Chapitre 2. Un tour de Rust

Rouillerlance un défi aux auteurs d'un livre comme celui-ci : 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 définis dans le chapitre précédent : une programmation système sûre et performante. Chaque partie du langage est mieux justifiée dans le contexte de tout le reste.

Ainsi, plutôt que de s'attaquer à une fonctionnalité du langage à la fois, nous avons préparé une visite guidée de quelques petits programmes complets, dont chacun présente quelques fonctionnalités supplémentaires du langage, en 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 les *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 plus de rapidité. Cela inclut un exemple de fonction générique, illustre comment gérer quelque chose comme un tampon de pixels et montre le support de Rust pour la concurrence.
- 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 un comportement indéfini avec un impact minimal sur les performances influence la conception de chaque partie du système, des structures de données standard comme 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

Le meilleurLa façon d'installer Rust est d'utiliser `rustup` . Allez sur *https://rustup.rs* et suivez les instructions.

Vous pouvez également vous rendre sur le site Web de Rust pour obtenir des packages prédéfinis.pour Linux, mac OS, et Windows. Rust est également inclus dans certaines distributions de systèmes d'exploitation. Nous préférons `rustup` car c'est un outil de gestion des installations Rust, comme RVM pour Ruby ou NVM pour Node. Par exemple, lorsqu'une nouvelle version de Rust est publiée, vous pourrez mettre à niveau sans aucun clic en tapant `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, `$` est l'invite de commande ; sous Windows, ce serait `C:\>` 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. Prenant chaque commande à tour de rôle :

- `cargo` est la compilation de Rustgestionnaire, gestionnaire de paquets et outil à usage général. Vous pouvez utiliser Cargo pour démarrer un nouveau projet, créer 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 il est parfois utile de l'exécuter directement.
- `rustdoc` est la documentation Rustoutil. Si vous écrivez une documentation dans les commentaires de la forme appropriée dans le code source de votre programme, `rustdoc` vous pouvez créer du code HTML bien formaté à partir d'eux. Par exemple `rustc` , nous laissons habituellement Cargo courir `rustdoc` pour nous.

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

```
$cargo nouveau bonjour
     Created binary (application) `hello` package
```

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

En regardant dans le répertoire de niveau supérieur du package :

```
$ cd bonjour
 $ls
 -latotal 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 paquet. 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 sur d'autres bibliothèques, nous pouvons les enregistrer dans ce fichier, et Cargo se chargera de télécharger, construire et mettre à jour ces bibliothèques pour nous. Nous aborderons le fichier *Cargo.toml* en détail au chapitre 8 .

Cargo a configuré notre package pour une utilisation avec le `git` 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 `--vcs none` à `cargo new` sur la ligne de commande.

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 :

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

Dans Rust, vous n'avez même pas besoin d'écrire votre propre "Hello, World!" programme. Et c'est l'étendue du passe-partout pour un nouveau

programme Rust : deux fichiers, totalisant treize lignes.

Nous pouvons invoquer la `cargo run` commande depuis n'importe quel répertoire du package pour construire et exécuter notre programme :

```
$course de fret
   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, `rustc` , puis a exécuté l'exécutable qu'il a produit. Cargo place l'exécutable dans le sous-répertoire *cible* en haut du package :

```
$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:

```
$cargaison propre
 $../target/debug/hello
bash: ../target/debug/hello: No such file or directory
```

Fonctions de rouille

La syntaxe de Rustest délibérément sans originalité. Si vous êtes familier avec C, C++, Java ou JavaScript, vous pouvez probablement vous y retrouver dans la structure générale d'un programme Rust. Voici une fonction qui calcule le plus grand commun diviseur 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 `fn` mot clé(prononcé "fun") introduit une fonction. Ici, nous définissons une fonction nommée `gcd` , quiprend deux paramètres `n` et `m` , dont chacun est de type `u64` , un entier 64 bits non signé. Le `->` jeton précède le type de retour : notre fonction retourne une `u64` valeur. L'indentation à quatre espaces est de style Rust standard.

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

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

Le corps de la fonction commence par un appel à la `assert!` macro, en vérifiant qu'aucun des arguments n'est nul. Le `!` caractère marque cela comme une invocation de macro, pas un appel de fonction. Comme la `assert` macro en C et C++, Rust `assert!` vérifie que son argument est vrai, et si ce n'est pas le cas, termine le programme avec un message utile incluant l'emplacement source de la vérification défaillante ; ce genre d'arrêt brutal 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 manière dont le programme a été compilé. Il y a aussi une `debug_assert!` macro, dont les assertions sont ignorées lorsque le programme est compilé pour la vitesse.

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

Une `let` déclarationdéclare une variable locale, comme `t` dans notre fonction. Nous n'avons pas besoin d'écrire `t` le type de , tant que Rust peut le déduire de la façon dont la variable est utilisée. Dans notre fonction, le seul type qui fonctionne pour `t` est `u64` , correspondant `m` à et `n` .

Rust ne déduit que les 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 `t` le type de , nous pourrions écrire :

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

Rust a une `return` déclaration, mais la `gcd` fonction n'en a pas besoin. Si le corps d'une fonction se termine par une expression *non* 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 renvoie `x.cos()` comme valeur :

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

Il est typique dans Rust d'utiliser ce formulaire pour établir la valeur de la fonction lorsque le contrôle "tombe à la fin" de la fonction et d'utiliser des `return` instructionsuniquement pour les premiers retours explicites au milieu d'une fonction.

Rédaction et exécution de tests unitaires

Rouillera un support simple pour les tests intégrés au langage. Pour tester notre `gcd` fonction, nous pouvons ajouter ce code à la fin de *src/main.rs* :

```
#[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 `test_gcd` , qui appelle `gcd` et vérifie qu'elle renvoie des valeurs correctes. Le `#[test]` au-dessus de la définition marque `test_gcd` comme une fonction de test, à ignorer dans les compilations normales, mais incluse et appelée automatiquement si nous exécutons notre programme avec la `cargo test` commande. Nous pouvons avoir des fonctions de test dispersées dans notre arbre source, placées à côté du code qu'elles exécutent, et `cargo test` les rassemblerons automatiquement et les exécuterons toutes.

Le `#[test]` marqueur est un exemple d' *attribut* . Les attributssont un système ouvert pour marquer les fonctions et autres déclarations avec des informations supplémentaires, comme 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 de manière conditionnelle (comme `#ifdef` 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.

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

```
$cargaisontest
   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 commandeet afficher leur plus grand diviseur commun, nous pouvons remplacer la `main` fonction dans *src/main.rs* par ce qui suit :

```
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 morceau par morceau :

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

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

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

Passons à la `main` fonction du programme:

```
fn main() {
```

Notre `main` fonction ne renvoie pas de valeur, nous pouvons donc simplement omettre le `->` type de retour and qui devrait normalement suivre la liste des paramètres.

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

Nous déclarons une variable locale mutable `numbers` et l'initialisons à un vecteur vide. `Vec` est le vecteur de croissance de Rusttype, analogue à C++ `std::vector` , une liste Python ou un tableau JavaScript. Même si les vecteurs sont conçus pour être agrandis et rétrécis de manière dynamique, nous devons toujours marquer la variable `mut` pour que Rust nous permette d'ajouter des nombres à la fin.

Le type de `numbers` est `Vec<u64>` , un vecteur de `u64` valeurs, mais comme précédemment, nous n'avons pas besoin de l'écrire. Rust le déduira pour nous, en partie parce que ce que nous poussons sur le vecteur

sont des `u64` valeurs, mais aussi parce que nous passons les éléments du vecteur à `gcd`, qui n'accepte que `u64` des valeurs.

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

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

Le `std::env` module `args` la fonction 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 .

Au-delà de leur utilisation avec des `for` 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 `args` est toujours le nom du programme en cours d'exécution. Nous voulons ignorer cela, nous appelons donc la `skip` méthode de l'itérateur pour produire un nouvel itérateur qui omet cette première valeur.

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

Ici, nous appelons `u64::from_str` pour tenter d'analyser notre argument de ligne de commande `arg` en tant qu'entier 64 bits non signé. Plutôt qu'une méthode que nous invoquons sur une `u64` valeur que nous avons sous la main, il `u64::from_str` y a une fonction associée au `u64` type, semblable à une méthode statique en C++ ou Java. La `from_str` fonction ne renvoie pas `u64` directement, mais plutôt une `Result` valeur qui indique si l'analyse a réussi ou échoué. Une `Result` valeur est l'une des deux variantes suivantes :

- Une valeur écrite `Ok(v)`, indiquant que l'analyse a réussi et `v` est la valeur produite
- Une valeur écrite `Err(e)`, indiquant que l'analyse a échoué et `e` est une valeur d'erreur expliquant pourquoi

Les fonctions qui font tout ce qui pourrait échouer, comme effectuer une entrée ou une sortie ou interagir autrement avec le système d'exploitation, peuvent renvoyer des `Result` typesdont les `Ok` variantes portent des résultats positifs (le nombre d'octets transférés, le fichier ouvert, etc.) et dont les `Err` variantes portent un code d'erreur indiquant ce qui s'est

mal passé. Contrairement à la plupart des langages modernes, Rust n'a pas d'exceptions : toutes les erreurs sont gérées à l'aide de `Result` ou panique, comme indiqué au chapitre 7 .

On utilise `Result` la `expect` méthode depour vérifier le succès de notre analyse. Si le résultat est un `Err(e)` , `expect` imprime un message qui inclut une description de `e` et quitte le programme immédiatement. Cependant, si le résultat est `Ok(v)` , `expect` retourne simplement `v` lui-même, que nous pouvons enfin pousser à la fin de notre vecteur de nombres.

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

Il n'y a pas de plus grand commun diviseur 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. On utilise la `eprintln!` macropour écrire notre message d'erreur dans le flux de sortie d'erreur standard.

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

Cette boucle utilise `d` comme valeur courante, 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 `d` comme mutable afin de pouvoir l'affecter dans la boucle.

La `for` boucle a deux parties surprenantes. D'abord, nous avons écrit `for m in &numbers[1..]` ; à quoi sert l' `&` opérateur ? Deuxièmement, nous avons écrit `gcd(d, *m)` ; a quoi sert le `*` in `*m` ? Ces deux détails sont complémentaires.

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 gère de telles valeurs : il veut laisser le programmeur contrôler la consommation de mémoire, en indiquant clairement la durée de vie de chaque valeur, tout en s'assurant que la mémoire est libérée rapidement lorsqu'elle n'est plus nécessaire.

Ainsi, lorsque nous itérons, nous voulons dire à Rust que *la propriété*du vecteur doit rester avec `numbers` ; nous ne faisons *qu'emprunter*ses éléments pour la boucle. L' `&` opérateuren `&numbers[1..]` emprunte une

*référence*aux éléments du vecteur à partir de la seconde. La `for` boucle itère sur les éléments référencés, laissant `m` emprunter chaque élément successivement. L' `*` opérateurdans `*m` *dereferences* `m` , donnant la valeur à laquelle il se réfère ; c'est la prochaine à laquelle `u64` nous voulons passer `gcd` . Enfin, puisque `numbers` possède le vecteur, Rust le libère automatiquement lorsqu'il `numbers` sort de la portée à la fin de `main` .

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

Continuons notre promenade à travers le programme:

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

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

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

La `cargo run` commande nous permet de passer des arguments à notre programme, afin que nous puissions essayer notre gestion de la ligne de commande :

```
$parcours cargo 4256
    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
$parcours cargo 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
$parcours cargo 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
```

```
$parcours cargo
    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 `rustup` commande installe automatiquement une copie sur votre ordinateur lorsque vous installez Rust lui-même. Vous pouvez consulter la documentation de la bibliothèque standard sur le site Web de Rust ou dans votre navigateur avec la commande:

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

Servir des pages sur le Web

UneL'un des points forts de Rust est la collection de packages de bibliothèques disponibles gratuitement publiés sur le site Web crates.io . La `cargo` commande permet à votre code d'utiliser facilement un package crates.io : il téléchargera la bonne version du package, le compilera et le mettra à jour comme demandé. Un paquet Rust, qu'il s'agisse d'une bibliothèque ou d'un exécutable, est appelé un *crate* ; Cargo et crates.io tirent tous deux leur nom de ce terme.

Pour montrer comment cela fonctionne, nous allons créer un serveur Web simple à l'aide du `actix-web` framework Webcrate, le `serde` crate de sérialisation et divers autres crates dont ils dépendent. Comme le montre la figure 2-1 , notre site Web demandera à l'utilisateur de saisir deux nombres et de calculer leur plus grand diviseur commun.

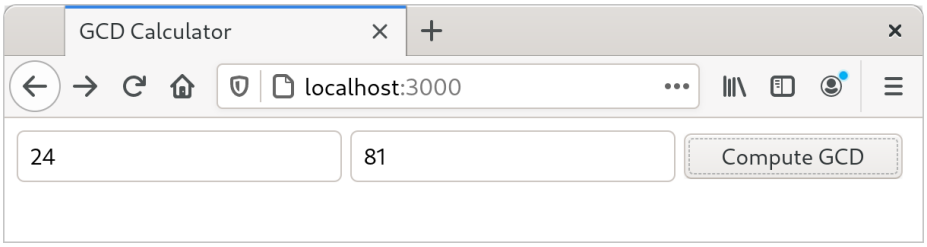

Illustration 2-1. Page Web proposant de calculer GCD

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

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

Ensuite, nous modifierons le fichier *Cargo.toml* de notre nouveau projet pour répertorier les packages que nous souhaitons utiliser. son contenu doit 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 `[dependencies]` section de *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 `1.0.8` de la `actix-web` caisse et la version `1.0` de la `serde` caisse. Il peut bien y avoir des versions de ces caisses sur crates.io plus récentes que celles présentées ici, mais en nommant les versions spécifiques avec lesquelles nous avons testé ce code, nous pouvons nous assurer que le code continuera à se compiler même si de nouvelles versions des packages sont publiées. Nous aborderons la gestion des versions plus en détail au chapitre 8 .

Les caisses peuvent avoir des fonctionnalités facultatives : des parties de l'interface ou de l'implémentation dont tous les utilisateurs n'ont pas besoin, mais qu'il est néanmoins logique d'inclure dans cette caisse. La `serde` caisse offre une manière merveilleusement concise de gérer les données des formulaires Web, mais selon `serde` la documentation de , elle n'est disponible que si nous sélectionnons la `derive` fonctionnalité de la caisse, nous l'avons donc demandée dans notre fichier *Cargo.toml* comme indiqué.

Notez que nous n'avons qu'à nommer les caisses que nous utiliserons directement ; `cargo` se charge d'apporter les autres caisses dont ceux-ci ont besoin à leur tour.

Pour notre première itération, nous garderons le serveur Web simple : il ne servira que la page qui invite l'utilisateur à entrer des nombres avec lesquels calculer. Dans *actix-gcd/src/main.rs* , nous placerons 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 `use` déclaration pour faciliter l'accès à certaines des `actix-web` définitions de la caisse. Lorsque nous écrivons `use actix_web::{...}`, chacun des noms listés à l'intérieur des accolades devient directement utilisable dans notre code ; au lieu d'avoir à épeler le nom complet à `actix_web::HttpResponse` chaque fois que nous l'utilisons, nous pouvons simplement nous y référer comme `HttpResponse`. (Nous arriverons à la `serde` caisse dans un instant.)

Notre `main` fonctionest simple : il appelle `HttpServer::new` à créer un serveur qui répond aux requêtes d'un seul chemin, `"/"` ; imprime un message nous rappelant comment s'y connecter ; puis le met à l'écoute sur le port TCP 3000 sur la machine locale.

L'argument auquel nous passons est l' expression de *fermeture* `HttpServer::new` Rust `|| { App::new() ... }`. 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. C'est `{ ... }` le corps de la fermeture. Lorsque nous démarrons notre serveur, Actix démarre un pool de threads pour gérer les requêtes entrantes. Chaque thread appelle notre fermeture pour obtenir une nouvelle copie de la `App` valeur qui lui indique comment acheminer et gérer les requêtes.

La fermeture appelle `App::new` à créer un nouveau vide `App` puis appelle sa `route` méthodepour ajouter une seule route pour le chemin `"/"`. Le gestionnaire fourni pour cette route, `web::get().to(get_index)`, traite les requêtes HTTP `GET` en appelant la fonction `get_index`. La `route` méthode renvoie la même `App` que

celle sur laquelle elle a été invoquée, maintenant améliorée avec la nouvelle route. Puisqu'il n'y a pas de point-virgule à la fin du corps de la fermeture, `App` est la valeur de retour de la fermeture, prête `HttpServer` à être utilisée par le thread.

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

Étant donné que le texte de la réponse contient de nombreux guillemets doubles, nous l'écrivons en utilisant la syntaxe "chaîne brute" de Rust : la lettre `r`, 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 `\"` n'est reconnue. Nous pouvons toujours nous assurer que la chaîne se termine là où nous le souhaitons en utilisant plus de marques de hachage autour des guillemets que jamais dans le texte.

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

```
$course de fret
    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 précédemment à la figure 2-1.

Malheureusement, cliquer sur Compute GCD ne fait rien, à part naviguer dans notre navigateur vers une page vierge. Corrigeons cela ensuite, en ajoutant une autre route à notre `App` pour gérer la `POST` demande de notre formulaire.

Il est enfin temps d'utiliser la `serde` 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 `use` directive suivante en haut de *src/main.rs* :

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

Les programmeurs Rust rassemblent généralement toutes leurs `use` 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.

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 `GcdParameters` qui a deux champs, `n` et `m`, dont chacun est un `u64` type d'argument que notre `gcd` fonction attend.

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

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 `post_gcd` fonctionprend un argument, `form` , dont le type est `web::Form<GcdParameters>` . Actix sait extraire une valeur de n'importe quel type `web::Form<T>` d'une requête HTTP si, et seulement si, `T` peut être désérialisée à partir de données de formulaire HTML `POST` . Puisque nous avons placé l' `#[derive(Deserialize)]` attribut sur notre `GcdParameters` définition de type, Actix peut le désérialiser à partir des données de formulaire, de sorte que les gestionnaires de requêtes peuvent attendre une `web::Form<GcdParameters>` 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.

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

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

```
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 `route`, établissant `web::post().to(post_gcd)` comme gestionnaire pour le chemin `"/gcd"`.

La dernière pièce restante est la `gcd` fonctionnous avons écrit plus tôt, qui va dans le *fichier actix-gcd/src/main.rs* . Avec cela en place, vous pouvez interrompre tous les serveurs que vous pourriez avoir laissés en cours d'exécution et reconstruire et redémarrer le programme :

```
$course de fret
   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 saisissant quelques chiffres et en cliquant sur le bouton Compute GCD, vous devriez voir des résultats(Figure 2-2).

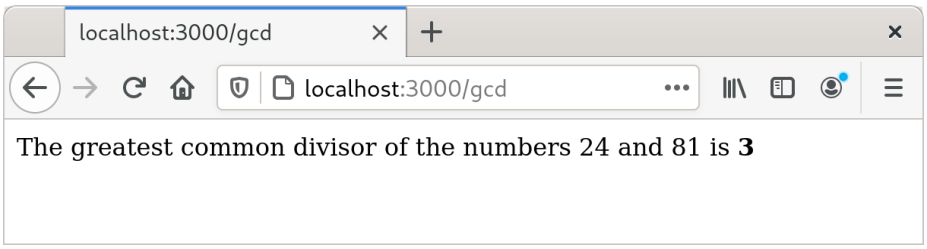

Illustration 2-2. Page Web montrant les résultats du calcul de GCD

Concurrence

UneL'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 apportant des modifications à une structure de données partagée, Rust garantit que vous ne pouvez accéder aux données que lorsque vous maintenez le verrou et libère le verrou automatiquement lorsque vous avez termi-

né. 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 types 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 bien renoncé à tout accès à celle-ci. En C et C++, c'est à vous de vérifier que rien sur le thread émetteur 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 du gestionnaire de requêtes. Si le serveur reçoit des requêtes simultanées, il peut exécuter les fonctions `get_form` et `post_gcd` dans plusieurs fils à la fois. Cela peut être un peu choquant, car nous n'avions certainement pas à l'esprit la concurrence lorsque nous avons écrit ces fonctions. Mais Rust garantit que cela peut être fait en toute sécurité, quelle que soit la complexité de votre serveur : si votre programme compile, il est exempt de courses de données. Toutes les fonctions Rust sont thread-safe.

Le programme de cette section trace le Mandelbrotensemble, 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 threads est si simple ; nous couvrirons des modèles plus complexes au chapitre 19 , mais cette tâche démontre certains des éléments essentiels.

Pour commencer, nous allons créer un nouveau projet Rust :

```
$cargo neuf 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'aborder l'implémentation Mandelbrot simultanée, nous devons décrire le calcul que nous allons effectuer.

Qu'est-ce que l'ensemble de 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 petite excursion dans les mathématiques pures. Nous commencerons par un cas simple, puis ajouterons des détails compliqués jusqu'à ce que nous arrivions au calcul au cœur de l'ensemble de Mandelbrot.

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

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

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

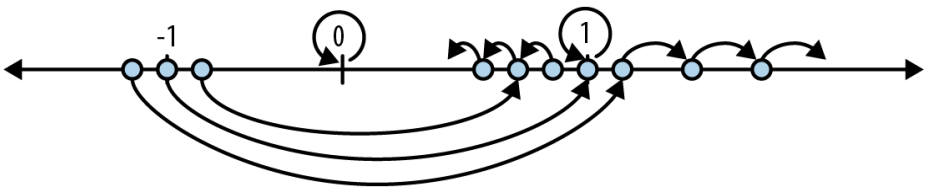

Illustration 2-3. Effets de la quadrature répétée d'un nombre

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

Considérons maintenant une boucle légèrement différente :

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

Cette fois, `x` commence à zéro, et nous ajustons sa progression à chaque itération en ajoutant `c` après l'avoir quadrillé. Cela rend plus difficile de voir comment `x` les tarifs, mais certaines expérimentations montrent que

si `c` est supérieur à 0,25 ou inférieur à -2,0, `x` devient finalement infiniment grand; sinon, il reste quelque part dans le voisinage de zéro.

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

```
[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"
```

Nous pouvons maintenant 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 de l'utiliser `z` pour les nombres complexes, nous avons donc renommé notre variable de bouclage. L'expression `Complex { re: 0.0, im: 0.0 }` est la façon dont nous écrivons le zéro complexe en utilisant le type `num` de la caisse `Complex` . `Complex` est une structure Rusttype (ou *struct*), défini comme ceci :

```
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 `Complex` , avec deux champs, `re` et `im` . `Complex` est une structure *générique* : vous pouvez lire le `<T>` après le nom du type comme "pour tout type `T` ". Par exemple,

`Complex<f64>` est un nombre complexe dont les champs `re` et sont des valeurs, utiliseraient des flottants 32 bits, etc. Compte tenu de cette définition, une expression comme produit une valeur avec son champ initialisé à 0,24 et son champ initialisé à 0,3. `im f64 Complex<f32> Complex { re: 0.24, im: 0.3 } Complex re im`

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

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

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

La boucle infinie prend un certain temps à s'exécuter, mais il existe deux astuces pour les impatients. Premièrement, 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 `z` jamais une fois quitte le cercle de rayon 2 centré à l'origine, il s'envolera définitivement infiniment loin de l'origine. Voici donc la version finale de notre boucle, et le coeur de notre programme :

```
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 the
/// 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 `c` 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 `c` être probablement membre.

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

```
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 `T`, une valeur de type `Option<T>` est soit `Some(v)`, où `v` est une valeur de type `T`, soit `None`, indiquant qu'aucune `T` valeur n'est disponible. Comme le `Complex` type dont nous avons parlé précédemment, `Option` est un type générique : vous pouvez l'utiliser `Option<T>` pour représenter une valeur facultative de n'importe quel type `T` que vous aimez.

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

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

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

L' `z.norm_sqr()` appel de méthode renvoie le carré de `z` la distance de à l'origine. Pour décider si `z` 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.

Vous avez peut-être remarqué que nous utilisons `///` pour marquer les lignes de commentaire au-dessus de la définition de la fonction ; les commentaires au dessus des membres de la `Complex` structure commencent `///` aussi par. Ce sont *des commentaires de documentation* ; l' `rustdoc` 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 la documentation au chapitre 8 .

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 la paire

Le programme prend plusieurs arguments en ligne de commandecontrôler la résolution de l'image que nous allons écrire et la partie de l'ensemble de 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> are
/// 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 `parse_pair` est une *fonction générique*:

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

Vous pouvez lire la clause `<T: FromStr>` à haute voix comme suit : "Pour tout type `T` qui implémente le `FromStr` trait...". Cela nous permet effectivement de définir une famille entière de fonctions à la fois : `parse_pair::<i32>` est une fonctionqui analyse des paires de `i32` valeurs, `parse_pair::<f64>` analyse des paires de valeurs à virgule flottante, etc. Cela ressemble beaucoup à un modèle de fonction en C++. Un programmeur Rust appellerait `T` un *paramètre de type*de `parse_pair`. Lorsque vous utilisez une fonction générique, Rust sera souvent capable de déduire les paramètres de type pour vous, et vous n'aurez pas besoin de les écrire comme nous l'avons fait dans le code de test.

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

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

Le `String` genre `find` méthoderecherche dans la chaîne un caractère qui correspond à `separator`. Si `find` renvoie `None`, ce qui signifie que le caractère séparateur n'apparaît pas dans la chaîne, l' `match` expression entièreest évalué à `None`, indiquant que l'analyse a échoué. Sinon, nous prenons `index` la position du séparateur dans la chaîne.

```
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' `match` expression. L'argument de la correspondance est cette expression de tuple :

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

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

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

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

```
_ => None
```

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

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

```
/// 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 `parse_complex` fonctionappelle `parse_pair`, construit une `Complex` valeur si les coordonnées ont été analysées avec succès et transmet les échecs à son appelant.

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

Mappage des pixels aux nombres complexes

Le programmedoit travailler dans deux espaces de coordonnées liés : chaque pixel de l'image de sortie correspond à un point sur le plan complexe. La relation entre ces deux espaces dépend de la portion du Mandelbrotset que nous allons tracer, et la résolution de l'image demandée, telle que déterminée par les arguments de la ligne de commande. La fonction suivante convertit à partir de *l'espace image*à *l'espace des 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 image
/// 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 `pixel_to_point` 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.0
```

Cela fait référence au premier élément du tuple `pixel`.

```
pixel.0 as f64
```

C'est la syntaxe de Rust pour une conversion de type : cela convertit `pixel.0` en une `f64` valeur. Contrairement à C et C++, Rust refuse généralement de convertir implicitement les types numériques ; vous devez écrire les conversions 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.

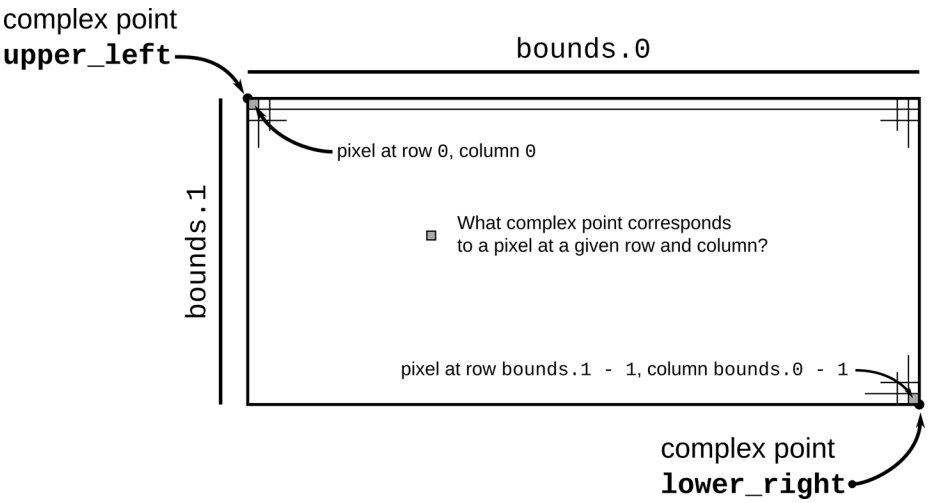

Illustration 2-4. La relation entre le plan complexe et les pixels de l'image

Tracer l'ensemble

Comploterl'ensemble de Mandelbrot, pour chaque pixel de l'image, on applique simplement `escape_time` au point correspondant sur le plan complexe, et on colore le pixel en fonction du résultat :

```
/// 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_right`
/// 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 vous 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 `escape_time` dit que `point` appartient à l'ensemble, `render` colore le pixel correspondant en noir (`0`). Sinon, `render` attribue des couleurs plus foncées aux nombres qui ont mis plus de temps à sortir du cercle.

Écriture de fichiers image

La `image` caissefournit des fonctions pour lire et écrire 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 `image` , ajoutez la ligne suivante à la `[dependencies]` section de *Cargo.toml* :

```
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 : elle ouvre un fichier et essaie d'y écrire l'image. Nous transmettons à l'encodeur les données de pixel réelles de `pixels`, ainsi que sa largeur et sa hauteur de `bounds`, puis un argument final qui indique comment interpréter les octets dans `pixels` : la valeur `ColorType::Gray(8)` indique que chaque octet est une valeur en niveaux de gris de huit bits.

C'est tout simple. Ce qui est intéressant avec cette fonction, c'est la façon dont elle réagit quand 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 de Rust doivent renvoyer une `Result` valeur, qui est soit `Ok(s)` en cas de succès, où `s` est la valeur réussie, soit `Err(e)` en cas d'échec, où `e` est un code d'erreur. Quels sont donc `write_image` les types de réussite et d'erreur de ?

Quand tout va bien, notre `write_image` fonctionn'a aucune valeur utile à renvoyer ; il a écrit tout ce qui est intéressant dans le fichier. Son type de réussite est donc le type *d'unité* `()`, ainsi appelé car il n'a qu'une seule valeur, également écrite `()`. Le type d'unité est similaire à `void` en C et C++.

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

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

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

```
    }
};
```

En cas de succès, laissez `output` être le `File` porté dans la `Ok` valeur. En cas d'échec, transmettre l'erreur à notre propre appelant.

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

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

En cas d' `File::create` échec, l' `?` opérateur revient de `write_image`, transmettant l'erreur. Sinon, `output` contient le fichier `File`.

NOTER

C'est une erreur courante de débutant d'essayer d'utiliser `?` la `main` fonction. Cependant, puisque `main` lui-même ne renvoie pas de valeur, cela ne fonctionnera pas ; à la place, vous devez utiliser une `match` instruction, ou l'une des méthodes abrégéescomme `unwrap` et `expect`. Il y a aussi la possibilité de simplement changer `main` pour retourner un `Result`, que nous aborderons plus tard.

Un programme Mandelbrot simultané

Toutes les piècessont en place, et nous pouvons vous montrer la `main` fonction, où nous pouvons mettre la simultanéité à notre service. Tout d'abord, une version non concurrente pour plus de simplicité :

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

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

Un appel de macro `vec![v; n]` crée un vecteur `n` éléments long dont les éléments sont initialisés à `v` , donc le code précédent crée un vecteur de zéros dont la longueur est `bounds.0 * bounds.1` , où `bounds` 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 .

La ligne d'intérêt suivante est celle-ci :

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

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

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

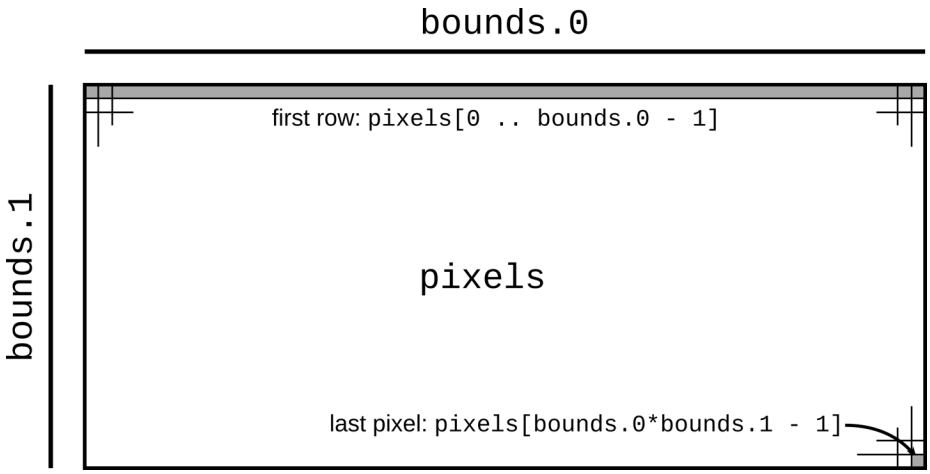

Illustration 2-5. Utilisation d'un vecteur comme tableau rectangulaire de pixels

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 `write_image` nous n'avons pas besoin de modifier le contenu du tampon.

À ce stade, nous pouvons construire et exécuter le programme en mode release, ce qui permet de nombreuses optimisations puissantes du compilateur, et après quelques 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 .

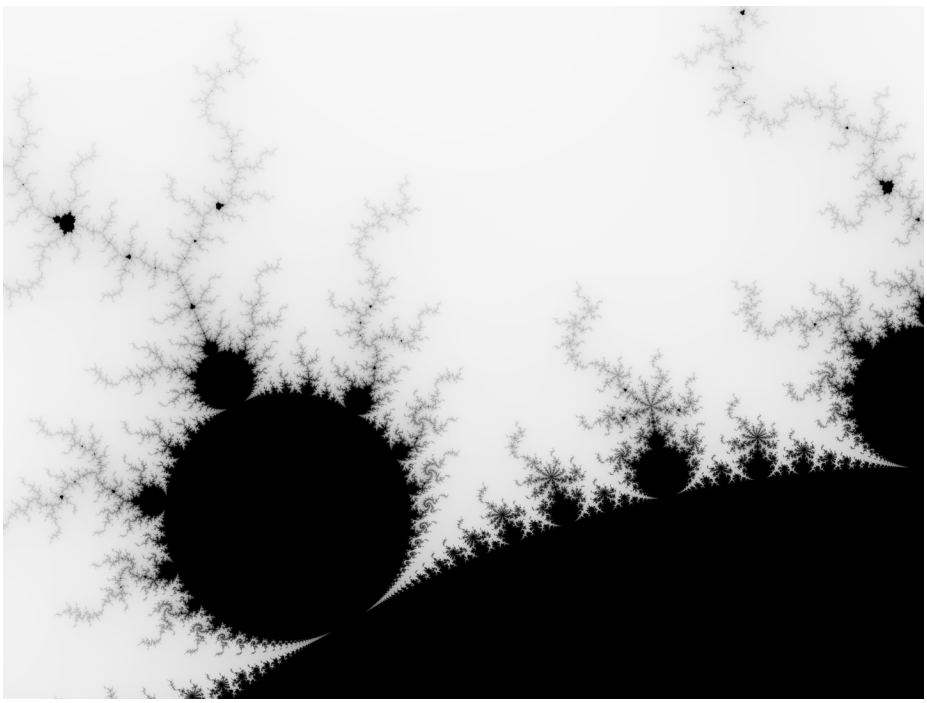

Illustration 2-6. Résultats du programme Mandelbrot parallèle

Dans la transcription précédente, nous avons utilisé le `time` programme Unix pour analyser le temps 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 seul.

Si nous pouvions répartir le travail sur toutes les ressources informatiques que la machine a à offrir, nous devrions pouvoir compléter l'image beaucoup plus rapidement.

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.

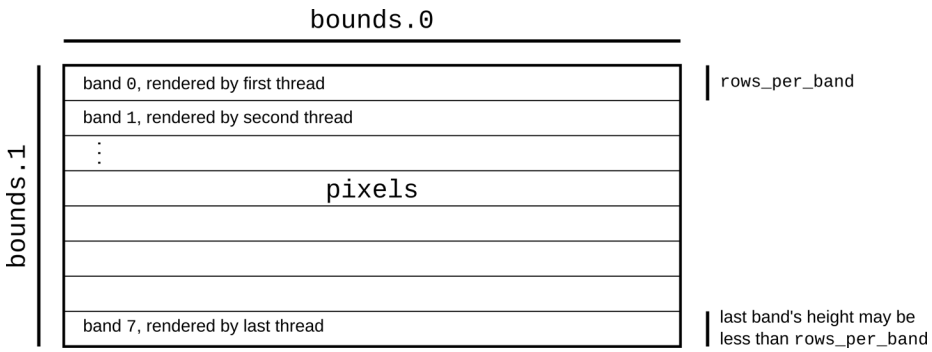

Illustration 2-7. Diviser le tampon de pixels en bandes pour un rendu parallèle

La `crossbeam` caissefournit un certain nombre d'installations de simultanéité utiles, y compris une installation de *thread* à portée qui fait exactement ce dont nous avons besoin ici. Pour l'utiliser, nous devons ajouter la ligne suivante à notre fichier *Cargo.toml* :

```
crossbeam = "0.8"
```

Ensuite, nous devons supprimer l'appel à une seule ligne `render` et le remplacer par ce qui suit :

```
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 threads. [1] Ensuite, nous calculons le nombre de rangées de pixels que chaque bande doit avoir. Nous arrondissons la ligne en comptant 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 `chunks_mut` méthode du tampon renvoie un itérateur produisant des tranches modifiables et non superposées du tampon, chacune contenant `rows_per_band * bounds.0` des pixels, en d'autres termes, des lignes `rows_per_band` complètes de pixels. La dernière tranche qui `chunks_mut` produit peut contenir moins de lignes, mais chaque ligne contiendra le même nombre de pixels. `collect` Enfin, la méthode de l'itérateurconstruit un vecteur contenant ces tranches mutables et non superposées.

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

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

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

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

Ici, nous parcourons les bandes du tampon de pixels. L' `into_iter()` itérateurdonne à chaque itération du corps de la 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' `enumerate` adaptateur produit des tuples associant chaque élément vectoriel à son index.

```
    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);
```

Étant donné l'indice et la taille réelle de la bande (rappelons que la dernière peut être plus courte que les autres), nous pouvons produire une boîte englobante du type `render` requis, mais qui se réfère uniquement à cette bande du tampon, pas à l'ensemble image. `pixel_to_point` De même, nous réaffectons la fonction du moteur de rendupour trouver où les coins supérieur gauche et inférieur droit de la bande tombent sur le plan complexe.

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

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

Comme nous l'avons mentionné précédemment, l' `crossbeam::scope` 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, qui est notre prochaine action..

Exécution du traceur de Mandelbrot

Nous avons utilisé plusieurs caisses externesdans ce programme : `num` pour l'arithmétique des nombres complexes, `image` pour l'écriture de fichiers PNG et `crossbeam` pour les primitives de création de threads

délimitées. Voici le fichier *Cargo.toml* final incluant toutes ces dépendances :

```
[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 `time` de nouveau utilisé pour voir combien de temps le programme a mis à s'exécuter ; notez que même si nous avons encore 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 .

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

La sécurité est invisible

Au final, le programme parallèlenous nous sommes retrouvés avec n'est pas sensiblement 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 fini, présentons le résultat . Alors, qu'y a-t-il de si spécial dans le support de la simultanéité 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 sur ce code qui ne le font pas (et introduisent donc des courses de données); aucune de ces variantes ne passera les vérifications statiques du compilateur Rust. Le compilateur AC 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 pour la sécurité de la mémoire. Le chapitre 19 explique comment ces règles garantissent également une bonne hygiène de la concurrence.

Systèmes de fichiers et outils de ligne de commande

Rouillera trouvé un créneau important dans le monde des outils en 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 reproduisent ou étendent les fonctionnalités des outils existants. Par exemple, la `bat` commandefournit une alternative sensible à la syntaxe `cat` avec prise en charge intégrée des outils de pagination et `hyperfine` peut évaluer automatiquement tout ce qui peut être exécuté avec une commande ou un pipeline.

Bien que quelque chose d'aussi complexe soit hors de portée de ce livre, Rust vous permet de vous plonger facilement dans le monde des applications ergonomiques en ligne de commande. Dans cette section, nous vous montrerons 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 nouveau remplacement
 rapide remplacement      Created binary (application) `quickreplace` package
$ cd rapide
```

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

```
[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 `1.0`, comme celles-ci, suivent les règles de «version sémantique»: jusqu'à ce que le numéro de version majeur `1` change, les nouvelles versions doivent toujours être des extensions compatibles de leurs prédécesseurs. Donc, si nous testons notre programme par rapport à la version `1.2` d'un certain crate, il devrait toujours fonctionner avec les versions `1.3`, `1.4`, etc. mais la version `2.0` pourrait introduire des modifications incompatibles. Lorsque nous demandons simplement la version `"1"` d'une caisse dans un fichier *Cargo.toml*, Cargo utilisera la dernière version disponible de la caisse avant `2.0`.

L'interface de ligne de commande

L'interface de ce programme est assez simple. Il prend quatre arguments : une chaîne (ou une expression régulière) à rechercher, une chaîne (ou une 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' `#[derive(Debug)]` attribut indique au compilateur de générer du code supplémentaire qui nous permet de formater la `Arguments` structure avec `{:?}` in `println!`.

Au cas où l'utilisateur entrerait 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 simpleappelé `print_usage` et importez tout de `text-colorizer` sorte que nous puissions ajouter de la couleur :

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

Nous pouvons maintenant 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 `args` itérateur que dans les exemples précédents. `.skip(1)` ignore la première valeur de l'itérateur (le nom du programme en cours d'exécution) afin que le résultat n'ait que les arguments de la ligne de commande.

La `collect()` méthodeproduit un `Vec` nombre d'arguments. Nous vérifions ensuite que le bon numéro est présent et, si ce n'est pas le cas, imprimons un message et sortons avec un code d'erreur. Nous colorons à nouveau une partie du message et l'utilisons `.bold()` également pour alourdir le texte. Si le bon nombre d'arguments est présent, nous les mettons dans une `Arguments` structure et la renvoyons.

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

```
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 :

```
$course de fret
 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 des arguments au programme, il affichera à la place une représentation de la `Arguments` structure :

```
$"find" "replace"sortie du fichier d'
 exécution de la cargaison    Finished dev [unoptimized + debuginfo] target(s) i
     Running `target/debug/quickreplace find replace file output`
Arguments { target: "find", replacement: "replace", filename: "file", output: "o
```

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

Lecture et écriture de fichiers

Prochain, nous avons besoin d'un moyen d'obtenir les données du système de fichiers afin de pouvoir les traiter et les réécrire lorsque nous avons terminé. Rust dispose d'un ensemble d'outils robustes 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 simplifié à dessein. Tout ce que nous avons à faire est d'importer un module, `std::fs` et nous avons accès aux fonctions `read_to_string` et `write` :

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

`std::fs::read_to_string` renvoie un `Result<String, std::io::Error>`. Si la fonction réussit, elle produit un `String`. S'il échoue, il produit un `std::io::Error`, le type de bibliothèque standard pour représenter les problèmes d'E/S. De même, `std::fs::write` renvoie un `Result<(), std::io::Error>` : rien en cas de succès, ou les mêmes détails d'erreur si quelque chose ne va pas.

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

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

```
$cargo run "find" "replace"Cargo.toml Copie.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 puisque nous n'avons pas écrit de code pour réellement rechercher 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 Copie.toml
```

Trouver et remplacer

Le finaltouch pour ce programme consiste à implémenter sa fonctionnalité réelle : rechercher et remplacer. Pour cela, nous allons utiliser le `regex` crate, qui compile et exécute des expressions régulières. Il fournit une structure appelée `Regex`, qui représente une expression régulière compilée. `Regex` aune méthode `replace_all` quifait exactement ce qu'il dit : il 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 :

```
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, `replace` renvoie un `Result`, cette fois avec un type d'erreur fourni par le `regex` crate.

`Regex::new` compile l'expression régulière fournie par l'utilisateur et peut échouer si une chaîne non valide lui est donnée. Comme dans le programme Mandelbrot, on `?` court-circuite en cas d' `Regex::new` échec, mais dans ce cas la fonction renvoie un type d'erreur spécifique à la `regex` caisse. Une fois l'expression régulière compilée, sa `replace_all` méthode remplace toutes les correspondances `text` avec la chaîne de remplacement donnée.

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

Il est maintenant temps d'intégrer la nouvelle fonction dans notre `main` code :

```
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`

$chat 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 manque bien sûr de nombreuses fonctionnalités à 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 une meilleure 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 des bases solides du chapitre suivant dans les données fondamentales de Rust.les types.

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