveau pour voir combien de temps le programme a pris pour s'exécuter; Notez que même si nous avons quand même passé près de cinq secondes de temps processeur, le temps réel écoulé n'était que d'environ 1,5 seconde. Vous pouvez vérifier qu'une partie de ce temps est consacrée à l'écriture du fichier image en commentant le code qui le fait et en mesurant à nouveau. Sur l'ordinateur portable où ce code a été testé, la version concurrente réduit le temps de calcul de Mandelbrot proprement dit d'un facteur de près de quatre. Nous montrerons comment améliorer considérablement cela au chapitre 19. `time`

Comme précédemment, ce programme aura créé un fichier appelé *mandel.png*. Avec cette version plus rapide, vous pouvez explorer plus facilement l'ensemble mandelbrot en modifiant les arguments de ligne de commande à votre guise.

La sécurité est invisible

En fin de compte, le programme parallèle avec lequel nous nous sommes retrouvés n'est pas substantiellement différent de ce que nous pourrions écrire dans n'importe quel autre langage: nous répartissons des morceaux du tampon de pixels entre les processeurs, laissons chacun travailler sur son morceau séparément, et quand ils ont tous terminé, présentons le résultat. Alors, qu'y a-t-il de si spécial dans la prise en charge de la concurrence de Rust ?

Ce que nous n'avons pas montré ici, ce sont tous les programmes Rust que nous *ne pouvons pas* écrire. Le code que nous avons examiné dans ce chapitre partitionne correctement le tampon entre les threads, mais il existe de nombreuses petites variations de ce code qui ne le font pas (et introduisent donc des courses de données); aucune de ces variantes ne

passera les contrôles statiques du compilateur Rust. Un compilateur C ou C ++ vous aidera joyeusement à explorer le vaste espace des programmes avec des courses de données subtiles; Rust vous dit, dès le départ, quand quelque chose pourrait mal tourner.

Dans les chapitres 4 et 5, nous décrirons les règles de Rust en matière de sécurité de la mémoire. Le chapitre 19 explique comment ces règles garantissent également une bonne hygiène de concurrence.

Systèmes de fichiers et outils de ligne de commande

Rust a trouvé un créneau important dans le monde des outils de ligne de commande. En tant que langage de programmation système moderne, sûr et rapide, il offre aux programmeurs une boîte à outils qu'ils peuvent utiliser pour assembler des interfaces de ligne de commande astucieuses qui répliquent ou étendent les fonctionnalités des outils existants. Par exemple, la commande fournit une alternative prenant en charge la coloration syntaxique avec prise en charge intégrée des outils de pagination et peut automatiquement comparer tout ce qui peut être exécuté avec une commande ou un pipeline. `bat cat hyperfine`

Bien que quelque chose d'aussi complexe soit hors de portée pour ce livre, Rust vous permet de plonger facilement vos orteils dans le monde des applications ergonomiques en ligne de commande. Dans cette section, nous allons vous montrer comment créer votre propre outil de recherche et de remplacement, avec une sortie colorée et des messages d'erreur conviviaux.

Pour commencer, nous allons créer un nouveau projet Rust :

```
$ cargo new quickreplace
     Created binary (application) `quickreplace` package
$ cd quickreplace
```

Pour notre programme, nous aurons besoin de deux autres caisses: pour créer une sortie colorée dans le terminal et pour la fonctionnalité de recherche et de remplacement réelle. Comme précédemment, nous avons mis ces caisses dans *Cargo.toml* pour dire que nous en avons besoin: `text-colorizer regex cargo`

```
[package]
name = "quickreplace"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
```

```
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
text-colorizer = "1"
regex = "1"
```

Les caisses de rouille qui ont atteint la version , comme celles-ci l'ont fait, suivent les règles de « versioning sémantique »: jusqu'à ce que le numéro de version majeur change, les nouvelles versions doivent toujours être des extensions compatibles de leurs prédécesseurs. Donc, si nous testons notre programme par rapport à la version d'une caisse, il devrait toujours fonctionner avec les versions , , et ainsi de suite; mais la version pourrait introduire des changements incompatibles. Lorsque nous demandons simplement la version d'une caisse dans un fichier *Cargo.toml*, Cargo utilisera la dernière version disponible de la caisse avant
`.1.0 1 1.2 1.3 1.4 2.0 "1" 2.0`

L'interface de ligne de commande

L'interface de ce programme est assez simple. Il faut quatre arguments : une chaîne (ou expression régulière) à rechercher, une chaîne (ou expression régulière) pour la remplacer, le nom d'un fichier d'entrée et le nom d'un fichier de sortie. Nous allons commencer notre fichier *main.rs* avec une structure contenant ces arguments :

```
#[derive(Debug)]
struct Arguments {
    target: String,
    replacement: String,
    filename: String,
    output: String,
}
```

L'attribut indique au compilateur de générer du code supplémentaire qui nous permet de formater la structure avec dans . `#`
`[derive(Debug)] Arguments {:?} println!`

Dans le cas où l'utilisateur entre le mauvais nombre d'arguments, il est d'usage d'imprimer une explication concise de la façon d'utiliser le programme. Nous allons le faire avec une fonction simple appelée et tout importer à partir de afin que nous puissions ajouter de la couleur: `print_usage text-colorizer`

```
use text_colorizer::*;

fn print_usage() {
    eprintln!("{} - change occurrences of one string into another",
              "quickreplace".green());
```

```
    eprintln!("Usage: quickreplace <target> <replacement> <INPUT> <OUTPUT>")
}
```

Le simple fait d'ajouter à la fin d'un littéral de chaîne produit une chaîne enveloppée dans les codes d'échappement ANSI appropriés pour s'afficher en vert dans un émulateur de terminal. Cette chaîne est ensuite interpolée dans le reste du message avant d'être imprimée. `.green()`

Maintenant, nous pouvons collecter et traiter les arguments du programme:

```
use std::env;

fn parse_args() -> Arguments {

    let args: Vec<String> = env::args().skip(1).collect();

    if args.len() != 4 {
        print_usage();
        eprintln!("{} wrong number of arguments: expected 4, got {}.",
            "Error:".red().bold(), args.len());
        std::process::exit(1);
    }

    Arguments {
        target: args[0].clone(),
        replacement: args[1].clone(),
        filename: args[2].clone(),
        output: args[3].clone()
    }
}
```

Afin d'obtenir les arguments saisis par l'utilisateur, nous utilisons le même itérateur que dans les exemples précédents. ignore la première valeur de l'itérateur (le nom du programme en cours d'exécution) afin que le résultat ne comporte que les arguments de ligne de commande. `args .skip(1)`

La méthode produit un nombre d'arguments. Nous vérifions ensuite que le bon numéro est présent et, sinon, imprimons un message et repartons avec un code d'erreur. Nous colorisons à nouveau une partie du message et l'utilisons pour rendre le texte plus lourd. Si le bon nombre d'arguments est présent, nous les mettons dans une structure et le retournons. `collect() Vec .bold() Arguments`

Ensuite, nous ajouterons une fonction qui appelle et imprime simplement la sortie: `main parse_args`

```
fn main() {
    let args = parse_args();
```

```
    println!("{:?}", args);
}
```

À ce stade, nous pouvons exécuter le programme et voir qu'il crache le bon message d'erreur:

```
$ cargo run
 Updating crates.io index
Compiling libc v0.2.82
Compiling lazy_static v1.4.0
Compiling memchr v2.3.4
Compiling regex-syntax v0.6.22
Compiling thread_local v1.1.0
Compiling aho-corasick v0.7.15
Compiling atty v0.2.14
Compiling text-colorizer v1.0.0
Compiling regex v1.4.3
Compiling quickreplace v0.1.0 (/home/jimb/quickreplace)
Finished dev [unoptimized + debuginfo] target(s) in 6.98s
Running `target/debug/quickreplace`
quickreplace - change occurrences of one string into another
Usage: quickreplace <target> <replacement> <INPUT> <OUTPUT>
Error: wrong number of arguments: expected 4, got 0
```

Si vous donnez au programme quelques arguments, il imprimera plutôt une représentation de la structure : `Arguments`

```
$ cargo run "find" "replace" file output
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/quickreplace find replace file output`
Arguments { target: "find", replacement: "replace", filename: "file", output
```

C'est un très bon début! Les arguments sont correctement repris et placés dans les parties correctes de la structure. `Arguments`

Lecture et écriture de fichiers

Ensuite, nous avons besoin d'un moyen d'obtenir des données du système de fichiers afin de pouvoir les traiter et les réécrire lorsque nous avons terminé. Rust dispose d'un ensemble robuste d'outils pour l'entrée et la sortie, mais les concepteurs de la bibliothèque standard savent que la lecture et l'écriture de fichiers sont très courantes, et ils l'ont fait à dessein. Tout ce que nous avons à faire est d'importer un module, , et nous avons accès aux fonctions et: `std::fs read_to_string write`

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

`std::fs::read_to_string` renvoie un fichier . Si la fonction réussit, elle produit un fichier . En cas d'échec, il produit un , le type de biblio-

thèque standard pour représenter les problèmes d'E/S. De même, renvoie un : rien dans le cas de réussite, ou les mêmes détails d'erreur si quelque chose ne va pas. `Result<String, std::io::Error> String std::io::Error std::fs::write Result <(), std::io::Error>`

```
fn main() {
    let args = parse_args();

    let data = match fs::read_to_string(&args.filename) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{} failed to read from file '{}': {:?}",
                      "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };

    match fs::write(&args.output, &data) {
        Ok(_) => {},
        Err(e) => {
            eprintln!("{} failed to write to file '{}': {:?}",
                "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };
}
```

Ici, nous utilisons la fonction que nous avons écrite au préalable et passons les noms de fichiers résultants à et . Les instructions sur les sorties de ces fonctions gèrent les erreurs avec élégance, en imprimant le nom du fichier, la raison fournie de l'erreur et une petite touche de couleur pour attirer l'attention de l'utilisateur. `parse_args() read_to_string write match`

Avec cette fonction mise à jour, nous pouvons exécuter le programme et voir que, bien sûr, le contenu des nouveaux et des anciens fichiers est exactement le même: `main`

```
$ cargo run "find" "replace" Cargo.toml Copy.toml
   Compiling quickreplace v0.1.0 (/home/jimb/rust/quickreplace)
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/quickreplace find replace Cargo.toml Copy.toml`
```

Le *programme lit* dans le fichier d'entrée *Cargo.toml, et il écrit* dans le fichier de sortie *Copy.toml*, mais comme nous n'avons écrit aucun code pour réellement trouver et remplacer, rien dans la sortie n'a changé. Nous pouvons facilement vérifier en exécutant la commande, qui ne détecte aucune différence: `diff`

```
$ diff Cargo.toml Copy.toml
```

Rechercher et remplacer

La touche finale pour ce programme est de mettre en œuvre sa fonctionnalité réelle: trouver et remplacer. Pour cela, nous utiliserons la caisse, qui compile et exécute des expressions régulières. Il fournit une structure appelée , qui représente une expression régulière compilée. a une méthode , qui fait exactement ce qu'elle dit: elle recherche dans une chaîne toutes les correspondances de l'expression régulière et remplace chacune par une chaîne de remplacement donnée. Nous pouvons extraire cette logique dans une fonction : `regex Regex Regex replace_all`

```
use regex::Regex;
fn replace(target: &str, replacement: &str, text: &str)
    -> Result<String, regex::Error>
{
    let regex = Regex::new(target)?;
    Ok(regex.replace_all(text, replacement).to_string())
}
```

Notez le type de retour de cette fonction. Tout comme les fonctions de bibliothèque standard que nous avons utilisées précédemment, renvoie un , cette fois avec un type d'erreur fourni par la caisse. `replace Result regex`

`Regex::new` compile le regex fourni par l'utilisateur et peut échouer si une chaîne n'est pas valide. Comme dans le programme Mandelbrot, nous avons l'habitude de court-circuiter en cas d'échec, mais dans ce cas, la fonction renvoie un type d'erreur spécifique à la caisse. Une fois le regex compilé, sa méthode remplace toutes les correspondances par la chaîne de remplacement donnée. `? Regex::new regex replace_all text`

Si trouve des correspondances, il renvoie un nouveau avec ces correspondances remplacées par le texte que nous lui avons donné. Sinon, renvoie un pointeur vers le texte d'origine, évitant ainsi l'allocation et la copie inutiles de mémoire. Dans ce cas, cependant, nous voulons toujours une copie indépendante, nous utilisons donc la méthode pour obtenir un dans les deux cas et renvoyer cette chaîne enveloppée dans , comme dans les autres fonctions. `replace_all String replace_all to_string String Result::Ok`

Maintenant, il est temps d'incorporer la nouvelle fonction dans notre code: `main`

```
fn main() {
    let args = parse_args();
```

```
    let data = match fs::read_to_string(&args.filename) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{} failed to read from file '{}': {:?}",
                "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };

    let replaced_data = match replace(&args.target, &args.replacement, &data
        Ok(v) => v,
        Err(e) => {
            eprintln!("{} failed to replace text: {:?}",
                "Error:".red().bold(), e);
            std::process::exit(1);
        }
    };

    match fs::write(&args.output, &replaced_data) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{} failed to write to file '{}': {:?}",
                "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };
}
```

Avec cette touche finale, le programme est prêt et vous devriez pouvoir le tester:

```
$ echo "Hello, world" > test.txt
$ cargo run "world" "Rust" test.txt test-modified.txt
   Compiling quickreplace v0.1.0 (/home/jimb/rust/quickreplace)
    Finished dev [unoptimized + debuginfo] target(s) in 0.88s
     Running `target/debug/quickreplace world Rust test.txt test-modified.txt

$ cat test-modified.txt
Hello, Rust
```

Et, bien sûr, la gestion des erreurs est également en place, signalant gracieusement les erreurs à l'utilisateur:

```
$ cargo run "[[a-z]" "0" test.txt test-modified.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/quickreplace '[[a-z]' 0 test.txt test-modified.txt
Error: failed to replace text: Syntax(
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
regex parse error:
    [[a-z]
    ^
```

```
error: unclosed character class
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
)
```

Il y a, bien sûr, de nombreuses fonctionnalités manquantes dans cette simple démonstration, mais les fondamentaux sont là. Vous avez vu comment lire et écrire des fichiers, propager et afficher des erreurs et coloriser la sortie pour améliorer l'expérience utilisateur dans le terminal.

Les prochains chapitres exploreront des techniques plus avancées pour le développement d'applications, des collections de données et de la programmation fonctionnelle avec des itérateurs aux techniques de programmation asynchrone pour une concurrence extrêmement efficace, mais d'abord, vous aurez besoin de la base solide du chapitre suivant dans les types de données fondamentaux de Rust.

1 La caisse fournit une fonction qui renvoie le nombre de CPU disponibles sur le système actuel. `num_cpus`