Le langage de programmation Rust

par Steve Klabnik et Carol Nichols, avec la participation de la Communauté Rust

Cette version du document suppose que vous utilisez Rust 1.58 (publié le 13/01/2022) ou ultérieur. Voir la section “Installation” du chapitre 1 pour installer ou mettre à jour Rust.

Le format HTML de la version anglaise est disponible en ligne à l'adresse https://doc.rust-lang.org/stable/book/ et en hors-ligne avec l'installation de Rust qui a été effectuée avec `rustup` ; vous pouvez lancer `rustup docs --book` pour l'ouvrir.

Vous avez aussi à votre disposition quelques traductions entretenues par la communauté.

La version anglaise de ce livre est disponible au format papier et e-book chez No Starch Press.

Avant-propos

Cela n'a pas toujours été aussi évident, mais le langage de programmation Rust apporte avant tout plus de *puissance* : peu importe le type de code que vous écrivez en ce moment, Rust vous permet d'aller plus loin et de programmer en toute confiance dans une plus grande diversité de domaines qu'auparavant.

Prenez par exemple la gestion des éléments au “niveau système” qui traite de détails bas niveau de gestion de mémoire, de modèles de données et de concurrence. Traditionnellement, ce domaine de la programmation est jugé ésotérique, compréhensible uniquement par une poignée de personnes qui ont consacré des années d'apprentissage à en déjouer les pièges infâmes. Et même ceux qui travaillent dans ce domaine le font avec beaucoup de prudence, de crainte que leur code ne puisse conduire à des problèmes de sécurité, des plantages ou des corruptions de mémoire.

Rust fait tomber ces obstacles en éliminant les vieux pièges et en apportant un ensemble d'outils soignés et conviviaux pour vous aider sur votre chemin. Les développeurs qui ont besoin de "se plonger" dans le contrôle de plus bas niveau peuvent ainsi le faire avec Rust, sans prendre le risque habituel de plantages ou de failles de sécurité, et sans avoir à apprendre les subtilités d'un enchevêtrement d'outils capricieux. Encore mieux, le langage est conçu pour vous guider naturellement vers un code fiable et efficace en termes de rapidité d'exécution et d'utilisation de la mémoire.

Les développeurs qui travaillent déjà avec du code bas niveau peuvent utiliser Rust pour accroître leurs ambitions. Par exemple, introduire du parallélisme en Rust est une opération à faible risque : le compilateur va détecter les erreurs classiques pour vous. Et vous pourrez vous lancer dans des améliorations plus agressives de votre code avec la certitude que vous n'introduirez pas accidentellement des causes de plantage ou des vulnérabilités.

Mais Rust n'est pas cantonné à la programmation de bas niveau. C'est un langage suffisamment expressif et ergonomique pour rendre les applications en ligne de commande, les serveurs web et bien d'autres types de code agréables à écrire — vous trouverez plus tard des exemples simples de ces types de programmes dans ce livre. Travailler avec Rust vous permet d'acquérir des compétences qui sont transposables d'un domaine à un autre ; vous pouvez apprendre Rust en écrivant une application web, puis appliquer les mêmes notions pour les utiliser avec votre Raspberry Pi.

Ce livre exploite pleinement le potentiel de Rust pour permettre à ses utilisateurs de se perfectionner. C'est une documentation conviviale et accessible destinée à améliorer vos connaissances en Rust, mais aussi à améliorer vos capacités et votre assurance en tant que développeur en général. Alors foncez, préparez-vous à apprendre, et bienvenue dans la communauté Rust !

— Nicholas Matsakis et Aaron Turon

Introduction

> Note : la version anglaise de ce livre est disponible au format papier et ebook chez No Starch Press à cette adresse : The Rust Programming Language

Bienvenue sur *Le langage de programmation Rust*, un livre d'initiation à Rust. Le langage de programmation Rust vous aide à écrire plus rapidement des logiciels plus fiables. L'ergonomie de haut-niveau et la maîtrise de bas-niveau sont souvent en opposition dans la conception des langages de programmation ; Rust remet en cause ce conflit. Grâce à l'équilibre entre ses puissantes capacités techniques et une bonne ergonomie de développement, Rust vous donne la possibilité de contrôler les détails de bas-niveau (comme l'utilisation de la mémoire) sans tous les soucis traditionnellement associés à ce genre de pratique.

À qui s'adresse Rust

Rust est idéal pour de nombreuses personnes pour diverses raisons. Analysons quelques-uns des groupes les plus importants.

Équipes de développeurs

Rust se révèle être un outil productif pour la collaboration entre de grandes équipes de développeurs ayant différents niveaux de connaissances en programmation système. Le code de bas-niveau est sujet à une multitude de bogues subtils, qui, dans la plupart des autres langages, ne peuvent être prévenus qu'au moyen de campagnes de test étendues et de minutieuses revues de code menées par des développeurs chevronnés. Avec Rust, le compilateur joue le rôle de gardien en refusant de compiler du code qui comprend ces bogues discrets et vicieux, y compris les bogues de concurrence. En travaillant avec le compilateur, l'équipe peut se concentrer sur la logique du programme plutôt que de traquer les bogues.

Rust offre aussi des outils de développement modernes au monde de la programmation système :

- Cargo, l'outil intégré de gestion de dépendances et de compilation, qui uniformise et facilite l'ajout, la compilation, et la gestion des dépendances dans l'écosystème Rust.
- Rustfmt, qui assure une cohérence de style de codage pour tous les développeurs.

- Le *Rust Langage Server* alimente les environnements de développement intégrés (IDE) pour la complétion du code et l'affichage direct des messages d'erreur.

En utilisant ces outils ainsi que d'autres dans l'écosystème Rust, les développeurs peuvent être plus productifs quand ils écrivent du code système.

Étudiants

Rust est conçu pour les étudiants et ceux qui s'intéressent à l'apprentissage des concepts système. En utilisant Rust, de nombreuses personnes ont appris des domaines comme le développement de systèmes d'exploitation. La communauté est très accueillante et répond volontiers aux questions des étudiants. Grâce à des initiatives comme ce livre, les équipes de Rust veulent rendre les notions système accessibles au plus grand nombre, particulièrement à ceux qui débutent dans la programmation.

Entreprises

Des centaines d'entreprises, petites et grosses, utilisent Rust en production pour différentes missions. Ils l'utilisent pour des outils en ligne de commande, des services web, des outils DevOps, des systèmes embarqués, de l'analyse et de la conversion audio et vidéo, des cryptomonnaies, de la bio-informatique, des moteurs de recherche, de l'internet des objets *(IoT)*, de l'apprentissage automatique *(marchine learning)*, et même des parties importantes du navigateur internet Firefox.

Développeurs de logiciel libre

Rust est ouvert aux personnes qui veulent développer le langage de programmation Rust, la communauté, les outils de développement et les bibliothèques. Nous serions ravis que vous contribuiez au langage Rust.

Les personnes qui recherchent la rapidité et la stabilité

Rust est une solution pour les personnes qui chérissent la rapidité et la stabilité dans un langage. Par rapidité, nous entendons la vitesse des programmes que vous pouvez créer avec Rust et la rapidité avec laquelle Rust vous permet de les écrire. Les vérifications du compilateur de Rust assurent la stabilité durant l'ajout de fonctionnalités ou le remaniement du code. Cela le démarque des langages qui ne font pas ces contrôles sur du code instable que le programme a hérité avec le temps, et que bien souvent les développeurs ont peur de

modifier. En s'efforçant de mettre en place des abstractions sans coût, des fonctionnalités de haut-niveau qui compilent vers du code bas-niveau aussi rapide que s'il avait été écrit à la main, Rust fait en sorte que le code sûr soit aussi du code rapide.

Le langage Rust espère aider beaucoup d'autres utilisateurs ; ceux cités ici ne font partie que d'un univers bien plus grand. Globalement, la plus grande ambition de Rust est d'éradiquer les compromis auxquels les développeurs se soumettaient depuis des décennies en leur apportant sécurité *et* productivité, rapidité *et* ergonomie. Essayez Rust et vérifiez si ses décisions vous conviennent.

À qui est destiné ce livre

Ce livre suppose que vous avez écrit du code dans un autre langage de programmation mais ne suppose pas lequel. Nous avons essayé de rendre son contenu le plus accessible au plus grand nombre d'expériences de programmation possible. Nous ne nous évertuons pas à nous questionner sur *ce qu'est* la programmation ou comment l'envisager. Si vous êtes débutant en programmation, vous seriez mieux avisé en lisant un livre qui vous initie à la programmation.

Comment utiliser ce livre

Globalement, ce livre est prévu pour être lu dans l'ordre. Les chapitres suivants s'appuient sur les notions abordées dans les chapitres précédents, et lorsque les chapitres précédents ne peuvent pas approfondir un sujet, ce sera généralement fait dans un chapitre suivant.

Vous allez rencontrer deux différents types de chapitres dans ce livre : les chapitres théoriques et les chapitres de projet. Dans les chapitres théoriques, vous allez apprendre un sujet à propos de Rust. Dans un chapitre de projet, nous allons construire ensemble des petits programmes, pour appliquer ce que vous avez appris précédemment. Les chapitres 2, 12 et 20 sont des chapitres de projet ; les autres sont des chapitres théoriques.

Le chapitre 1 explique comment installer Rust, comment écrire un programme "Hello, world!" et comment utiliser Cargo, le gestionnaire de paquets et outil de compilation. Le chapitre 2 est une initiation pratique au langage Rust. Nous y aborderons des concepts de haut-niveau, et les chapitres suivants apporteront plus de détails. Si vous voulez vous *salir les mains* tout de suite, le chapitre 2 est l'endroit pour cela. Au début, vous pouvez même sauter le chapitre 3, qui aborde les fonctionnalités de Rust semblables aux autres langages de programmation, et passer directement au chapitre 4 pour en savoir plus sur le système

de possession *(ownership)* de Rust. Toutefois, si vous êtes un apprenti particulièrement minutieux qui préfère apprendre chaque particularité avant de passer à la suivante, vous pouvez sauter le chapitre 2 et passer directement au chapitre 3, puis revenir au chapitre 2 lorsque vous souhaitez travailler sur un projet en appliquant les notions que vous avez apprises.

Le chapitre 5 traite des structures et des méthodes, et le chapitre 6 couvre les énumérations, les expressions `match`, et la structure de contrôle `if let`. Vous emploierez les structures et les énumérations pour créer des types personnalisés avec Rust.

Au chapitre 7, vous apprendrez le système de modules de Rust et les règles de visibilité, afin d'organiser votre code et son interface de programmation applicative (API) publique. Le chapitre 8 traitera des structures de collections de données usuelles fournies par la bibliothèque standard, comme les vecteurs, les chaînes de caractères et les tables de hachage *(hash maps)*. Le chapitre 9 explorera la philosophie et les techniques de gestion d'erreurs de Rust.

Le chapitre 10 nous plongera dans la généricité, les *traits* et les durées de vie, qui vous donneront la capacité de créer du code qui s'adapte à différents types. Le chapitre 11 traitera des techniques de test, qui restent nécessaires malgré les garanties de sécurité de Rust, pour s'assurer que la logique de votre programme est valide. Au chapitre 12, nous écrirons notre propre implémentation d'un sous-ensemble des fonctionnalités du programme en ligne de commande `grep`, qui recherche du texte dans des fichiers. Pour ce faire, nous utiliserons de nombreuses notions abordées dans les chapitres précédents.

Le chapitre 13 explorera les fermetures *(closures)* et itérateurs : ce sont les fonctionnalités de Rust inspirées des langages de programmation fonctionnels. Au chapitre 14, nous explorerons plus en profondeur Cargo et les bonnes pratiques pour partager vos propres bibliothèques avec les autres. Le chapitre 15 parlera de pointeurs intelligents qu'apporte la bibliothèque standard et des *traits* qui activent leurs fonctionnalités.

Au chapitre 16, nous passerons en revue les différents modes de programmation concurrente et comment Rust nous aide à développer dans des tâches parallèles sans crainte. Le chapitre 17 comparera les fonctionnalités de Rust aux principes de programmation orientée objet, que vous connaissez peut-être.

Le chapitre 18 est une référence sur les motifs et le filtrage de motif *(pattern matching)*, qui sont des moyens puissants permettant de communiquer des idées dans les programmes Rust. Le chapitre 19 contient une foultitude de sujets avancés intéressants, comme le code Rust non sécurisé *(unsafe)*, les macros et plus de détails sur les durées de vie, les *traits*, les types, les fonctions et les fermetures *(closures)*.

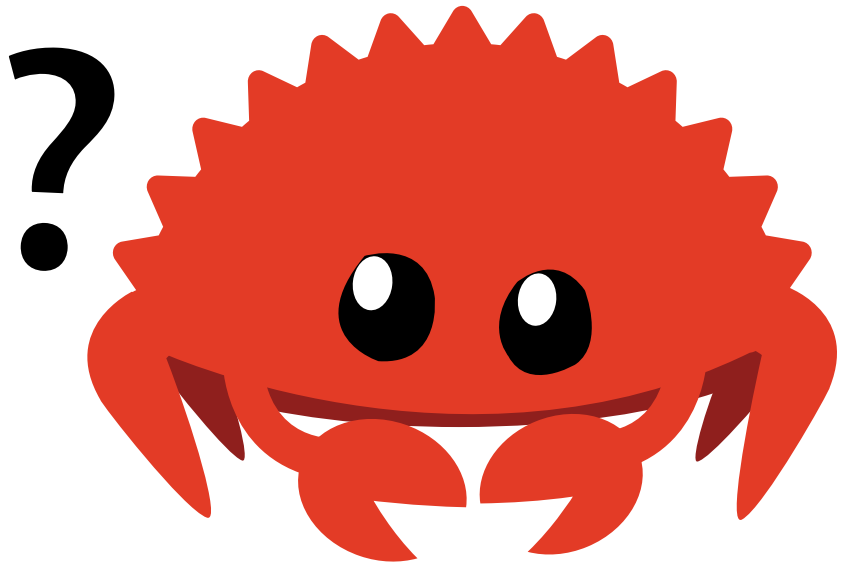
Au chapitre 20, nous terminerons un projet dans lequel nous allons implémenter en bas-

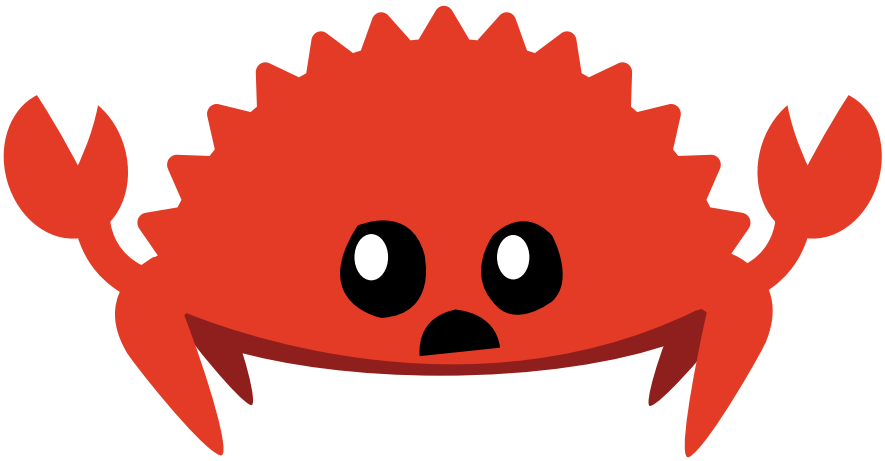
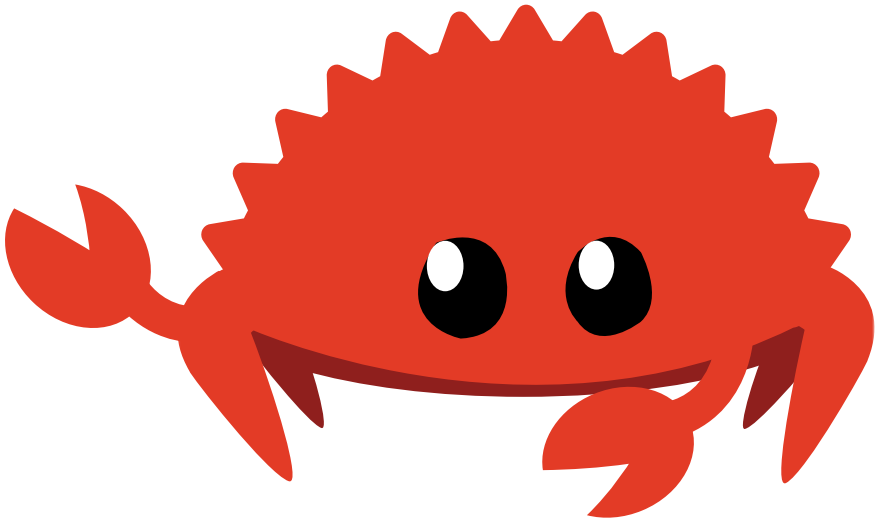
niveau un serveur web multitâches !

Et finalement, quelques annexes qui contiennent des informations utiles sur le langage sous forme de référentiels qui renvoient à d'autres documents. L'annexe A liste les mots-clés de Rust, l'annexe B couvre les opérateurs et symboles de Rust, l'annexe C parle des *traits* dérivables qu'apporte la bibliothèque standard, l'annexe D référence certains outils de développement utiles, et l'annexe E explique les différentes éditions de Rust.

Il n'y a pas de mauvaise manière de lire ce livre : si vous voulez sauter des étapes, allez-y ! Vous devrez alors peut-être revenir sur les chapitres précédents si vous éprouvez des difficultés. Mais faites comme bon vous semble.

Une composante importante du processus d'apprentissage de Rust est de comprendre comment lire les messages d'erreur qu'affiche le compilateur : ils vous guideront vers du code correct. Ainsi, nous citerons de nombreux exemples qui ne compilent pas, avec le message d'erreur que le compilateur devrait vous afficher dans chaque cas. C'est donc normal que dans certains cas, si vous copiez et exécutez un exemple au hasard, il ne compile pas ! Assurez-vous d'avoir lu le texte autour pour savoir si l'exemple que vous tentez de compiler doit échouer. Ferris va aussi vous aider à identifier du code qui ne devrait pas fonctionner :

Ferris	Signification
	Ce code ne compile pas !

Ferris	Signification
	Ce code panique !
	Ce code ne se comporte pas comme voulu.

Dans la plupart des cas, nous vous guiderons vers la version du code qui devrait fonctionner.

Code source

Les fichiers du code source qui a généré ce livre en anglais sont disponibles sur GitHub.

La version française est aussi disponible sur GitHub.

Traduction des termes

Voici les principaux termes techniques qui ont été traduits de l'anglais vers le français.

Anglais	Français	Remarques
adaptor	adaptateur	-
ahead-of-time compilation	compilation anticipée	sigle : AOT
alias	alias	-
allocated	alloué	-
angle bracket	chevrons	-
annotate	indiquer	-
anti-pattern	anti-patron	-
Appendix	annexe	tout en minuscule (sauf en début de phrase)
append	ajouter	-
Application Programming Interface (API)	interface de programmation applicative (API)	-
assertion	vérification	-
assign	assigner	-
argument	argument / paramètre	-
arm	branche	dans une expression `match`
array	tableau	-
artifact	artéfact	-
associated function	fonction associée	-
attribute	attribut	-
backend	application dorsale	-
backtrace	retraçage	-
benchmark	benchmark	-
binary crate	crate binaire	s'utilise au féminin
buffer overread	lecture hors limites	-
n-bit number	nombre encodé sur *n* bits	-
blanket implementation	implémentation générale	-

Anglais	Français	Remarques
blob	blob	-
boilerplate code	code standard	-
boolean	booléen	-
borrow	emprunt(er)	-
borrow checker	vérificateur d'emprunt	-
box	boite	-
buffer overread	sur-lecture de tampon	-
bug	bogue	-
build	compilation	-
build system	système de compilation	-
byte	octet	-
Cargo	Cargo	-
catchall value	valeur passe-partout	-
channel	canal	-
Chapter	chapitre	tout en minuscule (sauf en début de phrase)
CI system	système d'Intégration Continue	-
clause	clause	-
cleanup	nettoyage	-
closure	fermeture	-
code review	revue de code	-
coercion	extrapolation	-
collection	collection	-
command	commande	dans un terminal
commit	commit	-
compound	composé	-
concept chapter	chapitre théorique	-
concurrency	concurrence	-
concurrent	concurrent	-
concurrent programming	programmation concurrente	-
conditional	structure conditionnelle	-

Anglais	Français	Remarques
cons list	liste de construction	-
constant	constant / constante	-
construct	instruction	-
constructor	constructeur	-
consuming adaptor	adaptateur de consommation	-
control flow construct	structure de contrôle	-
core of the error	message d'erreur	-
corruption	corruption / être corrompu	-
CPU	processeur	-
crash	plantage	-
crate	crate	nom féminin (une *crate*)
curly bracket	accolade	-
dangling	pendouillant	-
data race	accès concurrent	-
data representation	modèle de données	-
deadlock	interblocage	-
deallocate	désalloué	-
debug	déboguer	-
debugging	débogage	-
deep copy	copie en profondeur	-
dependency	dépendance	-
deref coercion	extrapolation de déréférencement	-
dereference operator	opérateur de déréférencement	-
dereferencing	déréférencement	-
design pattern	patron de conception	-
destructor	destructeur	-
destructure	déstructurer	-
DevOps	DevOps	-
directory	dossier	-

Anglais	Français	Remarques
dot notation	la notation avec un point	-
double free	double libération	-
drop	libérér	-
elision	élision	-
enum	énumération	-
enumeration	énumération	-
enum’s variant	variante d'énumération	-
exploit	faille	-
expression	expression	-
field	champ	d'une structure
Figure	Illustration	-
flag	drapeau	pour les programmes en ligne de commande
float	nombre à virgule flottante	-
floating-point number	nombre à virgule flottante	-
framework	environnement de développement	-
frontend	interface frontale	-
fully qualified syntax	syntaxe totalement définie	-
function	fonction	-
functional programming	programmation fonctionnelle	-
garbage collector	ramasse-miettes	-
generics	génériques / généricité	-
generic type parameter	paramètre de type générique	-
getter	accesseur	-
glob	global	opérateur
global scope	portée globale	-
grapheme cluster	groupe de graphèmes	-
green thread	tâche virtuelle	-
guessing game	jeu de devinettes	-
handle	référence abstraite	-

Anglais	Français	Remarques
hash	hash / relatif au hachage	-
hash map	table de hachage	-
heap	tas	-
Hello, world!	Hello, world!	-
high-level	haut niveau	-
identifier	identificateur	-
idiomatic	idéal	-
immutability	immuabilité	-
immutable	immuable	-
index	indice	-
indexing	indexation	-
input/output	entrée/sortie	sigle : IO
instance	instance	-
instantiate	instancier	créer une instance
integer literal	littéral d'entiers	-
integer overflow	dépassement d'entier	-
Integrated Development Environment (IDE)	environnement de développement intégré (IDE)	-
interior mutability	mutabilité interne	-
interrupt signal	signal d'arrêt	-
invalidate	neutraliser	-
IOT	internet des objets (IOT)	-
iterator	itérateur	-
iterator adaptor	adaptateur d'itération	-
job	mission	-
just-in-time compilation	compilation à la volée	sigle : JIT
keyword	mot-clé	-
lazy	évaluation paresseuse	comportement d'un itérateur
legacy code	code instable que le programme a hérité avec le temps	-

Anglais	Français	Remarques
library	bibliothèque	-
library crate	crate de bibliothèque	s'utilise au féminin
lifetime	durée de vie	-
linker	linker	-
linter	analyse statique	-
literal value	valeur littérale	-
Listing	encart	tout en minuscule (sauf en début de phrase)
loop	boucle	-
low-level	bas niveau	-
machine learning	apprentissage automatique	-
macro	macro	-
main	main	-
map	tableau associatif	-
match guard	contrôle de correspondance	-
memory leak	fuite de mémoire	-
memory management	gestion de mémoire	-
message-passing	passage de messages	-
method	méthode	-
mock object	mock object	-
modern	récent	-
module	module	-
module system	système de modules	-
monomorphization	monomorphisation	-
move	déplacement	-
mutability	mutabilité	-
mutable	mutable	modifiable
mutate	muter	-
namespace	espace de nom	-
namespacing	l'espace de nom	-
nested (path)	(chemin) imbriqué	-

Anglais	Français	Remarques
newtype pattern	motif newtype	-
nightly Rust	version expérimentale de Rust	-
Note	remarque	tout en minuscule (sauf en début de phrase)
numerical characters	chiffres	-
object-oriented language	langage orienté objet	-
operating system	système d'exploitation	-
output	sortie	-
overload	surcharge	-
owner	propriétaire	-
ownership	possession	-
package manager	système de gestion de paquets	-
panic	panique(r)	-
parallel programming	parallélisme	-
parallelism	parallélisme	-
parameter	paramètre	-
parse	interpréter	-
PATH	PATH	-
pattern	motif	-
pattern-matching	filtrage par motif	-
placeholder	espace réservé	`{}` pour `fmt`
pointer	pointeur	-
popping off the stack	dépiler	-
prelude	étape préliminaire	-
primitive obsession	obsession primitive	-
privacy	visibilité	en parlant des éléments d'un module
procedural macro	macro procédurale	-
process	processus	-
project chapter	chapitre de projet	-

Anglais	Français	Remarques
propagate	propager	-
pushing onto the stack	empiler	-
race condition	situation de concurrence	-
raw identifier	identificateur brut	-
README	README	-
recursive type	type récursif	-
refactoring	remaniement	-
reference	référence	-
reference counting	compteur de références	-
reference cycle	boucle de références	-
release	publication	-
registry	registre	-
regression	régression	-
release	publication	-
remainder	modulo	opération `%`
reproducible build	compilation reproductible	-
Resource Acquisition Is Initialization (RAII)	l'acquisition d'une ressource est une initialisation (RAII)	-
return	retourner	-
run	exécuter	pour les programmes
Rustacean	Rustacé	-
section header	entête de section	-
semantic version	version sémantique	-
scalar	scalaire	-
scope	portée	-
script	script	-
secret	secret	-
section header	en-tête de section	-
semantic version	version sémantique	-
semantic versioning	versionnage sémantique	abréviation : SemVer
shadow	masquer	remplacer une variable par une autre de même

Anglais	Français	Remarques
		nom
shadowing	masquage	-
shallow copy	copie superficielle	-
shell	terminal / invite de commande	-
shorthand	abréviation	-
sidebar	volet latéral	-
signature	signature	d'une fonction
signed	signé	-
slash	barre oblique	-
slice	slice	-
smart pointer	pointeur intelligent	-
snake case	snake case	-
snip	partie masquée ici	dans un encart
space	espace	ce mot est féminin quand on parle du caractère typographique
square brackets	crochets	-
stack	pile	-
stack overflow	débordement de pile	-
standard	standard *(adj. inv.)* / norme *(n.f.)*	-
standard error	erreur standard	-
standard input	entrée standard	-
standard library	bibliothèque standard	-
standard output	sortie standard	-
statement	instruction	-
statically typed	statiquement typé	-
string	chaîne de caractères	-
string literal	un littéral de chaîne de caractères	-
`String`	`String`	nom féminin (une `String`)
struct	structure	-

Anglais	Français	Remarques
submodule	sous-module	-
supertrait	supertrait	-
syntax sugar	sucre syntaxique	-
systems concept	notion système	-
systems-level	niveau système	-
systems-level code	code système	-
terminal	terminal	-
test double	double de test	-
thread	tâche	-
thread pool	groupe de tâches	-
token	jeton	-
trait	trait	-
trait bound	trait lié	-
trait object	objet trait	-
tree	arborescence	-
troubleshooting	dépannage	-
tuple	tuple	-
tuple struct	structure tuple	-
tuple enum	énumération tuple	-
type	type	-
type annotation	annotation de type	-
type inference	inférence de types	-
two’s complement	complément à deux	-
two’s complement wrapping	rebouclage du complément à deux	-
underlying operating system	système d'exploitation sous-jacent	-
underscore	tiret bas	le caractère `_`
unit-like struct	structure unité	-
unit type	type unité	le `()`
unit value	valeur unité	-
unrolling	déroulage	pour une boucle à taille connue à la compilation

Anglais	Français	Remarques
unsafe	non sécurisé	-
unsigned	sans signe (toujours positif)	-
unsigned	non signé	-
unwind	dérouler	(la pile)
user input	saisie utilisateur	-
variable	variable	-
variant	variante	d'une énumération
vector	vecteur	-
version control system (VCS)	système de gestion de versions (VCS)	-
vertical pipe	barre verticale	la barre `
warning	avertissement	-
weak reference	référence faible	-
wildcard	joker	-
worker	opérateur	-
workspace	espace de travail	-
yank	déprécier	-
zero-cost abstraction	abstraction sans coût	-

Prise en main

Démarrons notre périple avec Rust ! Il y a beaucoup à apprendre, mais chaque aventure doit commencer quelque part. Dans ce chapitre, nous allons aborder :

- L'installation de Rust sur Linux, macOS et Windows
- L'écriture d'un programme qui affiche `Hello, world!`
- L'utilisation de `cargo`, le gestionnaire de paquets et système de compilation de Rust

Installation

La première étape consiste à installer Rust. Nous allons télécharger Rust via `rustup`, un outil en ligne de commande conçu pour gérer les versions de Rust et les outils qui leur sont associés. Vous allez avoir besoin d'une connexion Internet pour le téléchargement.

> Note : si vous préférez ne pas utiliser `rustup` pour une raison ou une autre, vous pouvez vous référer à la page des autres moyens d'installation de Rust pour d'autres méthodes d'installation.

L'étape suivante est d'installer la dernière version stable du compilateur Rust. La garantie de stabilité de Rust assurera que tous les exemples dans le livre qui se compilent bien vont continuer à se compiler avec les nouvelles versions de Rust. La sortie peut varier légèrement d'une version à une autre, car Rust améliore souvent les messages d'erreur et les avertissements. En résumé, toute nouvelle version stable de Rust que vous installez de cette manière devrait fonctionner en cohérence avec le contenu de ce livre.

> ### La notation en ligne de commande
>
> Dans ce chapitre et les suivants dans le livre, nous allons montrer quelques commandes tapées dans le terminal. Les lignes que vous devrez écrire dans le terminal commencent toutes par `$`. Vous n'avez pas besoin d'écrire le caractère `$` ; il marque le début de chaque commande. Les lignes qui ne commencent pas par `$` montrent généralement le résultat de la commande précédente. De plus, les exemples propres à PowerShell utiliseront `>` plutôt que `$`.

Installer `rustup` sur Linux ou macOS

Si vous utilisez Linux ou macOS, ouvrez un terminal et écrivez la commande suivante :

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Cette commande télécharge un script et lance l'installation de l'outil `rustup`, qui va installer la dernière version stable de Rust. Il est possible que l'on vous demande votre mot de passe. Si l'installation se déroule bien, vous devriez voir la ligne suivante s'afficher :

```
Rust is installed now. Great!
```

Vous aurez aussi besoin d'un *linker*, qui est un programme que Rust utilise pour regrouper ses multiples résultats de compilation dans un unique fichier. Il est probable que vous en ayez déjà un d'installé, mais si vous avez des erreurs à propos du *linker*, cela veut dire vous devrez installer un compilateur de langage C, qui inclura généralement un *linker*. Un compilateur est parfois utile car certains paquets Rust communs nécessitent du code C et auront besoin d'un compilateur C.

Sur macOS, vous pouvez obtenir un compilateur C en lançant la commande :

```
$ xcode-select --install
```

Les utilisateurs de Linux doivent généralement installer GCC ou Clang, en fonction de la documentation de leur distribution. Par exemple, si vous utilisez Ubuntu, vous pouvez installer le paquet `build-essential`.

Installer `rustup` sous Windows

Sous Windows, il faut aller sur https://www.rust-lang.org/tools/install et suivre les instructions pour installer Rust. À un moment donné durant l'installation, vous aurez un message vous expliquant qu'il va vous falloir l'outil de compilation C++ pour Visual Studio 2013 ou plus récent. La méthode la plus facile pour obtenir les outils de compilation est d'installer Build Tools pour Visual Studio 2019. Lorsque vous aurez à sélectionner les composants à installer, assurez-vous que les “Outils de compilation C++” sont bien sélectionnés, et que le SDK Windows 10 et les paquets de langage Anglais sont bien inclus.

La suite de ce livre utilisera des commandes qui fonctionnent à la fois dans *cmd.exe* et PowerShell. S'il y a des différences particulières, nous vous expliquerons lesquelles utiliser.

Mettre à jour et désinstaller

Après avoir installé Rust avec `rustup`, la mise à jour vers la dernière version est facile. Dans votre terminal, lancez le script de mise à jour suivant :

```
$ rustup update
```

Pour désinstaller Rust et `rustup`, exécutez le script de désinstallation suivant dans votre terminal :

```
$ rustup self uninstall
```

Dépannage

Pour vérifier si Rust est correctement installé, ouvrez un terminal et entrez cette ligne :

```
$ rustc --version
```

Vous devriez voir le numéro de version, le *hash* de *commit*, et la date de *commit* de la dernière version stable qui a été publiée, au format suivant :

```
rustc x.y.z (abcabcabc yyyy-mm-dd)
```

Si vous voyez cette information, c'est que vous avez installé Rust avec succès ! Si vous ne voyez pas cette information et que vous êtes sous Windows, vérifiez que Rust est présent dans votre variable d'environnement système `%PATH%`. Si tout est correct et que Rust ne fonctionne toujours pas, il y a quelques endroits où vous pourrez trouver de l'aide. Le plus accessible est le canal #beginners sur le Discord officiel de Rust. Là-bas, vous pouvez dialoguer en ligne avec d'autres *Rustacés* (un surnom ridicule que nous nous donnons entre nous) qui pourront vous aider. D'autres bonnes sources de données sont le forum d'utilisateurs et Stack Overflow.

Documentation en local

L'installation de Rust embarque aussi une copie de la documentation en local pour que vous puissiez la lire hors ligne. Lancez `rustup doc` afin d'ouvrir la documentation locale dans votre navigateur.

À chaque fois que vous n'êtes pas sûr de ce que fait un type ou une fonction fournie par la bibliothèque standard ou que vous ne savez pas comment l'utiliser, utilisez cette documentation de l'interface de programmation applicative *(API)* pour le savoir !

Hello, World!

Maintenant que vous avez installé Rust, écrivons notre premier programme Rust. Lorsqu'on apprend un nouveau langage, il est de tradition d'écrire un petit programme qui écrit le texte "Hello, world!" à l'écran, donc c'est ce que nous allons faire !

> Note : ce livre part du principe que vous êtes familier avec la ligne de commande. Rust n'impose pas d'exigences sur votre éditeur, vos outils ou l'endroit où vous mettez votre code, donc si vous préférez utiliser un environnement de développement intégré (IDE) au lieu de la ligne de commande, vous êtes libre d'utiliser votre IDE favori. De nombreux IDE prennent en charge Rust à des degrés divers ; consultez la documentation de l'IDE pour plus d'informations. Récemment, l'équipe Rust s'est attelée à améliorer l'intégration dans les IDE et des progrès ont rapidement été faits dans ce domaine !

Créer un dossier projet

Nous allons commencer par créer un dossier pour y ranger le code Rust. Là où vous mettez votre code n'est pas important pour Rust, mais pour les exercices et projets de ce livre, nous vous suggérons de créer un dossier *projects* dans votre dossier utilisateur et de ranger tous vos projets là-dedans.

Ouvrez un terminal et écrivez les commandes suivantes pour créer un dossier *projects* et un dossier pour le projet “Hello, world!” à l'intérieur de ce dossier *projects*.

Sous Linux, macOS et PowerShell sous Windows, écrivez ceci :

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

Avec CMD sous Windows, écrivez ceci :

```
> mkdir "%USERPROFILE%\projects"
> cd /d "%USERPROFILE%\projects"
> mkdir hello_world
> cd hello_world
```

Écrire et exécuter un programme Rust

Ensuite, créez un nouveau fichier source et appelez-le *main.rs*. Les fichiers Rust se terminent toujours par l'extension *.rs*. Si vous utilisez plusieurs mots dans votre nom de fichier, utilisez un tiret bas (_) pour les séparer. Par exemple, vous devriez utiliser *hello_world.rs* au lieu de *helloworld.rs*.

Maintenant, ouvrez le fichier *main.rs* que vous venez de créer et entrez le code de l'encart 1-1.

Fichier : main.rs

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

Encart 1-1 : Un programme qui affiche `Hello, world!`

Enregistrez le fichier et retournez dans votre terminal. Sur Linux ou macOS, écrivez les commandes suivantes pour compiler et exécuter le fichier :

```
$ rustc main.rs
$ ./main
Hello, world!
```

Sur Windows, écrivez la commande `.\main.exe` à la place de `.\main` :

```
> rustc main.rs
> .\main.exe
Hello, world!
```

Peu importe votre système d'exploitation, la chaîne de caractères `Hello, world!` devrait s'écrire dans votre terminal. Si cela ne s'affiche pas, référez-vous à la partie "Dépannage" du chapitre d'installation pour vous aider.

Si `Hello, world!` s'affiche, félicitations ! Vous avez officiellement écrit un programme Rust. Cela fait de vous un développeur Rust — bienvenue !

Structure d'un programme Rust

Regardons en détail ce qui s'est passé dans votre programme “Hello, world!”. Voici le premier morceau du puzzle :

```
fn main() {

}
```

Ces lignes définissent une fonction dans Rust. La fonction `main` est spéciale : c'est toujours le premier code qui est exécuté dans tous les programmes en Rust. La première ligne déclare une fonction qui s'appelle `main`, qui n'a pas de paramètre et qui ne retourne aucune valeur. S'il y avait des paramètres, ils seraient placés entre les parenthèses `()`.

À noter en outre que le corps de la fonction est placé entre des accolades `{}`. Rust en a besoin autour du corps de chaque fonction. C'est une bonne pratique d'insérer l'accolade ouvrante sur la même ligne que la déclaration de la fonction, en ajoutant une espace entre les deux.

Si vous souhaitez formater le code de vos projets Rust de manière standardisé, vous pouvez utiliser un outil de formatage automatique tel que `rustfmt`. L'équipe de Rust a intégré cet outil dans la distribution standard de Rust, comme pour `rustc` par exemple, donc il est probablement déjà installé sur votre ordinateur ! Consultez la documentation en ligne pour en savoir plus.

À l'intérieur de la fonction `main`, nous avons le code suivant :

```
    println!("Hello, world!");
```

Cette ligne fait tout le travail dans ce petit programme : il écrit le texte à l'écran. Il y a quatre détails importants à noter ici.

Premièrement, le style de Rust est d'indenter avec quatre espaces, et non pas avec une tabulation.

Deuxièmement, `println!` fait appel à une macro Rust. S'il appelait une fonction à la place, cela serait écrit `println` (sans le `!`). Nous aborderons les macros Rust plus en détail dans le chapitre 19. Pour l'instant, vous avez juste à savoir qu'utiliser un `!` signifie que vous utilisez une macro plutôt qu'une fonction classique. Les macros ne suivent pas toujours les mêmes règles que les fonctions.

Troisièmement, vous voyez la chaîne de caractères `"Hello, world!"`. Nous envoyons cette chaîne en argument à `println!` et cette chaîne est affichée à l'écran.

Quatrièmement, nous terminons la ligne avec un point-virgule (`;`), qui indique que cette expression est terminée et que la suivante est prête à commencer. La plupart des lignes de Rust se terminent avec un point-virgule.

La compilation et l'exécution sont des étapes séparées

Vous venez de lancer un nouveau programme fraîchement créé, donc penchons-nous sur chaque étape du processus.

Avant de lancer un programme Rust, vous devez le compiler en utilisant le compilateur Rust en entrant la commande `rustc` et en lui passant le nom de votre fichier source, comme ceci :

```
$ rustc main.rs
```

Si vous avez de l'expérience en C ou en C++, vous observerez des similarités avec `gcc` ou `clang`. Après avoir compilé avec succès, Rust produit un binaire exécutable.

Avec Linux, macOS et PowerShell sous Windows, vous pouvez voir l'exécutable en utilisant la commande `ls` dans votre terminal. Avec Linux et macOS, vous devriez voir deux fichiers. Avec PowerShell sous Windows, vous devriez voir les trois mêmes fichiers que vous verriez en utilisant CMD.

```
$ ls
main  main.rs
```

Avec CMD sous Windows, vous devez saisir la commande suivante :

```
> dir /B %= l'option /B demande à n'afficher que les noms de fichiers =%
main.exe
main.pdb
main.rs
```

Ceci affiche le fichier de code source avec l'extension *.rs*, le fichier exécutable (*main.exe* sous Windows, mais *main* sur toutes les autres plateformes) et, quand on utilise Windows, un fichier qui contient des informations de débogage avec l'extension *.pdb*. Dans ce dossier, vous pouvez exécuter le fichier *main* ou *main.exe* comme ceci :

```
$ ./main # ou .\main.exe sous Windows
```

Si *main.rs* était votre programme “Hello, world!”, cette ligne devrait afficher `Hello, world!` dans votre terminal.

Si vous connaissez un langage dynamique, comme Ruby, Python, ou JavaScript, vous n'avez peut-être pas l'habitude de compiler puis lancer votre programme dans des étapes séparées. Rust est un langage à *compilation anticipée*, ce qui veut dire que vous pouvez compiler le programme et le donner à quelqu'un d'autre, et il peut l'exécuter sans avoir Rust

d'installé. Si vous donnez à quelqu'un un fichier *.rb*, *.py* ou *.js*, il a besoin d'avoir respectivement un interpréteur Ruby, Python, ou Javascript d'installé. Cependant, avec ces langages, vous n'avez besoin que d'une seule commande pour compiler et exécuter votre programme. Dans la conception d'un langage, tout est une question de compromis.

Compiler avec `rustc` peut suffire pour de petits programmes, mais au fur et à mesure que votre programme grandit, vous allez avoir besoin de régler plus d'options et faciliter le partage de votre code. À la page suivante, nous allons découvrir l'outil Cargo, qui va vous aider à écrire des programmes Rust à l'épreuve de la réalité.

Hello, Cargo!

Cargo est le système de compilation et de gestion de paquets de Rust. La plupart des Rustacés utilisent cet outil pour gérer les projets Rust, car Cargo s'occupe de nombreuses tâches pour vous, comme compiler votre code, télécharger les bibliothèques dont votre code dépend, et compiler ces bibliothèques. (On appelle *dépendance* une bibliothèque nécessaire pour votre code.)

Des programmes Rust très simples, comme le petit que nous avons écrit précédemment, n'ont pas de dépendance. Donc si nous avions compilé le projet “Hello, world!” avec Cargo, cela n'aurait fait appel qu'à la fonctionnalité de Cargo qui s'occupe de la compilation de votre code. Quand vous écrirez des programmes Rust plus complexes, vous ajouterez des dépendances, et si vous créez un projet en utilisant Cargo, l'ajout des dépendances sera plus facile à faire.

Comme la large majorité des projets Rust utilisent Cargo, la suite de ce livre va supposer que vous utilisez aussi Cargo. Cargo s'installe avec Rust si vous avez utilisé l'installateur officiel présenté dans la section “Installation”. Si vous avez installé Rust autrement, vérifiez que Cargo est installé en utilisant la commande suivante dans votre terminal :

```
$ cargo --version
```

Si vous voyez un numéro de version, c'est qu'il est installé ! Si vous voyez une erreur comme `Commande non trouvée` (ou `command not found`), alors consultez la documentation de votre méthode d'installation pour savoir comment installer séparément Cargo.

Créer un projet avec Cargo

Créons un nouveau projet en utilisant Cargo et analysons les différences avec notre projet initial “Hello, world!”. Retournez dans votre dossier *projects* (ou là où vous avez décidé d'enregistrer votre code). Ensuite, sur n'importe quel système d'exploitation, lancez les commandes suivantes :

```
$ cargo new hello_cargo
$ cd hello_cargo
```

La première commande a crée un nouveau dossier appelé *hello_cargo*. Nous avons appelé notre projet *hello_cargo*, et Cargo crée ses fichiers dans un dossier avec le même nom.

Rendez-vous dans le dossier *hello_cargo* et afficher la liste des fichiers. Vous constaterez que Cargo a généré deux fichiers et un dossier pour nous : un fichier *Cargo.toml* et un dossier *src*

avec un fichier *main.rs* à l'intérieur.

Il a aussi créé un nouveau dépôt Git ainsi qu'un fichier *.gitignore*. Les fichiers de Git ne seront pas générés si vous lancez `cargo new` au sein d'un dépôt Git ; vous pouvez désactiver ce comportement temporairement en utilisant `cargo new --vcs=git`.

> Note : Git est un système de gestion de versions très répandu. Vous pouvez changer `cargo new` pour utiliser un autre système de gestion de versions ou ne pas en utiliser du tout en écrivant le drapeau `--vcs`. Lancez `cargo new --help` pour en savoir plus sur les options disponibles.

Ouvrez *Cargo.toml* dans votre éditeur de texte favori. Son contenu devrait être similaire au code dans l'encart 1-2.

Fichier : Cargo.toml

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

[dependencies]
```

Encart 1-2 : Contenu de *Cargo.toml* généré par `cargo new`

Ce fichier est au format *TOML* (*Tom's Obvious, Minimal Language*), qui est le format de configuration de Cargo.

La première ligne, `[package]`, est un en-tête de section qui indique que les instructions suivantes configurent un paquet. Au fur et à mesure que nous ajouterons plus de détails à ce fichier, nous ajouterons des sections supplémentaires.

Les trois lignes suivantes définissent les informations de configuration dont Cargo a besoin pour compiler votre programme : le nom, la version, et l'édition de Rust à utiliser. Nous aborderons la clé `edition` dans l'Annexe E.

La dernière ligne, `[dependencies]`, est le début d'une section qui vous permet de lister les dépendances de votre projet. Dans Rust, les paquets de code sont désignés sous le nom de *crates*. Nous n'allons pas utiliser de *crate* pour ce projet, mais nous le ferons pour le premier projet au chapitre 2 ; nous utiliserons alors cette section à ce moment-là.

Maintenant, ouvrez *src/main.rs* et jetez-y un coup d'œil :

Fichier : src/main.rs

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

Cargo a généré un programme “Hello, world!” pour vous, exactement comme celui que nous avons écrit dans l'encart 1-1 ! Pour le moment, les seules différences entre notre projet précédent et le projet que Cargo a généré sont que Cargo a placé le code dans le dossier *src*, et que nous avons un fichier de configuration *Cargo.toml* à la racine du dossier projet.

Cargo prévoit de stocker vos fichiers sources dans le dossier *src*. Le dossier parent est là uniquement pour les fichiers README, pour les informations à propos de la licence, pour les fichiers de configuration et tout ce qui n'est pas directement relié à votre code. Utiliser Cargo vous aide à structurer vos projets. Il y a un endroit pour tout, et tout est à sa place.

Si vous commencez un projet sans utiliser Cargo, comme nous l'avons fait avec le projet “Hello, world!”, vous pouvez le transformer en projet qui utilise Cargo. Déplacez le code de votre projet dans un dossier *src* et créez un fichier *Cargo.toml* adéquat.

Compiler et exécuter un projet Cargo

Maintenant, regardons ce qu'il y a de différent quand nous compilons et exécutons le programme “Hello, world!” avec Cargo ! À l'intérieur de votre dossier *hello_cargo*, compilez votre projet en utilisant la commande suivante :

```
$ cargo build
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

Cette commande crée un fichier exécutable dans *target/debug/hello_cargo* (ou *target\debug\hello_cargo.exe* sous Windows) plutôt que de le déposer dans votre dossier courant. Vous pouvez lancer l'exécutable avec cette commande :

```
$ ./target/debug/hello_cargo # ou .\target\debug\hello_cargo.exe sous Windows
Hello, world!
```

Si tout s'est bien passé, `Hello, world!` devrait s'afficher dans le terminal. Lancer `cargo build` pour la première fois devrait aussi mener Cargo à créer un nouveau fichier à la racine du dossier projet : *Cargo.lock*. Ce fichier garde une trace des versions exactes des dépendances de votre projet. Ce projet n'a pas de dépendance, donc le fichier est un peu vide. Vous n'aurez jamais besoin de changer ce fichier manuellement ; Cargo va gérer son

contenu pour vous.

Nous venons de compiler un projet avec `cargo build` avant de l'exécuter avec `./target/debug/hello_cargo`, mais nous pouvons aussi utiliser `cargo run` pour compiler le code et ensuite lancer l'exécutable dans une seule et même commande :

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

Notez que cette fois-ci, nous ne voyons pas de messages indiquant que Cargo a compilé `hello_cargo`. Cargo a détecté que les fichiers n'avaient pas changé, donc il a juste exécuté le binaire. Si vous aviez modifié votre code source, Cargo aurait recompilé le projet avant de le lancer, et vous auriez eu les messages suivants :

```
$ cargo run
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

Cargo fournit aussi une commande appelée `cargo check`. Elle vérifie rapidement votre code pour s'assurer qu'il est compilable, mais ne produit pas d'exécutable :

```
$ cargo check
   Checking hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

Dans quel cas n'aurions-nous pas besoin d'un exécutable ? Parfois, `cargo check` est bien plus rapide que `cargo build`, car il saute l'étape de création de l'exécutable. Si vous vérifiez votre travail continuellement pendant que vous écrivez votre code, utiliser `cargo check` accélèrera le processus ! C'est pourquoi de nombreux Rustacés utilisent périodiquement `cargo check` quand ils écrivent leur programme afin de s'assurer qu'il compile. Ensuite, ils lancent `cargo build` quand ils sont prêts à utiliser l'exécutable.

Récapitulons ce que nous avons appris sur Cargo :

- Nous pouvons créer un projet en utilisant `cargo new`.
- Nous pouvons compiler un projet en utilisant `cargo build`.
- Nous pouvons compiler puis exécuter un projet en une seule fois en utilisant `cargo run`.
- Nous pouvons compiler un projet sans produire de binaire afin de vérifier l'existance d'erreurs en utilisant `cargo check`.

- Au lieu d'enregistrer le résultat de la compilation dans le même dossier que votre code, Cargo l'enregistre dans le dossier *target/debug*.

Un autre avantage d'utiliser Cargo est que les commandes sont les mêmes peu importe le système d'exploitation que vous utilisez. Donc à partir de maintenant, nous n'allons plus faire d'opérations spécifiques à Linux et macOS par rapport à Windows.

Compiler pour diffuser

Quand votre projet est finalement prêt à être diffusé, vous pouvez utiliser `cargo build --release` pour le compiler en l'optimisant. Cette commande va créer un exécutable dans *target/release* au lieu de *target/debug*. Ces optimisations rendent votre code Rust plus rapide à exécuter, mais l'utiliser rallonge le temps de compilation de votre programme. C'est pourquoi il y a deux différents profils : un pour le développement, quand vous voulez recompiler rapidement et souvent, et un autre pour compiler le programme final qui sera livré à un utilisateur, qui n'aura pas besoin d'être recompilé à plusieurs reprises et qui s'exécutera aussi vite que possible. Si vous évaluez le temps d'exécution de votre code, assurez-vous de lancer `cargo build --release` et d'utiliser l'exécutable dans *target/release* pour vos bancs de test.

Cargo comme convention

Pour des projets simples, Cargo n'apporte pas grand-chose par rapport à `rustc`, mais il vous montrera son intérêt au fur et à mesure que vos programmes deviendront plus complexes. Avec des projets complexes composés de plusieurs *crates*, il est plus facile de laisser Cargo prendre en charge la coordination de la compilation.

Même si le projet `hello_cargo` est simple, il utilise maintenant une grande partie de l'outillage que vous rencontrerez dans votre carrière avec Rust. En effet, pour travailler sur n'importe quel projet Rust existant, vous n'avez qu'à saisir les commandes suivantes pour télécharger le code avec Git, vous déplacer dans le dossier projet et compiler :

```
$ git clone example.org/projet_quelconque
$ cd projet_quelconque
$ cargo build
```

Pour plus d'informations à propos de Cargo, vous pouvez consulter sa documentation.

Résumé

Vous êtes déjà bien lancé dans votre périple avec Rust ! Dans ce chapitre, vous avez appris comment :

- Installer la dernière version stable de Rust en utilisant `rustup`
- Mettre à jour Rust vers une nouvelle version
- Ouvrir la documentation installée en local
- Écrire et exécuter un programme “Hello, world!” en utilisant directement `rustc`
- Créer et exécuter un nouveau projet en utilisant les conventions de Cargo

C'est le moment idéal pour construire un programme plus ambitieux pour s'habituer à lire et écrire du code Rust. Donc, au chapitre 2, nous allons écrire un programme de *jeu de devinettes*. Si vous préférez commencer par apprendre comment les principes de programmation de base fonctionnent avec Rust, rendez-vous au chapitre 3, puis revenez au chapitre 2.

Programmer le jeu du plus ou du moins

Entrons dans le vif du sujet en travaillant ensemble sur un projet concret ! Ce chapitre présente quelques concepts couramment utilisés en Rust en vous montrant comment les utiliser dans un véritable programme. Nous aborderons notamment les instructions `let` et `match`, les méthodes et fonctions associées, l'utilisation des *crates*, et bien plus encore ! Dans les chapitres suivants, nous approfondirons ces notions. Dans ce chapitre, vous n'allez exercer que les principes de base.

Nous allons coder un programme fréquemment réalisé par les débutants en programmation : *le jeu du plus ou du moins*. Le principe de ce jeu est le suivant : le programme va tirer au sort un nombre entre 1 et 100. Il invitera ensuite le joueur à saisir un nombre qu'il pense deviner. Après la saisie, le programme indiquera si le nombre saisi par le joueur est trop grand ou trop petit. Si le nombre saisi est le bon, le jeu affichera un message de félicitations et se fermera.

Mise en place d'un nouveau projet

Pour créer un nouveau projet, rendez-vous dans le dossier *projects* que vous avez créé au chapitre 1 et utilisez Cargo pour créer votre projet, comme ceci :

```
$ cargo new jeu_du_plus_ou_du_moins
$ cd jeu_du_plus_ou_du_moins
```

La première commande, `cargo new`, prend comme premier argument le nom de notre projet (`jeu_du_plus_ou_du_moins`). La seconde commande nous déplace dans le dossier de notre nouveau projet créé par Cargo.

Regardons le fichier *Cargo.toml* qui a été généré :

Fichier : Cargo.toml

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

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

[dependencies]
```

Comme vous l'avez expérimenté dans le chapitre 1, `cargo new` génère un programme *“Hello, world!”* pour vous. Ouvrez le fichier *src/main.rs* :

Fichier : src/main.rs

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

Maintenant, lançons la compilation de ce programme “Hello, world!” et son exécution en une seule commande avec `cargo run` :

```
$ cargo run
   Compiling jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50s
     Running `target/debug/jeu_du_plus_ou_du_moins`
Hello, world!
```

Cette commande `run` est très pratique lorsqu'on souhaite itérer rapidement sur un projet, comme c'est le cas ici, pour tester rapidement chaque modification avant de passer à la suivante.

Ouvrez à nouveau le fichier *src/main.rs*. C'est dans ce fichier que nous écrirons la totalité de notre code.

Traitement d'un nombre saisi

La première partie du programme consiste à demander au joueur de saisir du texte, à traiter cette saisie, et à vérifier que la saisie correspond au format attendu. Commençons par permettre au joueur de saisir son nombre. Entrez le code de l'encart 2-1 dans le fichier *src/main.rs*.

Fichier : src/main.rs

```
use std::io;

fn main() {
    println!("Devinez le nombre !");

    println!("Veuillez entrer un nombre.");

    let mut supposition = String::new();

    io::stdin()
        .read_line(&mut supposition)
        .expect("Échec de la lecture de l'entrée utilisateur");

    println!("Votre nombre : {}", supposition);
}
```

Encart 2-1 : Code permettant de récupérer une saisie utilisateur et de l'afficher

Ce code contient beaucoup d'informations, nous allons donc l'analyser petit à petit. Pour obtenir la saisie utilisateur et ensuite l'afficher, nous avons besoin d'importer la bibliothèque d'entrée/sortie `io` (initiales de *input/output*) afin de pouvoir l'utiliser. La bibliothèque `io` provient de la bibliothèque standard, connue sous le nom de `std` :

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

Par défaut, Rust importe dans la portée de tous les programmes quelques fonctionnalités définies dans la bibliothèque standard. Cela s'appelle *l'étape préliminaire (the prelude)*, et vous pouvez en savoir plus dans sa documentation de la bibliothèque standard.

Si vous voulez utiliser un type qui ne s'y trouve pas, vous devrez l'importer explicitement avec l'instruction `use` . L'utilisation de la bibliothèque `std::io` vous apporte de nombreuses fonctionnalités utiles, comme ici la possibilité de récupérer une saisie utilisateur.

Comme vous l'avez vu au chapitre 1, la fonction `main` est le point d'entrée du programme :

```
fn main() {
```

Le mot clé `fn` déclare une nouvelle fonction, les parenthèses `()` indiquent que cette fonction n'accepte aucun paramètre, et l'accolade ouvrante `{` marque le début du corps de la fonction.

Comme vous l'avez également appris au chapitre 1, `println!` est une macro qui affiche une chaîne de caractères à l'écran :

```
    println!("Devinez le nombre !");

    println!("Veuillez entrer un nombre.");
```

Ce code affiche du texte qui indique le titre de notre jeu, et un autre qui demande au joueur d'entrer un nombre.

Enregistrer des données dans des variables

Ensuite, on crée une *variable* pour stocker la saisie de l'utilisateur, comme ceci :

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

Le programme commence à devenir intéressant ! Il se passe beaucoup de choses dans cette petite ligne. Nous utilisons l'instruction `let` pour créer la variable. Voici un autre exemple :

```
let pommes = 5;
```

Cette ligne permet de créer une nouvelle variable nommée `pommmes` et à lui assigner la valeur 5. Par défaut en Rust, les variables sont immuables. Nous aborderons plus en détail cette notion dans la section "Variables et Mutabilité" au chapitre 3. Pour rendre une variable mutable *(c'est-à-dire modifiable)*, nous ajoutons `mut` devant le nom de la variable :

```
let pommes = 5; // immuable
let mut bananes = 5; // mutable, modifiable
```

> Remarque : La syntaxe `//` permet de commencer un commentaire qui s'étend jusqu'à la fin de la ligne. Rust ignore tout ce qu'il y a dans un commentaire. Nous verrons plus en détail les commentaires dans le chapitre 3.

Lorsque vous revenez sur le jeu du plus ou du moins, vous comprenez donc maintenant que la ligne `let mut supposition` permet de créer une variable mutable nommée `supposition`. Le signe égal (`=`) indique à Rust que nous voulons désormais lier quelquechose à la variable. A la droite du signe égal, nous avons la valeur liée à `supposition`, qui est ici le résultat de l'utilisation de `String::new`, qui est une fonction qui retourne une nouvelle instance de `String`. `String` est un type de chaîne de caractères fourni par la bibliothèque standard, qui est une portion de texte encodée en UTF-8 et dont la longueur peut augmenter.

La syntaxe `::` dans `String::new()` indique que `new` est une fonction associée au type `String`. Une *fonction associée* est une fonction qui est implémentée sur un type, ici `String`. Cette fonction `new` crée une nouvelle chaîne de caractères vide, une nouvelle `String`. Vous trouverez fréquemment une fonction `new` sur d'autres types, car c'est un nom souvent donné à une fonction qui crée une nouvelle valeur ou instance d'un type.

En définitif, la ligne `let mut supposition = String::new();` crée une nouvelle variable mutable qui contient une nouvelle chaîne de caractères vide, une instance de `String`. Ouf !

Recueillir la saisie utilisateur

Rappelez-vous que nous avons importé les fonctionnalités d'entrée/sortie de la bibliothèque standard avec `use std::io;` à la première ligne de notre programme. Nous allons maintenant appeler la fonction `stdin` du module `io`, qui va nous permettre de traiter la saisie utilisateur :

```
io::stdin()
    .read_line(&mut supposition)
```

Si nous n'avions pas importé la bibliothèque `io` avec `use std::io` au début du programme, on aurait toujours pu utiliser la fonction en écrivant l'appel à la fonction de cette manière : `std::io::stdin`. La fonction `stdin` retourne une instance de `std::io::Stdin`, qui est un type qui représente une référence abstraite *(handle)* vers l'entrée standard du terminal dans lequel vous avez lancé le programme.

Ensuite, la ligne `.read_line(&mut supposition)` appelle la méthode `read_line` sur l'entrée standard afin d'obtenir la saisie utilisateur. Nous passons aussi `&mut supposition` en argument de `read_line` pour lui indiquer dans quelle chaîne de caractère il faut stocker la saisie utilisateur. Le but final de `read_line` est de récupérer tout ce que l'utilisateur écrit dans l'entrée standard et de l'ajouter à la fin d'une chaîne de caractères (sans écraser son contenu) ; c'est pourquoi nous passons cette chaîne de caractères en argument. Cet argument doit être mutable pour que `read_line` puisse en modifier le contenu.

Le `&` indique que cet argument est une *référence*, ce qui permet de laisser plusieurs morceaux de votre code accéder à une même donnée sans avoir besoin de copier ces données dans la mémoire plusieurs fois. Les références sont une fonctionnalité complexe, et un des avantages majeurs de Rust est qu'il rend sûr et simple l'utilisation des références. Il n'est pas nécessaire de trop s'apesantir sur les références pour terminer ce programme. Pour l'instant, tout ce que vous devez savoir est que comme les variables, les références sont immuables par défaut. D'où la nécessité d'écrire `&mut supposition` au lieu de

`&supposition` pour la rendre mutable. (Le chapitre 4 expliquera plus en détail les références.)

Gérer les erreurs potentielles avec le type `Result`

Nous avons encore du travail sur cette ligne de code. Même si nous allons rajouter une troisième ligne de code, elle ne fait partie que d'une seule ligne de code. Cette nouvelle partie rajoute cette méthode :

```
        .expect("Échec de la lecture de l'entrée utilisateur");
```

Nous aurions pu écrire ce code de cette manière :

```
io::stdin().read_line(&mut supposition).expect("Échec de la lecture de l'entrée utilisateur");
```

Cependant, une longue ligne de code n'est pas toujours facile à lire, c'est donc une bonne pratique de la diviser. Il est parfois utile d'ajouter une nouvelle ligne et des espaces afin de désagréger les longues lignes lorsque vous appelerez une méthode, comme ici avec la syntaxe `.nom_de_la_methode()`. Maintenant, voyons à quoi sert cette ligne.

Comme expliqué précédemment, `read_line` stocke dans la variable qu'on lui passe en argument tout ce que l'utilisateur a saisi, mais cette fonction retourne aussi une valeur – dans notre cas, de type `io::Result`. Il existe plusieurs types nommés `Result` dans la bibliothèque standard de Rust : un type générique `Result` ainsi que des déclinaisons spécifiques à des sous-modules, comme `io::Result`. Les types `Result` sont des *énumérations*, aussi appelées *enums*, qui peuvent avoir un certain nombre de valeurs prédéfinies que l'on appelle *variantes*. Les énumérations sont souvent utilisées avec `match`, une structure conditionelle qui facilite l'exécution d'un code différent en fonction de la variante dans l'énumération au moment de son évaluation.

Le chapitre 6 explorera les énumérations plus en détail. La raison d'être du type `Result` est de coder des informations pour la gestion des erreurs.

Les variantes de `Result` sont `Ok` et `Err`. La variante `Ok` signifie que l'opération a fonctionné, et à l'intérieur de `Ok` se trouve la valeur générée avec succès. La variante `Err` signifie que l'opération a échoué, et `Err` contient les informations décrivant comment ou pourquoi l'opération a échoué.

Les valeurs du type `Result`, comme pour tous les types, ont des méthodes qui leur sont associées. Par exemple, une instance de `io::Result` a une méthode `expect` que vous

pouvez utiliser. Si cette instance de `io::Result` a pour valeur la variante `Err`, l'appel à `expect` fera planter le programme et affichera le message que vous avez passé en argument de `expect`. Si l'appel à `read_line` retourne une variante `Err`, ce sera probablement dû à une erreur du système d'exploitation. Si en revanche `read_line` a pour valeur la variante `Ok`, `expect` récupèrera le contenu du `Ok`, qui est le résultat de l'opération, et vous le retournera afin que vous puissiez l'utiliser. Dans notre exemple, ce résultat est le nombre d'octets de la saisie utilisateur.

Si on n'appelle pas `expect`, le programme compilera, mais avec un avertissement :

```
$ cargo build
   Compiling jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
warning: unused `Result` that must be used
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut supposition);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(unused_must_use)]` on by default
   = note: this `Result` may be an `Err` variant, which should be handled

warning: `jeu_du_plus_ou_du_moins` (bin "jeu_du_plus_ou_du_moins") generated 1
warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

Rust nous prévient que l'on ne fait rien du `Result` que nous fournit `read_line`, et que par conséquent notre programme ne gère pas une erreur potentielle.

La meilleure façon de masquer cet avertissement est de réellement écrire le code permettant de gérer l'erreur, mais dans notre cas on a seulement besoin de faire planter le programme si un problème survient, on utilise donc `expect`. Nous verrons dans le chapitre 9 comment gérer correctement les erreurs.

Afficher des valeurs grâce aux espaces réservés de `println!`

Mis à part l'accolade fermante, il ne nous reste plus qu'une seule ligne à étudier dans le code que nous avons pour l'instant :

```
    println!("Votre nombre : {}", supposition);
```

Cette ligne affiche la chaîne de caractères qui contient maintenant ce que l'utilisateur a saisi. La paire d'accolades `{}` représente un espace réservé : imaginez qu'il s'agit de pinces de crabes qui gardent la place d'une valeur. Vous pouvez afficher plusieurs valeurs en utilisant

des accolades : la première paire d'accolades affichera la première valeur listée après la chaîne de formatage, la deuxième paire d'accolades affichera la deuxième valeur, et ainsi de suite. Pour afficher plusieurs valeurs en appelant `println!` une seule fois, on ferait comme ceci :

```
let x = 5;
let y = 10;

println!("x = {} et y = {}", x, y);
```

Ce code afficherait `x = 5 et y = 10`.

Test de la première partie

Pour tester notre début de programme, lançons-le à l'aide de la commande `cargo run` :

```
$ cargo run
   Compiling jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
    Finished dev [unoptimized + debuginfo] target(s) in 6.44s
     Running `target/debug/jeu_du_plus_ou_du_moins`
Devinez le nombre !
Veuillez entrer un nombre.
6
Votre nombre : 6
```

À ce stade, la première partie de notre programme est terminée : nous avons récupéré la saisie du clavier et nous l'affichons à l'écran.

Générer le nombre secret

Maintenant, il nous faut générer un nombre secret que notre joueur va devoir deviner. Ce nombre devra être différent à chaque fois pour qu'on puisse s'amuser à y jouer plusieurs fois. Nous allons tirer au sort un nombre compris entre 1 et 100 pour que le jeu ne soit pas trop difficile. Rust n'embarque pas pour l'instant de fonctionnalité de génération de nombres aléatoires dans sa bibliothèque standard. Cependant, l'équipe de Rust propose une *crate* `rand` qui offre la possibilité de le faire.

Étendre les fonctionnalités de Rust avec une *crate*

Souvenez-vous, une *crate* est un ensemble de fichiers de code source Rust. Le projet sur lequel nous travaillons est une *crate* binaire, qui est un programme exécutable. La *crate* `rand` est une *crate de bibliothèque*, qui contient du code qui peut être utilisé dans d'autres programmes, et qui ne peut pas être exécuté tout seul.

La coordination des *crates* externes est un domaine dans lequel Cargo excelle. Avant d'écrire le code qui utilisera `rand`, il nous faut éditer le fichier *Cargo.toml* pour y spécifier `rand` en tant que dépendance. Ouvrez donc maintenant ce fichier et ajoutez la ligne suivante à la fin, en dessous de l'en-tête de section `[dependencies]` que Cargo a créé pour vous. Assurez-vous de spécifier `rand` exactement comme dans le bout de code suivant, avec ce numéro de version, ou sinon les exemples de code de ce tutoriel pourraient ne pas fonctionner.

Fichier : Cargo.toml

```
rand = "0.8.3"
```

Dans le fichier *Cargo.toml*, tout ce qui suit une en-tête fait partie de cette section, et ce jusqu'à ce qu'une autre section débute. Dans `[dependencies]`, vous indiquez à Cargo de quelles *crates* externes votre projet dépend, et de quelle version de ces *crates* vous avez besoin. Dans notre cas, on ajoute comme dépendance la crate `rand` avec la version sémantique `0.8.3`. Cargo arrive à interpréter le versionnage sémantique (aussi appelé *SemVer*), qui est une convention d'écriture de numéros de version. En réalité, `0.8.3` est une abréviation pour `^0.8.3`, ce qui signifie “toute version ultérieure ou égale à `0.8.3` mais strictement antérieure à `0.9.0`”. Cargo considère que ces versions ont des API publiques compatibles avec la version `0.8.3`, et cette indication garantit que vous obtiendrez la dernière version de correction qui compilera encore avec le code de ce chapitre. Il n'est pas garanti que les versions `0.9.0` et ultérieures aient la même API que celle utilisée dans les exemples suivants.

Maintenant, sans apporter le moindre changement au code, lançons une compilation du projet, comme dans l'encart 2-2 :

```
$ cargo build
    Updating crates.io index
  Downloaded rand v0.8.3
  Downloaded libc v0.2.86
  Downloaded getrandom v0.2.2
  Downloaded cfg-if v1.0.0
  Downloaded ppv-lite86 v0.2.10
  Downloaded rand_chacha v0.3.0
  Downloaded rand_core v0.6.2
   Compiling rand_core v0.6.2
   Compiling libc v0.2.86
   Compiling getrandom v0.2.2
   Compiling cfg-if v1.0.0
   Compiling ppv-lite86 v0.2.10
   Compiling rand_chacha v0.3.0
   Compiling rand v0.8.3
   Compiling jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53s
```

Encart 2-2 : Résultat du lancement de `cargo build` après avoir ajouté la *crate* `rand` comme dépendance

Il est possible que vous ne voyiez pas exactement les mêmes numéros de version, (mais ils seront compatibles avec votre code, grâce au *versionnage sémantique* !), différentes lignes (en fonction de votre système d'exploitation), et les lignes ne seront pas forcément affichées dans le même ordre.

Lorsque nous ajoutons une dépendance externe, Cargo récupère les dernières versions de tout ce dont cette dépendance a besoin depuis le *registre*, qui est une copie des données de Crates.io. Crates.io est là où les développeurs de l'écosystème Rust publient leurs projets open source afin de les rendre disponibles aux autres.

Une fois le registre mis à jour, Cargo lit la section `[dependencies]` et se charge de télécharger les *crates* qui y sont listés que vous n'avez pas encore téléchargé. Dans notre cas, bien que nous n'ayons spécifié qu'une seule dépendance, `rand`, Cargo a aussi téléchargé d'autres *crates* dont dépend `rand` pour fonctionner. Une fois le téléchargement terminé des *crates*, Rust les compile, puis compile notre projet avec les dépendances disponibles.

Si vous relancez tout de suite `cargo build` sans changer quoi que ce soit, vous n'obtiendrez rien d'autre que la ligne `Finished`. Cargo sait qu'il a déjà téléchargé et compilé les dépendances, et que vous n'avez rien changé dans votre fichier *Cargo.toml*. Cargo sait aussi que vous n'avez rien changé dans votre code, donc il ne le recompile pas non plus. Étant donné qu'il n'a rien à faire, Cargo se termine tout simplement.

Si vous ouvrez le fichier *src/main.rs*, faites un changement très simple, enregistrez le fichier, et relancez la compilation, vous verrez s'afficher uniquement deux lignes :

```
$ cargo build
   Compiling jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

Ces lignes nous informent que Cargo a recompilé uniquement à cause de notre petit changement dans le fichier *src/main.rs*. Les dépendances n'ayant pas changé, Cargo sait qu'il peut simplement réutiliser ce qu'il a déjà téléchargé et compilé précédemment.

Assurer la reproductibilité des compilations avec le fichier *Cargo.lock*

Cargo embarque une fonctionnalité qui garantie que vous pouvez recompiler le même artéfact à chaque fois que vous ou quelqu'un d'autre compile votre code : Cargo va utiliser uniquement les versions de dépendances que vous avez utilisées jusqu'à ce que vous indiquiez le contraire. Par exemple, immaginons que la semaine prochaine, la version 0.8.4 de la *crate* `rand` est publiée, et qu'elle apporte une correction importante, mais aussi qu'elle produit une régression qui va casser votre code. Pour éviter cela, Rust crée le fichier *Cargo.lock* la première fois que vous utilisez `cargo build`, donc nous l'avons désormais dans le dossier *jeu_du_plus_ou_du_moins*.

Quand vous compilez un projet pour la première fois, Cargo détermine toutes les versions de dépendances qui correspondent à vos critères et les écrit dans le fichier *Cargo.lock*. Quand vous recompilerez votre projet plus tard, Cargo verra que le fichier *Cargo.lock* existe et utilisera les versions précisées à l'intérieur au lieu de recommencer à déterminer toutes les versions demandées. Ceci vous permet d'avoir automatiquement des compilations reproductibles. En d'autres termes, votre projet va rester sur la version `0.8.3` jusqu'à ce que vous le mettiez à jour explicitement, grâce au fichier *Cargo.lock*.

Mettre à jour une *crate* vers sa nouvelle version

Lorsque vous souhaitez réellement mettre à jour une *crate*, Cargo vous fournit la commande `update`, qui va ignorer le fichier *Cargo.lock* et va rechercher toutes les versions qui correspondent à vos critères dans *Cargo.toml*. Cargo va ensuite écrire ces versions dans le fichier *Cargo.lock*. Sinon par défaut, Cargo va rechercher uniquement les versions plus grandes que `0.8.3` et inférieures à `0.9.0`. Si la *crate* `rand` a été publiée en deux nouvelles versions `0.8.4` et `0.9.0`, alors vous verrez ceci si vous lancez `cargo update` :

```
$ cargo update
    Updating crates.io index
    Updating rand v0.8.3 -> v0.8.4
```

Cargo ignore la version `0.9.0`. À partir de ce moment, vous pouvez aussi constater un changement dans le fichier *Cargo.lock* indiquant que la version de la *crate* `rand` que vous utilisez maintenant est la `0.8.4`. Pour utiliser `rand` en version `0.9.0` ou toute autre version dans la série des `0.9.x`, il vous faut mettre à jour le fichier *Cargo.toml* comme ceci :

```
[dependencies]
rand = "0.9.0"
```

La prochaine fois que vous lancerez `cargo build`, Cargo mettra à jour son registre de *crates* disponibles et réévaluera vos exigences vis-à-vis de `rand` selon la nouvelle version que vous avez spécifiée.

Il y a encore plus à dire à propos de Cargo et de son écosystème que nous aborderons au chapitre 14, mais pour l'instant, c'est tout ce qu'il vous faut savoir. Cargo facilite la réutilisation des bibliothèques, pour que les Rustacés soient capables d'écrire des petits projets issus d'un assemblage d'un certain nombre de paquets.

Générer un nombre aléatoire

Commençons désormais à utiliser `rand` pour générer un nombre à deviner. La prochaine étape est de modifier *src/main.rs* comme dans l'encart 2-3.

Fichier : src/main.rs

```
use std::io;
use rand::Rng;

fn main() {
    println!("Devinez le nombre !");

    let nombre_secret = rand::thread_rng().gen_range(1..101);

    println!("Le nombre secret est : {}", nombre_secret);

    println!("Veuillez entrer un nombre.");

    let mut supposition = String::new();

    io::stdin()
        .read_line(&mut supposition)
        .expect("Échec de la lecture de l'entrée utilisateur");

        println!("Votre nombre : {}", supposition);
}
```

Encart 2-3 : Ajout du code pour générer un nombre aléatoire

D'abord, nous avons ajouté la ligne `use rand::Rng`. Le *trait* `Rng` définit les méthodes implémentées par les générateurs de nombres aléatoires, et ce *trait* doit être accessible à notre code pour qu'on puisse utiliser ces méthodes. Le chapitre 10 expliquera plus en détail les *traits*.

Ensuite, nous ajoutons deux lignes au milieu. A la première ligne, nous appelons la fonction `rand::thread_rng` qui nous fournit le générateur de nombres aléatoires particulier que nous allons utiliser : il est propre au fil d'exécution courant et généré par le système d'exploitation. Ensuite, nous appelons la méthode `gen_range` sur le générateur de nombres aléatoires. Cette méthode est définie par le *trait* `Rng` que nous avons importé avec l'instruction `use rand::Rng`. La méthode `gen_range` prend une expression d'intervalle en paramètre et génère un nombre aléatoire au sein de l'intervalle. Le genre d'expression d'intervalle utilisé ici est de la forme `début..fin` et inclut la borne inférieure mais exclut la borne supérieure, nous avons donc besoin de préciser `1..101` pour demander un nombre entre 1 et 100. De manière équivalente, nous pourrions également passer l'intervalle fermé `1..=100`

> Remarque : vous ne pourrez pas deviner quels *traits*, méthodes et fonctions utiliser avec une *crate*, donc chaque *crate* a une documentation qui donne des indications sur son utilisation. Une autre fonctionnalité intéressante de Cargo est que vous pouvez utiliser la commande `cargo doc --open`, qui va construire localement la documentation intégrée par toutes vos dépendances et va l'ouvrir dans votre

navigateur. Si vous vous intéressez à d'autres fonctionnalités de la *crate* `rand`, par exemple, vous pouvez lancer `cargo doc --open` et cliquer sur `rand` dans la barre latérale sur la gauche.

La seconde nouvelle ligne affiche le nombre secret. C'est pratique lors du développement pour pouvoir le tester, mais nous l'enlèverons dans la version finale. Ce n'est pas vraiment un jeu si le programme affiche la réponse dès qu'il démarre !

Essayez de lancer le programme plusieurs fois :

```
$ cargo run
   Compiling jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53s
     Running `target/debug/jeu_du_plus_ou_du_moins`
Devinez le nombre !
Le nombre secret est : 7
Veuillez entrer un nombre.
4
Votre nombre : 4

$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/jeu_du_plus_ou_du_moins`
Devinez le nombre !
Le nombre secret est : 83
Veuillez entrer un nombre.
5
Votre nombre : 5
```

Vous devriez obtenir des nombres aléatoires différents, et ils devraient être tous compris entre 1 et 100. Beau travail !

Comparer le nombre saisi au nombre secret

Maintenant que nous avons une saisie utilisateur et un nombre aléatoire, nous pouvons les comparer. Cette étape est écrite dans l'encart 2-4. Sachez toutefois que le code ne se compile pas encore, nous allons l'expliquer par la suite.

Fichier : src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    // -- partie masquée ici --

    println!("Votre nombre : {}", supposition);

    match supposition.cmp(&nombre_secret) {
        Ordering::Less => println!("C'est plus !"),
        Ordering::Greater => println!("C'est moins !"),
        Ordering::Equal => println!("Vous avez gagné !"),
    }
}
```

Encart 2-4 : Traitement des valeurs possibles saisies en comparant les deux nombres

Premièrement, nous ajoutons une autre instruction `use`, qui importe `std::cmp::Ordering` à portée de notre code depuis la bibliothèque standard. Le type `Ordering` est une autre énumération et a les variantes `Less` *(inférieur)*, `Greater` *(supérieur)* et `Equal` *(égal)*. Ce sont les trois issues possibles lorsqu'on compare deux valeurs.

Ensuite, nous ajoutons cinq nouvelles lignes à la fin qui utilisent le type `Ordering`. La méthode `cmp` compare deux valeurs et peut être appelée sur tout ce qui peut être comparé. Elle prend en paramètre une référence de ce qu'on veut comparer : ici, nous voulons comparer `supposition` et `nombre_secret`. Ensuite, cela retourne une variante de l'énumération `Ordering` que nous avons importée avec l'instruction `use`. Nous utilisons une expression `match` pour décider quoi faire ensuite en fonction de quelle variante de `Ordering` a été retournée à l'appel de `cmp` avec `supposition` et `nombre_secret`.

Une expression `match` est composée de *branches*. Une branche est constituée d'un *motif (pattern)* avec lequel elle doit correspondre et du code qui sera exécuté si la valeur donnée au `match` correspond bien au motif de cette branche. Rust prend la valeur donnée à `match` et la compare au motif de chaque branche à tour de rôle. Les motifs et la structure de contrôle `match` sont des fonctionnalités puissantes de Rust qui vous permettent de décrire une multitude de scénarios que votre code peut rencontrer et de s'assurer que vous les gérez toutes. Ces fonctionnalités seront expliquées plus en détail respectivement dans le chapitre 6 et le chapitre 18.

Voyons un exemple avec l'expression `match` que nous avons utilisé ici. Disons que l'utilisateur a saisi le nombre 50 et que le nombre secret généré aléatoirement a cette fois-ci comme valeur 38. Quand le code compare 50 à 38, la méthode `cmp` va retourner `Ordering::Greater`, car 50 est plus grand que 38. L'expression `match` obtient la valeur

`Ordering::Greater` et commence à vérifier le motif de chaque branche. Elle consulte le motif de la première branche, `Ordering::Less` et remarque que la valeur `Ordering::Greater` ne correspond pas au motif `Ordering::Less` ; elle ignore donc le code de cette branche et passe à la suivante. Le motif de la branche suivante est `Ordering::Greater`, qui correspond à `Ordering::Greater` ! Le code associé à cette branche va être exécuté et va afficher à l'écran `C'est moins !`. L'expression `match` se termine ensuite, car elle n'a pas besoin de consulter les autres branches de ce scénario.

Cependant, notre code dans l'encart 2-4 ne compile pas encore. Essayons de le faire :

```
$ cargo build
   Compiling libc v0.2.86
   Compiling getrandom v0.2.2
   Compiling cfg-if v1.0.0
   Compiling ppv-lite86 v0.2.10
   Compiling rand_core v0.6.2
   Compiling rand_chacha v0.3.0
   Compiling rand v0.8.3
   Compiling jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
error[E0308]: mismatched types
  --> src/main.rs:22:21
   |
22 |     match supposition.cmp(&nombre_secret) {
   |                           ^^^^^^^^^^^^^^^ expected struct `String`, found
integer
   |
   = note: expected reference `&String`
              found reference `&{integer}`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `jeu_du_plus_ou_du_moins` due to previous error

error[E0283]: type annotations needed for `{integer}`
   --> src/main.rs:8:44
    |
8   |     let nombre_secret = rand::thread_rng().gen_range(1..101);
    |         -------------                      ^^^^^^^^^ cannot infer type for
type `{integer}`
    |         |
    |         consider giving `nombre_secret` a type
    |
    = note: multiple `impl`s satisfying `{integer}: SampleUniform` found in the
`rand` crate:
            - impl SampleUniform for i128;
            - impl SampleUniform for i16;
            - impl SampleUniform for i32;
            - impl SampleUniform for i64;
            and 8 more
note: required by a bound in `gen_range`
   --> /Users/carolnichols/.cargo/registry/src/github.com-1ecc6299db9ec823/
rand-0.8.3/src/rng.rs:129:12
    |
129 |         T: SampleUniform,
    |            ^^^^^^^^^^^^^ required by this bound in `gen_range`
help: consider specifying the type arguments in the function call
    |
8   |     let nombre_secret = rand::thread_rng().gen_range::<T, R>(1..101);
    |                                                     ++++++++

Some errors have detailed explanations: E0283, E0308.
For more information about an error, try `rustc --explain E0283`.
```

```
error: could not compile `guessing_game` due to 2 previous errors
```

Le message d'erreur nous indique que nous sommes dans un cas de types non compatibles *(mismatched types)*. Rust a un système de types fort et statique. Cependant, il a aussi une fonctionnalité d'inférence de type. Quand nous avons écrit `let mut supposition = String::new()`, Rust a pu en déduire que `supposition` devait être une `String` et ne nous a pas demandé d'écrire le type. D'autre part, `nombre_secret` est d'un type de nombre. Quelques types de nombres de Rust peuvent avoir une valeur entre 1 et 100 : `i32`, un nombre entier encodé sur 32 bits ; `u32`, un nombre entier de 32 bits non signé (positif ou nul) ; `i64`, un nombre entier encodé sur 64 bits ; parmi tant d'autres. Rust utilise par défaut un `i32`, qui est le type de `nombre_secret`, à moins que vous précisiez quelque part une information de type qui amènerait Rust à inférer un type de nombre différent. La raison de cette erreur est que Rust ne peut pas comparer une chaîne de caractères à un nombre.

Au bout du compte, nous voulons convertir la `String` que le programme récupère de la saisie utilisateur en un nombre, pour qu'on puisse la comparer numériquement au nombre secret. Nous allons faire ceci en ajoutant cette ligne supplémentaire dans le corps de la fonction `main` :

Fichier : src/main.rs

```
    // -- partie masquée ici --

    let mut supposition = String::new();

    io::stdin()
        .read_line(&mut supposition)
        .expect("Échec de la lecture de l'entrée utilisateur");

    let supposition: u32 = supposition.trim().parse().expect("Veuillez entrer un
nombre !");

    println!("Votre nombre : {}", supposition);

    match supposition.cmp(&nombre_secret) {
        Ordering::Less => println!("C'est plus !"),
        Ordering::Greater => println!("C'est moins !"),
        Ordering::Equal => println!("Vous avez gagné !"),
    }
```

La nouvelle ligne est :

```
let supposition: u32 = supposition.trim().parse().expect("Veuillez entrer un
nombre !");
```

Nous créons une variable qui s'appelle `supposition`. Mais attendez, le programme n'a-t-il

pas déjà une variable qui s'appelle `supposition` ? C'est le cas, mais heureusement Rust nous permet de *masquer* la valeur précédente de `supposition` avec une nouvelle. Le masquage *(shadowing)* nous permet de réutiliser le nom de variable `supposition`, plutôt que de nous forcer à créer deux variables distinctes, telles que `supposition_str` et `supposition` par exemple. Nous verrons cela plus en détails au chapitre 3, mais pour le moment cette fonctionnalité est souvent utilisée dans des situations où on veut convertir une valeur d'un type à un autre.

Nous lions cette nouvelle variable à l'expression `supposition.trim().parse()`. Le `supposition` dans l'expression se réfère à la variable `supposition` initiale qui contenait la saisie utilisateur en tant que chaîne de caractères. `String` contenant la saisie utilisateur. La méthode `trim` sur une instance de `String` va enlever les espaces et autres *whitespaces* au début et à la fin, ce que nous devons faire pour comparer la chaîne au `u32`, qui ne peut être constitué que de chiffres. L'utilisateur doit appuyer sur entrée pour mettre fin à `read_line` et récupérer leur supposition, ce qui va rajouter un caractère de fin de ligne à la chaîne de caractères. Par exemple, si l'utilisateur écrit 5 et appuie sur entrée, `supposition` aura alors cette valeur : `5\n`. Le `\n` représente une fin de ligne (à noter que sur Windows, appuyer sur entrée résulte en un retour chariot suivi d'une fin de ligne, `\r\n`). La méthode `trim` enlève `\n` et `\r\n`, il ne reste donc plus que `5`.

La méthode `parse` des chaînes de caractères interprète une chaîne de caractères en une sorte de nombre. Comme cette méthode peut interpréter plusieurs types de nombres, nous devons indiquer à Rust le type exact de nombre que nous voulons en utilisant `let supposition: u32`. Le deux-points (`:`) après `supposition` indique à Rust que nous voulons préciser le type de la variable. Rust embarque quelques types de nombres ; le `u32` utilisé ici est un entier non signé sur 32 bits. C'est un bon choix par défaut pour un petit nombre positif. Vous découvrirez d'autres types de nombres dans le chapitre 3. De plus, l'annotation `u32` dans ce programme d'exemple et la comparaison avec `nombre_secret` permet à Rust d'en déduire que `nombre_secret` doit être lui aussi un `u32`. Donc maintenant, la comparaison se fera entre deux valeurs du même type !

La méthode `parse` va fonctionner uniquement sur des caractères qui peuvent être logiquement convertis en nombres et donc peut facilement mener à une erreur. Si par exemple, le texte contient `A👍%`, il ne sera pas possible de le convertir en nombre. Comme elle peut échouer, la méthode `parse` retourne un type `Result`, comme celui que la méthode `read_line` retourne (comme nous l'avons vu plus tôt dans "Gérer les erreurs potentielles avec le type `Result`"). Nous allons gérer ce `Result` de la même manière, avec à nouveau la méthode `expect`. Si `parse` retourne une variante `Err` de `Result` car elle ne peut pas créer un nombre à partir de la chaîne de caractères, l'appel à `expect` va faire planter le jeu et va afficher le message que nous lui avons passé en paramètre. Si `parse`

arrive à convertir la chaîne de caractères en nombre, alors elle retournera la variante `Ok` de `Result`, et `expect` va retourner le nombre qu'il nous faut qui est stocké dans la variante `Ok`.

Exécutons ce programme, maintenant !

```
$ cargo run
   Compiling jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/jeu_du_plus_ou_du_moins`
Devinez le nombre !
Le nombre secret est : 58
Veuillez entrer un nombre.
  76
Votre nombre : 76
C'est moins !
```

Très bien ! Même si des espaces ont été ajoutées avant la supposition, le programme a quand même compris que l'utilisateur a saisi 76. Lancez le programme plusieurs fois pour vérifier qu'il se comporte correctement avec différentes saisies : devinez le nombre correctement, saisissez un nombre qui est trop grand, et saisissez un nombre qui est trop petit.

La majeure partie du jeu fonctionne désormais, mais l'utilisateur ne peut faire qu'une seule supposition. Corrigeons cela en ajoutant une boucle !

Permettre plusieurs suppositions avec les boucles

Le mot-clé `loop` crée une boucle infinie. Nous allons ajouter une boucle pour donner aux utilisateurs plus de chances de deviner le nombre :

Fichier : src/main.rs

```
    // -- partie masquée ici --

    println!("Le nombre secret est : {}", nombre_secret);

    loop {
        println!("Veuillez entrer un nombre.");

        // -- partie masquée ici --

        match supposition.cmp(&nombre_secret) {
            Ordering::Less => println!("C'est plus !"),
            Ordering::Greater => println!("C'est moins !"),
            Ordering::Equal => println!("Vous avez gagné !"),
        }
    }
}
```

Comme vous pouvez le remarquer, nous avons déplacé dans une boucle tout le code de l'invite à entrer le nombre. Assurez-vous d'indenter correctement les lignes dans la boucle avec quatre nouvelles espaces pour chacune, et lancez à nouveau le programme. Le programme va désormais demander un nombre à l'infini, ce qui est un nouveau problème. Il n'est pas possible pour l'utilisateur de l'arrêter !

L'utilisateur pourrait quand même interrompre le programme en utilisant le raccourci clavier ctrl-c. Mais il y a une autre façon d'échapper à ce monstre insatiable, comme nous l'avons abordé dans la partie “Comparer le nombre saisi au nombre secret” : si l'utilisateur saisit quelque chose qui n'est pas un nombre, le programme va planter. Nous pouvons procéder ainsi pour permettre à l'utilisateur de quitter, comme ci-dessous :

```
$ cargo run
   Compiling jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
     Running `target/debug/jeu_du_plus_ou_du_moins`
Devinez le nombre !
Le nombre secret est : 59
Veuillez entrer un nombre.
45
Votre nombre : 45
C'est plus !
Veuillez entrer un nombre.
60
Votre nombre : 60
C'est moins !
Veuillez entrer un nombre.
59
Votre nombre : 59
Vous avez gagné !
Veuillez entrer un nombre.
quitter
thread 'main' panicked at 'Veuillez entrer un nombre !: ParseIntError { kind:
InvalidDigit }', src/main.rs:28:47
note: Run with `RUST_BACKTRACE=1` for a backtrace
```

Taper `quitter` va bien fermer le jeu, mais comme vous pouvez le remarquer, toute autre saisie qui n'est pas un nombre le ferait aussi. Ce mécanisme laisse franchement à désirer ; nous voudrions que le jeu s'arrête aussi lorsque le bon nombre est deviné.

Arrêter le programme après avoir gagné

Faisons en sorte que le jeu s'arrête quand le joueur gagne en ajoutant l'instruction `break` :

Fichier : src/main.rs

```
        // -- partie masquée ici --

        match supposition.cmp(&nombre_secret) {
            Ordering::Less => println!("C'est plus !"),
            Ordering::Greater => println!("C'est moins !"),
            Ordering::Equal => {
                println!("Vous avez gagné !");
                break;
            }
        }
    }
}
```

Ajouter la ligne `break` après `Vous avez gagné !` fait sortir le programme de la boucle quand le joueur a correctement deviné le nombre secret. Et quitter la boucle veut aussi dire terminer le programme, car ici la boucle est la dernière partie de `main`.

Gérer les saisies invalides

Pour améliorer le comportement du jeu, plutôt que de faire planter le programme quand l'utilisateur saisit quelque chose qui n'est pas un nombre, faisons en sorte que le jeu ignore ce qui n'est pas un nombre afin que l'utilisateur puisse continuer à essayer de deviner. Nous pouvons faire ceci en modifiant la ligne où `supposition` est converti d'une `String` en un `u32`, comme dans l'encart 2-5 :

Fichier : src/main.rs

```
// -- partie masquée ici --

io::stdin()
    .read_line(&mut supposition)
    .expect("Échec de la lecture de l'entrée utilisateur");

let supposition: u32 = match supposition.trim().parse() {
    Ok(nombre) => nombre,
    Err(_) => continue,
};

println!("Votre nombre : {}", supposition);

// -- partie masquée ici --
```

Encart 2-5 : Ignorer une saisie qui n'est pas un nombre et demander un nouveau nombre plutôt que de faire planter le programme

Nous remplaçons un appel à `expect` par une expression `match` pour passer d'une erreur qui fait planter le programme à une erreur proprement gérée. N'oubliez pas que `parse` retourne un type `Result` et que `Result` est une énumération qui a pour variantes `Ok` et `Err`. Nous utilisons ici une expression `match` comme nous l'avons déjà fait avec le résultat de type `Ordering` de la méthode `cmp`.

Si `parse` arrive à convertir la chaîne de caractères en nombre, cela va retourner la variante `Ok` qui contient le nombre qui en résulte. Cette variante va correspondre au motif de la première branche, et l'expression `match` va simplement retourner la valeur de `nombre` que `parse` a trouvée et qu'elle a mise dans la variante `Ok`. Ce nombre va se retrouver là où nous en avons besoin, dans la variable `supposition` que nous sommes en train de créer.

Si `parse` n'arrive *pas* à convertir la chaîne de caractères en nombre, elle va retourner la variante `Err` qui contient plus d'informations sur l'erreur. La variante `Err` ne correspond pas au motif `Ok(nombre)` de la première branche, mais elle correspond au motif `Err(_)` de la seconde branche. Le tiret bas, `_`, est une valeur passe-partout ; dans notre exemple, nous disons que nous voulons correspondre à toutes les valeurs de `Err`, peu importe quelle information elles ont à l'intérieur d'elles-mêmes. Donc le programme va exécuter le code de la seconde branche, `continue`, qui indique au programme de se rendre à la prochaine itération de `loop` et de demander un nouveau nombre. Ainsi, le programme ignore toutes les erreurs que `parse` pourrait rencontrer !

Maintenant, le programme devrait fonctionner correctement. Essayons-le :

```
$ cargo run
   Compiling jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
    Finished dev [unoptimized + debuginfo] target(s) in 4.45s
     Running `target/debug/jeu_du_plus_ou_du_moins`
Devinez le nombre !
Le nombre secret est : 61
Veuillez entrer un nombre.
10
Votre nombre : 10
C'est plus !
Veuillez entrer un nombre.
99
Votre nombre : 99
C'est moins !
Veuillez entrer un nombre.
foo
Veuillez entrer un nombre.
61
Votre nombre : 61
Vous avez gagné !
```

Super ! Avec notre petite touche finale, nous avons fini notre jeu du plus ou du moins. Rappelez-vous que le programme affiche toujours le nombre secret. C'était pratique pour les tests, mais cela gâche le jeu. Supprimons le `println!` qui affiche le nombre secret. L'encart 2-6 représente le code final.

Fichier : src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Devinez le nombre !");

    let nombre_secret = rand::thread_rng().gen_range(1..101);

    loop {
        println!("Veuillez entrer un nombre.");

        let mut supposition = String::new();

        io::stdin()
            .read_line(&mut supposition)
            .expect("Échec de la lecture de l'entrée utilisateur");

        let supposition: u32 = match supposition.trim().parse() {
            Ok(nombre) => nombre,
            Err(_) => continue,
        };

        println!("Votre nombre : {}", supposition);

        match supposition.cmp(&nombre_secret) {
            Ordering::Less => println!("C'est plus !"),
            Ordering::Greater => println!("C'est moins !"),
            Ordering::Equal => {
                println!("Vous avez gagné !");
                break;
            }
        }
    }
}
```

Encart 2-6 : Code complet du jeu du plus ou du moins

Résumé

Si vous êtes arrivé jusqu'ici, c'est que vous avez construit avec succès le jeu du plus ou du moins. Félicitations !

Ce projet était une mise en pratique pour vous initier à de nombreux concepts de Rust : `let`, `match`, les méthodes, les fonctions associées, l'utilisation de *crates* externes, et bien plus. Dans les prochains chapitres, vous allez en apprendre plus sur ces concepts. Le chapitre 3 va traiter des concepts utilisés par la plupart des langages de programmation,

comme les variables, les types de données, et les fonctions, et vous montrera comment les utiliser avec Rust. Le chapitre 4 expliquera la possession *(ownership)*, qui est une fonctionnalité qui distingue Rust des autres langages. Le chapitre 5 abordera les structures et les syntaxes des méthodes, et le chapitre 6 expliquera comment les énumérations fonctionnent.

Les concepts courants de programmation

Ce chapitre explique des concepts qui apparaissent dans presque tous les langages de programmation, et la manière dont ils fonctionnent en Rust. De nombreux langages sont basés sur des concepts communs. Les concepts présentés dans ce chapitre ne sont pas spécifiques à Rust, mais nous les appliquerons à Rust et nous expliquerons les conventions qui leur sont liées.

Plus précisément, vous allez apprendre les concepts de variables, les types de base, les fonctions, les commentaires, et les structures de contrôle. Ces notions fondamentales seront présentes dans tous les programmes Rust, et les apprendre dès le début vous procurera de solides bases pour débuter.

Mots-clés

Le langage Rust possède un ensemble de *mots-clés* qui ont été réservés pour l'usage exclusif du langage, tout comme le font d'autres langages. Gardez à l'esprit que vous ne pouvez pas utiliser ces mots pour des noms de variables ou de fonctions. La plupart des mots-clés ont une signification spéciale, et vous les utiliserez pour réaliser de différentes tâches dans vos programmes Rust ; quelques-uns n'ont aucune fonctionnalité active pour le moment, mais ont été réservés pour être ajoutés plus tard à Rust. Vous pouvez trouver la liste de ces mots-clés dans l'annexe A.

Les variables et la mutabilité

Tel qu'abordé au chapitre 2, par défaut, les variables sont *immuables*. C'est un des nombreux coups de pouce de Rust pour écrire votre code de façon à garantir la sécurité et la concurrence sans problème. Cependant, vous avez quand même la possibilité de rendre vos variables mutables *(modifiables)*. Explorons comment et pourquoi Rust vous encourage à favoriser l'immuabilité, et pourquoi parfois vous pourriez choisir d'y renoncer.

Lorsqu'une variable est immuable, cela signifie qu'une fois qu'une valeur est liée à un nom, vous ne pouvez pas changer cette valeur. À titre d'illustration, générons un nouveau projet appelé *variables* dans votre dossier *projects* en utilisant `cargo new variables`.

Ensuite, dans votre nouveau dossier *variables*, ouvrez *src/main.rs* et remplacez son code par le code suivant. Ce code ne se compile pas pour le moment, nous allons commencer par étudier l'erreur d'immutabilité.

Fichier : src/main.rs

```
fn main() {
    let x = 5;
    println!("La valeur de x est : {}", x);
    x = 6;
    println!("La valeur de x est : {}", x);
}
```

Sauvegardez et lancez le programme en utilisant `cargo run`. Vous devriez avoir un message d'erreur comme celui-ci :

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     println!("La valeur de x est : {}", x);
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `variables` due to previous error
```

Cet exemple montre comment le compilateur vous aide à trouver les erreurs dans vos

programmes. Les erreurs de compilation peuvent s'avérer frustrantes, mais elles signifient en réalité que, pour le moment, votre programme n'est pas en train de faire ce que vous voulez qu'il fasse en toute sécurité ; elles ne signifient *pas* que vous êtes un mauvais développeur ! Même les Rustacés expérimentés continuent d'avoir des erreurs de compilation.

Ce message d'erreur indique que la cause du problème est qu'il est `impossible d'assigner à deux reprises la variable immuable `x`` (`cannot assign twice to immutable variable `x``).

Il est important que nous obtenions des erreurs au moment de la compilation lorsque nous essayons de changer une valeur qui a été déclarée comme immuable, car cette situation particulière peut donner lieu à des bogues. Si une partie de notre code part du principe qu'une valeur ne changera jamais et qu'une autre partie de notre code modifie cette valeur, il est possible que la première partie du code ne fasse pas ce pour quoi elle a été conçue. La cause de ce genre de bogue peut être difficile à localiser après coup, en particulier lorsque la seconde partie du code ne modifie que *parfois* cette valeur. Le compilateur Rust garantit que lorsque vous déclarez qu'une valeur ne change pas, elle ne va jamais changer, donc vous n'avez pas à vous en soucier. Votre code est ainsi plus facile à maîtriser.

Mais la mutabilité peut s'avérer très utile, et peut faciliter la rédaction du code. Les variables sont immuables par défaut ; mais comme vous l'avez fait au chapitre 2, vous pouvez les rendre mutables en ajoutant `mut` devant le nom de la variable. L'ajout de `mut` va aussi signaler l'intention aux futurs lecteurs de ce code que d'autres parties du code vont modifier la valeur de cette variable.

Par exemple, modifions *src/main.rs* ainsi :

Fichier : src/main.rs

```
fn main() {
    let mut x = 5;
    println!("La valeur de x est : {}", x);
    x = 6;
    println!("La valeur de x est : {}", x);
}
```

Lorsque nous exécutons le programme, nous obtenons :

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/variables`
La valeur de x est : 5
La valeur de x est : 6
```

En utilisant `mut`, nous avons permis à la valeur liée à `x` de passer de `5` à `6`. Il y a d'autres compromis à envisager, en plus de la prévention des bogues. Par exemple, dans le cas où vous utiliseriez des grosses structures de données, muter une instance déjà existante peut être plus rapide que copier et retourner une instance nouvellement allouée. Avec des structures de données plus petites, créer de nouvelles instances avec un style de programmation fonctionnelle peut rendre le code plus facile à comprendre, donc il peut valoir le coup de sacrifier un peu de performance pour que le code gagne en clarté.

Les constantes

Comme les variables immuables, les *constantes* sont des valeurs qui sont liées à un nom et qui ne peuvent être modifiées, mais il y a quelques différences entre les constantes et les variables.

D'abord, vous ne pouvez pas utiliser `mut` avec les constantes. Les constantes ne sont pas seulement immuables par défaut − elles sont toujours immuables. On déclare les constantes en utilisant le mot-clé `const` à la place du mot-clé `let`, et le type de la valeur *doit* être indiqué. Nous allons aborder les types et les annotations de types dans la prochaine section, “Les types de données”, donc ne vous souciez pas des détails pour le moment. Sachez seulement que vous devez toujours indiquer le type.

Les constantes peuvent être déclarées à n'importe quel endroit du code, y compris la portée globale, ce qui les rend très utiles pour des valeurs que de nombreuses parties de votre code ont besoin de connaître.

La dernière différence est que les constantes ne peuvent être définies que par une expression constante, et non pas le résultat d'une valeur qui ne pourrait être calculée qu'à l'exécution.

Voici un exemple d'une déclaration de constante :

```
const TROIS_HEURES_EN_SECONDES: u32 = 60 * 60 * 3;
```

Le nom de la constante est `TROIS_HEURES_EN_SECONDES` et sa valeur est définie comme étant le résultat de la multiplication de 60 (le nombre de secondes dans une minute) par 60 (le nombre de minutes dans une heure) par 3 (le nombre d'heures que nous voulons calculer dans ce programme). En Rust, la convention de nommage des constantes est de les écrire tout en majuscule avec des tirets bas entre les mots. Le compilateur peut calculer un certain nombre d'opérations à la compilation, ce qui nous permet d'écrire cette valeur de façon à la comprendre plus facilement et à la vérifier, plutôt que de définir cette valeur à 10 800. Vous

pouvez consulter la section de la référence Rust à propos des évaluations des constantes pour en savoir plus sur les opérations qui peuvent être utilisées pour déclarer des constantes.

Les constantes sont valables pendant toute la durée d'exécution du programme au sein de la portée dans laquelle elles sont déclarées. Cette caractéristique rends les constantes très utiles lorsque plusieurs parties du programme doivent connaître certaines valeurs, comme par exemple le nombre maximum de points qu'un joueur est autorisé à gagner ou encore la vitesse de la lumière.

Déclarer des valeurs codées en dur et utilisées tout le long de votre programme en tant que constantes est utile pour faire comprendre la signification de ces valeurs dans votre code aux futurs développeurs. Cela permet également de n'avoir qu'un seul endroit de votre code à modifier si cette valeur codée en dur doit être mise à jour à l'avenir.

Le masquage

Comme nous l'avons vu dans le Chapitre 2, on peut déclarer une nouvelle variable avec le même nom qu'une variable précédente. Les Rustacés disent que la première variable est *masquée* par la seconde, ce qui signifie que la valeur de la seconde variable sera ce que le programme verra lorsque nous utiliserons cette variable. Nous pouvons créer un masque d'une variable en utilisant le même nom de variable et en réutilisant le mot-clé `let` comme ci-dessous :

Fichier : src/main.rs

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("La valeur de x dans la portée interne est : {}", x);
    }

    println!("La valeur de x est : {}", x);
}
```

Au début, ce programme lie `x` à la valeur `5`. Puis il crée un masque de `x` en répétant `let x =`, en récupérant la valeur d'origine et lui ajoutant `1` : la valeur de `x` est désormais `6`. Ensuite, à l'intérieur de la portée interne, la troisième instruction `let` crée un autre masque de `x`, en récupérant la précédente valeur et en la multipliant par `2` pour donner à `x` la

valeur finale de `12`. Dès que nous sortons de cette portée, le masque prends fin, et `x` revient à la valeur `6`. Lorsque nous exécutons ce programme, nous obtenons ceci :

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/variables`
La valeur de x dans la portée interne est : 12
La valeur de x est : 6
```

Créer un masque est différent que de marquer une variable comme étant `mut`, car à moins d'utiliser une nouvelle fois le mot-clé `let`, nous obtiendrons une erreur de compilation si nous essayons de réassigner cette variable par accident. Nous pouvons effectuer quelques transformations sur une valeur en utilisant `let`, mais faire en sorte que la variable soit immuable après que ces transformations ont été appliquées.

Comme nous créons une nouvelle variable lorsque nous utilisons le mot-clé `let` une nouvelle fois, l'autre différence entre le `mut` et la création d'un masque est que cela nous permet de changer le type de la valeur, mais en réutilisant le même nom. Par exemple, imaginons un programme qui demande à l'utilisateur le nombre d'espaces qu'il souhaite entre deux portions de texte en saisissant des espaces, et ensuite nous voulons stocker cette saisie sous forme de nombre :

```
let espaces = "   ";
let espaces = espaces.len();
```

La première variable `espaces` est du type chaîne de caractères *(string)* et la seconde variable `espaces` est du type nombre. L'utilisation du masquage nous évite ainsi d'avoir à trouver des noms différents, comme `espaces_str` et `espaces_num` ; nous pouvons plutôt simplement réutiliser le nom `espaces`. Cependant, si nous essayons d'utiliser `mut` pour faire ceci, comme ci-dessous, nous avons une erreur de compilation :

```
let mut espaces = "   ";
espaces = espaces.len();
```

L'erreur indique que nous ne pouvons pas muter le type d'une variable :

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0308]: mismatched types
 --> src/main.rs:3:14
  |
2 |     let mut espaces = "   ";
  |                       ----- expected due to this value
3 |     espaces = espaces.len();
  |               ^^^^^^^^^^^^ expected `&str`, found `usize`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `variables` due to previous error
```

Maintenant que nous avons découvert comment fonctionnent les variables, étudions les types de données qu'elles peuvent prendre.

Les types de données

Chaque valeur en Rust est d'un *type* bien déterminé, qui indique à Rust quel genre de données il manipule pour qu'il sache comment traiter ces données. Nous allons nous intéresser à deux catégories de types de données : les scalaires et les composés.

Gardez à l'esprit que Rust est un langage *statiquement typé*, ce qui signifie qu'il doit connaître les types de toutes les variables au moment de la compilation. Le compilateur peut souvent déduire quel type utiliser en se basant sur la valeur et sur la façon dont elle est utilisée. Dans les cas où plusieurs types sont envisageables, comme lorsque nous avons converti une chaîne de caractères en un type numérique en utilisant `parse` dans la section "Comparer le nombre saisi au nombre secret" du chapitre 2, nous devons ajouter une annotation de type, comme ceci :

```
let supposition: u32 = "42".parse().expect("Ce n'est pas un nombre !");
```

Si nous n'ajoutons pas l'annotation de type ici, Rust affichera l'erreur suivante, signifiant que le compilateur a besoin de plus d'informations pour déterminer quel type nous souhaitons utiliser :

```
$ cargo build
   Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let supposition = "42".parse().expect("Ce n'est pas un nombre !");
  |         ^^^^^^^^^^^ consider giving `supposition` a type

For more information about this error, try `rustc --explain E0282`.
error: could not compile `no_type_annotations` due to previous error
```

Vous découvrirez différentes annotations de type au fur et à mesure que nous aborderons les autres types de données.

Types scalaires

Un type *scalaire* représente une seule valeur. Rust possède quatre types principaux de scalaires : les entiers, les nombres à virgule flottante, les booléens et les caractères. Vous les connaissez sûrement d'autres langages de programmation. Regardons comment ils fonctionnent avec Rust.

Types de nombres entiers

Un *entier* est un nombre sans partie décimale. Nous avons utilisé un entier précédemment dans le chapitre 2, le type `u32`. Cette déclaration de type indique que la valeur à laquelle elle est associée doit être un entier non signé encodé sur 32 bits dans la mémoire (les entiers pouvant prendre des valeurs négatives commencent par un `i` (comme *integer* : “entier”), plutôt que par un `u` comme *unsigned* : “non signé”). Le tableau 3-1 montre les types d'entiers intégrés au langage. Nous pouvons utiliser chacune de ces variantes pour déclarer le type d'une valeur entière.

Tableau 3-1 : les types d'entiers en Rust

Taille	Signé	Non signé
8 bits	`i8`	`u8`
16 bits	`i16`	`u16`
32 bits	`i32`	`u32`
64 bits	`i64`	`u64`
128 bits	`i128`	`u128`
archi	`isize`	`usize`

Chaque variante peut être signée ou non signée et possède une taille explicite. *Signé* et *non signé* veut dire respectivement que le nombre peut prendre ou non des valeurs négatives — en d'autres termes, si l'on peut lui attribuer un signe (signé) ou s'il sera toujours positif et que l'on peut donc le représenter sans signe (non signé). C'est comme écrire des nombres sur du papier : quand le signe est important, le nombre est écrit avec un signe plus ou un signe moins ; en revanche, quand le nombre est forcément positif, on peut l'écrire sans son signe. Les nombres signés sont stockés en utilisant le complément à deux.

Chaque variante signée peut stocker des nombres allant de $-(2^{n-1})$ à $2^{n-1} - 1$ inclus, où n est le nombre de bits que cette variante utilise. Un `i8` peut donc stocker des nombres allant de $-(2^7)$ à $2^7 - 1$, c'est-à-dire de −128 à 127. Les variantes non signées peuvent stocker des nombres de 0 à $2^n - 1$, donc un `u8` peut stocker des nombres allant de 0 à $2^8 - 1$, c'est-à-dire de 0 à 255.

De plus, les types `isize` et `usize` dépendent de l'architecture de l'ordinateur sur lequel votre programme va s'exécuter, d'où la ligne “archi” : 64 bits si vous utilisez une architecture 64 bits, ou 32 bits si vous utilisez une architecture 32 bits.

Vous pouvez écrire des littéraux d'entiers dans chacune des formes décrites dans le tableau 3-2. Notez que les littéraux numériques qui peuvent être de plusieurs types numériques

autorisent l'utilisation d'un suffixe de type, tel que `57u8`, afin de préciser leur type. Les nombres littéraux peuvent aussi utiliser `_` comme séparateur visuel afin de les rendre plus lisible, comme par exemple `1_000`, qui a la même valeur que si vous aviez renseigné `1000`.

Tableau 3-2 : les littéraux d'entiers en Rust

Littéral numérique	Exemple
Décimal	`98_222`
Hexadécimal	`0xff`
Octal	`0o77`
Binaire	`0b1111_0000`
Octet (`u8` seulement)	`b'A'`

Comment pouvez-vous déterminer le type d'entier à utiliser ? Si vous n'êtes pas sûr, les choix par défaut de Rust sont généralement de bons choix : le type d'entier par défaut est le `i32`. La principale utilisation d'un `isize` ou d'un `usize` est lorsque l'on indexe une quelconque collection.

Dépassement d'entier

Imaginons que vous avez une variable de type `u8` qui peut stocker des valeurs entre 0 et 255. Si vous essayez de changer la variable pour une valeur en dehors de cet intervalle, comme 256, vous aurez un dépassement d'entier *(integer overflow)*, qui peut se compter de deux manière. Lorsque vous compilez en mode débogage, Rust embarque des vérifications pour détecter les cas de dépassements d'entiers qui pourraient faire *paniquer* votre programme à l'exécution si ce phénomène se produit. Rust utilise le terme *paniquer* quand un programme se termine avec une erreur ; nous verrons plus en détail les *paniques* dans une section du chapitre 9.

Lorsque vous compilez en mode publication *(release)* avec le drapeau `--release`, Rust ne va *pas* vérifier les potentiels dépassements d'entiers qui peuvent faire paniquer le programme. En revanche, en cas de dépassement, Rust va effectuer un *rebouclage du complément à deux*. Pour faire simple, les valeurs supérieures à la valeur maximale du type seront “rebouclées” depuis la valeur minimale que le type peut stocker. Dans le cas d'un `u8`, la valeur 256 devient 0, la valeur 257 devient 1, et ainsi de suite. Le programme ne va paniquer, mais la variable va avoir une valeur qui n'est probablement pas ce que vous attendez à avoir. Se fier au comportement du rebouclage lors du dépassement d'entier est considéré comme une faute.

Pour gérer explicitement le dépassement, vous pouvez utiliser les familles de

méthodes suivantes qu'offrent la bibliothèque standard sur les types de nombres primitifs :

- Enveloppez les opérations avec les méthodes `wrapping_*`, comme par exemple `wrapping_add`
- Retourner la valeur `None` s'il y a un dépassement avec des méthodes `checked_*`
- Retourner la valeur et un booléen qui indique s'il y a eu un dépassement avec des méthodes `overflowing_*`
- Saturer à la valeur minimale ou maximale avec des méthodes `saturating_*`

Types de nombres à virgule flottante

Rust possède également deux types primitifs pour les *nombres à virgule flottante* (ou *flottants*), qui sont des nombres avec des décimales. Les types de flottants en Rust sont les `f32` et les `f64`, qui ont respectivement une taille en mémoire de 32 bits et 64 bits. Le type par défaut est le `f64` car sur les processeurs récents ce type est quasiment aussi rapide qu'un `f32` mais est plus précis. Tous les flottants ont un signe.

Voici un exemple montrant l'utilisation de nombres à virgule flottante :

Ficher : src/main.rs

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

Les nombres à virgule flottante sont représentés selon la norme IEEE-754. Le type `f32` est un flottant à simple précision, et le `f64` est à double précision.

Les opérations numériques

Rust offre les opérations mathématiques de base dont vous auriez besoin pour tous les types de nombres : addition, soustraction, multiplication, division et modulo. Les divisions d'entiers arrondissent le résultat à l'entier le plus près. Le code suivant montre comment utiliser chacune des opérations numériques avec une instruction `let` :

Fichier : src/main.rs

```
fn main() {
    // addition
    let somme = 5 + 10;

    // soustraction
    let difference = 95.5 - 4.3;

    // multiplication
    let produit = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;
    let arrondi = 2 / 3; // retournera 0

    // modulo
    let reste = 43 % 5;
}
```

Chaque expression de ces instructions utilise un opérateur mathématique et calcule une valeur unique, qui est ensuite attribuée à une variable. L'annexe B présente une liste de tous les opérateurs que Rust fournit.

Le type booléen

Comme dans la plupart des langages de programmation, un type booléen a deux valeurs possibles en Rust : `true` (vrai) et `false` (faux). Les booléens prennent un octet en mémoire. Le type booléen est désigné en utilisant `bool`. Par exemple :

Fichier : src/main.rs

```
fn main() {
    let t = true;

    let f: bool = false; // avec une annotation de type explicite
}
```

Les valeurs booléennes sont principalement utilisées par les structures conditionnelles, comme l'expression `if`. Nous aborderons le fonctionnement de `if` en Rust dans la section “Les structures de contrôle”.

Le type caractère

Le type `char` (comme *character*) est le type de caractère le plus rudimentaire. Voici quelques exemples de déclaration de valeurs de type `char` :

Fichier : src/main.rs

```
fn main() {
    let c = 'z';
    let z = 'ℤ';
    let chat_aux_yeux_de_coeur = '😻';
}
```

Notez que nous renseignons un littéral `char` avec des guillemets simples, contrairement aux littéraux de chaîne de caractères, qui nécéssite des doubles guillemets. Le type `char` de Rust prend quatre octets en mémoire et représente une valeur scalaire Unicode, ce qui veut dire que cela représente plus de caractères que l'ASCII. Les lettres accentuées ; les caractères chinois, japonais et coréens ; les emoji ; les espaces de largeur nulle ont tous une valeur pour `char` avec Rust. Les valeurs scalaires Unicode vont de `U+0000` à `U+D7FF` et de `U+E000` à `U+10FFFF` inclus. Cependant, le concept de “caractère” n'est pas clairement défini par Unicode, donc votre notion de “caractère” peut ne pas correspondre à ce qu'est un `char` en Rust. Nous aborderons ce sujet plus en détail au chapitre 8.

Les types composés

Les *types composés* peuvent regrouper plusieurs valeurs dans un seul type. Rust a deux types composés de base : les *tuples* et les tableaux *(arrays)*.

Le type *tuple*

Un *tuple* est une manière générale de regrouper plusieurs valeurs de types différents en un seul type composé. Les tuples ont une taille fixée : à partir du moment où ils ont été déclarés, on ne peut pas y ajouter ou enlever des valeurs.

Nous créons un *tuple* en écrivant une liste séparée par des virgules entre des parenthèses. Chaque emplacement dans le tuple a un type, et les types de chacune des valeurs dans le tuple n'ont pas forcément besoin d'être les mêmes. Nous avons ajouté des annotations de type dans cet exemple, mais c'est optionnel :

Fichier : src/main.rs

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

La variable `tup` est liée à tout le tuple, car un tuple est considéré comme étant un unique élément composé. Pour obtenir un élément précis de ce tuple, nous pouvons utiliser un

filtrage par motif *(pattern matching)* pour déstructurer ce tuple, comme ceci :

Fichier : src/main.rs

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("La valeur de y est : {}", y);
}
```

Le programme commence par créer un tuple et il l'assigne à la variable `tup`. Il utilise ensuite un motif avec `let` pour prendre `tup` et le scinder en trois variables distinctes : `x`, `y`, et `z`. On appelle cela *déstructurer*, car il divise le tuple en trois parties. Puis finalement, le programme affiche la valeur de `y`, qui est `6.4`.

Nous pouvons aussi accéder directement à chaque élément du tuple en utilisant un point (`.`) suivi de l'indice de la valeur que nous souhaitons obtenir. Par exemple :

Fichier : src/main.rs

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let cinq_cents = x.0;

    let six_virgule_quatre = x.1;

    let un = x.2;
}
```

Ce programme crée le tuple `x` puis crée une nouvelle variable pour chaque élément en utilisant leur indices respectifs. Comme dans de nombreux langages de programmation, le premier indice d'un tuple est 0.

Le tuple sans aucune valeur, `()`, est un type spécial qui a une seule et unique valeur, qui s'écrit aussi `()`. Ce type est aussi appelé le *type unité* et la valeur est appelée *valeur unité*. Les expressions retournent implicitement la valeur unité si elles ne retournent aucune autre valeur.

Le type tableau

Un autre moyen d'avoir une collection de plusieurs valeurs est d'utiliser un *tableau*. Contrairement aux tuples, chaque élément d'un tableau doit être du même type.

Contrairement aux tableaux de certains autres langages, les tableaux de Rust ont une taille fixe.

Nous écrivons les valeurs dans un tableau via une liste entre des crochets, séparée par des virgules :

Fichier : src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

Les tableaux sont utiles quand vous voulez que vos données soient allouées sur la pile *(stack)* plutôt que sur le tas *(heap)* (nous expliquerons la pile et le tas au chapitre 4) ou lorsque vous voulez vous assurer que vous avez toujours un nombre fixe d'éléments. Cependant, un tableau n'est pas aussi flexible qu'un vecteur *(vector)*. Un vecteur est un type de collection de données similaire qui est fourni par la bibliothèque standard qui, lui, peut grandir ou rétrécir en taille. Si vous ne savez pas si vous devez utiliser un tableau ou un vecteur, il y a de fortes chances que vous devriez utiliser un vecteur. Le chapitre 8 expliquera les vecteurs.

Toutefois, les tableaux s'avèrent plus utiles lorsque vous savez que le nombre d'éléments n'aura pas besoin de changer. Par exemple, si vous utilisez les noms des mois dans un programme, vous devriez probablement utiliser un tableau plutôt qu'un vecteur car vous savez qu'il contient toujours 12 éléments :

```
let mois = ["Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet",
            "Août", "Septembre", "Octobre", "Novembre", "Décembre"];
```

Vous pouvez écrire le type d'un tableau en utilisant des crochets et entre ces crochets y ajouter le type de chaque élément, un point-virgule, et ensuite le nombre d'éléments dans le tableau, comme ceci :

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

Ici, `i32` est le type de chaque élément. Après le point-virgule, le nombre `5` indique que le tableau contient cinq éléments.

Vous pouvez initialiser un tableau pour qu'il contienne toujours la même valeur pour chaque élément, vous pouvez préciser la valeur initiale, suivie par un point-virgule, et ensuite la taille du tableau, le tout entre crochets, comme ci-dessous :

```
let a = [3; 5];
```

Le tableau `a` va contenir `5` éléments qui auront tous la valeur initiale `3`. C'est la même chose que d'écrire `let a = [3, 3, 3, 3, 3];` mais de manière plus concise.

Accéder aux éléments d'un tableau

Un tableau est un simple bloc de mémoire de taille connue et fixe, qui peut être alloué sur la pile. Vous pouvez accéder aux éléments d'un tableau en utilisant l'indexation, comme ceci :

Fichier : src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let premier = a[0];
    let second = a[1];
}
```

Dans cet exemple, la variable qui s'appelle `premier` aura la valeur `1`, car c'est la valeur à l'indice `[0]` dans le tableau. La variable `second` récupèrera la valeur `2` depuis l'indice `[1]` du tableau.

Accès incorrect à un élément d'un tableau

Découvrons ce qui se passe quand vous essayez d'accéder à un élément d'un tableau qui se trouve après la fin du tableau ? Imaginons que vous exécutiez le code suivant, similaire au jeu du plus ou du moins du chapitre 2, pour demander un indice de tableau à l'utilisateur :

Fichier : src/main.rs

```
use std::io;

fn main() {
    let a = [1, 2, 3, 4, 5];

    println!("Veuillez entrer un indice de tableau.");

    let mut indice = String::new();

    io::stdin()
        .read_line(&mut indice)
        .expect("Échec de la lecture de l'entrée utilisateur");

    let indice: usize = indice
        .trim()
        .parse()
        .expect("L'indice entré n'est pas un nombre");

    let element = a[indice];

    println!(
        "La valeur de l'élément d'indice {} est : {}",
        indice, element
    );
}
```

Ce code compile avec succès. Si vous exécutez ce code avec `cargo run` et que vous entrez 0, 1, 2, 3 ou 4, le programme affichera la valeur correspondante à cet indice dans le tableau. Si au contraire, vous entrez un indice après la fin du tableau tel que 10, ceci s'affichera :

```
thread 'main' panicked at 'index out of bounds: the len is 5 but the index is
10', src/main.rs:19:19
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Le programme a rencontré une erreur *à l'exécution*, au moment d'utiliser une valeur invalide comme indice. Le programme s'est arrêté avec un message d'erreur et n'a pas exécuté la dernière instruction `println!`. Quand vous essayez d'accéder à un élément en utilisant l'indexation, Rust va vérifier que l'indice que vous avez demandé est plus petit que la taille du tableau. Si l'indice est supérieur ou égal à la taille du tableau, Rust va *paniquer*. Cette vérification doit avoir lieu à l'exécution, surtout dans ce cas, parce que le compilateur ne peut pas deviner la valeur qu'entrera l'utilisateur quand il exécutera le code plus tard.

C'est un exemple de mise en pratique des principes de sécurité de la mémoire par Rust. Dans de nombreux langages de bas niveau, ce genre de vérification n'est pas effectuée, et quand vous utilisez un indice incorrect, de la mémoire invalide peut être récupérée. Rust vous protège de ce genre d'erreur en quittant immédiatement l'exécution au lieu de permettre l'accès en mémoire et continuer son déroulement. Le chapitre 9 expliquera la

gestion d'erreurs de Rust.

Les fonctions

Les fonctions sont très utilisées dans le code Rust. Vous avez déjà vu l'une des fonctions les plus importantes du langage : la fonction `main`, qui est le point d'entrée de beaucoup de programmes. Vous avez aussi vu le mot-clé `fn`, qui vous permet de déclarer des nouvelles fonctions.

Le code Rust utilise le *snake case* comme convention de style de nom des fonctions et des variables, toutes les lettres sont en minuscule et on utilise des tirets bas pour séparer les mots. Voici un programme qui est un exemple de définition de fonction :

Fichier : src/main.rs

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

    une_autre_fonction();
}

fn une_autre_fonction() {
    println!("Une autre fonction.");
}
```

Nous définissons une fonction avec Rust en saisissant `fn` suivi par un nom de fonction ainsi qu'une paire de parenthèses. Les accolades indiquent au compilateur où le corps de la fonction commence et où il se termine.

Nous pouvons appeler n'importe quelle fonction que nous avons définie en utilisant son nom, suivi d'une paire de parenthèses. Comme `une_autre_fonction` est définie dans le programme, elle peut être appelée à l'intérieur de la fonction `main`. Remarquez que nous avons défini `une_autre_fonction` *après* la fonction `main` dans le code source ; nous aurions aussi pu la définir avant. Rust ne se soucie pas de l'endroit où vous définissez vos fonctions, du moment qu'elles sont bien définies quelque part.

Créons un nouveau projet de binaire qui s'appellera *functions* afin d'en apprendre plus sur les fonctions. Ajoutez l'exemple `une_autre_fonction` dans le *src/main.rs* et exécutez-le. Vous devriez avoir ceci :

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s
     Running `target/debug/functions`
Hello, world!
Une autre fonction.
```

Les lignes s'exécutent dans l'ordre dans lequel elles apparaissent dans la fonction `main`. D'abord, le message `Hello, world!` est écrit, et ensuite `une_autre_fonction` est appelée et son message est affiché.

Les paramètres

Nous pouvons définir des fonctions avec des *paramètres*, qui sont des variables spéciales qui font partie de la signature de la fonction. Quand une fonction a des paramètres, vous pouvez lui fournir des valeurs concrètes avec ces paramètres. Techniquement, ces valeurs concrètes sont appelées des *arguments*, mais dans une conversation courante, on a tendance à confondre les termes *paramètres* et *arguments* pour désigner soit les variables dans la définition d'une fonction, soit les valeurs concrètes passées quand on appelle une fonction.

Dans cette version de `une_autre_fonction`, nous ajoutons un paramètre :

Fichier : src/main.rs

```
fn main() {
    une_autre_fonction(5);
}

fn une_autre_fonction(x: i32) {
    println!("La valeur de x est : {}", x);
}
```

En exécutant ce programme, vous devriez obtenir ceci :

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 1.21s
     Running `target/debug/functions`
La valeur de x est : 5
```

La déclaration de `une_autre_fonction` a un paramètre nommé `x`. Le type de `x` a été déclaré comme `i32`. Quand nous passons `5` à `une_autre_fonction`, la macro `println!` place `5` là où la paire d'accolades `{}` a été placée dans la chaîne de formatage.

Dans la signature d'une fonction, vous *devez* déclarer le type de chaque paramètre. C'est un choix délibéré de conception de Rust : exiger l'annotation de type dans la définition d'une fonction fait en sorte que le compilateur n'a presque plus besoin que vous les utilisiez autre part pour qu'il comprenne avec quel type vous souhaitez travailler.

Lorsque vous définissez plusieurs paramètres, séparez les paramètres avec des virgules,

comme ceci :

Fichier : src/main.rs

```
fn main() {
    afficher_mesure_avec_unite(5, 'h');
}

fn afficher_mesure_avec_unite(valeur: i32, unite: char) {
    println!("La mesure est : {}{}", valeur, unite);
}
```

Cet exemple crée la fonction `afficher_mesure_avec_unite` qui a deux paramètres. Le premier paramètre s'appelle `valeur` et est un `i32`. Le second, `nom_unite`, est de type `char`. La fonction affiche ensuite le texte qui contient les valeurs de `valeur` et de `nom_unite`.

Essayons d'exécuter ce code. Remplacez le programme présent actuellement dans votre fichier *src/main.rs* de votre projet *functions* par l'exemple précédent et lancez-le en utilisant `cargo run` :

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/functions`
La mesure est : 5h
```

Comme nous avons appelé la fonction avec la valeur `5` pour `valeur` et `'h'` pour `nom_unite`, la sortie de ce programme contient ces valeurs.

Instructions et expressions

Les corps de fonctions sont constitués d'une série d'instructions qui se termine éventuellement par une expression. Jusqu'à présent, les fonctions que nous avons vu n'avaient pas d'expression à la fin, mais vous avez déjà vu une expression faire partie d'une instruction. Comme Rust est un langage basé sur des expressions, il est important de faire la distinction. D'autres langages ne font pas de telles distinctions, donc penchons-nous sur ce que sont les instructions et les expressions et comment leurs différences influent sur le corps des fonctions.

Les *instructions* effectuent des actions et ne retournent aucune valeur. Les *expressions* sont évaluées pour retourner une valeur comme résultat. Voyons quelques exemples.

Nous avons déjà utilisé des instructions et des expressions. La création d'une variable en lui

assignant une valeur avec le mot-clé `let` est une instruction. Dans l'encart 3-1, `let y = 6;` est une instruction.

Fichier : src/main.rs

```
fn main() {
    let y = 6;
}
```

Encart 3-1 : une fonction `main` qui contient une instruction

La définition d'une fonction est aussi une instruction ; l'intégralité de l'exemple précédent est une instruction à elle toute seule.

Une instruction ne retourne pas de valeur. Ainsi, vous ne pouvez pas assigner le résultat d'une instruction `let` à une autre variable, comme le code suivant essaye de le faire, car vous obtiendrez une erreur :

Fichier : src/main.rs

```
fn main() {
    let x = (let y = 6);
}
```

Quand vous exécutez ce programme, l'erreur que vous obtenez devrait ressembler à ceci :

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found statement (`let`)
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^^^^^^^
  |
  = note: variable declaration using `let` is a statement

error[E0658]: `let` expressions in this position are experimental
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^^^^^^^
  |
  = note: see issue #53667 <https://github.com/rust-lang/rust/issues/53667> for
more information
  = help: you can write `matches!(<expr>, <pattern>)` instead of `let <pattern>
= <expr>`

warning: unnecessary parentheses around assigned value
 --> src/main.rs:2:13
  |
2 |     let x = (let y = 6);
  |             ^         ^
  |
  = note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
  |
2 -     let x = (let y = 6);
2 +     let x = let y = 6;
  |

For more information about this error, try `rustc --explain E0658`.
warning: `functions` (bin "functions") generated 1 warning
error: could not compile `functions` due to 2 previous errors; 1 warning emitted
```

L'instruction `let y = 6` ne retourne pas de valeur, donc cela ne peut pas devenir une valeur de `x`. Ceci est différent d'autres langages, comme le C ou Ruby, où l'assignation retourne la valeur de l'assignation. Dans ces langages, vous pouvez écrire `x = y = 6` et avoir ainsi `x` et `y` qui ont chacun la valeur `6` ; cela n'est pas possible avec Rust.

Les expressions sont calculées en tant que valeur et seront ce que vous écrirez le plus en Rust (hormis les instructions). Prenez une opération mathématique, comme `5 + 6`, qui est une expression qui s'évalue à la valeur `11`. Les expressions peuvent faire partie d'une instruction : dans l'encart 3-1, le `6` dans l'instruction `let y = 6;` est une expression qui s'évalue à la valeur `6`. L'appel de fonction est aussi une expression. L'appel de macro est une expression. Un nouveau bloc de portée que nous créons avec des accolades est une

expression, par exemple :

Fichier : src/main.rs

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

    println!("La valeur de y est : {}", y);
}
```

L'expression suivante...

```
{
    let x = 3;
    x + 1
}
```

... est un bloc qui, dans ce cas, s'évalue à `4`. Cette valeur est assignée à `y` dans le cadre de l'instruction `let`. Remarquez la ligne `x + 1` ne se termine pas par un point-virgule, ce qui est différent de la plupart des lignes que vous avez vues jusque là. Les expressions n'ont pas de point-virgule de fin de ligne. Si vous ajoutez un point-virgule à la fin de l'expression, vous la transformez en instruction, et elle ne va donc pas retourner de valeur. Gardez ceci à l'esprit quand nous aborderons prochainement les valeurs de retour des fonctions ainsi que les expressions.

Les fonctions qui retournent des valeurs

Les fonctions peuvent retourner des valeurs au code qui les appelle. Nous ne nommons pas les valeurs de retour, mais nous devons déclarer leur type après une flèche (`->`). En Rust, la valeur de retour de la fonction est la même que la valeur de l'expression finale dans le corps de la fonction. Vous pouvez sortir prématurément d'une fonction en utilisant le mot-clé `return` et en précisant la valeur de retour, mais la plupart des fonctions vont retourner implicitement la dernière expression. Voici un exemple d'une fonction qui retourne une valeur :

Fichier : src/main.rs

```
fn cinq() -> i32 {
    5
}

fn main() {
    let x = cinq();

    println!("La valeur de x est : {}", x);
}
```

Il n'y a pas d'appel de fonction, de macro, ni même d'instruction `let` dans la fonction `cinq` — uniquement le nombre `5` tout seul. C'est une fonction parfaitement valide avec Rust. Remarquez que le type de retour de la fonction a été précisé aussi, avec `-> i32`. Essayez d'exécuter ce code ; le résultat devrait ressembler à ceci :

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/functions`
La valeur de x est : 5
```

Le `5` dans `cinq` est la valeur de retour de la fonction, ce qui explique le type de retour de `i32`. Regardons cela plus en détail. Il y a deux éléments importants : premièrement, la ligne `let x = cinq();` dit que nous utilisons la valeur de retour de la fonction pour initialiser la variable. Comme la fonction `cinq` retourne un `5`, cette ligne revient à faire ceci :

```
let x = 5;
```

Deuxièmement, la fonction `cinq` n'a pas de paramètre et déclare le type de valeur de retour, mais le corps de la fonction est un simple `5` sans point-virgule car c'est une expression dont nous voulons retourner la valeur.

Regardons un autre exemple :

Fichier : src/main.rs

```
fn main() {
    let x = plus_un(5);

    println!("La valeur de x est : {}", x);
}

fn plus_un(x: i32) -> i32 {
    x + 1
}
```

Exécuter ce code va afficher `La valeur de x est : 6`. Mais si nous ajoutons un point-virgule à la fin de la ligne qui contient `x + 1`, ce qui la transforme d'une expression à une instruction, nous obtenons une erreur.

Fichier : src/main.rs

```
fn main() {
    let x = plus_un(5);

    println!("La valeur de x est : {}", x);
}

fn plus_un(x: i32) -> i32 {
    x + 1;
}
```

Compiler ce code va produire une erreur, comme ci-dessous :

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
7 | fn plus_un(x: i32) -> i32 {
  |    -------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
  |          - help: consider removing this semicolon

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions` due to previous error
```

Le message d'erreur principal, “mismatched types” *(types inadéquats)* donne le cœur du problème de ce code. La définition de la fonction `plus_un` dit qu'elle va retourner un `i32`, mais les instructions ne retournent pas de valeur, ceci est donc représenté par `()`, le type unité. Par conséquent, rien n'est retourné, ce qui contredit la définition de la fonction et provoque une erreur. Rust affiche un message qui peut aider à corriger ce problème : il suggère d'enlever le point-virgule, ce qui va résoudre notre problème.

Les commentaires

Tous les développeurs s'efforcent de rendre leur code facile à comprendre, mais parfois il est nécessaire d'écrire des explications supplémentaires. Dans ce cas, les développeurs laissent des *commentaires* dans leur code source que le compilateur va ignorer mais qui peuvent être utiles pour les personnes qui lisent le code source.

Voici un simple commentaire :

```
// hello, world
```

Avec Rust, les commentaires classiques commencent avec deux barres obliques et continuent jusqu'à la fin de la ligne. Pour les commentaires qui font plus d'une seule ligne, vous aurez besoin d'ajouter `//` sur chaque ligne, comme ceci :

```
// Donc ici on fait quelque chose de compliqué, tellement long que nous avons
// besoin de plusieurs lignes de commentaires pour le faire ! Heureusement,
// ce commentaire va expliquer ce qui se passe.
```

Les commentaires peuvent aussi être aussi ajoutés à la fin d'une ligne qui contient du code :

Fichier : src/main.rs

```
fn main() {
    let nombre_chanceux = 7; // Je me sens chanceux aujourd'hui
}
```

Mais parfois, vous pourrez les voir utilisés de cette manière, avec le commentaire sur une ligne séparée au-dessus du code qu'il annote :

Fichier : src/main.rs

```
fn main() {
    // Je me sens chanceux aujourd'hui
    let nombre_chanceux = 7;
}
```

Rust a aussi un autre type de commentaire, les commentaires de documentation, que nous aborderons au chapitre 14.

Les structures de contrôle

Pouvoir exécuter ou non du code si une condition est vérifiée, ou exécuter du code de façon répétée tant qu'une condition est vérifiée, sont des constructions élémentaires dans la plupart des langages de programmation. Les structures de contrôle les plus courantes en Rust sont les expressions `if` et les boucles.

Les expressions `if`

Une expression `if` vous permet de diviser votre code en fonction de conditions. Vous précisez une condition et vous choisissez ensuite : “Si cette condition est remplie, alors exécuter ce bloc de code. Si la condition n'est pas remplie, ne pas exécuter ce bloc de code.”

Créez un nouveau projet appelé *branches* dans votre dossier *projects* pour découvrir les expressions `if` . Dans le fichier *src/main.rs*, écrivez ceci :

Fichier : src/main.rs

```
fn main() {
    let nombre = 3;

    if nombre < 5 {
        println!("La condition est vérifiée");
    } else {
        println!("La condition n'est pas vérifiée");
    }
}
```

Une expression `if` commence par le mot-clé `if` , suivi d'une condition. Dans notre cas, la condition vérifie si oui ou non la variable `nombre` a une valeur inférieure à 5. Nous ajoutons le bloc de code à exécuter si la condition est vérifiée immédiatement après la condition entre des accolades. Les blocs de code associés à une condition dans une expression `if` sont parfois appelés des *branches*, exactement comme les branches dans les expressions `match` que nous avons vu dans la section “Comparer le nombre saisi au nombre secret” du chapitre 2.

Éventuellement, vous pouvez aussi ajouter une expression `else` , ce que nous avons fait ici, pour préciser un bloc alternatif de code qui sera exécuté dans le cas où la condition est fausse (elle n'est pas vérifiée). Si vous ne renseignez pas d'expression `else` et que la condition n'est pas vérifiée, le programme va simplement sauter le bloc de `if` et passer au prochain morceau de code.

Essayez d'exécuter ce code ; vous verrez ceci :

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
La condition est vérifiée
```

Essayons de changer la valeur de `nombre` pour une valeur qui rend la condition non vérifiée pour voir ce qui se passe :

```
    let nombre = 7;
```

Exécutez à nouveau le programme, et regardez le résultat :

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
La condition n'est pas vérifiée
```

Il est aussi intéressant de noter que la condition dans ce code *doit* être un `bool` . Si la condition n'est pas un `bool` , nous aurons une erreur. Par exemple, essayez d'exécuter le code suivant :

Fichier : src/main.rs

```
fn main() {
    let nombre = 3;

    if nombre {
        println!("Le nombre était trois");
    }
}
```

La condition `if` vaut `3` cette fois, et Rust lève une erreur :

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if nombre {
  |        ^^^^^^ expected bool, found integer

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` due to previous error
```

Cette erreur explique que Rust attendait un `bool` mais a obtenu un entier *(integer)*. Contrairement à des langages comme Ruby et JavaScript, Rust ne va pas essayer de convertir automatiquement les types non booléens en booléens. Vous devez être précis et toujours fournir un booléen à la condition d'un `if`. Si nous voulons que le bloc de code du `if` soit exécuté quand le nombre est différent de `0`, par exemple, nous pouvons changer l'expression `if` par la suivante :

Fichier: src/main.rs

```
fn main() {
    let nombre = 3;

    if nombre != 0 {
        println!("Le nombre valait autre chose que zéro");
    }
}
```

Exécuter ce code va bien afficher `Le nombre valait autre chose que zéro`.

Gérer plusieurs conditions avec `else if`

Vous pouvez utiliser plusieurs conditions en combinant `if` et `else` dans une expression `else if`. Par exemple :

Fichier : src/main.rs

```
fn main() {
    let nombre = 6;

    if nombre % 4 == 0 {
        println!("Le nombre est divisible par 4");
    } else if nombre % 3 == 0 {
        println!("Le nombre est divisible par 3");
    } else if nombre % 2 == 0 {
        println!("Le nombre est divisible par 2");
    } else {
        println!("Le nombre n'est pas divisible par 4, 3 ou 2");
    }
}
```

Ce programme peut choisir entre quatre chemins différents. Après l'avoir exécuté, vous devriez voir le résultat suivant :

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
Le nombre est divisible par 3
```

Quand ce programme s'exécute, il vérifie chaque expression `if` à tour de rôle et exécute le premier bloc dont la condition est vérifiée. Notez que même si 6 est divisible par 2, nous ne voyons pas le message `Le nombre est divisible par 2`, ni le message `Le nombre n'est pas divisible par 4, 3 ou 2` du bloc `else`. C'est parce que Rust n'exécute que le bloc de la première condition vérifiée, et dès lors qu'il en a trouvé une, il ne va pas chercher à vérifier les suivantes.

Utiliser trop d'expressions `else if` peut encombrer votre code, donc si vous en avez plus d'une, vous devriez envisager de remanier votre code. Le chapitre 6 présente une construction puissante appelée `match` pour de tels cas.

Utiliser `if` dans une instruction `let`

Comme `if` est une expression, nous pouvons l'utiliser à droite d'une instruction `let` pour assigner le résultat à une variable, comme dans l'encart 3-2.

Fichier : src/main.rs

```
fn main() {
    let condition = true;
    let nombre = if condition { 5 } else { 6 };

    println!("La valeur du nombre est : {}", nombre);
}
```

Encart 3-2 : assigner le résultat d'une expression `if` à une variable

La variable `nombre` va avoir la valeur du résultat de l'expression `if`. Exécutez ce code pour découvrir ce qui va se passer :

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/branches`
La valeur du nombre est : 5
```

Souvenez-vous que les blocs de code s'exécutent jusqu'à la dernière expression qu'ils contiennent, et que les nombres tout seuls sont aussi des expressions. Dans notre cas, la

valeur de toute l'expression `if` dépend de quel bloc de code elle va exécuter. Cela veut dire que chaque valeur qui peut être le résultat de chaque branche du `if` doivent être du même type ; dans l'encart 3-2, les résultats des branches `if` et `else` sont tous deux des entiers `i32` . Si les types ne sont pas identiques, comme dans l'exemple suivant, nous allons obtenir une erreur :

Fichier : src/main.rs

```
fn main() {
    let condition = true;

    let nombre = if condition { 5 } else { "six" };

    println!("La valeur du nombre est : {}", nombre);
}
```

Lorsque nous essayons de compiler ce code, nous obtenons une erreur. Les branches `if` et `else` ont des types de valeurs qui ne sont pas compatibles, et Rust indique exactement où trouver le problème dans le programme :

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: `if` and `else` have incompatible types
 --> src/main.rs:4:44
  |
4 |     let nombre = if condition { 5 } else { "six" };
  |                                 -          ^^^^^ expected integer, found
`&str`
  |                                 |
  |                                 expected because of this

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` due to previous error
```

L'expression dans le bloc `if` donne un entier, et l'expression dans le bloc `else` donne une chaîne de caractères. Ceci ne fonctionne pas car les variables doivent avoir un seul type, et Rust a besoin de savoir de quel type est la variable `nombre` au moment de la compilation. Savoir le type de `nombre` permet au compilateur de vérifier que le type est valable n'importe où nous utilisons `nombre` . Rust ne serait pas capable de faire cela si le type de `nombre` était déterminé uniquement à l'exécution ; car le compilateur deviendrait plus complexe et nous donnerait moins de garanties sur le code s'il devait prendre en compte tous les types hypothétiques pour une variable.

Les répétitions avec les boucles

Il est parfois utile d'exécuter un bloc de code plus d'une seule fois. Dans ce but, Rust propose plusieurs types de *boucles*, qui parcourt le code à l'intérieur du corps de la boucle jusqu'à la fin et recommence immédiatement du début. Pour tester les boucles, créons un nouveau projet appelé *loops*.

Rust a trois types de boucles : `loop`, `while`, et `for`. Essayons chacune d'elles.

Répéter du code avec `loop`

Le mot-clé `loop` demande à Rust d'exécuter un bloc de code encore et encore jusqu'à l'infini ou jusqu'à ce que vous lui demandiez explicitement de s'arrêter.

Par exemple, changez le fichier *src/main.rs* dans votre dossier *loops* comme ceci :

Fichier : src/main.rs

```
fn main() {
    loop {
        println!("À nouveau !");
    }
}
```

Quand nous exécutons ce programme, nous voyons `À nouveau !` s'afficher encore et encore en continu jusqu'à ce qu'on arrête le programme manuellement. La plupart des terminaux utilisent un raccourci clavier, ctrl-c, pour arrêter un programme qui est bloqué dans une boucle infinie. Essayons cela :

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s
     Running `target/debug/loops`
À nouveau !
À nouveau !
À nouveau !
À nouveau !
^CÀ nouveau !
```

Le symbole `^C` représente le moment où vous avez appuyé sur ctrl-c. Vous devriez voir ou non le texte `À nouveau !` après le `^C`, en fonction de là où la boucle en était dans votre code quand elle a reçu le signal d'arrêt.

Heureusement, Rust fournit aussi un autre moyen de sortir d'une boucle en utilisant du code. Vous pouvez ajouter le mot-clé `break` à l'intérieur de la boucle pour demander au programme d'arrêter la boucle. Souvenez-vous que nous avions fait ceci dans le jeu de devinettes, dans la section “Arrêter le programme après avoir gagné” du chapitre 2 afin de

quitter le programme quand l'utilisateur gagne le jeu en devinant le bon nombre.

Nous avons également `continue` dans le jeu du plus ou du moins, qui dans une boucle demande au programme de sauter le code restant dans cette iteration de la boucle et passer directement à la prochaine itération.

Si vous avez des boucles imbriquées dans d'autres boucles, `break` et `continue` s'appliquent uniquement à la boucle au plus bas niveau. Si vous en avez besoin, vous pouvez associer une *etiquette de boucle* à une boucle que nous pouvons ensuite utiliser en association avec `break` ou `continue` pour préciser que ces mot-clés s'appliquent sur la boucle correspondant à l'étiquette plutôt qu'à la boucle la plus proche possible. Voici un exemple avec deux boucles imbriquées :

```
fn main() {
    let mut compteur = 0;
    'increment: loop {
        println!("compteur = {}", compteur);
        let mut restant = 10;

        loop {
            println!("restant = {}", restant);
            if restant == 9 {
                break;
            }
            if compteur == 2 {
                break 'increment;
            }
            restant -= 1;
        }

        compteur += 1;
    }
    println!("Fin du compteur = {}", compteur);
}
```

La boucle la plus à l'extérieur a l'étiquette `increment`, et elle va incrémenter de 0 à 2. La boucle à l'intérieur n'a pas d'étiquette et va décrementer de 10 à 9. Le premier `break` qui ne précise pas d'étiquette va arrêter uniquement la boucle interne. L'instruction `break 'increment;` va arrêter la boucle la plus à l'extérieur. Ce code va afficher :

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.58s
     Running `target/debug/loops`
compteur = 0
restant = 10
restant = 9
compteur = 1
restant = 10
restant = 9
compteur = 2
restant = 10
Fin du compteur = 2
```

Retourner des valeurs d'une boucle

L'une des utilisations d'une boucle `loop` est de réessayer une opération qui peut échouer, comme vérifier si une tâche a terminé son travail. Vous aurez aussi peut-être besoin de passer le résultat de l'opération au reste de votre code à l'extérieur de cette boucle. Pour ce faire, vous pouvez ajouter la valeur que vous voulez retourner après l'expression `break` que vous utilisez pour stopper la boucle ; cette valeur sera retournée à l'extérieur de la boucle pour que vous puissiez l'utiliser, comme ci-dessous :

```
fn main() {
    let mut compteur = 0;

    let resultat = loop {
        compteur += 1;

        if compteur == 10 {
            break compteur * 2;
        }
    };

    println!("Le résultat est {}", resultat);
}
```

Avant la boucle, nous déclarons une variable avec le nom `compteur` et nous l'initialisons à `0`. Ensuite, nous déclarons une variable `resultat` pour stocker la valeur retournée de la boucle. À chaque itération de la boucle, nous ajoutons `1` à la variable `compteur`, et ensuite nous vérifions si le compteur est égal à `10`. Lorsque c'est le cas, nous utilisons le mot-clé `break` avec la valeur `compteur * 2`. Après la boucle, nous utilisons un point-virgule pour terminer l'instruction qui assigne la valeur à `resultat`. Enfin, nous affichons la valeur de `resultat`, qui est 20 dans ce cas-ci.

Les boucles conditionnelles avec `while`

Un programme a souvent besoin d'évaluer une condition dans une boucle. Tant que la condition est vraie, la boucle tourne. Quand la condition arrête d'être vraie, le programme appelle `break`, ce qui arrête la boucle. Il est possible d'implémenter un comportement comme celui-ci en combinant `loop`, `if`, `else` et `break` ; vous pouvez essayer de le faire, si vous voulez. Cependant, cette utilisation est si fréquente que Rust a une construction pour cela, intégrée dans le langage, qui s'appelle une boucle `while`. Dans l'encart 3-3, nous utilisons `while` pour boucler trois fois, en décrémentant à chaque fois, et ensuite, après la boucle, il va afficher un message et se fermer.

Fichier : src/main.rs

```
fn main() {
    let mut nombre = 3;

    while nombre != 0 {
        println!("{} !", nombre);

        nombre -= 1;
    }

    println!("DÉCOLLAGE !!!");
}
```

Encart 3-3: utiliser une boucle `while` pour exécuter du code tant qu'une condition est vraie

Cette construction élimine beaucoup d'imbrications qui seraient nécessaires si vous utilisiez `loop`, `if`, `else` et `break`, et c'est aussi plus clair. Tant que la condition est vraie, le code est exécuté ; sinon, il quitte la boucle.

Boucler dans une collection avec `for`

Vous pouvez choisir d'utiliser la construction `while` pour itérer sur les éléments d'une collection, comme les tableaux. Par exemple, la boucle dans l'encart 3-4 affiche chaque élément présent dans le tableau `a`.

Fichier : src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut indice = 0;

    while indice < 5 {
        println!("La valeur est : {}", a[indice]);

        indice += 1;
    }
}
```

Encart 3-4 : itération sur les éléments d'une collection en utilisant une boucle `while`

Ici, le code parcourt le tableau élément par élément. Il commence à l'indice `0`, et ensuite boucle jusqu'à ce qu'il atteigne l'indice final du tableau (ce qui correspond au moment où la condition `index < 5` n'est plus vraie). Exécuter ce code va afficher chaque élément du tableau :

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32s
     Running `target/debug/loops`
La valeur est : 10
La valeur est : 20
La valeur est : 30
La valeur est : 40
La valeur est : 50
```

Les cinq valeurs du tableau s'affichent toutes dans le terminal, comme attendu. Même si `indice` va atteindre la valeur `5` à un moment, la boucle arrêtera de s'exécuter avant d'essayer de récupérer une sixième valeur du tableau.

Cependant, cette approche pousse à l'erreur ; nous pourrions faire paniquer le programme si la valeur de l'indice est trop grand ou que la condition du test est incorrecte. Par exemple, si vous changez la définition du tableau `a` pour avoir quatre éléments, mais que nous oublions de modifier la condition dans `while indice < 4`, le code paniquera. De plus, c'est lent, car le compilateur ajoute du code pour effectuer à l'exécution la vérification que l'indice est compris dans les limites du tableau, et cela à chaque itération de la boucle.

Pour une alternative plus concise, vous pouvez utiliser une boucle `for` et exécuter du code pour chaque élément dans une collection. Une boucle `for` s'utilise comme dans le code de l'encart 3-5.

Fichier : src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("La valeur est : {}", element);
    }
}
```

Encart 3-5 : itérer sur chaque élément d'une collection en utilisant une boucle `for`

Lorsque nous exécutons ce code, nous obtenons les mêmes messages que dans l'encart 3-4. Mais ce qui est plus important, c'est que nous avons amélioré la sécurité de notre code et éliminé le risque de bogues qui pourraient survenir si on dépassait la fin du tableau, ou si on n'allait pas jusqu'au bout et qu'on ratait quelques éléments.

En utilisant la boucle `for`, vous n'aurez pas à vous rappeler de changer le code si vous changez le nombre de valeurs dans le tableau, comme vous devriez le faire dans la méthode utilisée dans l'encart 3-4.

La sécurité et la concision de la boucle `for` en font la construction de boucle la plus utilisée avec Rust. Même dans des situations dans lesquelles vous voudriez exécuter du code plusieurs fois, comme l'exemple du décompte qui utilisait une boucle `while` dans l'encart 3-3, la plupart des Rustacés utiliseraient une boucle `for`. Il faut pour cela utiliser un intervalle `Range`, fourni par la bibliothèque standard pour générer dans l'ordre tous les nombres compris entre un certain nombre et un autre nombre.

Voici ce que le décompte aurait donné en utilisant une boucle `for` et une autre méthode que nous n'avons pas encore vue, `rev`, qui inverse l'intervalle :

Fichier : src/main.rs

```
fn main() {
    for nombre in (1..4).rev() {
        println!("{} !", nombre);
    }
    println!("DÉCOLLAGE !!!");
}
```

Ce code est un peu plus sympa, non ?

Résumé

Vous y êtes arrivé ! C'était un chapitre important : vous avez appris les variables, les types

scalaires et composés, les fonctions, les commentaires, les expressions `if`, et les boucles ! Pour pratiquer un peu les concepts abordés dans ce chapitre, voici quelques programmes que vous pouvez essayer de créer :

- Convertir des températures entre les degrés Fahrenheit et Celsius.
- Générer le *n*-ième nombre de Fibonacci.
- Afficher les paroles de la chanson de Noël *The Twelve Days of Christmas* en profitant de l'aspect répétitif de la chanson.

Quand vous serez prêt à aller plus loin, nous aborderons une notion de Rust qui n'existe *pas* dans les autres langages de programmation : la possession *(ownership)*.

Comprendre la possession

La possession (*ownership*) est la fonctionnalité la plus remarquable de Rust, et a des implications en profondeur dans l'ensemble du langage. Elle permet à Rust de garantir la sécurité de la mémoire sans avoir besoin d'un ramasse-miettes (*garbage collector*), donc il est important de comprendre comment la possession fonctionne. Dans ce chapitre, nous aborderons la possession, ainsi que d'autres fonctionnalités associées : l'emprunt, les *slices* et la façon dont Rust agence les données en mémoire.

Qu'est-ce que la possession ?

La possession est un jeu de règles qui gouvernent la gestion de la mémoire par un programme Rust. Tous les programmes doivent gérer la façon dont ils utilisent la mémoire lorsqu'ils s'exécutent. Certains langages ont un ramasse-miettes qui scrute constamment la mémoire qui n'est plus utilisée pendant qu'il s'exécute ; dans d'autres langages, le développeur doit explicitement allouer et libérer la mémoire. Rust adopte une troisième approche : la mémoire est gérée avec un système de possession qui repose sur un jeu de règles que le compilateur vérifie au moment de la compilation. Si une de ces règles a été enfreinte, le programme ne sera pas compilé. Aucune des fonctionnalités de la possession ne ralentit votre programme à l'exécution.

Comme la possession est un nouveau principe pour de nombreux développeurs, cela prend un certain temps pour s'y familiariser. La bonne nouvelle est que plus vous devenez expérimenté avec Rust et ses règles de possession, plus vous développerez naturellement et facilement du code sûr et efficace. Gardez bien cela à l'esprit !

Lorsque vous comprendrez la possession, vous aurez des bases solides pour comprendre les fonctionnalités qui font la particularité de Rust. Dans ce chapitre, vous allez apprendre la possession en pratiquant avec plusieurs exemples qui se concentrent sur une structure de données très courante : les chaînes de caractères.

La pile et le tas

De nombreux langages ne nécessitent pas de se préoccuper de la pile (*stack*) et du tas (*heap*). Mais dans un langage de programmation système comme Rust, le fait qu'une donnée soit sur la pile ou sur le tas a une influence sur le comportement du langage et explique pourquoi nous devons faire certains choix. Nous décrirons plus loin dans ce chapitre comment la possession fonctionne vis-à-vis de la pile et du tas, voici donc une brève explication au préalable.

La pile et le tas sont tous les deux des emplacements de la mémoire à disposition de votre code lors de son exécution, mais sont organisés de façon différente. La pile enregistre les valeurs dans l'ordre qu'elle les reçoit et enlève les valeurs dans l'autre sens. C'est ce que l'on appelle le principe de *dernier entré, premier sorti*. C'est comme une pile d'assiettes : quand vous ajoutez des nouvelles assiettes, vous les déposez sur le dessus de la pile, et quand vous avez besoin d'une assiette, vous en prenez une sur le dessus. Ajouter ou enlever des assiettes au milieu ou en bas ne serait pas aussi efficace ! Ajouter une donnée sur la pile se dit *empiler* et en retirer une se dit *dépiler*. Toutes donnée stockée dans la pile doit avoir une taille connue et fixe. Les données

avec une taille inconnue au moment de la compilation ou une taille qui peut changer doivent plutôt être stockées sur le tas.

Le tas est moins bien organisé : lorsque vous ajoutez des données sur le tas, vous demandez une certaine quantité d'espace mémoire. Le gestionnaire de mémoire va trouver un emplacement dans le tas qui est suffisamment grand, va le marquer comme étant en cours d'utilisation, et va retourner un *pointeur*, qui est l'adresse de cet emplacement. Cette procédure est appelée *allocation sur le tas*, ce qu'on abrège parfois en *allocation* tout court. L'ajout de valeurs sur la pile n'est pas considéré comme une allocation. Comme le pointeur vers le tas a une taille connue et fixe, on peut stocker ce pointeur sur la pile, mais quand on veut la vraie donnée, il faut suivre le pointeur.

C'est comme si vous vouliez manger au restaurant. Quand vous entrez, vous indiquez le nombre de personnes dans votre groupe, et le personnel trouve une table vide qui peut recevoir tout le monde, et vous y conduit. Si quelqu'un dans votre groupe arrive en retard, il peut leur demander où vous êtes assis pour vous rejoindre.

Empiler sur la pile est plus rapide qu'allouer sur le tas car le gestionnaire ne va jamais avoir besoin de chercher un emplacement pour y stocker les nouvelles données ; il le fait toujours au sommet de la pile. En comparaison, allouer de la place sur le tas demande plus de travail, car le gestionnaire doit d'abord trouver un espace assez grand pour stocker les données et mettre à jour son suivi pour préparer la prochaine allocation.

Accéder à des données dans le tas est plus lent que d'accéder aux données sur la pile car nous devons suivre un pointeur pour les obtenir. Les processeurs modernes sont plus rapides s'ils se déplacent moins dans la mémoire. Pour continuer avec notre analogie, imaginez un serveur dans un restaurant qui prend les commandes de nombreuses tables. C'est plus efficace de récupérer toutes les commandes à une seule table avant de passer à la table suivante. Prendre une commande à la table A, puis prendre une commande à la table B, puis ensuite une autre à la table A, puis une autre à la table B serait un processus bien plus lent. De la même manière, un processeur sera plus efficace dans sa tâche s'il travaille sur des données qui sont proches les unes des autres (comme c'est le cas sur la pile) plutôt que si elles sont plus éloignées (comme cela peut être le cas sur le tas). Allouer une grande quantité de mémoire sur le tas peut aussi prendre beaucoup de temps.

Quand notre code utilise une fonction, les valeurs passées à la fonction (incluant, potentiellement, des pointeurs de données sur le tas) et les variables locales à la fonction sont déposées sur la pile. Quand l'utilisation de la fonction est terminée, ces données sont retirées de la pile.

La possession nous aide à ne pas nous préoccuper de faire attention à quelles parties

> du code utilisent quelles données sur le tas, de minimiser la quantité de données en double sur le tas, ou encore de veiller à libérer les données inutilisées sur le tas pour que nous ne soyons pas à court d'espace. Quand vous aurez compris la possession, vous n'aurez plus besoin de vous préoccuper de la pile et du tas très souvent, mais savoir que le but principal de la possession est de gérer les données du tas peut vous aider à comprendre pourquoi elle fonctionne de cette manière.

Les règles de la possession

Tout d'abord, définissons les règles de la possession. Gardez à l'esprit ces règles pendant que nous travaillons sur des exemples qui les illustrent :

- Chaque valeur en Rust a une variable qui s'appelle son *propriétaire*.
- Il ne peut y avoir qu'un seul propriétaire à la fois.
- Quand le propriétaire sortira de la portée, la valeur sera supprimée.

Portée de la variable

Maintenant que nous avons vu la syntaxe Rust de base, nous n'allons plus ajouter tout le code du style `fn main() {` dans les exemples, donc si vous voulez reproduire les exemples, assurez-vous de les placer manuellement dans une fonction `main`. Par conséquent, nos exemples seront plus concis, nous permettant de nous concentrer sur les détails de la situation plutôt que sur du code normalisé.

Pour le premier exemple de possession, nous allons analyser la *portée* de certaines variables. Une portée est une zone dans un programme dans laquelle un élément est en vigueur. Admettons la variable suivante :

```
let s = "hello";
```

La variable `s` fait référence à un littéral de chaîne de caractères, où la valeur de la chaîne est codée en dur dans notre programme. La variable est en vigueur à partir du moment où elle est déclarée jusqu'à la fin de la *portée* actuelle. L'encart 4-1 nous présente un programme avec des commentaires pour indiquer quand la variable `s` est en vigueur :

```
    {                      // s n'est pas en vigueur ici, elle n'est pas encore
déclarée
        let s = "hello"; // s est en vigueur à partir de ce point

        // on fait des choses avec s ici
    }                      // cette portée est maintenant terminée, et s n'est
plus en vigueur
```

Encart 4-1 : Une variable et la portée dans laquelle elle est en vigueur.

Autrement dit, il y a ici deux étapes importantes :

- Quand `s` rentre *dans la portée*, elle est en vigueur.
- Cela reste ainsi jusqu'à ce qu'elle *sorte de la portée*.

Pour le moment, la relation entre les portées et les conditions pour lesquelles les variables sont en vigueur sont similaires à d'autres langages de programmation. Maintenant, nous allons aller plus loin en y ajoutant le type `String` .

Le type `String`

Pour illustrer les règles de la possession, nous avons besoin d'un type de donnée qui est plus complexe que ceux que nous avons rencontrés dans la section “Types de données” du chapitre 3. Les types que nous avons vus précédemment ont tous une taille connue et peuvent être stockés sur la pile ainsi que retirés de la pile lorsque la portée n'en a plus besoin, et peuvent aussi être rapidement et facilement copiés afin de constituer une nouvelle instance indépendante si une autre partie du code a besoin d'utiliser la même valeur dans une portée différente. Mais nous voulons expérimenter le stockage de données sur le tas et découvrir comment Rust sait quand il doit nettoyer ces données, et le type `String` est un bon exemple.

Nous allons nous concentrer sur les caractéristiques de `String` qui sont liées à la possession. Ces aspects s'appliquent également à d'autres types de données complexes, qu'ils soient fournis par la bibliothèque standard ou qu'ils soient créés par vous. Nous verrons `String` plus en détail dans le chapitre 8.

Nous avons déjà vu les littéraux de chaînes de caractères, quand une valeur de chaîne est codée en dur dans notre programme. Les littéraux de chaînes sont pratiques, mais ils ne conviennent pas toujours à tous les cas où on veut utiliser du texte. Une des raisons est qu'ils sont immuables. Une autre raison est qu'on ne connaît pas forcément le contenu des chaînes de caractères quand nous écrivons notre code : par exemple, comment faire si nous voulons récupérer du texte saisi par l'utilisateur et l'enregistrer ? Pour ces cas-ci, Rust a un

second type de chaîne de caractères, `String`. Ce type gère ses données sur le tas et est ainsi capable de stocker une quantité de texte qui nous est inconnue au moment de la compilation. Vous pouvez créer une `String` à partir d'un littéral de chaîne de caractères en utilisant la fonction `from`, comme ceci :

```
let s = String::from("hello");
```

L'opérateur double deux-points `::` nous permet d'appeler cette fonction spécifique dans l'espace de nom du type `String` plutôt que d'utiliser un nom comme `string_from`. Nous verrons cette syntaxe plus en détail dans la section "Syntaxe de méthode" du chapitre 5 et lorsque nous aborderons les espaces de noms dans la section "Les chemins pour désigner un élément dans l'arborescence de module" du chapitre 7.

Ce type de chaîne de caractères *peut* être mutable :

```
    let mut s = String::from("hello");

    s.push_str(", world!"); // push_str() ajoute un littéral de chaîne dans une
String

    println!("{}", s); // Cela va afficher `hello, world!`
```

Donc, quelle est la différence ici ? Pourquoi `String` peut être mutable, mais pourquoi les littéraux de chaînes ne peuvent pas l'être ? La différence se trouve dans la façon dont ces deux types travaillent avec la mémoire.

Mémoire et allocation

Dans le cas d'un littéral de chaîne de caractères, nous connaissons le contenu au moment de la compilation donc le texte est codé en dur directement dans l'exécutable final. Voilà pourquoi ces littéraux de chaînes de caractères sont performants et rapides. Mais ces caractéristiques viennent de leur immuabilité. Malheureusement, on ne peut pas accorder une grosse région de mémoire dans le binaire pour chaque morceau de texte qui n'a pas de taille connue au moment de la compilation et dont la taille pourrait changer pendant l'exécution de ce programme.

Avec le type `String`, pour nous permettre d'avoir un texte mutable et qui peut s'agrandir, nous devons allouer une quantité de mémoire sur le tas, inconnue au moment de la compilation, pour stocker le contenu. Cela signifie que :

- La mémoire doit être demandée auprès du gestionnaire de mémoire lors de

l'exécution.
- Nous avons besoin d'un moyen de rendre cette mémoire au gestionnaire lorsque nous aurons fini d'utiliser notre `String`.

Nous nous occupons de ce premier point : quand nous appelons `String::from`, son implémentation demande la mémoire dont elle a besoin. C'est pratiquement toujours ainsi dans la majorité des langages de programmation.

Cependant, le deuxième point est différent. Dans des langages avec un *ramasse-miettes*, le ramasse-miettes surveille et nettoie la mémoire qui n'est plus utilisée, sans que nous n'ayons à nous en préoccuper. Dans la pluspart des langages sans ramasse-miettes, c'est de notre responsabilité d'identifier quand cette mémoire n'est plus utilisée et d'appeler du code pour explicitement la libérer, comme nous l'avons fait pour la demander auparavant. Historiquement, faire ceci correctement a toujours été une difficulté pour les développeurs. Si nous oublions de le faire, nous allons gaspiller de la mémoire. Si nous le faisons trop tôt, nous allons avoir une variable invalide. Si nous le faisons deux fois, cela produit aussi un bogue. Nous devons associer exactement un `allocate` avec exactement un `free`.

Rust prend un chemin différent : la mémoire est automatiquement libérée dès que la variable qui la possède sort de la portée. Voici une version de notre exemple de portée de l'encart 4-1 qui utilise une `String` plutôt qu'un littéral de chaîne de caractères :

```
    {
        let s = String::from("hello"); // s est en vigueur à partir de ce point

        // on fait des choses avec s ici
    }                                  // cette portée est désormais terminée,
et s
                                       // n'est plus en vigueur maintenant
```

Il y a un moment naturel où nous devons rendre la mémoire de notre `String` au gestionnaire : quand `s` sort de la portée. Quand une variable sort de la portée, Rust appelle une fonction spéciale pour nous. Cette fonction s'appelle `drop`, et c'est dans celle-ci que l'auteur de `String` a pu mettre le code pour libérer la mémoire. Rust appelle automatiquement `drop` à l'accolade fermante `}`.

> Remarque : en C++, cette façon de libérer des ressources à la fin de la durée de vie d'un élément est parfois appelée *l'acquisition d'une ressource est une initialisation (RAII)*. La fonction `drop` de Rust vous sera familière si vous avez déjà utilisé des techniques de RAII.

Cette façon de faire a un impact profond sur la façon dont le code Rust est écrit. Cela peut

sembler simple dans notre cas, mais le comportement du code peut être surprenant dans des situations plus compliquées où nous voulons avoir plusieurs variables utilisant des données que nous avons affectées sur le tas. Examinons une de ces situations dès à présent.

Les interactions entre les variables et les données : le déplacement

Plusieurs variables peuvent interagir avec les mêmes données de différentes manières en Rust. Regardons un exemple avec un entier dans l'encart 4-2 :

```
let x = 5;
let y = x;
```

Encart 4-2 : Assigner l'entier de la variable `x` à `y`

Nous pouvons probablement deviner ce que ce code fait : “Assigner la valeur `5` à `x` ; ensuite faire une copie de cette valeur de `x` et l'assigner à `y`.” Nous avons maintenant deux variables, `x` et `y`, et chacune vaut `5`. C'est effectivement ce qui se passe, car les entiers sont des valeurs simples avec une taille connue et fixée, et ces deux valeurs `5` sont stockées sur la pile.

Maintenant, essayons une nouvelle version avec `String` :

```
let s1 = String::from("hello");
let s2 = s1;
```

Cela ressemble beaucoup, donc nous allons supposer que cela fonctionne pareil que précédemment : ainsi, la seconde ligne va faire une copie de la valeur de `s1` et l'assigner à `s2`. Mais ce n'est pas tout à fait ce qu'il se passe.

Regardons l'illustration 4-1 pour découvrir ce qui arrive à `String` sous le capot. Une `String` est constituée de trois éléments, présents sur la gauche : un pointeur vers la mémoire qui contient le contenu de la chaîne de caractères, une taille, et une capacité. Ce groupe de données est stocké sur la pile. À droite, nous avons la mémoire sur le tas qui contient les données.

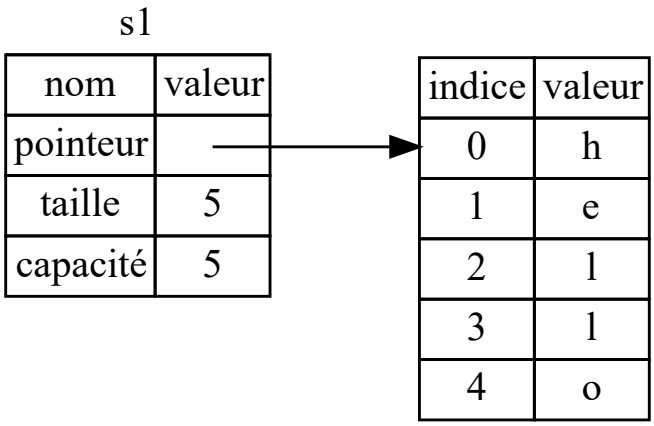


Illustration 4-1 : Représentation en mémoire d'une `String` qui contient la valeur `"hello"` assignée à `s1` .

La taille est la quantité de mémoire, en octets, que le contenu de la `String` utilise actuellement. La capacité est la quantité totale de mémoire, en octets, que la `String` a reçue du gestionnaire. La différence entre la taille et la capacité est importante, mais pas pour notre exemple, donc pour l'instant, ce n'est pas grave d'ignorer la capacité.

Quand nous assignons `s1` à `s2` , les données de la `String` sont copiées, ce qui veut dire que nous copions le pointeur, la taille et la capacité qui sont stockés sur la pile. Nous ne copions pas les données stockées sur le tas auxquelles le pointeur se réfère. Autrement dit, la représentation des données dans la mémoire ressemble à l'illustration 4-2.

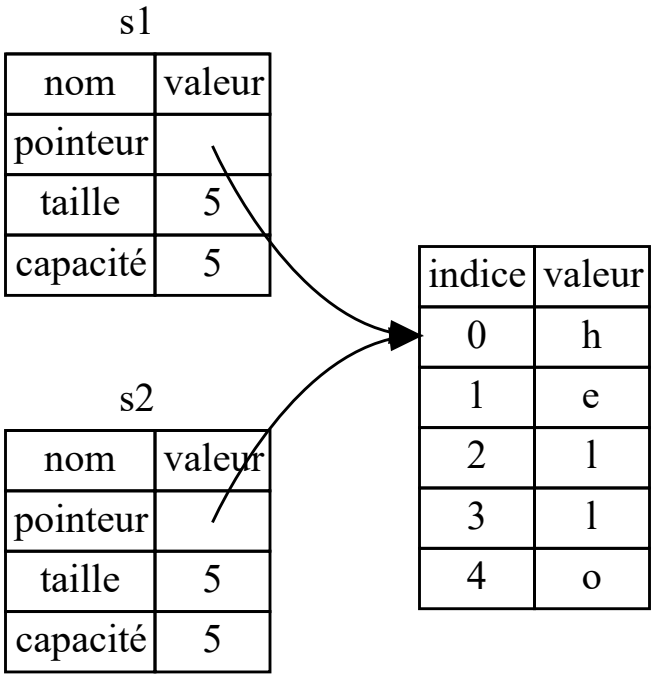


Illustration 4-2 : Représentation en mémoire de la variable `s2` qui a une copie du pointeur,

de la taille et de la capacité de `s1`

Cette représentation *n'est pas* comme l'illustration 4-3, qui représenterait la mémoire si Rust avait aussi copié les données sur le tas. Si Rust faisait ceci, l'opération `s2 = s1` pourrait potentiellement être très coûteuse en termes de performances d'exécution si les données sur le tas étaient volumineuses.

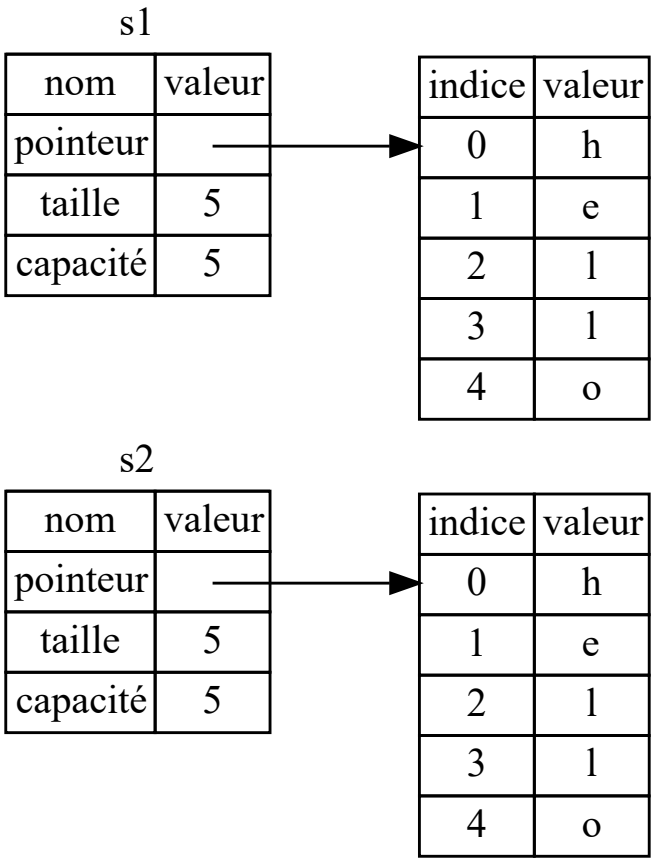


Illustration 4-3 : Une autre possibilité de ce que pourrait faire `s2 = s1` si Rust copiait aussi les données du tas

Précédemment, nous avons dit que quand une variable sortait de la portée, Rust appelait automatiquement la fonction `drop` et nettoyait la mémoire sur le tas allouée pour cette variable. Mais l'illustration 4-2 montre que les deux pointeurs de données pointeraient au même endroit. C'est un problème : quand `s2` et `s1` sortent de la portée, elles vont essayer toutes les deux de libérer la même mémoire. C'est ce qu'on appelle une erreur de *double libération* et c'est un des bogues de sécurité de mémoire que nous avons mentionnés précédemment. Libérer la mémoire deux fois peut mener à des corruptions de mémoire, ce qui peut potentiellement mener à des vulnérabilités de sécurité.

Pour garantir la sécurité de la mémoire, après la ligne `let s2 = s1`, Rust considère que `s1` n'est plus en vigueur. Par conséquent, Rust n'a pas besoin de libérer quoi que ce soit

lorsque `s1` sort de la portée. Regardez ce qu'il se passe quand vous essayez d'utiliser `s1` après que `s2` est créé, cela ne va pas fonctionner :

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

Vous allez avoir une erreur comme celle-ci, car Rust vous défend d'utiliser la référence qui n'est plus en vigueur :

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not
implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` due to previous error
```

Si vous avez déjà entendu parler de *copie superficielle* et de *copie profonde* en utilisant d'autres langages, l'idée de copier le pointeur, la taille et la capacité sans copier les données peut vous faire penser à de la copie superficielle. Mais comme Rust neutralise aussi la première variable, au lieu d'appeler cela une copie superficielle, on appelle cela un *déplacement*. Ici, nous pourrions dire que `s1` a été *déplacé* dans `s2` . Donc ce qui se passe réellement est décrit par l'illustration 4-4.

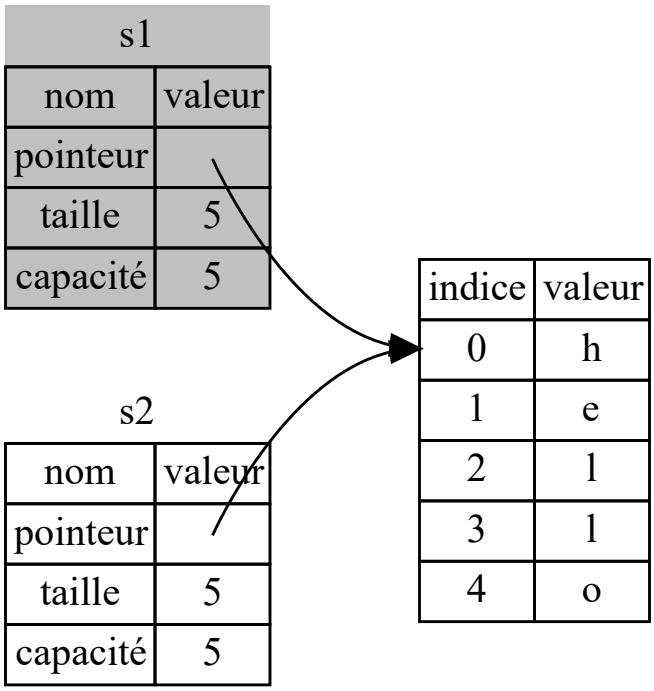


Illustration 4-4 : Représentation de la mémoire après que `s1` a été neutralisée

Cela résout notre problème ! Avec seulement `s2` en vigueur, quand elle sortira de la portée, elle seule va libérer la mémoire, et c'est tout.

De plus, cela signifie qu'il y a eu un choix de conception : Rust ne va jamais créer automatiquement de copie “profonde” de vos données. Par conséquent, toute copie *automatique* peut être considérée comme peu coûteuse en termes de performances d'exécution.

Les interactions entre les variables et les données : le clonage

Si nous *voulons* faire une copie profonde des données sur le tas d'une `String`, et pas seulement des données sur la pile, nous pouvons utiliser une méthode commune qui s'appelle `clone`. Nous aborderons la syntaxe des méthodes au chapitre 5, mais comme les méthodes sont des outils courants dans de nombreux langages, vous les avez probablement utilisées auparavant.

Voici un exemple d'utilisation de la méthode `clone` :

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

Cela fonctionne très bien et c'est ainsi que vous pouvez reproduire le comportement décrit

dans l'illustration 4-3, où les données du tas sont copiées.

Quand vous voyez un appel à `clone`, vous savez que du code arbitraire est exécuté et que ce code peut être coûteux. C'est un indicateur visuel qu'il se passe quelque chose de différent.

Données uniquement sur la pile : la copie

Il y a un autre détail dont on n'a pas encore parlé. Le code suivant utilise des entiers - on en a vu une partie dans l'encart 4-2 - il fonctionne et est correct :

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

Mais ce code semble contredire ce que nous venons d'apprendre : nous n'avons pas appelé `clone`, mais `x` est toujours en vigueur et n'a pas été déplacé dans `y`.

La raison est que les types comme les entiers ont une taille connue au moment de la compilation et sont entièrement stockés sur la pile, donc la copie des vraies valeurs est rapide à faire. Cela signifie qu'il n'y a pas de raison que nous voudrions neutraliser `x` après avoir créé la variable `y`. En d'autres termes, il n'y a pas ici de différence entre la copie superficielle et profonde, donc appeler `clone` ne ferait rien d'autre qu'une copie superficielle classique et on peut s'en passer.

Rust a une annotation spéciale appelée le trait `Copy` que nous pouvons utiliser sur des types comme les entiers qui sont stockés sur la pile (nous verrons les traits dans le chapitre 10). Si un type implémente le trait `Copy`, une variable sera toujours en vigueur après avoir été affectée à une autre variable. Rust ne nous autorisera pas à annoter un type avec le trait `Copy` si ce type, ou un de ses éléments, a implémenté le trait `Drop`. Si ce type a besoin que quelque chose de spécial se produise quand la valeur sort de la portée et que nous ajoutons l'annotation `Copy` sur ce type, nous aurons une erreur au moment de la compilation. Pour savoir comment ajouter l'annotation `Copy` sur votre type pour implémenter le trait, référez-vous à l'annexe C sur les traits dérivables.

Donc, quels sont les types qui implémentent le trait `Copy` ? Vous pouvez regarder dans la documentation pour un type donné pour vous en assurer, mais de manière générale, tout groupe de valeur scalaire peut implémenter `Copy`, et tout ce qui ne nécessite pas d'allocation de mémoire ou tout autre forme de ressource qui implémente `Copy`. Voici quelques types qui implémentent `Copy` :

- Tous les types d'entiers, comme `u32`.
- Le type booléen, `bool`, avec les valeurs `true` et `false`.
- Tous les types de flottants, comme `f64`.
- Le type de caractère, `char`.
- Les tuples, mais uniquement s'ils contiennent des types qui implémentent aussi `Copy`. Par exemple, le `(i32, i32)` implémente `Copy`, mais pas `(i32, String)`.

La possession et les fonctions

La syntaxe pour passer une valeur à une fonction est similaire à celle pour assigner une valeur à une variable. Passer une variable à une fonction va la déplacer ou la copier, comme l'assignation. L'encart 4-3 est un exemple avec quelques commentaires qui montrent où les variables rentrent et sortent de la portée :

Fichier : src/main.rs

```
fn main() {
  let s = String::from("hello");  // s rentre dans la portée.

  prendre_possession(s);  // La valeur de s est déplacée dans la fonction…
                          // … et n'est plus en vigueur à partir d'ici

  let x = 5;              // x rentre dans la portée.

  creer_copie(x);         // x va être déplacée dans la fonction,
                          // mais i32 est Copy, donc on peut
                          // utiliser x ensuite.

} // Ici, x sort de la portée, puis ensuite s. Mais puisque la valeur de s a
// été déplacée, il ne se passe rien de spécial.

fn prendre_possession(texte: String) { // texte rentre dans la portée.
  println!("{}", texte);
} // Ici, texte sort de la portée et `drop` est appelé. La mémoire est libérée.

fn creer_copie(entier: i32) { // entier rentre dans la portée.
  println!("{}", entier);
} // Ici, entier sort de la portée. Il ne se passe rien de spécial.
```

Encart 4-3 : Les fonctions avec les possessions et les portées qui sont commentées

Si on essayait d'utiliser `s` après l'appel à `prendre_possession`, Rust déclencherait une erreur à la compilation. Ces vérifications statiques nous protègent des erreurs. Essayez d'ajouter du code au `main` qui utilise `s` et `x` pour découvrir lorsque vous pouvez les utiliser et lorsque les règles de la possession vous empêchent de le faire.

Les valeurs de retour et les portées

Retourner des valeurs peut aussi transférer leur possession. L'encart 4-4 montre un exemple d'une fonction qui retourne une valeur, avec des annotations similaires à celles de l'encart 4-3 :

Fichier : src/main.rs

```
fn main() {
  let s1 = donne_possession();      // donne_possession déplace sa valeur de
                                    // retour dans s1

  let s2 = String::from("hello");   // s2 rentre dans la portée

  let s3 = prend_et_rend(s2);       // s2 est déplacée dans
                                    // prend_et_rend, qui elle aussi
                                    // déplace sa valeur de retour dans s3.
} // Ici, s3 sort de la portée et est éliminée. s2 a été déplacée donc il ne se
  // passe rien. s1 sort aussi de la portée et est éliminée.

fn donne_possession() -> String {       // donne_possession va déplacer sa
                                        // valeur de retour dans la
                                        // fonction qui l'appelle.

  let texte = String::from("yours");    // texte rentre dans la portée.

  texte                                 // texte est retournée et
                                        // est déplacée vers le code qui
                                        // l'appelle.
}

// Cette fonction va prendre une String et en retourne aussi une.
fn prend_et_rend(texte: String) -> String { // texte rentre dans la portée.

  texte  // texte est retournée et déplacée vers le code qui l'appelle.
}
```

Encart 4-4 : Transferts de possession des valeurs de retour

La possession d'une variable suit toujours le même schéma à chaque fois : assigner une valeur à une autre variable la déplace. Quand une variable qui contient des données sur le tas sort de la portée, la valeur sera nettoyée avec `drop` à moins que la possession de cette donnée soit donnée à une autre variable.

Même si cela fonctionne, il est un peu fastidieux de prendre la possession puis ensuite de retourner la possession à chaque fonction. Et qu'est-ce qu'il se passe si nous voulons qu'une fonction utilise une valeur, mais n'en prenne pas possession ? C'est assez pénible que tout ce que nous passons doit être retourné si nous voulons l'utiliser à nouveau, en plus de

toutes les données qui découlent du corps de la fonction que nous voulons aussi récupérer.

Rust nous permet de retourner plusieurs valeurs à l'aide d'un tuple, comme ceci :

Fichier : src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let (s2, taille) = calculer_taille(s1);

    println!("La taille de '{}' est {}.", s2, taille);
}

fn calculer_taille(s: String) -> (String, usize) {
    let taille = s.len(); // len() retourne la taille d'une String.

    (s, taille)
}
```

Encart 4-5 : Retourner la possession des paramètres

Mais c'est trop laborieux et beaucoup de travail pour un principe qui devrait être banal. Heureusement pour nous, Rust a une fonctionnalité pour utiliser une valeur sans avoir à transférer la possession, avec ce qu'on appelle les *références*.

Les références et l'emprunt

La difficulté avec le code du tuple à la fin de la section précédente est que nous avons besoin de retourner la `String` au code appelant pour qu'il puisse continuer à utiliser la `String` après l'appel à `calculer_taille`, car la `String` a été déplacée dans `calculer_taille`. À la place, nous pouvons fournir une référence à la valeur de la String. Une *référence* est comme un pointeur dans le sens où c'est une adresse que nous pouvons suivre pour accéder à la donnée stockée à cette adresse qui est possédée par une autre variable. Mais contrairement aux pointeurs, une référence garantit de pointer vers une valeur en vigueur, d'un type bien déterminé. Voici comment définir et utiliser une fonction `calculer_taille` qui prend une *référence* à un objet en paramètre plutôt que de prendre possession de la valeur :

Fichier : src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let long = calculer_taille(&s1);

    println!("La taille de '{}' est {}.", s1, long);
}

fn calculer_taille(s: &String) -> usize {
    s.len()
}
```

Premièrement, on peut observer que tout le code des *tuples* dans la déclaration des variables et dans la valeur de retour de la fonction a été enlevé. Deuxièmement, remarquez que nous passons `&s1` à `calculer_taille`, et que dans sa définition, nous utilisons `&String` plutôt que `String`. Ces esperluettes représentent les *références*, et elles permettent de vous référer à une valeur sans en prendre possession. L'illustration 4-5 illustre ce concept.

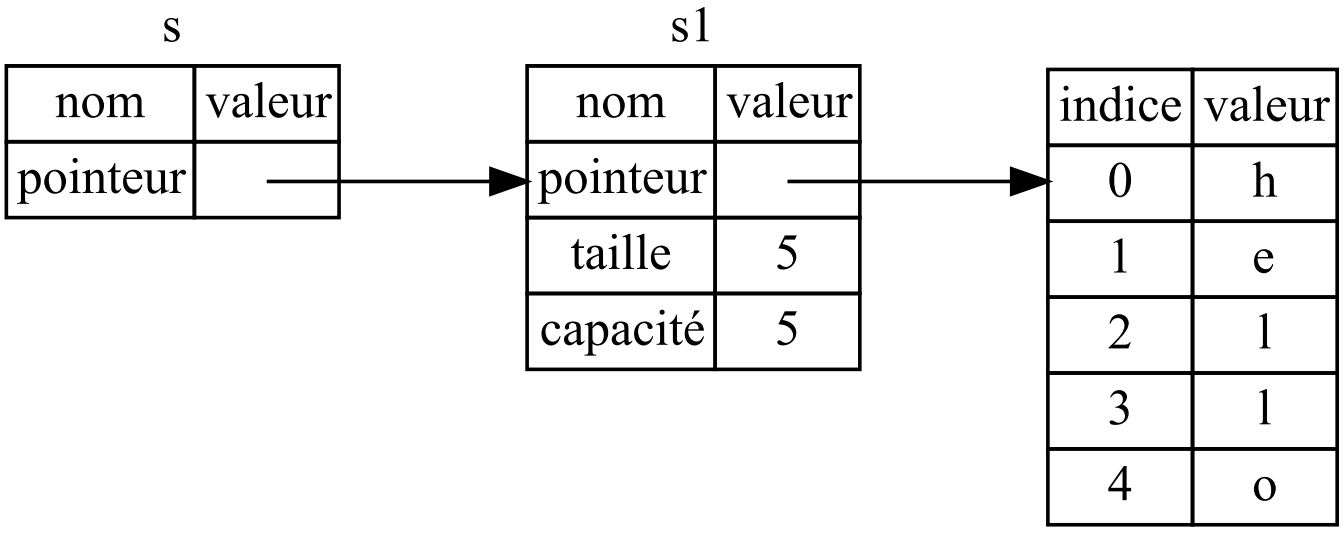


Illustration 4-5 : Un schéma de la `&String s` qui pointe vers la `String s1`

Remarque : l'opposé de la création de références avec `&` est le *déréférencement*, qui s'effectue avec l'opérateur de déréférencement, `*`. Nous allons voir quelques utilisations de l'opérateur de déréférencement dans le chapitre 8 et nous aborderons les détails du déréférencement dans le chapitre 15.

Regardons de plus près l'appel à la fonction :

```
let s1 = String::from("hello");

let long = calculer_taille(&s1);
```

La syntaxe `&s1` nous permet de créer une référence qui se *réfère* à la valeur de `s1` mais n'en prend pas possession. Et comme elle ne la possède pas, la valeur vers laquelle elle pointe ne sera pas libérée quand cette référence ne sera plus utilisée.

De la même manière, la signature de la fonction utilise `&` pour indiquer que le type du paramètre `s` est une référence. Ajoutons quelques commentaires explicatifs :

```
fn calculer_taille(s: &String) -> usize { // s est une référence à une String
  s.len()
} // Ici, s sort de la portée. Mais comme elle ne prend pas possession de ce
  // à quoi elle fait référence, il ne se passe rien.
```

La portée dans laquelle la variable `s` est en vigueur est la même que toute portée d'un paramètre de fonction, mais la valeur pointée par la référence n'est pas libérée quand `s`

n'est plus utilisé, car `s` n'en prends pas possession. Lorsque les fonctions ont des références en paramètres au lieu des valeurs réelles, nous n'avons pas besoin de retourner les valeurs pour les rendre, car nous n'en avons jamais pris possession.

Nous appelons *l'emprunt* l'action de créer une référence. Comme dans la vie réelle, quand un objet appartient à quelqu'un, vous pouvez le lui emprunter. Et quand vous avez fini, vous devez le lui rendre. Vous ne le possédez pas.

Donc qu'est-ce qui se passe si nous essayons de modifier quelque chose que nous empruntons ? Essayez le code dans l'encart 4-6. Attention, spoiler : cela ne fonctionne pas !

Fichier : src/main.rs

```
fn main() {
    let s = String::from("hello");

    changer(&s);
}

fn changer(texte: &String) {
    texte.push_str(", world");
}
```

Entrée 4-6 : Tentative de modification d'une valeur empruntée.

Voici l'erreur :

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0596]: cannot borrow `*texte` as mutable, as it is behind a `&` reference
 --> src/main.rs:8:5
  |
7 | fn changer(texte: &String) {
  |                   ------- help: consider changing this to be a mutable
reference: `&mut String`
8 |     texte.push_str(", world");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^ `texte` is a `&` reference, so the data it
refers to cannot be borrowed as mutable

For more information about this error, try `rustc --explain E0596`.
error: could not compile `ownership` due to previous error
```

Comme les variables sont immuables par défaut, les références le sont aussi. Nous ne sommes pas autorisés à modifier une chose quand nous avons une référence vers elle.

Les références mutables

Nous pouvons résoudre le code de l'encart 4-6 pour nous permettre de modifier une valeur empruntée avec quelques petites modification qui utilisent plutôt une *référence mutable* :

Fichier : src/main.rs

```
fn main() {
    let mut s = String::from("hello");

    changer(&mut s);
}

fn changer(texte: &mut String) {
    texte.push_str(", world");
}
```

D'abord, nous précisons que `s` est `mut`. Ensuite, nous avons créé une référence mutable avec `&mut s` où nous appelons la fonction `change` et nous avons modifié la signature pour accepter de prendre une référence mutable avec `texte: &mut String`. Cela précise clairement que la fonction `change` va faire muter la valeur qu'elle emprunte.

Les références mutables ont une grosse contrainte : vous ne pouvez avoir qu'une seule référence mutable pour chaque donnée au même moment. Le code suivant qui va tenter de créer deux références mutables à `s` va échouer :

Fichier : src/main.rs

```
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

    println!("{}, {}", r1, r2);
```

Voici l'erreur :

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:5:14
  |
4 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{}, {}", r1, r2);
  |                        -- first borrow later used here

For more information about this error, try `rustc --explain E0499`.
error: could not compile `ownership` due to previous error
```

Cette erreur nous explique que ce code est invalide car nous ne pouvons pas emprunter `s` de manière mutable plus d'une fois au même moment. Le premier emprunt mutable est dans `r1` et doit perdurer jusqu'à ce qu'il soit utilisé dans le `println!`, mais pourtant entre la création de cette référence mutable et son utilisation, nous avons essayé de créer une autre référence mutable dans `r2` qui emprunte la même donnée que dans `r1`.

La limitation qui empêche d'avoir plusieurs références mutables vers la même donnée au même moment autorise les mutations, mais de manière très contrôlée. C'est quelque chose que les nouveaux Rustacés ont du mal à surmonter, car la plupart des langages vous permettent de modifier les données quand vous le voulez. L'avantage d'avoir cette contrainte est que Rust peut empêcher les accès concurrents au moment de la compilation. Un *accès concurrent* est une situation de concurrence qui se produit lorsque ces trois facteurs se combinent :

- Deux pointeurs ou plus accèdent à la même donnée au même moment.
- Au moins un des pointeurs est utilisé pour écrire dans cette donnée.
- On n'utilise aucun mécanisme pour synchroniser l'accès aux données.

L'accès concurrent provoque des comportements indéfinis et rend difficile le diagnostic et la résolution de problèmes lorsque vous essayez de les reproduire au moment de l'exécution ; Rust évite ce problème en refusant de compiler du code avec des accès concurrents !

Comme d'habitude, nous pouvons utiliser des accolades pour créer une nouvelle portée, pour nous permettre d'avoir plusieurs références mutables, mais pas *en même temps* :

```
    let mut s = String::from("hello");

    {
        let r1 = &mut s;
    } // r1 sort de la portée ici, donc nous pouvons créer une nouvelle
référence
      // sans problèmes.

    let r2 = &mut s;
```

Rust impose une règle similaire pour combiner les références immuables et mutables. Ce code va mener à une erreur :

```
    let mut s = String::from("hello");

    let r1 = &s; // sans problème
    let r2 = &s; // sans problème
    let r3 = &mut s; // GROS PROBLEME

    println!("{}, {}, and {}", r1, r2, r3);
```

Voici l'erreur :

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s; // sans problème
  |              -- immutable borrow occurs here
5 |     let r2 = &s; // sans problème
6 |     let r3 = &mut s; // GROS PROBLEME
  |              ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

Ouah ! Nous ne pouvons pas *non plus* avoir une référence mutable pendant que nous en avons une autre immuable vers la même valeur. Les utilisateurs d'une référence immuable ne s'attendent pas à ce que sa valeur change soudainement ! Cependant, l'utilisation de plusieurs références immuables ne pose pas de problème, car simplement lire une donnée ne va pas affecter la lecture de la donnée par les autres.

Notez bien que la portée d'une référence commence dès qu'elle est introduite et se poursuit

jusqu'au dernier endroit où cette référence est utilisée. Par exemple, le code suivant va se compiler car la dernière utilisation de la référence immuable, le `println!`, est située avant l'introduction de la référence mutable :

```
let mut s = String::from("hello");

let r1 = &s; // sans problème
let r2 = &s; // sans problème
println!("{} et {}", r1, r2);
//les variables r1 et r2 ne seront plus utilisés à partir d'ici

let r3 = &mut s; // sans problème
println!("{}", r3);
```

Les portées des références immuables `r1` et `r2` se terminent après le `println!` où elles sont utilisées pour la dernière fois, c'est-à-dire avant que la référence mutable `r3` soit créée. Ces portées ne se chevauchent pas, donc ce code est autorisé. La capacité du compilateur à dire si une référence n'est plus utilisée à un endroit avant la fin de la portée s'appelle en Anglais les *Non-Lexical Lifetimes* (ou NLL), et vous pouvez en apprendre plus dans le Guide de l'édition.

Même si ces erreurs d'emprunt peuvent parfois être frustrantes, n'oubliez pas que le compilateur de Rust nous signale un bogue potentiel en avance (au moment de la compilation plutôt que l'exécution) et vous montre où se situe exactement le problème. Ainsi, vous n'avez pas à chercher pourquoi vos données ne correspondent pas à ce que vous pensiez qu'elles devraient être.

Les références pendouillantes

Avec les langages qui utilisent les pointeurs, il est facile de créer par erreur un *pointeur pendouillant* (*dangling pointer*), qui est un pointeur qui pointe vers un emplacement mémoire qui a été donné à quelqu'un d'autre, en libérant de la mémoire tout en conservant un pointeur vers cette mémoire. En revanche, avec Rust, le compilateur garantit que les références ne seront jamais des références pendouillantes : si nous avons une référence vers une donnée, le compilateur va s'assurer que cette donnée ne va pas sortir de la portée avant que la référence vers cette donnée en soit elle-même sortie.

Essayons de créer une référence pendouillante pour voir comment Rust va les empêcher via une erreur au moment de la compilation :

Fichier : src/main.rs

```
fn main() {
    let reference_vers_rien = pendouille();
}

fn pendouille() -> &String {
    let s = String::from("hello");

    &s
}
```

Voici l'erreur :

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0106]: missing lifetime specifier
 --> src/main.rs:5:16
  |
5 | fn pendouille() -> &String {
  |                    ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no
value for it to be borrowed from
help: consider using the `'static` lifetime
  |
5 | fn pendouille() -> &'static String {
  |                    ~~~~~~~~

For more information about this error, try `rustc --explain E0106`.
error: could not compile `ownership` due to previous error
```

Ce message d'erreur fait référence à une fonctionnalité que nous n'avons pas encore vue : les *durées de vie*. Nous aborderons les durées de vie dans le chapitre 10. Mais, si vous mettez de côté les parties qui parlent de durées de vie, le message explique pourquoi le code pose problème :

```
this function's return type contains a borrowed value, but there is no value
for it to be borrowed from
```

Ce qui peut se traduire par :

```
Le type de retour de cette fonction contient une valeur empruntée, mais il n'y a
plus aucune valeur qui peut être empruntée.
```

Regardons de plus près ce qui se passe exactement à chaque étape de notre code de `pendouille` :

Fichier : src/main.rs

```
fn pendouille() -> &String { // pendouille retourne une référence vers une
String

  let s = String::from("hello"); // s est une nouvelle String

  &s // nous retournons une référence vers la String, s
} // Ici, s sort de la portée, et est libéré. Sa mémoire disparaît.
  // Attention, danger !
```

Comme `s` est créé dans `pendouille`, lorsque le code de `pendouille` est terminé, la variable `s` sera désallouée. Mais nous avons essayé de retourner une référence vers elle. Cela veut dire que cette référence va pointer vers une `String` invalide. Ce n'est pas bon ! Rust ne nous laissera pas faire cela.

Ici la solution est de renvoyer la `String` directement :

```
fn ne_pendouille_pas() -> String {
    let s = String::from("hello");

    s
}
```

Cela fonctionne sans problème. La possession est transférée à la valeur de retour de la fonction, et rien n'est désalloué.

Les règles de référencement

Récapitulons ce que nous avons vu à propos des références :

- À un instant donné, vous pouvez avoir *soit* une référence mutable, *soit* un nombre quelconque de références immuables.
- Les références doivent toujours être en vigueur.

Ensuite, nous aborderons un autre type de référence : les *slices*.

Le type slice

Une *slice* vous permet d'obtenir une référence vers une séquence continue d'éléments d'une collection plutôt que toute la collection. Une slice est un genre de référence, donc elle ne prend pas possession.

Voici un petit problème de programmation : écrire une fonction qui prend une chaîne de caractères et retourne le premier mot qu'elle trouve dans cette chaîne. Si la fonction ne trouve pas d'espace dans la chaîne, cela veut dire que la chaîne est en un seul mot, donc la chaîne en entier doit être retournée.

Voyons comment écrire la signature de cette fonction sans utiliser les slices, afin de comprendre le problème que règlent les slices :

```
fn premier_mot(s: &String) -> ?
```

La fonction `premier_mot` prend un `&String` comme paramètre. Nous ne voulons pas en prendre possession, donc c'est ce qu'il nous faut. Mais que devons-nous retourner ? Nous n'avons aucun moyen de désigner une *partie* d'une chaîne de caractères. Cependant, nous pouvons retourner l'indice de la fin du mot, qui se produit lorsqu'il y a un espace. Essayons cela, dans l'encart 4-7 :

Fichier : src/main.rs

```
fn premier_mot(s: &String) -> usize {
    let octets = s.as_bytes();

    for (i, &element) in octets.iter().enumerate() {
        if element == b' ' {
            return i;
        }
    }

    s.len()
}
```

Encart 4-7 : La fonction `premier_mot` qui retourne l'indice d'un octet provenant du paramètre `String`

Comme nous avons besoin de parcourir la `String` élément par élément et de vérifier si la valeur est une espace, nous convertissons notre `String` en un tableau d'octets en utilisant la méthode `as_bytes` :

```
    let octets = s.as_bytes();
```

Ensuite, nous créons un itérateur sur le tableau d'octets en utilisant la méthode `iter` :

```
    for (i, &element) in octets.iter().enumerate() {
```

Nous aborderons plus en détail les itérateurs dans le chapitre 13. Pour le moment, sachez que `iter` est une méthode qui retourne chaque élément d'une collection, et que `enumerate` transforme le résultat de `iter` pour retourner plutôt chaque élément comme un tuple. Le premier élément du tuple retourné par `enumerate` est l'indice, et le second élément est une référence vers l'élément. C'est un peu plus pratique que de calculer les indices par nous-mêmes.

Comme la méthode `enumerate` retourne un tuple, nous pouvons utiliser des motifs pour déstructurer ce tuple. Nous verrons les motifs au chapitre 6. Dans la boucle `for` , nous précisons un motif qui indique que nous définissons `i` pour l'indice au sein du tuple et `&element` pour l'octet dans le tuple. Comme nous obtenons une référence vers l'élément avec `.iter().enumerate()` , nous utilisons `&` dans le motif.

Au sein de la boucle `for` , nous recherchons l'octet qui représente l'espace en utilisant la syntaxe de littéral d'octet. Si nous trouvons une espace, nous retournons sa position. Sinon, nous retournons la taille de la chaîne en utilisant `s.len()` :

```
        if element == b' ' {
            return i;
        }
    }

    s.len()
```

Nous avons maintenant une façon de trouver l'indice de la fin du premier mot dans la chaîne de caractères, mais il y a un problème. Nous retournons un `usize` tout seul, mais il n'a du sens que lorsqu'il est lié au `&String` . Autrement dit, comme il a une valeur séparée de la `String` , il n'y a pas de garantie qu'il restera toujours valide dans le futur. Imaginons le programme dans l'encart 4-8 qui utilise la fonction `premier_mot` de l'encart 4-7 :

Fichier : src/main.rs

```
fn main() {
    let mut s = String::from("hello world");

    let mot = premier_mot(&s); // la variable mot aura 5 comme valeur.

    s.clear(); // ceci vide la String, elle vaut maintenant "".

    // mot a toujours la valeur 5 ici, mais il n'y a plus de chaîne qui donne
    // du sens à la valeur 5. mot est maintenant complètement invalide !
}
```

Encart 4-8 : On stocke le résultat de l'appel à la fonction `premier_mot` et ensuite on change le contenu de la `String`

Ce programme se compile sans aucune erreur et le ferait toujours si nous utilisions `mot` après avoir appelé `s.clear()`. Comme `mot` n'est pas du tout lié à `s`, `mot` contient toujours la valeur `5`. Nous pourrions utiliser cette valeur `5` avec la variable `s` pour essayer d'en extraire le premier mot, mais cela serait un bogue, car le contenu de `s` a changé depuis que nous avons enregistré `5` dans `mot`.

Se préoccuper en permanence que l'indice présent dans `mot` ne soit plus synchronisé avec les données présentes dans `s` est fastidieux et source d'erreur ! La gestion de ces indices est encore plus risquée si nous écrivons une fonction `second_mot`. Sa signature ressemblerait à ceci :

```
fn second_mot(s: &String) -> (usize, usize) {
```

Maintenant, nous avons un indice de début *et* un indice de fin, donc nous avons encore plus de valeurs qui sont calculées à partir d'une donnée dans un état donné, mais qui ne sont pas liées du tout à l'état de cette donnée. Nous avons trois variables isolées qui ont besoin d'être maintenues à jour.

Heureusement, Rust a une solution pour ce problème : les *slices* de chaînes de caractères.

Les slices de chaînes de caractères

Une *slice de chaîne de caractères* (ou *slice de chaîne*) est une référence à une partie d'une `String`, et ressemble à ceci :

```
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
```

Plutôt que d'être une référence vers toute la `String`, `hello` est une référence vers une partie de la `String`, comme indiqué dans la partie supplémentaire `[0..5]`. Nous créons des slices en utilisant un intervalle entre crochets en spécifiant `[indice_debut..indice_fin]`, où `indice_debut` est la position du premier octet de la slice et `indice_fin` est la position juste après le dernier octet de la slice. En interne, la structure de données de la slice stocke la position de départ et la longueur de la slice, ce qui correspond à `indice_fin` moins `indice_debut`. Donc dans le cas de `let world = &s[6..11];`, `world` est une slice qui contient un pointeur vers le sixième octet de `s` et une longueur de 5.

L'illustration 4-6 montre ceci dans un schéma.

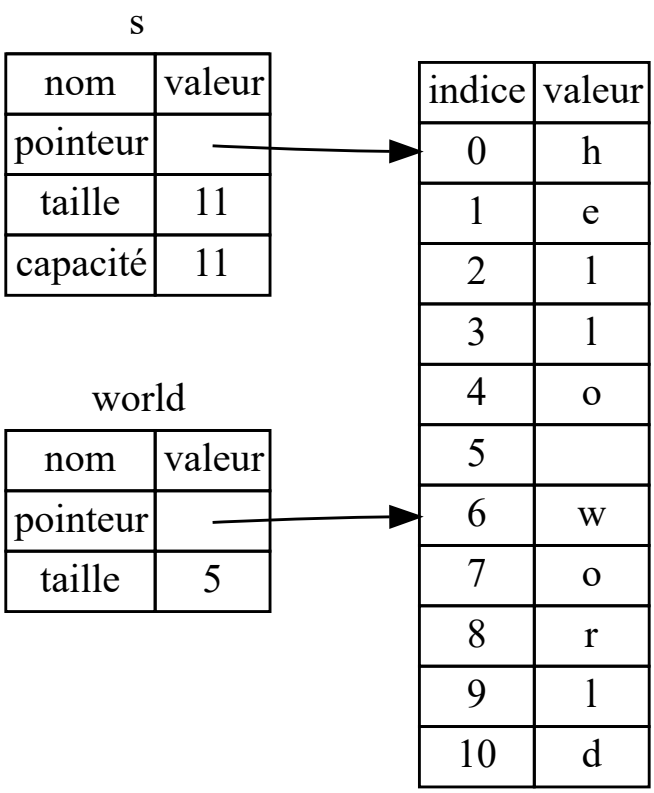


Illustration 4-6 : Une slice de chaîne qui pointe vers une partie d'une `String`

Avec la syntaxe d'intervalle `..` de Rust, si vous voulez commencer à l'indice zéro, vous pouvez ne rien mettre avant les deux points. Autrement dit, ces deux cas sont identiques :

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

De la même manière, si votre slice contient le dernier octet de la `String`, vous pouvez ne

rien mettre à la fin. Cela veut dire que ces deux cas sont identiques :

```
let s = String::from("hello");

let taille = s.len();

let slice = &s[3..taille];
let slice = &s[3..];
```

Vous pouvez aussi ne mettre aucune limite pour créer une slice de toute la chaîne de caractères. Ces deux cas sont donc identiques :

```
let s = String::from("hello");

let taille = s.len();

let slice = &s[0..taille];
let slice = &s[..];
```

> Remarque : Les indices de l'intervalle d'une slice de chaîne doivent toujours se trouver dans les zones acceptables de séparation des caractères encodés en UTF-8. Si vous essayez de créer une slice de chaîne qui s'arrête au milieu d'un caractère encodé sur plusieurs octets, votre programme va se fermer avec une erreur. Afin de simplifier l'explication des slices de chaînes, nous utiliserons uniquement l'ASCII dans cette section ; nous verrons la gestion d'UTF-8 dans la section "Stocker du texte encodé en UTF-8 avec les chaînes de caractères" du chapitre 8.

Maintenant que nous savons tout cela, essayons de réécrire `premier_mot` pour qu'il retourne une slice. Le type pour les slices de chaînes de caractères s'écrit `&str` :

Fichier : src/main.rs

```
fn premier_mot(s: &String) -> &str {
    let octets = s.as_bytes();

    for (i, &element) in octets.iter().enumerate() {
        if element == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Nous récupérons l'indice de la fin du mot de la même façon que nous l'avions fait dans l'encart 4-7, en cherchant la première occurrence d'une espace. Lorsque nous trouvons une espace, nous retournons une slice de chaîne en utilisant le début de la chaîne de caractères et l'indice de l'espace comme indices de début et de fin respectivement.

Désormais, quand nous appelons `premier_mot`, nous récupérons une unique valeur qui est liée à la donnée de base. La valeur se compose d'une référence vers le point de départ de la slice et du nombre d'éléments dans la slice.

Retourner une slice fonctionnerait aussi pour une fonction `second_mot` :

```
fn second_mot(s: &String) -> &str {
```

Nous avons maintenant une API simple qui est bien plus difficile à mal utiliser, puisque le compilateur va s'assurer que les références dans la `String` seront toujours en vigueur. Vous souvenez-vous du bogue du programme de l'encart 4-8, lorsque nous avions un indice vers la fin du premier mot mais qu'ensuite nous avions vidé la chaîne de caractères et que notre indice n'était plus valide ? Ce code était logiquement incorrect, mais ne montrait pas immédiatement une erreur. Les problèmes apparaîtront plus tard si nous essayons d'utiliser l'indice du premier mot avec une chaîne de caractères qui a été vidée. Les slices rendent ce bogue impossible et nous signalent bien plus tôt que nous avons un problème avec notre code. Utiliser la version avec la slice de `premier_mot` va causer une erreur de compilation :

Fichier : src/main.rs

```
fn main() {
    let mut s = String::from("hello world");

    let mot = premier_mot(&s);

    s.clear(); // Erreur !

    println!("Le premier mot est : {}", mot);
}
```

Voici l'erreur du compilateur :

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
  --> src/main.rs:18:5
   |
16 |     let mot = premier_mot(&s);
   |                           -- immutable borrow occurs here
17 |
18 |     s.clear(); // Erreur !
   |     ^^^^^^^^^ mutable borrow occurs here
19 |
20 |     println!("Le premier mot est : {}", mot);
   |                                         --- immutable borrow later used
here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

Rappelons-nous que d'après les règles d'emprunt, si nous avons une référence immuable vers quelque chose, nous ne pouvons pas avoir une référence mutable en même temps. Étant donné que `clear` a besoin de modifier la `String`, il a besoin d'une référence mutable. Le `println!` qui a lieu après l'appel à `clear` utilise la référence à `mot`, donc la référence immuable sera toujours en vigueur à cet endroit. Rust interdit la référence mutable dans `clear` et la référence immuable pour `mot` au même moment, et la compilation échoue. Non seulement Rust a simplifié l'utilisation de notre API, mais il a aussi éliminé une catégorie entière d'erreurs au moment de la compilation !

Les littéraux de chaîne de caractères sont aussi des slices

Rappelez-vous lorsque nous avons appris que les littéraux de chaîne de caractères étaient enregistrés dans le binaire. Maintenant que nous connaissons les slices, nous pouvons désormais comprendre les littéraux de chaîne.

```
let s = "Hello, world!";
```

Ici, le type de `s` est un `&str` : c'est une slice qui pointe vers un endroit précis du binaire. C'est aussi la raison pour laquelle les littéraux de chaîne sont immuables ; `&str` est une référence immuable.

Les slices de chaînes de caractères en paramètres

Savoir que l'on peut utiliser des slices de littéraux et de `String` nous incite à apporter une

petite amélioration à `premier_mot`, dont voici la signature :

```
fn premier_mot(s: &String) -> &str {
```

Un Rustacé plus expérimenté écrirait plutôt la signature de l'encart 4-9, car cela nous permet d'utiliser la même fonction sur les `&String` et aussi les `&str` :

```
fn premier_mot(s: &str) -> &str {
```

Encart 4-9 : Amélioration de la fonction `premier_mot` en utilisant une slice de chaîne de caractères comme type du paramètre `s`

Si nous avons une slice de chaîne, nous pouvons la passer en argument directement. Si nous avons une `String`, nous pouvons envoyer une référence ou une slice de la `String`. Cette flexibilité nous est offerte par l'*extrapolation de déréferencement*, une fonctionnalité que nous allons découvrir dans une section du Chapitre 15. Définir une fonction qui prend une slice de chaîne plutôt qu'une référence à une `String` rend notre API plus générique et plus utile sans perdre aucune fonctionnalité :

Fichier : src/main.rs

```
fn main() {
    let ma_string = String::from("hello world");

    // `premier_mot` fonctionne avec les slices de `String`, que ce soit sur
    // une partie ou sur sur son intégralité
    let mot = premier_mot(&ma_string[0..6]);
    let mot = premier_mot(&ma_string[..]);

    // `premier_mot` fonctionne également sur des références vers des `String`,
    // qui sont équivalentes à des slices de toute la `String`
    let mot = premier_mot(&ma_string);

    let mon_litteral_de_chaine = "hello world";

    // `premier_mot` fonctionne avec les slices de littéraux de chaîne, qu'elles
    // soient partielles ou intégrales
    let mot = premier_mot(&mon_litteral_de_chaine[0..6]);
    let mot = premier_mot(&mon_litteral_de_chaine[..]);

    // Comme les littéraux de chaîne *sont* déjà des slices de chaînes,
    // cela fonctionne aussi, sans la syntaxe de slice !
    let mot = premier_mot(mon_litteral_de_chaine);
}
```

Les autres slices

Les slices de chaînes de caractères, comme vous pouvez l'imaginer, sont spécifiques aux chaînes de caractères. Mais il existe aussi un type de slice plus générique. Imaginons ce tableau de données :

```
let a = [1, 2, 3, 4, 5];
```

Tout comme nous pouvons nous référer à une partie d'une chaîne de caractères, nous pouvons nous référer à une partie d'un tableau. Nous pouvons le faire comme ceci :

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];

assert_eq!(slice, &[2, 3]);
```

Cette slice est de type `&[i32]`. Elle fonctionne de la même manière que les slices de chaînes de caractères, en enregistrant une référence vers le premier élément et une longueur. Vous utiliserez ce type de slice pour tous les autres types de collections. Nous aborderons ces collections en détail quand nous verrons les vecteurs au chapitre 8.

Résumé

Les concepts de possession, d'emprunt et de slices garantissent la sécurité de la mémoire dans les programmes Rust au moment de la compilation. Le langage Rust vous donne le contrôle sur l'utilisation de la mémoire comme tous les autres langages de programmation système, mais le fait que celui qui possède des données nettoie automatiquement ces données quand il sort de la portée vous permet de ne pas avoir à écrire et déboguer du code en plus pour avoir cette fonctionnalité.

La possession influe sur de nombreuses autres fonctionnalités de Rust, c'est pourquoi nous allons encore parler de ces concepts plus loin dans le livre. Passons maintenant au chapitre 5 et découvrons comment regrouper des données ensemble dans une `struct`.

Utiliser les structures pour structurer des données apparentées

Une *struct*, ou *structure*, est un type de données personnalisé qui vous permet de rassembler plusieurs valeurs associées et les nommer pour former un groupe cohérent. Si vous êtes familier avec un langage orienté objet, une structure est en quelque sorte l'ensemble des attributs d'un objet. Dans ce chapitre, nous comparerons les tuples avec les structures afin de construire ce que vous connaissez déjà et de montrer à quels moments les structures sont plus pertinentes pour grouper des données. Nous verrons comment définir les fonctions associées, en particulier le type de fonctions associées que l'on appelle les *méthodes*, dans le but d'implémenter un comportement associé au type d'une structure. Les structures et les énumérations (traitées au chapitre 6) sont les fondements de la création de nouveaux types au sein de votre programme pour tirer pleinement parti des vérifications de types effectuées par Rust à la compilation.

Définir et instancier des structures

Les structures sont similaires aux tuples, qu'on a vus dans une section du chapitre 3, car tous les deux portent plusieurs valeurs associées. Comme pour les tuples, les éléments d'une structure peuvent être de différents types. Contrairement aux tuples, dans une structure on doit nommer chaque élément des données afin de clarifier le rôle de chaque valeur. L'ajout de ces noms font que les structures sont plus flexibles que les tuples : on n'a pas à utiliser l'ordre des données pour spécifier ou accéder aux valeurs d'une instance.

Pour définir une structure, on tape le mot-clé `struct` et on donne un nom à toute la structure. Le nom d'une structure devrait décrire l'utilisation des éléments des données regroupés. Ensuite, entre des accolades, on définit le nom et le type de chaque élément des données, qu'on appelle un *champ*. Par exemple, l'encart 5-1 montre une structure qui stocke des informations à propos d'un compte d'utilisateur.

```
struct Utilisateur {
    actif: bool,
    pseudo: String,
    email: String,
    nombre_de_connexions: u64,
}
```

Encart 5-1 : la définition d'une structure `Utilisateur`

Pour utiliser une structure après l'avoir définie, on crée une *instance* de cette structure en indiquant des valeurs concrètes pour chacun des champs. On crée une instance en indiquant le nom de la structure puis en ajoutant des accolades qui contiennent des paires de `clé: valeur`, où les clés sont les noms des champs et les valeurs sont les données que l'on souhaite stocker dans ces champs. Nous n'avons pas à préciser les champs dans le même ordre qu'on les a déclarés dans la structure. En d'autres termes, la définition de la structure décrit un gabarit pour le type, et les instances remplissent ce gabarit avec des données précises pour créer des valeurs de ce type. Par exemple, nous pouvons déclarer un utilisateur précis comme dans l'encart 5-2.

```
fn main() {
    let utilisateur1 = Utilisateur {
        email: String::from("quelquun@example.com"),
        pseudo: String::from("pseudoquelconque123"),
        actif: true,
        nombre_de_connexions: 1,
    };
}
```

Encart 5-2 : création d'une instance de la structure `Utilisateur`

Pour obtenir une valeur spécifique depuis une structure, on utilise la notation avec le point. Si nous voulions seulement l'adresse e-mail de cet utilisateur, on pourrait utiliser `utilisateur1.email` partout où on voudrait utiliser cette valeur. Si l'instance est mutable, nous pourrions changer une valeur en utilisant la notation avec le point et assigner une valeur à ce champ en particulier. L'encart 5-3 montre comment changer la valeur du champ `email` d'une instance mutable de `Utilisateur` .

```
fn main() {
    let mut utilisateur1 = Utilisateur {
        email: String::from("quelquun@example.com"),
        pseudo: String::from("pseudoquelconque123"),
        actif: true,
        nombre_de_connexions: 1,
    };

    utilisateur1.email = String::from("unautremail@example.com");
}
```

Encart 5-3 : changement de la valeur du champ `email` d'une instance de `Utilisateur`

À noter que l'instance tout entière doit être mutable ; Rust ne nous permet pas de marquer seulement certains champs comme mutables. Comme pour toute expression, nous pouvons construire une nouvelle instance de la structure comme dernière expression du corps d'une fonction pour retourner implicitement cette nouvelle instance.

L'encart 5-4 montre une fonction `creer_utilisateur` qui retourne une instance de `Utilisateur` avec l'adresse e-mail et le pseudo fournis. Le champ `actif` prend la valeur `true` et le `nombre_de_connexions` prend la valeur `1` .

```
fn creer_utilisateur(email: String, pseudo: String) -> Utilisateur {
    Utilisateur {
        email: email,
        pseudo: pseudo,
        actif: true,
        nombre_de_connexions: 1,
    }
}
```

Encart 5-4 : une fonction `creer_utilisateur` qui prend en entrée une adresse e-mail et un pseudo et retourne une instance de `Utilisateur`

Il est logique de nommer les paramètres de fonction avec le même nom que les champs de la structure, mais devoir répéter les noms de variables et de champs `email` et `pseudo` est un peu pénible. Si la structure avait plus de champs, répéter chaque nom serait encore plus fatigant. Heureusement, il existe un raccourci pratique !

Utiliser le raccourci d'initialisation des champs

Puisque les noms des paramètres et les noms de champs de la structure sont exactement les mêmes dans l'encart 5-4, on peut utiliser la syntaxe de *raccourci d'initialisation des champs* pour réécrire `creer_utilisateur` de sorte qu'elle se comporte exactement de la même façon sans avoir à répéter `email` et `pseudo`, comme le montre l'encart 5-5.

```
fn creer_utilisateur(email: String, pseudo: String) -> Utilisateur {
    Utilisateur {
        email,
        pseudo,
        actif: true,
        nombre_de_connexions: 1,
    }
}
```

Encart 5-5 : une fonction `creer_utilisateur` qui utilise le raccourci d'initialisation des champs parce que les paramètres `email` et `pseudo` ont le même nom que les champs de la structure

Ici, on crée une nouvelle instance de la structure `Utilisateur`, qui possède un champ nommé `email`. On veut donner au champ `email` la valeur du paramètre `email` de la fonction `creer_utilisateur`. Comme le champ `email` et le paramètre `email` ont le même nom, on a uniquement besoin d'écrire `email` plutôt que `email: email`.

Créer des instances à partir d'autres instances avec la syntaxe de mise à jour de structure

Il est souvent utile de créer une nouvelle instance de structure qui comporte la plupart des valeurs d'une autre instance tout en en changeant certaines. Vous pouvez utiliser pour cela la *syntaxe de mise à jour de structure*.

Tout d'abord, dans l'encart 5-6 nous montrons comment créer une nouvelle instance de `Utilisateur` dans `utilisateur2` sans la syntaxe de mise à jour de structure. On donne de nouvelles valeurs à `email` et `pseudo` mais on utilise pour les autres champs les mêmes valeurs que dans `utilisateur1` qu'on a créé à l'encart 5-2.

```
fn main() {
    // -- partie masquée ici --

    let utilisateur2 = Utilisateur {
        actif: utilisateur1.actif,
        pseudo: utilisateur1.email,
        email: String::from("quelquundautre@example.com"),
        nombre_de_connexions: utilisateur1.nombre_de_connexions,
    };
}
```

Encart 5-6 : création d'une nouvelle instance de `Utilisateur` en utilisant une des valeurs de `utilisateur1`.

En utilisant la syntaxe de mise à jour de structure, on peut produire le même résultat avec moins de code, comme le montre l'encart 5-7. La syntaxe `..` indique que les autres champs auxquels on ne donne pas explicitement de valeur devraient avoir la même valeur que dans l'instance précisée.

```
fn main() {
    // -- partie masquée ici --

    let utilisateur2 = Utilisateur {
        email: String::from("quelquundautre@example.com"),
        ..utilisateur1
    };
}
```

Encart 5-7 : utilisation de la syntaxe de mise à jour de structure pour assigner de nouvelles valeurs à `email` d'une nouvelle instance de `Utilisateur` tout en utilisant les autres valeurs de `utilisateur1`

Le code dans l'encart 5-7 crée aussi une instance dans `utilisateur2` qui a une valeur différente pour `email`, mais qui as les mêmes valeurs pour les champs `pseudo`, `actif` et `nombre_de_connexions` que `utilisateur1`. Le `..utilisateur1` doit être inséré à la fin pour préciser que tous les champs restants obtiendrons les valeurs des champs correspondants de `utilisateur1`, mais nous pouvons renseigner les valeurs des champs dans n'importe quel ordre, peu importe leur position dans la définition de la structure.

Veuillez notez que la syntaxe de la mise à jour de structure utilise un `=` comme le ferait une assignation ; car cela déplace les données, comme nous l'avons vu dans une des sections au chapitre 4. Dans cet exemple, nous ne pouvons plus utiliser `utilisateur1` après avoir créé `utilisateur2` car la `String` dans le champ `pseudo` de `utilisateur1` a été déplacée dans `utilisateur2`. Si nous avions donné des nouvelles valeurs pour chacune des `String` `email` et `pseudo`, et que par conséquent nous aurions déplacé uniquement les valeurs de

`actif` et de `nombre_de_connexions` à partir de `utilisateur1`, alors `utilisateur1` restera en vigueur après avoir créé `utilisateur2`. Les types de `actif` et de `nombre_de_connexions` sont de types qui implémentent le trait `Copy`, donc le comportement décris dans la section à propos de copy aura lieu ici.

Utilisation de structures tuples sans champ nommé pour créer des types différents

Rust prend aussi en charge des structures qui ressemblent à des tuples, appelées *structures tuples*. La signification d'une structure tuple est donnée par son nom. En revanche, ses champs ne sont pas nommés ; on ne précise que leurs types. Les structures tuples servent lorsqu'on veut donner un nom à un tuple pour qu'il soit d'un type différent des autres tuples, et lorsque nommer chaque champ comme dans une structure classique serait trop verbeux ou redondant.

La définition d'une structure tuple commence par le mot-clé `struct` et le nom de la structure suivis des types des champs du tuple. Par exemple ci-dessous, nous définissons et utilisons deux structures tuples nommées `Couleur` et `Point` :

```
struct Couleur(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let noir = Couleur(0, 0, 0);
    let origine = Point(0, 0, 0);
}
```

Notez que les valeurs `noir` et `origine` sont de types différents parce que ce sont des instances de structures tuples différentes. Chaque structure que l'on définit constitue son propre type, même si les champs au sein de la structure ont les mêmes types. Par exemple, une fonction qui prend un paramètre de type `Couleur` ne peut pas prendre un argument de type `Point` à la place, bien que ces deux types soient tous les deux constitués de trois valeurs `i32`. Mis à part cela, les instances de stuctures tuples se comportent comme des tuples : on peut les déstructurer en éléments individuels, on peut utiliser un `.` suivi de l'indice pour accéder individuellement à une valeur, et ainsi de suite.

Les structures unité sans champs

On peut aussi définir des structures qui n'ont pas de champs ! Cela s'appelle des *structures unité* parce qu'elles se comportent d'une façon analogue au type unité, `()`, que nous avons

vu dans la section sur les tuples. Les structures unité sont utiles lorsqu'on doit implémenter un trait sur un type mais qu'on n'a aucune donnée à stocker dans le type en lui-même. Nous aborderons les traits au chapitre 10. Voici un exemple de déclaration et d'instanciation d'une structure unité `ToujoursEgal` :

```
struct ToujoursEgal;

fn main() {
    let sujet = ToujoursEgal;
}
```

Pour définir `ToujoursEgal`, nous utilisons le mot-clé `struct`, puis le nom que nous voulons lui donner, et enfin un point-virgule. Pas besoin d'accolades ou de parenthèses ! Ensuite, nous pouvons obtenir une instance de `ToujourEgal` dans la variable `sujet` de la même manière : utilisez le nom que vous avez défini, sans aucune accolade ou parenthèse. Imaginez que plus tard nous allons implémenter un comportement pour ce type pour que toutes les instances de `ToujourEgal` soient toujours égales à chaque instance de n'importe quel autre type, peut-être pour avoir un résultat connu pour des besoins de tests. Nous n'avons besoin d'aucune donnée pour implémenter ce comportement ! Vous verrez au chapitre 10 comment définir des traits et les implémenter sur n'importe quel type, y compris sur les structures unité.

La possession des données d'une structure

Dans la définition de la structure `Utilisateur` de l'encart 5-1, nous avions utilisé le type possédé `String` plutôt que le type de *slice* de chaîne de caractères `&str`. Il s'agit d'un choix délibéré puisque nous voulons que chacune des instances de cette structure possèdent toutes leurs données et que ces données restent valides tant que la structure tout entière est valide.

Il est aussi possible pour les structures de stocker des références vers des données possédées par autre chose, mais cela nécessiterait d'utiliser des *durées de vie*, une fonctionnalité de Rust que nous aborderons au chapitre 10. Les durées de vie assurent que les données référencées par une structure restent valides tant que la structure l'est aussi. Disons que vous essayiez de stocker une référence dans une structure sans indiquer de durées de vie, comme ce qui suit, ce qui ne fonctionnera pas :

Fichier : src/main.rs

```
struct Utilisateur {
    actif: bool,
    pseudo: &str,
    email: &str,
    nombre_de_connexions: u64,
}

fn main() {
    let utilisateur1 = Utilisateur {
        email: "quelquun@example.com",
        pseudo: "pseudoquelconque123",
        actif: true,
        nombre_de_connexions: 1,
    };
}
```

Le compilateur réclamera l'ajout des durées de vie :

```
$ cargo run
   Compiling structs v0.1.0 (file:///projects/structs)
error[E0106]: missing lifetime specifier
 --> src/main.rs:3:15
  |
3 |     pseudo: &str,
  |             ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct Utilisateur<'a> {
2 |     actif: bool,
3 ~     pseudo: &'a str,
  |

error[E0106]: missing lifetime specifier
 --> src/main.rs:4:12
  |
4 |     email: &str,
  |            ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct Utilisateur<'a> {
2 |     actif: bool,
3 |     pseudo: &str,
4 ~     email: &'a str,
  |

For more information about this error, try `rustc --explain E0106`.
error: could not compile `structs` due to 2 previous errors
```

Au chapitre 10, nous aborderons la façon de corriger ces erreurs pour qu'on puisse

stocker des références dans des structures, mais pour le moment, nous résoudrons les erreurs comme celles-ci en utilisant des types possédés comme `String` plutôt que des références comme `&str`.

Un exemple de programme qui utilise des structures

Pour comprendre dans quels cas nous voudrions utiliser des structures, écrivons un programme qui calcule l'aire d'un rectangle. Nous commencerons en utilisant de simples variables, puis on remaniera le code jusqu'à utiliser des structures à la place.

Créons un nouveau projet binaire avec Cargo nommé *rectangles* qui prendra la largeur et la hauteur en pixels d'un rectangle et qui calculera l'aire de ce rectangle. L'encart 5-8 montre un petit programme qui effectue cette tâche d'une certaine manière dans le *src/main.rs* de notre projet.

Fichier: src/main.rs

```
fn main() {
    let largeur1 = 30;
    let hauteur1 = 50;

    println!(
        "L'aire du rectangle est de {} pixels carrés.",
        aire(largeur1, hauteur1)
    );
}

fn aire(largeur: u32, hauteur: u32) -> u32 {
    largeur * hauteur
}
```

Encart 5-8 : calcul de l'aire d'un rectangle défini par les variables distinctes `largeur` et `hauteur`

Maintenant, lancez ce programme avec `cargo run` :

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/rectangles`
L'aire du rectangle est de 1500 pixels carrés.
```

Ce code arrive à déterminer l'aire du rectangle en appelant la fonction `aire` avec chaque dimension, mais on peut faire mieux pour clarifier ce code et le rendre plus lisible.

Le problème de ce code se voit dans la signature de `aire` :

```
fn aire(largeur: u32, hauteur: u32) -> u32 {
```

La fonction `aire` est censée calculer l'aire d'un rectangle, mais la fonction que nous avons

écrite a deux paramètres, et il n'est pas précisé nulle part dans notre programme à quoi sont liés les paramètres. Il serait plus lisible et plus gérable de regrouper ensemble la largeur et la hauteur. Nous avons déjà vu dans la section "Le type *tuple*" du chapitre 3 une façon qui nous permettrait de le faire : en utilisant des tuples.

Remanier le code avec des tuples

L'encart 5-9 nous montre une autre version de notre programme qui utilise des tuples.

Fichier : src/main.rs

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "L'aire du rectangle est de {} pixels carrés.",
        aire(rect1)
    );
}

fn aire(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

Encart 5-9 : Renseigner la largeur et la hauteur du rectangle dans un tuple

D'une certaine façon, ce programme est meilleur. Les tuples nous permettent de structurer un peu plus et nous ne passons plus qu'un argument. Mais d'une autre façon, cette version est moins claire : les tuples ne donnent pas de noms à leurs éléments, donc il faut accéder aux éléments du tuple via leur indice, ce qui rends plus compliqué notre calcul.

Le mélange de la largeur et la hauteur n'est pas important pour calculer l'aire, mais si on voulait afficher le rectangle à l'écran, cela serait problématique ! Il nous faut garder à l'esprit que la `largeur` est l'élément à l'indice `0` du tuple et que la `hauteur` est l'élément à l'indice `1`. Cela complexifie le travail de quelqu'un d'autre de le comprendre et s'en souvenir pour qu'il puisse l'utiliser. Comme on n'a pas exprimé la signification de nos données dans notre code, il est plus facile de faire des erreurs.

Remanier avec des structures : donner plus de sens

On utilise des structures pour rendre les données plus expressives en leur donnant des noms. On peut transformer le tuple que nous avons utilisé en une structure nommée dont ses éléments sont aussi nommés, comme le montre l'encart 5-10.

Fichier : src/main.rs

```
struct Rectangle {
    largeur: u32,
    hauteur: u32,
}

fn main() {
    let rect1 = Rectangle { largeur: 30, hauteur: 50 };

    println!(
        "L'aire du rectangle est de {} pixels carrés.",
        aire(&rect1)
    );
}

fn aire(rectangle: &Rectangle) -> u32 {
    rectangle.largeur * rectangle.hauteur
}
```

Encart 5-10 : Définition d'une structure `Rectangle`

Ici, on a défini une structure et on l'a appelée `Rectangle`. Entre les accolades, on a défini les champs `largeur` et `hauteur`, tous deux du type `u32`. Puis dans `main`, on crée une instance de `Rectangle` de largeur 30 et de hauteur 50.

Notre fonction `aire` est désormais définie avec un unique paramètre, nommé `rectangle`, et dont le type est une référence immuable vers une instance de la structure `Rectangle`. Comme mentionné au chapitre 4, on préfère emprunter la structure au lieu d'en prendre possession. Ainsi, elle reste en possession de `main` qui peut continuer à utiliser `rect1` ; c'est pourquoi on utilise le `&` dans la signature de la fonction ainsi que dans l'appel de fonction.

La fonction `aire` accède aux champs `largeur` et `hauteur` de l'instance de `Rectangle`. Notre signature de fonction pour `aire` est enfin explicite : calculer l'aire d'un `Rectangle` en utilisant ses champs `largeur` et `hauteur`. Cela explique que la largeur et la hauteur sont liées entre elles, et cela donne des noms descriptifs aux valeurs plutôt que d'utiliser les valeurs du tuple avec les indices `0` et `1`. On gagne en clarté.

Ajouter des fonctionnalités utiles avec les traits dérivés

Cela serait pratique de pouvoir afficher une instance de `Rectangle` pendant qu'on débogue notre programme et de voir la valeur de chacun de ses champs. L'encart 5-11 essaye de le faire en utilisant la macro `println!` comme on l'a fait dans les chapitres précédents.

Cependant, cela ne fonctionne pas.

Fichier : src/main.rs

```
struct Rectangle {
    largeur: u32,
    hauteur: u32,
}

fn main() {
    let rect1 = Rectangle {
        largeur: 30,
        hauteur: 50
    };

    println!("rect1 est {}", rect1);
}
```

Encart 5-11 : Tentative d'afficher une instance de `Rectangle`

Lorsqu'on compile ce code, on obtient ce message d'erreur qui nous informe que `Rectangle` n'implémente pas le trait `std::fmt::Display` :

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

La macro `println!` peut faire toutes sortes de formatages textuels, et par défaut, les accolades demandent à `println!` d'utiliser le formatage appelé `Display`, pour convertir en texte destiné à être vu par l'utilisateur final. Les types primitifs qu'on a vus jusqu'ici implémentent `Display` par défaut puisqu'il n'existe qu'une seule façon d'afficher un `1` ou tout autre type primitif à l'utilisateur. Mais pour les structures, la façon dont `println!` devrait formater son résultat est moins claire car il y a plus de possibilités d'affichage : Voulez-vous des virgules ? Voulez-vous afficher les accolades ? Est-ce que tous les champs devraient être affichés ? À cause de ces ambiguïtés, Rust n'essaye pas de deviner ce qu'on veut, et les structures n'implémentent pas `Display` par défaut pour l'utiliser avec `println!` et les espaces réservés `{}`.

Si nous continuons de lire les erreurs, nous trouvons cette remarque utile :

```
   = help: the trait `std::fmt::Display` is not implemented for `Rectangle`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-
print) instead
```

Le compilateur nous informe que dans notre chaîne de formatage, on est peut-être en mesure d'utiliser `{:?}` (ou `{:#?}` pour un affichage plus élégant).

Essayons cela ! L'appel de la macro `println!` ressemble maintenant à `println!("rect1 est {:?}", rect1);`. Insérer le sélecteur `:?` entre les accolades permet d'indiquer à `println!` que nous voulons utiliser le formatage appelé `Debug`. Le trait `Debug` nous permet d'afficher notre structure d'une manière utile aux développeurs pour qu'on puisse voir sa valeur pendant qu'on débogue le code.

Compilez le code avec ce changement. Zut ! On a encore une erreur, nous informant cette fois-ci que `Rectangle` n'implémente pas `std::fmt::Debug` :

```
error[E0277]: `Rectangle` doesn't implement `Debug`
```

Mais une nouvelle fois, le compilateur nous fait une remarque utile :

```
    = help: the trait `Debug` is not implemented for `Rectangle`
    = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for
Rectangle`
```

Il nous conseille d'ajouter `#[derive(Debug)]` ou d'implémenter manuellement `std::fmt::Debug`.

Rust *inclut* bel et bien une fonctionnalité pour afficher des informations de débogage, mais nous devons l'activer explicitement pour la rendre disponible sur notre structure. Pour ce faire, on ajoute l'attribut externe `#[derive(Debug)]` juste avant la définition de la structure, comme le montre l'encart 5-12.

Fichier : src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    largeur: u32,
    hauteur: u32,
}

fn main() {
    let rect1 = Rectangle {
        largeur: 30,
        hauteur: 50
    };

    println!("rect1 est {:?}", rect1);
}
```

Encart 5-12 : ajout de l'attribut pour dériver le trait `Debug` et afficher l'instance de `Rectangle` en utilisant le formatage de débogage

Maintenant, quand on exécute le programme, nous n'avons plus d'erreurs et ce texte

s'affiche à l'écran :

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 est Rectangle { largeur: 30, hauteur: 50 }
```

Super ! Ce n'est pas le plus beau des affichages, mais cela montre les valeurs de tous les champs de cette instance, ce qui serait assurément utile lors du débogage. Quand on a des structures plus grandes, il serait bien d'avoir un affichage un peu plus lisible ; dans ces cas-là, on pourra utiliser `{:#?}` au lieu de `{:?}` dans la chaîne de formatage. Dans cette exemple, l'utilisation du style `{:#?}` va afficher ceci :

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 est Rectangle {
    largeur: 30,
    hauteur: 50,
}
```

Une autre façon d'afficher une valeur en utilisant le format `Debug` est d'utiliser la macro `dbg!`, qui prend possession de l'expression, affiche le nom du fichier et la ligne de votre code où se trouve cet appel à la macro `dbg!` ainsi que le résultat de cette expression, puis rend la possession de cette valeur.

> Remarque : l'appel à la macro `dbg!` écrit dans le flux d'erreur standard de la console (`stderr`), contrairement à `println!` qui écrit dans le flux de sortie standard de la console (`stdout`). Nous reparlerons de `stderr` et de `stdout` dans une section du chapitre 12.

Voici un exemple dans lequel nous nous intéressons à la valeur assignée au champ `largeur`, ainsi que la valeur de toute la structure `rect1` :

```
#[derive(Debug)]
struct Rectangle {
    largeur: u32,
    hauteur: u32,
}

fn main() {
    let echelle = 2;
    let rect1 = Rectangle {
        largeur: dbg!(30 * echelle),
        hauteur: 50,
    };

    dbg!(&rect1);
}
```

Nous pouvons placer le `dbg!` autour de l'expression `30 * echelle` et, comme `dbg!` retourne la possession de la valeur issue de l'expression, le champ `largeur` va avoir la même valeur que si nous n'avions pas appelé `dbg!` ici. Nous ne voulons pas que `dbg!` prenne possession de `rect1`, donc nous donnons une référence à `rect1` lors de son prochain appel. Voici à quoi ressemble la sortie de cet exemple :

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/rectangles`
[src/main.rs:10] 30 * echelle = 60
[src/main.rs:14] &rect1 = Rectangle {
    largeur: 60,
    hauteur: 50,
}
```

Nous pouvons constater que la première sortie provient de la ligne 10 de *src/main.rs*, où nous déboguons l'expression `30 * echelle`, et son résultat est 60 (le formattage de `Debug` pour les entiers est d'afficher uniquement sa valeur). L'appel à `dbg!` à la ligne 14 de *src/main.rs* affiche la valeur de `&rect1`, qui est une structure `Rectangle`. La macro `dbg!` peut être très utile lorsque vous essayez de comprendre ce que fait votre code !

En plus du trait `Debug`, Rust nous offre d'autres traits pour que nous puissions les utiliser avec l'attribut `derive` pour ajouter des comportements utiles à nos propres types. Ces traits et leurs comportements sont listés à l'annexe C. Nous expliquerons comment implémenter ces traits avec des comportements personnalisés et comment créer vos propres traits au chapitre 10. Il existe aussi de nombreux attributs autres que `derive` ; pour en savoir plus, consultez la section “Attributs” de la référence de Rust.

Notre fonction `aire` est très spécifique : elle ne fait que calculer l'aire d'un rectangle. Il

serait utile de lier un peu plus ce comportement à notre structure `Rectangle`, puisque cela ne fonctionnera pas avec un autre type. Voyons comment on peut continuer de remanier ce code en transformant la fonction `aire` en *méthode* `aire` définie sur notre type `Rectangle`.

La syntaxe des méthodes

Les *méthodes* sont similaires aux fonctions : nous les déclarons avec le mot-clé `fn` et un nom, elles peuvent avoir des paramètres et une valeur de retour, et elles contiennent du code qui est exécuté quand on la méthode est appellée depuis un autre endroit. Contrairement aux fonctions, les méthodes diffèrent des fonctions parce qu'elles sont définies dans le contexte d'une structure (ou d'une énumération ou d'un objet de trait, que nous aborderons respectivement aux chapitres 6 et 17) et que leur premier paramètre est toujours `self`, un mot-clé qui représente l'instance de la structure sur laquelle on appelle la méthode.

Définir des méthodes

Remplaçons la fonction `aire` qui prend une instance de `Rectangle` en paramètre par une méthode `aire` définie sur la structure `Rectangle`, comme dans l'encart 5-13.

Fichier : src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    largeur: u32,
    hauteur: u32,
}

impl Rectangle {
    fn aire(&self) -> u32 {
        self.largeur * self.hauteur
    }
}

fn main() {
    let rect1 = Rectangle { largeur: 30, hauteur: 50 };

    println!(
        "L'aire du rectangle est de {} pixels carrés.",
        rect1.aire()
    );
}
```

Encart 5-13 : Définition d'une méthode `aire` sur la structure `Rectangle`

Pour définir la fonction dans le contexte de `Rectangle`, nous démarrons un bloc `impl` (*implémentation*) pour `Rectangle`. Tout ce qui sera dans ce bloc `impl` sera lié au type `Rectangle`. Puis nous déplaçons la fonction `aire` entre les accolades du `impl` et nous

remplaçons le premier paramètre (et dans notre cas, le seul) par `self` dans la signature et dans tout le corps. Dans `main`, où nous avons appelé la fonction `aire` et passé `rect1` en argument, nous pouvons utiliser à la place la *syntaxe des méthodes* pour appeler la méthode `aire` sur notre instance de `Rectangle`. La syntaxe des méthodes se place après l'instance : on ajoute un point suivi du nom de la méthode et des parenthèses contenant les arguments s'il y en a.

Dans la signature de `aire`, nous utilisons `&self` à la place de `rectangle: &Rectangle`. Le `&self` est un raccourci pour `self: &Self`. Au sein d'un bloc `impl`, le type de `Self` est un alias pour le type sur lequel porte le `impl`. Les méthodes doivent avoir un paramètre `self` du type `Self` comme premier paramètre afin que Rust puisse vous permettre d'abréger en renseignant uniquement `self` en premier paramètre. Veuillez noter qu'il nous faut quand même utiliser le `&` devant le raccourci `self`, pour indiquer que cette méthode emprunte l'instance de `Self`, comme nous l'avions fait pour `rectangle: &Rectangle`. Les méthodes peuvent prendre possession de `self`, emprunter `self` de façon immuable comme nous l'avons fait ici, ou emprunter `self` de façon mutable, comme pour n'importe quel autre paramètre.

Nous avons choisi `&self` ici pour la même raison que nous avions utilisé `&Rectangle` quand il s'agissait d'une fonction ; nous ne voulons pas en prendre possession, et nous voulons seulement lire les données de la structure, pas les modifier. Si nous voulions que la méthode modifie l'instance sur laquelle on l'appelle, on utiliserait `&mut self` comme premier paramètre. Il est rare d'avoir une méthode qui prend possession de l'instance en utilisant uniquement `self` comme premier argument ; cette technique est généralement utilisée lorsque la méthode transforme `self` en quelque chose d'autre et que vous voulez empêcher le code appelant d'utiliser l'instance d'origine après la transformation.

En complément de l'application de la syntaxe des méthodes et ainsi de ne pas être obligé de répéter le type de `self` dans la signature de chaque méthode, la principale raison d'utiliser les méthodes plutôt que de fonctions est pour l'organisation. Nous avons mis tout ce qu'on pouvait faire avec une instance de notre type dans un bloc `impl` plutôt que d'imposer aux futurs utilisateurs de notre code à rechercher les fonctionnalités de `Rectangle` à divers endroits de la bibliothèque que nous fournissons.

Notez que nous pourions faire en sorte qu'une méthode porte le même nom qu'un des champs de la structure. Par exemple, nous pourions définir une méthode sur `Rectangle` qui s'appelle elle aussi `largeur` :

Fichier : src/main.rs

```
impl Rectangle {
    fn largeur(&self) -> bool {
        self.largeur > 0
    }
}

fn main() {
    let rect1 = Rectangle {
        largeur: 30,
        hauteur: 50,
    };

    if rect1.largeur() {
        println!("Le rectangle a une largeur non nulle ; elle vaut {}",
rect1.largeur);
    }
}
```

Ici, nous avons défini la méthode `largeur` pour qu'elle retourne `true` si la valeur dans le champ `largeur` est supérieur ou égal à 0, et `false` si la valeur est 0 : nous pouvons utiliser un champ à l'intérieur d'une méthode du même nom, pour n'importe quel usage. Dans le `main`, lorsque nous ajoutons des parenthèses après `rect1.largeur`, Rust comprend que nous parlons de la méthode `largeur`. Lorsque nous n'utilisons pas les parenthèses, Rust sait nous parlons du champ `largeur`.

Souvent, mais pas toujours, lorsque nous appellons une méthode avec le même nom qu'un champ, nous voulons qu'elle renvoie uniquement la valeur de ce champ et ne fasse rien d'autre. Ces méthodes sont appelées des *accesseurs*, et Rust ne les implémente pas automatiquement pour les champs des structures comme le font certains langages. Les accesseurs sont utiles pour rendre le champ privé mais rendre la méthode publique et ainsi donner un accès en lecture seule à ce champ dans l'API publique de ce type. Nous développerons les notions de publique et privé et comment définir un champ ou une méthode publique ou privée au chapitre 7.

Où est l'opérateur `->` ?

En C et en C++, deux opérateurs différents sont utilisés pour appeler les méthodes : on utilise `.` si on appelle une méthode directement sur l'objet et `->` si on appelle la méthode sur un pointeur vers l'objet et qu'il faut d'abord déréférencer le pointeur. En d'autres termes, si `objet` est un pointeur, `objet->methode()` est similaire à `(*objet).methode()`.

Rust n'a pas d'équivalent à l'opérateur `->` ; à la place, Rust a une fonctionnalité

appelée *référencement et déréférencement automatiques*. L'appel de méthodes est l'un des rares endroits de Rust où on retrouve ce comportement.

Voilà comment cela fonctionne : quand on appelle une méthode avec `objet.methode()`, Rust ajoute automatiquement le `&`, `&mut` ou `*` pour que `objet` corresponde à la signature de la méthode. Autrement dit, ces deux lignes sont identiques :

```
p1.distance(&p2);
(&p1).distance(&p2);
```

La première ligne semble bien plus propre. Ce comportement du (dé)référencement automatique fonctionne parce que les méthodes ont une cible claire : le type de `self`. Compte tenu du nom de la méthode et de l'instance sur laquelle elle s'applique, Rust peut déterminer de manière irréfutable si la méthode lit (`&self`), modifie (`&mut self`) ou consomme (`self`) l'instance. Le fait que Rust rend implicite l'emprunt pour les instances sur lesquelles on appelle les méthodes améliore significativement l'ergonomie de la possession.

Les méthodes avec davantage de paramètres

Entraînons-nous à utiliser des méthodes en implémentant une seconde méthode sur la structure `Rectangle`. Cette fois-ci, nous voulons qu'une instance de `Rectangle` prenne une autre instance de `Rectangle` et qu'on retourne `true` si le second `Rectangle` peut se dessiner intégralement à l'intérieur de `self` (le premier `Rectangle`) ; sinon, on renverra `false`. En d'autres termes, une fois qu'on aura défini la méthode `peut_contenir`, on veut pouvoir écrire le programme de l'encart 5-14.

Fichier : src/main.rs

```
fn main() {
    let rect1 = Rectangle {
        largeur: 30,
        hauteur: 50
    };
    let rect2 = Rectangle {
        largeur: 10,
        hauteur: 40
    };
    let rect3 = Rectangle {
        largeur: 60,
        hauteur: 45
    };

    println!("rect1 peut-il contenir rect2 ? {}", rect1.peut_contenir(&rect2));
    println!("rect1 peut-il contenir rect3 ? {}", rect1.peut_contenir(&rect3));
}
```

Encart 5-14 : Utilisation de la méthode `peut_contenir` qui reste à écrire

Et on s'attend à ce que le texte suivant s'affiche, puisque les deux dimensions de `rect2` sont plus petites que les dimensions de `rect1`, mais `rect3` est plus large que `rect1` :

```
rect1 peut-il contenir rect2 ? true
rect1 peut-il contenir rect3 ? false
```

Nous voulons définir une méthode, donc elle doit se trouver dans le bloc `impl Rectangle`. Le nom de la méthode sera `peut_contenir` et elle prendra une référence immuable vers un autre `Rectangle` en paramètre. On peut déterminer le type du paramètre en regardant le code qui appelle la méthode : `rect1.peut_contenir(&rect2)` prend en argument `&rect2`, une référence immuable vers `rect2`, une instance de `Rectangle`. Cela est logique puisque nous voulons uniquement lire `rect2` (plutôt que de la modifier, ce qui aurait nécessité une référence mutable) et nous souhaitons que `main` garde possession de `rect2` pour qu'on puisse le réutiliser après avoir appelé la méthode `peut_contenir`. La valeur de retour de `peut_contenir` sera un booléen et l'implémentation de la méthode vérifiera si la largeur et la hauteur de `self` sont respectivement plus grandes que la largeur et la hauteur de l'autre `Rectangle`. Ajoutons la nouvelle méthode `peut_contenir` dans le bloc `impl` de l'encart 5-13, comme le montre l'encart 5-15.

Fichier : src/main.rs

```
impl Rectangle {
    fn aire(&self) -> u32 {
        self.largeur * self.hauteur
    }

    fn peut_contenir(&self, autre: &Rectangle) -> bool {
        self.largeur > autre.largeur && self.hauteur > autre.hauteur
    }
}
```

Encart 5-15 : Implémentation de la méthode `peut_contenir` sur `Rectangle` qui prend une autre instance de `Rectangle` en paramètre

Lorsque nous exécutons ce code avec la fonction `main` de l'encart 5-14, nous obtenons l'affichage attendu. Les méthodes peuvent prendre plusieurs paramètres qu'on peut ajouter à la signature après le paramètre `self`, et ces paramètres fonctionnent de la même manière que les paramètres des fonctions.

Les fonctions associées

Toutes les fonctions définies dans un bloc `impl` s'appellent des *fonctions associées* car elles sont associées au type renseigné après le `impl`. Nous pouvons aussi y définir des fonctions associées qui n'ont pas de `self` en premier paramètre (et donc ce ne sont pas des méthodes) car elles n'ont pas besoin d'une instance du type sur lequel elles travaillent. Nous avons déjà utilisé une fonction comme celle-ci : la fonction `String::from` qui est définie sur le type `String`.

Les fonctions associées qui ne ne sont pas des méthodes sont souvent utilisées comme constructeurs qui vont retourner une nouvelle instance de la structure. Par exemple, on pourrait écrire une fonction associée qui prend une unique dimension en paramètre et l'utilise à la fois pour la largeur et pour la hauteur, ce qui rend plus aisé la création d'un `Rectangle` carré plutôt que d'avoir à indiquer la même valeur deux fois :

Fichier : src/main.rs

```
impl Rectangle {
    fn carre(cote: u32) -> Rectangle {
        Rectangle {
            largeur: cote,
            hauteur: cote
        }
    }
}
```

Pour appeler cette fonction associée, on utilise la syntaxe `::` avec le nom de la structure ; `let mon_carre = Rectangle::carre(3);` en est un exemple. Cette fonction est cloisonnée dans l'espace de noms de la structure : la syntaxe `::` s'utilise aussi bien pour les fonctions associées que pour les espaces de noms créés par des modules. Nous aborderons les modules au chapitre 7.

Plusieurs blocs `impl`

Chaque structure peut avoir plusieurs blocs `impl` . Par exemple, l'encart 5-15 est équivalent au code de l'encart 5-16, où chaque méthode est dans son propre bloc `impl` .

```
impl Rectangle {
    fn aire(&self) -> u32 {
        self.largeur * self.hauteur
    }
}

impl Rectangle {
    fn peut_contenir(&self, autre: &Rectangle) -> bool {
        self.largeur > autre.largeur && self.hauteur > autre.hauteur
    }
}
```

Encart 5-16 : Réécriture de l'encart 5-15 en utilisant plusieurs blocs `impl`

Il n'y a aucune raison de séparer ces méthodes dans plusieurs blocs `impl` dans notre exemple, mais c'est une syntaxe valide. Nous verrons un exemple de l'utilité d'avoir plusieurs blocs `impl` au chapitre 10, où nous aborderons les types génériques et les traits.

Résumé

Les structures vous permettent de créer des types personnalisés significatifs pour votre domaine. En utilisant des structures, on peut relier entre elles des données associées et nommer chaque donnée pour rendre le code plus clair. Dans des blocs `impl` , vous pouvez définir des fonctions qui sont associées à votre type, et les méthodes sont un genre de fonction associée qui vous permet de renseigner le comportement que doivent suivre les instances de votre structure.

Mais les structures ne sont pas le seul moyen de créer des types personnalisés : nous allons maintenant voir les énumérations de Rust, une fonctionnalité que vous pourrez bientôt ajouter à votre boîte à outils.

Les énumérations et le filtrage par motif

Dans ce chapitre, nous allons aborder les *énumérations*, aussi appelées *enums*. Les énumérations vous permettent de définir un type en énumérant ses *variantes* possibles. Pour commencer, nous allons définir et utiliser une énumération pour voir comment une énumération peut donner du sens aux données. Ensuite, nous examinerons une énumération particulièrement utile qui s'appelle `Option` et qui permet de décrire des situations où la valeur peut être soit quelque chose, soit rien. Ensuite, nous regarderons comment le filtrage par motif avec l'expression `match` peut faciliter l'exécution de codes différents pour chaque valeur d'une énumération. Enfin, nous analyserons pourquoi la construction `if let` est un autre outil commode et concis à disposition pour traiter les énumérations dans votre code.

Les énumérations sont des fonctionnalités présentes dans de nombreux langages, mais leurs aptitudes varient d'un langage à l'autre. Les énumérations de Rust sont plus proches des *types de données algébriques* des langages fonctionnels, comme F#, OCaml et Haskell.

Définir une énumération

Les énumérations permettent de définir des types de données personnalisés de manière différente que vous l'avez fait avec les structures. Imaginons une situation que nous voudrions exprimer avec du code et regardons pourquoi les énumérations sont utiles et plus appropriées que les structures dans ce cas. Disons que nous avons besoin de travailler avec des adresses IP. Pour le moment, il existe deux normes principales pour les adresses IP : la version quatre et la version six. Comme ce seront les seules possibilités d'adresse IP que notre programme va rencontrer, nous pouvons *énumérer* toutes les variantes possibles, d'où vient le nom de l'énumération.

N'importe quelle adresse IP peut être soit une adresse en version quatre, soit en version six, mais pas les deux en même temps. Cette propriété des adresses IP est appropriée à la structure de données d'énumérations, car une valeur de l'énumération ne peut être qu'une de ses variantes. Les adresses en version quatre et six sont toujours fondamentalement des adresses IP, donc elles doivent être traitées comme étant du même type lorsque le code travaille avec des situations qui s'appliquent à n'importe quelle sorte d'adresse IP.

Nous pouvons exprimer ce concept dans le code en définissant une énumération `SorteAdresseIp` et en listant les différentes sortes possibles d'adresses IP qu'elle peut avoir, `V4` et `V6`. Ce sont les variantes de l'énumération :

```
enum SorteAdresseIp {
    V4,
    V6,
}
```

`SorteAdresseIp` est maintenant un type de données personnalisé que nous pouvons utiliser n'importe où dans notre code.

Les valeurs d'énumérations

Nous pouvons créer des instances de chacune des deux variantes de `SorteAdresseIp` de cette manière :

```
    let quatre = SorteAdresseIp::V4;
    let six = SorteAdresseIp::V6;
```

Remarquez que les variantes de l'énumération sont dans un espace de nom qui se situe avant leur nom, et nous utilisons un double deux-points pour les séparer tous les deux. C'est utile car maintenant les deux valeurs `SorteAdresseIp::V4` et `SorteAdresseIp::V6`

sont du même type : `SorteAdresseIp`. Ensuite, nous pouvons, par exemple, définir une fonction qui accepte n'importe quelle `SorteAdresseIp` :

```
fn router(sorte_ip: SorteAdresseIp) { }
```

Et nous pouvons appeler cette fonction avec chacune des variantes :

```
router(SorteAdresseIp::V4);
router(SorteAdresseIp::V6);
```

L'utilisation des énumérations a encore plus d'avantages. En étudiant un peu plus notre type d'adresse IP, nous constatons que pour le moment, nous ne pouvons pas stocker *la donnée* de l'adresse IP ; nous savons seulement de quelle sorte elle est. Avec ce que vous avez appris au chapitre 5, vous pourriez être tenté de résoudre ce problème avec des structures comme dans l'encart 6-1.

```
enum SorteAdresseIp {
    V4,
    V6,
}

struct AdresseIp {
    sorte: SorteAdresseIp,
    adresse: String,
}

let local = AdresseIp {
    sorte: SorteAdresseIp::V4,
    adresse: String::from("127.0.0.1"),
};

let rebouclage = AdresseIp {
    sorte: SorteAdresseIp::V6,
    adresse: String::from("::1"),
};
```

Encart 6-1 : Stockage de la donnée et de la variante de `SorteAdresseIp` d'une adresse IP en utilisant une `struct`

Ainsi, nous avons défini une structure `AdresseIp` qui a deux champs : un champ `sorte` qui est du type `SorteAdresseIp` (l'énumération que nous avons définie précédemment) et un champ `adresse` qui est du type `String`. Nous avons deux instances de cette structure. La première est `local`, et a la valeur `SorteAdresseIp::V4` pour son champ `sorte`, associé à la donnée d'adresse qui est `127.0.0.1`. La seconde instance est `rebouclage`. Elle a comme valeur de champ `sorte` l'autre variante de `SorteAdresseIp`, `V6`, et a l'adresse `::1` qui lui est associée. Nous avons utilisé une structure pour relier ensemble la `sorte` et l'`adresse`,

donc maintenant la variante est liée à la valeur.

Cependant, suivre le même principe en utilisant uniquement une énumération est plus concis : plutôt que d'utiliser une énumération dans une structure, nous pouvons insérer directement la donnée dans chaque variante de l'énumération. Cette nouvelle définition de l'énumération `AdresseIp` indique que chacune des variantes `V4` et `V6` auront des valeurs associées de type `String` :

```
enum AdresseIp {
    V4(String),
    V6(String),
}

let local = AdresseIp::V4(String::from("127.0.0.1"));

let rebouclage = AdresseIp::V6(String::from("::1"));
```

Nous relions les données de chaque variante directement à l'énumération, donc il n'est pas nécessaire d'avoir une structure en plus. Ceci nous permet de voir plus facilement un détail de fonctionnement des énumérations : le nom de chaque variante d'énumération que nous définissons devient aussi une fonction qui construit une instance de l'énumération. Ainsi, `AdresseIp::V4()` est un appel de fonction qui prend une `String` en argument et qui retourne une instance du type `AdresseIp`. Nous obtenons automatiquement cette fonction de constructeur qui est définie lorsque nous définissons l'énumération.

Il y a un autre avantage à utiliser une énumération plutôt qu'une structure : chaque variante peut stocker des types différents, et aussi avoir une quantité différente de données associées. Les adresses IP version quatre vont toujours avoir quatre composantes numériques qui auront une valeur entre 0 et 255. Si nous voulions stocker les adresses `V4` avec quatre valeurs de type `u8` mais continuer à stocker les adresses `V6` dans une `String`, nous ne pourrions pas le faire avec une structure. Les énumérations permettent de faire cela facilement :

```
enum AdresseIp {
    V4(u8, u8, u8, u8),
    V6(String),
}

let local = AdresseIp::V4(127, 0, 0, 1);

let rebouclage = AdresseIp::V6(String::from("::1"));
```

Nous avons vu différentes manières de définir des structures de données pour enregistrer des adresses IP en version quatre et version six. Cependant, il s'avère que vouloir stocker des adresses IP et identifier de quelle sorte elles sont est si fréquent que la bibliothèque

standard a une définition que nous pouvons utiliser ! Analysons comment la bibliothèque standard a défini `IpAddr` (l'équivalent de notre `AdresseIp`) : nous retrouvons la même énumération et les variantes que nous avons définies et utilisées, mais stocke les données d'adresse dans des variantes dans deux structures différentes, qui sont définies chacune pour chaque variante :

```
struct Ipv4Addr {
    // -- code masqué ici --
}

struct Ipv6Addr {
    // -- code masqué ici --
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

Ce code montre comment vous pouvez insérer n'importe quel type de données dans une variante d'énumération : des chaînes de caractères, des nombres ou des structures, par exemple. Vous pouvez même y intégrer d'autres énumérations ! Par ailleurs, les types de la bibliothèque standard ne sont parfois pas plus compliqués que ce que vous pourriez inventer.

Notez aussi que même si la bibliothèque standard embarque une définition de `IpAddr`, nous pouvons quand même créer et utiliser notre propre définition de ce type sans avoir de conflit de nom car nous n'avons pas importé cette définition de la bibliothèque standard dans la portée. Nous verrons plus en détail comment importer les types dans la portée au chapitre 7.

Analysons un autre exemple d'une énumération dans l'encart 6-2 : celle-ci a une grande diversité de types dans ses variantes.

```
enum Message {
    Quitter,
    Deplacer { x: i32, y: i32 },
    Ecrire(String),
    ChangerCouleur(i32, i32, i32),
}
```

Encart 6-2 : Une énumération `Message` dont chaque variante stocke des valeurs de différents types et en différentes quantités

Cette énumération a quatre variantes avec des types différents :

- `Quitter` n'a pas du tout de donnée associée.
- `Deplacer` intègre une structure anonyme en son sein.
- `Ecrire` intègre une seule `String`.
- `ChangerCouleur` intègre trois valeurs de type `i32`.

Définir une énumération avec des variantes comme celles dans l'encart 6-2 ressemble à la définition de différentes sortes de structures, sauf que l'énumération n'utilise pas le mot-clé `struct` et que toutes les variantes sont regroupées ensemble sous le type `Message`. Les structures suivantes peuvent stocker les mêmes données que celles stockées par les variantes précédentes :

```
struct MessageQuitter; // une structure unité
struct MessageDeplacer {
    x: i32,
    y: i32,
}
struct MessageEcrire(String); // une structure tuple
struct MessageChangerCouleur(i32, i32, i32); // une structure tuple
```

Mais si nous utilisions les différentes structures, qui ont chacune leur propre type, nous ne pourrions pas définir facilement une fonction qui prend en paramètre toutes les sortes de messages, tel que nous pourrions le faire avec l'énumération `Message` que nous avons définie dans l'encart 6-2, qui est un seul type.

Il y a un autre point commun entre les énumérations et les structures : tout comme on peut définir des méthodes sur les structures en utilisant `impl`, on peut aussi définir des méthodes sur des énumérations. Voici une méthode appelée `appeler` que nous pouvons définir sur notre énumération `Message` :

```
impl Message {
    fn appeler(&self) {
        // le corps de la méthode sera défini ici
    }
}

let m = Message::Ecrire(String::from("hello"));
m.appeler();
```

Le corps de la méthode va utiliser `self` pour obtenir la valeur sur laquelle nous avons utilisé la méthode. Dans cet exemple, nous avons créé une variable `m` qui a la valeur `Message::Ecrire(String::from("hello"))`, et cela sera ce que `self` aura comme valeur dans le corps de la méthode `appeler` quand nous lancerons `m.appeler()`.

Regardons maintenant une autre énumération de la bibliothèque standard qui est très

utilisée et utile : `Option`.

L'énumération `Option` et ses avantages par rapport à la valeur null

Cette section étudie le cas de `Option`, qui est une autre énumération définie dans la bibliothèque standard. Le type `Option` décrit un scénario très courant où une valeur peut être soit quelque chose, soit rien du tout. Par exemple, si vous demandez le premier élément dans une liste non vide, vous devriez obtenir une valeur. Si vous demandez le premier élément d'une liste vide, vous ne devriez rien obtenir. Exprimer ce concept avec le système de types implique que le compilateur peut vérifier si vous avez géré tous les cas que vous pourriez rencontrer ; cette fonctionnalité peut éviter des bogues qui sont très courants dans d'autres langages de programmation.

La conception d'un langage de programmation est souvent pensée en fonction des fonctionnalités qu'on inclut, mais les fonctionnalités qu'on refuse sont elles aussi importantes. Rust n'a pas de fonctionnalité *null* qu'ont de nombreux langages. *Null* est une valeur qui signifie qu'il n'y a pas de valeur à cet endroit. Avec les langages qui utilisent null, les variables peuvent toujours être dans deux états : null ou non null.

Dans sa thèse de 2009 “Null References: The Billion Dollar Mistake” (les références nulles : l'erreur à un milliard de dollars), Tony Hoare, l'inventeur de null, a écrit ceci :

> Je l'appelle mon erreur à un milliard de dollars. À cette époque, je concevais le premier système de type complet pour des références dans un langage orienté objet. Mon objectif était de garantir que toutes les utilisations des références soient totalement sûres, et soient vérifiées automatiquement par le compilateur. Mais je n'ai pas pu résister à la tentation d'inclure la référence nulle, simplement parce que c'était si simple à implémenter. Cela a conduit à d'innombrables erreurs, vulnérabilités, et pannes systèmes, qui ont probablement causé un milliard de dollars de dommages au cours des quarante dernières années.

Le problème avec les valeurs nulles, c'est que si vous essayez d'utiliser une valeur nulle comme si elle n'était pas nulle, vous obtiendrez une erreur d'une façon ou d'une autre. Comme cette propriété nulle ou non nulle est omniprésente, il est très facile de faire cette erreur.

Cependant, le concept que null essaye d'exprimer reste utile : une valeur nulle est une valeur qui est actuellement invalide ou absente pour une raison ou une autre.

Le problème ne vient pas vraiment du concept, mais de son implémentation. C'est pourquoi

Rust n'a pas de valeurs nulles, mais il a une énumération qui décrit le concept d'une valeur qui peut être soit présente, soit absente. Cette énumération est `Option<T>`, et elle est définie dans la bibliothèque standard comme ci-dessous :

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

L'énumération `Option<T>` est tellement utile qu'elle est intégrée dans l'étape préliminaire ; vous n'avez pas besoin de l'importer explicitement dans la portée. Ses variantes sont aussi intégrées dans l'étape préliminaire : vous pouvez utiliser directement `Some` (*quelque chose*) et `None` (*rien*) sans les préfixer par `Option::`. L'énumération `Option<T>` reste une énumération normale, et `Some(T)` ainsi que `None` sont toujours des variantes de type `Option<T>`.

La syntaxe `<T>` est une fonctionnalité de Rust que nous n'avons pas encore abordée. Il s'agit d'un paramètre de type générique, et nous verrons la généricité plus en détail au chapitre 10. Pour le moment, dites-vous que ce `<T>` signifie que la variante `Some` de l'énumération `Option` peut stocker un élément de donnée de n'importe quel type, et que chaque type concret qui est utilisé à la place du `T` transforme tout le type `Option<T>` en un type différent. Voici quelques exemples d'utilisation de valeurs de `Option` pour stocker des types de nombres et des types de chaînes de caractères :

```
let un_nombre = Some(5);
let une_chaine = Some("une chaîne");

let nombre_absent: Option<i32> = None;
```

La variable `un_nombre` est du type `Option<i32>`. Mais la variable `une_chaine` est du type `Option<&str>`, qui est un tout autre type. Rust peut déduire ces types car nous avons renseigné une valeur dans la variante `Some`. Pour `nombre_absent`, Rust nécessite que nous annotions le type de tout le `Option` : le compilateur ne peut pas déduire le type qui devrait être stocké dans la variante `Some` à partir de la valeur `None`. Ici, nous avons renseigné à Rust que nous voulions que `nombre_absent` soit du type `Option<i32>`.

Lorsque nous avons une valeur `Some`, nous savons que la valeur est présente et que la valeur est stockée dans le `Some`. Lorsque nous avons une valeur `None`, en quelque sorte, cela veut dire la même chose que null : nous n'avons pas une valeur valide. Donc pourquoi obtenir `Option<T>` est meilleur que d'avoir null ?

En bref, comme `Option<T>` et `T` (où `T` représente n'importe quel type) sont de types

différents, le compilateur ne va pas nous autoriser à utiliser une valeur `Option<T>` comme si cela était bien une valeur valide. Par exemple, le code suivant ne se compile pas car il essaye d'additionner un `i8` et une `Option<i8>` :

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let somme = x + y;
```

Si nous lançons ce code, nous aurons un message d'erreur comme celui-ci :

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0277]: cannot add `Option<i8>` to `i8`
 --> src/main.rs:5:17
  |
5 |     let somme = x + y;
  |                   ^ no implementation for `i8 + Option<i8>`
  |
  = help: the trait `Add<Option<i8>>` is not implemented for `i8`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `enums` due to previous error
```

Intense ! Effectivement, ce message d'erreur signifie que Rust ne comprend pas comment additionner un `i8` et une `Option<i8>` , car ils sont de types différents. Quand nous avons une valeur d'un type comme `i8` avec Rust, le compilateur va s'assurer que nous avons toujours une valeur valide. Nous pouvons continuer en toute confiance sans avoir à vérifier que cette valeur n'est pas nulle avant de l'utiliser. Ce n'est que lorsque nous avons une `Option<i8>` (ou tout autre type de valeur avec lequel nous travaillons) que nous devons nous inquiéter de ne pas avoir de valeur, et le compilateur va s'assurer que nous gérons ce cas avant d'utiliser la valeur.

Autrement dit, vous devez convertir une `Option<T>` en `T` pour pouvoir faire avec elle des opérations du type `T` . Généralement, cela permet de résoudre l'un des problèmes les plus courants avec null : supposer qu'une valeur n'est pas nulle alors qu'en réalité, elle l'est.

Eliminer le risque que des valeurs nulles puissent être mal gérées vous aide à être plus confiant en votre code. Pour avoir une valeur qui peut potentiellement être nulle, vous devez l'indiquer explicitement en déclarant que le type de cette valeur est `Option<T>` . Ensuite, quand vous utiliserez cette valeur, il vous faudra gérer explicitement le cas où cette valeur est nulle. Si vous utilisez une valeur qui n'est pas une `Option<T>` , alors vous *pouvez* considérer que cette valeur ne sera jamais nulle sans prendre de risques. Il s'agit d'un choix de conception délibéré de Rust pour limiter l'omniprésence de null et augmenter la sécurité du code en Rust.

Donc, comment récupérer la valeur de type `T` d'une variante `Some` quand vous avez une valeur de type `Option<T>` afin de l'utiliser ? L'énumération `Option<T>` a un large choix de méthodes qui sont plus ou moins utiles selon les cas ; vous pouvez les découvrir dans sa documentation. Se familiariser avec les méthodes de `Option<T>` peut être très utile dans votre aventure avec Rust.

De manière générale, pour pouvoir utiliser une valeur de `Option<T>`, votre code doit gérer chaque variante. On veut que du code soit exécuté uniquement quand on a une valeur `Some(T)`, et que ce code soit autorisé à utiliser la valeur de type `T` à l'intérieur. On veut aussi qu'un autre code soit exécuté si on a une valeur `None`, et ce code n'aura pas de valeur de type `T` de disponible. L'expression `match` est une structure de contrôle qui fait bien ceci lorsqu'elle est utilisée avec les énumérations : elle va exécuter du code différent en fonction de quelle variante de l'énumération elle obtient, et ce code pourra utiliser la donnée présente dans la valeur correspondante.

La structure de contrôle de flux `match`

Rust a une structure de contrôle de flux très puissante appelée `match` qui vous permet de comparer une valeur avec une série de motifs et d'exécuter du code en fonction du motif qui correspond. Les motifs peuvent être constitués de valeurs littérales, de noms de variables, de jokers, parmi tant d'autres ; le chapitre 18 va couvrir tous les différents types de motifs et ce qu'ils font. Ce qui fait la puissance de `match` est l'expressivité des motifs et le fait que le compilateur vérifie que tous les cas possibles sont bien gérés.

Considérez l'expression `match` comme une machine à trier les pièces de monnaie : les pièces descendent le long d'une piste avec des trous de tailles différentes, et chaque pièce tombe dans le premier trou à sa taille qu'elle rencontre. De manière similaire, les valeurs parcourent tous les motifs dans un `match`, et au premier motif auquel la valeur “correspond”, la valeur va descendre dans le bloc de code correspondant afin d'être utilisée pendant son exécution. En parlant des pièces, utilisons-les avec un exemple qui utilise `match` ! Nous pouvons écrire une fonction qui prend en paramètre une pièce inconnue des États-Unis d'Amérique et qui peut, de la même manière qu'une machine à trier, déterminer quelle pièce c'est et retourner sa valeur en centimes, comme ci-dessous dans l'encart 6-3.

```
enum PieceUs {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn valeur_en_centimes(piece: PieceUs) -> u8 {
    match piece {
        PieceUs::Penny => 1,
        PieceUs::Nickel => 5,
        PieceUs::Dime => 10,
        PieceUs::Quarter => 25,
    }
}
```

Encart 6-3 : Une énumération et une expression `match` qui trie les variantes de l'énumération dans ses motifs

Décomposons le `match` dans la fonction `valeur_en_centimes`. En premier lieu, nous utilisons le mot-clé `match` suivi par une expression, qui dans notre cas est la valeur de `piece`. Cela ressemble beaucoup à une expression utilisée avec `if`, mais il y a une grosse différence : avec `if`, l'expression doit retourner une valeur booléenne, mais ici, elle retourne n'importe quel type. Dans cet exemple, `piece` est de type `PieceUs`, qui est l'énumération que nous avons définie à la première ligne.

Ensuite, nous avons les branches du `match`. Une branche a deux parties : un motif et du code. La première branche a ici pour motif la valeur `PieceUs::Penny` et ensuite l'opérateur `=>` qui sépare le motif et le code à exécuter. Le code dans ce cas est uniquement la valeur `1`. Chaque branche est séparée de la suivante par une virgule.

Lorsqu'une expression `match` est exécutée, elle compare la valeur de `piece` avec le motif de chaque branche, dans l'ordre. Si un motif correspond à la valeur, le code correspondant à ce motif est alors exécuté. Si ce motif ne correspond pas à la valeur, l'exécution passe à la prochaine branche, un peu comme dans une machine de tri de pièces. Nous pouvons avoir autant de branches que nécessaire : dans l'encart 6-3, notre `match` a quatre branches.

Le code correspondant à chaque branche est une expression, et la valeur qui résulte de l'expression dans la branche correspondante est la valeur qui sera retournée par l'expression `match`.

Habituellement, nous n'utilisons pas les accolades si le code de la branche correspondante est court, comme c'est le cas dans l'encart 6-3 où chaque branche retourne simplement une valeur. Si vous voulez exécuter plusieurs lignes de code dans une branche d'un `match`, vous devez utiliser les accolades. Par exemple, le code suivant va afficher “Un centime porte-bonheur !” à chaque fois que la méthode est appelée avec une valeur `PieceUs::Penny`, mais va continuer à retourner la dernière valeur du bloc, `1` :

```
fn valeur_en_centimes(piece: PieceUs) -> u8 {
    match piece {
        PieceUs::Penny => {
            println!("Un centime porte-bonheur !");
            1
        }
        PieceUs::Nickel => 5,
        PieceUs::Dime => 10,
        PieceUs::Quarter => 25,
    }
}
```

Des motifs reliés à des valeurs

Une autre fonctionnalité intéressante des branches de `match` est qu'elles peuvent se lier aux valeurs qui correspondent au motif. C'est ainsi que nous pouvons extraire des valeurs d'une variante d'énumération.

En guise d'exemple, changeons une de nos variantes d'énumération pour stocker une donnée à l'intérieur. Entre 1999 et 2008, les États-Unis d'Amérique ont frappé un côté des *quarters* (pièces de 25 centimes) avec des dessins différents pour chacun des 50 États. Les

autres pièces n'ont pas eu de dessins d'États, donc seul le *quarter* a cette valeur en plus. Nous pouvons ajouter cette information à notre `enum` en changeant la variante `Quarter` pour y ajouter une valeur `EtatUs` qui y sera stockée à l'intérieur, comme nous l'avons fait dans l'encart 6-4.

```
#[derive(Debug)] // pour pouvoir afficher l'État
enum EtatUs {
    Alabama,
    Alaska,
    // -- partie masquée ici --
}

enum PieceUs {
    Penny,
    Nickel,
    Dime,
    Quarter(EtatUs),
}
```

Encart 6-4 : Une énumération `PieceUs` dans laquelle la variante `Quarter` stocke en plus une valeur de type `EtatUs`

Imaginons qu'un de vos amis essaye de collectionner tous les *quarters* des 50 États. Pendant que nous trions notre monnaie en vrac par type de pièce, nous mentionnerons aussi le nom de l'État correspondant à chaque *quarter* de sorte que si notre ami ne l'a pas, il puisse l'ajouter à sa collection.

Dans l'expression `match` de ce code, nous avons ajouté une variable `etat` au motif qui correspond à la variante `PieceUs::Quarter`. Quand on aura une correspondance `PieceUs::Quarter`, la variable `etat` sera liée à la valeur de l'État de cette pièce. Ensuite, nous pourrons utiliser `etat` dans le code de cette branche, comme ceci :

```
fn valeur_en_centimes(piece: PieceUs) -> u8 {
    match piece {
        PieceUs::Penny => 1,
        PieceUs::Nickel => 5,
        PieceUs::Dime => 10,
        PieceUs::Quarter(etat) => {
            println!("Il s'agit d'un quarter de l'État de {:?} !", etat);
            25
        },
    }
}
```

Si nous appelons `valeur_en_centimes(PieceUs::Quarter(EtatUs::Alaska))`, `piece` vaudra `PieceUs::Quarter(EtatUs::Alaska)`. Quand nous comparons cette valeur avec

toutes les branches du `match`, aucune d'entre elles ne correspondra jusqu'à ce qu'on arrive à `PieceUs::Quarter(etat)`. À partir de ce moment, la variable `etat` aura la valeur `EtatUs::Alaska`. Nous pouvons alors utiliser cette variable dans l'expression `println!`, ce qui nous permet d'afficher la valeur de l'État à l'intérieur de la variante `Quarter` de l'énumération `PieceUs`.

Utiliser `match` avec `Option<T>`

Dans la section précédente, nous voulions obtenir la valeur interne `T` dans le cas de `Some` lorsqu'on utilisait `Option<T>` ; nous pouvons aussi gérer les `Option<T>` en utilisant `match` comme nous l'avons fait avec l'énumération `PieceUs` ! Au lieu de comparer des pièces, nous allons comparer les variantes de `Option<T>`, mais la façon d'utiliser l'expression `match` reste la même.

Disons que nous voulons écrire une fonction qui prend une `Option<i32>` et qui, s'il y a une valeur à l'intérieur, ajoute 1 à cette valeur. S'il n'y a pas de valeur à l'intérieur, la fonction retournera la valeur `None` et ne va rien faire de plus.

Cette fonction est très facile à écrire, grâce à `match`, et ressemblera à l'encart 6-5.

```
fn plus_un(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let cinq = Some(5);
let six = plus_un(cinq);
let none = plus_un(None);
```

Encart 6-5 : Une fonction qui utilise une expression `match` sur une `Option<i32>`

Examinons la première exécution de `plus_un` en détail. Lorsque nous appelons `plus_un(cinq)`, la variable `x` dans le corps de `plus_un` aura la valeur `Some(5)`. Ensuite, nous comparons cela à chaque branche du `match`.

```
        None => None,
```

La valeur `Some(5)` ne correspond pas au motif `None`, donc nous continuons à la branche suivante.

```
        Some(i) => Some(i + 1),
```

Est-ce que `Some(5)` correspond au motif `Some(i)` ? Bien sûr ! Nous avons la même variante. Le `i` va prendre la valeur contenue dans le `Some`, donc `i` prend la valeur `5`. Le code dans la branche du `match` est exécuté, donc nous ajoutons 1 à la valeur de `i` et nous créons une nouvelle valeur `Some` avec notre résultat `6` à l'intérieur.

Maintenant, regardons le second appel à `plus_un` dans l'encart 6-5, où `x` vaut `None`. Nous entrons dans le `match` et nous le comparons à la première branche.

```
        None => None,
```

Cela correspond ! Il n'y a pas de valeur à additionner, donc le programme s'arrête et retourne la valeur `None` qui est dans le côté droit du `=>`. Comme la première branche correspond, les autres branches ne sont pas comparées.

La combinaison de `match` et des énumérations est utile dans de nombreuses situations. Vous allez revoir de nombreuses fois ce schéma dans du code Rust : utiliser `match` sur une énumération, récupérer la valeur qu'elle renferme, et exécuter du code en fonction de sa valeur. C'est un peu délicat au début, mais une fois que vous vous y êtes habitué, vous regretterez de ne pas l'avoir dans les autres langages. Cela devient toujours l'outil préféré de ses utilisateurs.

Les `match` sont toujours exhaustifs

Il y a un autre point de `match` que nous devons aborder. Examinez cette version de notre fonction `plus_un` qui a un bogue et ne va pas se compiler :

```
fn plus_un(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

Nous n'avons pas géré le cas du `None`, donc ce code va générer un bogue. Heureusement, c'est un bogue que Rust sait gérer. Si nous essayons de compiler ce code, nous allons obtenir cette erreur :

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0004]: non-exhaustive patterns: `None` not covered
   --> src/main.rs:3:15
    |
3   |         match x {
    |               ^ pattern `None` not covered
    |
    = help: ensure that all possible cases are being handled, possibly by adding
wildcards or more match arms
    = note: the matched value is of type `Option<i32>`

For more information about this error, try `rustc --explain E0004`.
error: could not compile `enums` due to previous error
```

Rust sait que nous n'avons pas couvert toutes les possibilités et sait même quel motif nous avons oublié ! Les `match` de Rust sont *exhaustifs* : nous devons traiter toutes les possibilités afin que le code soit valide. C'est notamment le cas avec `Option<T>` : quand Rust nous empêche d'oublier de gérer explicitement le cas de `None`, il nous protège d'une situation où nous supposons que nous avons une valeur alors que nous pourrions avoir null, ce qui rend impossible l'erreur à un milliard de dollars que nous avons vue précédemment.

Les motifs génériques et le motif _

En utilisant les énumérations, nous pouvons aussi appliquer des actions spéciales pour certaines valeurs précises, mais une action par défaut pour toutes les autres valeurs. Imaginons que nous implémentons un jeu dans lequel, si vous obtenez une valeur de 3 sur un lancé de dé, votre joueur ne se déplace pas, mais à la place il obtient un nouveau chapeau fataisie. Si vous obtenez un 7, votre joueur perd son chapeau fantaisie. Pour toutes les autres valeurs, votre joueur se déplace de ce nombre de cases sur le plateau du jeu. Voici un `match` qui implémente cette logique, avec le résultat du lancé de dé codé en dur plutôt qu'issu d'une génération aléatoire, et toute la logique des autres fonctions sont des corps vides car leur implémentation n'est pas le sujet de cet exemple :

```
let jete_de_de = 9;
match jete_de_de {
    3 => ajouter_chapeau_fantaisie(),
    7 => enleve_chapeau_fantaisie(),
    autre => deplace_joueur(autre),
}

fn ajouter_chapeau_fantaisie() {}
fn enleve_chapeau_fantaisie() {}
fn deplace_joueur(nombre_cases: u8) {}
```

Dans les deux premières branches, les motifs sont les valeurs litérales 3 et 7. La dernière branche couvre toutes les autres valeurs possibles, le motif est la variable `autre`. Le code qui s'exécute pour la branche `autre` utilise la variable en la passant dans la fonction `deplacer_joueur`.

Ce code se compile, même si nous n'avons pas listé toutes les valeurs possibles qu'un `u8` puisse avoir, car le dernier motif va correspondre à toutes les valeurs qui ne sont pas spécifiquement listés. Ce motif générique répond à la condition qu'un `match` doive être exhaustif. Notez que nous devons placer la branche avec le motif générique en tout dernier, car les motifs sont évalués dans l'ordre. Rust va nous prévenir si nous ajoutons des branches après un motif générique car toutes ces autres branches ne seront jamais vérifiées !

Rust a aussi un motif que nous pouvons utiliser lorsque nous n'avons pas besoin d'utiliser la valeur dans le motif générique : `_`, qui est un motif spécial qui vérifie n'importe quelle valeur et ne récupère pas cette valeur. Ceci indique à Rust que nous n'allons pas utiliser la valeur, donc Rust ne va pas nous prévenir qu'il y a une variable non utilisée.

Changeons les règles du jeu pour que si nous obtenions autre chose qu'un 3 ou un 7, nous jetions à nouveau le dé. Nous n'avons pas besoin d'utiliser la valeur dans ce cas, donc nous pouvons changer notre code pour utiliser `_` au lieu de la variable `autre` :

```
let jete_de_de = 9;
match jete_de_de {
    3 => ajouter_chapeau_fantaisie(),
    7 => enleve_chapeau_fantaisie(),
    _ => relancer(),
}

fn ajouter_chapeau_fantaisie() {}
fn enleve_chapeau_fantaisie() {}
fn relancer() {}
```

Cet exemple répond bien aux critères d'exhaustivité car nous ignorons explicitement toutes les autres valeurs dans la dernière branche ; nous n'avons rien oublié.

Si nous changeons à nouveau les règles du jeu, afin que rien se passe si vous obtenez autre chose qu'un 3 ou un 7, nous pouvons exprimer cela en utilisant la valeur unité (le type tuple vide que nous avons cité dans une section précédente) dans le code de la branche `_` :

```
let jete_de_de = 9;
match jete_de_de {
    3 => ajouter_chapeau_fantaisie(),
    7 => enleve_chapeau_fantaisie(),
    _ => (),
}

fn ajouter_chapeau_fantaisie() {}
fn enleve_chapeau_fantaisie() {}
```

Ici, nous indiquons explicitement à Rust que nous n'allons pas utiliser d'autres valeurs qui ne correspondent pas à un motif des branches antérieures, et nous ne voulons lancer aucun code dans ce cas.

Il existe aussi d'autres motifs que nous allons voir dans le chapitre 18. Pour l'instant, nous allons voir l'autre syntaxe `if let`, qui peut se rendre utile dans des cas où l'expression `match` est trop verbeuse.

Une structure de contrôle concise : `if let`

La syntaxe `if let` vous permet de combiner `if` et `let` afin de gérer les valeurs qui correspondent à un motif donné, tout en ignorant les autres. Imaginons le programme dans l'encart 6-6 qui fait un `match` sur la valeur `Option<u8>` de la variable `une_valeur_u8` mais n'a besoin d'exécuter du code que si la valeur est la variante `Some` .

```
let une_valeur_u8 = Some(3u8);
match une_valeur_u8 {
    Some(max) => println!("Le maximum est réglé sur {}", max),
    _ => (),
}
```

Encart 6-6 : Un `match` qui n'exécute du code que si la valeur est `Some`

Si la valeur est un `Some` , nous affichons la valeur dans la variante `Some` en associant la valeur à la variable `max` dans le motif. Nous ne voulons rien faire avec la valeur `None` . Pour satisfaire l'expression `match` , nous devons ajouter `_ => ()` après avoir géré une seule variante, ce qui est du code inutile.

À la place, nous pourrions écrire le même programme de manière plus concise en utilisant `if let` . Le code suivant se comporte comme le `match` de l'encart 6-6 :

```
let une_valeur_u8 = Some(3u8);
if let Some(max) = une_valeur_u8 {
    println!("Le maximum est réglé sur {}", max);
}
```

La syntaxe `if let` prend un motif et une expression séparés par un signe égal. Elle fonctionne de la même manière qu'un `match` où l'expression est donnée au `match` et où le motif est sa première branche. Dans ce cas, le motif est `Some(max)` , et le `max` est associé à la valeur dans le `Some` . Nous pouvons ensuite utiliser `max` dans le corps du bloc `if let` de la même manière que nous avons utilisé `max` dans la branche correspondante au `match` . Le code dans le bloc `if let` n'est pas exécuté si la valeur ne correspond pas au motif.

Utiliser `if let` permet d'écrire moins de code, et de moins l'indenter. Cependant, vous perdez la vérification de l'exhaustivité qu'assure le `match` . Choisir entre `match` et `if let` dépend de la situation : à vous de choisir s'il vaut mieux être concis ou appliquer une vérification exhaustive.

Autrement dit, vous pouvez considérer le `if let` comme du sucre syntaxique pour un `match` qui exécute du code uniquement quand la valeur correspond à un motif donné et ignore toutes les autres valeurs.

Nous pouvons joindre un `else` à un `if let`. Le bloc de code qui va dans le `else` est le même que le bloc de code qui va dans le cas `_` avec l'expression `match`. Souvenez-vous de la définition de l'énumération `PieceUs` de l'encart 6-4, où la variante `Quarter` stockait aussi une valeur `EtatUs`. Si nous voulions compter toutes les pièces qui ne sont pas des *quarters* que nous voyons passer, tout en affichant l'État des *quarters*, nous pourrions le faire avec une expression `match` comme ceci :

```
    let mut compteur = 0;
    match piece {
        PieceUs::Quarter(etat) => println!("Il s'agit d'un quarter de l'État de
{:?} !", etat),
        _ => compteur += 1,
    }
```

Ou nous pourrions utiliser une expression `if let` / `else` comme ceci :

```
    let mut compteur = 0;
    if let PieceUs::Quarter(etat) = piece {
        println!("Il s'agit d'un quarter de l'État de {:?} !", etat);
    } else {
        compteur += 1;
    }
```

Si vous trouvez que votre programme est alourdi par l'utilisation d'un `match`, souvenez-vous que `if let` est aussi présent dans votre boite à outils Rust.

Résumé

Nous avons désormais appris comment utiliser les énumérations pour créer des types personnalisés qui peuvent faire partie d'un jeu de valeurs recensées. Nous avons montré comment le type `Option<T>` de la bibliothèque standard vous aide à utiliser le système de types pour éviter les erreurs. Lorsque les valeurs d'énumération contiennent des données, vous pouvez utiliser `match` ou `if let` pour extraire et utiliser ces valeurs, à choisir en fonction du nombre de cas que vous voulez gérer.

Vos programmes Rust peuvent maintenant décrire des concepts métier à l'aide de structures et d'énumérations. Créer des types personnalisés à utiliser dans votre API assure la sécurité des types : le compilateur s'assurera que vos fonctions ne reçoivent que des valeurs du type attendu.

Afin de fournir une API bien organisée, simple à utiliser et qui n'expose que ce dont vos utilisateurs auront besoin, découvrons maintenant les modules de Rust.

Gérer des projets grandissants avec les paquets, crates et modules

Lorsque vous commencerez à écrire des gros programmes, organiser votre code va devenir important car vous ne pourrez plus garder en tête l'intégralité de votre programme. En regroupant des fonctionnalités qui ont des points communs et en les séparant des autres fonctionnalités, vous clarifiez l'endroit où trouver le code qui implémente une fonctionnalité spécifique afin de pouvoir le relire ou le modifier.

Les programmes que nous avons écrits jusqu'à présent étaient dans un module au sein d'un seul fichier. À mesure que le projet grandit, vous pouvez organiser votre code en le découpant en plusieurs modules et ensuite en plusieurs fichiers. Un paquet peut contenir plusieurs crates binaires et accessoirement une crate de bibliothèque. À mesure qu'un paquet grandit, vous pouvez en extraire des parties dans des crates séparées qui deviennent des dépendances externes. Ce chapitre va aborder toutes ces techniques. Pour un projet de très grande envergure qui a des paquets interconnectés qui évoluent ensemble, Cargo propose les espaces de travail, que nous découvrirons dans une section du chapitre 14.

En plus de regrouper des fonctionnalités, les modules vous permettent d'encapsuler les détails de l'implémentation d'une opération : vous pouvez écrire du code puis l'utiliser comme une abstraction à travers l'interface de programmation publique (API) du code sans se soucier de connaître les détails de son implémentation. La façon dont vous écrivez votre code définit quelles parties sont publiques et donc utilisables par un autre code, et quelles parties sont des détails d'implémentation privés dont vous vous réservez le droit de modifier. C'est un autre moyen de limiter le nombre d'éléments de l'API pour celui qui l'utilise.

Un concept qui lui est associé est la portée : le contexte dans lequel le code est écrit a un jeu de noms qui sont définis comme “dans la portée”. Quand ils lisent, écrivent et compilent du code, les développeurs et les compilateurs ont besoin de savoir ce que tel nom désigne à tel endroit, et s'il s'agit d'une variable, d'une fonction, d'une structure, d'une énumération, d'un module, d'une constante, etc. Vous pouvez créer des portées et décider quels noms sont dans la portée ou non. Vous ne pouvez pas avoir deux entités avec le même nom dans la même portée ; cependant, des outils existent pour résoudre les conflits de nom.

Rust a de nombreuses fonctionnalités qui vous permettent de gérer l'organisation de votre code, grâce à ce que la communauté Rust appelle le *système de modules*. Ce système définit quels sont les éléments qui sont accessibles depuis l'extérieur de la bibliothèque (notion de privé ou public), ainsi que leur portée. Ces fonctionnalités comprennent :

- **les paquets :** une fonctionnalité de Cargo qui vous permet de compiler, tester, et partager des crates ;
- **les *crates* :** une arborescence de modules qui fournit une bibliothèque ou un exécutable ;
- **les modules** : utilisés avec le mot-clé `use`, ils vous permettent de contrôler l'organisation, la portée et la visibilité des chemins ;
- **les chemins :** une façon de nommer un élément, comme une structure, une fonction ou un module.

Dans ce chapitre, nous allons découvrir ces fonctionnalités, voir comment elles interagissent, et expliquer comment les utiliser pour gérer les portées. À l'issue de ce chapitre, vous aurez de solides connaissances sur le système de modules et vous pourrez travailler avec les portées comme un pro !

Les paquets et les crates

La première partie du système de modules que nous allons aborder concerne les paquets et les *crates*. Une crate est un binaire ou une bibliothèque. Pour la compiler, le compilateur Rust part d'un fichier source, la racine de la *crate*, à partir duquel est alors créé le *module racine* de votre *crate* (nous verrons les modules plus en détail dans la section suivante). Un *paquet* se compose d'une ou plusieurs crates qui fournissent un ensemble de fonctionnalités. Un paquet contient un fichier *Cargo.toml* qui décrit comment construire ces crates.

Il y a plusieurs règles qui déterminent ce qu'un paquet peut contenir. Il *doit* contenir au maximum une seule crate de bibliothèque. Il peut contenir autant de crates binaires que vous le souhaitez, mais il doit contenir au moins une crate (que ce soit une bibliothèque ou un binaire).

Découvrons ce qui se passe quand nous créons un paquet. D'abord, nous utilisons la commande `cargo new` :

```
$ cargo new mon-projet
     Created binary (application) `mon-projet` package
$ ls mon-projet
Cargo.toml
src
$ ls mon-projet/src
main.rs
```

Lorsque nous avons saisi la commande, Cargo a créé un fichier *Cargo.toml*, qui définit un paquet. Si on regarde le contenu de *Cargo.toml*, le fichier *src/main.rs* n'est pas mentionné car Cargo obéit à une convention selon laquelle *src/main.rs* est la racine de la crate binaire portant le même nom que le paquet. De la même façon, Cargo sait que si le dossier du paquet contient *src/lib.rs*, alors le paquet contient une crate de bibliothèque qui a le même nom que le paquet, et que *src/lib.rs* est sa racine. Cargo transmet les fichiers de la crate racine à `rustc` pour compiler la bibliothèque ou le binaire.

Dans notre cas, nous avons un paquet qui contient uniquement *src/main.rs*, ce qui veut dire qu'il contient uniquement une crate binaire qui s'appelle `mon-projet` . Si un paquet contient *src/main.rs* et *src/lib.rs*, il a deux crates : une binaire et une bibliothèque, chacune avec le même nom que le paquet. Un paquet peut avoir plusieurs crates binaires en ajoutant des fichiers dans le répertoire *src/bin* : chaque fichier sera une crate séparée.

Une crate regroupe plusieurs fonctionnalités associées ensemble dans une portée afin que les fonctionnalités soient faciles à partager entre plusieurs projets. Par exemple, la crate `rand` que nous avons utilisée dans le chapitre 2 nous permet de générer des nombres

aléatoires. Nous pouvons utiliser cette fonctionnalité dans notre propre projet en important la crate `rand` dans la portée de notre projet. Toutes les fonctionnalités fournies par la crate `rand` sont accessibles via le nom de la crate, `rand`.

Ranger une fonctionnalité d'une crate dans sa propre portée clarifie si une fonctionnalité précise est définie dans notre crate ou dans la crate `rand` et évite ainsi de potentiels conflits. Par exemple, la crate `rand` fournit un *trait* qui s'appelle `Rng`. Nous pouvons nous aussi définir une structure qui s'appelle `Rng` dans notre propre crate. Comme les fonctionnalités des crates sont dans la portée de leur propre espace de nom, quand nous ajoutons `rand` en dépendance, il n'y a pas d'ambiguïté pour le compilateur sur le nom `Rng`. Dans notre crate, il se réfère au `struct Rng` que nous avons défini. Nous accédons au *trait* `Rng` de la crate `rand` via `rand::Rng`.

Poursuivons et parlons maintenant du système de modules !

Définir des modules pour gérer la portée et la visibilité

Dans cette section, nous allons aborder les modules et les autres outils du système de modules, à savoir les *chemins* qui nous permettent de nommer les éléments ; l'utilisation du mot-clé `use` qui importe un chemin dans la portée ; et le mot-clé `pub` qui rend publics les éléments. Nous verrons aussi le mot-clé `as`, les paquets externes, et l'opérateur glob. Pour commencer, penchons-nous sur les modules !

Les *modules* nous permettent de regrouper le code d'une crate pour une meilleure lisibilité et pour la facilité de réutilisation. Les modules permettent aussi de gérer la *visibilité* des éléments, qui précise si un élément peut être utilisé à l'extérieur du module (*c'est public*) ou s'il est un constituant interne et n'est pas disponible pour une utilisation externe (*c'est privé*).

Voici un exemple : écrivons une crate de bibliothèque qui permet de simuler un restaurant. Nous allons définir les signatures des fonctions mais nous allons laisser leurs corps vides pour nous concentrer sur l'organisation du code, plutôt que de coder pour de vrai un restaurant.

Dans le secteur de la restauration, certaines parties d'un restaurant sont assimilées à la *salle à manger* et d'autres *aux cuisines*. La partie salle à manger est l'endroit où se trouvent les clients ; c'est l'endroit où les hôtes installent les clients, où les serveurs prennent les commandes et encaissent les clients, et où les barmans préparent des boissons. Dans la partie cuisines, nous retrouvons les chefs et les cuisiniers qui travaillent dans la cuisine, mais aussi les plongeurs qui nettoient la vaisselle et les gestionnaires qui s'occupent des tâches administratives.

Pour organiser notre crate de la même manière qu'un vrai restaurant, nous pouvons organiser les fonctions avec des modules imbriqués. Créez une nouvelle bibliothèque qui s'appelle `restaurant` en utilisant `cargo new --lib restaurant` ; puis écrivez le code de l'encart 7-1 dans *src/lib.rs* afin de définir quelques modules et quelques signatures de fonctions.

Fichier : src/lib.rs

```
mod salle_a_manger {
    mod accueil {
        fn ajouter_a_la_liste_attente() {}

        fn installer_a_une_table() {}
    }

    mod service {
        fn prendre_commande() {}

        fn servir_commande() {}

        fn encaisser() {}
    }
}
```

Encart 7-1 : Un module `salle_a_manger` qui contient d'autres modules qui contiennent eux-mêmes des fonctions

Nous définissons un module en commençant avec le mot-clé `mod` et nous précisons ensuite le nom du module (dans notre cas, `salle_a_manger`) et nous ajoutons des accolades autour du corps du module. Dans les modules, nous pouvons avoir d'autres modules, comme dans notre cas avec les modules `accueil` et `service`. Les modules peuvent aussi contenir des définitions pour d'autres éléments, comme des structures, des énumérations, des constantes, des traits, ou des fonctions (comme c'est le cas dans l'encart 7-1).

Grâce aux modules, nous pouvons regrouper ensemble des définitions qui sont liées et donner un nom à ce lien. Les développeurs qui utiliseront ce code pourront plus facilement trouver les définitions dont ils ont besoin car ils peuvent parcourir le code en fonction des groupes plutôt que d'avoir à lire toutes les définitions. Les développeurs qui veulent rajouter des nouvelles fonctionnalités à ce code sauront maintenant où placer le code tout en gardant le programme organisé.

Précédemment, nous avons dit que *src/main.rs* et *src/lib.rs* étaient des racines de crates. Nous les appelons ainsi car le contenu de chacun de ces deux fichiers constituent un module qui s'appelle `crate` à la racine de *l'arborescence du module*.

L'encart 7-2 présente l'arborescence du module pour la structure de l'encart 7-1.

```
crate
 └── salle_a_manger
     ├── accueil
     │   ├── ajouter_a_la_liste_attente
     │   └── installer_a_une_table
     └── service
         ├── prendre_commande
         ├── servir_commande
         └── encaisser
```

Encart 7-2 : L'arborescence des modules pour le code de l'encart 7-1

Cette arborescence montre comment les modules sont imbriqués entre eux (par exemple, `accueil` est imbriqué dans `salle_a_manger`). L'arborescence montre aussi que certains modules sont les *frères* d'autres modules, ce qui veut dire qu'ils sont définis dans le même module (`accueil` et `service` sont définis dans `salle_a_manger`). Pour prolonger la métaphore familiale, si le module A est contenu dans le module B, on dit que le module A est *l'enfant* du module B et que ce module B est le *parent* du module A. Notez aussi que le module implicite nommé `crate` est le parent de toute cette arborescence.

L'arborescence des modules peut rappeler les dossiers du système de fichiers de votre ordinateur ; et c'est une excellente comparaison ! Comme les dossiers dans un système de fichiers, vous utilisez les modules pour organiser votre code. Et comme pour les fichiers dans un dossier, nous avons besoin d'un moyen de trouver nos modules.

Désigner un élément dans l'arborescence de modules

Pour indiquer à Rust où trouver un élément dans l'arborescence de modules, nous utilisons un chemin à l'instar des chemins que nous utilisons lorsque nous naviguons dans un système de fichiers. Si nous voulons appeler une fonction, nous avons besoin de connaître son chemin.

Il existe deux types de chemins :

- Un *chemin absolu* qui commence à partir de la racine de la crate en utilisant le nom d'une crate, ou le mot `crate`.
- Un *chemin relatif* qui commence à partir du module courant et qui utilise `self`, `super`, ou un identificateur à l'intérieur du module.

Les chemins absolus et relatifs sont suivis par un ou plusieurs identificateurs séparés par `::`.

Reprenons notre exemple de l'encart 7-1. Comment pouvons-nous appeler la fonction `ajouter_a_la_liste_attente` ? Cela revient à se demander : quel est le chemin de la fonction `ajouter_a_la_liste_attente` ? Dans l'encart 7-3, nous avons un peu simplifié notre code en enlevant quelques modules et quelques fonctions. Nous allons voir deux façons d'appeler la fonction `ajouter_a_la_liste_attente` à partir d'une nouvelle fonction `manger_au_restaurant` définie à la racine de la crate. La fonction `manger_au_restaurant` fait partie de l'API publique de notre crate de bibliothèque, donc nous la marquons avec le mot-clé `pub`. Dans la section "Exposer les chemins avec le mot-clé `pub`", nous en apprendrons plus sur `pub`. Notez que cet exemple ne se compile pas pour le moment ; nous allons l'expliquer un peu plus tard.

Fichier : src/lib.rs

```
mod salle_a_manger {
    mod accueil {
        fn ajouter_a_la_liste_attente() {}
    }
}

pub fn manger_au_restaurant() {
    // Chemin absolu
    crate::salle_a_manger::accueil::ajouter_a_la_liste_attente();

    // Chemin relatif
    salle_a_manger::accueil::ajouter_a_la_liste_attente();
}
```

Encart 7-3 : appel à la fonction `ajouter_a_la_liste_attente` en utilisant un chemin absolu

et relatif

Au premier appel de la fonction `ajouter_a_la_liste_attente` dans `manger_au_restaurant`, nous utilisons un chemin absolu. La fonction `ajouter_a_la_liste_attente` est définie dans la même crate que `manger_au_restaurant`, ce qui veut dire que nous pouvons utiliser le mot-clé `crate` pour démarrer un chemin absolu.

Après `crate`, nous ajoutons chacun des modules successifs jusqu'à `ajouter_a_la_liste_attente`. Nous pouvons faire l'analogie avec un système de fichiers qui aurait la même structure, où nous pourrions utiliser le chemin `/salle_a_manger/accueil/ajouter_a_la_liste_attente` pour lancer le programme `ajouter_a_la_liste_attente` ; utiliser le nom `crate` pour partir de la racine de la crate revient à utiliser `/` pour partir de la racine de votre système de fichiers dans votre invite de commande.

Lors du second appel à `ajouter_a_la_liste_attente` dans `manger_au_restaurant`, nous utilisons un chemin relatif. Le chemin commence par `salle_a_manger`, le nom du module qui est défini au même niveau que `manger_au_restaurant` dans l'arborescence de modules. Ici, l'équivalent en terme de système de fichier serait le chemin `salle_a_manger/accueil/ajouter_a_la_liste_attente`. Commencer par un nom signifie que le chemin est relatif.

Choisir entre utiliser un chemin relatif ou absolu sera une décision que vous ferez en fonction de votre projet. Le choix se fera en fonction de si vous êtes susceptible de déplacer la définition de l'élément souhaité séparément ou en même temps que le code qui l'utilise. Par exemple, si nous déplaçons le module `salle_a_manger` ainsi que la fonction `manger_au_restaurant` dans un module qui s'appelle `experience_client`, nous aurons besoin de mettre à jour le chemin absolu vers `ajouter_a_la_liste_attente`, mais le chemin relatif restera valide. Cependant, si nous avions déplacé uniquement la fonction `manger_au_restaurant` dans un module `repas` séparé, le chemin absolu de l'appel à `ajouter_a_la_liste_attente` restera le même, mais le chemin relatif aura besoin d'être mis à jour. Notre préférence est d'utiliser un chemin absolu car il est plus facile de déplacer les définitions de code et les appels aux éléments indépendamment les uns des autres.

Essayons de compiler l'encart 7-3 et essayons de comprendre pourquoi il ne se compile pas pour le moment ! L'erreur que nous obtenons est affichée dans l'encart 7-4.

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: module `accueil` is private
 --> src/lib.rs:9:28
  |
9 |     crate::salle_a_manger::accueil::ajouter_a_la_liste_attente();
  |                            ^^^^^^^ private module
  |
note: the module `accueil` is defined here
 --> src/lib.rs:2:5
  |
2 |     mod accueil {
  |     ^^^^^^^^^^^

error[E0603]: module `accueil` is private
  --> src/lib.rs:12:21
   |
12 |     salle_a_manger::accueil::ajouter_a_la_liste_attente();
   |                     ^^^^^^^ private module
   |
note: the module `accueil` is defined here
  --> src/lib.rs:2:5
   |
2  |     mod accueil {
   |     ^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

Encart 7-4 : les erreurs de compilation du code de l'encart 7-3

Le message d'erreur nous rappelle que ce module `accueil` est privé. Autrement dit, nous avons des chemins corrects pour le module `accueil` et pour la fonction `ajouter_a_la_liste_attente`, mais Rust ne nous laisse pas les utiliser car il n'a pas accès aux sections privées.

Les modules ne servent pas uniquement à organiser votre code. Ils définissent aussi les *limites de visibilité* de Rust : le code externe n'est pas autorisé à connaître, à appeler ou à se fier à des éléments internes au module. Donc, si vous voulez rendre un élément privé comme une fonction ou une structure, vous devez le placer dans un module.

La visibilité en Rust fait en sorte que tous les éléments (fonctions, méthodes, structures, énumérations, modules et constantes) sont privés par défaut. Les éléments dans un module parent ne peuvent pas utiliser les éléments privés dans les modules enfants, mais les éléments dans les modules enfants peuvent utiliser les éléments dans les modules parents. C'est parce que les modules enfants englobent et cachent les détails de leur implémentation, mais les modules enfants peuvent voir dans quel contexte ils sont définis. Pour continuer la métaphore du restaurant, considérez que les règles de visibilité de Rust

fonctionnent comme les cuisines d'un restaurant : ce qui s'y passe n'est pas connu des clients, mais les gestionnaires peuvent tout voir et tout faire dans le restaurant dans lequel ils travaillent.

Rust a décidé de faire fonctionner le système de modules de façon à ce que les détails d'implémentation interne sont cachés par défaut. Ainsi, vous savez quelles parties du code interne vous pouvez changer sans casser le code externe. Mais vous pouvez exposer aux parents des parties internes des modules enfants en utilisant le mot-clé `pub` afin de les rendre publiques.

Exposer des chemins avec le mot-clé `pub`

Retournons à l'erreur de l'encart 7-4 qui nous informe que le module `accueil` est privé. Nous voulons que la fonction `manger_au_restaurant` du module parent ait accès à la fonction `ajouter_a_la_liste_attente` du module enfant, donc nous utilisons le mot-clé `pub` sur le module `accueil`, comme dans l'encart 7-5.

Fichier : src/lib.rs

```
mod salle_a_manger {
    pub mod accueil {
        fn ajouter_a_la_liste_attente() {}
    }
}

pub fn manger_au_restaurant() {
    // Chemin absolu
    crate::salle_a_manger::accueil::ajouter_a_la_liste_attente();

    // Chemin relatif
    salle_a_manger::accueil::ajouter_a_la_liste_attente();
}
```

Encart 7-5 : utiliser `pub` sur le module `accueil` permet de l'utiliser dans `manger_au_restaurant`

Malheureusement, il reste une erreur dans le code de l'encart 7-5, la voici dans l'encart 7-6.

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `ajouter_a_la_liste_attente` is private
 --> src/lib.rs:9:37
  |
9 |     crate::salle_a_manger::accueil::ajouter_a_la_liste_attente();
  |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^ private
function
  |
note: the function `ajouter_a_la_liste_attente` is defined here
 --> src/lib.rs:3:9
  |
3 |         fn ajouter_a_la_liste_attente() {}
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error[E0603]: function `ajouter_a_la_liste_attente` is private
  --> src/lib.rs:12:30
   |
12 |     salle_a_manger::accueil::ajouter_a_la_liste_attente();
   |                              ^^^^^^^^^^^^^^^^^^^^^^^^^^ private function
   |
note: the function `ajouter_a_la_liste_attente` is defined here
  --> src/lib.rs:3:9
   |
3  |         fn ajouter_a_la_liste_attente() {}
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

Encart 7-6 : erreurs de compilation du code de l'encart 7-5

Que s'est-il passé ? Ajouter le mot-clé `pub` devant `mod accueil` rend public le module. Avec cette modification, si nous pouvons accéder à `salle_a_manger` , alors nous pouvons accéder à `accueil` . Mais le *contenu* de `accueil` reste privé ; rendre le module public ne rend pas son contenu public. Le mot-clé `pub` sur un module permet uniquement au code de ses parents d'y faire référence.

Les erreurs dans l'encart 7-6 nous informent que la fonction `ajouter_a_la_liste_attente` est privée. Les règles de visibilité s'appliquent aussi bien aux modules qu'aux structures, énumérations, fonctions et méthodes.

Rendons publique la fonction `ajouter_a_la_liste_attente` , en ajoutant le mot-clé `pub` devant sa définition, comme dans l'encart 7-7.

Fichier : src/lib.rs

```
mod salle_a_manger {
    pub mod accueil {
        pub fn ajouter_a_la_liste_attente() {}
    }
}

pub fn manger_au_restaurant() {
    // Chemin absolu
    crate::salle_a_manger::accueil::ajouter_a_la_liste_attente();

    // Chemin relatif
    salle_a_manger::accueil::ajouter_a_la_liste_attente();
}
```

Encart 7-7 : ajout du mot-clé `pub` devant `mod accueil` et `fn ajouter_a_la_liste_attente` pour nous permettre d'appeler la fonction à partir de `manger_au_restaurant`

Maintenant, le code va compiler ! Analysons les chemins relatif et absolu et vérifions pourquoi l'ajout du mot-clé `pub` nous permet d'utiliser ces chemins dans `ajouter_a_la_liste_attente` tout en respectant les règles de visibilité.

Dans le chemin absolu, nous commençons avec `crate`, la racine de l'arborescence de modules de notre crate. Ensuite, le module `salle_a_manger` est défini à la racine de la crate. Le module `salle_a_manger` n'est pas public, mais comme la fonction `manger_au_restaurant` est définie dans le même module que `salle_a_manger` (car `manger_au_restaurant` et `salle_a_manger` sont frères), nous pouvons utiliser `salle_a_manger` à partir de `manger_au_restaurant`. Ensuite, nous avons le module `accueil`, défini avec `pub`. Nous pouvons accéder au module parent de `accueil`, donc nous pouvons accéder à `accueil`. Enfin, la fonction `ajouter_a_la_liste_attente` est elle aussi définie avec `pub` et nous pouvons accéder à son module parent, donc au final cet appel à la fonction fonctionne bien !

Dans le chemin relatif, le fonctionnement est le même que le chemin absolu sauf pour la première étape : plutôt que de démarrer de la racine de la crate, le chemin commence à partir de `salle_a_manger`. Le module `salle_a_manger` est défini dans le même module que `manger_au_restaurant`, donc le chemin relatif qui commence à partir du module où est défini `manger_au_restaurant` fonctionne bien. Ensuite, comme `accueil` et `ajouter_a_la_liste_attente` sont définis avec `pub`, le reste du chemin fonctionne, et cet appel à la fonction est donc valide !

Commencer les chemins relatifs avec `super`

Nous pouvons aussi créer des chemins relatifs qui commencent à partir du module parent

en utilisant `super` au début du chemin. C'est comme débuter un chemin dans un système de fichiers avec la syntaxe `..`. Mais pourquoi voudrions-nous faire cela ?

Imaginons le code dans l'encart 7-8 qui représente le cas où le chef corrige une commande erronée et l'apporte personnellement au client pour s'excuser. La fonction `corriger_commande_erronee` appelle la fonction `servir_commande` en commençant le chemin de `servir_commande` avec `super` :

Fichier : src/lib.rs

```
fn servir_commande() {}

mod cuisines {
    fn corriger_commande_erronee() {
        cuisiner_commande();
        super::servir_commande();
    }

    fn cuisiner_commande() {}
}
```

Encart 7-8 : appel d'une fonction en utilisant un chemin relatif qui commence par `super`

La fonction `corriger_commande_erronee` est dans le module `cuisines`, donc nous pouvons utiliser `super` pour nous rendre au module parent de `cuisines`, qui dans notre cas est `crate`, la racine. De là, nous cherchons `servir_commande` et nous la trouvons. Avec succès ! Nous pensons que le module `cuisines` et la fonction `servir_commande` vont toujours garder la même relation et devrons être déplacés ensemble si nous réorganisons l'arborescence de modules de la crate. Ainsi, nous avons utilisé `super` pour avoir moins de code à mettre à jour à l'avenir si ce code est déplacé dans un module différent.

Rendre publiques des structures et des énumérations

Nous pouvons aussi utiliser `pub` pour déclarer des structures et des énumérations publiquement, mais il y a d'autres points à prendre en compte. Si nous utilisons `pub` avant la définition d'une structure, nous rendons la structure publique, mais les champs de la structure restent privés. Nous pouvons rendre chaque champ public ou non au cas par cas. Dans l'encart 7-9, nous avons défini une structure publique `cuisines::PetitDejeuner` avec un champ public `tartine_grillee` mais avec un champ privé `fruit_de_saison`. Cela simule un restaurant où le client peut choisir le type de pain qui accompagne le repas, mais le chef décide des fruits qui accompagnent le repas en fonction de la saison et ce qu'il y a en stock. Les fruits disponibles changent rapidement, donc les clients ne peuvent pas choisir le

fruit ou même voir quel fruit ils obtiendront.

Fichier : src/lib.rs

```
mod cuisines {
    pub struct PetitDejeuner {
        pub tartine_grillee: String,
        fruit_de_saison: String,
    }

    impl PetitDejeuner {
        pub fn en_ete(tartine_grillee: &str) -> PetitDejeuner {
            PetitDejeuner {
                tartine_grillee: String::from(tartine_grillee),
                fruit_de_saison: String::from("pêches"),
            }
        }
    }
}

pub fn manger_au_restaurant() {
    // On commande un petit-déjeuner en été avec tartine grillée au seigle
    let mut repas = cuisines::PetitDejeuner::en_ete("seigle");
    // On change d'avis sur le pain que nous souhaitons
    repas.tartine_grillee = String::from("blé");
    println!( "Je voudrais une tartine grillée au {}, s'il vous plaît.",
              repas.tartine_grillee);

    // La prochaine ligne ne va pas se compiler si nous ne la commentons pas,
    // car nous ne sommes pas autorisés à voir ou modifier le fruit de saison
    // qui accompagne le repas.

    // repas.fruit_de_saison = String::from("myrtilles");
}
```

Encart 7-9 : une structure avec certains champs publics et d'autres privés

Comme le champ `tartine_grillee` est public dans la structure `cuisines::PetitDejeuner`, nous pouvons lire et écrire dans le champ `tartine_grillee` à partir de `manger_au_restaurant` en utilisant `.`. Notez aussi que nous ne pouvons pas utiliser le champ `fruit_de_saison` dans `manger_au_restaurant` car `fruit_de_saison` est privé. Essayez de dé-commenter la ligne qui tente de modifier la valeur du champ `fruit_de_saison` et voyez l'erreur que vous obtenez !

Aussi, remarquez que comme `cuisines::PetitDejeuner` a un champ privé, la structure a besoin de fournir une fonction associée publique qui construit une instance de `PetitDejeuner` (que nous avons nommée `en_ete` ici). Si `PetitDejeuner` n'avait pas une fonction comme celle-ci, nous ne pourrions pas créer une instance de `PetitDejeuner` dans

`manger_au_restaurant` car nous ne pourrions pas donner une valeur au champ privé `fruit_de_saison` dans `manger_au_restaurant`.

Par contre, si nous rendons publique une énumération, toutes ses variantes seront publiques. Nous avons simplement besoin d'un `pub` devant le mot-clé `enum`, comme dans l'encart 7-10.

Fichier : src/lib.rs

```
mod cuisines {
    pub enum AmuseBouche {
        Soupe,
        Salade,
    }
}

pub fn manger_au_restaurant() {
    let commande1 = cuisines::AmuseBouche::Soupe;
    let commande2 = cuisines::AmuseBouche::Salade;
}
```

Encart 7-10 : on rend publique une énumération et cela rend aussi toutes ses variantes publiques

Comme nous rendons l'énumération `AmuseBouche` publique, nous pouvons utiliser les variantes `Soupe` et `Salade` dans `manger_au_restaurant`. Les énumérations ne sont pas très utiles si elles n'ont pas leurs variantes publiques ; et cela serait pénible d'avoir à marquer toutes les variantes de l'énumération avec `pub`, donc par défaut les variantes d'énumérations sont publiques. Les structures sont souvent utiles sans avoir de champs publics, donc les champs des structures sont tous privés par défaut, sauf si ces éléments sont marqués d'un `pub`.

Il y a encore une chose que nous n'avons pas abordée concernant `pub`, et c'est la dernière fonctionnalité du système de modules : le mot-clé `use`. Nous commencerons par parler de l'utilisation de `use` de manière générale, puis nous verrons comment combiner `pub` et `use`.

Importer des chemins dans la portée via le mot-clé `use`

Les chemins que nous avons écrits jusqu'ici peuvent paraître pénibles car trop longs et répétitifs. Par exemple, dans l'encart 7-7, que nous ayons choisi d'utiliser le chemin absolu ou relatif pour la fonction `ajouter_a_la_liste_attente`, nous aurions dû aussi écrire `salle_a_manger` et `accueil` à chaque fois que nous voulions appeler `ajouter_a_la_liste_attente`. Heureusement, il existe une solution pour simplifier ce cheminement. Nous pouvons importer un chemin dans la portée et appeler ensuite les éléments de ce chemin comme s'ils étaient locaux grâce au mot-clé `use`.

Dans l'encart 7-11, nous importons le module `crate::salle_a_manger::accueil` dans la portée de la fonction `manger_au_restaurant` afin que nous n'ayons plus qu'à utiliser `accueil::ajouter_a_la_liste_attente` pour appeler la fonction `ajouter_a_la_liste_attente` dans `manger_au_restaurant`.

Fichier : src/lib.rs

```
mod salle_a_manger {
    pub mod accueil {
        pub fn ajouter_a_la_liste_attente() {}
    }
}

use crate::salle_a_manger::accueil;

pub fn manger_au_restaurant() {
    accueil::ajouter_a_la_liste_attente();
    accueil::ajouter_a_la_liste_attente();
    accueil::ajouter_a_la_liste_attente();
}
```

Encart 7-11 : importer un module dans la portée via `use`

Dans une portée, utiliser un `use` et un chemin s'apparente à créer un lien symbolique dans le système de fichier. Grâce à l'ajout de `use crate::salle_a_manger::accueil` à la racine de la crate, `accueil` est maintenant un nom valide dans cette portée, comme si le module `accueil` avait été défini à la racine de la crate. Les chemins importés dans la portée via `use` doivent respecter les règles de visibilité, tout comme les autres chemins.

Vous pouvez aussi importer un élément dans la portée avec `use` et un chemin relatif. L'encart 7-12 nous montre comment utiliser un chemin relatif pour obtenir le même résultat que l'encart 7-11.

Fichier : src/lib.rs

```
mod salle_a_manger {
    pub mod accueil {
        pub fn ajouter_a_la_liste_attente() {}
    }
}

use salle_a_manger::accueil;

pub fn manger_au_restaurant() {
    accueil::ajouter_a_la_liste_attente();
    accueil::ajouter_a_la_liste_attente();
    accueil::ajouter_a_la_liste_attente();
}
```

Encart 7-12 : importer un module dans la portée avec `use` et un chemin relatif

Créer des chemins idéaux pour `use`

Dans l'encart 7-11, vous vous êtes peut-être demandé pourquoi nous avions utilisé `use crate::salle_a_manger::accueil` et appelé ensuite `accueil::ajouter_a_la_liste_attente` dans `manger_au_restaurant` plutôt que d'écrire le chemin du `use` jusqu'à la fonction `ajouter_a_la_liste_attente` pour avoir le même résultat, comme dans l'encart 7-13.

Fichier : src/lib.rs

```
mod salle_a_manger {
    pub mod accueil {
        pub fn ajouter_a_la_liste_attente() {}
    }
}

use crate::salle_a_manger::accueil::ajouter_a_la_liste_attente;

pub fn manger_au_restaurant() {
    ajouter_a_la_liste_attente();
    ajouter_a_la_liste_attente();
    ajouter_a_la_liste_attente();
}
```

Encart 7-13 : importer la fonction `ajouter_a_la_liste_attente` dans la portée avec `use`, ce qui n'est pas idéal

Bien que l'encart 7-11 et 7-13 accomplissent la même tâche, l'encart 7-11 est la façon idéale d'importer une fonction dans la portée via `use`. L'import du module parent de la fonction dans notre portée avec `use` nécessite que nous ayons à préciser le module parent quand

nous appelons la fonction. Renseigner le module parent lorsque nous appelons la fonction précise clairement que la fonction n'est pas définie localement, tout en minimisant la répétition du chemin complet. Nous ne pouvons pas repérer facilement là où est défini `ajouter_a_la_liste_attente` dans l'encart 7-13.

Cela dit, lorsque nous importons des structures, des énumérations, et d'autres éléments avec `use`, il est idéal de préciser le chemin complet. L'encart 7-14 montre la manière idéale d'importer la structure `HashMap` de la bibliothèque standard dans la portée d'une crate binaire.

Fichier : src/main.rs

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

Encart 7-14 : import de `HashMap` dans la portée de manière idéale

Il n'y a pas de forte justification à cette pratique : c'est simplement une convention qui a germé, et les gens se sont habitués à lire et écrire du code Rust de cette façon.

Il y a une exception à cette pratique : nous ne pouvons pas utiliser l'instruction `use` pour importer deux éléments avec le même nom dans la portée, car Rust ne l'autorise pas. L'encart 7-15 nous montre comment importer puis utiliser deux types `Result` ayant le même nom mais dont les modules parents sont distincts.

Fichier : src/lib.rs

```
use std::fmt;
use std::io;

fn fonction1() -> fmt::Result {
    // -- partie masquée ici --
}

fn fonction2() -> io::Result<()> {
    // -- partie masquée ici --
}
```

Encart 7-15 : l'import de deux types ayant le même nom dans la même portée nécessite d'utiliser leurs modules parents.

Comme vous pouvez le constater, l'utilisation des modules parents permet de distinguer les

deux types `Result`. Si nous avions utilisé `use std::fmt::Result` et `use std::io::Result`, nous aurions deux types nommés `Result` dans la même portée et donc Rust ne pourrait pas comprendre lequel nous voudrions utiliser en demandant `Result`.

Renommer des éléments avec le mot-clé `as`

Il y a une autre solution au fait d'avoir deux types du même nom dans la même portée à cause de `use` : après le chemin, nous pouvons rajouter `as` suivi d'un nouveau nom local, ou alias, sur le type. L'encart 7-16 nous montre une autre façon d'écrire le code de l'encart 7-15 en utilisant `as` pour renommer un des deux types `Result`.

Fichier : src/lib.rs

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn fonction1() -> Result {
    // -- partie masquée ici --
}

fn fonction2() -> IoResult<()> {
    // -- partie masquée ici --
}
```

Encart 7-16 : renommer un type lorsqu'il est importé dans la portée, avec le mot-clé `as`

Dans la seconde instruction `use`, nous avons choisi `IoResult` comme nouveau nom du type `std::io::Result`, qui n'est plus en conflit avec le `Result` de `std::fmt` que nous avons aussi importé dans la portée. Les encarts 7-15 et 7-16 sont idéaux, donc le choix vous revient !

Réexporter des éléments avec `pub use`

Lorsque nous importons un élément dans la portée avec le mot-clé `use`, son nom dans la nouvelle portée est privé. Pour permettre au code appelant d'utiliser ce nom comme s'il était défini dans cette portée, nous pouvons associer `pub` et `use`. Cette technique est appelée *réexporter* car nous importons un élément dans la portée, mais nous rendons aussi cet élément disponible aux portées des autres.

L'encart 7-17 nous montre le code de l'encart 7-11 où le `use` du module racine a été remplacé par `pub use`.

Fichier : src/lib.rs

```
mod salle_a_manger {
    pub mod accueil {
        pub fn ajouter_a_la_liste_attente() {}
    }
}

pub use crate::salle_a_manger::accueil;

pub fn manger_au_restaurant() {
    accueil::ajouter_a_la_liste_attente();
    accueil::ajouter_a_la_liste_attente();
    accueil::ajouter_a_la_liste_attente();
}
```

Encart 7-17 : rendre un élément disponible pour n'importe quel code qui l'importera dans sa portée, avec `pub use`

Grâce à `pub use`, le code externe peut maintenant appeler la fonction `ajouter_a_la_liste_attente` en utilisant `accueil::ajouter_a_la_liste_attente`. Si nous n'avions pas utilisé `pub use`, la fonction `manger_au_restaurant` aurait pu appeler `accueil::ajouter_a_la_liste_attente` dans sa portée, mais le code externe n'aurait pas pu profiter de ce nouveau chemin.

Réexporter est utile quand la structure interne de votre code est différente de la façon dont les développeurs qui utilisent votre code se la représentent. Par exemple, dans cette métaphore du restaurant, les personnes qui font fonctionner le restaurant se structurent en fonction de la “salle à manger” et des “cuisines”. Mais les clients qui utilisent le restaurant ne vont probablement pas voir les choses ainsi. Avec `pub use`, nous pouvons écrire notre code selon une certaine organisation, mais l'exposer avec une organisation différente. En faisant ainsi, la bibliothèque est bien organisée autant pour les développeurs qui travaillent sur la bibliothèque que pour les développeurs qui utilisent la bibliothèque.

Utiliser des paquets externes

Dans le chapitre 2, nous avions développé un projet de jeu du plus ou du moins qui utilisait le paquet externe `rand` afin d'obtenir des nombres aléatoires. Pour pouvoir utiliser `rand` dans notre projet, nous avions ajouté cette ligne dans *Cargo.toml* :

Fichier : Cargo.toml

```
rand = "0.8.3"
```

L'ajout de `rand` comme dépendance dans *Cargo.toml* demande à Cargo de télécharger le paquet `rand` et toutes ses dépendances à partir de crates.io et rend disponible `rand` pour notre projet.

Ensuite, pour importer les définitions de `rand` dans la portée de notre paquet, nous avions ajouté une ligne `use` qui commence avec le nom de la crate, `rand`, et nous avions listé les éléments que nous voulions importer dans notre portée. Dans la section "Générer le nombre secret" du chapitre 2, nous avions importé le trait `Rng` dans la portée, puis nous avions appelé la fonction `rand::thread_rng` :

```
use rand::Rng;

fn main() {
    let nombre_secret = rand::thread_rng().gen_range(1..101);
}
```

Les membres de la communauté Rust ont mis à disposition de nombreux paquets sur crates.io, et utiliser l'un d'entre eux dans votre paquet implique toujours ces mêmes étapes : les lister dans le fichier *Cargo.toml* de votre paquet et utiliser `use` pour importer certains éléments de ces crates dans la portée.

Notez que la bibliothèque standard (`std`) est aussi une crate qui est externe à notre paquet. Comme la bibliothèque standard est livrée avec le langage Rust, nous n'avons pas à modifier le *Cargo.toml* pour y inclure `std` . Mais nous devons utiliser `use` pour importer les éléments qui se trouvent dans la portée de notre paquet. Par exemple, pour `HashMap` , nous pourrions utiliser cette ligne :

```
use std::collections::HashMap;
```

C'est un chemin absolu qui commence par `std` , le nom de la crate de la bibliothèque standard.

Utiliser des chemins imbriqués pour simplifier les grandes listes de `use`

Si vous utilisez de nombreux éléments définis dans une même crate ou dans un même module, lister chaque élément sur sa propre ligne prendra beaucoup d'espace vertical dans vos fichiers. Par exemple, ces deux instructions `use` , que nous avions dans le jeu du plus ou du moins dans l'encart 2-4, importaient des éléments de `std` dans la portée :

Fichier : src/main.rs

```
// -- partie masquée ici --
use std::cmp::Ordering;
use std::io;
// -- partie masquée ici --
```

À la place, nous pouvons utiliser des chemins imbriqués afin d'importer ces mêmes éléments dans la portée en une seule ligne. Nous pouvons faire cela en indiquant la partie commune du chemin, suivi d'un double deux-points, puis d'accolades autour d'une liste des éléments qui diffèrent entre les chemins, comme dans l'encart 7-18 :

Fichier : src/main.rs

```
// -- partie masquée ici --
use std::{cmp::Ordering, io};
// -- partie masquée ici --
```

Encart 7-18 : utiliser un chemin imbriqué pour importer plusieurs éléments avec le même préfixe dans la portée

Pour des programmes plus gros, importer plusieurs éléments dans la portée depuis la même crate ou module en utilisant des chemins imbriqués peut réduire considérablement le nombre de `use` utilisés !

Nous pouvons utiliser un chemin imbriqué à tous les niveaux d'un chemin, ce qui peut être utile lorsqu'on utilise deux instructions `use` qui partagent un sous-chemin. Par exemple, l'encart 7-19 nous montre deux instructions `use` : une qui importe `std::io` dans la portée et une autre qui importe `std::io::Write` dans la portée.

Fichier : src/lib.rs

```
use std::io;
use std::io::Write;
```

Encart 7-19 : deux instructions `use` où l'une est un sous-chemin de l'autre

La partie commune entre ces deux chemins est `std::io` , et c'est le premier chemin complet. Pour imbriquer ces deux chemins en une seule instruction `use` , nous pouvons utiliser `self` dans le chemin imbriqué, comme dans l'encart 7-20.

Fichier : src/lib.rs

```
use std::io::{self, Write};
```

Encart 7-20 : imbrication des chemins de l'encart 7-19 dans une seule instruction `use`

Cette ligne importe `std::io` et `std::io::Write` dans la portée.

L'opérateur global

Si nous voulons importer, dans la portée, *tous* les éléments publics définis dans un chemin, nous pouvons indiquer ce chemin suivi par `*`, l'opérateur global :

```
use std::collections::*;
```

Cette instruction `use` va importer tous les éléments publics définis dans `std::collections` dans la portée courante. Mais soyez prudent quand vous utilisez l'opérateur global ! L'opérateur global rend difficile à dire quels éléments sont dans la portée et là où un élément utilisé dans notre programme a été défini.

L'opérateur global est souvent utilisé lorsque nous écrivons des tests, pour importer tout ce qu'il y a à tester dans le module `tests` ; nous verrons cela dans une section du chapitre 11. L'opérateur global est parfois aussi utilisé pour l'étape préliminaire : rendez-vous dans la documentation de la bibliothèque standard pour plus d'informations sur cela.

Séparer les modules dans différents fichiers

Jusqu'à présent, tous les exemples de ce chapitre ont défini plusieurs modules dans un seul fichier. Quand les modules vont grossir, vous allez probablement vouloir déplacer leurs définitions dans un fichier séparé pour faciliter le parcours de votre code.

Prenons par exemple le code de l'encart 7-17 et déplaçons le module `salle_a_manger` dans son propre fichier *src/salle_a_manger.rs* en changeant le fichier à la racine de la crate afin qu'il corresponde au code de l'encart 7-21. Dans notre cas, le fichier à la racine de la crate est *src/lib.rs*, mais cette procédure fonctionne aussi avec les crates binaires dans lesquelles le fichier à la racine de la crate est *src/main.rs*.

Fichier : src/lib.rs

```
mod salle_a_manger;

pub use crate::salle_a_manger::accueil;

pub fn manger_au_restaurant() {
    accueil::ajouter_a_la_liste_attente();
    accueil::ajouter_a_la_liste_attente();
    accueil::ajouter_a_la_liste_attente();
}
```

Encart 7-21 : Déclaration du module `salle_a_manger` dont le corps sera dans *src/salle_a_manger.rs*

Et *src/salle_a_manger.rs* contiendra la définition du corps du module `salle_a_manger`, comme dans l'encart 7-22.

Fichier : src/salle_a_manger.rs

```
pub mod accueil {
    pub fn ajouter_a_la_liste_attente() {}
}
```

Encart 7-22 : Les définitions à l'intérieur du module `salle_a_manger` dans *src/salle_a_manger.rs*

Utiliser un point-virgule après `mod salle_a_manger` plutôt que de créer un bloc indique à Rust de charger le contenu du module à partir d'un autre fichier qui porte le même nom que le module. Pour continuer avec notre exemple et déplacer également le module `accueil` dans son propre fichier, nous modifions *src/salle_a_manger.rs* pour avoir uniquement la déclaration du module `accueil` :

Fichier : src/salle_a_manger.rs

```
pub mod accueil;
```

Ensuite, nous créons un dossier *src/salle_a_manger* et un fichier *src/salle_a_manger/accueil.rs* qui contiendra les définitions du module `accueil` :

Fichier : src/salle_a_manger/accueil.rs

```
pub fn ajouter_a_la_liste_attente() {}
```

L'arborescence des modules reste identique, et les appels aux fonctions de `manger_au_restaurant` vont continuer à fonctionner sans aucune modification, même si les définitions se retrouvent dans des fichiers différents. Cette technique vous permet de déplacer des modules dans de nouveaux fichiers au fur et à mesure qu'ils s'agrandissent.

Remarquez que l'instruction `pub use crate::salle_a_manger::accueil` dans *src/lib.rs* n'a pas changé, et que `use` n'a aucun impact sur quels fichiers sont compilés pour constituer la crate. Le mot-clé `mod` déclare un module, et Rust recherche un fichier de code qui porte le nom dudit module.

Résumé

Rust vous permet de découper un paquet en plusieurs crates et une crate en modules afin que vous puissiez réutiliser vos éléments d'un module à un autre. Vous pouvez faire cela en utilisant des chemins absolus ou relatifs. Ces chemins peuvent être importés dans la portée avec l'instruction `use` pour pouvoir utiliser l'élément plusieurs fois dans la portée avec un chemin plus court. Le code du module est privé par défaut, mais vous pouvez rendre publiques des définitions en ajoutant le mot-clé `pub` .

Au prochain chapitre, nous allons nous intéresser à quelques collections de structures de données de la bibliothèque standard que vous pourrez utiliser dans votre code soigneusement organisé.

Les collections standard

La bibliothèque standard de Rust apporte quelques structures de données très utiles appelées *collections*. La plupart des autres types de données représentent une seule valeur précise, mais les collections peuvent contenir plusieurs valeurs. Contrairement aux tableaux et aux tuples, les données que ces collections contiennent sont stockées sur le tas, ce qui veut dire que la quantité de données n'a pas à être connue au moment de la compilation et peut augmenter ou diminuer pendant l'exécution du programme. Chaque type de collection a ses avantages et ses inconvénients, et en choisir un qui répond à votre besoin sur le moment est une aptitude que vous allez développer avec le temps. Dans ce chapitre, nous allons découvrir trois collections qui sont très utilisées dans les programmes Rust :

- Le *vecteur* qui vous permet de stocker un nombre variable de valeurs les unes à côté des autres.
- La *String*, qui est une collection de caractères. Nous avons déjà aperçu le type `String` précédemment, mais dans ce chapitre, nous allons l'étudier en détail.
- La *table de hachage* qui vous permet d'associer une valeur à une clé précise. C'est une implémentation spécifique d'une structure de données plus générique : le *tableau associatif*.

Pour en savoir plus sur les autres types de collections fournies par la bibliothèque standard, allez voir la documentation.

Nous allons voir comment créer et modifier les vecteurs, les Strings et les tables de hachage, et étudier leurs différences.

Stocker des listes de valeurs avec des vecteurs

Le premier type de collection que nous allons voir est `Vec<T>`, aussi appelé *vecteur*. Les vecteurs vous permettent de stocker plus d'une valeur dans une seule structure de données qui stocke les valeurs les unes à côté des autres dans la mémoire. Les vecteurs peuvent stocker uniquement des valeurs du même type. Ils sont utiles lorsque vous avez une liste d'éléments, tels que les lignes de texte provenant d'un fichier ou les prix des articles d'un panier d'achat.

Créer un nouveau vecteur

Pour créer un nouveau vecteur vide, nous appelons la fonction `Vec::new`, comme dans l'encart 8-1.

```
let v: Vec<i32> = Vec::new();
```

Encart 8-1 : création d'un nouveau vecteur vide pour y stocker des valeurs de type `i32`

Remarquez que nous avons ajouté ici une annotation de type. Comme nous n'ajoutons pas de valeurs dans ce vecteur, Rust ne sait pas quel type d'éléments nous souhaitons stocker. C'est une information importante. Les vecteurs sont implémentés avec la généricité ; nous verrons comment utiliser la généricité sur vos propres types au chapitre 10. Pour l'instant, sachez que le type `Vec<T>` qui est fourni par la bibliothèque standard peut stocker n'importe quel type. Lorsque nous créons un vecteur pour stocker un type précis, nous pouvons renseigner ce type entre des chevrons. Dans l'encart 8-1, nous précisons à Rust que le `Vec<T>` dans `v` va stocker des éléments de type `i32`.

Le plus souvent, vous allez créer un `Vec<T>` avec des valeurs initiales et Rust va deviner le type de la valeur que vous souhaitez stocker, donc vous n'aurez pas souvent besoin de faire cette annotation de type. Rust propose la macro très pratique `vec!`, qui va créer un nouveau vecteur qui stockera les valeurs que vous lui donnerez. L'encart 8-2 crée un nouveau `Vec<i32>` qui stocke les valeurs `1`, `2` et `3`. Le type d'entier est `i32` car c'est le type d'entier par défaut, comme nous l'avons évoqué dans la section "Les types de données" du chapitre 3.

```
let v = vec![1, 2, 3];
```

Encart 8-2 : création d'un nouveau vecteur qui contient des valeurs

Comme nous avons donné des valeurs initiales `i32`, Rust peut en déduire que le type de `v` est `Vec<i32>`, et l'annotation de type n'est plus nécessaire. Maintenant, nous allons voir

comment modifier un vecteur.

Modifier un vecteur

Pour créer un vecteur et ensuite lui ajouter des éléments, nous pouvons utiliser la méthode `push`, comme dans l'encart 8-3.

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

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

Encart 8-3 : utilisation de la méthode `push` pour ajouter des valeurs à un vecteur

Comme pour toute variable, si nous voulons pouvoir modifier sa valeur, nous devons la rendre mutable en utilisant le mot-clé `mut`, comme nous l'avons vu au chapitre 3. Les nombres que nous ajoutons dedans sont tous du type `i32`, et Rust le devine à partir des données, donc nous n'avons pas besoin de l'annotation `Vec<i32>`.

Libérer un vecteur libère aussi ses éléments

Comme toutes les autres structures, un vecteur est libéré quand il sort de la portée, comme précisé dans l'encart 8-4.

```
{
    let v = vec![1, 2, 3, 4];

    // on fait des choses avec v

} // <- v sort de la portée et est libéré ici
```

Encart 8-4 : mise en évidence de là où le vecteur et ses éléments sont libérés

Lorsque le vecteur est libéré, tout son contenu est aussi libéré, ce qui veut dire que les nombres entiers qu'il stocke vont être effacés de la mémoire. Cela semble très simple mais cela peut devenir plus compliqué quand vous commencez à utiliser des références vers les éléments du vecteur. Voyons ceci dès à présent !

Lire les éléments des vecteurs

Il existe deux façons de désigner une valeur enregistrée dans un vecteur : via les indices ou en utilisant la méthode `get`. Dans les exemples suivants, nous avons précisé les types des valeurs qui sont retournées par ces fonctions pour plus de clarté.

L'encart 8-5 nous montre les deux façons d'accéder à une valeur d'un vecteur, via la syntaxe d'indexation et avec la méthode `get`.

```
let v = vec![1, 2, 3, 4, 5];

let troisieme: &i32 = &v[2];
println!("Le troisième élément est {}", troisieme);

match v.get(2) {
    Some(troisieme) => println!("Le troisième élément est {}", troisieme),
    None => println!("Il n'y a pas de troisième élément."),
}
```

Encart 8-5 : utilisation de la syntaxe d'indexation ainsi que la méthode `get` pour accéder à un élément d'un vecteur

Il y a deux détails à remarquer ici. Premièrement, nous avons utilisé l'indice `2` pour obtenir le troisième élément car les vecteurs sont indexés par des nombres, qui commencent à partir de zéro. Deuxièmement, nous obtenons le troisième élément soit en utilisant `&` et `[]`, ce qui nous donne une référence, soit en utilisant la méthode `get` avec l'indice en argument, ce qui nous fournit une `Option<&T>`.

La raison pour laquelle Rust offre ces deux manières d'obtenir une référence vers un élement est de vous permettre de choisir le comportement du programme lorsque vous essayez d'utiliser une valeur dont l'indice est à l'extérieur de la plage des éléments existants. Par exemple, voyons dans l'encart 8-6 ce qui se passe lorsque nous avons un vecteur de cinq éléments et qu'ensuite nous essayons d'accéder à un élément à l'indice 100 avec chaque technique.

```
let v = vec![1, 2, 3, 4, 5];

let existe_pas = &v[100];
let existe_pas = v.get(100);
```

Encart 8-6 : tentative d'accès à l'élément à l'indice 100 dans un vecteur qui contient cinq éléments

Lorsque nous exécutons ce code, la première méthode `[]` va faire paniquer le programme car il demande un élément non existant. Cette méthode doit être favorisée lorsque vous souhaitez que votre programme plante s'il y a une tentative d'accéder à un élément après la fin du vecteur.

Lorsque nous passons un indice en dehors de l'intervalle du vecteur à la méthode `get`, elle retourne `None` sans paniquer. Vous devriez utiliser cette méthode s'il peut arriver occasionnellement de vouloir accéder à un élément en dehors de l'intervalle du vecteur en temps normal. Votre code va ensuite devoir gérer les deux valeurs `Some(&element)` ou `None`, comme nous l'avons vu au chapitre 6. Par exemple, l'indice peut provenir d'une saisie utilisateur. Si par accident il saisit un nombre qui est trop grand et que le programme obtient une valeur `None`, vous pouvez alors dire à l'utilisateur combien il y a d'éléments dans le vecteur courant et lui donner une nouvelle chance de saisir une valeur valide. Cela sera plus convivial que de faire planter le programme à cause d'une faute de frappe !

Lorsque le programme obtient une référence valide, le vérificateur d'emprunt va faire appliquer les règles de possession et d'emprunt (que nous avons vues au chapitre 4) pour s'assurer que cette référence ainsi que toutes les autres références au contenu de ce vecteur restent valides. Souvenez-vous de la règle qui dit que vous ne pouvez pas avoir des références mutables et immuables dans la même portée. Cette règle s'applique à l'encart 8-7, où nous obtenons une référence immuable vers le premier élément d'un vecteur et nous essayons d'ajouter un élément à la fin. Ce programme ne fonctionnera pas si nous essayons aussi d'utiliser cet élément plus tard dans la fonction :

```
let mut v = vec![1, 2, 3, 4, 5];

let premier = &v[0];

v.push(6);

println!("Le premier élément est : {}", premier);
```

Encart 8-7 : tentative d'ajout d'un élément à un vecteur alors que nous utilisons une référence à un élément

Compiler ce code va nous mener à cette erreur :

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as
immutable
 --> src/main.rs:6:5
  |
4 |     let premier = &v[0];
  |                    - immutable borrow occurs here
5 |
6 |     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("Le premier élément est : {}", premier);
  |                                             ------- immutable borrow later
used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `collections` due to previous error
```

Le code dans l'encart 8-7 semble pourtant marcher : pourquoi une référence au premier élément devrait se soucier de ce qui se passe à la fin du vecteur ? Cette erreur s'explique par la façon dont les vecteurs fonctionnent : comme les vecteurs ajoutent les valeurs les unes à côté des autres dans la mémoire, l'ajout d'un nouvel élément à la fin du vecteur peut nécessiter d'allouer un nouvel espace mémoire et copier tous les anciens éléments dans ce nouvel espace, s'il n'y a pas assez de place pour placer tous les éléments les uns à côté des autres dans la mémoire là où est actuellement stocké le vecteur. Dans ce cas, la référence au premier élément pointerait vers de la mémoire désallouée. Les règles d'emprunt évitent aux programmes de se retrouver dans cette situation.

Remarque : pour plus de détails sur l'implémentation du type `Vec<T>`, consultez le “Rustonomicon”.

Itérer sur les valeurs d'un vecteur

Pour accéder à chaque élément d'un vecteur chacun son tour, nous devrions itérer sur tous les éléments plutôt que d'utiliser individuellement les indices. L'encart 8-8 nous montre comment utiliser une boucle `for` pour obtenir des références immuables pour chacun des éléments dans un vecteur de `i32`, et les afficher.

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

Encart 8-8 : affichage de chaque élément d'un vecteur en itérant sur les éléments en utilisant une boucle `for`

Nous pouvons aussi itérer avec des références mutables pour chacun des éléments d'un vecteur mutable afin de modifier tous les éléments. La boucle `for` de l'encart 8-9 va ajouter `50` à chacun des éléments.

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

Encart 8-9 : itérations sur des références mutables vers des éléments d'un vecteur

Afin de changer la valeur vers laquelle pointe la référence mutable, nous devons utiliser l'opérateur de déréférencement `*` pour obtenir la valeur dans `i` avant que nous puissions utiliser l'opérateur `+=`. Nous verrons plus en détail l'opérateur de déréférencement dans une section du chapitre 15.

Utiliser une énumération pour stocker différents types

Les vecteurs ne peuvent stocker que des valeurs du même type. Cela peut être un problème ; il y a forcément des cas où on a besoin de stocker une liste d'éléments de types différents. Heureusement, les variantes d'une énumération sont définies sous le même type d'énumération, donc lorsque nous avons besoin d'un type pour représenter les éléments de types différents, nous pouvons définir et utiliser une énumération !

Par exemple, imaginons que nous voulions obtenir les valeurs d'une ligne d'une feuille de calcul dans laquelle quelques colonnes sont des entiers, d'autres des nombres à virgule flottante, et quelques chaînes de caractères. Nous pouvons définir une énumération dont les variantes vont avoir les différents types, et toutes les variantes de l'énumération seront du même type : celui de l'énumération. Ensuite, nous pouvons créer un vecteur pour stocker cette énumération et ainsi, au final, qui stocke différents types. La démonstration de cette technique est dans l'encart 8-10.

```
enum Cellule {
    Int(i32),
    Float(f64),
    Text(String),
}

let ligne = vec![
    Cellule::Int(3),
    Cellule::Text(String::from("bleu")),
    Cellule::Float(10.12),
];
```

Encart 8-10 : définition d'une `enum` pour stocker des valeurs de différents types dans un seul vecteur

Rust a besoin de savoir quel type de donnée sera stocké dans le vecteur au moment de la compilation afin de connaître la quantité de mémoire nécessaire pour stocker chaque élément sur le tas. Nous devons être précis sur les types autorisés dans ce vecteur. Si Rust avait permis qu'un vecteur stocke n'importe quel type, il y aurait pu avoir un risque qu'un ou plusieurs des types provoquent une erreur avec les manipulations effectuées sur les éléments du vecteur. L'utilisation d'une énumération ainsi qu'une expression `match` permet à Rust de garantir au moment de la compilation que tous les cas possibles sont traités, comme nous l'avons appris au chapitre 6.

Si vous n'avez pas une liste exhaustive des types que votre programme va stocker dans un vecteur, la technique de l'énumération ne va pas fonctionner. À la place, vous pouvez utiliser un objet trait, que nous verrons au chapitre 17.

Maintenant que nous avons vu les manières les plus courantes d'utiliser les vecteurs, prenez le temps de consulter la documentation de l'API pour découvrir toutes les méthodes très utiles définies dans la bibliothèque standard pour `Vec<T>`. Par exemple, en plus de `push`, nous avons une méthode `pop` qui retire et retourne le dernier élément. Intéressons-nous maintenant au prochain type de collection : la `String` !

Stocker du texte encodé en UTF-8 avec les Strings

Nous avons déjà parlé des chaînes de caractères dans le chapitre 4, mais nous allons à présent les analyser plus en détail. Les nouveaux Rustacés bloquent souvent avec les chaînes de caractères pour trois raisons : la tendance de Rust à prévenir les erreurs, le fait que les chaînes de caractères sont des structures de données plus compliquées que ne le pensent la plupart des développeurs, et l'UTF-8. Ces raisons cumulées rendent les choses compliquées lorsque vous venez d'un autre langage de programmation.

Nous avons présenté les chaînes de caractères comme des collections car les chaînes de caractères sont en réalité des suites d'octets, avec quelques méthodes supplémentaires qui sont utiles lorsque ces octets sont considérés comme du texte. Dans cette section, nous allons voir les points communs entre le fonctionnement des `String` et celui des autres collections, comme la création, la modification et la lecture. Nous verrons les raisons pour lesquelles les `String` sont différentes des autres collections, en particulier pourquoi l'indexation d'une `String` est compliquée à cause des différences entre la façon dont les gens et les ordinateurs interprètent les données d'une `String`.

Qu'est-ce qu'une chaîne de caractères ?

Nous allons d'abord définir ce que nous entendons par le terme *chaîne de caractères*. Rust a un seul type de chaînes de caractères dans le noyau du langage, qui est la slice de chaîne de caractères `str` qui est habituellement utilisée sous sa forme empruntée, `&str`. Dans le chapitre 4, nous avons abordé les *slices de chaînes de caractères*, qui sont des références à des données d'une chaîne de caractères encodée en UTF-8 qui sont stockées autre part. Les littéraux de chaînes de caractères, par exemple, sont stockés dans le binaire du programme et sont des slices de chaînes de caractères.

Le type `String`, qui est fourni par la bibliothèque standard de Rust plutôt que d'être intégré au noyau du langage, est un type de chaîne de caractères encodé en UTF-8 qui peut s'agrandir, être mutable, et être possédé. Lorsque les Rustacés parlent de “chaînes de caractères” en Rust, ils entendent soit le type `String`, soit le type de slice de chaînes de caractères `&str`, et non pas un seul de ces types. Bien que cette section traite essentiellement de `String`, ces deux types sont utilisés massivement dans la bibliothèque standard de Rust, et tous les deux sont encodés en UTF-8.

La bibliothèque standard de Rust apporte aussi un certain nombre d'autres types de chaînes de caractères, comme `OsString`, `OsStr`, `CString`, et `CStr`. Les crates de bibliothèque peuvent fournir encore plus de solutions pour stocker des chaînes de caractères. Avez-vous remarqué que ces noms finissent tous par `String` ou `Str` ? Cela fait référence aux

variantes possédées et empruntées, comme les types `String` et `str` que nous avons vus précédemment. Ces types de chaînes de caractères peuvent stocker leur texte dans de différents encodages, ou le stocker en mémoire de manière différente, par exemple. Nous n'allons pas traiter de ces autres types de chaînes de caractères dans ce chapitre ; référez-vous à la documentation de leur API pour en savoir plus sur leur utilisation et leur utilité.

Créer une nouvelle String

De nombreuses opérations disponibles avec `Vec<T>` sont aussi disponibles avec `String`, en commençant par la fonction `new` pour créer une `String`, utilisée dans l'encart 8-11.

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

Encart 8-11 : Création d'une nouvelle `String` vide

Cette ligne crée une nouvelle `String` vide qui s'appelle `s`, dans laquelle nous pouvons ensuite charger des données. Souvent, nous aurons quelques données initiales que nous voudrions ajouter dans la `String`. Pour cela, nous utilisons la méthode `to_string`, qui est disponible sur tous les types qui implémentent le trait `Display`, comme le font les littéraux de chaînes de caractères. L'encart 8-12 nous montre deux exemples.

```
let donnee = "contenu initial";

let s = donnee.to_string();

// cette méthode fonctionne aussi directement sur un
// littéral de chaîne de caractères :
let s = "contenu initial".to_string();
```

Encart 8-12 : Utilisation de la méthode `to_string` pour créer une `String` à partir d'un littéral de chaîne

Ce code crée une `String` qui contient `contenu initial`.

Nous pouvons aussi utiliser la fonction `String::from` pour créer une `String` à partir d'un littéral de chaîne. Le code dans l'encart 8-13 est équivalent au code dans l'encart 8-12 qui utilisait `to_string`.

```
let s = String::from("contenu initial");
```

Encart 8-13 : Utilisation de la fonction `String::from` afin de créer une `String` à partir d'un littéral de chaîne

Comme les chaînes de caractères sont utilisées pour de nombreuses choses, nous pouvons utiliser beaucoup d'API génériques pour les chaînes de caractères. Certaines d'entre elles peuvent paraître redondantes, mais elles ont toutes leur place ! Dans notre cas, `String::from` et `to_string` font la même chose, donc votre choix est une question de goût et de lisibilité.

Souvenez-vous que les chaînes de caractères sont encodées en UTF-8, donc nous pouvons y intégrer n'importe quelle donnée valide, comme nous le voyons dans l'encart 8-14.

```
let bonjour = String::from("السلام عليكم");
let bonjour = String::from("Dobrý den");
let bonjour = String::from("Hello");
let bonjour = String::from("שָׁלוֹם");
let bonjour = String::from("नमस्ते");
let bonjour = String::from("こんにちは");
let bonjour = String::from("안녕하세요");
let bonjour = String::from("你好");
let bonjour = String::from("Olá");
let bonjour = String::from("Здравствуйте");
let bonjour = String::from("Hola");
```

Encart 8-14 : Stockage de salutations dans différentes langues dans des chaînes de caractères

Toutes ces chaînes sont des valeurs `String` valides.

Modifier une `String`

Une `String` peut s'agrandir et son contenu peut changer, exactement comme le contenu d'un `Vec<T>`, si on y ajoute des données. De plus, vous pouvez aisément utiliser l'opérateur `+` ou la macro `format!` pour concaténer des valeurs `String`.

Ajouter du texte à une chaîne avec `push_str` et `push`

Nous pouvons agrandir une `String` en utilisant la méthode `push_str` pour ajouter une slice de chaîne de caractères, comme dans l'encart 8-15.

```
let mut s = String::from("foo");
s.push_str("bar");
```

Encart 8-15 : Ajout d'une slice de chaîne de caractères dans une `String` en utilisant la méthode `push_str`

À l'issue de ces deux lignes, `s` va contenir `foobar`. La méthode `push_str` prend une slice de chaîne de caractères car nous ne souhaitons pas forcément prendre possession du paramètre. Par exemple, dans le code de l'encart 8-16, nous voulons pouvoir utiliser `s2` après avoir ajouté son contenu dans `s1`.

```
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 est {}", s2);
```

Encart 8-16 : Utilisation d'une slice de chaîne de caractères après avoir ajouté son contenu dans une `String`

Si la méthode `push_str` prenait possession de `s2`, à la dernière ligne, nous ne pourrions pas afficher sa valeur. Cependant, ce code fonctionne comme nous l'espérions !

La méthode `push` prend un seul caractère en paramètre et l'ajoute à la `String`. L'encart 8-17 ajoute la lettre “l” à une `String` en utilisant la méthode `push`.

```
let mut s = String::from("lo");
s.push('l');
```

Encart 8-17 : Ajout d'un unique caractère à la valeur d'une `String` en utilisant `push`

Après l'exécution, `s` contiendra `lol`.

Concaténation avec l'opérateur + ou la macro `format!`

Souvent, vous aurez besoin de combiner deux chaînes de caractères existantes. Une façon de faire cela est d'utiliser l'opérateur `+`, comme dans l'encart 8-18.

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // notez que s1 a été déplacé ici
                   // et ne pourra plus être utilisé
```

Encart 8-18 : Utilisation de l'opérateur `+` pour combiner deux valeurs de `String`

La chaîne de caractères `s3` va contenir `Hello, world!`. La raison pour laquelle `s1` n'est plus utilisable après avoir été ajouté, et pour laquelle nous utilisons une référence vers `s2`, est la signature de la méthode qui est appelée lorsque nous utilisons l'opérateur `+`. L'opérateur `+` utilise la méthode `add`, dont la signature ressemble à ceci :

```
fn add(self, s: &str) -> String {
```

Dans la bibliothèque standard, vous pouvez constater que `add` est défini en utilisant des génériques. Ici, nous avons remplacé par des types concrets à la place des génériques, ce qui se passe lorsque nous utilisons cette méthode avec des valeurs de type `String`. Nous verrons la généricité au chapitre 10. Cette signature nous donne les éléments dont nous avons besoin pour comprendre les subtilités de l'opérateur `+`.

Premièrement, `s2` a un `&`, ce qui veut dire que nous ajoutons une *référence* vers la seconde chaîne de caractères à la première chaîne. C'est à cause du paramètre `s` de la fonction `add` : nous pouvons seulement ajouter un `&str` à une `String` ; nous ne pouvons pas ajouter deux valeurs de type `String` ensemble. Mais attendez — le type de `&s2` est `&String`, et non pas `&str`, comme c'est écrit dans le second paramètre de `add`. Alors pourquoi est-ce que le code de l'encart 8-18 se compile ?

La raison pour laquelle nous pouvons utiliser `&s2` dans l'appel à `add` est que le compilateur peut *extrapoler* l'argument `&String` en un `&str`. Lorsque nous appelons la méthode `add`, Rust va utiliser une *extrapolation de déréférencement*, qui transforme ici `&s2` en `&s2[..]`. Nous verrons plus en détail l'extrapolation de déréférencement au chapitre 15. Comme `add` ne prend pas possession du paramètre `s`, `s2` sera toujours une `String` valide après cette opération.

Ensuite, nous pouvons constater que la signature de `add` prend possession de `self`, car `self` n'a *pas* de `&`. Cela signifie que `s1` dans l'encart 8-18 va être déplacé dans l'appel à `add` et ne sera plus en vigueur après cela. Donc bien que `let s3 = s1 + &s2` semble copier les deux chaînes de caractères pour en créer une nouvelle, cette instruction va en réalité prendre possession de `s1`, y ajouter une copie du contenu de `s2` et nous redonner la possession du résultat. Autrement dit, cela semble faire beaucoup de copies mais en réalité non ; son implémentation est plus efficace que la copie.

Si nous avons besoin de concaténer plusieurs chaînes de caractères, le comportement de l'opérateur `+` devient difficile à utiliser :

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = s1 + "-" + &s2 + "-" + &s3;
```

Au final, `s` vaudra `tic-tac-toe`. Avec tous les caractères `+` et `"`, il est difficile de visualiser ce qui se passe. Pour une combinaison de chaînes de caractères plus complexe, nous pouvons utiliser à la place la macro `format!` :

```
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");

    let s = format!("{}-{}-{}", s1, s2, s3);
```

Ce code assigne lui aussi à `s` la valeur `tic-tac-toe`. La macro `format!` fonctionne comme `println!`, mais au lieu d'afficher son résultat à l'écran, elle retourne une `String` avec son contenu. La version du code qui utilise `format!` est plus facile à lire, et le code généré par la macro `format!` utilise des références afin qu'il ne prenne pas possession de ses paramètres.

L'indexation des Strings

Dans de nombreux autres langages de programmation, l'accès individuel aux caractères d'une chaîne de caractères en utilisant leur indice est une opération valide et courante. Cependant, si vous essayez d'accéder à des éléments d'une `String` en utilisant la syntaxe d'indexation avec Rust, vous allez avoir une erreur. Nous tentons cela dans le code invalide de l'encart 8-19.

```
    let s1 = String::from("hello");
    let h = s1[0];
```

Encart 8-19 : Tentative d'utilisation de la syntaxe d'indexation avec une `String`

Ce code va produire l'erreur suivante :

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0277]: the type `String` cannot be indexed by `{integer}`
 --> src/main.rs:3:13
  |
3 |     let h = s1[0];
  |             ^^^^^ `String` cannot be indexed by `{integer}`
  |
  = help: the trait `Index<{integer}>` is not implemented for `String`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `collections` due to previous error
```

L'erreur et la remarque nous expliquent le problème : les `String` de Rust n'acceptent pas l'utilisation des indices. Mais pourquoi ? Pour répondre à cette question, nous avons besoin de savoir comment Rust enregistre les chaînes de caractères dans la mémoire.

Représentation interne

Une `String` est une surcouche de `Vec<u8>`. Revenons sur certains exemples de chaînes de caractères correctement encodées en UTF-8 que nous avions dans l'encart 8-14. Premièrement, celle-ci :

```
let bonjour = String::from("Hola");
```

Dans ce cas-ci, `len` vaudra 4, ce qui veut dire que le vecteur qui stocke la chaîne “Hola” a une taille de 4 octets. Chacune des lettres prend 1 octet lorsqu'elles sont encodées en UTF-8. Cependant, la ligne suivante peut surprendre. (Notez que cette chaîne de caractères commence avec la lettre majuscule cyrillique Zé, et non pas le chiffre arabe 3.)

```
let bonjour = String::from("Здравствуйте");
```

Si on vous demandait la longueur de la chaîne de caractères, vous répondriez probablement 12. En réalité, la réponse de Rust sera 24 : c'est le nombre d'octets nécessaires pour encoder “Здравствуйте” en UTF-8, car chaque valeur scalaire Unicode dans cette chaîne de caractères prend 2 octets en mémoire. Par conséquent, un indice dans les octets de la chaîne de caractères ne correspondra pas forcément à une valeur scalaire Unicode valide. Pour démontrer cela, utilisons ce code Rust invalide :

```
let bonjour = "Здравствуйте";
let reponse = &bonjour[0];
```

Vous savez déjà que `reponse` ne vaudra pas `3`, la première lettre. Lorsqu'il est encodé en UTF-8, le premier octet de `3` est `208` et le second est `151`, donc on dirait que `reponse` vaudrait `208`, mais `208` n'est pas un caractère valide à lui seul. Retourner `208` n'est pas ce qu'un utilisateur attend s'il demande la première lettre de cette chaîne de caractères ; cependant, c'est la seule valeur que Rust a à l'indice 0 des octets. Les utilisateurs ne souhaitent généralement pas obtenir la valeur d'un octet, même si la chaîne de caractères contient uniquement des lettres latines : si `&"hello"[0]` était un code valide qui retournait la valeur de l'octet, il retournerait `104` et non pas `h`.

La solution est donc, pour éviter de retourner une valeur inattendue et générer des bogues qui ne seraient pas découverts immédiatement, que Rust ne va pas compiler ce code et va ainsi éviter des erreurs dès le début du processus de développement.

Des octets, des valeurs scalaires et des groupes de graphèmes !? Oh mon Dieu !

Un autre problème avec l'UTF-8 est qu'il a en fait trois manières pertinentes de considérer les chaînes de caractères avec Rust : comme des octets, comme des valeurs scalaires ou

comme des groupes de graphèmes (ce qui se rapproche le plus de ce que nous pourrions appeler des *lettres*).

Si l'on considère le mot hindi "नमस्ते" écrit en écriture devanagari, il est stocké comme un vecteur de valeurs `u8` qui sont les suivantes :

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164,
224, 165, 135]
```

Cela fait 18 octets et c'est ainsi que les ordinateurs stockeront cette donnée. Si nous les voyons comme des valeurs scalaires Unicode, ce qu'est le type `char` de Rust, ces octets seront les suivants :

```
['न', 'म', 'स', '्', 'त', 'े']
```

Nous avons six valeurs `char` ici, mais les quatrième et sixième valeurs ne sont pas des lettres : ce sont des signes diacritiques qui n'ont pas de sens employés seuls. Enfin, si nous les voyons comme des groupes de graphèmes, on obtient ce qu'on pourrait appeler les quatre lettres qui constituent le mot hindi :

```
["न", "म", "स्", "ते"]
```

Rust fournit différentes manières d'interpréter les données brutes des chaînes de caractères que les ordinateurs stockent afin que chaque programme puisse choisir l'interprétation dont il a besoin, peu importe la langue dans laquelle sont les données.

Une dernière raison pour laquelle Rust ne nous autorise pas à indexer une `String` pour récupérer un caractère est que les opérations d'indexation sont censées prendre un temps constant (O(1)). Mais il n'est pas possible de garantir cette performance avec une `String`, car Rust devrait parcourir le contenu depuis le début jusqu'à l'indice pour déterminer combien il y a de caractères valides.

Les slices de chaînes de caractères

L'indexation sur une chaîne de caractères est souvent une mauvaise idée car le type de retour de l'opération n'est pas toujours évident : un octet, un caractère, un groupe de graphèmes ou une slice de chaîne de caractères ? Si vous avez vraiment besoin d'utiliser des indices pour créer des slices de chaînes, Rust vous demande plus de précisions.

Plutôt que d'utiliser `[]` avec un nombre seul, vous pouvez utiliser `[]` avec un intervalle d'indices pour créer une slice de chaîne contenant des octets bien précis, plutôt que

d'utiliser `[]` avec un seul nombre :

```
let bonjour = "Здравствуйте";

let s = &bonjour[0..4];
```

Ici, `s` sera un `&str` qui contiendra les 4 premiers octets de la chaîne de caractères. Précédemment, nous avions mentionné que chacun de ces caractères était encodé sur 2 octets, ce qui veut dire que `s` vaudra `Зд`.

Si vous essayons de produire une slice d'une partie des octets d'un caractère avec quelquechose comme `&bonjour[0..1]`, Rust va paniquer au moment de l'exécution de la même façon que si nous utilisions un indice invalide pour accéder à un élément d'un vecteur :

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/collections`
thread 'main' panicked at 'byte index 1 is not a char boundary; it is inside 'З'
(bytes 0..2) of `Здравствуйте`', src/main.rs:4:14
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Vous devriez utiliser les intervalles pour créer des slices avec prudence, car cela peut provoquer un plantage de votre programme.

Les méthodes pour parcourir les chaînes de caractères

La meilleure manière de travailler sur des parties de chaînes de caractères est d'exprimer clairement si vous voulez travailler avec des caractères ou des octets. Pour les valeurs scalaires Unicode une par une, utilisez la méthode `chars`. Appeler `chars` sur “नमस्ते” sépare et retourne six valeurs de type `char`, et vous pouvez itérer sur le résultat pour accéder à chaque élément :

```
for c in "नमस्ते".chars() {
    println!("{}", c);
}
```

Ce code va afficher ceci :

न
म
स
ॅ
त
े

Aussi, la méthode `bytes` va retourner chaque octet brut, ce qui sera peut-être plus utile selon ce que vous voulez faire :

```
for b in "नमस्ते".bytes() {
    println!("{}", b);
}
```

Ce code va afficher les 18 octets qui constituent cette `String` :

```
224
164
// -- éléments masqués ici --
165
135
```

Rappelez-vous bien que des valeurs scalaires Unicode peuvent être constituées de plus d'un octet.

L'obtention des groupes de graphèmes à partir de chaînes de caractères est complexe, donc cette fonctionnalité n'est pas fournie par la bibliothèque standard. Des crates sont disponibles sur crates.io si c'est la fonctionnalité dont vous avez besoin.

Les chaînes de caractères ne sont pas si simples

Pour résumer, les chaînes de caractères sont complexes. Les différents langages de programmation ont fait différents choix sur la façon de présenter cette complexité aux développeurs. Rust a choisi d'appliquer par défaut la gestion rigoureuse des données de `String` pour tous les programmes Rust, ce qui veut dire que les développeurs doivent réfléchir davantage à la gestion des données UTF-8. Ce compromis révèle davantage la complexité des chaînes de caractères par rapport à ce que les autres langages de programmation laissent paraître, mais vous évite d'avoir à gérer plus tard dans votre cycle de développement des erreurs à cause de caractères non ASCII.

Passons maintenant à quelque chose de moins complexe : les tables de hachage !

Stocker des clés associées à des valeurs dans des tables de hachage

La dernière des collections les plus courantes est la *table de hachage (hash map)*. Le type `HashMap<K, V>` stocke une association de clés de type `K` à des valeurs de type `V` en utilisant une *fonction de hachage*, qui détermine comment elle va ranger ces clés et valeurs dans la mémoire. De nombreux langages de programmation prennent en charge ce genre de structure de données, mais elles ont souvent un nom différent, tel que hash, map, objet, table d'association, dictionnaire ou tableau associatif, pour n'en nommer que quelques-uns.

Les tables de hachage sont utiles lorsque vous voulez rechercher des données non pas en utilisant des indices, comme vous pouvez le faire avec les vecteurs, mais en utilisant une clé qui peut être de n'importe quel type. Par exemple, dans un jeu, vous pouvez consigner le score de chaque équipe dans une table de hachage dans laquelle chaque clé est le nom d'une équipe et la valeur est le score de l'équipe. Si vous avez le nom d'une équipe, vous pouvez récupérer son score.

Nous allons passer en revue l'API de base des tables de hachage dans cette section, mais bien d'autres fonctionnalités se cachent dans les fonctions définies sur `HashMap<K, V>` par la bibliothèque standard. Comme d'habitude, consultez la documentation de la bibliothèque standard pour plus d'informations.

Créer une nouvelle table de hachage

Une façon de créer une table de hachage vide est d'utiliser `new` et d'ajouter des éléments avec `insert`. Dans l'encart 8-20, nous consignons les scores de deux équipes qui s'appellent *Bleu* et *Jaune*. L'équipe Bleu commence avec 10 points, et l'équipe Jaune commence avec 50.

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Bleu"), 10);
scores.insert(String::from("Jaune"), 50);
```

Encart 8-20 : création d'une nouvelle table de hachage et insertion de quelques clés et valeurs

Notez que nous devons d'abord importer `HashMap` via `use` depuis la partie des collections de la bibliothèque standard. De nos trois collections courantes, cette dernière est la moins utilisée, donc elle n'est pas présente dans les fonctionnalités importées automatiquement

dans la portée par l'étape préliminaire. Les tables de hachage sont aussi moins gérées par la bibliothèque standard ; il n'y a pas de macro intégrée pour les construire, par exemple.

Exactement comme les vecteurs, les tables de hachage stockent leurs données sur le tas. Cette `HashMap` a des clés de type `String` et des valeurs de type `i32`. Et comme les vecteurs, les tables de hachage sont homogènes : toutes les clés doivent être du même type, et toutes les valeurs doivent aussi être du même type.

Une autre façon de construire une table de hachage est d'utiliser les itérateurs et la méthode `collect` sur un vecteur de tuples, où chaque tuple représente une clé et sa valeur. Nous aborderons en détail les itérateurs et leurs méthodes associées dans une section du chapitre 13. La méthode `collect` regroupe les données dans quelques types de collections, dont `HashMap`. Par exemple, si nous avions les noms des équipes et les scores initiaux dans deux vecteurs séparés, nous pourrions utiliser la méthode `zip` pour créer un itérateur de tuples où “Bleu” est associé à 10, et ainsi de suite. Ensuite, nous pourrions utiliser la méthode `collect` pour transformer cet itérateur de tuples en table de hachage, comme dans l'encart 8-21.

```
use std::collections::HashMap;

let equipes = vec![String::from("Bleu"), String::from("Jaune")];
let scores_initiaux = vec![10, 50];

let mut scores: HashMap<_, _> =
    equipes.into_iter().zip(scores_initiaux.into_iter()).collect();
```

Encart 8-21 : création d'une table de hachage à partir d'une liste d'équipes et d'une liste de scores

L'annotation de type `HashMap<_, _>` est nécessaire ici car `collect` peut générer plusieurs types de structures de données et Rust ne sait pas laquelle vous souhaitez si vous ne le précisez pas. Mais pour les paramètres qui correspondent aux types de clé et de valeur, nous utilisons des tirets bas, et Rust peut déduire les types que la table de hachage contient en fonction des types des données présentes dans les vecteurs. Dans l'encart 8-21, le type des clés sera `String` et le type des valeurs sera `i32`, comme dans l'encart 8-20.

Les tables de hachage et la possession

Pour les types qui implémentent le trait `Copy`, comme `i32`, les valeurs sont copiées dans la table de hachage. Pour les valeurs qui sont possédées comme `String`, les valeurs seront déplacées et la table de hachage sera la propriétaire de ces valeurs, comme démontré dans l'encart 8-22.

```
use std::collections::HashMap;

let nom_champ = String::from("Couleur favorite");
let valeur_champ = String::from("Bleu");

let mut table = HashMap::new();
table.insert(nom_champ, valeur_champ);
// nom_champ et valeur_champ ne sont plus en vigueur à partir d'ici,
// essayez de les utiliser et vous verrez l'erreur du compilateur que
// vous obtiendrez !
```

Encart 8-22 : démonstration que les clés et les valeurs sont possédées par la table de hachage une fois qu'elles sont insérées

Nous ne pouvons plus utiliser les variables `nom_champ` et `valeur_champ` après qu'elles ont été déplacées dans la table de hachage lors de l'appel à `insert`.

Si nous insérons dans la table de hachage des références vers des valeurs, ces valeurs ne seront pas déplacées dans la table de hachage. Les valeurs vers lesquelles les références pointent doivent rester en vigueur au moins aussi longtemps que la table de hachage. Nous verrons ces problématiques dans une section du chapitre 10.

Accéder aux valeurs dans une table de hachage

Nous pouvons obtenir une valeur d'une table de hachage en passant sa clé à la méthode `get`, comme dans l'encart 8-23.

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Bleu"), 10);
scores.insert(String::from("Jaune"), 50);

let nom_equipe = String::from("Bleu");
let score = scores.get(&nom_equipe);
```

Encart 8-23 : récupération du score de l'équipe `Bleu`, stocké dans la table de hachage

Dans notre cas, `score` aura la valeur qui est associée à l'équipe `Bleu`, et le résultat sera `Some(&10)`. Le résultat est encapsulé dans un `Some` car `get` retourne une `Option<&V>` : s'il n'y a pas de valeur pour cette clé dans la table de hachage, `get` va retourner `None`. Le programme doit gérer cette `Option` d'une des manières dont nous avons parlé au chapitre 6.

Nous pouvons itérer sur chaque paire de clé/valeur dans une table de hachage de la même manière que nous le faisons avec les vecteurs, en utilisant une boucle `for` :

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Bleu"), 10);
scores.insert(String::from("Jaune"), 50);

for (cle, valeur) in &scores {
    println!("{} : {}", cle, valeur);
}
```

Ce code va afficher chaque paire dans un ordre arbitraire :

```
Jaune : 50
Bleu : 10
```

Modifier une table de hachage

Bien que le nombre de paires de clé-valeur puisse augmenter, chaque clé ne peut être associée qu'à une seule valeur à la fois. Lorsque vous souhaitez modifier les données d'une table de hachage, vous devez choisir comment gérer le cas où une clé a déjà une valeur qui lui est associée. Vous pouvez remplacer l'ancienne valeur avec la nouvelle valeur, en ignorant complètement l'ancienne valeur. Vous pouvez garder l'ancienne valeur et ignorer la nouvelle valeur, en insérant la nouvelle valeur uniquement si la clé *n'a pas* déjà une valeur. Ou vous pouvez fusionner l'ancienne valeur et la nouvelle. Découvrons dès maintenant comment faire chacune de ces actions !

Réécrire une valeur

Si nous ajoutons une clé et une valeur dans une table de hachage et que nous ajoutons à nouveau la même clé avec une valeur différente, la valeur associée à cette clé sera remplacée. Même si le code dans l'encart 8-24 appelle deux fois `insert`, la table de hachage contiendra une seule paire de clé/valeur car nous ajoutons la valeur pour l'équipe `Bleu` à deux reprises.

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Bleu"), 10);
scores.insert(String::from("Bleu"), 25);

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

Encart 8-24 : remplacement d'une valeur stockée sous une clé spécifique

Ce code va afficher `{"Bleu": 25}`. La valeur initiale `10` a été remplacée.

Ajouter une valeur seulement si la clé n'a pas déjà de valeur

Il est courant de vérifier si une clé spécifique a déjà une valeur, et si ce n'est pas le cas, de lui associer une valeur. Les tables de hachage ont une API spécifique pour ce cas-là qui s'appelle `entry` et qui prend en paramètre la clé que vous voulez vérifier. La valeur de retour de la méthode `entry` est une énumération qui s'appelle `Entry` qui représente une valeur qui existe ou non. Imaginons que nous souhaitons vérifier si la clé pour l'équipe `Jaune` a une valeur qui lui est associée. Si ce n'est pas le cas, nous voulons lui associer la valeur 50, et faire de même pour l'équipe `Bleu`. En utilisant l'API `entry`, ce code va ressembler à l'encart 8-25.

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Bleu"), 10);

scores.entry(String::from("Jaune")).or_insert(50);
scores.entry(String::from("Bleu")).or_insert(50);

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

Encart 8-25 : utilisation de la méthode `entry` pour ajouter la clé uniquement si elle n'a pas déjà de valeur associée

La méthode `or_insert` sur `Entry` est conçue pour retourner une référence mutable vers la valeur correspondant à la clé du `Entry` si cette clé existe, et sinon, d'ajouter son paramètre comme nouvelle valeur pour cette clé et retourner une référence mutable vers la nouvelle valeur. Cette technique est plus propre que d'écrire la logique nous-mêmes et, de plus, elle fonctionne mieux avec le vérificateur d'emprunt.

L'exécution du code de l'encart 8-25 va afficher `{"Jaune": 50, "Bleu": 10}`. Le premier appel à `entry` va ajouter la clé pour l'équipe `Jaune` avec la valeur `50` car l'équipe `Jaune`

n'a pas encore de valeur. Le second appel à `entry` ne va pas changer la table de hachage car l'équipe `Bleu` a déjà la valeur `10`.

Modifier une valeur en fonction de l'ancienne valeur

Une autre utilisation courante avec les tables de hachage est de regarder la valeur d'une clé et ensuite la modifier en fonction de l'ancienne valeur. Par exemple, l'encart 8-26 contient du code qui compte combien de fois chaque mot apparaît dans du texte. Nous utilisons une table de hachage avec les mots comme clés et nous incrémentons la valeur pour compter combien de fois nous avons vu ce mot. Si c'est la première fois que nous voyons un mot, nous allons d'abord insérer la valeur `0`.

```
use std::collections::HashMap;

let texte = "bonjour le monde magnifique monde";

let mut table = HashMap::new();

for mot in texte.split_whitespace() {
    let compteur = table.entry(mot).or_insert(0);
    *compteur += 1;
}

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

Encart 8-26 : comptage des occurrences des mots en utilisant une table de hachage qui stocke les mots et leur quantité

Ce code va afficher `{"monde": 2, "bonjour": 1, "magnifique": 1, "le": 1}`. La méthode `split_whitespace` va itérer sur les sous-slices, séparées par des espaces vides, sur la valeur dans `texte`. La méthode `or_insert` retourne une référence mutable (`&mut V`) vers la valeur de la clé spécifiée. Nous stockons ici la référence mutable dans la variable `compteur`, donc pour affecter une valeur, nous devons d'abord déréférencer `compteur` en utilisant l'astérisque (`*`). La référence mutable sort de la portée à la fin de la boucle `for`, donc tous ces changements sont sûrs et autorisés par les règles d'emprunt.

Fonctions de hachage

Par défaut, `HashMap` utilise une fonction de hachage nommée SipHash qui résiste aux attaques par déni de service (DoS) envers les tables de hachage[1]. Ce n'est pas l'algorithme de hachage le plus rapide qui existe, mais le compromis entre une meilleure sécurité et la baisse de performances en vaut la peine. Si vous analysez la performance de votre code et

que vous vous rendez compte que la fonction de hachage par défaut est trop lente pour vos besoins, vous pouvez la remplacer par une autre fonction en spécifiant un hacheur différent. Un *hacheur* est un type qui implémente le trait `BuildHasher`. Nous verrons les traits et comment les implémenter au chapitre 10. Vous n'avez pas forcément besoin d'implémenter votre propre hacheur à partir de zéro ; crates.io héberge des bibliothèques partagées par d'autres utilisateurs de Rust qui fournissent de nombreux algorithmes de hachage répandus.

[1] https://en.wikipedia.org/wiki/SipHash

Résumé

Les vecteurs, Strings, et tables de hachage vont vous apporter de nombreuses fonctionnalités nécessaires à vos programmes lorsque vous aurez besoin de stocker, accéder, et modifier des données. Voici quelques exercices que vous devriez maintenant être en mesure de résoudre :

- À partir d'une liste d'entiers, utiliser un vecteur et retourner la médiane (la valeur au milieu lorsque la liste est triée) et le mode (la valeur qui apparaît le plus souvent ; une table de hachage sera utile dans ce cas) de la liste.
- Convertir des chaînes de caractères dans une variante du louchébem. La consonne initiale de chaque mot est remplacée par la lettre `l` et est rétablie à la fin du mot suivie du suffixe argotique “em” ; ainsi, “bonjour” devient “*l*onjour*bem*”. Si le mot commence par une voyelle, ajouter un `l` au début du mot et ajouter à la fin le suffixe “muche”. Et gardez en tête les détails à propos de l'encodage UTF-8 !
- En utilisant une table de hachage et des vecteurs, créez une interface textuelle pour permettre à un utilisateur d'ajouter des noms d'employés dans un département d'une entreprise. Par exemple, “Ajouter Sally au bureau d'études” ou “Ajouter Amir au service commercial”. Ensuite, donnez la possibilité à l'utilisateur de récupérer une liste de toutes les personnes dans un département, ou tout le monde dans l'entreprise trié par département, et classés dans l'ordre alphabétique dans tous les cas.

La documentation de l'API de la bibliothèque standard décrit les méthodes qu'ont les vecteurs, chaînes de caractères et tables de hachage, ce qui vous sera bien utile pour mener à bien ces exercices !

Nous nous lançons dans des programmes de plus en plus complexes dans lesquels les opérations peuvent échouer, c'est donc le moment idéal pour voir comment bien gérer les erreurs. C'est ce que nous allons faire au prochain chapitre !

La gestion des erreurs

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

Rust classe les erreurs dans deux catégories principales : les erreurs *récupérables* et *irrécupérables*. Pour les erreurs récupérables, comme l'erreur *le fichier n'a pas été trouvé*, nous préférons probablement signaler le problème à l'utilisateur et relancer l'opération. Les erreurs irrécupérables sont toujours des symptômes de bogues, comme par exemple essayer d'accéder à un élément en dehors de l'intervalle de données d'un tableau, et alors dans ce cas nous voulons arrêter immédiatement l'exécution du programme.

La plupart des langages de programmation ne font pas de distinction entre ces deux types d'erreurs et les gèrent de la même manière, en utilisant des fonctionnalités comme les exceptions. Rust n'a pas d'exception. À la place, il a les types `Result<T, E>` pour les erreurs récupérables, et la macro `panic!` qui arrête l'exécution quand le programme se heurte à des erreurs irrécupérables. Nous allons commencer ce chapitre par expliquer l'utilisation de `panic!`, puis nous allons voir les valeurs de retour `Result<T, E>`. Enfin, nous allons voir les éléments à prendre en compte pour décider si nous devons essayer de rattraper une erreur ou alors arrêter l'exécution.

Les erreurs irrécupérables avec `panic!`

Parfois, des choses se passent mal dans votre code, et vous ne pouvez rien y faire. Pour ces cas-là, Rust a la macro `panic!`. Quand la macro `panic!` s'exécute, votre programme va afficher un message d'erreur, dérouler et nettoyer la pile, et ensuite fermer le programme. Nous allons souvent faire paniquer le programme lorsqu'un bogue a été détecté, et qu"on ne sait comment gérer cette erreur au moment de l'écriture de notre programme.

Dérouler la pile ou abandonner suite à un `panic!`

Par défaut, quand un *panic* se produit, le programme se met à *dérouler*, ce qui veut dire que Rust retourne en arrière dans la pile et nettoie les données de chaque fonction qu'il rencontre sur son passage. Cependant, cette marche arrière et le nettoyage demandent beaucoup de travail. Toutefois, Rust vous permet de choisir l'alternative *d'abandonner* immédiatement, ce qui arrête le programme sans nettoyage. La mémoire qu'utilisait le programme devra ensuite être nettoyée par le système d'exploitation. Si dans votre projet vous avez besoin de construire un exécutable le plus petit possible, vous pouvez passer du déroulage à l'abandon lors d'un panic en ajoutant `panic = 'abort'` aux sections `[profile]` appropriées dans votre fichier *Cargo.toml*. Par exemple, si vous souhaitez abandonner lors d'un panic en mode publication *(release)*, ajoutez ceci :

```
[profile.release]
panic = 'abort'
```

Essayons d'appeler `panic!` dans un programme simple :

Fichier : src/main.rs

```
fn main() {
    panic!("crash and burn");
}
```

Lorsque vous lancez le programme, vous allez voir quelque chose comme ceci :

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

L'appel à `panic!` déclenche le message d'erreur présent dans les deux dernières lignes. La première ligne affiche notre message associé au panic et l'emplacement dans notre code source où se produit le panic : *src/main.rs:2:5* indique que c'est à la seconde ligne et au cinquième caractère de notre fichier *src/main.rs*.

Dans cet exemple, la ligne indiquée fait partie de notre code, et si nous allons voir cette ligne, nous verrons l'appel à la macro `panic!`. Dans d'autres cas, l'appel de `panic!` pourrait se produire dans du code que notre code utilise. Le nom du fichier et la ligne indiquée par le message d'erreur seront alors ceux du code de quelqu'un d'autre où la macro `panic!` est appelée, et non pas la ligne de notre code qui nous a mené à cet appel de `panic!`. Nous pouvons utiliser le retraçage des fonctions qui ont appelé `panic!` pour repérer la partie de notre code qui pose problème. Nous allons maintenant parler plus en détail du retraçage.

Utiliser le retraçage de `panic!`

Analysons un autre exemple pour voir ce qui se passe lors d'un appel de `panic!` qui se produit dans une bibliothèque à cause d'un bogue dans notre code plutôt qu'un appel à la macro directement. L'encart 9-1 montre du code qui essaye d'accéder à un indice d'un vecteur en dehors de l'intervalle des indices valides.

Fichier : src/main.rs

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

    v[99];
}
```

Encart 9-1 : tentative d'accès à un élément qui dépasse de l'intervalle d'un vecteur, ce qui provoque un `panic!`

Ici, nous essayons d'accéder au centième élément de notre vecteur (qui est à l'indice 99 car l'indexation commence à zéro), mais le vecteur a seulement trois éléments. Dans ce cas, Rust va paniquer. Utiliser `[]` est censé retourner un élément, mais si vous lui donnez un indice invalide, Rust ne pourra pas retourner un élément acceptable dans ce cas.

En C, tenter de lire au-delà de la fin d'une structure de données suit un comportement indéfini. Vous pourriez récupérer la valeur à l'emplacement mémoire qui correspondrait à l'élément demandé de la structure de données, même si cette partie de la mémoire n'appartient pas à cette structure de données. C'est ce qu'on appelle une *lecture hors limites* et cela peut mener à des failles de sécurité si un attaquant a la possibilité de contrôler l'indice de telle manière qu'il puisse lire les données qui ne devraient pas être lisibles en dehors de la structure de données.

Afin de protéger votre programme de ce genre de vulnérabilité, si vous essayez de lire un élément à un indice qui n'existe pas, Rust va arrêter l'exécution et refuser de continuer. Essayez et vous verrez :

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
99', src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Cette erreur mentionne la ligne 4 de notre fichier *main.rs* où on essaie d'accéder à l'indice 99. La ligne suivante nous informe que nous pouvons régler la variable d'environnement `RUST_BACKTRACE` pour obtenir le retraçage de ce qui s'est exactement passé pour mener à cette erreur. Un *retraçage* consiste à lister toutes les fonctions qui ont été appelées pour arriver jusqu'à ce point. En Rust, le retraçage fonctionne comme dans d'autres langages : le secret pour lire le retraçage est de commencer d'en haut et lire jusqu'à ce que vous voyiez les fichiers que vous avez écrits. C'est l'endroit où s'est produit le problème. Les lignes avant cet endroit est du code qui a été appelé par votre propre code ; les lignes qui suivent représentent le code qui a appelé votre code. Ces lignes "avant et après" peuvent être du code du cœur de Rust, du code de la bibliothèque standard, ou des crates que vous utilisez. Essayons d'obtenir un retraçage en réglant la variable d'environnement `RUST_BACKTRACE` à n'importe quelle valeur autre que 0. L'encart 9-2 nous montre un retour similaire à ce que vous devriez voir :

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
99', src/main.rs:4:5
stack backtrace:
   0: rust_begin_unwind
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/std/src/
panicking.rs:483
   1: core::panicking::panic_fmt
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/
src/panicking.rs:85
   2: core::panicking::panic_bounds_check
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/
src/panicking.rs:62
   3: <usize as core::slice::index::SliceIndex<[T]>>::index
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/
src/slice/index.rs:255
   4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/
src/slice/index.rs:15
   5: <alloc::vec::Vec<T> as core::ops::index::Index<I>>::index
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/alloc/
src/vec.rs:1982
   6: panic::main
             at ./src/main.rs:4
   7: core::ops::function::FnOnce::call_once
             at /rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/
src/ops/function.rs:227
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.
```

Encart 9-2 : le retraçage généré par l'appel de `panic!` qui s'affiche quand la variable d'environnement `RUST_BACKTRACE` est définie

Cela fait beaucoup de contenu ! Ce que vous voyez sur votre machine peut être différent en fonction de votre système d'exploitation et de votre version de Rust. Pour avoir le retraçage avec ces informations, les symboles de débogage doivent être activés. Les symboles de débogage sont activés par défaut quand on utilise `cargo build` ou `cargo run` sans le drapeau `--release`, comme c'est le cas ici.

Dans l'encart 9-2, la ligne 6 du retraçage nous montre la ligne de notre projet qui provoque le problème : la ligne 4 de *src/main.rs*. Si nous ne voulons pas que notre programme panique, le premier endroit que nous devrions inspecter est l'emplacement cité par la première ligne qui mentionne du code que nous avons écrit. Dans l'encart 9-1, où nous avons délibérément écrit du code qui panique, la solution pour ne pas paniquer est de ne pas demander un élément en dehors de l'intervalle des indices du vecteur. À l'avenir, quand votre code paniquera, vous aurez besoin de prendre des dispositions dans votre code pour les valeurs qui font paniquer et de coder quoi faire lorsque cela se produit.

Nous reviendrons sur le cas du `panic!` et sur les cas où nous devrions et ne devrions pas utiliser `panic!` pour gérer les conditions d'erreur plus tard à la fin de ce chapitre. Pour le moment, nous allons voir comment gérer une erreur en utilisant `Result`.

Des erreurs récupérables avec `Result`

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

Souvenez-vous de la section “Gérer les erreurs potentielles avec le type `Result`” du chapitre 2 que l'énumération `Result` possède deux variantes, `Ok` et `Err`, comme ci-dessous :

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

Le `T` et le `E` sont des paramètres de type génériques : nous parlerons plus en détail de la généricité au chapitre 10. Tout ce que vous avez besoin de savoir pour le moment, c'est que `T` représente le type de valeur imbriquée dans la variante `Ok` qui sera retournée dans le cas d'un succès, et `E` représente le type d'erreur imbriquée dans la variante `Err` qui sera retournée dans le cas d'un échec. Comme `Result` a ces paramètres de type génériques, nous pouvons utiliser le type `Result` et les fonctions associées dans différentes situations où la valeur de succès et la valeur d'erreur peuvent varier.

Utilisons une fonction qui retourne une valeur de type `Result` car la fonction peut échouer. Dans l'encart 9-3, nous essayons d'ouvrir un fichier :

Fichier : src/main.rs

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

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

Encart 9-3 : ouverture d'un fichier

Comment savons-nous que `File::open` retourne un `Result` ? Nous pouvons consulter la documentation de l'API de la bibliothèque standard, ou nous pouvons demander au compilateur ! Si nous appliquons à `f` une annotation de type dont nous savons qu'elle n'est *pas* le type de retour de la fonction et que nous essayons ensuite de compiler le code, le compilateur va nous dire que les types ne correspondent pas. Le message d'erreur va

ensuite nous dire *quel est le type* de `f`. Essayons cela ! Nous savons que le type de retour de `File::open` n'est pas `u32`, alors essayons de changer l'instruction `let f` par ceci :

```
    let f: u32 = File::open("hello.txt");
```

Tenter de compiler ce code nous donne maintenant le résultat suivant :

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0308]: mismatched types
 --> src/main.rs:4:18
  |
4 |     let f: u32 = File::open("hello.txt");
  |            ---   ^^^^^^^^^^^^^^^^^^^^^^^^ expected `u32`, found enum `Result`
  |            |
  |            expected due to this
  |
  = note: expected type `u32`
             found enum `Result<File, std::io::Error>`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `error-handling` due to previous error
```

Cela nous dit que le type de retour de la fonction `File::open` est de la forme `Result<T, E>`. Le paramètre générique `T` a été remplacé dans ce cas par le type en cas de succès, `std::fs::File`, qui permet d'interagir avec le fichier. Le `E` utilisé pour la valeur d'erreur est du type `std::io::Error`.

Ce type de retour veut dire que l'appel à `File::open` peut réussir et nous retourner un manipulateur de fichier qui peut nous permettre de le lire ou d'y écrire. L'utilisation de cette fonction peut aussi échouer : par exemple, si le fichier n'existe pas, ou si nous n'avons pas le droit d'accéder au fichier. La fonction `File::open` doit avoir un moyen de nous dire si son utilisation a réussi ou échoué et en même temps nous fournir soit le manipulateur de fichier, soit des informations sur l'erreur. C'est exactement ces informations que l'énumération `Result` se charge de nous transmettre.

Dans le cas où `File::open` réussit, la valeur que nous obtiendrons dans la variable `f` sera une instance de `Ok` qui contiendra un manipulateur de fichier. Dans le cas où cela échoue, la valeur dans `f` sera une instance de `Err` qui contiendra plus d'information sur le type d'erreur qui a eu lieu.

Nous avons besoin d'ajouter différentes actions dans le code de l'encart 9-3 en fonction de la valeur que `File::open` retourne. L'encart 9-4 montre une façon de gérer le `Result` en utilisant un outil basique, l'expression `match` que nous avons vue au chapitre 6.

Fichier : src/main.rs

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

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(fichier) => fichier,
        Err(erreur) => panic!("Erreur d'ouverture du fichier : {:?}", erreur),
    };
}
```

Encart 9-4 : utilisation de l'expression `match` pour gérer les variantes de `Result` qui peuvent être retournées

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

Lorsque le résultat est `Ok`, ce code va retourner la valeur `fichier` contenue dans la variante `Ok`, et nous assignons ensuite cette valeur à la variable `f`. Après le `match`, nous pourrons ensuite utiliser le manipulateur de fichier pour lire ou écrire.

L'autre branche du bloc `match` gère le cas où nous obtenons un `Err` à l'appel de `File::open`. Dans cet exemple, nous avons choisi de faire appel à la macro `panic!`. S'il n'y a pas de fichier qui s'appelle *hello.txt* dans notre répertoire actuel et que nous exécutons ce code, nous allons voir la sortie suivante suite à l'appel de la macro `panic!` :

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/error-handling`
thread 'main' panicked at 'Erreur d'ouverture du fichier : Os { code: 2, kind:
NotFound, message: "No such file or directory" }', src/main.rs:8:24
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Comme d'habitude, cette sortie nous explique avec précision ce qui s'est mal passé.

Gérer les différentes erreurs

Le code dans l'encart 9-4 va faire un `panic!` peu importe la raison de l'échec de `File::open`. Cependant, nous voulons réagir différemment en fonction de différents cas d'erreurs : si `File::open` a échoué parce que le fichier n'existe pas, nous voulons créer le

fichier et retourner le manipulateur de fichier pour ce nouveau fichier. Si `File::open` échoue pour toute autre raison, par exemple si nous n'avons pas l'autorisation d'ouvrir le fichier, nous voulons quand même que le code lance un `panic!` de la même manière qu'il l'a fait dans l'encart 9-4. C'est pourquoi nous avons ajouté dans l'encart 9-5 une expression `match` imbriquée :

Fichier : src/main.rs

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

fn main() {
    let f = File::open("hello.txt");

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

Encart 9-5 : gestion des différents cas d'erreurs avec des actions différentes

La valeur de retour de `File::open` logée dans la variante `Err` est de type `io::Error`, qui est une structure fournie par la bibliothèque standard. Cette structure a une méthode `kind` que nous pouvons appeler pour obtenir une valeur de type `io::ErrorKind`. L'énumération `io::ErrorKind` est fournie elle aussi par la bibliothèque standard et a des variantes qui représentent les différents types d'erreurs qui pourraient résulter d'une opération provenant du module `io`. La variante que nous voulons utiliser est `ErrorKind::NotFound`, qui indique que le fichier que nous essayons d'ouvrir n'existe pas encore. Donc nous utilisons `match` sur `f`, mais nous avons dans celle-ci un autre `match` sur `erreur.kind()`.

Nous souhaitons vérifier dans le `match` interne si la valeur de retour de `error.kind()` est la variante `NotFound` de l'énumération `ErrorKind`. Si c'est le cas, nous essayons de créer le fichier avec `File::create`. Cependant, comme `File::create` peut aussi échouer, nous avons besoin d'une seconde branche dans le `match` interne. Lorsque le fichier ne peut pas être créé, un message d'erreur différent est affiché. La seconde branche du `match` principal reste inchangée, donc le programme panique lorsqu'on rencontre une autre erreur que

l'absence de fichier.

D'autres solutions pour utiliser `match` avec `Result<T, E>`

Cela commence à faire beaucoup de `match` ! L'expression `match` est très utile mais elle est aussi assez rudimentaire. Dans le chapitre 13, vous en apprendrez plus sur les fermetures, qui sont utilisées avec de nombreuses méthodes définies sur `Result<T, E>` . Ces méthodes peuvent s'avérer être plus concises que l'utilisation de `match` lorsque vous travaillez avec des valeurs `Result<T, E>` dans votre code.

Par exemple, voici une autre manière d'écrire la même logique que celle dans l'encart 9-5 mais en utilisant les fermetures et la méthode `unwrap_or_else` :

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

fn main() {
    let f = File::open("hello.txt").unwrap_or_else(|erreur| {
        if erreur.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|erreur| {
                panic!("Erreur de création du fichier : {:?}", erreur);
            })
        } else {
            panic!("Erreur d'ouverture du fichier : {:?}", erreur);
        }
    });
}
```

Bien que ce code ait le même comportement que celui de l'encart 9-5, il ne contient aucune expression `match` et est plus facile à lire. Revenez sur cet exemple après avoir lu le chapitre 13, et renseignez-vous sur la méthode `unwrap_or_else` dans la documentation de la bibliothèque standard. De nombreuses méthodes de ce type peuvent clarifier de grosses expressions `match` imbriquées lorsque vous traitez les erreurs.

Raccourcis pour faire un panic lors d'une erreur : `unwrap` et `expect`

L'utilisation de `match` fonctionne assez bien, mais elle peut être un peu verbeuse et ne communique pas forcément bien son intention. Le type `Result<T, E>` a de nombreuses méthodes qui lui ont été définies pour différents cas. La méthode `unwrap` est une méthode

de raccourci implémentée comme l'expression `match` que nous avons écrite dans l'encart 9-4. Si la valeur de `Result` est la variante `Ok`, `unwrap` va retourner la valeur contenue dans le `Ok`. Si le `Result` est la variante `Err`, `unwrap` va appeler la macro `panic!` pour nous. Voici un exemple de `unwrap` en action :

Fichier : src/main.rs

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

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

Si nous exécutons ce code alors qu'il n'y a pas de fichier *hello.txt*, nous allons voir un message d'erreur suite à l'appel à `panic!` que la méthode `unwrap` a fait :

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Error {
repr: Os { code: 2, message: "No such file or directory" } }',
src/libcore/result.rs:906:4
```

De la même manière, la méthode `expect` nous donne la possibilité de définir le message d'erreur du `panic!`. Utiliser `expect` plutôt que `unwrap` et lui fournir un bon message d'erreur permet de mieux exprimer le problème et faciliter la recherche de la source d'un panic. La syntaxe de `expect` est la suivante :

Fichier : src/main.rs

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

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

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

```
thread 'main' panicked at 'Échec à l'ouverture de hello.txt: Error { repr: Os {
code: 2, message: "No such file or directory" } }', src/libcore/result.rs:906:4
```

Comme ce message d'erreur commence par le texte que nous avons précisé, `Échec à l'ouverture de hello.txt`, ce sera plus facile de trouver là d'où provient ce message d'erreur dans le code. Si nous utilisons `unwrap` à plusieurs endroits, cela peut prendre plus

de temps de comprendre exactement quel `unwrap` a causé le panic, car tous les appels à `unwrap` vont afficher le même message.

Propager les erreurs

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

Par exemple, l'encart 9-6 montre une fonction qui lit un pseudo à partir d'un fichier. Si ce fichier n'existe pas ou ne peut pas être lu, cette fonction va retourner ces erreurs au code qui a appelé la fonction.

Fichier : src/main.rs

```
use std::fs::File;
use std::io::{self, Read};

fn lire_pseudo_depuis_fichier() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(fichier) => fichier,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

Encart 9-6 : une fonction qui retourne les erreurs au code qui l'appelle en utilisant `match`

Cette fonction peut être écrite de façon plus concise, mais nous avons décidé de commencer par faire un maximum de choses manuellement pour découvrir la gestion d'erreurs ; mais à la fin, nous verrons comment raccourcir le code. Commençons par regarder le type de retour de la fonction : `Result<String, io::Error>`. Cela signifie que la fonction retourne une valeur de type `Result<T, E>` où le paramètre générique `T` a été remplacé par le type `String` et le paramètre générique `E` a été remplacé par le type `io::Error`. Si cette

fonction réussit sans problème, le code qui appellant va obtenir une valeur `Ok` qui contient une `String`, le pseudo que cette fonction lit dans le fichier. Si cette fonction rencontre un problème, le code qui appelle cette fonction va obtenir une valeur `Err` qui contient une instance de `io::Error` qui donne plus d'informations sur la raison du problème. Nous avons choisi `io::Error` comme type de retour de cette fonction parce qu'il se trouve que c'est le type d'erreur de retour pour les deux opérations qui peuvent échouer que l'on utilise dans le corps de cette fonction : la fonction `File::open` et la méthode `read_to_string`.

Le corps de la fonction commence par appeler la fonction `File::open`. Ensuite, nous gérons la valeur du `Result` avec un `match` similaire au `match` de l'encart 9-4. Si le `File::open` est un succès, le manipulateur de fichier dans la variable `fichier` du motif devient la valeur dans la variable mutable `f` et la fonction continue son déroulement. Dans le cas d'un `Err`, au lieu d'appeler `panic!`, nous utilisons `return` pour sortir prématurément de toute la fonction et en passant la valeur du `File::open`, désormais dans la variable `e`, au code appelant comme valeur de retour de cette fonction.

Donc si nous avons un manipulateur de fichier dans `f`, la fonction crée ensuite une nouvelle `String` dans la variable `s` et nous appelons la méthode `read_to_string` sur le manipulateur de fichier `f` pour extraire le contenu du fichier dans `s`. La méthode `read_to_string` retourne aussi un `Result` car elle peut échouer, même si `File::open` a réussi. Nous avons donc besoin d'un nouveau `match` pour gérer ce `Result` : si `read_to_string` réussit, alors notre fonction a réussi, et nous retournons le pseudo que nous avons extrait du fichier qui est maintenant intégré dans un `Ok`, lui-même stocké dans `s`. Si `read_to_string` échoue, nous retournons la valeur d'erreur de la même façon que nous avons retourné la valeur d'erreur dans le `match` qui gérait la valeur de retour de `File::open`. Cependant, nous n'avons pas besoin d'écrire explicitement `return`, car c'est la dernière expression de la fonction.

Le code qui appelle ce code va devoir ensuite gérer les cas où il récupère une valeur `Ok` qui contient un pseudo, ou une valeur `Err` qui contient une `io::Error`. Il revient au code appelant de décider quoi faire avec ces valeurs. Si le code appelant obtient une valeur `Err`, il peut appeler `panic!` et faire planter le programme, utiliser un pseudo par défaut, ou chercher le pseudo autre part que dans ce fichier, par exemple. Nous n'avons pas assez d'informations sur ce que le code appelant a l'intention de faire, donc nous remontons toutes les informations de succès ou d'erreur pour qu'elles soient gérées correctement.

Cette façon de propager les erreurs est si courante en Rust que Rust fournit l'opérateur point d'interrogation `?` pour faciliter ceci.

Un raccourci pour propager les erreurs : l'opérateur ?

L'encart 9-7 montre une implémentation de `lire_pseudo_depuis_fichier` qui a les mêmes fonctionnalités que dans l'encart 9-6, mais cette implémentation utilise l'opérateur point d'interrogation `?` :

Fichier : src/main.rs

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

fn lire_pseudo_depuis_fichier() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

Encart 9-7 : une fonction qui retourne les erreurs au code appelant en utilisant l'opérateur `?`

Le `?` placé après une valeur `Result` est conçu pour fonctionner presque de la même manière que les expressions `match` que nous avons définies pour gérer les valeurs `Result` dans l'encart 9-6. Si la valeur du `Result` est un `Ok`, la valeur dans le `Ok` sera retournée par cette expression et le programme continuera. Si la valeur est un `Err`, le `Err` sera retourné par la fonction comme si nous avions utilisé le mot-clé `return` afin que la valeur d'erreur soit propagée au code appelant.

Il y a une différence entre ce que fait l'expression `match` de l'encart 9-6 et ce que fait l'opérateur `?` : les valeurs d'erreurs sur lesquelles est utilisé l'opérateur `?` passent par la fonction `from`, définie dans le trait `From` de la bibliothèque standard, qui est utilisée pour convertir les erreurs d'un type à un autre. Lorsque l'opérateur `?` appelle la fonction `from`, le type d'erreur reçu est converti dans le type d'erreur déclaré dans le type de retour de la fonction concernée. C'est utile lorsqu'une fonction retourne un type d'erreur qui peut couvrir tous les cas d'échec de la fonction, même si certaines de ses parties peuvent échouer pour différentes raisons. À partir du moment qu'il y a un `impl From<AutreErreur>` sur `ErreurRetournee` pour expliquer la conversion dans la fonction `from` du trait, l'opérateur `?` se charge d'appeler la fonction `from` automatiquement.

Dans le cas de l'encart 9-7, le `?` à la fin de l'appel à `File::open` va retourner la valeur à l'intérieur d'un `Ok` à la variable `f`. Si une erreur se produit, l'opérateur `?` va quitter prématurément la fonction et retourner une valeur `Err` au code appelant. La même chose se produira au `?` à la fin de l'appel à `read_to_string`.

L'opérateur `?` allège l'écriture de code et facilite l'implémentation de la fonction. Nous pouvons même encore plus réduire ce code en enchaînant immédiatement les appels aux méthodes après le `?` comme dans l'encart 9-8 :

Fichier : src/main.rs

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

fn lire_pseudo_depuis_fichier() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

Encart 9-8 : enchaînement des appels aux méthodes après l'opérateur `?`

Nous avons déplacé la création de la nouvelle `String` dans `s` au début de la fonction ; cette partie n'a pas changé. Au lieu de créer la variable `f`, nous enchaînons directement l'appel à `read_to_string` sur le résultat de `File::open("hello.txt")?`. Nous avons toujours le `?` à la fin de l'appel à `read_to_string`, et nous retournons toujours une valeur `Ok` contenant le pseudo dans `s` lorsque `File::open` et `read_to_string` réussissent toutes les deux plutôt que de retourner des erreurs. Cette fonctionnalité est toujours la même que dans l'encart 9-6 et l'encart 9-7 ; c'est juste une façon différente et plus ergonomique de l'écrire.

L'encart 9-9 nous montre comment encore plus raccourcir tout ceci en utilisant `fs::read_to_string`.

Fichier : src/main.rs

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

fn lire_pseudo_depuis_fichier() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

Encart 9-9 : utilisation de `fs::read_to_string` plutôt que d'ouvrir puis lire le fichier

Récupérer le contenu d'un fichier dans une `String` est une opération assez courante, donc

la bibliothèque standard fournit la fonction assez pratique `fs::read_to_string`, qui ouvre le fichier, crée une nouvelle `String`, lit le contenu du fichier, insère ce contenu dans cette `String`, et la retourne. Évidemment, l'utilisation de `fs:read_to_string` ne nous offre pas l'occasion d'expliquer toute la gestion des erreurs, donc nous avons d'abord utilisé la manière la plus longue.

Où l'opérateur ? peut être utilisé

L'opérateur `?` ne peut être utilisé uniquement que dans des fonctions dont le type de retour compatible avec ce sur quoi le `?` est utilisé. C'est parce que l'opérateur `?` est conçu pour retourner prématurémment une valeur de la fonction, de la même manière que le faisait l'expression `match` que nous avons définie dans l'encart 9-6. Dans l'encart 9-6, le `match` utilisait une valeur de type `Result`, et la branche de retour prématuré retournait une valeur de type `Err(e)`. Le type de retour de cette fonction doit être un `Result` afin d'être compatible avec ce `return`.

Dans l'encart 9-10, découvrons l'erreur que nous allons obtenir si nous utilisons l'opérateur `?` dans une fonction `main` qui a un type de retour incompatible avec le type de valeur sur laquelle nous utilisons `?` :

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

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

Encart 9-10 : tentative d'utilisation du `?` dans la fonction `main` qui retourne un `()`, qui ne devrait pas pouvoir se compiler

Ce code ouvre un fichier, ce qui devrait échouer. L'opérateur `?` est placée derrière la valeur de type `Result` retournée par `File::open`, mais cette fonction `main` a un type de retour `()` et non pas `Result`. Lorsque nous compilons ce code, nous obtenons le message d'erreur suivant :

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0277]: the `?` operator can only be used in a function that returns
`Result` or `Option` (or another type that implements `FromResidual`)
 --> src/main.rs:4:36
  |
3 | / fn main() {
4 | |     let f = File::open("hello.txt")?;
  | |                                    ^ cannot use the `?` operator in a
function that returns `()`
5 | | }
  | |_- this function should return `Result` or `Option` to accept `?`
  |
  = help: the trait `FromResidual<Result<Infallible, std::io::Error>>` is not
implemented for `()`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `error-handling` due to previous error
```

Cette erreur explique que nous sommes autorisés à utiliser l'opérateur `?` uniquement dans une fonction qui retourne `Result`, `Option`, ou un autre type qui implémente `FromResidual`. Pour corriger l'erreur, vous avez deux choix. Le premier est de changer le type de retour de votre fonction pour être compatible avec la valeur avec lequel vous utilisez l'opérateur `?`, si vous pouvez le faire. L'autre solution est d'utiliser un `match` ou une des méthodes de `Result<T, E>` pour gérer le `Result<T, E>` de la manière la plus appropriée.

Le message d'erreur indique également que `?` peut aussi être utilisé avec des valeurs de type `Option<T>`. Comme pour pouvoir utiliser `?` sur un `Result`, vous devez utiliser `?` sur `Option` uniquement dans une fonction qui retourne une `Option`. Le comportement de l'opérateur `?` sur une `Option<T>` est identique au comportement sur un `Result<T, E>` : si la valeur est `None`, le `None` sera retourné prématurémment à la fonction dans laquelle il est utilisé. Si la valeur est `Some`, la valeur dans le `Some` sera la valeur résultante de l'expression et la fonction continuera son déroulement. L'encart 9-11 est un exemple de fonction qui trouve le dernier caractère de la première ligne dans le texte qu'on lui fournit :

```
fn dernier_caractere_de_la_premiere_ligne(texte: &str) -> Option<char> {
    texte.lines().next()?.chars().last()
}
```

Encart 9-11 : utilisation de l'opérateur `?` sur une valeur du type `Option<T>`

Cette fonction retourne un type `Option<char>` car il est possible qu'il y ait un caractère à cet endroit, mais il est aussi possible qu'il n'y soit pas. Ce code prends l'argument `texte` slice de chaîne de caractère et appelle sur elle la méthode `lines`, qui retourne un itérateur des lignes dans la chaîne. Comme cette fonction veut traiter la première ligne, elle appelle `next`

sur l'itérateur afin d'obtenir la première valeur de cet itérateur. Si `texte` est une chaîne vide, cet appel à `next` va retourner `None`, et dans ce cas nous utilisons `?` pour arrêter le déroulement de la fonction et retourner `None`. Si `texte` n'est pas une chaîne vide, `next` va retourner une valeur de type `Some` contenant une slice de chaîne de caractères de la première ligne de `texte`.

Le `?` extrait la slice de la chaîne de caractères, et nous pouvons ainsi appeller `chars` sur cette slice de chaîne de caractères afin d'obtenir un itérateur de ses caractères. Nous nous intéressons au dernier caractère de cette première ligne, donc nous appelons `last` pour retourner le dernier élément dans l'itérateur. C'est une `Option` car il est possible que la première ligne soit une chaîne de caractères vide, par exemple si `texte` commence par une ligne vide mais a des caractères sur les autres lignes, comme par exemple `"\nhi"`. Cependant, si il y a un caractère à la fin de la première ligne, il sera retourné dans la variante `Some`. L'opérateur `?` au millieu nous donne un moyen concret d'exprimer cette logique, nous permettant d'implémenter la fonction en une ligne. Si nous n'avions pas pu utiliser l'opérateur `?` sur `Option`, nous aurions dû implémenter cette logique en utilisant plus d'appels à des méthodes ou des expressions `match`.

Notez bien que vous pouvez utiliser l'opérateur `?` sur un `Result` dans une fonction qui retourne `Result`, et vous pouvez utiliser l'opérateur `?` sur une `Option` dans une fonction qui retourne une `Option`, mais vous ne pouvez pas mélanger les deux. L'opérateur `?` ne va pas convertir un `Result` en `Option` et vice-versa ; dans ce cas, vous pouvez utiliser des méthodes comme la méthode `ok` sur `Result` ou la méthode `ok_or` sur `Option` pour faire explicitement la conversion.

Jusqu'ici, toutes les fonctions `main` que nous avons utilisé retournent `()`. La fonction `main` est spéciale car c'est le point d'entrée et de sortie des programmes exécutables, et il y a quelques limitations sur ce que peut être le type de retour pour que les programmes se comportent correctement.

Heureusement, `main` peut aussi retourner un `Result<(), E>`. L'encart 9-12 reprend le code de l'encart 9-10 mais nous avons changé le type de retour du `main` pour être `Result<(), Box<dyn Error>>` et nous avons ajouté la valeur de retour `Ok(())` à la fin. Ce code devrait maintenant pouvoir se compiler :

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let f = File::open("hello.txt")?;

    Ok(())
}
```

Encart 9-12 : changement du `main` pour qu'elle retourne un `Result<(), E>` permettant d'utiliser l'opérateur `?` sur des valeurs de type `Result`

Le type `Box<dyn Error>` est un *objet trait*, que nous verrons dans une section du chapitre 17. Pour l'instant, vous pouvez interpréter `Box<dyn Error>` en “tout type d'erreur”. L'utilisation de `?` sur une valeur type `Result` dans la fonction `main` avec le type `Box<dyn Error>` est donc permise, car cela permet à n'importe quelle une valeur de type `Err` d'être retournée prématurément.

Lorsqu'une fonction `main` retourne un `Result<(), E>`, l'exécutable va terminer son exécution avec une valeur de `0` si le `main` retourne `Ok(())` et va se terminer avec une valeur différente de zéro si `main` retourne une valeur `Err`. Les exécutables écrits en C retournent des entiers lorsqu'ils se terminent : les programmes qui se terminent avec succès retournent l'entier `0`, et les programmes qui sont en erreur retournent un entier autre que `0`. Rust retourne également des entiers avec des exécutables pour être compatible avec cette convention.

La fonction `main` peut retourner n'importe quel type qui implémente le trait `std::process::Termination`. Au moment de l'écriture de ces mots, le trait `Termination` est une fonctionnalité instable seulement disponible avec la version expérimentale de Rust, donc vous ne pouvez pas l'implémenter sur vos propres types avec la version stable de Rust, mais vous pourrez peut-être le faire un jour !

Maintenant que nous avons vu les détails pour utiliser `panic!` ou retourner `Result`, voyons maintenant comment choisir ce qu'il faut faire en fonction des cas.

Paniquer ou ne pas paniquer, telle est la question

Comment décider si vous devez utiliser `panic!` ou si vous devez retourner un `Result` ? Quand un code panique, il n'y a pas de moyen de récupérer la situation. Vous pourriez utiliser `panic!` pour n'importe quelle situation d'erreur, peu importe s'il est possible de récupérer la situation ou non, mais vous prenez alors la décision de tout arrêter à la place du code appellant. Lorsque vous choisissez de retourner une valeur `Result`, vous donnez le choix au code appelant. Le code appelant peut choisir d'essayer de récupérer l'erreur de manière appropriée à la situation, ou il peut décider que dans ce cas une valeur `Err` est irrécupérable, et va donc utiliser `panic!` et transformer votre erreur récupérable en erreur irrécupérable. Ainsi, retourner `Result` est un bon choix par défaut lorsque vous définissez une fonction qui peut échouer.

Dans certains cas comme les exemples, les prototypes, et les tests, il est plus approprié d'écrire du code qui panique plutôt que de retourner un `Result`. Nous allons voir pourquoi, puis nous verrons des situations dans lesquelles vous savez en tant qu'humain qu'un code ne peut pas échouer, mais que le compilateur ne peut pas le déduire par lui-même. Enfin, nous allons conclure le chapitre par quelques lignes directrices générales pour décider s'il faut paniquer dans le code d'une bibliothèque.

Les exemples, les prototypes et les tests

Lorsque vous écrivez un exemple pour illustrer un concept, y rajouter un code de gestion des erreurs très résilient peut nuire à la clarté de l'exemple. Dans les exemples, il est courant d'utiliser une méthode comme `unwrap` (qui peut faire un panic) pour remplacer le code de gestion de l'erreur que vous utiliseriez en temps normal dans votre application, et qui peut changer en fonction de ce que le reste de votre code va faire.

De la même manière, les méthodes `unwrap` et `expect` sont très pratiques pour coder des prototypes, avant même de décider comment gérer les erreurs. Ce sont des indicateurs clairs dans votre code pour plus tard quand vous serez prêt à rendre votre code plus résilient aux échecs.

Si l'appel à une méthode échoue dans un test, nous voulons que tout le test échoue, même si cette méthode n'est pas la fonctionnalité que nous testons. Puisque c'est `panic!` qui indique qu'un test a échoué, utiliser `unwrap` ou `expect` est exactement ce qu'il faut faire.

Les cas où vous avez plus d'informations que le compilateur

Vous pouvez utiliser `unwrap` lorsque vous avez une certaine logique qui garantit que le `Result` sera toujours une valeur `Ok`, mais que ce n'est pas le genre de logique que le compilateur arrive à comprendre. Vous aurez quand même une valeur `Result` à gérer : l'opération que vous utilisez peut échouer de manière générale, même si dans votre cas c'est logiquement impossible. Si en inspectant manuellement le code vous vous rendez compte que vous n'aurez jamais une variante `Err`, vous pouvez tout à fait utiliser `unwrap`. Voici un exemple :

```
use std::net::IpAddr;

let home: IpAddr = "127.0.0.1".parse().unwrap();
```

Nous créons une instance de `IpAddr` en interprétant une chaîne de caractères codée en dur dans le code. Nous savons que `127.0.0.1` est une adresse IP valide, donc il est acceptable d'utiliser `unwrap` ici. Toutefois, avoir une chaîne de caractères valide et codée en dur ne change pas le type de retour de la méthode `parse` : nous obtenons toujours une valeur de type `Result` et le compilateur va nous demander de gérer le `Result` comme si on pouvait obtenir la variante `Err`, car le compilateur n'est pas suffisamment intelligent pour comprendre que cette chaîne de caractères est toujours une adresse IP valide. Si le texte de l'adresse IP provient de l'utilisateur au lieu d'être codé en dur dans le programme et donc qu'il y a désormais une possibilité d'erreur, alors nous devrions vouloir gérer le `Result` d'une manière plus résiliente.

Recommandations pour gérer les erreurs

Il est recommandé de faire paniquer votre code dès qu'il risque d'aboutir à un état invalide. Dans ce contexte, un *état invalide* est lorsqu'un postulat, une garantie, un contrat ou un invariant a été rompu, comme des valeurs invalides, contradictoires ou manquantes qui sont fournies à votre code, ainsi qu'un ou plusieurs des éléments suivants :

- L'état invalide est quelque chose qui est inattendu, contrairement à quelque chose qui devrait arriver occasionnellement, comme par exemple un utilisateur qui saisit une donnée dans un mauvais format.
- Après cette instruction, votre code a besoin de ne pas être dans cet état invalide, plutôt que d'avoir à vérifier le problème à chaque étape.
- Il n'y a pas de bonne façon d'encoder cette information dans les types que vous utilisez. Nous allons pratiquer ceci via un exemple dans une section du chapitre 17.

Si une personne utilise votre bibliothèque et lui fournit des valeurs qui n'ont pas de sens, la meilleure des choses à faire est d'utiliser `panic!` et d'avertir cette personne du bogue dans son code afin qu'elle le règle pendant la phase de développement. De la même manière,

`panic!` est parfois approprié si vous appelez du code externe sur lequel vous n'avez pas la main, et qu'il retourne un état invalide que vous ne pouvez pas corriger.

Cependant, si l'on s'attend à rencontrer des échecs, il est plus approprié de retourner un `Result` plutôt que de faire appel à `panic!`. Il peut s'agir par exemple d'un interpréteur qui reçoit des données erronées, ou une requête HTTP qui retourne un statut qui indique que vous avez atteint une limite de débit. Dans ces cas-là, vous devriez indiquer qu'il est possible que cela puisse échouer en retournant un `Result` afin que le code appelant puisse décider quoi faire pour gérer le problème.

Lorsque votre code effectue des opérations sur des valeurs, votre code devrait d'abord vérifier que ces valeurs sont valides, et faire un panic si les valeurs ne sont pas correctes. C'est essentiellement pour des raisons de sécurité : tenter de travailler avec des données invalides peut exposer votre code à des vulnérabilités. C'est la principale raison pour laquelle la bibliothèque standard va appeler `panic!` si vous essayez d'accéder à la mémoire hors limite : essayer d'accéder à de la mémoire qui n'appartient pas à la structure de données actuelle est un problème de sécurité fréquent. Les fonctions ont souvent des *contrats* : leur comportement est garanti uniquement si les données d'entrée remplissent des conditions particulières. Paniquer lorsque le contrat est violé est justifié, car une violation de contrat signifie toujours un bogue du côté de l'appelant, et ce n'est pas le genre d'erreur que vous voulez que le code appelant gère explicitement. En fait, il n'y a aucun moyen rationnel pour que le code appelant se corrige : le *développeur* du code appelant doit corriger le code. Les contrats d'une fonction, en particulier lorsqu'une violation va causer un panic, doivent être expliqués dans la documentation de l'API de ladite fonction.

Cependant, avoir beaucoup de vérifications d'erreurs dans toutes vos fonctions serait verbeux et pénible. Heureusement, vous pouvez utiliser le système de types de Rust (et donc la vérification de type que fait le compilateur) pour assurer une partie des vérifications à votre place. Si votre fonction a un paramètre d'un type précis, vous pouvez continuer à écrire votre code en sachant que le compilateur s'est déjà assuré que vous avez une valeur valide. Par exemple, si vous obtenez un type de valeur plutôt qu'une `Option`, votre programme s'attend à obtenir *quelque chose* plutôt que *rien*. Votre code n'a donc pas à gérer les deux cas de variantes `Some` et `None` : la seule possibilité est qu'il y a une valeur. Du code qui essaye de ne rien fournir à votre fonction ne compilera même pas, donc votre fonction n'a pas besoin de vérifier ce cas-là lors de l'exécution. Un autre exemple est d'utiliser un type d'entier non signé comme `u32`, qui garantit que le paramètre n'est jamais strictement négatif.

Créer des types personnalisés pour la vérification

Allons plus loin dans l'idée d'utiliser le système de types de Rust pour s'assurer d'avoir une valeur valide en créant un type personnalisé pour la vérification. Souvenez-vous du jeu du plus ou du moins du chapitre 2 dans lequel notre code demandait à l'utilisateur de deviner un nombre entre 1 et 100. Nous n'avons jamais validé que le nombre saisi par l'utilisateur était entre ces nombres avant de le comparer à notre nombre secret ; nous avons seulement vérifié que le nombre était positif. Dans ce cas, les conséquences ne sont pas très graves : notre résultat “C'est plus !” ou “C'est moins !” sera toujours correct. Mais ce serait une amélioration utile pour aider l'utilisateur à faire des suppositions valides et pour avoir un comportement différent selon qu'un utilisateur propose un nombre en dehors des limites ou qu'il saisit, par exemple, des lettres à la place.

Une façon de faire cela serait de stocker le nombre saisi dans un `i32` plutôt que dans un `u32` afin de permettre d'obtenir potentiellement des nombres négatifs, et ensuite vérifier que le nombre est dans la plage autorisée, comme ceci :

```
loop {
    // -- partie masquée ici --

    let supposition: i32 = match supposition.trim().parse() {
        Ok(nombre) => nombre,
        Err(_) => continue,
    };

    if supposition < 1 || supposition > 100 {
        println!("Le nombre secret est entre 1 et 100.");
        continue;
    }

    match supposition.cmp(&nombre_secret) {
        // -- partie masquée ici --
}
```

L'expression `if` vérifie si la valeur est en dehors des limites et informe l'utilisateur du problème le cas échéant, puis utilise `continue` pour passer à la prochaine itération de la boucle et ainsi demander de saisir une nouvelle supposition. Après l'expression `if`, nous pouvons continuer avec la comparaison entre `supposition` et le nombre secret tout en sachant que `supposition` est entre 1 et 100.

Cependant, ce n'est pas une solution idéale : si c'était absolument critique que le programme ne travaille qu'avec des valeurs entre 1 et 100 et qu'il aurait de nombreuses fonctions qui reposent sur cette condition, cela pourrait être fastidieux (et cela impacterait potentiellement la performance) de faire une vérification comme celle-ci dans chacune de ces fonctions.

À la place, nous pourrions construire un nouveau type et intégrer les vérifications dans la

fonction de création d'une instance de ce type plutôt que de répéter partout les vérifications. Il est ainsi plus sûr pour les fonctions d'utiliser ce nouveau type dans leurs signatures et d'utiliser avec confiance les valeurs qu'elles reçoivent. L'encart 9-13 montre une façon de définir un type `Supposition` qui ne créera une instance de `Supposition` que si la fonction `new` reçoit une valeur entre 1 et 100 :

```
pub struct Supposition {
    valeur: i32,
}

impl Supposition {
    pub fn new(valeur: i32) -> Supposition {
        if valeur < 1 || valeur > 100 {
            panic!("Supposition valeur must be between 1 and 100, got {}.",
valeur);
        }

        Supposition { valeur }
    }

    pub fn valeur(&self) -> i32 {
        self.valeur
    }
}
```

Encart 9-13 : un type `Supposition` qui ne va continuer que si la valeur est entre 1 et 100

D'abord, nous définissons une structure qui s'appelle `Supposition` qui a un champ `valeur` qui stocke un `i32`. C'est dans ce dernier que le nombre sera stocké.

Ensuite, nous implémentons une fonction associée `new` sur `Supposition` qui crée des instances de `Supposition`. La fonction `new` est conçue pour recevoir un paramètre `valeur` de type `i32` et retourner une `Supposition`. Le code dans le corps de la fonction `new` teste `valeur` pour s'assurer qu'elle est bien entre 1 et 100. Si `valeur` échoue à ce test, nous faisons appel à `panic!`, qui alertera le développeur qui écrit le code appelant qu'il a un bogue qu'il doit régler, car créer une `Supposition` avec `valeur` en dehors de cette plage va violer le contrat sur lequel s'appuie `Supposition::new`. Les conditions dans lesquelles `Supposition::new` va paniquer devraient être expliquées dans la documentation publique de l'API ; nous verrons les conventions pour indiquer l'éventualité d'un `panic!` dans la documentation de l'API que vous créerez au chapitre 14. Si `valeur` passe le test, nous créons une nouvelle `Supposition` avec son champ `valeur` qui prend la valeur du paramètre `valeur` et retourne cette `Supposition`.

Enfin, nous implémentons une méthode `valeur` qui emprunte `self`, n'a aucun autre

paramètre, et retourne un `i32`. Ce genre de méthode est parfois appelé un *accesseur*, car son rôle est d'accéder aux données des champs et de les retourner. Cette méthode publique est nécessaire car le champ `valeur` de la structure `Supposition` est privé. Il est important que le champ `valeur` soit privé pour que le code qui utilise la structure `Supposition` ne puisse pas directement assigner une valeur à `valeur` : le code en dehors du module *doit* utiliser la fonction `Supposition::new` pour créer une instance de `Supposition`, ce qui permet d'empêcher la création d'une `Supposition` avec un champ `valeur` qui n'a pas été vérifié par les conditions dans la fonction `Supposition:new`.

Une fonction qui prend en paramètre ou qui retourne des nombres uniquement entre 1 et 100 peut ensuite déclarer dans sa signature qu'elle prend en paramètre ou qu'elle retourne une `Supposition` plutôt qu'un `i32` et n'aura pas besoin de faire de vérifications supplémentaires dans son corps.

Résumé

Les fonctionnalités de gestion d'erreurs de Rust sont conçues pour vous aider à écrire du code plus résilient. La macro `panic!` signale que votre programme est dans un état qu'il ne peut pas gérer et vous permet de dire au processus de s'arrêter au lieu d'essayer de continuer avec des valeurs invalides ou incorrectes. L'énumération `Result` utilise le système de types de Rust pour signaler que des opérations peuvent échouer de telle façon que votre code puisse rattraper l'erreur. Vous pouvez utiliser `Result` pour dire au code qui appelle votre code qu'il a besoin de gérer le résultat et aussi les potentielles erreurs. Utiliser `panic!` et `Result` de manière appropriée rendra votre code plus fiable face à des problèmes inévitables.

Maintenant que vous avez vu la façon dont la bibliothèque standard tire parti de la généricité avec les énumérations `Option` et `Result`, nous allons voir comment la généricité fonctionne et comment vous pouvez l'utiliser dans votre code.

Les types génériques, les traits et les durées de vie

Tous les langages de programmation ont des outils pour gérer la duplication des concepts. En Rust, un de ces outils est la *généricité*. La généricité permet de remplacer des types concrets ou d'autres propriétés par des paramètres abstraits appelés *génériques*. Lorsque nous écrivons du code, nous pouvons exprimer le comportement des génériques, ou comment ils interagissent avec d'autres génériques, sans savoir ce qu'il y aura à leur place lors de la compilation et de l'exécution du code.

De la même manière qu'une fonction prend des paramètres avec des valeurs inconnues pour exécuter le même code sur plusieurs valeurs concrètes, les fonctions peuvent prendre des paramètres d'un type générique plutôt que d'un type concret comme `i32` ou `String`. En fait, nous avons déjà utilisé des types génériques au chapitre 6 avec `Option<T>`, au chapitre 8 avec `Vec<T>` et `HashMap<K, V>`, et au chapitre 9 avec `Result<T, E>`. Dans ce chapitre, nous allons voir comment définir nos propres types, fonctions et méthodes utilisant des types génériques !

Pour commencer, nous allons examiner comment construire une fonction pour réduire la duplication de code. Ensuite, nous utiliserons la même technique pour construire une fonction générique à partir de deux fonctions qui se distinguent uniquement par le type de leurs paramètres. Nous expliquerons aussi comment utiliser les types génériques dans les définitions de structures et d'énumérations.

Ensuite, vous apprendrez comment utiliser les *traits* pour définir un comportement de manière générique. Vous pouvez combiner les traits avec des types génériques pour contraindre un type générique uniquement à des types qui ont un comportement particulier, et non pas accepter n'importe quel type.

Enfin, nous verrons les *durées de vie*, un genre de générique qui indique au compilateur comment les références s'articulent entre elles. Les durées de vie nous permettent d'emprunter des valeurs dans différentes situations tout en donnant les éléments au compilateur pour vérifier que les références sont toujours valides.

Supprimer les doublons en construisant une fonction

Avant de plonger dans la syntaxe des génériques, nous allons regarder comment supprimer les doublons, sans utiliser de types génériques, en construisant une fonction. Ensuite, nous

allons appliquer cette technique pour construire une fonction générique ! De la même manière que vous détectez du code dupliqué pour l'extraire dans une fonction, vous allez commencer à reconnaître du code dupliqué qui peut utiliser la généricité.

Imaginons un petit programme qui trouve le nombre le plus grand dans une liste, comme dans l'encart 10-1.

Fichier: src/main.rs

```
fn main() {
    let liste_de_nombres = vec![34, 50, 25, 100, 65];

    let mut le_plus_grand = liste_de_nombres[0];

    for nombre in liste_de_nombres {
        if nombre > le_plus_grand {
            le_plus_grand = nombre;
        }
    }

    println!("Le nombre le plus grand est {}", le_plus_grand);
}
```

Encart 10-1 : le code pour trouver le nombre le plus grand dans une liste de nombres

Ce code stocke une liste de nombres entiers dans la variable `liste_de_nombres` et place le premier nombre de la liste dans une variable qui s'appelle `le_plus_grand`. Ensuite, il parcourt tous les nombres dans la liste, et si le nombre courant est plus grand que le nombre stocké dans `le_plus_grand`, il remplace le nombre dans cette variable. Cependant, si le nombre courant est plus petit ou égal au nombre le plus grand trouvé précédemment, la variable ne change pas, et le code passe au nombre suivant de la liste. Après avoir parcouru tous les nombres de la liste, `le_plus_grand` devrait stocker le plus grand nombre, qui est 100 dans notre cas.

Pour trouver le nombre le plus grand dans deux listes de nombres différentes, nous pourrions dupliquer le code de l'encart 10-1 et suivre la même logique à deux endroits différents du programme, comme dans l'encart 10-2.

Fichier : src/main.rs

```
fn main() {
    let liste_de_nombres = vec![34, 50, 25, 100, 65];

    let mut le_plus_grand = liste_de_nombres[0];

    for nombre in liste_de_nombres {
        if nombre > le_plus_grand {
            le_plus_grand = nombre;
        }
    }

    println!("Le nombre le plus grand est {}", le_plus_grand);

    let liste_de_nombres = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut le_plus_grand = liste_de_nombres[0];

    for nombre in liste_de_nombres {
        if nombre > le_plus_grand {
            le_plus_grand = nombre;
        }
    }

    println!("Le nombre le plus grand est {}", le_plus_grand);
}
```

Encart 10-2 : le code pour trouver le plus grand nombre dans *deux* listes de nombres

Bien que ce code fonctionne, la duplication de code est fastidieuse et source d'erreurs. Nous devons aussi mettre à jour le code à plusieurs endroits si nous souhaitons le modifier.

Pour éviter cette duplication, nous pouvons créer un niveau d'abstraction en définissant une fonction qui travaille avec n'importe quelle liste de nombres entiers qu'on lui donne en paramètre. Cette solution rend notre code plus clair et nous permet d'exprimer le concept de trouver le nombre le plus grand dans une liste de manière abstraite.

Dans l'encart 10-3, nous avons extrait le code qui trouve le nombre le plus grand dans une fonction qui s'appelle `le_plus_grand`. Contrairement au code de l'encart 10-1, qui pouvait trouver le nombre le plus grand dans une seule liste en particulier, ce programme peut trouver le nombre le plus grand dans deux listes différentes.

Fichier : src/main.rs

```
fn le_plus_grand(liste: &[i32]) -> i32 {
    let mut le_plus_grand = liste[0];

    for &element in liste {
        if element > le_plus_grand {
            le_plus_grand = element;
        }
    }

    le_plus_grand
}

fn main() {
    let liste_de_nombres = vec![34, 50, 25, 100, 65];

    let resultat = le_plus_grand(&liste_de_nombres);
    println!("Le nombre le plus grand est {}", resultat);

    let liste_de_nombres = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let resultat = le_plus_grand(&liste_de_nombres);
    println!("Le nombre le plus grand est {}", resultat);
}
```

Encart 10-3 : du code abstrait qui trouve le plus grand nombre dans deux listes

La fonction `le_plus_grand` a un paramètre qui s'appelle `liste`, qui représente n'importe quelle slice concrète de valeurs `i32` que nous pouvons passer à la fonction. Au final, lorsque nous appelons la fonction, le code s'exécute sur les valeurs précises que nous lui avons fournies. Mais ne nous préoccupons pas de la syntaxe de la boucle `for` pour l'instant. Ici, nous n'utilisons pas une référence vers un `i32`, nous destructurons via le filtrage par motif chaque `&i32` afin que la boucle `for` utilise cet `element` en tant que `i32` dans le corps de la boucle. Nous parlerons plus en détails du filtrage par motif au chapitre 18.

En résumé, voici les étapes que nous avons suivies pour changer le code de l'encart 10-2 pour obtenir celui de l'encart 10-3 :

1. Identification du code dupliqué.
2. Extraction du code dupliqué dans le corps de la fonction et ajout de précisions sur les entrées et les valeurs de retour de ce code dans la signature de la fonction.
3. Remplacement des deux instances du code dupliqué par des appels à la fonction.

Ensuite, nous allons utiliser les mêmes étapes avec la généricité pour réduire la duplication de code de différentes façons. De la même manière que le corps d'une fonction peut opérer sur une `liste` abstraite plutôt que sur des valeurs spécifiques, la généricité permet de travailler sur des types abstraits.

Par exemple, imaginons que nous ayons deux fonctions : une qui trouve l'élément le plus grand dans une slice de valeurs `i32` et une qui trouve l'élément le plus grand dans une slice de valeurs `char`. Comment pourrions-nous éviter la duplication ? Voyons cela dès maintenant !

Les types de données génériques

Nous pouvons utiliser la généricité pour créer des définitions pour des éléments comme les signatures de fonctions ou les structures, que nous pouvons ensuite utiliser sur de nombreux types de données concrets. Commençons par regarder comment définir des fonctions, des structures, des énumérations, et des méthodes en utilisant la généricité. Ensuite nous verrons comment la généricité impacte la performance du code.

Dans la définition d'une fonction

Lorsque nous définissons une fonction en utilisant la généricité, nous utilisons des types génériques dans la signature de la fonction là où nous précisons habituellement les types de données des paramètres et de la valeur de retour. Faire ainsi rend notre code plus flexible et apporte plus de fonctionnalités au code appelant notre fonction, tout en évitant la duplication de code.

Pour continuer avec notre fonction `le_plus_grand`, l'encart 10-4 nous montre deux fonctions qui trouvent toutes les deux la valeur la plus grande dans une slice.

Fichier : src/main.rs

```
fn le_plus_grand_i32(liste: &[i32]) -> i32 {
    let mut le_plus_grand = liste[0];

    for &element in liste.iter() {
        if element > le_plus_grand {
            le_plus_grand = element;
        }
    }

    le_plus_grand
}

fn le_plus_grand_caractere(liste: &[char]) -> char {
    let mut le_plus_grand = liste[0];

    for &element in liste.iter() {
        if element > le_plus_grand {
            le_plus_grand = element;
        }
    }

    le_plus_grand
}

fn main() {
    let liste_de_nombres = vec![34, 50, 25, 100, 65];

    let resultat = le_plus_grand_i32(&liste_de_nombres);
    println!("Le plus grand nombre est {}", resultat);

    let liste_de_caracteres = vec!['y', 'm', 'a', 'q'];

    let resultat = le_plus_grand_caractere(&liste_de_caracteres);
    println!("Le plus grand caractère est {}", resultat);
}
```

Encart 10-4 : deux fonctions qui se distinguent seulement par leur nom et le type dans leur signature

La fonction `le_plus_grand_i32` est celle que nous avons construite à l'encart 10-3 lorsqu'elle trouvait le plus grand `i32` dans une slice. La fonction `le_plus_grand_caractere` recherche le plus grand `char` dans une slice. Les corps des fonctions ont le même code, donc essayons d'éviter cette duplication en utilisant un paramètre de type générique dans une seule et unique fonction.

Pour paramétrer les types dans la nouvelle fonction que nous allons définir, nous avons besoin de donner un nom au paramètre de type, comme nous l'avons fait pour les paramètres de valeur des fonctions. Vous pouvez utiliser n'importe quel identificateur pour nommer le paramètre de type. Mais ici nous allons utiliser `T` car, par convention, les noms

de paramètres en Rust sont courts, souvent même une seule lettre, et la convention de nommage des types en Rust est d'utiliser le CamelCase. Et puisque la version courte de “type” est `T`, c'est le choix par défaut de nombreux développeurs Rust.

Lorsqu'on utilise un paramètre dans le corps de la fonction, nous devons déclarer le nom du paramètre dans la signature afin que le compilateur puisse savoir à quoi réfère ce nom. De la même manière, lorsqu'on utilise un nom de paramètre de type dans la signature d'une fonction, nous devons déclarer le nom du paramètre de type avant de pouvoir l'utiliser. Pour déclarer la fonction générique `le_plus_grand`, il faut placer la déclaration du nom du type entre des chevrons `<>`, le tout entre le nom de la fonction et la liste des paramètres, comme ceci :

```
fn le_plus_grand<T>(liste: &[T]) -> T {
```

Cette définition se lit comme ceci : la fonction `le_plus_grand` est générique en fonction du type `T`. Cette fonction a un paramètre qui s'appelle `liste`, qui est une slice de valeurs de type `T`. Cette fonction `le_plus_grand` va retourner une valeur du même type `T`.

L'encart 10-5 nous montre la combinaison de la définition de la fonction `le_plus_grand` avec le type de données générique présent dans sa signature. L'encart montre aussi que nous pouvons appeler la fonction avec une slice soit de valeurs `i32`, soit de valeurs `char`. Notez que ce code ne se compile pas encore, mais nous allons y remédier plus tard dans ce chapitre.

Fichier : src/main.rs

```
fn le_plus_grand<T>(liste: &[T]) -> T {
    let mut le_plus_grand = liste[0];

    for &element in liste {
        if element > le_plus_grand {
            le_plus_grand = element;
        }
    }

    le_plus_grand
}

fn main() {
    let liste_de_nombres = vec![34, 50, 25, 100, 65];

    let resultat = le_plus_grand(&liste_de_nombres);
    println!("Le nombre le plus grand est {}", resultat);

    let liste_de_caracteres = vec!['y', 'm', 'a', 'q'];

    let resultat = le_plus_grand(&liste_de_caracteres);
    println!("Le plus grand caractère est {}", resultat);
}
```

Encart 10-5 : une définition de la fonction `le_plus_grand` qui utilise des paramètres de type génériques, mais qui ne compile pas encore

Si nous essayons de compiler ce code dès maintenant, nous aurons l'erreur suivante :

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:17
  |
5 |         if element > le_plus_grand {
  |            ------- ^ ------------- T
  |            |
  |            T
  |
help: consider restricting type parameter `T`
  |
1 | fn le_plus_grand<T: std::cmp::PartialOrd>(liste: &[T]) -> T {
  |                   ++++++++++++++++++++++

For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` due to previous error
```

La note cite `std::cmp::PartialOrd`, qui est un *trait*. Nous allons voir les traits dans la prochaine section. Pour le moment, cette erreur nous informe que le corps de `le_plus_grand` ne va pas fonctionner pour tous les types possibles que `T` peut

représenter. Comme nous voulons comparer des valeurs de type `T` dans le corps, nous pouvons utiliser uniquement des types dont les valeurs peuvent être triées dans l'ordre. Pour effectuer des comparaisons, la bibliothèque standard propose le trait `std::cmp::PartialOrd` que vous pouvez implémenter sur des types (voir l'annexe C pour en savoir plus sur ce trait). Vous allez apprendre à indiquer qu'un type générique a un trait spécifique dans la section "Des traits en paramètres", mais d'abord nous allons explorer d'autres manières d'utiliser les paramètres de types génériques.

Dans la définition des structures

Nous pouvons aussi définir des structures en utilisant des paramètres de type génériques dans un ou plusieurs champs en utilisant la syntaxe `<>` . L'encart 10-6 nous montre comment définir une structure `Point<T>` pour stocker des valeurs de coordonnées `x` et `y` de n'importe quel type.

Fichier : src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let entiers = Point { x: 5, y: 10 };
    let flottants = Point { x: 1.0, y: 4.0 };
}
```

Encart 10-6 : une structure `Point<T>` qui stocke les valeurs `x` et `y` de type `T`

La syntaxe pour l'utilisation des génériques dans les définitions de structures est similaire à celle utilisée dans les définitions de fonctions. D'abord, on déclare le nom du paramètre de type entre des chevrons juste après le nom de la structure. Ensuite, on peut utiliser le type générique dans la définition de la structure là où on indiquerait en temps normal des types de données concrets.

Notez que comme nous n'avons utilisé qu'un seul type générique pour définir `Point<T>` , cette définition dit que la structure `Point<T>` est générique en fonction d'un type `T` , et les champs `x` et `y` sont *tous les deux* de ce même type, quel qu'il soit. Si nous créons une instance de `Point<T>` qui a des valeurs de types différents, comme dans l'encart 10-7, notre code ne va pas se compiler.

Fichier : src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let ne_fonctionnera_pas = Point { x: 5, y: 4.0 };
}
```

Encart 10-7 : les champs `x` et `y` doivent être du même type car ils ont tous les deux le même type de données générique `T`.

Dans cet exemple, lorsque nous assignons l'entier 5 à `x`, nous laissons entendre au compilateur que le type générique `T` sera un entier pour cette instance de `Point<T>`. Ensuite, lorsque nous assignons 4.0 à `y`, que nous avons défini comme ayant le même type que `x`, nous obtenons une erreur d'incompatibilité de type comme celle-ci :

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0308]: mismatched types
 --> src/main.rs:7:38
  |
7 |     let ne_fonctionnera_pas = Point { x: 5, y: 4.0 };
  |                                            ^^^ expected integer, found
floating-point number

For more information about this error, try `rustc --explain E0308`.
error: could not compile `chapter10` due to previous error
```

Pour définir une structure `Point` où `x` et `y` sont tous les deux génériques mais peuvent avoir des types différents, nous pouvons utiliser plusieurs paramètres de types génériques différents. Par exemple, dans l'encart 10-8, nous pouvons changer la définition de `Point` pour être générique en fonction des types `T` et `U` où `x` est de type `T` et `y` est de type `U`.

Fichier : src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let deux_entiers = Point { x: 5, y: 10 };
    let deux_flottants = Point { x: 1.0, y: 4.0 };
    let un_entier_et_un_flottant = Point { x: 5, y: 4.0 };
}
```

Encart 10-8: un `Point<T, U>` générique en fonction de deux types `x` et `y` qui peuvent être des valeurs de types différents

Maintenant, toutes les instances de `Point` montrées ici sont valides ! Vous pouvez utiliser autant de paramètres de type génériques que vous souhaitez dans la déclaration de la définition, mais en utiliser plus de quelques-uns rend votre code difficile à lire. Lorsque vous avez besoin de nombreux types génériques dans votre code, cela peut être un signe que votre code a besoin d'être remanié en éléments plus petits.

Dans les définitions d'énumérations

Comme nous l'avons fait avec les structures, nous pouvons définir des énumérations qui utilisent des types de données génériques dans leurs variantes. Commençons par regarder à nouveau l'énumération `Option<T>` que fournit la bibliothèque standard, et que nous avons utilisée au chapitre 6 :

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

Cette définition devrait désormais avoir plus de sens pour vous. Comme vous pouvez le constater, `Option<T>` est une énumération qui est générique en fonction du type `T` et a deux variantes : `Some`, qui contient une valeur de type `T`, et une variante `None` qui ne contient aucune valeur. En utilisant l'énumération `Option<T>`, nous pouvons exprimer le concept abstrait d'avoir une valeur optionnelle, et comme `Option<T>` est générique, nous pouvons utiliser cette abstraction peu importe le type de la valeur optionnelle.

Les énumérations peuvent aussi utiliser plusieurs types génériques. La définition de l'énumération `Result` que nous avons utilisée au chapitre 9 en est un exemple :

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

L'énumération `Result` est générique en fonction de deux types, `T` et `E`, et a deux variantes : `Ok`, qui contient une valeur de type `T`, et `Err`, qui contient une valeur de type `E`. Cette définition rend possible l'utilisation de l'énumération `Result` partout où nous avons une opération qui peut réussir (et retourner une valeur du type `T`) ou échouer (et

retourner une erreur du type `E`). En fait, c'est ce qui est utilisé pour ouvrir un fichier dans l'encart 9-3, où `T` contenait un type `std::fs::File` lorsque le fichier était ouvert avec succès et `E` contenait un type `std::io::Error` lorsqu'il y avait des problèmes pour ouvrir le fichier.

Lorsque vous reconnaîtrez des cas dans votre code où vous aurez plusieurs définitions de structures ou d'énumérations qui se distinguent uniquement par le type de valeurs qu'elles stockent, vous pourrez éviter les doublons en utilisant des types génériques à la place.

Dans la définition des méthodes

Nous pouvons implémenter des méthodes sur des structures et des énumérations (comme nous l'avons fait dans le chapitre 5) et aussi utiliser des types génériques dans leurs définitions. L'encart 10-9 montre la structure `Point<T>` que nous avons définie dans l'encart 10-6 avec une méthode qui s'appelle `x` implémentée sur cette dernière.

Fichier : src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

Encart 10-9 : implémentation d'une méthode `x` sur la structure `Point<T>` qui va retourner une référence au champ `x`, de type `T`

Ici, nous avons défini une méthode qui s'appelle `x` sur `Point<T>` qui retourne une référence à la donnée présente dans le champ `x`.

Notez que nous devons déclarer `T` juste après `impl` afin de pouvoir l'utiliser pour préciser que nous implémentons des méthodes sur le type `Point<T>`. En déclarant `T` comme un type générique après `impl`, Rust peut comprendre que le type entre les chevrons dans

`Point` est un type générique plutôt qu'un type concret. Comme cela revient à déclarer à nouveau le générique, nous aurions pu choisir un nom différent pour le paramètre générique plutôt que de réutiliser le même nom que dans la définition de la structure, mais c'est devenu une convention d'utiliser le même nom. Les méthodes écrites dans un `impl` qui déclarent un type générique peuvent être définies sur n'importe quelle instance du type, peu importe quel type concret sera substitué dans le type générique.

L'autre possibilité que nous avons est de définir les méthodes sur le type avec des contraintes sur le type générique. Nous pouvons par exemple implémenter des méthodes uniquement sur des instances de `Point<f32>` plutôt que sur des instances de n'importe quel type `Point<T>` . Dans l'encart 10-10, nous utilisons le type concret `f32` , ce qui veut dire que nous n'avons pas besoin de déclarer un type après `impl` .

Fichier : src/main.rs

```
impl Point<f32> {
    fn distance_depuis_lorigine(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

Encart 10-10 : un bloc `impl` qui ne s'applique que sur une structure d'un type concret particulier pour le paramètre de type générique `T`

Ce code signifie que le type `Point<f32>` va avoir une méthode qui s'appelle `distance_depuis_lorigine` et les autres instances de `Point<T>` où `T` n'est pas du type `f32` ne pourront pas appeler cette méthode. Cette méthode calcule la distance entre notre point et la coordonnée (0.0, 0.0) et utilise des opérations mathématiques qui ne sont disponibles que pour les types de flottants.

Les paramètres de type génériques dans la définition d'une structure ne sont pas toujours les mêmes que ceux qui sont utilisés dans la signature des méthodes de cette structure. Par exemple, l'encart 10-11 utilise les types génériques `X1` et `Y1` pour la structure `Point` , ainsi que `X2` et `Y2` pour la signature de la méthode `melange` pour rendre l'exemple plus clair. La méthode crée une nouvelle instance de `Point` avec la valeur de `x` provenant du `Point` `self` (de type `X1`) et la valeur de `y` provenant du `Point` en paramètre (de type `Y2`).

Fichier : src/main.rs

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn melange<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.melange(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

Encart 10-11 : une méthode qui utilise différents types génériques provenant de la définition de la structure

Dans le `main`, nous avons défini un `Point` qui a un `i32` pour `x` (avec la valeur `5`) et un `f64` pour `y` (avec la valeur 10.4). La variable `p2` est une structure `Point` qui a une slice de chaine de caractères pour `x` (avec la valeur `"Hello"`) et un caractère `char` pour `y` (avec la valeur `c`). L'appel à `melange` sur `p1` avec l'argument `p2` nous donne `p3`, qui aura un `i32` pour `x`, car `x` provient de `p1`. La variable `p3` aura un caractère `char` pour `y`, car `y` provient de `p2`. L'appel à la macro `println!` va afficher `p3.x = 5, p3.y = c`.

Le but de cet exemple est de montrer une situation dans laquelle des paramètres génériques sont déclarés avec `impl` et d'autres sont déclarés dans la définition de la méthode. Ici, les paramètres génériques `X1` et `Y1` sont déclarés après `impl`, car ils sont liés à la définition de la structure. Les paramètres génériques `X2` et `Y2` sont déclarés après `fn melange`, car ils ne sont liés qu'à cette méthode.

Performance du code utilisant les génériques

Vous vous demandez peut-être s'il y a un coût à l'exécution lorsque vous utilisez des paramètres de type génériques. La bonne nouvelle est que Rust implémente les génériques de manière à ce que votre code ne s'exécute pas plus lentement que vous utilisiez les types génériques ou des types concrets.

Rust accomplit cela en pratiquant la monomorphisation à la compilation du code qui utilise les génériques. La *monomorphisation* est un processus qui transforme du code générique en code spécifique en définissant au moment de la compilation les types concrets utilisés dans le code.

Dans ce processus, le compilateur fait l'inverse des étapes que nous avons suivies pour créer la fonction générique de l'encart 10-5 : le compilateur cherche tous les endroits où le code générique est utilisé et génère du code pour les types concrets avec lesquels le code générique est appelé.

Regardons comment cela fonctionne avec un exemple qui utilise l'énumération `Option<T>` de la bibliothèque standard :

```
let entier = Some(5);
let flottant = Some(5.0);
```

Lorsque Rust compile ce code, il applique la monomorphisation. Pendant ce processus, le compilateur lit les valeurs qui ont été utilisées dans les instances de `Option<T>` et en déduit les deux sortes de `Option<T>` : une est `i32` et l'autre est `f64` . Ainsi, il décompose la définition générique de `Option<T>` en `Option_i32` et en `Option_f64` , remplaçant ainsi la définition générique par deux définitions concrètes.

La version monomorphe du code ressemble à ce qui suit. Le `Option<T>` générique est remplacé par deux définitions concrètes créées par le compilateur :

Fichier : src/main.rs

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let entier = Option_i32::Some(5);
    let flottant = Option_f64::Some(5.0);
}
```

Comme Rust compile le code générique dans du code qui précise le type dans chaque instance, l'utilisation des génériques n'a pas de conséquence sur les performances de l'exécution. Quand le code s'exécute, il fonctionne comme il devrait le faire si nous avions

dupliqué chaque définition à la main. Le processus de monomorphisation rend les génériques de Rust très performants au moment de l'exécution.

Définir des comportements partagés avec les traits

Un *trait* décrit une fonctionnalité qu'a un type particulier et qu'il peut partager avec d'autres types, à destination du compilateur Rust. Nous pouvons utiliser les traits pour définir un comportement partagé de manière abstraite. Nous pouvons lier ces traits à un type générique pour exprimer le fait qu'il puisse être de n'importe quel type à condition qu'il ait un comportement donné.

> Remarque : les traits sont similaires à ce qu'on appelle parfois les *interfaces* dans d'autres langages, malgré quelques différences.

Définir un trait

Le comportement d'un type s'exprime via les méthodes que nous pouvons appeler sur ce type. Différents types peuvent partager le même comportement si nous pouvons appeler les mêmes méthodes sur tous ces types. Définir un trait est une manière de regrouper des signatures de méthodes pour définir un comportement nécessaire pour accomplir un objectif.

Par exemple, imaginons que nous avons plusieurs structures qui stockent différents types et quantités de texte : une structure `ArticleDePresse`, qui contient un reportage dans un endroit donné et un `Tweet` qui peut avoir jusqu'à 280 caractères maximum et des métadonnées qui indiquent si cela est un nouveau tweet, un retweet, ou une réponse à un autre tweet.

Nous voulons construire une crate de bibliothèque `agregateur` pour des agrégateurs de médias qui peut afficher le résumé des données stockées dans une instance de `ArticleDePresse` ou de `Tweet`. Pour cela, il nous faut un résumé pour chaque type, et nous allons demander ce résumé en appelant la méthode `resumer` sur une instance. L'encart 10-12 nous montre la définition d'un trait public `Resumable` qui décrit ce comportement.

Fichier : src/lib.rs

```
pub trait Resumable {
    fn resumer(&self) -> String;
}
```

Encart 10-12 : un trait `Resumable` qui représente le comportement fourni par une méthode `resumer`

Ici, nous déclarons un trait en utilisant le mot-clé `trait` et ensuite le nom du trait, qui est `Resumable` dans notre cas. Nous avons aussi déclaré le trait comme `pub` afin que les crates qui dépendent de cette crate puissent aussi utiliser ce trait, comme nous allons le voir dans quelques exemples. Entre les accolades, nous déclarons la signature de la méthode qui décrit le comportement des types qui implémentent ce trait, qui est dans notre cas `fn resumer(&self) -> String`.

A la fin de la signature de la méthode, au lieu de renseigner une implémentation entre des accolades, nous utilisons un point-virgule. Chaque type qui implémente ce trait doit renseigner son propre comportement dans le corps de la méthode. Le compilateur va s'assurer que tous les types qui ont le trait `Resumable` auront la méthode `resumer` définie avec cette signature précise.

Un trait peut avoir plusieurs méthodes dans son corps : les signatures des méthodes sont ajoutées ligne par ligne et chaque ligne se termine avec un point-virgule.

Implémenter un trait sur un type

Maintenant que nous avons défini les signatures souhaitées des méthodes du trait `Resumable`, nous pouvons maintenant l'implémenter sur les types de notre agrégateur de médias. L'encart 10-13 montre une implémentation du trait `Resumable` sur la structure `ArticleDePresse` qui utilise le titre, le nom de l'auteur et le lieu pour créer la valeur de retour de `resumer`. Pour la structure `Tweet`, nous définissons `resumer` avec le nom d'utilisateur suivi par le texte entier du tweet, en supposant que le contenu du tweet est déjà limité à 280 caractères.

Fichier : src/lib.rs

```
pub struct ArticleDePresse {
    pub titre: String,
    pub lieu: String,
    pub auteur: String,
    pub contenu: String,
}

impl Resumable for ArticleDePresse {
    fn resumer(&self) -> String {
        format!("{}, par {} ({})", self.titre, self.auteur, self.lieu)
    }
}

pub struct Tweet {
    pub nom_utilisateur: String,
    pub contenu: String,
    pub reponse: bool,
    pub retweet: bool,
}

impl Resumable for Tweet {
    fn resumer(&self) -> String {
        format!("{} : {}", self.nom_utilisateur, self.contenu)
    }
}
```

Encart 10-13 : implémentation du trait `Resumable` sur les types `ArticleDePresse` et `Tweet`

L'implémentation d'un trait sur un type est similaire à l'implémentation d'une méthode classique. La différence est que nous ajoutons le nom du trait que nous voulons implémenter après le `impl`, et que nous utilisons ensuite le mot-clé `for` suivi du nom du type sur lequel nous souhaitons implémenter le trait. À l'intérieur du bloc `impl`, nous ajoutons les signatures des méthodes présentes dans la définition du trait. Au lieu d'ajouter un point-virgule après chaque signature, nous plaçons les accolades et on remplit le corps de la méthode avec le comportement spécifique que nous voulons que les méthodes du trait suivent pour le type en question.

Maintenant que la bibliothèque a implémenté le trait `Resumable` sur `ArticleDePresse` et `Tweet`, les utilisateurs de cette crate peuvent appeler les méthodes de l'instance de `ArticleDePresse` et `Tweet` comme si elles étaient des méthodes classiques. La seule différence est que le trait ainsi que les types doivent être introduits dans la portée pour obtenir les méthodes de trait additionnelles. Voici un exemple de comment la crate binaire pourra utiliser notre crate de bibliothèque `agregateur` :

```
use agregateur::{Resumable, Tweet};

fn main() {
    let tweet = Tweet {
        nom_utilisateur: String::from("jean"),
        contenu: String::from("Bien sûr, les amis, comme vous le savez
probablement déjà"),
        reponse: false,
        retweet: false,
    };

    println!("1 nouveau tweet : {}", tweet.resumer());
}
```

Ce code affichera `1 nouveau tweet : jean : Bien sûr, les amis, comme vous le savez probablement déjà .`

Les autres crates qui dépendent de la crate `agregateur` peuvent aussi importer dans la portée le trait `Resumable` afin d'implémenter le trait sur leurs propres types. Il y a une limitation à souligner avec l'implémentation des traits, c'est que nous ne pouvons implémenter un trait sur un type qu'à condition qu'au moins le trait ou le type soit défini localement dans notre crate. Par exemple, nous pouvons implémenter des traits de la bibliothèque standard comme `Display` sur un type personnalisé comme `Tweet` comme une fonctionnalité de notre crate `agregateur` , car le type `Tweet` est défini localement dans notre crate `agregateur` . Nous pouvons aussi implémenter `Resumable` sur `Vec<T>` dans notre crate `agregateur` , car le trait `Resumable` est défini localement dans notre crate `agregateur` .

Mais nous ne pouvons pas implémenter des traits externes sur des types externes. Par exemple, nous ne pouvons pas implémenter le trait `Display` sur `Vec<T>` à l'intérieur de notre crate `agregateur` , car `Display` et `Vec<T>` sont définis dans la bibliothèque standard et ne sont donc pas définis localement dans notre crate `agregateur` . Cette limitation fait partie d'une propriété des programmes que l'on appelle la *cohérence*, et plus précisément la *règle de l'orphelin*, qui s'appelle ainsi car le type parent n'est pas présent. Cette règle s'assure que le code des autres personnes ne casse pas votre code et réciproquement. Sans cette règle, deux crates pourraient implémenter le même trait sur le même type, et Rust ne saurait pas quelle implémentation utiliser.

Implémentations par défaut

Il est parfois utile d'avoir un comportement par défaut pour toutes ou une partie des méthodes d'un trait plutôt que de demander l'implémentation de toutes les méthodes sur chaque type. Ainsi, si nous implémentons le trait sur un type particulier, nous pouvons

garder ou réécrire le comportement par défaut de chaque méthode.

L'encart 10-14 nous montre comment préciser une String par défaut pour la méthode `resumer` du trait `Resumable` plutôt que de définir uniquement la signature de la méthode, comme nous l'avons fait dans l'encart 10-12.

Fichier : src/lib.rs

```
pub trait Resumable {
    fn resumer(&self) -> String {
        String::from("(En savoir plus ...)")
    }
}
```

Encart 10-14 : définition du trait `Resumable` avec une implémentation par défaut de la méthode `resumer`

Pour utiliser l'implémentation par défaut pour résumer des instances de `ArticleDePresse` au lieu de préciser une implémentation personnalisée, nous précisons un bloc `impl` vide avec `impl Resumable for ArticleDePresse {}`.

Même si nous ne définissons plus directement la méthode `resumer` sur `ArticleDePresse`, nous avons fourni une implémentation par défaut et précisé que `ArticleDePresse` implémente le trait `Resumable`. Par conséquent, nous pouvons toujours appeler la méthode `resumer` sur une instance de `ArticleDePresse`, comme ceci :

```
    let article = ArticleDePresse {
        titre: String::from("Les Penguins ont remporté la Coupe Stanley !"),
        lieu: String::from("Pittsburgh, PA, USA"),
        auteur: String::from("Iceburgh"),
        contenu: String::from(
            "Les Penguins de Pittsburgh sont une nouvelle fois la meilleure \
            équipe de hockey de la LNH.",
        ),
    };

    println!("Nouvel article disponible ! {}", article.resumer());
```

Ce code va afficher `Nouvel article disponible ! (En savoir plus ...)`.

La création d'une implémentation par défaut pour `resumer` n'a pas besoin que nous modifiions quelque chose dans l'implémentation de `Resumable` sur `Tweet` dans l'encart 10-13. C'est parce que la syntaxe pour réécrire l'implémentation par défaut est la même que la syntaxe pour implémenter une méthode qui n'a pas d'implémentation par défaut.

Les implémentations par défaut peuvent appeler d'autres méthodes du même trait, même

si ces autres méthodes n'ont pas d'implémentation par défaut. Ainsi, un trait peut fournir de nombreuses fonctionnalités utiles et n'exiger du développeur qui l'utilise que d'en implémenter une petite partie. Par exemple, nous pouvons définir le trait `Resumable` comme ayant une méthode `resumer_auteur` dont l'implémentation est nécessaire, et ensuite définir une méthode `resumer` qui a une implémentation par défaut qui appelle la méthode `resumer_auteur` :

```
pub trait Resumable {
    fn resumer_auteur(&self) -> String;

    fn resumer(&self) -> String {
        format!("(Lire plus d'éléments de {} ...)", self.resumer_auteur())
    }
}
```

Pour pouvoir utiliser cette version de `Resumable`, nous avons seulement besoin de définir `resumer_auteur` lorsqu'on implémente le trait sur le type :

```
impl Resumable for Tweet {
    fn resumer_auteur(&self) -> String {
        format!("@{}", self.nom_utilisateur)
    }
}
```

Après avoir défini `resumer_auteur`, nous pouvons appeler `resumer` sur des instances de la structure `Tweet`, et l'implémentation par défaut de `resumer` va appeler `resumer_auteur`, que nous avons défini. Comme nous avons implémenté `resumer_auteur`, le trait `Resumable` nous a donné le comportement de la méthode `resumer` sans nous obliger à écrire une ligne de code supplémentaire.

```
    let tweet = Tweet {
        nom_utilisateur: String::from("jean"),
        contenu: String::from("Bien sûr, les amis, comme vous le savez
probablement déjà"),
        reponse: false,
        retweet: false,
    };

    println!("1 nouveau tweet : {}", tweet.resumer());
```

Ce code affichera `1 nouveau tweet : (Lire plus d'éléments de @jean ...)`.

Notez qu'il n'est pas possible d'appeler l'implémentation par défaut à partir d'une réécriture de cette même méthode.

Des traits en paramètres

Maintenant que vous savez comment définir et implémenter les traits, nous pouvons regarder comment utiliser les traits pour définir des fonctions qui acceptent plusieurs types différents.

Par exemple, dans l'encart 10-13, nous avons implémenté le trait `Resumable` sur les types `ArticleDePresse` et `Tweet`. Nous pouvons définir une fonction `notifier` qui va appeler la méthode `resumer` sur son paramètre `element`, qui est d'un type qui implémente le trait `Resumable`. Pour faire ceci, nous pouvons utiliser la syntaxe `impl Trait`, comme ceci :

```
pub fn notifier(element: &impl Resumable) {
    println!("Flash info ! {}", element.resumer());
}
```

Au lieu d'un type concret pour le paramètre `element`, nous précisons le mot-clé `impl` et le nom du trait. Ce paramètre accepte n'importe quel type qui implémente le trait spécifié. Dans le corps de `notifier`, nous pouvons appeler toutes les méthodes sur `element` qui proviennent du trait `Resumable`, comme `resumer`. Nous pouvons appeler `notifier` et passer une instance de `ArticleDePresse` ou de `Tweet`. Le code qui appellera la fonction avec un autre type, comme une `String` ou un `i32`, ne va pas se compiler car ces types n'implémentent pas `Resumable`.

La syntaxe du trait lié

La syntaxe `impl Trait` fonctionne bien pour des cas simples, mais est en réalité du sucre syntaxique pour une forme plus longue, qui s'appelle le *trait lié*, qui ressemble à ceci :

```
pub fn notifier<T: Resumable>(element: &T) {
    println!("Flash info ! {}", element.resumer());
}
```

Cette forme plus longue est équivalente à l'exemple dans la section précédente, mais est plus verbeuse. Nous plaçons les traits liés dans la déclaration des paramètres de type génériques après un deux-point entre des chevrons.

La syntaxe `impl Trait` est pratique pour rendre du code plus concis dans des cas simples. La syntaxe du trait lié exprime plus de complexité dans certains cas. Par exemple, nous pouvons avoir deux paramètres qui implémentent `Resumable`. En utilisant la syntaxe `impl Trait`, nous aurons ceci :

```
pub fn notifier(element1: &impl Resumable, element2: &impl Resumable) {
```

Si nous souhaitons permettre à `element1` et `element2` d'avoir des types différents, l'utilisation de `impl Trait` est appropriée (du moment que chacun de ces types implémentent `Resumable`). Mais si nous souhaitons forcer les deux paramètres à être du même type, cela n'est possible à exprimer qu'avec un trait lié, comme ceci :

```
pub fn notifier<T: Resumable>(element1: &T, element2: &T) {
```

Le type générique `T` renseigné comme type des paramètres `element1` et `element2` contraint la fonction de manière à ce que les types concrets des valeurs passées en arguments pour `element1` et `element2` soient identiques.

Renseigner plusieurs traits liés avec la syntaxe `+`

Nous pouvons aussi préciser que nous attendons plus d'un trait lié. Imaginons que nous souhaitons que `notifier` utilise le formatage d'affichage sur `element` ainsi que la méthode `resumer` : nous indiquons dans la définition de `notify` que `element` doit implémenter à la fois `Display` et `Resumable`. Nous pouvons faire ceci avec la syntaxe `+` :

```
pub fn notifier(element: &(impl Resumable + Display)) {
```

La syntaxe `+` fonctionne aussi avec les traits liés sur des types génériques :

```
pub fn notifier<T: Resumable + Display>(element: &T) {
```

Avec les deux traits liés renseignés, le corps de `notifier` va appeler `resumer` et utiliser `{}` pour formater `element`.

Des traits liés plus clairs avec la clause `where`

L'utilisation de trop nombreux traits liés a aussi ses désavantages. Chaque type générique a ses propres traits liés, donc les fonctions avec plusieurs paramètres de type génériques peuvent aussi avoir de nombreuses informations de traits liés entre le nom de la fonction et la liste de ses paramètres, ce qui rend la signature de la fonction difficile à lire. Pour cette raison, Rust a une syntaxe alternative pour renseigner les traits liés, dans une clause `where` après la signature de la fonction. Donc, au lieu d'écrire ceci ...

```
fn une_fonction<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

... nous pouvons utiliser la clause `where`, comme ceci :

```
fn une_fonction<T, U>(t: &T, u: &U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

La signature de cette fonction est moins encombrée : le nom de la fonction, la liste des paramètres et le type de retour sont plus proches les uns des autres, comme une fonction sans traits liés.

Retourner des types qui implémentent des traits

Nous pouvons aussi utiliser la syntaxe `impl Trait` à la place du type de retour afin de retourner une valeur d'un type qui implémente un trait, comme ci-dessous :

```
fn retourne_resumable() -> impl Resumable {
    Tweet {
        nom_utilisateur: String::from("jean"),
        contenu: String::from("Bien sûr, les amis, comme vous le savez
probablement déjà"),
        reponse: false,
        retweet: false,
    }
}
```

En utilisant `impl Resumable` pour le type de retour, nous indiquons que la fonction `retourne_resumable` retourne un type qui implémente le trait `Resumable` sans avoir à écrire le nom du type concret. Dans notre cas, `retourne_resumable` retourne un `Tweet`, mais le code qui appellera cette fonction ne le saura pas.

La capacité de retourner un type qui est uniquement caractérisé par le trait qu'il implémente est tout particulièrement utile dans le cas des fermetures et des itérateurs, que nous verrons au chapitre 13. Les fermetures et les itérateurs créent des types que seul le compilateur est en mesure de comprendre ou alors des types qui sont très longs à définir. La syntaxe `impl Trait` vous permet de renseigner de manière concise qu'une fonction retourne un type particulier qui implémente le trait `Iterator` sans avoir à écrire un très long type.

Cependant, vous pouvez seulement utiliser `impl Trait` si vous retournez un seul type possible. Par exemple, ce code va retourner soit un `ArticleDePresse`, soit un `Tweet`, alors que le type de retour avec `impl Resumable` ne va pas fonctionner :

```
fn retourne_resumable(estArticle: bool) -> impl Resumable {
    if estArticle {
        ArticleDePresse {
            titre: String::from("Les Penguins ont remporté la Coupe Stanley !"),
            lieu: String::from("Pittsburgh, PA, USA"),
            auteur: String::from("Iceburgh"),
            contenu: String::from("Les Penguins de Pittsburgh sont une nouvelle
fois la \
            meilleure équipe de hockey de la LNH."),
        }
    } else {
        Tweet {
            nom_utilisateur: String::from("jean"),
            contenu: String::from("Bien sûr, les amis, comme vous le savez
probablement déjà"),
            reponse: false,
            retweet: false,
        }
    }
}
```

Retourner soit un `ArticleDePresse`, soit un `Tweet` n'est pas autorisé à cause des restrictions sur la façon dont la syntaxe `impl Trait` est implémentée dans le compilateur. Nous verrons comment écrire une fonction avec ce comportement dans une section du chapitre 17.

Corriger la fonction `le_plus_grand` avec les traits liés

Maintenant que vous savez comment renseigner le comportement que vous souhaitez utiliser en utilisant les traits liés des paramètres de type génériques, retournons à l'encart 10-5 pour corriger la définition de la fonction `le_plus_grand` qui utilise un paramètre de type générique ! La dernière fois que nous avons essayé de lancer ce code, nous avions l'erreur suivante :

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:17
  |
5 |         if element > le_plus_grand {
  |            ------- ^ ------------- T
  |            |
  |            T
  |
help: consider restricting type parameter `T`
  |
1 | fn le_plus_grand<T: std::cmp::PartialOrd>(liste: &[T]) -> T {
  |                   ++++++++++++++++++++++

For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` due to previous error
```

Dans le corps de `le_plus_grand`, nous voulions comparer les deux valeurs du type `T` en utilisant l'opérateur *plus grand que* (`>`). Comme cet opérateur est défini comme une méthode par défaut dans le trait de la bibliothèque standard `std::cmp::PartialOrd`, nous devons préciser `PartialOrd` dans les traits liés pour `T` afin que la fonction `le_plus_grand` puisse fonctionner sur les slices de n'importe quel type que nous pouvons comparer. Nous n'avons pas besoin d'importer `PartialOrd` dans la portée car il est importé dans l'étape préliminaire. Changez la signature de `le_plus_grand` par quelque chose comme ceci :

```
fn le_plus_grand<T: PartialOrd>(liste: &[T]) -> T {
```

Cette fois, lorsque nous allons compiler le code, nous aurons un ensemble d'erreurs différent :

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0508]: cannot move out of type `[T]`, a non-copy slice
 -- > src/main.rs:2:23
  |
2 |     let mut le_plus_grand = liste[0];
  |                             ^^^^^^^^
  |                             |
  |                             cannot move out of here
  |                             move occurs because `liste[_]` has type `T`,
which does not implement the `Copy` trait
  |                             help: consider borrowing here: `&liste[0]`

error[E0507]: cannot move out of a shared reference
 -- > src/main.rs:4:18
  |
4 |     for &element in liste {
  |         --------    ^^^^^
  |         ||
  |         |data moved here
  |         |move occurs because `element` has type `T`, which does not
implement the `Copy` trait
  |         help: consider removing the `&`: `element`

Some errors have detailed explanations: E0507, E0508.
For more information about an error, try `rustc --explain E0507`.
error: could not compile `chapter10` due to 2 previous errors
```

L'élement-clé dans ces erreurs est `cannot move out of type [T], a non-copy slice` (*impossible de déplacer une valeur hors du type `[T]`, slice non `Copy`*). Avec notre version non générique de la fonction `le_plus_grand`, nous avions essayé de trouver le plus grand `i32` ou `char`. Comme nous l'avons vu dans la section "Données uniquement sur la pile : la copie" du chapitre 4, les types comme `i32` et `char` ont une taille connue et peuvent être stockés sur la pile, donc ils implémentent le trait `Copy`. Mais quand nous avons rendu générique la fonction `le_plus_grand`, il est devenu possible que le paramètre `liste` contienne des types qui n'implémentent pas le trait `Copy`. Par conséquent, nous ne pouvons pas forcément déplacer la valeur de `list[0]` dans notre variable `le_plus_grand`, ce qui engendre cette erreur.

Pour pouvoir appeler ce code avec seulement les types qui implémentent le trait `Copy`, nous pouvons ajouter `Copy` aux traits liés de `T` ! L'encart 10-15 nous montre le code complet d'une fonction générique `le_plus_grand` qui va se compiler tant que le type des valeurs dans la slice que nous passons dans la fonction implémente les traits `PartialOrd` *et* `Copy`, comme le font `i32` et `char`.

Fichier : src/main.rs

```
fn le_plus_grand<T: PartialOrd + Copy>(liste: &[T]) -> T {
    let mut le_plus_grand = liste[0];

    for &element in liste {
        if element > le_plus_grand {
            le_plus_grand = element;
        }
    }

    le_plus_grand
}

fn main() {
    let liste_de_nombres = vec![34, 50, 25, 100, 65];

    let resultat = le_plus_grand(&liste_de_nombres);
    println!("Le nombre le plus grand est {}", resultat);

    let liste_de_caracteres = vec!['y', 'm', 'a', 'q'];

    let resultat = le_plus_grand(&liste_de_caracteres);
    println!("Le plus grand caractère est {}", resultat);
}
```

Encart 10-15 : une définition de la fonction `le_plus_grand` qui fonctionne et s'applique sur n'importe quel type générique qui implémente les traits `PartialOrd` et `Copy`

Si nous ne souhaitons pas restreindre la fonction `le_plus_grand` aux types qui implémentent le trait `Copy`, nous pouvons préciser que `T` a le trait lié `Clone` plutôt que `Copy`. Ainsi, nous pouvons cloner chaque valeur dans la slice lorsque nous souhaitons que la fonction `le_plus_grand` en prenne possession. L'utilisation de la fonction `clone` signifie que nous allons potentiellement allouer plus d'espace sur le tas dans le cas des types qui possèdent des données sur le tas, comme `String`, et les allocations sur le tas peuvent être lentes si nous travaillons avec des grandes quantités de données.

Une autre façon d'implémenter `le_plus_grand` est de faire en sorte que la fonction retourne une référence à une valeur `T` de la slice. Si nous changeons le type de retour en `&T` à la place de `T` et que nous adaptons le corps de la fonction afin de retourner une référence, nous n'aurions alors plus besoin des traits liés `Clone` ou `Copy` et nous pourrions ainsi éviter l'allocation sur le tas. Essayez d'implémenter ces solutions alternatives par vous-même ! Si vous bloquez sur des erreurs à propos des durées de vie *(lifetimes)*, lisez la suite : la section suivante, “La conformité des références avec les durées de vies” vous expliquera cela, mais les durées de vie ne sont pas nécessaires pour résoudre ces exercices.

Utiliser les traits liés pour conditionner l'implémentation des méthodes

En utilisant un trait lié avec un bloc `impl` qui utilise les paramètres de type génériques, nous pouvons implémenter des méthodes en fonction des types qui implémentent des traits particuliers. Par exemple, le type `Paire<T>` de l'encart 10-16 implémente toujours la fonction `new` pour retourner une nouvelle instance de `Paire<T>` (pour rappel dans la section "Définir des méthodes" du chapitre 5 que `Self` est un alias de type pour le type du bloc `impl`, qui est dans ce cas le `Paire<T>`). Mais dans le bloc `impl` suivant, `Paire<T>` implémente la méthode `afficher_comparaison` uniquement si son type interne `T` implémente le trait `PartialOrd` qui active la comparaison *et* le trait `Display` qui permet l'affichage.

Fichier : src/lib.rs

```
use std::fmt::Display;

struct Paire<T> {
    x: T,
    y: T,
}

impl<T> Paire<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Paire<T> {
    fn afficher_comparaison(&self) {
        if self.x >= self.y {
            println!("Le plus grand élément est x = {}", self.x);
        } else {
            println!("Le plus grand élément est y = {}", self.y);
        }
    }
}
```

Encart 10-16 : implémentation de méthodes sur un type générique en fonction des traits liés

Nous pouvons également implémenter un trait sur tout type qui implémente un autre trait en particulier. L'implémentation d'un trait sur n'importe quel type qui a un trait lié est appelée *implémentation générale* et est largement utilisée dans la bibliothèque standard Rust. Par exemple, la bibliothèque standard implémente le trait `ToString` sur tous les types qui implémentent le trait `Display`. Le bloc `impl` de la bibliothèque standard ressemble au code suivant :

```
impl<T: Display> ToString for T {
    // -- partie masquée ici --
}
```

Comme la bibliothèque standard a cette implémentation générale, nous pouvons appeler la méthode `to_string` définie par le trait `ToString` sur n'importe quel type qui implémente le trait `Display`. Par exemple, nous pouvons transformer les nombres entiers en leur équivalent dans une `String` comme ci-dessous car les entiers implémentent `Display` :

```
let s = 3.to_string();
```

Les implémentations générales sont décrites dans la documentation du trait, dans la section “Implementors”.

Les traits et les traits liés nous permettent d'écrire du code qui utilise des paramètres de type génériques pour réduire la duplication de code, mais aussi pour indiquer au compilateur que nous voulons que le type générique ait un comportement particulier. Le compilateur peut ensuite utiliser les informations liées aux traits pour vérifier que tous les types concrets utilisés dans notre code suivent le comportement souhaité. Dans les langages typés dynamiquement, nous aurions une erreur à l'exécution si nous appelions une méthode sur un type qui n'implémenterait pas la méthode. Mais Rust décale l'apparition de ces erreurs au moment de la compilation afin de nous forcer à résoudre les problèmes avant même que notre code soit capable de s'exécuter. De plus, nous n'avons pas besoin d'écrire un code qui vérifie le comportement lors de l'exécution car nous l'avons déjà vérifié au moment de la compilation. Cela permet d'améliorer les performances sans avoir à sacrifier la flexibilité des types génériques.

Une autre sorte de générique que nous avons déjà utilisée est la *durée de vie*. Plutôt que de s'assurer qu'un type a le comportement que nous voulons, la durée de vie s'assure que les références sont en vigueur aussi longtemps que nous avons besoin qu'elles le soient. Nous allons voir à la page suivante comment la durée de vie fait cela.

La conformité des références avec les durées de vies

Il reste un détail que nous n'avons pas abordé dans la section “Les références et l'emprunt” du chapitre 4, c'est que toutes les références ont une *durée de vie* dans Rust, qui est la portée pour laquelle cette référence est en vigueur. La plupart du temps, les durées de vies sont implicites et sont déduites automatiquement, comme pour la plupart du temps les types sont déduits. Nous devons renseigner le type lorsque plusieurs types sont possibles. De la même manière, nous devons renseigner les durées de vie lorsque les durées de vies des références peuvent être déduites de différentes manières. Rust nécessite que nous renseignons ces relations en utilisant des paramètres de durée de vie génériques pour s'assurer que les références utilisées au moment de la compilation restent bien en vigueur.

L'annotation de la durée de vie n'est pas un concept présent dans la pluspart des langages de programmation, donc cela n'est pas très familier. Bien que nous ne puissions couvrir l'intégralité de la durée de vie dans ce chapitre, nous allons voir les cas les plus courants où vous allez rencontrer la syntaxe de la durée de vie, pour vous introduire ces concept.

Eviter les références pendouillantes avec les durées de vie

L'objectif principal des durées de vies est d'éviter les références pendouillantes qui font qu'un programme pointe des données autres que celles sur lesquelles il était censé pointer. Soit le programme de l'encart 10-17, qui a une portée externe et une portée interne.

```
{
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

Encart 10-17 : tentative d'utiliser une référence vers une valeur qui est sortie de la portée

> Remarque : Les exemples dans les encarts 10-17, 10-18 et 10-24 déclarent des variables sans initialiser leur valeur, donc les noms de ces variables existent dans la portée externe. A première vue, cela semble être en conflit avec le fonctionnement de Rust qui n'utilise pas les valeurs nulles. Cependant, si nous essayons d'utiliser une variable avant de lui donner une valeur, nous aurons une erreur au moment de la compilation, qui confirme que Rust ne fonctionne pas avec des valeurs nulles.

La portée externe déclare une variable `r` sans valeur initiale, et la portée interne déclare une variable `x` avec la valeur initiale à `5`. Au sein de la portée interne, nous essayons d'assigner la valeur de `r` comme étant une référence à `x`. Puis la portée interne se ferme, et nous essayons d'afficher la valeur dans `r`. Ce code ne va pas se compiler car la valeur `r` se réfère à quelque chose qui est sorti de la portée avant que nous essayons de l'utiliser. Voici le message d'erreur :

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `x` does not live long enough
  --> src/main.rs:7:17
   |
7  |             r = &x;
   |                 ^^ borrowed value does not live long enough
8  |         }
   |         - `x` dropped here while still borrowed
9  |
10 |         println!("r: {}", r);
   |                           - borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

La variable `x` n'existe plus (“does not live long enough”). La raison à cela est que `x` est sortie de la portée lorsque la portée interne s'est fermée à la ligne 7. Mais `r` reste en vigueur dans la portée externe ; car sa portée est plus grande, on dit qu'il “vit plus longtemps”. Si Rust avait permis à ce code de s'exécuter, `r` pointerait sur de la mémoire désallouée dès que `x` est sortie de la portée, ainsi tout ce que nous pourrions faire avec `r` ne fonctionnerait pas correctement. Mais comment Rust détecte que ce code est invalide ? Il utilise le vérificateur d'emprunt.

Le vérificateur d'emprunt

Le compilateur de Rust embarque un *vérificateur d'emprunt* (borrow checker) qui compare les portées pour déterminer si les emprunts sont valides. L'encart 10-18 montre le même

code que l'encart 10-17, mais avec des commentaires qui montrent les durées de vies des variables.

```
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

Encart 10-18 : commentaires pour montrer les durées de vie de `r` et `x`, qui s'appellent respectivement `'a` et `'b`

Ici, nous avons montré la durée de vie de `r` avec `'a` et la durée de vie de `x` avec `'b`. Comme vous pouvez le constater, le bloc interne `'b` est bien plus petit que le bloc externe `'a`. Au moment de la compilation, Rust compare les tailles des deux durées de vies et constate que `r` a la durée de vie `'a` mais fait référence à de la mémoire qui a une durée de vie de `'b`. Ce programme est refusé car `'b` est plus court que `'a` : l'élément pointé par la référence n'existe pas aussi longtemps que la référence.

L'encart 10-19 résout le code afin qu'il n'ait plus de référence pendouillante et qu'il se compile sans erreur.

```
{
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

Encart 10-19 : la référence est valide puisque la donnée a une durée de vie plus longue que la référence

Ici, `x` a la durée de vie `'b`, qui est plus grande dans ce cas que `'a`. Cela signifie que `r` peut référencer `x` car Rust sait que la référence présente dans `r` sera toujours valide du moment que `x` est en vigueur.

Maintenant que vous savez où se situent les durées de vie des références et comment Rust analyse les durées de vies pour s'assurer que les références soient toujours en vigueur, découvrons les durées de vies génériques des paramètres et des valeurs de retour dans le

cas des fonctions.

Les durées de vies génériques dans les fonctions

Ecrivons une fonction qui retourne la plus longue des slice d'une chaîne de caractères. Cette fonction va prendre en argument deux slices de chaîne de caractères et retourner une slice d'une chaîne de caractères. Après avoir implémenté la fonction `la_plus_longue`, le code de l'encart 10-20 devrait afficher `La plus grande chaîne est abcd`.

Fichier : src/main.rs

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let resultat = la_plus_longue(string1.as_str(), string2);
    println!("La plus grande chaîne est {}", resultat);
}
```

Encart 10-20 : une fonction `main` qui appelle la fonction `la_plus_longue` pour trouver la plus grande des deux slices de chaîne de caractères

Remarquez que nous souhaitons que la fonction prenne deux slices de chaînes de caractères, qui sont des références, car nous ne voulons pas que la fonction `la_plus_longue` prenne possession de ses paramètres. Rendez-vous à la section "Les slices de chaînes de caractères en paramètres" du chapitre 4 pour savoir pourquoi nous utilisons ce type de paramètres dans l'encart 10-20.

Si nous essayons d'implémenter la fonction `la_plus_longue` comme dans l'encart 10-21, cela ne va pas se compiler.

Fichier : src/main.rs

```
fn la_plus_longue(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Encart 10-21 : une implémentation de la fonction `la_plus_longue` qui retourne la plus longue des deux slices de chaînes de caractères, mais ne se compile pas encore

A la place, nous obtenons l'erreur suivante qui nous parle de durées de vie :

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn la_plus_longue(x: &str, y: &str) -> &str {
  |                      ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn la_plus_longue<'a>(x: &'a str, y: &'a str) -> &'a str {
  |                   ++++     ++          ++          ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `chapter10` due to previous error
```

La partie “help” nous explique que le type de retour a besoin d'un paramètre de durée de vie générique car Rust ne sait pas si la référence retournée est liée à `x` ou à `y`. Pour le moment, nous ne le savons pas nous non plus, car le bloc `if` dans le corps de cette fonction retourne une référence à `x` et le bloc `else` retourne une référence à `y` !

Lorsque nous définissons cette fonction, nous ne connaissons pas les valeurs concrètes qui vont passer dans cette fonction, donc nous ne savons pas si nous allons exécuter le cas du `if` ou du `else`. Nous ne connaissons pas non plus les durées de vie des références qui vont passer dans la fonction, donc nous ne pouvons pas vérifier les portées comme nous l'avons fait dans les encarts 10-18 et 10-19 pour déterminer si la référence que nous allons retourner sera toujours en vigueur. Le vérificateur d'emprunt ne va pas pouvoir non plus déterminer cela, car il ne sait comment les durées de vie de `x` et de `y` sont reliées à la durée de vie de la valeur de retour. Pour résoudre cette erreur, nous allons ajouter des paramètres de durée de vie génériques qui définissent la relation entre les références, afin que le vérificateur d'emprunt puisse faire cette analyse.

La syntaxe pour annoter les durées de vies

L'annotation des durées de vie ne change pas la longueur de leur durée de vie. De la même façon qu'une fonction accepte n'importe quel type lorsque la signature utilise un paramètre de type générique, les fonctions peuvent accepter des références avec n'importe quelle durée de vie en précisant un paramètre de durée de vie générique. L'annotation des durées de vie décrit la relation des durées de vies de plusieurs références entre elles sans influencer les durées de vie.

L'annotation des durées de vies a une syntaxe un peu inhabituelle : le nom des paramètres de durées de vies doit commencer par une apostrophe (`'`) et est habituellement en minuscule et très court, comme les types génériques. La plupart des personnes utilisent le nom `'a` . Nous plaçons le paramètre de type après le `&` d'une référence, en utilisant un espace pour séparer l'annotation du type de la référence.

Voici quelques exemples : une référence à un `i32` sans paramètre de durée de vie, une référence à un `i32` qui a un paramètre de durée de vie `'a` , et une référence mutable à un `i32` qui a aussi la durée de vie `'a` .

```
&i32        // une référence
&'a i32     // une référence avec une durée de vie explicite
&'a mut i32 // une référence mutable avec une durée de vie explicite
```

Une annotation de durée de vie toute seule n'a pas vraiment de sens, car les annotations sont faites pour indiquer à Rust quels paramètres de durée de vie génériques de plusieurs références sont liés aux autres. Par exemple, disons que nous avons une fonction avec le paramètre `premier` qui est une référence à un `i32` avec la durée de vie `'a` . La fonction a aussi un autre paramètre `second` qui est une autre référence à un `i32` qui a aussi la durée de vie `'a` . Les annotations de durée de vie indiquent que les références `premier` et `second` doivent tous les deux exister aussi longtemps que la durée de vie générique.

Les annotations de durée de vie dans les signatures des fonctions

Maintenant, examinons les annotations de durée de vie dans contexte de la fonction `la_plus_longue` . Comme avec les paramètres de type génériques, nous devons déclarer les paramètres de durée de vie génériques dans des chevrons entre le nom de la fonction et la liste des paramètres. Nous souhaitons contraindre les durées de vie des deux paramètres et la durée de vie de la référence retournée de telle manière que la valeur retournée restera en vigueur tant que les deux paramètres le seront aussi. Nous allons appeler la durée de vie `'a` et ensuite l'ajouter à chaque référence, comme nous le faisons dans l'encart 10-22.

Fichier : src/main.rs

```
fn la_plus_longue<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Encart 10-22 : définition de la fonction `la_plus_longue` qui indique que toutes les références présentes dans la signature doivent avoir la même durée de vie `'a`

Le code devrait se compiler et devrait produire le résultat que nous souhaitions lorsque nous l'utilisions dans la fonction `main` de l'encart 10-20.

La signature de la fonction indique maintenant à Rust que pour la durée de vie `'a`, la fonction prend deux paramètres, les deux étant des slices de chaîne de caractères qui vivent aussi longtemps que la durée de vie `'a`. La signature de la fonction indique également à Rust que la slice de chaîne de caractères qui est retournée par la fonction vivra au moins aussi longtemps que la durée de vie `'a`. Dans la pratique, cela veut dire que durée de vie de la référence retournée par la fonction `la_plus_longue` est la même que celle de la plus petite des durées de vies des références qu'on lui donne. Cette relation est ce que nous voulons que Rust mette en place lorsqu'il analysera ce code.

Souvenez-vous, lorsque nous précisons les paramètres de durée de vie dans la signature de cette fonction, nous ne changeons pas les durées de vies des valeurs qui lui sont envoyées ou qu'elle retourne. Ce que nous faisons, c'est plutôt indiquer au vérificateur d'emprunt qu'il doit rejeter toute valeur qui ne répond pas à ces conditions. Notez que la fonction `la_plus_longue` n'a pas besoin de savoir exactement combien de temps `x` et `y` vont exister, mais seulement que cette portée peut être substituée par `'a`, qui satisfera cette signature.

Lorsqu'on précise les durées de vie dans les fonctions, les annotations se placent dans la signature de la fonction, pas dans le corps de la fonction. Les annotations de durée de vie sont devenues partie intégrante de la fonction, exactement comme les types dans la signature. Avoir des signatures de fonction qui intègrent la durée de vie signifie que l'analyse que va faire le compilateur Rust sera plus simple. S'il y a un problème avec la façon dont la fonction est annotée ou appelée, les erreurs de compilation peuvent pointer plus précisément sur la partie de notre code qui impose ces contraintes. Mais si au contraire, le compilateur Rust avait dû faire plus de suppositions sur ce que nous voulions créer comme lien de durée de vie, le compilateur n'aurait pu qu'évoquer une utilisation de notre code bien plus éloignée de la véritable raison du problème.

Lorsque nous donnons une référence concrète à `la_plus_longue`, la durée de vie concrète qui est modélisée par `'a` est la partie de la portée de `x` qui se chevauche avec la portée de `y`. Autrement dit, la durée vie générique `'a` aura la durée de vie concrète qui est égale à la plus petite des durées de vies entre `x` et `y`. Comme nous avons marqué la référence retournée avec le même paramètre de durée de vie `'a`, la référence retournée sera toujours en vigueur pour la durée de la plus petite des durées de vies de `x` et de `y`.

Regardons comment les annotations de durée de vie restreignent la fonction

`la_plus_longue` en y passant des références qui ont des durées de vies concrètement différentes. L'encart 10-23 en est un exemple.

Fichier : src/main.rs

```
fn main() {
    let string1 = String::from("une longue chaîne est longue");

    {
        let string2 = String::from("xyz");
        let resultat = la_plus_longue(string1.as_str(), string2.as_str());
        println!("La chaîne la plus longue est {}", resultat);
    }
}
```

Encart 10-23 : utilisation de la fonction `la_plus_longue` sur des références à des valeurs `String` qui ont concrètement des durées de vie différentes

Dans cet exemple, `string1` est en vigueur jusqu'à la fin de la portée externe, `string2` n'est valide que jusqu'à la fin de la portée interne, et `resultat` est une référence vers quelque chose qui est en vigueur jusqu'à la fin de la portée interne. Lorsque vous lancez ce code, vous constaterez que le vérificateur d'emprunt accepte ce code ; il va se compiler et afficher `La chaîne la plus longue est une longue chaîne est longue`.

Maintenant, essayons un exemple qui fait en sorte que la durée de vie de la référence dans `resultat` sera plus petite que celles des deux arguments. Nous allons déplacer la déclaration de la variable `resultat` à l'extérieur de la portée interne mais on va laisser l'affectation de la valeur de la variable `resultat` à l'intérieur de la portée de `string2`. Nous allons ensuite déplacer le `println!`, qui utilise `resultat`, à l'extérieur de la portée interne, après que la portée soit terminée. Le code de l'encart 10-24 ne va pas se compiler.

Fichier : src/main.rs

```
fn main() {
    let string1 = String::from("une longue chaîne est longue");
    let resultat;
    {
        let string2 = String::from("xyz");
        resultat = la_plus_longue(string1.as_str(), string2.as_str());
    }
    println!("La chaîne la plus longue est {}", resultat);
}
```

Encart 10-24 : tentative d'utilisation de `resultat` après `string2`, qui est sortie de la portée

Lorsque nous essayons de compiler ce code, nous aurons cette erreur :

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
6 |         result = la_plus_longue(string1.as_str(), string2.as_str());
  |                                                   ^^^^^^^^^^^^^^^^ borrowed
value does not live long enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("La chaîne la plus longue est {}", resultat);
  |                                                  -------- borrow later used
here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

L'erreur explique que pour que `resultat` soit en vigueur pour l'instruction `println!`, `string2` doit toujours être valide jusqu'à la fin de la portée externe. Rust a déduit cela car nous avons précisé les durées de vie des paramètres de la fonction et des valeurs de retour en utilisant le même paramètre de durée de vie `'a`.

En tant qu'humain, nous pouvons lire ce code et constater que `string1` est plus grand que `string2` et ainsi que `resultat` contiendra une référence vers `string1`. Comme `string1` n'est pas encore sorti de portée, une référence vers `string1` sera toujours valide pour l'instruction `println!`. Cependant, le compilateur ne peut pas déduire que la référence est valide dans notre cas. Nous avons dit à Rust que la durée de vie de la référence qui est retournée par la fonction `la_plus_longue` est la même que la plus petite des durées de vie des références qu'on lui passe en argument. C'est pourquoi le vérificateur d'emprunt rejette le code de l'encart 10-24 car il a potentiellement une référence invalide.

Essayez d'expérimenter d'autres situations en variant les valeurs et durées de vie des références passées en argument de la fonction `la_plus_longue`, et aussi pour voir comment on utilise la référence retournée. Faites des hypothèses pour savoir si ces situations vont passer ou non le vérificateur d'emprunt avant que vous ne compiliez ; et vérifiez ensuite si vous aviez raison !

Penser en termes de durées de vie

La façon dont vous avez à préciser les paramètres de durées de vie dépend de ce que fait votre fonction. Par exemple, si nous changions l'implémentation de la fonction `la_plus_longue` pour qu'elle retourne systématiquement le premier paramètre plutôt que la slice de chaîne de caractères la plus longue, nous n'aurions pas besoin de renseigner une

durée de vie sur le paramètre `y`. Le code suivant se compile :

Fichier : src/main.rs

```
fn la_plus_longue<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

Dans cet exemple, nous avons précisé un paramètre de durée de vie `'a` sur le paramètre `x` et sur le type de retour, mais pas sur le paramètre `y`, car la durée de vie de `y` n'a pas de lien avec la durée de vie de `x` ou de la valeur de retour.

Lorsqu'on retourne une référence à partir d'une fonction, le paramètre de la durée de vie pour le type de retour doit correspondre à une des durées des paramètres. Si la référence retournée ne se réfère *pas* à un de ses paramètres, elle se réfère probablement à une valeur créée à l'intérieur de cette fonction, et elle deviendra une référence pendouillante car sa valeur va sortir de la portée à la fin de la fonction. Imaginons cette tentative d'implémentation de la fonction `la_plus_longue` qui ne se compile pas :

Fichier : src/main.rs

```
fn la_plus_longue<'a>(x: &str, y: &str) -> &'a str {
    let resultat = String::from("très longue chaîne");
    resultat.as_str()
}
```

Ici, même si nous avons précisé un paramètre de durée de vie `'a` sur le type de retour, cette implémentation va échouer à la compilation car la durée de vie de la valeur de retour n'est pas du tout liée à la durée de vie des paramètres. Voici le message d'erreur que nous obtenons :

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0515]: cannot return reference to local variable `result`
  --> src/main.rs:11:5
   |
11 |     resultat.as_str()
   |     ^^^^^^^^^^^^^^^^^^ returns a reference to data owned by the current
function

For more information about this error, try `rustc --explain E0515`.
error: could not compile `chapter10` due to previous error
```

Le problème est que `resultat` sort de la portée et est effacée à la fin de la fonction `la_plus_longue`. Nous avons aussi essayé de retourner une référence vers `resultat` à

partir de la fonction. Il n'existe aucune façon d'écrire les paramètres de durée de vie de telle manière que cela changerait la référence pendouillante, et Rust ne nous laissera pas créer une référence pendouillante. Dans notre cas, la meilleure solution consiste à retourner un type de donnée dont on va prendre possession plutôt qu'une référence, ainsi le code appelant sera responsable du nettoyage de la valeur.

Enfin, la syntaxe de la durée de vie sert à interconnecter les durées de vie de plusieurs paramètres ainsi que les valeurs de retour des fonctions. Une fois celles-ci interconnectés, Rust a assez d'informations pour autoriser les opérations sécurisées dans la mémoire et refuser les opérations qui pourraient créer des pointeurs pendouillants ou alors enfreindre la sécurité de la mémoire.

L'ajout des durées de vies dans les définitions des structures

Jusqu'à présent, nous avons défini des structures pour contenir des types qui sont possédés par elles-mêmes. Il est possible qu'une structure puisse contenir des références, mais dans ce cas nous devons préciser une durée de vie sur chaque référence dans la définition de la structure. L'encart 10-25 montre une structure `ExtraitImportant` qui stocke une slice de chaîne de caractères.

Fichier : src/main.rs

```
struct ExtraitImportant<'a> {
    partie: &'a str,
}

fn main() {
    let roman = String::from("Appelez-moi Ismaël. Il y a quelques années ...");
    let premiere_phrase = roman.split('.')
        .next()
        .expect("Impossible de trouver un '.'");
    let i = ExtraitImportant { partie: premiere_phrase };
}
```

Encart 10-25 : une structure qui stocke une référence, par conséquent sa définition a besoin d'une annotation de durée de vie

Cette structure a un champ, `partie`, qui stocke une slice de chaîne de caractères, qui est une référence. Comme pour les types de données génériques, nous déclarons le nom du paramètre de durée de vie générique entre des chevrons après le nom de la structure pour que nous puissions utiliser le paramètre de durée de vie dans le corps de la définition de la structure. Cette annotation signifie qu'une instance de `ExtraitImportant` ne peut pas vivre plus longtemps que la référence qu'elle stocke dans son champ `partie`.

La fonction `main` crée ici une instance de la structure `ExtraitImportant` qui stocke une référence vers la première phrase de la `String` possédée par la variable `roman`. Les données dans `roman` existent avant que l'instance de `ExtraitImportant` soit crée. De plus, `roman` ne sort pas de la portée avant que l'instance de `ExtraitImportant` sorte de la portée, donc la référence dans l'instance de `ExtraitImportant` est toujours valide.

L'élision des durées de vie

Vous avez appris que toute référence a une durée de vie et que vous devez renseigner des paramètres de durée de vie sur des fonctions ou des structures qui utilisent des références. Cependant, dans le chapitre 4 nous avions une fonction dans l'encart 4-9, qui est montrée à nouveau dans l'encart 10-26, qui compilait sans informations de durée de vie.

Fichier : src/lib.rs

```
fn premier_mot(s: &str) -> &str {
    let octets = s.as_bytes();

    for (i, &element) in octets.iter().enumerate() {
        if element == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Encart 10-26 : une fonction que nous avons défini dans l'encart 4-9 qui se compilait sans avoir d'indications sur la durée de vie, même si les paramètres et le type de retour sont des références

La raison pour laquelle cette fonction se compile sans annotation de durée de vie est historique : dans les premières versions de Rust (avant la 1.0), ce code ne se serait pas compilé parce que chaque référence devait avoir une durée de vie explicite. A l'époque, la signature de la fonction devait être écrite ainsi :

```
fn premier_mot<'a>(s: &'a str) -> &'a str {
```

Après avoir écrit une grande quantité de code Rust, l'équipe de Rust s'est rendu compte que les développeurs Rust saisissaient toujours les mêmes durées de vie encore et encore dans des situations spécifiques. Ces situations étaient prévisibles et suivaient des schémas prédéterminés. Les développeurs ont programmé ces schémas dans le code du compilateur afin que le vérificateur d'emprunt puisse deviner les durées de vie dans ces situations et

n'auront plus besoin d'annotations explicites.

Cette partie de l'histoire de Rust est intéressante car il est possible que d'autres modèles prédéterminés émergent et soient ajoutés au compilateur. A l'avenir, il est possible qu'encore moins d'annotations de durée de vie soient nécessaires.

Les schémas programmés dans l'analyse des références de Rust s'appellent les *règles d'élision des durées de vie*. Ce ne sont pas des règles que les développeurs doivent suivre ; c'est un jeu de cas particuliers que le compilateur va essayer de comparer à votre code, et s'il y a une correspondance alors vous n'aurez pas besoin d'écrire explicitement les durées de vie.

Les règles d'élision ne permettent pas de faire des déductions complètes. Si Rust applique les règles de façon stricte, mais qu'il existe toujours une ambiguïté quant à la durée de vie des références, le compilateur ne devinera pas quelle devrait être la durée de vie des autres références. Dans ce cas, au lieu de tenter de deviner, le compilateur va vous afficher une erreur que vous devrez résoudre en précisant les durées de vie qui clarifieront les liens entre chaque référence.

Les durées de vies sur les fonctions ou les paramètres des fonctions sont appelées les *durées de vie des entrées*, et les durées de vie sur les valeurs de retour sont appelées les *durées de vie des sorties*.

Le compilateur utilise trois règles pour déterminer quelles devraient être les durées de vie des références si cela n'est pas indiqué explicitement. La première règle s'applique sur les durées de vie des entrées, et les deuxième et troisième règles s'appliquent sur les durées de vie des sorties. Si le compilateur arrive à la fin des trois règles et qu'il y a encore des références pour lesquelles il ne peut pas savoir leur durée de vie, le compilateur s'arrête avec une erreur. Ces règles s'appliquent sur les définitions des `fn` ainsi que sur celles des blocs `impl`.

La première règle dit que chaque paramètre qui est une référence a sa propre durée de vie. Autrement dit, une fonction avec un seul paramètre va avoir un seul paramètre de durée de vie : `fn foo<'a>(x: &'a i32)` ; une fonction avec deux paramètres va avoir deux paramètres de durée de vie séparés : `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)` ; et ainsi de suite.

La deuxième règle dit que s'il y a exactement un seul paramètre de durée de vie d'entrée, cette durée de vie est assignée à tous les paramètres de durée de vie des sorties : `fn foo<'a>(x: &'a i32) -> &'a i32`.

La troisième règle est que lorsque nous avons plusieurs paramètres de durée de vie, mais qu'un d'entre eux est `&self` ou `&mut self` parce que c'est une méthode, la durée de vie de

`self` sera associée à tous les paramètres de durée de vie des sorties. Cette troisième règle rend les méthodes plus faciles à lire et à écrire car il y a moins de caractères nécessaires.

Imaginons que nous soyons le compilateur. Nous allons appliquer ces règles pour déduire quelles seront les durées de vie des références dans la signature de la fonction `premier_mot` de l'encart 10-26.

```
fn premier_mot(s: &str) -> &str {
```

Le compilateur applique alors la première règle, qui dit que chaque référence a sa propre durée de vie. Appellons-la `'a` comme d'habitude, donc maintenant la signature devient ceci :

```
fn premier_mot<'a>(s: &'a str) -> &str {
```

La deuxième règle s'applique car il y a exactement une durée de vie d'entrée ici. La deuxième règle dit que la durée de vie du seul paramètre d'entrée est affectée à la durée de vie des sorties, donc la signature est maintenant ceci :

```
fn premier_mot<'a>(s: &'a str) -> &'a str {
```

Maintenant, toutes les références de cette signature de fonction ont des durées de vie, et le compilateur peut continuer son analyse sans avoir besoin que le développeur renseigne les durées de vie dans cette signature de fonction.

Voyons un autre exemple, qui utilise cette fois la fonction `la_plus_longue` qui n'avait pas de paramètres de durée de vie lorsque nous avons commencé à l'utiliser dans l'encart 10-21 :

```
fn la_plus_longue(x: &str, y: &str) -> &str {
```

Appliquons la première règle : chaque référence a sa propre durée de vie. Cette fois, nous avons avons deux références au lieu d'une seule, donc nous avons deux durées de vie :

```
fn la_plus_longue<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

Vous pouvez constater que la deuxième règle ne s'applique pas car il y a plus d'une seule durée de vie. La troisième ne s'applique pas non plus, car `la_plus_longue` est une fonction et pas une méthode, donc aucun de ses paramètres ne sont `self` . Après avoir utilisé ces trois règles, nous n'avons pas pu en déduire la durée de vie de la valeur de retour. C'est pourquoi nous obtenons une erreur en essayant de compiler le code dans l'encart 10-21 : le compilateur a utilisé les règles d'élision des durées de vie mais n'est pas capable d'en déduire toutes les durées de vie des références présentes dans la signature.

Comme la troisième règle ne s'applique que sur les signatures des méthodes, nous allons examiner les durées de vie dans ce contexte pour comprendre pourquoi la troisième règle signifie que nous n'avons pas souvent besoin d'annoter les durées de vie dans les signatures des méthodes.

Informations de durée de vie dans les définitions des méthodes

Lorsque nous implémentons des méthodes sur une structure avec des durées de vie, nous utilisons la même syntaxe que celle des paramètres de type génériques que nous avons vue dans l'encart 10-11. L'endroit où nous déclarons et utilisons les paramètres de durée de vie dépend de s'ils sont reliés aux champs des structures ou aux paramètres de la méthode et aux valeurs de retour.

Les noms des durées de vie pour les champs de structure ont toujours besoin d'être déclarés après le mot-clé `impl` et sont ensuite utilisés après le nom de la structure, car ces durées vie font partie du type de la structure.

Sur les signatures des méthodes à l'intérieur du bloc `impl`, les références peuvent être liées à la durée de vie des références de champs de la structure, ou elles peuvent être indépendantes. De plus, les règles d'élision des durées de vie font parfois en sorte que l'ajout de durées de vie n'est parfois pas nécessaire dans les signatures des méthodes. Voyons quelques exemples en utilisant la structure `ExtraitImportant` que nous avons définie dans l'encart 10-25.

Premièrement, nous allons utiliser une méthode `niveau` dont le seul paramètre est une référence à `self` et dont la valeur de retour sera un `i32`, qui n'est pas une référence :

```
impl<'a> ExtraitImportant<'a> {
    fn niveau(&self) -> i32 {
        3
    }
}
```

La déclaration du paramètre de durée de vie après `impl` et son utilisation après le nom du type sont nécessaires, mais nous n'avons pas à préciser la durée de vie de la référence à `self` grâce à la première règle d'élision.

Voici un exemple où la troisième règle d'élision des durées de vie s'applique :

```
impl<'a> ExtraitImportant<'a> {
    fn annoncer_et_retourner_partie(&self, annonce: &str) -> &str {
        println!("Votre attention s'il vous plaît : {}", annonce);
        self.partie
    }
}
```

Il y a deux durées de vies des entrées, donc Rust applique la première règle d'élision des durées de vie et donne à `&self` et `annonce` leur propre durée de vie. Ensuite, comme un des paramètres est `&self`, le type de retour obtient la durée de vie de `&self`, de sorte que toutes les durées de vie ont été calculées.

La durée de vie statique

Une durée de vie particulière que nous devons aborder est `'static`, qui signifie que cette référence *peut* vivre pendant la totalité de la durée du programme. Tous les littéraux de chaînes de caractères ont la durée de vie `'static`, que nous pouvons écrire comme ceci :

```
let s: &'static str = "J'ai une durée de vie statique.";
```

Le texte de cette chaîne de caractères est stocké directement dans le binaire du programme, qui est toujours disponible. C'est pourquoi la durée de vie de tous les littéraux de chaînes de caractères est `'static`.

Il se peut que voyiez des suggestions pour utiliser la durée de vie `'static` dans les messages d'erreur. Mais avant d'utiliser `'static` comme durée de vie pour une référence, demandez-vous si la référence en question vit bien pendant toute la vie de votre programme, ou non. Vous devriez vous demander si vous voulez qu'elle vive aussi longtemps, même si si c'était possible. La plupart du temps, le problème résulte d'une tentative de création d'une référence pendouillante ou d'une inadéquation des durées de vie disponibles. Dans ces cas-là, la solution consiste à résoudre ces problèmes, et pas à renseigner la durée de vie comme étant `'static`.

Les paramètres de type génériques, les traits liés, et les durées de vies ensemble

Regardons brièvement la syntaxe pour renseigner tous les paramètres de type génériques, les traits liés, et les durées de vies sur une seule fonction !

```
use std::fmt::Display;

fn la_plus_longue_avec_annonce<'a, T>(x: &'a str, y: &'a str, ann: T) -> &'a str
    where T: Display
{
    println!("Annonce ! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

C'est la fonction `la_plus_longue` de l'encart 10-22 qui retourne la plus grande de deux slices de chaînes de caractères. Mais maintenant elle a un paramètre supplémentaire `ann` de type générique `T`, qui peut être remplacé par n'importe quel type qui implémente le trait `Display` comme le précise la clause `where`. Ce paramètre supplémentaire sera affiché avec `{}`, c'est pourquoi le trait lié `Display` est nécessaire. Comme les durées de vie sont un type de génériques, les déclarations du paramètre de durée de vie `'a` et le paramètre de type générique `T` vont dans la même liste à l'intérieur des chevrons après le nom de la fonction.

Résumé

Nous avons vu beaucoup de choses dans ce chapitre ! Maintenant que vous en savez plus sur les paramètres de type génériques, les traits et les traits liés, ainsi que sur les paramètres de durée de vie génériques, vous pouvez maintenant écrire du code en évitant les doublons qui va bien fonctionner dans de nombreuses situations. Les paramètres de type génériques vous permettent d'appliquer du code à différents types. Les traits et les traits liés s'assurent que bien que les types soient génériques, ils auront un comportement particulier sur lequel le code peut compter. Vous avez appris comment utiliser les indications de durée de vie pour s'assurer que ce code flexible n'aura pas de références pendouillantes. Et toutes ces vérifications se font au moment de la compilation, ce qui n'influe pas sur les performances au moment de l'exécution du programme !

Croyez-le ou non, mais il y a encore des choses à apprendre sur les sujets que nous avons traités dans ce chapitre : le chapitre 17 expliquera les objets de trait, qui est une façon d'utiliser les traits. Il existe aussi des situations plus complexes impliquant des indications de durée de vie dont vous n'aurez besoin que dans certains cas de figure très avancés; pour ces cas-là, vous devriez consulter la Référence de Rust. Maintenant, nous allons voir au chapitre suivant comment écrire des tests en Rust afin que vous puissiez vous assurer que votre code fonctionne comme il devrait le faire.

Ecrire des tests automatisés

Dans son essai de 1972 “The Humble Programmer”, Edsger W. Dijkstra a dit qu'un “test de programme peut être une manière très efficace de prouver la présence de bogues, mais qu'il est totalement inadéquat pour prouver leur absence”. Mais cela ne veut pas dire que nous ne devrions pas tester notre programme autant que faire se peut !

L'exactitude de nos programmes est le niveau de conformité de notre code par rapport à ce que nous voulons qu'il fasse. Rust est conçu dans un grand souci d'exactitude des programmes, mais l'exactitude est complexe et difficile à confirmer. Le système de type de Rust endosse une grande partie de cette charge, mais le système de type ne peut pas détecter tous les genres d'erreurs. Ainsi, Rust embarque des fonctionnalités pour écrire des tests automatisés de logiciels à l'intérieur du langage.

Par exemple, imaginons que nous écrivons une fonction `ajouter_deux` qui ajoute 2 à n'importe quel nombre qu'on lui envoie. La signature de cette fonction prend un entier en paramètre et retourne un entier comme résultat. Lorsque nous implémentons et compilons cette fonction, Rust fait toutes les vérifications de type et d'emprunt que vous avez apprises précédemment afin de s'assurer que, par exemple, nous ne passions pas une valeur de type `String` ou une référence invalide à cette fonction. Mais Rust *ne peut pas* vérifier que cette fonction va faire précisément ce que nous avions prévu qu'elle fasse, qui en l'occurence est de retourner le paramètre incrémenté de 2 plutôt que d'ajouter 10 ou d'enlever 50, par exemple ! C'est pour cette situation que les tests sont utiles.

Nous pouvons écrire des tests qui vérifient, par exemple, que lorsque nous donnons `3` à la fonction `ajouter_deux`, elle retourne bien `5`. Nous pouvons lancer ces tests à chaque fois que nous modifions notre code pour s'assurer qu'aucun comportement existant et satisfaisant n'a changé.

Les tests restent une discipline complexe : bien que nous ne puissions couvrir chaque détail sur l'écriture de bons tests en un seul chapitre, nous allons découvrir les mécanismes des moyens de test de Rust. Nous allons voir les annotations et les macros que vous pourrez utiliser lorsque vous écrirez vos tests, le comportement par défaut et les options disponibles pour lancer vos tests, et comment organiser les tests en tests unitaires et tests d'intégration.

Comment écrire des tests

Les tests sont des fonctions Rust qui vérifient que le code qui n'est pas un test se comporte bien de la manière attendue. Les corps des fonctions de test effectuent généralement ces trois actions :

1. Initialiser toutes les données ou les états,
2. Lancer le code que vous voulez tester,
3. Vérifier que les résultats correspondent bien à ce que vous souhaitez.

Découvrons les fonctionnalités spécifiques qu'offre Rust pour écrire des tests qui font ces actions, dont l'attribut `test`, quelques macros et l'attribut `should_panic`.

L'anatomie d'une fonction de test

Dans la forme la plus simple, un test en Rust est une fonction qui est marquée avec l'attribut `test`. Les attributs sont des métadonnées sur des parties de code Rust ; un exemple est l'attribut `derive` que nous avons utilisé sur les structures au chapitre 5. Pour transformer une fonction en une fonction de test, il faut ajouter `#[test]` dans la ligne avant le `fn`. Lorsque vous lancez vos tests avec la commande `cargo test`, Rust construit un binaire d'exécution de tests qui exécute les fonctions marquées avec l'attribut `test` et fait un rapport sur quelles fonctions ont réussi ou échoué.

Lorsque nous créons une nouvelle bibliothèque avec Cargo, un module de tests qui contient une fonction de test est automatiquement créé pour nous. Ce module vous aide à démarrer l'écriture de vos tests afin que vous n'ayez pas à chercher la structure et la syntaxe exacte d'une fonction de test à chaque fois que vous débutez un nouveau projet. Vous pouvez ajouter autant de fonctions de test et autant de modules de tests que vous le souhaitez !

Nous allons découvrir quelques aspects du fonctionnement des tests en expérimentant avec le modèle de tests généré pour nous, mais qui ne teste aucun code pour le moment. Ensuite, nous écrirons quelques tests plus proches de la réalité, qui utiliseront du code que nous avons écrit et qui valideront son bon comportement.

Commençons par créer un nouveau projet de bibliothèque que nous appellerons `addition` :

```
$ cargo new addition --lib
     Created library `addition` project
$ cd addition
```

Le contenu de votre fichier *src/lib.rs* dans votre bibliothèque `addition` devrait ressembler à l'encart 11-1.

Fichier : src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let resultat = 2 + 2;
        assert_eq!(resultat, 4);
    }
}
```

Encart 11-1 : le module de test et la fonction générés automatiquement par `cargo new`

Pour l'instant, ignorons les deux premières lignes et concentrons-nous sur la fonction pour voir comment elle fonctionne. Remarquez l'annotation `#[test]` avant la ligne `fn` : cet attribut indique que c'est une fonction de test, donc l'exécuteur de tests sait qu'il doit considérer cette fonction comme étant un test. Nous pouvons aussi avoir des fonctions qui ne font pas de tests dans le module `tests` afin de configurer des scénarios communs ou exécuter des opérations communes, c'est pourquoi nous devons indiquer quelles fonctions sont des tests en utilisant l'attribut `#[test]` .

Le corps de la fonction utilise la macro `assert_eq!` pour vérifier que 2 + 2 vaut bien 4. Cette vérification sert d'exemple pour expliquer le format d'un test classique. Lançons-le pour vérifier si ce test est validé.

La commande `cargo test` lance tous les tests présents dans votre projet, comme le montre l'encart 11-2.

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.57s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Encart 11-2 : le résultat du lancement des tests sur le test généré automatiquement

Cargo a compilé et lancé le test. Après les lignes `Compiling`, `Finished`, et `Running`, on trouve la ligne `running 1 test`. La ligne suivante montre le nom de la fonction de test `it_works`, qui a été générée précédemment, et le résultat de l'exécution de ce test, `ok`. Le résumé général de l'exécution des tests s'affiche ensuite. Le texte `test result: ok.` signifie que tous les tests ont réussi, et la partie `1 passed; 0 failed` compte le nombre total de tests qui ont réussi ou échoué.

Comme nous n'avons aucun test que nous avons marqué comme ignoré, le résumé affiche `0 ignored`. Nous n'avons pas non plus filtré les tests qui ont été exécutés, donc la fin du résumé affiche `0 filtered out`. Nous verrons comment ignorer et filtrer les tests dans la prochaine section, “Contrôler comment les tests sont exécutés”.

La statistique `0 measured` sert pour des tests de benchmark qui mesurent les performances. Les tests de benchmark ne sont disponibles pour le moment que dans la version expérimentale de Rust (nightly), au moment de la rédaction. Rendez-vous sur la documentation sur les tests de benchmark pour en savoir plus.

La partie suivante du résultat des tests, qui commence par `Doc-tests addition`, concerne les résultats de tous les tests présents dans la documentation. Nous n'avons pas de tests dans la documentation pour le moment, mais Rust peut compiler tous les exemples de code qui sont présents dans la documentation de notre API. Cette fonctionnalité nous aide à garder synchronisés notre documentation et notre code ! Nous verrons comment écrire nos tests dans la documentation dans une section du chapitre 14. Pour le moment, nous allons ignorer la partie `Doc-tests` du résultat.

Changeons le nom de notre test pour voir comment cela change le résultat du test. Changeons le nom de la fonction `it_works` pour un nom différent, comme `exploration` ci-dessous :

Fichier : src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
}
```

Lancez ensuite à nouveau `cargo test` . Le résultat affiche désormais `exploration` plutôt que `it_works` :

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.59s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Ajoutons un autre test, mais cette fois nous allons construire un test qui échoue ! Les tests échouent lorsque quelque chose dans la fonction de test panique. Chaque test est lancé dans une nouvelle tâche, et lorsque la tâche principale voit qu'une tâche de test a été interrompue par panique, le test est considéré comme ayant échoué. Nous avons vu la façon la plus simple de faire paniquer au chapitre 9, qui consiste à appeler la macro `panic!` . Ecrivez ce nouveau test, `un_autre` , de sorte que votre fichier `src/lib.rs` ressemble à ceci :

Fichier : src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn un_autre() {
        panic!("Fait échouer ce test");
    }
}
```

Encart 11-3 : ajout d'un second test qui va échouer car nous appelons la macro `panic!`

Lancez à nouveau les tests en utilisant `cargo test`. Le résultat devrait ressembler à l'encart 11-4, qui va afficher que notre test `exploration` a réussi et que `un_autre` a échoué.

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.72s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::un_autre ... FAILED
test tests::exploration ... ok

failures:

---- tests::un_autre stdout ----
thread 'main' panicked at 'Fait échouer ce test', src/lib.rs:10:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::un_autre

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Encart 11-4 : les résultats de tests lorsque un test réussit et un autre test échoue

A la place du `ok`, la ligne `test tests:un_autre` affiche `FAILED`. Deux nouvelles sections apparaissent entre la liste des tests et le résumé : la première section affiche les raisons détaillées de chaque échec de test. Dans notre cas, `un_autre` a échoué car il a paniqué à 'Fait échouer ce test', qui est placé à la ligne 10 du fichier *src/lib.rs*. La partie suivante liste simplement les noms de tous les tests qui ont échoué, ce qui est utile lorsqu'il y a de

nombreux tests et beaucoup de détails provenant des tests qui échouent. Nous pouvons utiliser le nom d'un test qui échoue pour lancer uniquement ce test afin de déboguer plus facilement ; nous allons voir plus de façons de lancer des tests dans la section suivante.

La ligne de résumé s'affiche à la fin : au final, le résultat de nos tests est au statut `FAILED` (échoué). Nous avons un test réussi et un test échoué.

Maintenant que vous avez vu à quoi ressemblent les résultats de tests dans différents scénarios, voyons d'autres macros que `panic!` qui nous serons utiles pour les tests.

Vérifier les résultats avec la macro `assert!`

La macro `assert!`, fournie par la bibliothèque standard, est utile lorsque vous voulez vous assurer qu'une condition dans un test vaut `true`. Nous fournissons à la macro `assert!` un argument qui donne un Booléen une fois interprété. Si la valeur est `true`, `assert!` ne fait rien et le test est réussi. Si la valeur est `false`, la macro `assert!` appelle la macro `panic!`, qui fait échouer le test. L'utilisation de la macro `assert!` nous aide à vérifier que notre code fonctionne bien comme nous le souhaitions.

Dans le chapitre 5, dans l'encart 5-15, nous avons utilisé une structure `Rectangle` et une méthode `peut_contenir`, qui sont recopiés dans l'encart 11-5 ci-dessous. Ajoutons ce code dans le fichier *src/lib.rs* et écrivons quelques tests en utilisant la macro `assert!`.

Fichier : src/lib.rs

```
#[derive(Debug)]
struct Rectangle {
    largeur: u32,
    hauteur: u32,
}

impl Rectangle {
    fn peut_contenir(&self, other: &Rectangle) -> bool {
        self.largeur > other.largeur && self.hauteur > other.hauteur
    }
}
```

Encart 11-5 : utilisation de la structure `Rectangle` et sa méthode `peut_contenir` du chapitre 5

La méthode `peut_contenir` retourne un Booléen, ce qui veut dire que c'est un cas parfait pour tester la macro `assert!`. Dans l'encart 11-6, nous écrivons un test qui s'applique sur la méthode `peut_contenir` en créant une instance de `Rectangle` qui a une largeur de 8 et

une hauteur de 7, et qui vérifie qu'il peut contenir une autre instance de `Rectangle` qui a une largeur de 6 et une hauteur de 1.

Fichier : src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn un_grand_peut_contenir_un_petit() {
        let le_grand = Rectangle { largeur: 8, hauteur: 7 };
        let le_petit = Rectangle { largeur: 5, hauteur: 1 };

        assert!(le_grand.peut_contenir(&le_petit));
    }
}
```

Encart 11-6 : un test pour `peut_contenir` qui vérifie le cas où un grand rectangle peut contenir un plus petit rectangle

Remarquez que nous avons ajouté une nouvelle ligne à l'intérieur du module `test` : `use super::*;` . Le module `tests` est un module classique qui suit les règles de visibilité que nous avons vues au chapitre 7 dans la section "Les chemins pour désigner un élément dans l'arborescence de module". Comme le module `tests` est un module interne, nous avons besoin de ramener le code à tester qui se trouve dans son module parent dans la portée interne du module. Nous utilisons ici un opérateur global afin que tout ce que nous avons défini dans le module parent soit disponible dans le module `tests` .

Nous avons nommé notre test `un_grand_peut_contenir_un_petit` , et nous avons créé les deux instances `Rectangle` que nous avions besoin. Ensuite, nous avons appelé la macro `assert!` et nous lui avons passé le résultat de l'appel à `le_grand.peut_contenir(&le_petit)` . Cette expression est censée retourner `true` , donc notre test devrait réussir. Vérifions cela !

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests (target/debug/deps/rectangle-6584c4561e48942e)

running 1 test
test tests::un_grand_peut_contenir_un_petit ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Il a réussi ! Ajoutons maintenant un autre test, qui vérifie cette fois qu'un petit rectangle ne peut contenir un rectangle plus grand :

Fichier : src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn un_grand_peut_contenir_un_petit() {
        // --snip--
    }

    #[test]
    fn un_petit_ne_peut_pas_contenir_un_plus_grand() {
        let le_grand = Rectangle {
            largeur: 8,
            hauteur: 7,
        };
        let le_petit = Rectangle {
            largeur: 5,
            hauteur: 1,
        };

        assert!(!le_petit.peut_contenir(&le_grand));
    }
}
```

Comme le résultat correct de la fonction `peut_contenir` dans ce cas doit être `false`, nous devons faire un négatif de cette fonction avant de l'envoyer à la macro `assert!`. Cela aura

pour effet de faire réussir notre test si `peut_contenir` retourne `false` :

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::un_grand_peut_contenir_un_petit ... ok
test tests::un_petit_ne_peut_pas_contenir_un_plus_grand ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests rectangle

running 0 tests

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

Voilà deux tests qui réussissent ! Maintenant, voyons ce qu'il se passe dans les résultats de nos tests lorsque nous introduisons un bogue dans notre code. Changeons l'implémentation de la méthode `peut_contenir` en remplaçant l'opérateur *plus grand que* par un *plus petit que* au moment de la comparaison des largeurs :

```
// -- partie masquée ici --
impl Rectangle {
    fn peut_contenir(&self, other: &Rectangle) -> bool {
        self.largeur < other.largeur && self.hauteur > other.hauteur
    }
}
```

Le lancement des tests donne maintenant le résultat suivant :

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::un_grand_peut_contenir_un_petit ... FAILED
test tests::un_petit_ne_peut_pas_contenir_un_plus_grand ... ok

failures:

---- tests::un_grand_peut_contenir_un_petit stdout ----
thread 'main' panicked at 'assertion failed: le_grand.can_hold(&le_petit)', src/
lib.rs:28:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::un_grand_peut_contenir_un_petit

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Nos tests ont repéré le bogue ! Comme `le_grand.largeur` est 8 et `le_petit.largeur` est 5, la comparaison des largeurs dans `peut_contenir` retourne maintenant `false` : 8 n'est pas plus petit que 5.

Tester l'égalité avec les macros `assert_eq!` et `assert_ne!`

Une façon courante de tester des fonctionnalités est de comparer le résultat du code à tester par rapport à une valeur que vous souhaitez que le code retourne, afin de vous assurer qu'elles soient bien égales. Vous pouvez faire cela avec la macro `assert!` et en lui passant une expression qui utilise l'opérateur `==` . Cependant, c'est un test si courant que la bibliothèque standard fournit une paire de macros (`assert_eq!` et `assert_ne!`) pour procéder à ce test plus facilement. Les macros comparent respectivement l'égalité ou la non égalité de deux arguments. Elles vont aussi afficher les deux valeurs si la vérification échoue, ce qui va nous aider à comprendre *pourquoi* le test a échoué ; paradoxalement, la macro `assert!` indique seulement qu'elle a obtenu une valeur `false` de l'expression avec le `==` , mais n'affiche pas les valeurs qui l'ont mené à la valeur `false` .

Dans l'encart 11-7, nous écrivons une fonction `ajouter_deux` qui ajoute `2` à son paramètre et retourne le résultat. Ensuite, nous testons cette fonction en utilisant la macro `assert_eq!` .

Fichier : src/lib.rs

```
pub fn ajouter_deux(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn cela_ajoute_deux() {
        assert_eq!(4, ajouter_deux(2));
    }
}
```

Encart 11-7 : test de la fonction `ajouter_deux` en utilisant la macro `assert_eq!`.

Vérifions si cela fonctionne !

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::cela_ajoute_deux ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Le premier argument que nous avons donné à la macro `assert_eq!`, `4`, est bien égal au résultat de l'appel à `ajouter_deux`. La ligne correspondant à ce test est `test tests::cela_ajoute_deux ... ok`, et le texte `ok` indique que notre test a réussi !

Ajoutons un bogue dans notre code pour voir ce qu'il se passe lorsque un test qui utilise `assert_eq!` échoue. Changez l'implémentation de la fonction `ajouter_deux` pour ajouter plutôt `3` :

```
pub fn ajouter_deux(a: i32) -> i32 {
    a + 3
}
```

Lancez à nouveau les tests :

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::cela_ajoute_deux ... FAILED

failures:

---- tests::cela_ajoute_deux stdout ----
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `4`,
 right: `5`', src/lib.rs:11:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::cela_ajoute_deux

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Notre test a détecté le bogue ! Le test `cela_ajoute_deux` a échoué, ce qui a affiché le message `assertion failed: `(left == right)`` qui nous explique qu'à gauche nous avions `4` et qu'à droite nous avions `5`. Ce message utile nous aide au déboguage : cela veut dire que l'argument de gauche de `assert_eq!` valait `4` mais que l'argument de droite, où nous avions `ajouter_deux(2)`, valait `5`.

Notez que dans certains langages et environnements de test, les paramètres des fonctions qui vérifient que deux valeurs soient égales sont appelés `attendu` et `effectif`, et l'ordre dans lesquels nous renseignons les arguments est important. Cependant, dans Rust, on les appelle `gauche` et `droite`, et l'ordre dans lesquels nous renseignons la valeur que nous attendons et la valeur que produit le code à tester n'est pas important. Nous pouvons écrire la vérification de ce test dans la forme `assert_eq!(ajouter_deux(2), 4)`, ce qui donnera un message d'échec qui affichera `assertion failed: `(left == right)`` et que gauche vaudra `5` et droit vaudra `4`.

La macro `assert_ne!` va réussir si les deux valeurs que nous lui donnons ne sont pas égales

et va échouer si elles sont égales. Cette macro est utile dans les cas où nous ne sommes pas sûr de ce que *devrait* valoir une valeur, mais que nous savons ce que la valeur ne devrait surtout *pas* être si notre code fonctionne comme nous le souhaitons. Par exemple, si nous testons une fonction qui doit transformer sa valeur d'entrée de manière à ce qu'elle dépend du jour de la semaine où nous lançons nos tests, la meilleure façon de vérifier serait que la sortie de la fonction ne soit pas égale à son entrée.

Sous la surface, les macros `assert_eq!` et `assert_ne!` utilisent respectivement les opérateurs `==` et `!=`. Lorsque les vérifications échouent, ces macros affichent leurs arguments en utilisant le formatage de déboguage, ce qui veut dire que les valeurs comparées doivent implémenter les traits `PartialEq` et `Debug`. Tous les types primitifs et la plupart des types de la bibliothèque standard implémentent ces traits. Concernant les structures et les énumérations que vous définissez, vous allez avoir besoin de leur implémenter `Debug` pour afficher les valeurs lorsque les vérifications échouent. Comme ces traits sont des traits dérivables, comme nous l'avons évoqué dans l'encart 5-12 du chapitre 5, il suffit généralement de simplement ajouter l'annotation `#[derive(PartialEq, Debug)]` sur les définitions de vos structures ou énumérations. Rendez-vous à l'annexe C pour en savoir plus sur ces derniers et les autres traits dérivables.

Ajouter des messages d'échec personnalisés

Vous pouvez aussi ajouter un message personnalisé qui peut être affiché avec le message d'échec comme un argument optionnel aux macros `assert!`, `assert_eq!`, et `assert_ne!`. Tous les arguments renseignés après celui qui est obligatoire dans `assert!` ou les deux arguments obligatoires de `assert_eq!` et `assert_ne!` sont envoyés à la macro `format!` (que nous avons vue dans une section du chapitre 8), ainsi vous pouvez passer une chaine de caractères de formatage qui contient des espaces réservés `{}` et les valeurs iront dans ces espaces réservés. Les messages personnalisés sont utiles pour documenter ce que fait une vérification ; lorsqu'un test échoue, vous aurez une idée plus précise du problème avec ce code.

Par exemple, disons que nous avons une fonction qui accueille les gens par leur nom et que nous voulons tester que le nom que nous envoyons à la fonction apparaît dans le résultat :

Fichier : src/lib.rs

```
pub fn accueil(nom: &str) -> String {
    format!("Salut, {} !", nom)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn accueil_contient_le_nom() {
        let resultat = accueil("Carole");
        assert!(resultat.contains("Carole"));
    }
}
```

Les spécifications de ce programme n'ont pas été validées entièrement pour le moment, et on est quasiment sûr que le texte `Salut` au début va changer. Nous avons décidé que nous ne devrions pas à avoir à changer le test si les spécifications changent, donc plutôt que de vérifier l'égalité exacte de la valeur retournée par la fonction `accueil`, nous allons uniquement vérifier que le résultat contient le texte correspondant au paramètre d'entrée de la fonction.

Introduisons un bogue dans ce code en changeant `accueil` pour ne pas ajouter `nom` afin de voir ce que donne l'échec de ce test :

```
pub fn accueil(name: &str) -> String {
    String::from("Salut !")
}
```

L'exécution du test va donner ceci :

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished test [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::accueil_contient_le_nom ... FAILED

failures:

---- tests::accueil_contient_le_nom stdout ----
thread 'main' panicked at 'assertion failed: resultat.contains(\"Carole\")',
src/lib.rs:12:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::accueil_contient_le_nom

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Ce résultat indique simplement que la vérification a échoué, et à quel endroit. Le message d'échec serait plus utile dans notre cas s'il affichait la valeur que nous obtenons de la fonction `accueil`. Changeons la fonction de test, pour lui donner un message d'erreur personnalisé, qui est une chaîne de caractères de formatage avec un espace réservé qui contiendra la valeur que nous avons obtenue de la fonction `accueil` :

```
    #[test]
    fn accueil_contient_le_nom() {
        let resultat = accueil("Carole");
        assert!(
            resultat.contains("Carole"),
            "Le message d'accueil ne contient pas le nom, il vaut `{}`",
            resultat
        );
    }
```

Maintenant, lorsque nous lançons à nouveau le test, nous obtenons un message d'échec plus explicite :

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished test [unoptimized + debuginfo] target(s) in 0.93s
     Running unittests (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::accueil_contient_le_nom ... FAILED

failures:

---- tests::accueil_contient_le_nom stdout ----
thread 'main' panicked at 'Le message d'accueil ne contient pas le nom, il
vaut `Salut !`', src/lib.rs:12:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::accueil_contient_le_nom

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Nous pouvons voir la valeur que nous avons obtenue lors de la lecture du résultat du test, ce qui va nous aider à déboguer ce qui s'est passé à la place de ce que nous voulions qu'il se passe.

Vérifier le fonctionnement des paniques avec `should_panic`

En plus de vérifier que notre code retourne bien les valeurs que nous souhaitons, il est aussi important de vérifier que notre code gère bien les cas d'erreurs comme nous le souhaitons. Par exemple, utilisons le type `Supposition` que nous avons créé au chapitre 9, dans l'encart 9-13. Les autres codes qui utilisent `Supposition` reposent sur la garantie que les instances de `Supposition` contiennent uniquement des valeurs entre 1 et 100. Nous pouvons écrire un test qui s'assure que la création d'une instance de `Supposition` avec une valeur en dehors de cette intervalle va faire paniquer le programme.

Nous allons vérifier cela en ajoutant un autre attribut, `should_panic`, à notre fonction de test. Cet attribut fait réussir le test si le code à l'intérieur de la fonction fait paniquer ; le test va échouer si le code à l'intérieur de la fonction ne panique pas.

L'encart 11-8 nous montre un test qui vérifie que les conditions d'erreur de `Supposition::new` fonctionne bien comme nous l'avons prévu.

Fichier : src/lib.rs

```
pub struct Supposition {
    valeur: i32,
}

impl Supposition {
    pub fn new(valeur: i32) -> Supposition {
        if valeur < 1 || valeur > 100 {
            panic!("La supposition doit se trouver entre 1 et 100, et nous avons
{}.", valeur);
        }

        Supposition { valeur }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic]
    fn plus_grand_que_100() {
        Supposition::new(200);
    }
}
```

Encart 11-8 : tester qu'une condition va faire un `panic`

Nous plaçons l'attribut `#[should_panic]` après l'attribut `#[test]` et avant la fonction de test sur laquelle il s'applique. Voyons le résultat lorsque ce test réussit :

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::plus_grand_que_100 - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests guessing_game

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Ca fonctionne ! Maintenant, ajoutons un bogue dans notre code en enlevant la condition dans laquelle la fonction `new` panique lorsque la valeur est plus grande que 100 :

```
// -- partie masquée ici --
impl Supposition {
    pub fn new(valeur: i32) -> Supposition {
        if valeur < 1 {
            panic!("La supposition doit se trouver entre 1 et 100, et nous avons
{}.", valeur);
        }

        Supposition { valeur }
    }
}
```

Lorsque nous lançons le test de l'encart 11-8, il va échouer :

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::plus_grand_que_100 - should panic ... FAILED

failures:

---- tests::plus_grand_que_100 stdout ----
note: test did not panic as expected

failures:
    tests::plus_grand_que_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Dans ce cas, nous n'obtenons pas de message très utile, mais lorsque nous regardons la fonction de test, nous constatons qu'elle est marquée avec `#[should_panic]`. L'échec que nous obtenons signifie que le code dans la fonction de test n'a pas fait paniquer.

Les tests qui utilisent `should_panic` ne sont parfois pas assez explicites car ils indiquent seulement que le code a paniqué. Un test `should_panic` peut réussir, même si le test panique pour une raison différente de celle que nous attendions. Pour rendre les tests `should_panic` plus précis, nous pouvons ajouter un paramètre optionnel `expected` à l'attribut `should_panic`. Le système de test va s'assurer que le message d'échec contient bien le texte renseigné. Par exemple, imaginons le code modifié de `Supposition` dans l'encart 11-9 où la fonction `new` panique avec des messages différents si la valeur est trop petite ou trop grande.

Fichier : src/lib.rs

```
// -- partie masquée ici --
impl Supposition {
    pub fn new(valeur: i32) -> Supposition {
        if valeur < 1 {
            panic!(
                "La supposition doit être plus grande ou égale à 1, et nous
avons {}.",
                valeur
            );
        } else if valeur > 100 {
            panic!(
                "La supposition doit être plus petite ou égale à 100, et nous
avons {}.",
                valeur
            );
        }

        Supposition { valeur }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic(expected = "La supposition doit être plus petite ou égale à
100")]
    fn plus_grand_que_100() {
        Supposition::new(200);
    }
}
```

Encart 11-9 : on vérifie qu'une situation va provoquer un `panic!` avec un message de panique bien précis

Ce test va réussir car la valeur que nous insérons dans l'attribut `expected` de `should_panic` est une partie du message de panique de la fonction `Supposition::new`. Nous aurions pu renseigner le message de panique en entier que nous attendions, qui dans ce cas est `La supposition doit être plus petite ou égale à 100, et nous avons 200.`. Ce que vous choisissez de renseigner dans le paramètre `expected` de `should_panic` dépend de la mesure dans laquelle le message de panique est unique ou dynamique et de la précision de votre test que vous souhaitez appliquer. Dans ce cas, un extrait du message de panique est suffisant pour s'assurer que le code de la fonction de test s'exécute dans le cas du `else if valeur > 100`.

Pour voir ce qui se passe lorsqu'un test `should_panic` qui a un message `expected` qui échoue, essayons à nouveau d'introduire un bogue dans notre code en permutant les corps

des blocs de `if valeur < 1` et de `else if valeur > 100` :

```
        if valeur < 1 {
            panic!(
                "La supposition doit être plus petite ou égale à 100, et nous
avons {}.",
                valeur
            );
        } else if valeur > 100 {
            panic!(
                "La supposition doit être plus grande ou égale à 1, et nous
avons {}.",
                valeur
            );
        }
```

Cette fois, lorsque nous lançons le test avec `should_panic`, il devrait échouer :

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::plus_grand_que_100 - should panic ... FAILED

failures:

---- tests::plus_grand_que_100 stdout ----
thread 'main' panicked at 'La supposition doit être plus grande ou égale à 1, et
nous avons 200.', src/lib.rs:13:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
note: panic did not contain expected string
      panic message: `"La supposition doit être plus grande ou égale à 1, et
nous avons 200."`,
 expected substring: `"La supposition doit être plus petite ou égale à 100"`

failures:
    tests::plus_grand_que_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Le message d'échec nous informe que ce test a paniqué comme prévu, mais que le message de panique n'inclus pas la chaîne de caractères prévue `'La supposition doit être plus petite ou égale à 100'`. Le message de panique que nous avons obtenu dans ce cas était `La supposition doit être plus grande ou égale à 1, et nous avons 200.`. Maintenant,

on comprend mieux où est le bogue !

Utiliser `Result<T, E>` dans les tests

Précédemment, nous avons écrit des tests qui paniquent lorsqu'ils échouent. Nous pouvons également écrire des tests qui utilisent `Result<T, E>` ! Voici le test de l'encart 11-1, réécrit pour utiliser `Result<T, E>` et retourner une `Err` au lieu de paniquer :

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("deux plus deux ne vaut pas quatre"))
        }
    }
}
```

La fonction `it_works` a maintenant un type de retour, `Result<(), String>`. Dans le corps de la fonction, plutôt que d'appeler la macro `assert_eq!`, nous retournons `Ok(())` lorsque le test réussit et une `Err` avec une `String` à l'intérieur lorsque le test échoue.

Ecrire vos tests afin qu'ils retournent un `Result<T, E>` vous permet d'utiliser l'opérateur *point d'interrogation* dans le corps des tests, ce qui est un outil facile à utiliser pour écrire des tests qui peuvent échouer si n'importe quelle opération en son sein retourne une variante de `Err`.

Vous ne pouvez pas utiliser l'annotation `#[should_panic]` sur les tests qui utilisent `Result<T, E>`. Pour vérifier qu'une opération retourne une variante `Err`, *n'utilisez pas* l'opérateur "point d'interrogation" sur la valeur de type `Result<T, E>`. A la place, utilisez plutôt `assert!(valeur.is_err())`.

Maintenant que vous avez appris différentes manières d'écrire des tests, voyons ce qui se passe lorsque nous lançons nos tests et explorons les différentes options que nous pouvons utiliser avec `cargo test`.

Gérer l'exécution des tests

Comme `cargo run` qui compile votre code et qui exécute ensuite le binaire qui en résulte, `cargo test` compile votre code en mode test et lance le binaire de tests qu'il produit. Vous pouvez rajouter des options en ligne de commande pour changer le comportement par défaut de `cargo test`. Par exemple, le comportement par défaut des binaires produits par `cargo test` est de lancer tous les tests en parallèle et de capturer la sortie pendant l'exécution des tests, ce qui lui évite d'être affichée sur l'écran pendant ce temps, facilitant la lecture des messages relatifs aux résultats de l'exécution des tests.

Certaines options de la ligne de commande s'appliquent à `cargo test`, et certaines au binaire de tests qui en résulte. Pour séparer ces types d'arguments, il faut lister les arguments qui s'appliquent à `cargo test`, suivis du séparateur `--`, puis ajouter ceux qui s'appliquent au binaire de tests. L'exécution de `cargo test --help` affiche les options que vous pouvez utiliser sur `cargo test`, et l'exécution de `cargo test -- --help` affiche les options que vous pouvez utiliser après le séparateur `--`.

Lancer les tests en parallèle ou en séquence

Lorsque vous lancez de nombreux tests, par défaut ils s'exécutent en parallèle dans des tâches. Cela veut dire que tous les tests vont finir de s'exécuter plus rapidement afin que vous sachiez si votre code fonctionne ou non. Comme les tests s'exécutent en même temps, il faut s'assurer qu'ils ne dépendent pas les uns des autres ou d'un état partagé, y compris un environnement partagé, comme le dossier de travail actuel ou des variables d'environnement.

Par exemple, disons que chacun de vos tests exécute du code qui crée un fichier *test-sortie.txt* sur le disque dur et qu'il écrit quelques données dans ce fichier. Ensuite, chaque test lit les données de ce fichier et vérifie que le fichier contient une valeur précise, qui est différente dans chaque test. Comme les tests sont lancés en même temps, un test risque d'écraser le contenu du fichier entre le moment où un autre test lit et écrit sur ce fichier. Le second test va ensuite échouer, non pas parce que le code est incorrect mais parce que les tests se sont perturbés mutuellement pendant qu'ils s'exécutaient en parallèle. Une solution serait de s'assurer que chaque test écrit dans un fichier différent ; une autre serait de lancer les tests les uns après les autres.

Si vous ne souhaitez pas exécuter les tests en parallèle ou si vous voulez un contrôle plus précis du nombre de tâches utilisées, vous pouvez utiliser l'option `--test-threads` suivie du nombre de tâches que vous souhaitez que le binaire de test exécute en parallèle. Regardez cet exemple :

```
$ cargo test -- --test-threads=1
```

Nous avons réglé le nombre de tâches à `1`, ce qui indique au programme de ne pas utiliser le parallélisme. Exécuter ces tests en n'effectuant qu'une seule tâche à la fois va prendre plus de temps que de les lancer en parallèle, mais cela assure que les tests ne vont pas s'influencer mutuellement s'ils partagent le même état.

Afficher la sortie de la fonction

Par défaut, si un test réussit, la bibliothèque de test de Rust récupère tout ce qui est affiché sur la sortie standard. Par exemple, si nous appelons `println!` dans un test et que le test réussit, nous ne verrons pas la sortie correspondant au `println!` dans le terminal ; on verra seulement la ligne qui indique que le test a réussi. Si un test échoue, nous verrons ce qui a été affiché sur la sortie standard avec le reste des messages d'erreur.

Par exemple, l'encart 11-10 a une fonction stupide qui affiche la valeur de ses paramètres et retourne 10, ainsi qu'un test qui réussit et un test qui échoue.

Fichier : src/lib.rs

```
fn affiche_et_retourne_10(a: i32) -> i32 {
    println!("J'ai obtenu la valeur {}", a);
    10
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn ce_test_reussit() {
        let valeur = affiche_et_retourne_10(4);
        assert_eq!(10, valeur);
    }

    #[test]
    fn ce_test_echoue() {
        let valeur = affiche_et_retourne_10(8);
        assert_eq!(5, valeur);
    }
}
```

Encart 11-10 : tests d'une fonction qui fait appel à `println!`

Lorsque nous lançons ces tests avec `cargo test`, nous voyons cette sortie :

```
$ cargo test
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests (target/debug/deps/silly_function-160869f38cff9166)

running 2 tests
test tests::ce_test_echoue ... FAILED
test tests::ce_test_reussit ... ok

failures:

---- tests::ce_test_echoue stdout ----
J'ai obtenu la valeur 8
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `5`,
 right: `10`', src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::ce_test_echoue

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Remarquez que nous n'avons jamais vu `J'ai obtenu la valeur 4` dans cette sortie, qui est ce qui est affiché lors de l'exécution du test qui réussit. Cette sortie a été capturée. La sortie pour le test qui a échoué, `J'ai obtenu la valeur 8`, s'affiche dans la section de la sortie correspondante au résumé des tests, qui affiche aussi les causes de l'échec du test.

Si nous voulons aussi voir les valeurs affichées pour les tests réussis, nous pouvons demander à Rust d'afficher également la sortie des tests fructueux en lui rajoutant à la fin `--show-output`.

```
$ cargo test -- --show-output
```

Lorsque nous lançons à nouveau les tests de l'encart 11-10 avec l'option `--show-output`, nous voyons la sortie suivante :

```
$ cargo test -- --show-output
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished test [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests (target/debug/deps/silly_function-160869f38cff9166)

running 2 tests
test tests::ce_test_echoue ... FAILED
test tests::ce_test_reussit ... ok

successes:

---- tests::ce_test_reussit stdout ----
J'ai obtenu la valeur 4


successes:
    tests::ce_test_reussit

failures:

---- tests::ce_test_echoue stdout ----
J'ai obtenu la valeur 8
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `5`,
 right: `10`', src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::ce_test_echoue

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Exécuter un sous-ensemble de tests en fonction de son nom

Parfois, lancer une suite de tests entière peut prendre beaucoup de temps. Si vous travaillez sur du code d'un périmètre bien défini, vous pourriez avoir besoin d'exécuter uniquement les tests relatifs à ce code. Vous pouvez choisir quels tests exécuter en envoyant le ou les noms du ou des tests que vous souhaitez exécuter en argument de `cargo test`.

Dans le but de démontrer comment lancer un sous-ensemble de tests, nous allons créer trois tests pour notre fonction `ajouter_deux` dans l'encart 11-11, et choisir lesquels nous allons exécuter.

Fichier : src/lib.rs

```
pub fn ajouter_deux(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn ajouter_deux_a_deux() {
        assert_eq!(4, ajouter_deux(2));
    }

    #[test]
    fn ajouter_deux_a_trois() {
        assert_eq!(5, ajouter_deux(3));
    }

    #[test]
    fn cent() {
        assert_eq!(102, ajouter_deux(100));
    }
}
```

Encart 11-11 : trois tests avec trois noms différents

Si nous exécutons les tests sans ajouter d'arguments, comme nous l'avons vu précédemment, tous les tests vont s'exécuter en parallèle :

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 3 tests
test tests::ajouter_deux_a_trois ... ok
test tests::ajouter_deux_a_deux ... ok
test tests::cent ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Exécuter des tests individuellement

Nous pouvons donner le nom de n'importe quelle fonction de test à `cargo test` afin d'exécuter uniquement ce test :

```
$ cargo test cent
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.69s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::cent ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out;
finished in 0.00s
```

Le test avec le nom `cent` est le seul exécuté ; les deux autres tests ne correspondent pas à ce nom. La sortie du test nous indique que nous avons d'autres tests en plus de celui que cette commande a exécuté en affichant `2 filtered out` à la fin de la ligne de résumé.

Nous ne pouvons pas renseigner plusieurs noms de tests de cette manière ; il n'y a que la première valeur fournie à `cargo test` qui sera utilisée. Mais il existe un moyen d'exécuter plusieurs tests.

Filtrer pour exécuter plusieurs tests

Nous pouvons ne renseigner qu'une partie d'un nom de test, et tous les tests dont les noms correspondent à cette valeur vont être exécutés. Par exemple, comme deux de nos noms de tests contiennent `ajouter`, nous pouvons exécuter ces deux en lançant `cargo test ajouter` :

```
$ cargo test ajouter
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::ajouter_deux_a_trois ... ok
test tests::ajouter_deux_a_deux ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
finished in 0.00s
```

Cette commande a lancé tous les tests qui contiennent `ajouter` dans leur nom et a filtré le test `cent`. Notez aussi que le module dans lequel un test est présent fait partie du nom du

test, ainsi nous pouvons exécuter tous les tests d'un module en filtrant avec le nom du module.

Ignorer certains tests sauf s'ils sont demandés explicitement

Parfois, certains tests spécifiques peuvent prendre beaucoup de temps à s'exécuter, de sorte que vous voulez les exclure de la majorité des exécutions de `cargo test`. Plutôt que de lister en argument tous les tests que vous souhaitez exécuter, vous pouvez plutôt faire une annotation sur les tests qui prennent du temps en utilisant l'attribut `ignore` pour les exclure, comme ci-dessous :

Fichier : src/lib.rs

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn test_long() {
    // du code qui prend une heure à s'exécuter
}
```

Après `#[test]`, nous avons ajouté la ligne `#[ignore]` pour le test que nous souhaitons exclure. Maintenant lorsque nous exécutons nos tests, `it_works` s'exécute, mais pas `test_long` :

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test test_long ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

La fonction `test_long` est listée comme `ignored`. Si nous voulons exécuter uniquement les tests ignorés, nous pouvons utiliser `cargo test -- --ignored` :

```
$ cargo test -- --ignored
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test test_long ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out;
finished in 0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

En gérant quels tests sont exécutés, vous pouvez vous assurer que vos résultats de `cargo test` seront rapides. Lorsque vous arrivez à un stade où il est justifié de vérifier le résultat des tests `ignored` et que vous avez le temps d'attendre ces résultats, vous pouvez lancer à la place `cargo test -- --ignored`. Si vous voulez exécuter tous les tests q'uils soient ignorés ou non, vous pouvez lancer `cargo test -- --include-ignored`.

L'organisation des tests

Comme nous l'avons évoqué au début du chapitre, le test est une discipline complexe, et différentes personnes utilisent des terminologies et organisations différentes. La communauté Rust a conçu les tests dans deux catégories principales : *les tests unitaires* et *les tests d'intégration*. Les tests unitaires sont petits et plus précis, testent un module isolé à la fois, et peuvent tester les interfaces privées. Les tests d'intégration sont uniquement externes à notre bibliothèque et consomment notre code exactement de la même manière que tout autre code externe le ferait, en utilisant uniquement l'interface publique et éventuellement en utilisant plusieurs modules dans un test.

L'écriture de ces deux types de tests est importante pour s'assurer que chaque élément de notre bibliothèque fait bien ce que vous attendiez d'eux, de manière isolée et conjuguée avec d'autres.

Les tests unitaires

Le but des tests unitaires est de tester chaque élément du code de manière séparée du reste du code pour identifier rapidement où le code fonctionne ou non comme prévu. Vous devriez insérer les tests unitaires dans le dossier *src* dans chaque fichier, à côté du code qu'ils testent. La convention est de créer un module `tests` dans chaque fichier qui contient les fonctions de test et de marquer le module avec `cfg(test)` .

Les modules de tests et `#[cfg(test)]`

L'annotation `#[cfg(test)]` sur les modules de tests indique à Rust de compiler et d'exécuter le code de test seulement lorsque vous lancez `cargo test` , et non pas lorsque vous lancez `cargo build` . Cela diminue la durée de compilation lorsque vous souhaitez uniquement compiler la bibliothèque et cela réduit la taille dans l'artefact compilé qui en résulte car les tests n'y sont pas intégrés. Vous verrez plus tard que comme les tests d'intégration se placent dans un répertoire différent, ils n'ont pas besoin de l'annotation `#[cfg(test)]` . Cependant, comme les tests unitaires vont dans les mêmes fichiers que le code, vous devriez utiliser `#[cfg(test)]` pour marquer qu'ils ne devraient pas être inclus dans les résultats de compilation.

Souvenez-vous, lorsque nous avons généré le nouveau projet `addition` dans la première section de ce chapitre, Cargo a généré ce code pour nous :

Fichier : src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let resultat = 2 + 2;
        assert_eq!(resultat, 4);
    }
}
```

Ce code est le module de test généré automatiquement. L'attribut `cfg` est l'abréviation de *configuration* et indique à Rust que l'élément suivant ne doit être intégré que lorsqu'une certaine option de configuration est donnée. Dans ce cas, l'option de configuration est `test`, qui est fournie par Rust pour la compilation et l'exécution des tests. En utilisant l'attribut `cfg`, Cargo compile notre code de tests uniquement si nous avons exécuté les tests avec `cargo test`. Cela inclut toutes les fonctions auxiliaires qui pourraient se trouver dans ce module, en plus des fonctions marquées d'un `#[test]`.

Tester des fonctions privées

Il existe un débat dans la communauté des testeurs au sujet de la nécessité ou non de tester directement les fonctions privées, et d'autres langages rendent difficile, voir impossible, de tester les fonctions privées. Quelle que soit votre approche des tests, les règles de protection de Rust vous permettent de tester des fonctions privées. Imaginons le code de l'encart 11-12 qui contient la fonction privée `addition_interne`.

Fichier : src/lib.rs

```
pub fn ajouter_deux(a: i32) -> i32 {
    addition_interne(a, 2)
}

fn addition_interne(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn interne() {
        assert_eq!(4, addition_interne(2, 2));
    }
}
```

Encart 11-12 : test d'une fonction privée

Remarquez que la fonction `addition_interne` n'est pas marquée comme `pub`. Les tests sont uniquement du code Rust, et le module `test` est simplement un autre module. Comme nous l'avons vu dans la section "Désigner un élément dans l'arborescence de modules", les éléments dans les modules enfants peuvent utiliser les éléments dans leurs modules parents. Dans ce test, nous importons dans la portée tous les éléments du parent du module `test` grâce à `use super::*;`, permettant ensuite au test de faire appel à `addition_interne`. Si vous pensez qu'une fonction privée ne doit pas être testée, il n'y a rien qui vous y force avec Rust.

Les tests d'intégration

En Rust, les tests d'intégration sont exclusivement externes à votre bibliothèque. Ils consomment votre bibliothèque de la même manière que n'importe quel autre code, ce qui signifie qu'ils ne peuvent appeler que les fonctions qui font partie de l'interface de programmation applicative (API) publique de votre bibliothèque. Leur but est de tester si les multiples parties de votre bibliothèque fonctionnent correctement ensemble. Les portions de code qui fonctionnent bien toutes seules pourraient rencontrer des problèmes une fois imbriquées avec d'autres, donc les tests qui couvrent l'intégration du code sont tout aussi importants. Pour créer des tests d'intégration, vous avez d'abord besoin d'un dossier *tests*.

Le dossier *tests*

Nous créons un dossier *tests* au niveau le plus haut de notre dossier projet, juste à côté de *src*. Cargo sait qu'il doit rechercher les fichiers de test d'intégration dans ce dossier. Nous pouvons ensuite construire autant de fichiers de test que nous le souhaitons dans ce dossier, et Cargo va compiler chacun de ces fichiers comme une crate individuelle.

Commençons à créer un test d'intégration. Avec le code de l'encart 11-12 toujours présent dans le fichier *src/lib.rs*, créez un dossier *tests*, puis un nouveau fichier *tests/test_integration.rs* et insérez-y le code de l'encart 11-13.

Fichier : tests/test_integration.rs

```
use addition;

#[test]
fn cela_ajoute_deux() {
    assert_eq!(4, addition::ajouter_deux(2));
}
```

Encart 11-13 : un test d'intégration d'une fonction présente dans la crate `addition`

Nous avons ajouté `use addition` en haut du code, ce que nous n'avions pas besoin de faire dans les tests unitaires. La raison à cela est que chaque fichier dans le dossier `tests` est une crate séparée, donc nous devons importer notre bibliothèque dans la portée de chaque crate de test.

Nous n'avons pas besoin de marquer du code avec `#[cfg(test)]` dans *tests/test_integration.rs*. Cargo traite le dossier `tests` de manière particulière et compile les fichiers présents dans ce dossier uniquement si nous lançons `cargo test`. Lancez dès maintenant `cargo test` :

```
$ cargo test
   Compiling addition v0.1.0 (file:///projects/addition)
    Finished test [unoptimized + debuginfo] target(s) in 1.31s
     Running unittests (target/debug/deps/addition-1082c4b063a8fbe6)

running 1 test
test tests::interne ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running tests/integration_test.rs (target/debug/deps/
integration_test-1082c4b063a8fbe6)

running 1 test
test cela_ajoute_deux ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests addition

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Les trois sections de la sortie concernent les tests unitaires, les tests d'intégration et les tests de documentation. La première section relative aux tests unitaires est la même que celle que nous avons déjà vue : une ligne pour chaque test unitaire (celui qui s'appelle `interne` que nous avons inséré dans l'encart 11-12) suivie d'une ligne de résumé des tests unitaires.

La section des tests d'intégration commence avec la ligne `Running target/debug/deps/test_integration-1082c4b063a8fbe6` (le hachage à la fin de votre sortie pourrait être différent). Ensuite, il y a une ligne pour chaque fonction de test présente dans ce test d'intégration et une ligne de résumé pour les résultats des tests d'intégration, juste avant

que la section `Doc-tests addition` ne commence.

De la même façon que plus vous ajoutiez de fonctions de tests unitaires et plus vous aviez de lignes de résultats dans la section des tests unitaires, plus vous ajoutez des fonctions de tests aux fichiers de tests d'intégration et plus vous obtenez de lignes de résultat dans la section correspondant aux fichiers des tests d'intégration. Chaque fichier de test d'intégration a sa propre section, donc si nous ajoutons plus de fichiers dans le dossier *tests*, il y aura plus de sections de tests d'intégration.

Nous pouvons aussi exécuter une fonction de test d'intégration précise en utilisant le nom de la fonction de test comme argument à `cargo test`. Pour exécuter tous les tests d'un fichier de tests d'intégration précis, utilisez l'argument `--test` de `cargo test` suivi du nom du fichier :

```
$ cargo test --test integration_test
   Compiling addition v0.1.0 (file:///projects/addition)
    Finished test [unoptimized + debuginfo] target(s) in 0.64s
     Running tests/integration_test.rs (target/debug/deps/
integration_test-82e7799c1bc62298)

running 1 test
test cela_ajoute_deux ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Cette commande exécute seulement les tests dans le fichier *tests/test_integration.rs*.

Les sous-modules des tests d'intégration

Au fur et à mesure que vous ajouterez des tests d'intégration, vous pourriez avoir besoin de les diviser en plusieurs fichiers dans le dossier *tests* pour vous aider à les organiser ; par exemple, vous pouvez regrouper les fonctions de test par fonctionnalités qu'elles testent. Comme mentionné précédemment, chaque fichier dans le dossier *tests* est compilé comme étant sa propre crate séparée de tous les autres.

Le fait que chaque fichier de test d'intégration soit sa propre crate est utile pour créer des portées séparées qui ressemblent à la manière dont les développeurs vont consommer votre crate. Cependant, cela veut aussi dire que les fichiers dans le dossier *tests* ne partagent pas le même comportement que les les fichiers dans *src*, comme vous l'avez appris au chapitre 7 à propos de la manière de séparer le code dans des modules et des fichiers.

Ce comportement différent des fichiers dans le dossier *tests* est encore plus notable lorsque

vous avez un jeu de fonctions d'aide qui s'avèrent utiles pour plusieurs fichiers de test d'intégration et que vous essayez de suivre les étapes de la section “Séparer les modules dans différents fichiers” du chapitre 7 afin de les extraire dans un module en commun. Par exemple, si nous créons *tests/commun.rs* et que nous y plaçons une fonction `parametrage` à l'intérieur, nous pourrions ajouter du code à `parametrage` que nous voudrions appeler à partir de différentes fonctions de test dans différents fichiers de test :

Fichier : tests/commun.rs

```
pub fn parametrage() {
    // code de paramétrage spécifique à vos tests de votre bibliothèque ici
}
```

Lorsque nous lançons les tests à nouveau, nous allons voir une nouvelle section dans la sortie des tests, correspondant au fichier *commun.rs*, même si ce fichier ne contient aucune fonction de test et que nous n'avons utilisé nulle part la fonction `parametrage` :

```
$ cargo test
   Compiling addition v0.1.0 (file:///projects/addition)
    Finished test [unoptimized + debuginfo] target(s) in 0.89s
     Running unittests (target/debug/deps/addition-92948b65e88960b4)

running 1 test
test tests::interne ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running tests/common.rs (target/debug/deps/common-92948b65e88960b4)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running tests/integration_test.rs (target/debug/deps/
integration_test-92948b65e88960b4)

running 1 test
test cela_ajoute_deux ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests addition

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Nous ne voulons pas que `commun` apparaisse dans les résultats, ni que cela affiche `running 0 tests`. Nous voulons juste partager du code avec les autres fichiers de test d'intégration.

Pour éviter que `commun` s'affiche sur la sortie de test, au lieu de créer le fichier *tests/commun.rs*, nous allons créer *tests/commun/mod.rs*. C'est une convention de nommage alternative que Rust comprend aussi. Nommer le fichier ainsi indique à Rust de ne pas traiter le module `commun` comme un fichier de test d'intégration. Lorsque nous déplaçons le code de la fonction `parametrage` dans *tests/commun/mod.rs* et que nous supprimons le fichier *tests/commun.rs*, la section dans la sortie des tests ne va plus s'afficher. Les fichiers dans les sous-répertoires du dossier *tests* ne seront pas compilés comme étant une crate séparée et n'auront pas de sections dans la sortie des tests.

Après avoir créé *tests/commun/mod.rs*, nous pouvons l'utiliser à partir de n'importe quel fichier de test d'intégration comme un module. Voici un exemple d'appel à la fonction

`parametrage` à partir du test `cela_ajoute_deux` dans *tests/test_integration.rs* :

Fichier : tests/integration_test.rs

```
use addition;

mod common;

#[test]
fn cela_ajoute_deux() {
    common::parametrage();
    assert_eq!(4, addition::ajouter_deux(2));
}
```

Remarquez que la déclaration `mod commun;` est la même que la déclaration d'un module que nous avons montrée dans l'encart 7-21. Ensuite, dans la fonction de tests, nous pouvons appeler la fonction `commun::parametrage`.

Tests d'intégration pour les crates binaires

Si notre projet est une crate binaire qui contient uniquement un fichier *src/main.rs* et n'a pas de fichier *src/lib.rs*, nous ne pouvons pas créer de tests d'intégration dans le dossier *tests* et importer les fonctions définies dans le fichier *src/main.rs* dans notre portée avec une instruction `use`. Seules les crates de bibliothèque exposent des fonctions que les autres crates peuvent utiliser ; les crates binaires sont conçues pour être exécutées de manière isolée.

C'est une des raisons pour lesquelles les projets Rust qui fournissent un binaire ont un simple fichier *src/main.rs* qui fait appel à la logique présente dans le fichier *src/lib.rs*. En utilisant cette structure, les tests d'intégration *peuvent* tester la crate de bibliothèque avec le `use` pour importer les importantes fonctionnalités disponibles. Si les fonctionnalités importantes fonctionnent, la petite portion de code dans le fichier *src/main.rs* va fonctionner, et cette petite partie de code n'a pas besoin d'être testée.

Résumé

Les fonctionnalités de test de Rust permettent de spécifier comment le code doit fonctionner pour garantir qu'il va continuer à fonctionner comme vous le souhaitez, même si vous faites des changements. Les tests unitaires permettent de tester séparément différentes parties d'une bibliothèque et peuvent tester l'implémentation des éléments privés. Les tests d'intégration vérifient que de nombreuses parties de la bibliothèque

fonctionnent correctement ensemble, et ils utilisent l'API publique de la bibliothèque pour tester le code, de la même manière que le ferait du code externe qui l'utiliserait. Même si le système de type de Rust et les règles de possession aident à empêcher certains types de bogues, les tests restent toujours importants pour réduire les bogues de logique concernant le comportement attendu de votre code.

Et maintenant, combinons le savoir que vous avez accumulé dans ce chapitre et dans les chapitres précédents en travaillant sur un nouveau projet !

Un projet d'entrée/sortie : construire un programme en ligne de commande

Ce chapitre est un résumé de toutes les nombreuses compétences que vous avez apprises précédemment et une découverte de quelques fonctionnalités supplémentaires de la bibliothèque standard. Nous allons construire un outil en ligne de commande qui interagit avec des fichiers et les entrées/sorties de la ligne de commande pour mettre en pratique certains concepts Rust dont vous avez maintenant connaissance.

Sa rapiditié, ses fonctionnalités de sécurité, sa sortie binaire unifiée et sa prise en charge de multiples plateformes font de Rust le langage idéal pour créer des outils en ligne de commande, donc pour notre projet, nous allons construire notre version de l'outil en ligne de commande `grep` (qui signifie **g**lobally search a **r**egular **e**xpression and **p**rint, soit *recherche globale et affichage d'une expression régulière*). Dans des cas d'usage très simple, `grep` recherche une chaîne de caractères précise dans un fichier précis. Pour ce faire, `grep` prend en argument un nom de fichier et une chaîne de caractères. Ensuite, il lit le fichier, trouve les lignes de ce fichier qui contiennent la chaîne de caractères passée en argument, puis affiche ces lignes.

En chemin, nous allons vous montrer comment utiliser dans votre outil en ligne de commande les fonctionnalités des terminaux que de nombreux outils en ligne de commande utilisent. Nous allons lire la valeur d'une variable d'environnement pour permettre à l'utilisateur de configurer le comportement de notre outil. Nous allons aussi afficher des messages d'erreur vers le flux d'erreur standard de la console (`stderr`) plutôt que vers la sortie standard (`stdout`), pour, par exemple, que l'utilisateur puisse rediriger la sortie fructueuse vers un fichier, tout en affichant les messages d'erreur à l'écran.

Un membre de la communauté Rust, Andrew Gallant, a déjà créé une version complète et très performante de `grep` , qu'il a appelée `ripgrep` . En comparaison, notre version de `grep` sera plutôt simple, mais ce chapitre va vous donner les connaissances de base dont vous avez besoin pour appréhender un projet réel comme `ripgrep` .

Notre projet `grep` va combiner un certain nombre de concepts que vous avez déjà acquis à ce stade :

- Organiser le code (en utilisant ce que vous avez appris sur les modules au chapitre 7)
- Utiliser les vecteurs et les chaînes de caractères (les collections du chapitre 8)
- Gérer les erreurs (chapitre 9)
- Utiliser les traits et les durées de vie lorsque c'est approprié (chapitre 10)
- Ecrire les tests (chapitre 11)

Nous vous présenterons aussi brièvement les fermetures, les itérateurs et les objets de trait, que les chapitres 13 et 17 traiteront en détails.

Récupérer les arguments de la ligne de commande

Créons un nouveau projet comme à l'accoutumée avec `cargo new`. Appelons notre projet `minigrep` pour le distinguer de l'outil `grep` que vous avez probablement déjà sur votre système.

```
$ cargo new minigrep
     Created binary (application) `minigrep` project
$ cd minigrep
```

La première tâche est de faire en sorte que `minigrep` utilise ses deux arguments en ligne de commande : le nom du fichier et la chaîne de caractères à rechercher. Autrement dit, nous voulons pouvoir exécuter notre programme avec `cargo run`, une chaîne de caractères à rechercher, et un chemin vers un fichier dans lequel chercher, comme ceci :

```
$ cargo run chaine_a_chercher fichier-exemple.txt
```

Pour l'instant, le programme généré par `cargo new` ne peut pas traiter les arguments que nous lui donnons. Certaines bibliothèques qui existent sur crates.io peuvent vous aider à écrire un programme qui prend des arguments en ligne de commande, mais comme vous apprenez juste ce concept, implémentons cette capacité par nous-mêmes.

Lire les valeurs des arguments

Pour permettre à `minigrep` de lire les valeurs des arguments de la ligne de commande que nous lui envoyons, nous allons avoir besoin d'une fonction fournie par la bibliothèque standard de Rust, qui est `std::env::args`. Cette fonction retourne un itérateur des arguments de la ligne de commande qui ont été donnés à `minigrep`. Nous verrons les itérateurs plus précisément au chapitre 13. Pour l'instant, vous avez juste à savoir deux choses à propos des itérateurs : les itérateurs engendrent une série de valeurs, et nous pouvons appeler la méthode `collect` sur un itérateur pour le transformer en collection, comme les vecteurs, qui contiennent tous les éléments qu'un itérateur engendrent.

Utilisez le code de l'encart 12-1 pour permettre à votre programme `minigrep` de lire tous les arguments qui lui sont envoyés et ensuite collecter les valeurs dans un vecteur.

Fichier : src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{:?}", args);
}
```

Encart 12-1 : Collecter les arguments de la ligne de commande dans un vecteur et les afficher

D'abord, nous importons le module `std::env` dans la portée avec une instruction `use` afin que nous puissions utiliser sa fonction `args`. Notez que la fonction `std::env::args` est imbriquée sur deux niveaux de modules. Comme nous l'avons vu dans le chapitre 7, il est courant d'importer le module parent dans la portée plutôt que la fonction. En faisant ainsi, nous pouvons facilement utiliser les autres fonctions de `std::env`. C'est aussi moins ambigü que d'importer uniquement `std::env::args` et ensuite d'appeler la fonction avec seulement `args`, car `args` peu facilement être confondu avec une fonction qui est définie dans le module courant.

La fonction `args` et l'unicode invalide

Notez que `std::env::args` va paniquer si un des arguments contient de l'unicode invalide. Si votre programme a besoin d'utiliser des arguments qui contiennent de l'unicode invalide, utilisez plutôt `std::env::args_os` à la place. Cette fonction retourne un itérateur qui engendre des valeurs `OsString` plutôt que des valeurs `String`. Nous avons choisi d'utiliser ici `std::env::args` par simplicité, car les valeurs `OsString` diffèrent selon la plateforme et c'est plus complexe de travailler avec par rapport aux valeurs de type `String`.

Dans la première ligne du `main`, nous appelons `env::args`, et nous utilisons immédiatement `collect` pour retourner un itérateur dans un vecteur qui contient toutes les valeurs engendrées par l'itérateur. Nous pouvons utiliser la fonction `collect` pour créer n'importe quel genre de collection, donc nous avons annoté explicitement le type de `args` pour préciser que nous attendions un vecteur de chaînes de caractères. Bien que nous n'ayons que très rarement d'annoter les types en Rust, `collect` est une fonction que vous aurez souvent besoin d'annoter car Rust n'est pas capable de déduire le type de collection que vous attendez.

Enfin, nous affichons le vecteur en utilisant la chaîne de formatage `:?`. Essayons d'abord de

lancer le code sans arguments, puis ensuite avec deux arguments :

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/minigrep`
["target/debug/minigrep"]


$ cargo run aiguille botte_de_foin
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 1.57s
     Running `target/debug/minigrep aiguille botte_de_foin`
["target/debug/minigrep", "aiguille", "botte_de_foin"]
```

Remarquez que la première valeur dans le vecteur est `"target/debug/minigrep"`, qui est le nom de notre binaire. Cela correspond au fonctionnement de la liste d'arguments en C, qui laissent les programmes utiliser le nom sous lequel ils ont été invoqués dans leur exécution. C'est parfois pratique pour avoir accès au nom du programme dans le cas où vous souhaitez l'afficher dans des messages, ou changer le comportement du programme en fonction de ce que l'alias de la ligne de commande utilise pour invoquer le programme. Mais pour les besoins de ce chapitre, nous allons l'ignorer et récupérer uniquement les deux arguments dont nous avons besoin.

Enregistrer les valeurs des arguments dans des variables

L'affichage des valeurs du vecteur des arguments nous a démontré que le programme peut avoir accès aux valeurs envoyées en arguments d'une ligne de commande. Maintenant, nous avons besoin d'enregistrer les valeurs des deux arguments dans des variables afin que nous puissions utiliser les valeurs pour le reste du programme. C'est que nous faisons dans l'encart 12-2.

Fichier : src/main.rs

```
use std::env;

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

    let recherche = &args[1];
    let nom_fichier = &args[2];

    println!("On recherche : {}", recherche);
    println!("Dans le fichier : {}", nom_fichier);
}
```

Encart 12-2 : Création de variables pour récupérer les arguments recherche et nom_fichier

Comme nous l'avons vu lorsque nous avons affiché le vecteur, le nom du programme prend la première valeur dans le vecteur, dans `args[0]`, donc nous allons commencer à l'indice `1`. Le premier argument que prend `minigrep` est la chaîne de caractères que nous recherchons, donc nous insérons la référence vers le premier argument dans la variable `recherche`. Le second argument sera le nom du fichier, donc nous insérons une référence vers le second argument dans la variable `nom_fichier`.

Nous affichons temporairement les valeurs de ces variables pour prouver que le code fonctionne bien comme nous le souhaitons. Lançons à nouveau ce programme avec les arguments `test` et `example.txt` :

```
$ cargo run test exemple.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep test exemple.txt`
On recherche : test
Dans le fichier : exemple.txt
```

Très bien, notre programme fonctionne ! Les valeurs des arguments dont nous avons besoin sont enregistrées dans les bonnes variables. Plus tard, nous allons ajouter de la gestion d'erreurs pour pallier aux potentielles situations d'erreurs, comme lorsque l'utilisateur ne fournit pas d'arguments ; pour le moment, nous allons ignorer ces situations et continuer à travailler pour l'ajout d'une capacité de lecture de fichier, à la place.

Lire un fichier

Maintenant, nous allons ajouter une fonctionnalité pour lire le fichier qui est renseigné dans l'argument `nom_fichier` de la ligne de commande. D'abord, nous avons besoin d'un fichier d'exemple pour le tester : le meilleur type de fichier pour s'assurer que `minigrep` fonctionne est un fichier avec une petite quantité de texte sur plusieurs lignes avec quelques mots répétés. L'encart 12-3 présente un poème en Anglais de Emily Dickinson qui fonctionnera bien pour ce test ! Créez un fichier *poem.txt* à la racine de votre projet, et saisissez ce poème “I'm Nobody! Who are you?”.

Filename: poem.txt

```
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

Encart 12-3 : Un poème Anglais d'Emily Dickinson qui fait un bon sujet d'essai

Une fois ce texte enregistré, éditez le *src/main.rs* et ajoutez-y le code pour lire le fichier, comme indiqué dans l'encart 12-4.

Fichier : src/main.rs

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

fn main() {
    // -- partie masquée ici --
    println!("Dans le fichier : {}", nom_fichier);

    let contenu = fs::read_to_string(nom_fichier)
        .expect("Quelque chose s'est mal passé lors de la lecture du fichier");

    println!("Dans le texte :\n{}", contenu);
}
```

Encart 12-4 : Lecture du contenu du fichier renseigné en second argument

Premièrement, nous ajoutons une autre instruction `use` pour importer une partie significative de la bibliothèque standard : nous avons besoin de `std::fs` pour manipuler les

fichiers.

Dans le `main`, nous avons ajouté une nouvelle instruction : `fs::read_to_string` qui prend le `nom_fichier`, ouvre ce fichier, et retourne un `Result<String>` du contenu du fichier.

Après cette instruction, nous avons ajouté à nouveau une instruction `println!` qui affiche la valeur de `contenu` après la lecture de ce fichier, afin que nous puissions vérifier que ce programme fonctionne correctement.

Exécutons ce code avec n'importe quelle chaîne de caractères dans le premier argument de la ligne de commande (car nous n'avons pas encore implémenté la partie de recherche pour l'instant), ainsi que le fichier *poem.txt* en second argument :

```
$ cargo run the poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep the poem.txt`
On recherche : the
Dans le fichier : poem.txt
Dans le texte :
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

Très bien ! Notre code lit et affiche ensuite le contenu du fichier. Mais le code a quelques défauts. La fonction `main` a plusieurs responsabilités : généralement, les rôles des fonctions sont plus clairs et faciles à entretenir si chaque fonction est en charge d'une seule tâche. L'autre problème est que nous ne gérons pas les erreurs correctement. Le programme est encore très modeste, donc ces imperfections ne sont pas un gros problème, mais dès que le programme va grossir, il sera plus difficile de les corriger proprement. Le remaniement du code très tôt lors du développement d'un logiciel est une bonne pratique, car c'est beaucoup plus facile de remanier des petites portions de code. C'est ce que nous allons faire dès maintenant.

Remanier le code pour améliorer sa modularité et la gestion des erreurs

Pour améliorer notre programme, nous allons résoudre quatre problèmes liés à la structure du programme et à la façon dont il gère de potentielles erreurs.

Premièrement, notre fonction `main` assure deux tâches : elle interprète les arguments et elle lit des fichiers. Pour une fonction aussi petite, ce n'est pas un problème majeur. Cependant, si nous continuons à faire grossir notre programme dans le `main`, le nombre des différentes tâches qu'assure la fonction `main` va continuer à s'agrandir. Plus une fonction assure des tâches différentes, plus cela devient difficile de la comprendre, de la tester, et d'y faire des changements sans casser ses autres constituants. Cela est mieux de séparer les fonctionnalités afin que chaque fonction n'assure qu'une seule tâche.

Cette problématique est aussi liée au deuxième problème : bien que `recherche` et `nom_fichier` soient des variables de configuration de notre programme, les variables telles que `contenu` sont utilisées pour appuyer la logique du programme. Plus `main` est grand, plus nous aurons des variables à importer dans la portée ; plus nous avons des variables dans notre portée, plus il sera difficile de se souvenir à quoi elles servent. Il est préférable de regrouper les variables de configuration dans une structure pour clarifier leur usage.

Le troisième problème est que nous avons utilisé `expect` pour afficher un message d'erreur lorsque la lecture du fichier échoue, mais le message affiche uniquement `Quelque chose s'est mal passé lors de la lecture du fichier`. Lire un fichier peut échouer pour de nombreuses raisons : par exemple, le fichier peut ne pas exister, ou parce que nous n'avons pas le droit de l'ouvrir. Pour le moment, quelle que soit la raison, nous affichons le message d'erreur `Quelque chose s'est mal passé lors de la lecture du fichier`, ce qui ne donne aucune information à l'utilisateur !

Quatrièmement, nous utilisons `expect` à répétition pour gérer les différentes erreurs, et si l'utilisateur lance notre programme sans renseigner d'arguments, il va avoir une erreur `index out of bounds` provenant de Rust, qui n'explique pas clairement le problème. Il serait plus judicieux que tout le code de gestion des erreurs se trouve au même endroit afin que les futurs mainteneurs n'aient qu'un seul endroit à consulter dans le code si la logique de gestion des erreurs doit être modifiée. Avoir tout le code de gestion des erreurs dans un seul endroit va aussi garantir que nous affichons des messages qui ont du sens pour les utilisateurs.

Corrigeons ces quatre problèmes en remaniant notre projet.

Séparation des tâches des projets de binaires

Le problème de l'organisation de la répartition des tâches multiples dans la fonction `main` est commun à de nombreux projets binaires. En conséquence, la communauté Rust a développé une procédure à utiliser comme ligne conductrice pour partager les tâches d'un programme binaire lorsque `main` commence à grossir. Le processus se décompose selon les étapes suivantes :

- Diviser votre programme dans un *main.rs* et un *lib.rs* et déplacer la logique de votre programme dans *lib.rs*.
- Tant que votre logique d'interprétation de la ligne de commande est peu volumineuse, elle peut rester dans le *main.rs*
- Lorsque la logique d'interprétation de la ligne de commande commence à devenir compliquée, il faut la déplacer du *main.rs* vers le *lib.rs*.

Les fonctionnalités qui restent dans la fonction `main` après cette procédure seront les suivantes :

- Appeler la logique d'interprétation de ligne de commande avec les valeurs des arguments
- Régler toutes les autres configurations
- Appeler une fonction `run` de *lib.rs*
- Gérer l'erreur si `run` retourne une erreur

Cette structure permet de séparer les responsabilités : *main.rs* se charge de lancer le programme, et *lib.rs* renferme toute la logique des tâches à accomplir. Comme vous ne pouvez pas directement tester la fonction `main` , cette structure vous permet de tester toute la logique de votre programme en les déplaçant dans des fonctions dans *lib.rs*. Le seul code qui restera dans le *main.rs* sera suffisamment petit pour s'assurer qu'il soit correct en le lisant. Lançons-nous dans le remaniement de notre programme en suivant cette procédure.

Extraction de l'interpréteur des arguments

Nous allons déplacer la fonctionnalité de l'interprétation des arguments dans une fonction que `main` va appeler afin de préparer le déplacement de la logique de l'interpréteur dans *src/lib.rs*. L'encart 12-5 montre le nouveau début du `main` qui appelle une nouvelle fonction `interpreter_config` , que nous allons définir dans *src/main.rs* pour le moment.

Fichier : src/main.rs

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

    let (recherche, nom_fichier) = interpreter_config(&args);

    // -- partie masquée ici --
}

fn interpreter_config(args: &[String]) -> (&str, &str) {
    let recherche = &args[1];
    let nom_fichier = &args[2];

    (recherche, nom_fichier)
}
```

Encart 12-5 : Extraction d'une fonction `interpreter_config` à partir de `main`

Nous continuons à récupérer les arguments de la ligne de commande dans un vecteur, mais au lieu d'assigner la valeur de l'argument d'indice 1 à la variable `recherche` et la valeur de l'argument d'indice 2 à la variable `nom_fichier` dans la fonction `main`, nous passons le vecteur entier à la fonction `interpreter_config`. La fonction `interpreter_config` renferme la logique qui détermine quel argument va dans quelle variable et renvoie les valeurs au `main`. Nous continuons à créer les variables `recherche` et `nom_fichier` dans le `main`, mais `main` n'a plus la responsabilité de déterminer quelles sont les variables qui correspondent aux arguments de la ligne de commande.

Ce remaniement peut sembler excessif pour notre petit programme, mais nous remanions de manière incrémentale par de petites étapes. Après avoir fait ces changements, lancez à nouveau le programme pour vérifier que l'envoi des arguments fonctionne toujours. C'est une bonne chose de vérifier souvent lorsque vous avancez, pour vous aider à mieux identifier les causes de problèmes lorsqu'ils apparaissent.

Grouper les valeurs de configuration

Nous pouvons appliquer une nouvelle petite étape pour améliorer la fonction `interpreter_config`. Pour le moment, nous retournons un tuple, mais ensuite nous divisons immédiatement ce tuple à nouveau en plusieurs éléments. C'est un signe que nous n'avons peut-être pas la bonne approche.

Un autre signe qui indique qu'il y a encore de la place pour de l'amélioration est la partie `config` de `interpreter_config` qui sous-entend que les deux valeurs que nous retournons sont liées et font partie d'une même valeur de configuration. Or, à ce stade, nous ne tenons pas compte de cela dans la structure des données que nous utilisons si ce n'est en regroupant les deux valeurs dans un tuple ; nous pourrions mettre les deux valeurs dans

une seule structure et donner un nom significatif à chacun des champs de la structure. Faire ainsi permet de faciliter la compréhension du code par les futurs développeurs de ce code pour mettre en évidence le lien entre les deux valeurs et leurs rôles respectifs.

L'encart 12-6 montre les améliorations apportées à la fonction `interpreter_config`.

Fichier : src/main.rs

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

    let config = interpreter_config(&args);

    println!("On recherche : {}", config.recherche);
    println!("Dans le fichier : {}", config.nom_fichier);

    let contenu = fs::read_to_string(config.nom_fichier)
        .expect("Quelque chose s'est mal passé lors de la lecture du fichier");

    // -- partie masquée ici --
}

struct Config {
    recherche: String,
    nom_fichier: String,
}

fn interpreter_config(args: &[String]) -> Config {
    let recherche = args[1].clone();
    let nom_fichier = args[2].clone();

    Config { recherche, nom_fichier }
}
```

Encart 12-6 : Remaniement de `interpreter_config` pour retourner une instance de la structure `Config`

Nous avons ajouté une structure `Config` qui a deux champs `recherche` et `nom_fichier`. La signature de `interpreter_config` indique maintenant qu'elle retourne une valeur `Config`. Dans le corps de `interpreter_config`, où nous retournions une slice de chaînes de caractères qui pointaient sur des valeurs `String` présentes dans `args`, nous définissons maintenant la structure `Config` pour contenir des valeurs `String` qu'elle possède. La variable `args` du `main` est la propriétaire des valeurs des arguments et permet uniquement à la fonction `interpreter_config` de les emprunter, ce qui signifie que nous violons les règles d'emprunt de Rust si `Config` essaye de prendre possession des valeurs provenant de `args`.

Nous pourrions gérer les données `String` de plusieurs manières, mais la façon la plus facile, bien que non optimisée, est d'appeler la méthode `clone` sur les valeurs. Cela va produire une copie complète des données pour que l'instance de `Config` puisse se les approprier, ce qui va prendre plus de temps et de mémoire que de stocker une référence vers les données de la chaîne de caractères. Cependant le clonage des données rend votre code très simple car nous n'avons pas à gérer les durées de vie des références ; dans ces circonstances, sacrifier un peu de performances pour gagner en simplicité est un compromis qui en vaut la peine.

Les contre-parties de l'utilisation de `clone`

Il y a une tendance chez les Rustacés de s'interdire l'utilisation de `clone` pour régler les problèmes d'appartenance à cause du coût à l'exécution. Dans le chapitre 13, vous allez apprendre à utiliser des méthodes plus efficaces dans ce genre de situation. Mais pour le moment, ce n'est pas un problème de copier quelques chaînes de caractères pour continuer à progresser car vous allez le faire une seule fois et les chaînes de caractères `nom_fichier` et `recherche` sont très courtes. Il est plus important d'avoir un programme fonctionnel qui n'est pas très optimisé plutôt que d'essayer d'optimiser à outrance le code dès sa première écriture. Plus vous deviendrez expérimenté en Rust, plus il sera facile de commencer par la solution la plus performante, mais pour le moment, il est parfaitement acceptable de faire appel à `clone` .

Nous avons actualisé `main` pour qu'il utilise l'instance de `Config` retournée par `interpreter_config` dans une variable `config` , et nous avons rafraîchi le code qui utilisait les variables séparées `recherche` et `nom_fichier` pour qu'il utilise maintenant les champs de la structure `Config` à la place.

Maintenant, notre code indique clairement que `recherche` et `nom_fichier` sont reliés et que leur but est de configurer le fonctionnement du programme. N'importe quel code qui utilise ces valeurs sait comment les retrouver dans les champs de l'instance `config` grâce à leurs noms donnés à cet effet.

Créer un constructeur pour `Config`

Pour l'instant, nous avons extrait la logique en charge d'interpréter les arguments de la ligne de commande à partir du `main` et nous l'avons placé dans la fonction `interpreter_config` . Cela nous a aidé à découvrir que les valeurs `recherche` et `nom_fichier` étaient liées et que ce lien devait être retranscrit dans notre code. Nous avons ensuite créé une structure

`Config` afin de donner un nom au rôle apparenté à `recherche` et à `nom_fichier`, et pour pouvoir retourner les noms des valeurs sous la forme de noms de champs à partir de la fonction `interpreter_config`.

Maintenant que le but de la fonction `interpreter_config` est de créer une instance de `Config`, nous pouvons transformer `interpreter_config` d'une simple fonction à une fonction `new` qui est associée à la structure `Config`. Ce changement rendra le code plus familier. Habituellement, nous créons des instances de types de la bibliothèque standard, comme `String`, en appelant `String::new`. Si on change le `interpreter_config` en une fonction `new` associée à `Config`, nous pourrons créer de la même façon des instances de `Config` en appelant `Config::new`. L'encart 12-7 nous montre les changements que nous devons faire pour cela.

Fichier : src/main.rs

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

    let config = Config::new(&args);

    // -- partie masquée ici --
}

// -- partie masquée ici --

impl Config {
    fn new(args: &[String]) -> Config {
        let recherche = args[1].clone();
        let nom_fichier = args[2].clone();

        Config { recherche, nom_fichier }
    }
}
```

Encart 12-7 : Transformer `interpreter_config` en `Config::new`

Nous avons actualisé le `main` où nous appelions `interpreter_config` pour appeler à la place le `Config::new`. Nous avons changé le nom de `interpreter_config` par `new` et nous l'avons déplacé dans un bloc `impl`, ce qui relie la fonction `new` à `Config`. Essayez à nouveau de compiler ce code pour vous assurer qu'il fonctionne.

Corriger la gestion des erreurs

Maintenant, nous allons nous pencher sur la correction de la gestion des erreurs. Rappellez-

vous que la tentative d'accéder aux valeurs dans le vecteur `args` aux indices 1 ou 2 va faire paniquer le programme si le vecteur contient moins de trois éléments. Essayez de lancer le programme sans aucun argument ; cela donnera quelque chose comme ceci :

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1 but the index is
1', src/main.rs:27:21
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

La ligne `index out of bounds: the len is 1 but the index is 1` est un message d'erreur destiné aux développeurs. Il n'aidera pas nos utilisateurs finaux à comprendre ce qu'il s'est passé et ce qu'ils devraient faire à la place. Corrigeons cela dès maintenant.

Améliorer le message d'erreur

Dans l'encart 12-8, nous ajoutons une vérification dans la fonction `new`, qui va vérifier que le slice est suffisamment grand avant d'accéder aux indices 1 et 2. Si le slice n'est pas suffisamment grand, le programme va paniquer et afficher un meilleur message d'erreur que le message `index out of bounds`.

Fichier : src/main.rs

```
// -- partie masquée ici --
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("il n'y a pas assez d'arguments");
    }
    // -- partie masquée ici --
```

Encart 12-8 : Ajout d'une vérification du nombre d'arguments

Ce code est similaire à la fonction Supposition::new que nous avons écrit dans l'encart 9-13, dans laquelle nous appelions `panic!` lorsque l'argument `valeur` était hors de l'intervalle des valeurs valides. Plutôt que de vérifier un intervalle de valeurs dans le cas présent, nous vérifions que la taille de `args` est au moins de 3 et que le reste de la fonction puisse fonctionner en s'appuyant sur l'affirmation que cette condition a bien été remplie. Si `args` avait moins de trois éléments, cette fonction serait vraie, et nous appellerions alors la macro `panic!` pour mettre fin au programme immédiatement.

Avec ces quelques lignes de code en plus dans `new`, lançons le programme sans aucun argument à nouveau pour voir à quoi ressemble désormais l'erreur :

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep`
thread 'main' panicked at 'il n'y a pas assez d'arguments', src/main.rs:26:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Cette sortie est meilleure : nous avons maintenant un message d'erreur compréhensible. Cependant, nous avons aussi des informations superflues que nous ne souhaitons pas afficher à nos utilisateurs. Peut-être que la technique que nous avons utilisée dans l'encart 9-13 n'est pas la plus appropriée dans ce cas : un appel à `panic!` est plus approprié pour un problème de développement qu'un problème d'utilisation, comme nous l'avons appris au chapitre 9. A la place, nous pourrions utiliser une autre technique que vous avez apprise au chapitre 9 — retourner un `Result` qui indique si c'est un succès ou une erreur.

Retourner un `Result` à partir de `new` plutôt que d'appeler `panic!`

Nous pouvons à la place retourner une valeur `Result` qui contiendra une instance de `Config` dans le cas d'un succès et va décrire le problème dans le cas d'une erreur. Lorsque `Config::new` communiquera avec le `main`, nous pourrons utiliser le type de `Result` pour signaler où il y a un problème. Ensuite, nous pourrons changer le `main` pour convertir une variante de `Err` dans une erreur plus pratique pour nos utilisateurs sans avoir le texte à propos de `thread 'main'` et de `RUST_BACKTRACE` qui sont provoqués par l'appel à `panic!`.

L'encart 12-9 nous montre les changements que nous devons apporter à la valeur de retour de `Config::new` et le corps de la fonction pour pouvoir retourner un `Result`. Notez que cela ne va pas se compiler tant que nous ne corrigeons pas aussi le `main`, ce que nous allons faire dans le prochain encart.

Fichier : src/main.rs

```
impl Config {
    fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("il n'y a pas assez d'arguments");
        }

        let recherche = args[1].clone();
        let nom_fichier = args[2].clone();

        Ok(Config { recherche, nom_fichier })
    }
}
```

Encart 12-9 : Retourner un `Result` à partir de `Config::new`

Notre fonction `new` retourne désormais un `Result` contenant une instance de `Config` dans le cas d'un succès et une `&'static str` dans le cas d'une erreur. Nos valeurs d'erreur seront toujours des litéraux de chaîne de caractères qui ont la durée de vie `'static`.

Nous avons fait deux changements dans le corps de notre fonction `new` : plutôt que d'avoir à appeler `panic!` lorsque l'utilisateur n'envoie pas assez d'arguments, nous retournons maintenant une valeur `Err`, et nous avons intégré la valeur de retour `Config` dans un `Ok`. Ces modifications rendent la fonction conforme à son nouveau type de signature.

Retourner une valeur `Err` à partir de `Config::new` permet à la fonction `main` de gérer la valeur `Result` retournée par la fonction `new` et de terminer plus proprement le processus dans le cas d'une erreur.

Appeler `Config::new` et gérer les erreurs

Pour gérer les cas d'erreurs et afficher un message correct pour l'utilisateur, nous devons mettre à jour `main` pour gérer le `Result` retourné par `Config::new`, comme dans l'encart 12-10. Nous allons aussi prendre la décision de quitter l'outil en ligne de commande avec un code d'erreur différent de zéro avec `panic!` et nous allons l'implémenter manuellement. Un statut de sortie différent de zéro est une convention pour signaler au processus qui a appelé notre programme que le programme s'est terminé dans un état d'erreur.

Fichier : src/main.rs

```
use std::process;

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

    let config = Config::new(&args).unwrap_or_else(|err| {
        println!("Problème rencontré lors de l'interprétation des arguments :
{}", err);
        process::exit(1);
    });

    // -- partie masquée ici --
```

Encart 12-10 : Quitter avec un code d'erreur si la création d'une nouvelle `Config` échoue.

Dans cet encart, nous avons utilisé une méthode que nous n'avons pas encore détaillée pour l'instant : `unwrap_or_else`, qui est définie sur `Result<T, E>` par la bibliothèque standard. L'utilisation de `unwrap_or_else` nous permet de définir une gestion des erreurs

personnalisée, exempt de `panic!`. Si le `Result` est une valeur `Ok`, le comportement de cette méthode est similaire à `unwrap` : elle retourne la valeur à l'intérieur du `Ok`. Cependant, si la valeur est une valeur `Err`, cette méthode appelle le code dans la *fermeture*, qui est une fonction anonyme que nous définissons et passons en argument de `unwrap_or_else`. Nous verrons les fermetures plus en détail dans le chapitre 13. Pour l'instant, vous avez juste à savoir que le `unwrap_or_else` va passer la valeur interne du `Err` (qui dans ce cas est la chaîne de caractères statique `"pas assez d'arguments"` que nous avons ajoutée dans l'encart 12-9) à notre fermeture dans l'argument `err` qui est présent entre deux barres verticales. Le code dans la fermeture peut ensuite utiliser la valeur `err` lorsqu'il est exécuté.

Nous avons ajouté une nouvelle ligne `use` pour importer `process` dans la portée à partir de la bibliothèque standard. Le code dans la fermeture qui sera exécuté dans le cas d'une erreur fait uniquement deux lignes : nous affichons la valeur de `err` et nous appelons ensuite `process::exit`. La fonction `process::exit` va stopper le programme immédiatement et retourner le nombre qui lui a été donné en paramètre comme code de statut de sortie. C'est semblable à la gestion basée sur `panic!` que nous avons utilisée à l'encart 12-8, mais nous n'avons plus tout le texte en plus. Essayons cela :

```
$ cargo run
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/minigrep`
Problème rencontré lors de l'interprétation des arguments : il n'y a pas assez
d'arguments
```

Très bien ! Cette sortie est bien plus compréhensible pour nos utilisateurs.

Extraction de la logique du `main`

Maintenant que nous avons fini le remaniement de l'interprétation de la configuration, occupons-nous de la logique du programme. Comme nous l'avons dit dans “Séparation des tâches des projets de binaires”, nous allons extraire une fonction `run` qui va contenir toute la logique qui est actuellement dans la fonction `main` qui n'est pas liée au réglage de la configuration ou la gestion des erreurs. Lorsque nous aurons terminé, `main` sera plus concise et facile à vérifier en l'inspectant, et nous pourrons écrire des tests pour toutes les autres logiques.

L'encart 12-11 montre la fonction `run` extraite. Pour le moment, nous faisons des petites améliorations progressives pour extraire les fonctions. Nous continuons à définir la fonction dans *src/main.rs*.

Fichier : src/main.rs

```
fn main() {
    // -- partie masquée ici --

    println!("On recherche : {}", config.recherche);
    println!("Dans le fichier : {}", config.nom_fichier);

    run(config);
}

fn run(config: Config) {
    let contenu = fs::read_to_string(config.nom_fichier)
        .expect("Quelque chose s'est mal passé lors de la lecture du fichier");

    println!("Dans le texte :\n{}", contenu);
}

// -- partie masquée ici --
```

Encart 12-11 : Extraction d'une fonction `run` qui contient le reste de la logique du programme

La fonction `run` contient maintenant toute la logique qui restait dans le `main`, en commençant par la lecture du fichier. La fonction `run` prend l'instance de `Config` en argument.

Retourner des erreurs avec la fonction `run`

Avec le restant de la logique du programme maintenant séparée dans la fonction `run`, nous pouvons améliorer la gestion des erreurs, comme nous l'avons fait avec `Config::new` dans l'encart 12-9. Plutôt que de permettre au programme de paniquer en appelant `expect`, la fonction `run` va retourner un `Result<T, E>` lorsque quelque chose se passe mal. Cela va nous permettre de consolider davantage la logique de gestion des erreurs dans le `main` pour qu'elle soit plus conviviale pour l'utilisateur. L'encart 12-12 montre les changements que nous devons appliquer à la signature et au corps du `run`.

Fichier : src/main.rs

```
use std::error::Error;

// -- partie masquée ici --

fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contenu = fs::read_to_string(config.nom_fichier)?;

    println!("Dans le texte :\n{}", contenu);

    Ok(())
}
```

Encart 12-12 : Changer la fonction `run` pour retourner un `Result`

Nous avons fait trois changements significatifs ici. Premièrement, nous avons changé le type de retour de la fonction `run` en `Result<(), Box<dyn Error>>`. Cette fonction renvoyait précédemment le type unité, `()`, que nous gardons comme valeur de retour dans le cas de `Ok`.

En ce qui concerne le type d'erreur, nous avons utilisé *l'objet trait* `Box<dyn Error>` (et nous avons importé `std::error::Error` dans la portée avec une instruction `use` en haut). Nous allons voir les objets trait dans le chapitre 17. Pour l'instant, retenez juste que `Box<dyn Error>` signifie que la fonction va retourner un type qui implémente le trait `Error`, mais que nous n'avons pas à spécifier quel sera précisément le type de la valeur de retour. Cela nous donne la flexibilité de retourner des valeurs d'erreurs qui peuvent être de différents types dans différents cas d'erreurs. Le mot-clé `dyn` est un raccourci pour “dynamique”.

Deuxièmement, nous avons enlevé l'appel à `expect` pour privilégier l'opérateur `?`, que nous avons vu dans le chapitre 9. Au lieu de faire un `panic!` sur une erreur, `?` va retourner la valeur d'erreur de la fonction courante vers le code qui l'a appelé pour qu'il la gère.

Troisièmement, la fonction `run` retourne maintenant une valeur `Ok` dans les cas de succès. Nous avons déclaré dans la signature que le type de succès de la fonction `run` était `()`, ce qui signifie que nous avons enveloppé la valeur de type unité dans la valeur `Ok`. Cette syntaxe `Ok(())` peut sembler un peu étrange au départ, mais utiliser `()` de cette manière est la façon idéale d'indiquer que nous appelons `run` uniquement pour ses effets de bord ; elle ne retourne pas de valeur dont nous pourrions avoir besoin.

Lorsque vous exécutez ce code, il va se compiler mais il va afficher un avertissement :

```
$ cargo run the poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
warning: unused `Result` that must be used
  --> src/main.rs:19:5
   |
19 |     run(config);
   |     ^^^^^^^^^^^^
   |
   = note: `#[warn(unused_must_use)]` on by default
   = note: this `Result` may be an `Err` variant, which should be handled

warning: `minigrep` (bin "minigrep") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.71s
     Running `target/debug/minigrep the poem.txt`
On recherche : the
Dans le fichier : poem.txt
Dans le texte :
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

Rust nous informe que notre code ignore la valeur `Result` et que cette valeur `Result` pourrait indiquer qu'une erreur s'est passée. Mais nous ne vérifions pas pour savoir si oui ou non il y a eu une erreur, et le compilateur nous rappelle que nous devrions avoir du code de gestion des erreurs ici ! Corrigeons dès à présent ce problème.

Gérer les erreurs retournées par `run` dans `main`

Nous allons vérifier les erreurs et les gérer en utilisant une technique similaire à celle que nous avons utilisée avec `Config::new` dans l'encart 12-10, mais avec une légère différence :

Fichier : src/main.rs

```
fn main() {
    // -- partie masquée ici --

    println!("On recherche : {}", config.recherche);
    println!("Dans le fichier : {}", config.nom_fichier);

    if let Err(e) = run(config) {
        println!("Erreur applicative : {}", e);

        process::exit(1);
    }
}
```

Nous utilisons `if let` plutôt que `unwrap_or_else` pour vérifier si `run` retourne un valeur `Err` et appeler `process::exit(1)` le cas échéant. La fonction `run` ne retourne pas de valeur sur laquelle nous aurions besoin d'utiliser `unwrap` comme avec le `Config::new` qui retournait une instance de `Config`. Comme `run` retourne `()` dans le cas d'un succès, nous nous préoccupons uniquement de détecter les erreurs, donc nous n'avons pas besoin de `unwrap_or_else` pour retourner la valeur extraite car elle sera toujours `()`.

Les corps du `if let` et de la fonction `unwrap_or_else` sont identiques dans les deux cas : nous affichons l'erreur et nous quittons.

Déplacer le code dans une crate de bibliothèque

Notre projet `minigrep` se présente plutôt bien pour le moment ! Maintenant, nous allons diviser notre fichier *src/main.rs* et déplacer du code dans le fichier *src/lib.rs* pour que nous puissions le tester et avoir un fichier *src/main.rs* qui héberge moins de fonctionnalités.

Déplaçons tout le code qui ne fait pas partie de la fonction `main` dans le *src/main.rs* vers le *src/lib.rs* :

- La définition de la fonction `run`
- Les instructions `use` correspondantes
- La définition de `Config`
- La définition de la fonction `Config::new`

Le contenu du *src/lib.rs* devrait contenir les signatures de l'encart 12-13 (nous avons enlevé les corps des fonctions pour des raisons de brièveté). Notez que cela ne va pas se compiler jusqu'à ce que nous modifions le *src/main.rs* dans l'encart 12-14.

Fichier : src/lib.rs

```
use std::error::Error;
use std::fs;

pub struct Config {
    pub recherche: String,
    pub nom_fichier: String,
}

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        // -- partie masquée ici --
    }
}

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // -- partie masquée ici --
}
```

Encart 12-13 : Déplacement de `Config` et de `run` dans *src/lib.rs*

Nous avons fait un usage généreux du mot-clé `pub` : sur `Config`, sur ses champs et sur la méthode `new` et enfin sur la fonction `run`. Nous avons maintenant une crate de bibliothèque qui a une API publique que nous pouvons tester !

Maintenant nous devons importer le code que nous avons déplacé dans *src/lib.rs* dans la portée de la crate binaire dans *src/main.rs*, comme dans l'encart 12-14.

Fichier : src/main.rs

```
use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // -- partie masquée ici --
    if let Err(e) = minigrep::run(config) {
        // -- partie masquée ici --
    }
}
```

Encart 12-14 : Utilisation de la crate de bibliothèque `minigrep` dans *src/main.rs*

Nous avons ajouté une ligne `use minigrep::Config` pour importer le type `Config` de la crate de bibliothèque dans la portée de la crate binaire, et nous avons avons préfixé la fonction `run` avec le nom de notre crate. Maintenant, toutes les fonctionnalités devraient être connectées et devraient fonctionner. Lancez le programme avec `cargo run` pour vous assurer que tout fonctionne correctement.

Ouah ! C'était pas mal de travail, mais nous nous sommes organisés pour nous assurer le succès à venir. Maintenant il est bien plus facile de gérer les erreurs, et nous avons rendu le code plus modulaire. A partir de maintenant, l'essentiel de notre travail sera effectué dans *src/lib.rs*.

Profitons de cette nouvelle modularité en accomplissant quelque chose qui aurait été difficile à faire avec l'ancien code, mais qui est facile avec ce nouveau code : nous allons écrire des tests !

Développer les fonctionnalités de la bibliothèque avec le TDD

Maintenant que nous avons extrait la logique dans *src/lib.rs* et que nous avons laissé la récupération des arguments et la gestion des erreurs dans *src/main.rs*, il est bien plus facile d'écrire les tests pour les fonctionnalités de base de notre code. Nous pouvons appeler les fonctions directement avec différents arguments et vérifier les valeurs de retour sans avoir à appeler notre binaire dans la ligne de commande.

Dans cette section, nous allons ajouter la logique de recherche au programme `minigrep` en utilisant le processus de développement orienté par les tests (c'est le TDD : *Test-Driven Development*). Cette technique de développement de logiciels suit ces trois étapes :

1. Ecrire un test qui échoue et lancez-le pour vous assurer qu'il va échouer pour la raison que vous attendiez.
2. Ecrire ou modifier juste assez de code pour faire réussir ce nouveau test.
3. Remanier le code que vous venez d'ajouter ou de changer pour vous assurer que les tests continuent à réussir.
4. Recommencer à l'étape 1 !

Ce processus n'est qu'une des différentes manières d'écrire des programmes, mais le TDD peut aussi aider à piloter sa conception. Ecrire les tests avant d'écrire le code qui fait réussir les tests aide à maintenir une haute couverture de tests tout le long du processus.

Nous allons expérimenter cela avec l'implémentation de la fonctionnalité qui va rechercher la chaîne de caractères demandée dans le contenu du fichier et générer une liste de lignes qui correspond à cette recherche. Nous ajouterons cette fonctionnalité dans une fonction `rechercher`.

Ecrire un test qui échoue

Comme nous n'en avons plus besoin, enlevons les instructions `println!` de *src/lib.rs* et *src/main.rs* que nous avions utilisé pour vérifier le bon comportement du programme. Ensuite, dans *src/lib.rs*, nous allons ajouter un module `tests` avec une fonction de test, comme nous l'avions fait dans le chapitre 11. La fonction de test définit le comportement que nous voulons qu'ait la fonction `rechercher` : elle va prendre en arguments une recherche et le texte dans lequel rechercher, et elle va retourner seulement les lignes du texte qui correspondent à la recherche. L'encart 12-15 montre ce test, qui ne se compile pas encore.

Fichier : src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn un_resultat() {
        let recherche = "duct";
        let contenu = "\
Rust:
sécurité, rapidité, productivité.
Obtenez les trois en même temps.";

        assert_eq!(
            vec!["sécurité, rapidité, productivité."],
            rechercher(recherche, contenu)
        );
    }
}
```

Encart 12-15 : Création d'un test qui échoue pour la fonction `rechercher` que nous souhaitons concevoir

Ce test recherche la chaîne de caractères `"duct"`. Le texte dans lequel nous recherchons fait trois lignes, et seulement une d'entre elles contient `"duct"` (remarquez que l'antislash après la double-guillet ouvrante indique à Rust de ne pas insérer un caractère de nouvelle ligne au début du contenu de ce litéral de chaîne de caractère). Nous vérifions que la valeur retournée par la fonction `rechercher` contient seulement la ligne que nous avions prévu.

Nous ne pouvons pas encore exécuter ce test et vérifier s'il échoue car même le test ne peut pas se compiler : la fonction `rechercher` n'existe pas encore ! Donc pour le moment nous allons ajouter juste assez de code pour que le test puisse compiler et s'exécuter en ajoutant une définition de la fonction `rechercher` qui retourne un vecteur vide, comme dans l'encart 12-16. Ensuite le test va compiler et échouer car un vecteur vide ne correspond pas au vecteur qui contient la ligne `"sécurité, rapidité, productivité."`

Fichier : src/lib.rs

```
pub fn rechercher<'a>(recherche: &str, contenu: &'a str) -> Vec<&'a str> {
    vec![]
}
```

Encart 12-16 : Définition du strict minimum de la fonction `rechercher` pour que notre test puisse compiler

Remarquez que nous avons besoin de préciser explicitement une durée de vie `'a` définie dans la signature de `rechercher` et l'utiliser sur l'argument `contenu` et la valeur de retour.

Rappelez-vous que dans le chapitre 10 nous avions vu que le paramètre de durée de vie indique quelle durée de vie d'argument est connectée à la durée de vie de la valeur de retour. Dans notre cas, nous indiquons que le vecteur retourné devrait contenir des slices de chaînes de caractères qui proviennent des slices de l'argument `contenu` (et pas de l'argument `recherche`).

Autrement dit, nous disons à Rust que les données retournées par la fonction `rechercher` vont vivre aussi longtemps que la donnée dans l'argument `contenu` de la fonction `rechercher`. C'est très important ! Les données sur lesquelles pointent les slices doivent toujours être en vigueur pour que la référence reste valide ; si le compilateur croit que nous créons des slices de `recherche` plutôt que de `contenu`, ses vérifications de sécurité seront incorrectes.

Si nous oublions les annotations de durée de vie et que nous essayons de compiler cette fonction, nous allons obtenir cette erreur :

```
$ cargo build
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
error[E0106]: missing lifetime specifier
  --> src/lib.rs:28:51
   |
28 | pub fn rechercher(recherche: &str, contenu: &str) -> Vec<&str> {
   |                                 ----           ----         ^ expected named
lifetime parameter
   |
   = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `recherche` or `contenu`
help: consider introducing a named lifetime parameter
   |
28 | pub fn rechercher<'a>(recherche: &'a str, contenu: &'a str) -> Vec<&'a str>
{
   |                   ++++             ++            ++              ++

error: aborting due to previous error

For more information about this error, try `rustc --explain E0106`.
error: could not compile `minigrep` due to previous error
```

Rust ne peut pas deviner lequel des deux arguments nous allons utiliser, donc nous devons lui dire. Comme `contenu` est l'argument qui contient tout notre texte et que nous voulons retourner des extraits de ce texte qui correspondent à la recherche, nous savons que `contenu` est l'argument qui doit être connecté à la valeur de retour, en utilisant la syntaxe de durée de vie.

Les autres langages de programmation n'ont pas besoin que vous connectiez les arguments aux valeurs de retour dans la signature. Bien que cela puisse paraître étrange, cela devient

plus facile au fil du temps. Vous devriez peut-être comparer cet exemple à la section 3 du chapitre 10.

Maintenant, exécutons le test :

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished test [unoptimized + debuginfo] target(s) in 0.97s
     Running unittests (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::un_resultat ... FAILED

failures:

---- tests::un_resultat stdout ----
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `["sécurité, rapidité, productivité."]`,
 right: `[]`', src/lib.rs:44:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::un_resultat

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Très bien, le test a échoué, comme nous nous y attendions. Faisons maintenant en sorte qu'il réussisse !

Ecrire du code pour réussir au test

Pour le moment, notre test échoue car nous retournons toujours un vecteur vide. Pour corriger cela et implémenter `rechercher`, notre programme doit suivre les étapes suivantes :

- Itérer sur chacune des lignes de `contenu`.
- Vérifier si la ligne contient la chaîne de caractères recherchée.
- Si c'est le cas, l'ajouter à la liste des valeurs que nous retournerons.
- Si ce n'est pas le cas, ne rien faire.
- Retourner la liste des résultats qui ont été trouvés.

Travaillons sur chacune de ces étapes, en commençant par l'itération sur les lignes.

Itérer sur chacune des lignes avec la méthode `lines`

Rust a une méthode très pratique pour gérer l'itération ligne-par-ligne des chaînes de caractères, judicieusement appelée `lines`, qui fonctionne comme dans l'encart 12-17. Notez que cela ne se compile pas encore.

Fichier : src/lib.rs

```
pub fn rechercher<'a>(recherche: &str, contenu: &'a str) -> Vec<&'a str> {
    for ligne in contenu.lines() {
        // faire quelquechose avec ligne ici
    }
}
```

Encart 12-17 : Itération sur chacune des lignes de `contenu`

La méthode `lines` retourne un itérateur. Nous verrons plus tard les itérateurs dans le chapitre 13, mais souvenez-vous que vous avez vu cette façon d'utiliser un itérateur dans l'encart 3-5, dans lequel nous avions utilisé une boucle `for` sur un itérateur pour exécuter du code sur chaque élément d'une collection.

Trouver chaque ligne correspondante à la recherche

Ensuite, nous allons vérifier que la ligne courante contient la chaîne de caractères que nous recherchons. Heureusement, les chaînes de caractères ont une méthode `contains` assez pratique qui fait cela pour nous ! Ajoutez l'appel à la méthode `contains` dans la fonction `rechercher`, comme dans l'encart 12-18. Notez qu'ici non plus nous ne pouvons pas encore compiler.

Fichier : src/lib.rs

```
pub fn rechercher<'a>(recherche: &str, contenu: &'a str) -> Vec<&'a str> {
    for ligne in contenu.lines() {
        if ligne.contains(recherche) {
            // faire quelquechose avec la ligne ici
        }
    }
}
```

Encart 12-18 : Ajout d'une fonctionnalité pour trouver quelle ligne contient la chaîne de caractères `recherche`

Stocker les lignes trouvées

Nous avons aussi besoin d'un moyen de stocker les lignes qui contiennent la chaîne de caractères que nous recherchons. Pour cela, nous pouvons créer un vecteur mutable avant la boucle `for` et appeler la méthode `push` pour enregistrer la `ligne` dans le vecteur. Après la boucle `for`, nous retournons le vecteur, comme dans l'encart 12-19 :

Fichier : src/lib.rs

```
pub fn rechercher<'a>(recherche: &str, contenu: &'a str) -> Vec<&'a str> {
    let mut resultats = Vec::new();

    for ligne in contenu.lines() {
        if ligne.contains(recherche) {
            resultats.push(ligne);
        }
    }

    resultats
}
```

Encart 12-19 : Enregistrement des lignes qui sont trouvées afin que nous puissions les retourner

Maintenant, notre fonction `rechercher` retourne uniquement les lignes qui contiennent `recherche`, et notre test devrait réussir. Exécutons le test :

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished test [unoptimized + debuginfo] target(s) in 1.22s
     Running unittests (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 1 test
test tests::un_resultat ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running unittests (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Notre test a réussi, donc nous savons que cela fonctionne !

Arrivé à ce stade, nous pourrions envisager des pistes de remaniement pour l'implémentation de la fonction de recherche tout en faisant en sorte que les tests réussissent toujours afin de conserver les mêmes fonctionnalités. Le code de la fonction de recherche n'est pas mauvais, mais il ne profite pas de quelques fonctionnalités utiles des itérateurs. Nous retrouverons cet exemple dans le chapitre 13, dans lequel nous explorerons les itérateurs en détail, et ainsi découvrir comment nous pourrions l'améliorer.

Utiliser la fonction `rechercher` dans la fonction `run`

Maintenant que la fonction `rechercher` fonctionne et est testée, nous devons appeler `rechercher` dans notre fonction `run`. Nous devons passer à `rechercher` la valeur de `config.recherche` et le `contenu` que `run` obtient en lisant le fichier. Ensuite, `run` devra afficher chaque ligne retournée par `rechercher` :

Fichier : src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contenu = fs::read_to_string(config.nom_fichier)?;

    for ligne in rechercher(&config.recherche, &contenu) {
        println!("{}", ligne);
    }

    Ok(())
}
```

Nous utilisons ici aussi une boucle `for` pour récupérer chaque ligne provenant de `rechercher` et l'afficher.

Maintenant, l'intégralité du programme devrait fonctionner ! Essayons-le, pour commencer avec un mot qui devrait retourner exactement une seule ligne du poème d'Emily Dickinson, “frog” :

```
$ cargo run frog poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.38s
     Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

Super ! Maintenant, essayons un mot qui devrait retourner plusieurs lignes, comme “body” :

```
$ cargo run body poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!
```

Et enfin, assurons-nous que nous n'obtenons aucune ligne lorsque nous cherchons un mot qui n'est nulle part dans le poème, comme “monomorphization” :

```
$ cargo run monomorphization poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep monomorphization poem.txt`
```

Très bien ! Nous avons construit notre propre mini-version d'un outil classique et nous avons beaucoup appris sur la façon de structurer nos applications. Nous en avons aussi appris un peu sur les entrées et sorties des fichiers, les durées de vie, les tests et l'interprétation de la ligne de commande.

Pour clôturer ce projet, nous allons brièvement voir comment travailler avec les variables

d'environnement et comment écrire sur la sortie standard des erreurs, ce qui peut s'avérer utile lorsque vous écrivez des programmes en ligne de commande.

Travailler avec des variables d'environnement

Nous allons améliorer `minigrep` en lui ajoutant une fonctionnalité supplémentaire : une option pour rechercher sans être sensible à la casse que l'utilisateur pourra activer via une variable d'environnement. Nous pourrions appliquer cette fonctionnalité avec une option en ligne de commande et demander à l'utilisateur de la renseigner à chaque fois qu'il veut l'activer, mais à la place nous allons utiliser une variable d'environnement. Ceci permet à nos utilisateurs de régler la variable d'environnement une seule fois et d'avoir leurs recherches insensibles à la casse dans cette session du terminal.

Ecrire un test qui échoue pour la fonction `rechercher` insensible à la casse

Nous souhaitons ajouter une nouvelle fonction `rechercher_insensible_casse` que nous allons appeler lorsque la variable d'environnement est active. Nous allons continuer à suivre le processus de TDD, donc la première étape est d'écrire à nouveau un test qui échoue. Nous allons ajouter un nouveau test pour la nouvelle fonction `rechercher_insensible_casse` et renommer notre ancien test `un_resultat` en `sensible_casse` pour clarifier les différences entre les deux tests, comme dans l'encart 12-20.

Fichier : src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn sensible_casse() {
        let recherche = "duct";
        let contenu = "\
Rust:
sécurité, rapidité, productivité.
Obtenez les trois en même temps.
Duck tape.";

        assert_eq!(vec!["sécurité, rapidité, productivité."],
rechercher(recherche, contenu));
    }

    #[test]
    fn insensible_casse() {
        let recherche = "rUsT";
        let contenu = "\
Rust:
sécurité, rapidité, productivité.
Obtenez les trois en même temps.
C'est pas rustique.";

        assert_eq!(
            vec!["Rust:", "C'est pas rustique."],
            rechercher_insensible_casse(recherche, contenu)
        );
    }
}
```

Encart 12-20 : Ajout d'un nouveau test qui échoue pour la fonction insensible à la casse que nous sommes en train d'ajouter

Remarquez que nous avons aussi modifié le `contenu` de l'ancien test. Nous avons ajouté une nouvelle ligne avec le texte `"Duct tape."` en utilisant un D majuscule qui ne devrait pas correspondre à la recherche `"duct"` lorsque nous recherchons de manière à être sensible à la casse. Ce changement de l'ancien test permet de nous assurer que nous ne casserons pas accidentellement la fonction de recherche sensible à la casse que nous avons déjà implémenté. Ce test devrait toujours continuer à réussir au fur et à mesure que nous progressons sur la recherche insensible à la casse.

Le nouveau test pour la recherche insensible à la casse utilise `"rUsT"` comme recherche. Dans la fonction `rechercher_insensible_casse` que nous sommes en train d'ajouter, la recherche `"rUsT"` devrait correspondre à la ligne qui contient `"Rust:"` avec un R majuscule ainsi que la ligne `C'est pas rustique.` même si ces deux cas ont des casses différentes de

la recherche. C'est notre test qui doit échouer, et il ne devrait pas se compiler car nous n'avons pas encore défini la fonction `rechercher_insensible_casse`. Ajoutez son implémentation qui retourne toujours un vecteur vide, de la même manière que nous l'avions fait pour la fonction `rechercher` dans l'encart 12-16 pour voir si les tests se compilent et échouent.

Implémenter la fonction `rechercher_insensible_casse`

La fonction `rechercher_insensible_casse`, présente dans l'encart 12-21, sera presque la même que la fonction `rechercher`. La seule différence est que nous allons transformer en minuscule le contenu de `recherche` et de chaque `ligne` pour que quelle que soit la casse des arguments d'entrée, nous aurons toujours la même casse lorsque nous vérifierons si la ligne contient la recherche.

Fichier : src/lib.rs

```
pub fn rechercher_insensible_casse<'a>(
    recherche: &str,
    contenu: &'a str
) -> Vec<&'a str> {
    let recherche = recherche.to_lowercase();
    let mut resultats = Vec::new();

    for ligne in contenu.lines() {
        if ligne.to_lowercase().contains(&recherche) {
            resultats.push(ligne);
        }
    }

    resultats
}
```

Encart 12-21 : Définition de la fonction `rechercher_insensible_casse` pour obtenir en minuscule la recherche et la ligne avant de les comparer

D'abord, nous obtenons la chaîne de caractères `recherche` en minuscule et nous l'enregistrons dans une variable masquée avec le même nom. L'appel à `to_lowercase` sur la recherche est nécessaire afin que quel que soit la recherche de l'utilisateur, comme `"rust"`, `"RUST"`, `"Rust"`, ou `"rUsT"`, nous traitons la recherche comme si elle était `"rust"` et par conséquent elle est insensible à la casse. La méthode `to_lowercase` devrait gérer de l'Unicode de base, mais ne sera pas fiable à 100%. Si nous avions écrit une application sérieuse, nous aurions dû faire plus de choses à ce sujet, toutefois vu que la section actuelle traite des variables d'environnement et pas de la gestion de l'Unicode, nous allons conserver

ce code simplifié.

Notez que `recherche` est désormais une `String` et non plus une slice de chaîne de caractères, car l'appel à `to_lowercase` crée des nouvelles données au lieu de modifier les données déjà existantes. Par exemple, disons que la recherche est `"rUsT"` : cette slice de chaîne de caractères ne contient pas de `u` ou de `t` minuscule que nous pourrions utiliser, donc nous devons allouer une nouvelle `String` qui contient `"rust"`. Maintenant, lorsque nous passons `recherche` en argument de la méthode `contains`, nous devons rajouter une esperluette car la signature de `contains` est définie pour prendre une slice de chaîne de caractères.

Ensuite, nous ajoutons un appel à `to_lowercase` sur chaque `ligne` avant de vérifier si elle contient `recherche` afin d'obtenir tous ses caractères en minuscule. Maintenant que nous avons `ligne` et `recherche` en minuscules, nous allons rechercher les correspondances peu importe la casse de la recherche.

Voyons si cette implémentation passe les tests :

```
$ cargo test
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished test [unoptimized + debuginfo] target(s) in 1.33s
     Running unittests (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 2 tests
test tests::sensible_casse ... ok
test tests::insensible_casse ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running unittests (target/debug/deps/minigrep-9cd200e5fac0fc94)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests minigrep

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Très bien ! Elles ont réussi. Maintenant, utilisons la nouvelle fonction `rechercher_insensible_casse` dans la fonction `run`. Pour commencer, nous allons ajouter

une option de configuration à la structure `Config` pour changer entre la recherche sensible et non sensible à la casse. L'ajout de ce champ va causer des erreurs de compilation car nous n'avons jamais initialisé ce champ pour le moment :

Fichier : src/lib.rs

```
pub struct Config {
    pub recherche: String,
    pub nom_fichier: String,
    pub sensible_casse: bool,
}
```

Remarquez que le champ `sensible_casse` que nous avons ajouté est un Booléen. Ensuite, nous devons faire en sorte que la fonction `run` vérifie la valeur du champ `sensible_casse` et l'utilise pour décider si elle doit appeler la fonction `rechercher` ou la fonction `rechercher_insensible_casse`, comme dans l'encart 12-22. Notez que cela ne se compile pas encore.

Fichier : src/lib.rs

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contenu = fs::read_to_string(config.nom_fichier)?;

    let resultats = if config.sensible_casse {
        rechercher(&config.recherche, &contenu)
    } else {
        rechercher_insensible_casse(&config.recherche, &contenu)
    };

    for ligne in resultats {
        println!("{}", ligne);
    }

    Ok(())
}
```

Encart 12-22 : Appeler `rechercher` ou `rechercher_insensible_casse` en fonction de la valeur dans `config.sensible_casse`

Enfin, nous devons vérifier la variable d'environnement. Les fonctions pour travailler avec les variables d'environnement sont dans le module `env` de la bibliothèque standard, donc nous allons importer ce module dans la portée avec une ligne `use std::env;` en haut de *src/lib.rs*. Ensuite, nous allons utiliser la fonction `var` du module `env` pour vérifier la présence d'une variable d'environnement `MINIGREP_INSENSIBLE_CASSE`, comme dans l'encart 12-23.

Fichier : src/lib.rs

```
use std::env;
// -- partie masquée ici --

impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("il n'y a pas assez d'arguments");
        }

        let recherche = args[1].clone();
        let nom_fichier = args[2].clone();

        let sensible_casse = env::var("MINIGREP_INSENSIBLE_CASSE").is_err();

        Ok(Config {
            recherche,
            nom_fichier,
            sensible_casse,
        })
    }
}
```

Encart 12-23 : Vérification de la présence de la variable d'environnement `MINIGREP_INSENSIBLE_CASSE`

Ici, nous créons une nouvelle variable `sensible_casse`. Pour lui donner une valeur, nous appelons la fonction `env::var` et nous lui passons le nom de la variable d'environnement `MINIGREP_INSENSIBLE_CASSE`. La fonction `env::var` retourne un `Result` qui sera en cas de succès la variante `Ok` qui contiendra la valeur de la variable d'environnement si cette variable d'environnement est définie. Elle retournera la variante `Err` si cette variable d'environnement n'est pas définie.

Nous utilisons la méthode `is_err` sur le `Result` pour vérifier si nous obtenons une erreur, signalant par conséquent que la variable d'environnement n'est pas définie et donc que nous *devons* effectuer une recherche sensible à la casse. Si la variable d'environnement `MINIGREP_INSENSIBLE_CASSE` a une valeur qui lui a été assignée, `is_err` va retourner `false` et le programme va procéder à une recherche non sensible à la casse. Nous ne nous préoccupons pas de la *valeur* de la variable d'environnement, mais uniquement de savoir si elle est définie ou non, donc nous utilisons `is_err` plutôt que `unwrap`, `expect` ou toute autre méthode que nous avons vue avec `Result`.

Nous passons la valeur de la variable `sensible_casse` à l'instance de `Config` afin que la fonction `run` puisse lire cette valeur et décider d'appeler `rechercher` ou `rechercher_insensible_casse`, comme nous l'avons implémenté dans l'encart 12-22.

Faisons un essai ! D'abord, nous allons lancer notre programme avec la variable

d'environnement non définie et avec la recherche `to`, qui devrait trouver toutes les lignes qui contiennent le mot “to” en minuscules :

```
$ cargo run to poem.txt
   Compiling minigrep v0.1.0 (file:///projects/minigrep)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

On dirait que cela fonctionne ! Maintenant, lançons le programme avec `MINIGREP_INSENSIBLE_CASSE` définie à `1` mais avec la même recherche `to`.

Si vous utilisez PowerShell, vous allez avoir besoin d'affecter la variable d'environnement puis exécuter le programme avec deux commandes distinctes :

```
PS> $Env:MINIGREP_INSENSIBLE_CASSE=1; cargo run to poem.txt
```

Cela va faire persister la variable `MINIGREP_INSENSIBLE_CASSE` pour la durée de votre session de terminal. Elle peut être désaffectée avec la cmdlet `Remove-Item` :

```
PS> Remove-Item Env:MINIGREP_INSENSIBLE_CASSE
```

Nous devrions trouver cette fois-ci également toutes les lignes qui contiennent “to” écrit avec certaines lettres en majuscule:

```
$ CASE_INSENSITIVE=1 cargo run to poem.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

Très bien, nous avons aussi obtenu les lignes qui contiennent “To” ! Notre programme `minigrep` peut maintenant faire des recherches insensibles à la casse, contrôlées par une variable d'environnement. Vous savez maintenant comment gérer des options définies soit par des arguments en ligne de commande, soit par des variables d'environnement.

Certains programmes permettent d'utiliser les arguments *et* les variables d'environnement pour un même réglage. Dans ce cas, le programme décide si l'un ou l'autre a la priorité. Pour vous exercer à nouveau, essayez de contrôler la sensibilité à la casse via un argument de ligne de commande ou une variable d'environnement. Vous devrez choisir qui de l'argument de la ligne de commande ou de la variable d'environnement doit être prioritaire lorsque les deux sont configurés simultanément mais de manière contradictoire quand le programme

est exécuté.

Le module `std::env` contient plein d'autres fonctionnalités utiles pour utiliser les variables d'environnement : regardez sa documentation pour voir ce qu'il est possible de faire.

Ecrire les messages d'erreur sur la sortie d'erreur standard au lieu de la sortie normale

Pour l'instant, nous avons écrit toutes nos sorties du terminal en utilisant la macro `println!`. Dans la plupart des terminaux, il y a deux genres de sorties : la *sortie standard* (`stdout`) pour les informations générales et *la sortie d'erreur standard* (`stderr`) pour les messages d'erreur. Cette distinction permet à l'utilisateur de choisir de rediriger la sortie des messages sans erreurs d'un programme vers un fichier mais continuer à afficher les messages d'erreur à l'écran.

La macro `println!` ne peut écrire que sur la sortie standard, donc nous devons utiliser autre chose pour écrire sur la sortie d'erreur standard.

Vérifier où sont écrites les erreurs

Commençons par observer comment le contenu écrit par `minigrep` est actuellement écrit sur la sortie standard, y compris les messages d'erreur que nous souhaitons plutôt écrire sur la sortie d'erreur standard. Nous allons faire cela en redirigeant le flux de sortie standard vers un fichier pendant que nous déclencherons intentionnellement une erreur. Nous ne redirigerons pas le flux de sortie d'erreur standard, si bien que n'importe quel contenu envoyé à la sortie d'erreur standard va continuer à s'afficher à l'écran.

Les programmes en ligne de commande sont censés envoyer leurs messages d'erreur dans le flux d'erreurs standard afin que nous puissions continuer à voir les messages d'erreurs à l'écran même si nous redirigeons le flux de la sortie standard dans un fichier. Notre programme ne se comporte pas comme il le devrait : nous allons voir qu'à la place, il envoie les messages d'erreur dans le fichier !

Pour démontrer ce comportement, il faut exécuter le programme avec `>` suivi du nom du fichier, *sortie.txt*, dans lequel nous souhaitons rediriger le flux de sortie standard. Nous ne fournissons aucun argument, ce qui va causer une erreur :

```
$ cargo run > sortie.txt
```

La syntaxe indique à l'invite de commande d'écrire le contenu de la sortie standard dans *sortie.txt* plutôt qu'à l'écran. Nous n'avons pas vu le message d'erreur que nous nous attendions de voir à l'écran, ce qui veut dire qu'il a dû finir dans le fichier. Voici ce que *sortie.txt* contient :

```
Problème rencontré lors de l'interprétation des arguments : il n'y a pas assez
d'arguments
```

Effectivement, notre message d'erreur est écrit sur la sortie standard. Il serait bien plus utile que les messages d'erreur comme celui-ci soient écrits sur la sortie d'erreur standard afin que seules les données produites par exécution fructueuse finissent dans le fichier. Nous allons corriger cela.

Ecrire les erreurs sur la sortie d'erreur standard

Nous allons utiliser le code de l'encart 12-24 pour changer la manière dont les messages d'erreur sont écrits. Grâce au remaniement que nous avons fait plus tôt dans ce chapitre, tout le code qui écrit les messages d'erreurs se trouve dans une seule fonction, `main`. La bibliothèque standard fournit la macro `eprintln!` qui écrit dans le flux d'erreur standard, donc changeons les deux endroits où nous appelons `println!` afin d'utiliser `eprintln!` à la place.

Fichier : src/main.rs

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

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problème rencontré lors de l'interprétation des arguments :
{}", err);
        process::exit(1);
    });

    if let Err(e) = minigrep::run(config) {
        eprintln!("Erreur applicative : {}", e);

        process::exit(1);
    }
}
```

Encart 12-24 : Ecrire les messages d'erreur sur la sortie d'erreur standard au lieu de la sortie standard en utilisant `eprintln!`

Après avoir changé `println!` en `eprintln!`, exécutons à nouveau le programme de la même manière, sans aucun argument et en redirigeant la sortie standard avec `>` :

```
$ cargo run > sortie.txt
Problème rencontré lors de l'interprétation des arguments : il n'y a pas assez
d'arguments
```

Désormais nous pouvons voir l'erreur à l'écran et *sortie.txt* ne contient rien, ce qui est le comportement que nous attendons d'un programme en ligne de commande.

Exécutons le programme à nouveau avec des arguments qui ne causent pas d'erreur tout en continuant à rediriger la sortie standard vers un fichier, comme ceci :

```
$ cargo run to poem.txt > sortie.txt
```

Nous ne voyons rien sur la sortie du terminal, et *sortie.txt* devrait contenir notre résultat :

Fichier : sortie.txt

```
Are you nobody, too?
How dreary to be somebody!
```

Ceci prouve qu'en fonction des circonstances, nous utilisons maintenant la sortie standard pour la sortie sans les erreurs et l'erreur standard pour la sortie d'erreur.

Résumé

Ce chapitre a résumé certains des concepts majeurs que vous avez appris précédemment et expliqué comment procéder à des opérations courantes sur les entrées/sorties en Rust. En utilisant les arguments en ligne de commande, les fichiers, les variables d'environnement et la macro `eprintln!` pour écrire les erreurs, vous pouvez désormais écrire des applications en ligne de commande. En suivant les concepts vus dans les chapitres précédents, votre code restera bien organisé, stockera les données dans les bonnes structures de données, gérera correctement les erreurs et sera correctement testé.

Maintenant, nous allons découvrir quelques fonctionnalités de Rust qui ont été influencées par les langages fonctionnels : les fermetures et les itérateurs.

Les fonctionnalités des langages fonctionnels : les itérateurs et les fermetures

La conception de Rust s'est inspirée de nombreux langages et technologies existantes, et une de ses influences la plus marquante est la *programmation fonctionnelle*. La programmation fonctionnelle consiste souvent à utiliser une fonction comme une valeur en la passant en argument d'une autre fonction, la retourner en résultat d'une autre fonction ou l'assigner à une variable pour l'exécuter plus tard, par exemple.

Dans ce chapitre, nous n'allons pas débattre sur ce qu'est ou non la programmation fonctionnelle, mais nous allons plutôt voir quelques fonctionnalités de Rust qui sont similaires à celles des autres langages souvent considérés comme fonctionnels.

Plus précisément, nous allons voir :

- *les fermetures*, une construction qui ressemble à une fonction que vous pouvez stocker dans une variable
- *les itérateurs*, une façon de travailler sur une série d'éléments
- Comment utiliser ces deux fonctionnalités pour améliorer le projet d'entrée/sortie du chapitre 12
- Etudier la performance de ces deux fonctionnalités (divulgâchage : elles sont probablement plus rapides que ce que vous pensez !)

Les autres fonctionnalités de Rust, comme le filtrage par motif et les énumérations, que nous avons vues dans les chapitres précédents sont influencés par la programmation fonctionnelle. La maîtrise des fermetures et des itérateurs est une étape importante pour écrire du code Rust performant, c'est pourquoi nous allons leur dédier ce chapitre entier.

Les fermetures : fonctions anonymes qui peuvent utiliser leur environnement

Les fermetures en Rust sont des fonctions anonymes qui peuvent être sauvegardées dans une variable ou qui peuvent être passées en argument à d'autres fonctions. Il est possible de créer une fermeture à un endroit du code et ensuite de l'appeler dans un contexte différent pour l'exécuter. Contrairement aux fonctions, les fermetures ont la possibilité de capturer les valeurs présentes dans le contexte où elles sont appelées. Nous allons montrer comment les fonctionnalités des fermetures permettent de réutiliser du code et suivre des comportements personnalisés.

Créer une abstraction de comportement avec une fermeture

Travaillons sur un exemple d'une situation où il est utile de stocker une fermeture qui s'exécutera ultérieurement. Au cours de ce chapitre, nous allons parler de la syntaxe des fermetures, de l'inférence de type et des traits.

Imaginons la situation suivante : nous travaillons dans une *startup* qui crée une application destinée à générer des programmes d'entraînements physiques personnalisés. L'application dorsale est écrite en Rust et repose sur un algorithme qui génère les exercices en fonction de beaucoup de facteurs tels que l'âge de l'utilisateur, son indice de masse corporelle, ses préférences et une intensité qu'il aura paramétré. L'algorithme réellement utilisé n'est pas important pour cet exemple : ce qui est important c'est que le calcul prenne plusieurs secondes. Nous voulons appeler l'algorithme uniquement lorsque nous en avons besoin, et seulement une fois, afin que l'utilisateur n'ait pas à attendre plus longtemps que nécessaire.

Pour simuler l'appel à cet algorithme hypothétique, nous allons utiliser la fonction `simuler_gros_calcul` présent dans l'encart 13-1, qui affichera `calcul très lent ...` et attendra deux secondes avant de retourner le nombre qui lui a été donné :

Fichier : src/main.rs

```
use std::thread;
use std::time::Duration;

fn simuler_gros_calcul(intensite: u32) -> u32 {
    println!("calcul très lent ...");
    thread::sleep(Duration::from_secs(2));
    intensite
}
```

Encart 13-1 : une fonction pour remplacer un calcul hypothétique qui prend environ deux

secondes à s'exécuter

Ensuite, nous avons la fonction `main` qui contient les parties de l'application d'entraînement qui sont importantes pour cet exemple. Cette fonction représente le code que l'application appellera lorsqu'un utilisateur demande un programme d'entraînement. Comme l'interaction avec l'interface frontale de l'application n'apporte rien dans l'étude de l'utilisation des fermetures qui nous occupe ici, nous allons nous contenter de coder en dur les valeurs représentant les entrées de notre programme puis afficher les résultats obtenus.

Les paramètres d'entrées nécessaires sont :

- `intensite` qui est un nombre saisi par utilisateur lorsqu'il demande un entraînement afin d'indiquer s'il veut un entraînement de faible ou de haute intensité.
- Un nombre aléatoire faisant varier les programmes d'entraînement

Le résultat sera le programme d'entraînement recommandé. L'encart 13-2 montre la fonction `main` que nous allons utiliser.

Fichier : src/main.rs

```
fn main() {
    let valeur_utilisateur_simule = 10;
    let nombre_aleatoire_simule = 7;

    generer_exercices(valeur_utilisateur_simule, nombre_aleatoire_simule);
}
```

Encart 13-2 : une fonction `main` avec des valeurs codées en dur pour simuler la saisie d'une valeur d'intensité par l'utilisateur et la génération d'un nombre aléatoire

Nous avons codé en dur la variable `valeur_utilisateur_simule` à 10 et la variable `nombre_aleatoire_simule` à 7 pour des raisons de simplicité ; dans un vrai programme nous obtiendrions la valeur d'intensité à partir de l'interface frontale et nous utiliserions la crate `rand` pour générer un nombre aléatoire, comme nous l'avons fait dans l'exemple du jeu du plus ou du moins dans le chapitre 2. La fonction `main` appelle une fonction `generer_exercices` avec ces valeurs d'entrée simulées.

Maintenant que nous avons le contexte, passons à l'algorithme. La fonction `generer_exercices` dans l'encart 13-3 contient la logique métier de l'application qui nous préoccupe le plus dans cet exemple. Le reste des changements de code dans cet exemple seront appliqués à cette fonction :

Fichier : src/main.rs

```
fn generer_exercices(intensite: u32, nombre_aleatoire: u32) {
    if intensite < 25 {
        println!(
            "Aujourd'hui, faire {} pompes !",
            simuler_gros_calcul(intensite)
        );
        println!(
            "Ensuite, faire {} abdominaux !",
            simuler_gros_calcul(intensite)
        );
    } else {
        if nombre_aleatoire == 3 {
            println!("Faites une pause aujourd'hui ! Rappelez-vous de bien vous
hydrater !");
        } else {
            println!(
                "Aujourd'hui, courrez pendant {} minutes !",
                simuler_gros_calcul(intensite)
            );
        }
    }
}
```

Encart 13-3 : la logique métier qui affiche les programmes d'entraînement en fonction des entrées et des appels à la fonction `simuler_gros_calcul`.

Le code de l'encart 13-3 a plusieurs appels à la fonction de calcul lent : le premier bloc `if` appelle `simuler_gros_calcul` deux fois, le `if` à l'intérieur du `else` ne l'appelle pas du tout, et le code à l'intérieur du second `else` l'appelle une seule fois.

Le comportement souhaité de la fonction `generer_exercices` est de vérifier d'abord si l'utilisateur veut un entraînement de faible intensité (indiqué par un nombre inférieur à 25) ou un entraînement de haute intensité (un nombre de 25 ou plus).

Les plans d'entraînement à faible intensité recommanderont un certain nombre de pompes et d'abdominaux basés sur l'algorithme complexe que nous simulons.

Si l'utilisateur souhaite un entraînement de haute intensité, il y a une logique en plus : si la valeur du nombre aléatoire généré par l'application est 3, l'application recommandera une pause et une hydratation à la place. Sinon, l'utilisateur recevra un nombre de minutes de course à pied calculé par l'algorithme complexe.

Ce code fonctionne comme la logique métier le souhaite, mais imaginons que l'équipe de science des données nous informe qu'il va y avoir des changements dans la façon dont nous devrons appeler l'algorithme à l'avenir. Pour simplifier la mise à jour lorsque ces changements se produiront, nous voulons remanier ce code de sorte qu'il n'appelle la fonction `simuler_gros_calcul` qu'une seule fois. Nous voulons également nous

débarrasser de l'endroit où nous appelons la fonction deux fois inutilement, sans ajouter d'autres appels à cette fonction au cours de ce processus. Autrement dit, nous ne voulons pas l'appeler si le résultat n'en a pas besoin, et nous voulons ne l'appeler qu'une seule fois.

Remaniement en utilisant des fonctions

Nous pourrions restructurer le programme d'entraînement de plusieurs manières. Tout d'abord, nous allons essayer d'extraire l'appel en double à la fonction `simuler_gros_calcul` dans une variable, comme dans l'encart 13-4 :

Fichier : src/main.rs

```
fn generer_exercices(intensite: u32, nombre_aleatoire: u32) {
    let resultat_lent = simuler_gros_calcul(intensite);

    if intensite < 25 {
        println!("Aujourd'hui, faire {} pompes !", resultat_lent);
        println!("Ensuite, faire {} abdominaux !", resultat_lent);
    } else {
        if nombre_aleatoire == 3 {
            println!("Faites une pause aujourd'hui ! Rappelez-vous de bien vous
hydrater !");
        } else {
            println!("Aujourd'hui, courrez pendant {} minutes !",
resultat_lent);
        }
    }
}
```

Encart 13-4 : extraction des appels à `simuler_gros_calcul` dans un seul endroit et stockage du résultat dans la variable `resultat_lent` .

Ce changement unifie tous les appels à `simuler_gros_calcul` et résout le problème du premier bloc `if` qui appelle inutilement la fonction à deux reprises. Malheureusement, nous appelons maintenant cette fonction et attendons le résultat dans tous les cas, ce qui inclut le bloc `if` interne qui n'utilise pas du tout la valeur du résultat.

Nous voulons nous référer à `simuler_gros_calcul` qu'une seule fois dans `generer_exercices` , mais retarder le gros calcul jusqu'au moment où nous avons avons réellement besoin du résultat. C'est un cas d'utilisation des fermetures !

Remanier le code avec des fermetures pour stocker du code

Au lieu d'appeler systématiquement la fonction `simuler_gros_calcul` avant les blocs `if` ,

nous pouvons définir une fermeture et la stocker dans une variable au lieu de le faire pour le résultat, comme le montre l'encart 13-5. Nous pouvons en fait déplacer l'ensemble du corps de `simuler_gros_calcul` dans la fermeture que nous introduisons ici.

Fichier : src/main.rs

```
let fermeture_lente = |nombre| {
    println!("calcul très lent ...");
    thread::sleep(Duration::from_secs(2));
    nombre
};
```

Encart 13-5 : définition d'une fermeture et son enregistrement dans la variable `fermeture_lente`.

La définition de la fermeture vient après le `=` pour l'assigner à la variable `fermeture_lente`. Pour définir une fermeture, on commence par une paire de barres verticales (`|`), à l'intérieur desquelles on renseigne les paramètres de la fermeture ; cette syntaxe a été choisie en raison de sa similitude avec les définitions des fermetures en Smalltalk et en Ruby. Cette fermeture a un paramètre `nombre` : si nous avions plus d'un paramètre, nous les séparerions par des virgules, comme ceci : `|param1, param2|`.

Après les paramètres, on ajoute des accolades qui contiennent le corps de la fermeture, celles-ci sont facultatives si le corps de la fermeture est une seule expression. Après les accolades, nous avons besoin d'un point-virgule pour terminer l'instruction `let`. La valeur à la dernière ligne dans le corps de la fermeture (`nombre`) sera la valeur retournée par la fermeture lorsqu'elle sera exécutée, et cette ligne ne se termine pas par un point-virgule, exactement comme dans le corps des fonctions.

Notez que cette instruction `let` signifie que la variable `fermeture_lente` contient la *définition* d'une fonction anonyme, pas la *valeur résultante* à l'appel de cette fonction anonyme. Rappelons que nous utilisons une fermeture pour définir le code à appeler dans un seul endroit, stocker ce code et l'appeler plus tard ; le code que nous voulons appeler est maintenant stocké dans `fermeture_lente`.

Maintenant que nous avons défini la fermeture, nous pouvons changer le code dans les blocs `if` pour appeler la fermeture afin d'exécuter le code et obtenir la valeur résultante. L'appel d'une fermeture fonctionne comme pour l'appel d'une fonction : nous renseignons le nom de la variable qui stocke la définition de la fermeture et la complétons avec des parenthèses contenant les valeurs du ou des arguments que nous voulons utiliser pour cet appel, comme dans l'encart 13-6.

Fichier : src/main.rs

```
fn generer_exercices(intensite: u32, nombre_aleatoire: u32) {
    let fermeture_lente = |nombre| {
        println!("calcul très lent ...");
        thread::sleep(Duration::from_secs(2));
        nombre
    };

    if intensite < 25 {
        println!("Aujourd'hui, faire {} pompes !", fermeture_lente(intensite));
        println!("Ensuite, faire {} abdominaux !", fermeture_lente(intensite));
    } else {
        if nombre_aleatoire == 3 {
            println!("Faites une pause aujourd'hui ! Rappelez-vous de bien vous
hydrater !");
        } else {
            println!(
                "Aujourd'hui, courrez pendant {} minutes !",
                fermeture_lente(intensite)
            );
        }
    }
}
```

Encart 13-6 : appel de la fermeture `fermeture_lente` que nous avons définie

Désormais, le calcul lent n'est défini qu'à un seul endroit et nous n'exécutons ce code qu'aux endroits où nous avons besoin des résultats.

Cependant, nous avons réintroduit l'un des problèmes de l'encart 13-3 : nous continuons d'appeler la fermeture deux fois dans le premier bloc `if`, qui appellera le code lent à deux reprises et fera attendre l'utilisateur deux fois plus longtemps que nécessaire. Nous pourrions résoudre ce problème en créant une variable locale à ce bloc `if` pour conserver le résultat de l'appel à la fermeture, mais les fermetures nous ouvrent d'autres solutions. Commençons d'abord par expliquer pourquoi il n'y a pas d'annotation de type dans la définition des fermetures et des traits liés aux fermetures.

L'inférence de type et l'annotation des fermetures

Les fermetures ne nécessitent pas d'annoter le type des paramètres ou de la valeur de retour comme le font les fonctions `fn`. Les annotations de type sont nécessaires pour les fonctions car elles font partie d'une interface explicite exposée à leurs utilisateurs. Définir cette interface de manière rigide est nécessaire pour s'assurer que tout le monde s'accorde sur les types de valeurs qu'une fonction utilise et retourne. Mais les fermetures ne sont pas utilisées dans une interface exposée de cette façon : elles sont stockées dans des variables et utilisées sans les nommer ni les exposer aux utilisateurs de notre bibliothèque.

En outre, les fermetures sont généralement brèves et ne sont pertinentes que dans un contexte précis plutôt que pour des cas génériques. Dans ce contexte précis, le compilateur est capable de déduire le type des paramètres et le type de retour, tout comme il est capable d'inférer le type de la plupart des variables.

Demander aux développeurs d'annoter le type dans ces petites fonctions anonymes serait pénible et largement redondant avec l'information dont dispose déjà le compilateur.

Comme pour les variables, nous pouvons ajouter des annotations de type si nous voulons rendre explicite et clarifier le code au risque d'être plus verbeux que ce qui est strictement nécessaire. Annoter les types de la fermeture que nous avons définie dans l'encart 13-5 ressemblerait à l'encart 13-7.

Fichier : src/main.rs

```
let fermeture_lente = |nombre: u32| -> u32 {
    println!("calcul très lent ...");
    thread::sleep(Duration::from_secs(2));
    nombre
};
```

Encart 13-7 : ajout d'annotations de type optionnelles sur les paramètres et les valeurs de retour de la fermeture

La syntaxe des fermetures et des fonctions semble plus similaire avec les annotations de type. Ce qui suit est une comparaison verticale entre la syntaxe d'une définition d'une fonction qui ajoute 1 à son paramètre, et d'une fermeture qui a le même comportement. Nous avons ajouté des espaces pour aligner les parties pertinentes. Ceci met en évidence la similarité entre la syntaxe des fermetures et celle des fonctions, hormis l'utilisation des barres verticales et certaines syntaxes facultatives :

```
fn  ajouter_un_v1   (x: u32) -> u32 { x + 1 }
let ajouter_un_v2 = |x: u32| -> u32 { x + 1 };
let ajouter_un_v3 = |x|             { x + 1 };
let ajouter_un_v4 = |x|               x + 1  ;
```

La première ligne affiche la définition d'une fonction et la deuxième ligne une définition d'une fermeture entièrement annotée. La troisième ligne supprime les annotations de type de la définition de la fermeture, et la quatrième ligne supprime les accolades qui sont facultatives, parce que le corps d'une fermeture n'a qu'une seule expression. Ce sont toutes des définitions valides qui suivront le même comportement lorsqu'on les appellera. L'appel aux fermetures est nécessaire pour que `ajouter_un_v3` et `ajouter_un_v4` puissent être compilés car les types seront déduits en fonction de leur utilisation.

Les définitions des fermetures auront un type concret déduit pour chacun de leurs paramètres et pour leur valeur de retour. Par exemple, l'encart 13-8 montre la définition d'une petite fermeture qui renvoie simplement la valeur qu'elle reçoit comme paramètre. Cette fermeture n'est pas très utile sauf pour les besoins de cet exemple. Notez que nous n'avons pas ajouté d'annotation de type à la définition : si nous essayons alors d'appeler la fermeture deux fois, en utilisant une `String` comme argument la première fois et un `u32` la deuxième fois, nous obtiendrons une erreur :

Fichier : src/main.rs

```
    let fermeture_exemple = |x| x;

    let s = fermeture_exemple(String::from("hello"));
    let n = fermeture_exemple(5);
```

Encart 13-8 : tentative d'appeler une fermeture dont les types sont déduits avec deux types différents

Le compilateur nous renvoie l'erreur suivante :

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
error[E0308]: mismatched types
 --> src/main.rs:5:29
  |
5 |     let n = fermeture_exemple(5);
  |                               ^- help: try using a conversion method:
`.to_string()`
  |                               |
  |                               expected struct `String`, found integer

For more information about this error, try `rustc --explain E0308`.
error: could not compile `closure-example` due to previous error
```

La première fois que nous appelons `fermeture_exemple` avec une `String`, le compilateur déduit que le type de `x` et le type de retour de la fermeture sont de type `String`. Ces types sont ensuite verrouillés dans `fermeture_exemple`, et nous obtenons une erreur de type si nous essayons d'utiliser un type différent avec la même fermeture.

Stockage des fermetures avec des paramètres génériques et le trait `Fn`

Revenons à notre application de génération d'entraînements. Dans l'encart 13-6, notre code appelait toujours la fermeture lente plus de fois que nécessaire. Une option pour résoudre ce problème est de sauvegarder le résultat de la fermeture lente dans une variable pour une

future utilisation et d'utiliser la variable à chaque endroit où nous en avons besoin au lieu de rappeler la fermeture à nouveau. Cependant, cette méthode pourrait donner lieu à du code très répété.

Heureusement, une autre solution s'offre à nous. Nous pouvons créer une structure qui stockera la fermeture et la valeur qui en résulte. La structure n'exécutera la fermeture que si nous avons besoin de la valeur résultante, et elle mettra en cache la valeur résultante pour que le reste de notre code n'ait pas la responsabilité de sauvegarder et de réutiliser le résultat. Vous connaissez peut-être cette technique sous le nom de *mémoïsation* ou *d'évaluation paresseuse*.

Pour faire en sorte qu'une structure détienne une fermeture, il faut préciser le type de fermeture, car une définition de structure a besoin de connaître les types de chacun de ses champs. Chaque instance de fermeture a son propre type anonyme unique : cela signifie que même si deux fermetures ont la même signature, leurs types sont toujours considérés comme différents. Pour définir des structures, des énumérations ou des paramètres de fonction qui utilisent des fermetures, nous utilisons des génériques et des traits liés, comme nous l'avons vu au chapitre 10.

Les traits `Fn` sont fournis par la bibliothèque standard. Toutes les fermetures implémentent au moins un des traits suivants : `Fn`, `FnMut` ou `FnOnce`. Nous verrons la différence entre ces traits dans la section "Capturer l'environnement avec les fermetures" ; dans cet exemple, nous pouvons utiliser le trait `Fn`.

Nous ajoutons des types au trait lié `Fn` pour représenter les types de paramètres et les valeurs de retour que les fermetures doivent avoir pour correspondre à ce trait lié. Dans ce cas, notre fermeture a un paramètre de type `u32` et renvoie un `u32`, le trait lié que nous précisons est donc `Fn (u32) -> u32`.

L'encart 13-9 montre la définition de la structure `Cache` qui possède une fermeture et une valeur de résultat optionnelle :

Fichier : src/main.rs

```
struct Cache<T>
where
    T: Fn(u32) -> u32,
{
    calcul: T,
    valeur: Option<u32>,
}
```

Encart 13-9 : définition d'une structure `Cache` qui possède une fermeture dans `calcul` et un résultat optionnel dans `valeur`.

La structure `Cache` a un champ `calcul` du type générique `T`. Le trait lié `T` précise que c'est une fermeture en utilisant le trait `Fn`. Toute fermeture que l'on veut stocker dans le champ `calcul` doit avoir un paramètre `u32` (ce qui est précisé entre parenthèse après le `Fn`) et doit retourner un `u32` (ce qui est précisé après le `->`).

> Remarque : les fonctions peuvent aussi implémenter chacun de ces trois traits `Fn`. Si ce que nous voulons faire ne nécessite pas de capturer une valeur de l'environnement, nous pouvons utiliser une fonction plutôt qu'une fermeture lorsque nous avons besoin de quelque chose qui implémente un trait `Fn`.

Le champ `valeur` est de type `Option<u32>`. Avant d'exécuter la fermeture, `valeur` sera initialisée à `None`. Lorsque du code utilisant un `Cache` demande le *résultat* de la fermeture, le `Cache` exécutera la fermeture à ce moment-là et stockera le résultat dans une variante `Some` dans le champ `valeur`. Ensuite, si le code demande à nouveau le résultat de la fermeture, le `Cache` renverra le résultat contenu dans la variante `Some` au lieu d'exécuter à nouveau la fermeture.

La logique autour du champ `valeur` que nous venons de décrire est définie dans l'encart 13-10 :

Fichier : src/main.rs

```
impl<T> Cache<T>
where
    T: Fn(u32) -> u32
{
    fn new(calcul: T) -> Cache<T> {
        Cache {
            calcul,
            valeur: None,
        }
    }

    fn valeur(&mut self, arg: u32) -> u32 {
        match self.valeur {
            Some(v) => v,
            None => {
                let v = (self.calcul)(arg);
                self.valeur = Some(v);
                v
            },
        }
    }
}
```

Encart 13-10 : la logique de `Cache`

Nous voulons que `Cache` gère les valeurs des champs de structure plutôt que de laisser la possibilité au code appelant la possibilité de modifier directement les valeurs dans ces champs, donc nous faisons en sorte que ces champs soient privés.

La fonction `Cache::new` prend un paramètre générique `T`, que nous avons défini comme ayant le même trait lié que la structure `Cache`. Puis `Cache::new` renvoie une instance `Cache` qui contient la fermeture présente dans le champ `calcul` et une valeur `None` dans le champ `valeur`, car nous n'avons pas encore exécuté la fermeture.

Lorsque le code appelant veut le résultat de l'exécution de la fermeture, au lieu d'appeler directement la fermeture, il appellera la méthode `valeur`. Cette méthode vérifie si nous avons déjà une valeur dans un `Some` dans `self.valeur` ; et si c'est le cas, elle renvoie la valeur contenue dans le `Some` sans exécuter de nouveau la fermeture.

Si `self.valeur` est `None`, nous appelons la fermeture stockée dans `self.calcul`, et nous sauvegardons le résultat dans `self.valeur` pour une utilisation future, puis nous retournons la valeur.

L'encart 13-11 montre comment utiliser cette structure `Cache` dans la fonction `generer_exercices` de l'encart 13-6 :

Fichier : src/main.rs

```
fn generer_exercices(intensite: u32, nombre_aleatoire: u32) {
    let mut resultat_lent = Cache::new(|nombre| {
        println!("calcul très lent ...");
        thread::sleep(Duration::from_secs(2));
        nombre
    });

    if intensite < 25 {
        println!("Aujourd'hui, faire {} pompes !",
resultat_lent.valeur(intensite));
        println!("Ensuite, faire {} abdominaux !",
resultat_lent.valeur(intensite));
    } else {
        if nombre_aleatoire == 3 {
            println!("Faites une pause aujourd'hui ! Rappelez-vous de bien vous
hydrater !");
        } else {
            println!(
                "Aujourd'hui, courrez pendant {} minutes !",
                resultat_lent.valeur(intensite)
            );
        }
    }
}
```

Encart 13-11 : utilisation de `Cache` dans la fonction `generer_exercices` pour masquer la logique du cache.

Au lieu de sauvegarder la fermeture dans une variable directement, nous sauvegardons une nouvelle instance de `Cache` qui contient la fermeture. Ensuite, à chaque fois que nous voulons le résultat, nous appelons la méthode `valeur` sur cette instance de `Cache`. Nous pouvons appeler la méthode `valeur` autant de fois que nous le souhaitons, ou ne pas l'appeler du tout, et le calcul lent sera exécuté une fois au maximum.

Essayez d'exécuter ce programme avec la fonction `main` de l'encart 13-2. Modifiez les valeurs des variables `valeur_utilisateur_simule` et `nombre_aleatoire_simule` pour vérifier que dans tous les cas des différents blocs `if` et `else`, `calcul très lent ...` n'apparaît qu'une seule fois et seulement si nécessaire. Le `Cache` se charge de la logique nécessaire pour s'assurer que nous n'appelons pas le calcul lent plus que nous n'en avons besoin afin que `generer_exercices` puisse se concentrer sur la logique métier.

Limitations de l'implémentation de `Cache`

La mise en cache des valeurs est un comportement généralement utile que nous pourrions vouloir utiliser dans d'autres parties de notre code avec différentes fermetures. Cependant,

il y a deux problèmes avec l'implémentation actuelle de `Cache` qui rendraient difficile sa réutilisation dans des contextes différents.

Le premier problème est qu'une instance de `Cache` suppose qu'elle obtienne toujours la même valeur, indépendamment du paramètre `arg` de la méthode `valeur`. Autrement dit, ce test sur `Cache` échouera :

```
#[test]
fn appel_avec_differentes_valeurs() {
    let mut c = Cache::new(|a| a);

    let v1 = c.valeur(1);
    let v2 = c.valeur(2);

    assert_eq!(v2, 2);
}
```

Ce test crée une nouvelle instance de `Cache` avec une fermeture qui retourne la valeur qui lui est passée. Nous appelons la méthode `valeur` sur cette instance de `Cache` avec une valeur `arg` de 1 et ensuite une valeur `arg` de 2, et nous nous attendons à ce que l'appel à `valeur` avec la valeur `arg` de 2 retourne 2.

Exécutez ce test avec l'implémentation de `Cache` de l'encart 13-9 et de l'encart 13-10, et le test échouera sur le `assert_eq!` avec ce message :

```
$ cargo test
   Compiling cacher v0.1.0 (file:///projects/cacher)
    Finished test [unoptimized + debuginfo] target(s) in 0.72s
     Running unittests (target/debug/deps/cacher-074d7c200c000afa)

running 1 test
test tests::appel_avec_differentes_valeurs ... FAILED

failures:

---- tests::appel_avec_differentes_valeurs stdout ----
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `1`,
 right: `2`', src/lib.rs:43:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::appel_avec_differentes_valeurs

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Le problème est que la première fois que nous avons appelé `c.valeur` avec 1, l'instance `Cache` a sauvegardé `Some(1)` dans `self.valeur`. Par la suite, peu importe ce que nous passons à la méthode `valeur`, elle retournera toujours 1.

Essayez de modifier `Cache` pour tenir une table de hachage plutôt qu'une seule valeur. Les clés de la table de hachage seront les valeurs `arg` qui lui sont passées, et les valeurs de la table de hachage seront le résultat de l'appel à la fermeture avec cette clé. Plutôt que de regarder directement si `self.valeur` a une valeur `Some` ou une valeur `None`, la fonction `valeur` recherchera `arg` dans la table de hachage et retournera la valeur si elle est présente. S'il n'est pas présent, le `Cache` appellera la fermeture et sauvegardera la valeur résultante dans la table de hachage associée à sa clé `arg`.

Le second problème avec l'implémentation actuelle de `Cache` est qu'il n'accepte que les fermetures qui prennent un paramètre de type `u32` et renvoient un `u32`. Nous pourrions vouloir mettre en cache les résultats des fermetures qui prennent une slice d'une chaîne de caractères et renvoient des valeurs `usize`, par exemple. Pour corriger ce problème, essayez d'introduire des paramètres plus génériques pour augmenter la flexibilité de la fonctionnalité offerte par `Cache`.

Capturer l'environnement avec les fermetures

Dans l'exemple du générateur d'entraînement, nous n'avons utilisé les fermetures que comme des fonctions anonymes internes. Cependant, les fermetures ont une capacité supplémentaire que les fonctions n'ont pas : elles peuvent capturer leur environnement et accéder aux variables de la portée dans laquelle elles sont définies.

L'encart 13-12 montre un exemple de fermeture stockée dans la variable `egal_a_x` qui utilise la variable `x` de l'environnement environnant de la fermeture :

Fichier : src/main.rs

```
fn main() {
    let x = 4;

    let egal_a_x = |z| z == x;

    let y = 4;

    assert!(egal_a_x(y));
}
```

Encart 13-12 : exemple d'une fermeture qui se réfère à une variable présente dans la portée qui la contient.

Ici, même si `x` n'est pas un des paramètres de `egal_a_x`, la fermeture `egal_a_x` est autorisée à utiliser la variable `x` définie dans la même portée que celle où est définie `egal_a_x`.

Nous ne pouvons pas faire la même chose avec les fonctions ; si nous essayons avec l'exemple suivant, notre code ne se compilera pas :

Fichier : src/main.rs

```
fn main() {
    let x = 4;

    fn egal_a_x(z: i32) -> bool {
        z == x
    }

    let y = 4;

    assert!(egal_a_x(y));
}
```

Nous obtenons l'erreur suivante :

```
$ cargo run
   Compiling equal-to-x v0.1.0 (file:///projects/equal-to-x)
error[E0434]: can't capture dynamic environment in a fn item
 --> src/main.rs:5:14
  |
5 |         z == x
  |              ^
  |
  = help: use the `|| { ... }` closure form instead

For more information about this error, try `rustc --explain E0434`.
error: could not compile `equal-to-x` due to previous error
```

Le compilateur nous rappelle même que cela ne fonctionne qu'avec les fermetures !

Lorsqu'une fermeture capture une valeur de son environnement, elle utilise la mémoire pour stocker les valeurs à utiliser dans son corps. Cette utilisation de la mémoire a un coût supplémentaire que nous ne voulons pas payer dans les cas les plus courants où nous voulons exécuter du code qui ne capture pas son environnement. Comme les fonctions ne sont jamais autorisées à capturer leur environnement, la définition et l'utilisation des fonctions n'occasionneront jamais cette surcharge.

Les fermetures peuvent capturer les valeurs de leur environnement de trois façons différentes, qui correspondent directement aux trois façons dont une fonction peut prendre un paramètre : prendre possession, emprunter de manière immuable et emprunter de manière mutable. Ces moyens sont codés dans les trois traits `Fn` comme ceci :

- `FnOnce` consomme les variables qu'il capture à partir de sa portée, désignée sous le nom de *l'environnement* de la fermeture. Pour consommer les variables capturées, la fermeture doit prendre possession de ces variables et les déplacer dans la fermeture lorsqu'elle est définie. La partie `Once` du nom représente le fait que la fermeture ne puisse pas prendre prendre possession des mêmes variables plus d'une fois, donc elle ne peut être appelée qu'une seule fois.
- `FnMut` peut changer l'environnement car elle emprunte des valeurs de manière mutable.
- `Fn` emprunte des valeurs de l'environnement de manière immuable.

Lorsque nous créons une fermeture, Rust déduit quel trait utiliser en se basant sur la façon dont la fermeture utilise les valeurs de l'environnement. Toutes les fermetures implémentent `FnOne` car elles peuvent toute être appelées au moins une fois. Les fermetures qui ne déplacent pas les variables capturées implémentent également `FnMut`, et les fermetures qui n'ont pas besoin d'accès mutable aux variables capturées implémentent aussi `Fn`. Dans l'encart 13-12, la fermeture `egal_a_x` emprunte `x` immuablement (donc `egal_a_x` a le trait `Fn`) parce que le corps de la fermeture ne fait que lire la valeur de `x`.

Si nous voulons forcer la fermeture à prendre possession des valeurs qu'elle utilise dans l'environnement, nous pouvons utiliser le mot-clé `move` avant la liste des paramètres. Cette technique est très utile lorsque vous passez une fermeture à une nouvelle tâche pour déplacer les données afin qu'elles appartiennent à la nouvelle tâche.

> Remarque : les fermetures `move` peuvent toujours implémenter `Fn` ou `FnMut`, même si elles capturent les variables en les déplaçant. C'est possible car les traits implémentés par un type de fermeture sont déterminés par ce que font ces fermetures avec les valeurs déplacées et pas d'après la façon dont elles les capturent. Le mot-clé `move` ne définit que ce dernier aspect.

Nous verrons d'autres exemples de fermetures utilisant `move` au chapitre 16 lorsque nous parlerons de la concurrence. Pour l'instant, voici le code de l'encart 13-12 avec le mot-clé `move` ajouté à la définition de la fermeture et utilisant des vecteurs au lieu d'entiers, car les entiers peuvent être copiés plutôt que déplacés ; notez aussi que ce code ne compile pas encore.

Fichier : src/main.rs

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

    let egal_a_x = move |z| z == x;

    println!("On ne peut pas utiliser x ici : {:?}", x);

    let y = vec![1, 2, 3];

    assert!(egal_a_x(y));
}
```

Nous obtenons l'erreur suivante :

```
$ cargo run
   Compiling equal-to-x v0.1.0 (file:///projects/equal-to-x)
error[E0382]: borrow of moved value: `x`
 --> src/main.rs:6:40
  |
2 |     let x = vec![1, 2, 3];
  |         - move occurs because `x` has type `Vec<i32>`, which does not
implement the `Copy` trait
3 |
4 |     let egal_a_x = move |z| z == x;
  |                    --------      - variable moved due to use in closure
  |                    |
  |                    value moved into closure here
5 |
6 |     println!("On ne peut pas utiliser x ici : {:?}", x);
  |                                                      ^ value borrowed here
after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `equal-to-x` due to previous error
```

La valeur `x` est déplacée dans la fermeture lorsque la fermeture est définie, parce que nous avons ajouté le mot-clé `move`. La fermeture a alors la propriété de `x`, et `main` n'est plus autorisé à utiliser `x` dans l'instruction `println!`. Supprimer `println!` corrigera cet exemple.

La plupart du temps, lorsque vous renseignez l'un des traits liés `Fn`, vous pouvez commencer par `Fn` et le compilateur vous dira si vous avez besoin de `FnMut` ou `FnOnce` en fonction de ce qui se passe dans le corps de la fermeture.

Pour illustrer les situations où des fermetures qui capturent leur environnement sont utiles comme paramètres de fonction, passons à notre sujet suivant : les itérateurs.

Traiter une série d'éléments avec un itérateur

Les itérateurs vous permettent d'effectuer une tâche sur une séquence d'éléments à tour de rôle. Un *itérateur* est responsable de la logique d'itération sur chaque élément et de déterminer lorsque la séquence est terminée. Lorsque nous utilisons des itérateurs, nous n'avons pas besoin de ré-implémenter cette logique nous-mêmes.

En Rust, un itérateur est *une évaluation paresseuse*, ce qui signifie qu'il n'a aucun effet jusqu'à ce que nous appelions des méthodes qui consomment l'itérateur pour l'utiliser. Par exemple, le code dans l'encart 13-13 crée un itérateur sur les éléments du vecteur `v1` en appelant la méthode `iter` définie sur `Vec<T>`. Ce code en lui-même ne fait rien d'utile.

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();
```

Encart 13-13 : création d'un itérateur

Une fois que nous avons créé un itérateur, nous pouvons l'utiliser de diverses manières. Dans l'encart 3-4 du chapitre 3, nous avions utilisé des itérateurs avec des boucles `for` pour exécuter du code sur chaque élément, bien que nous ayons laissé de côté ce que l'appel à `iter` faisait jusqu'à présent.

L'exemple dans l'encart 13-14 sépare la création de l'itérateur de son utilisation dans la boucle `for`. L'itérateur est stocké dans la variable `v1_iter`, et aucune itération n'a lieu à ce moment-là. Lorsque la boucle `for` est appelée en utilisant l'itérateur `v1_iter`, chaque élément de l'itérateur est utilisé à chaque itération de la boucle, qui affiche chaque valeur.

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("On a : {}", val);
}
```

Encart 13-14 : utilisation d'un itérateur dans une boucle `for`

Dans les langages qui n'ont pas d'itérateurs fournis par leur bibliothèque standard, nous écririons probablement cette même fonctionnalité en démarrant une variable à l'indice 0, en utilisant cette variable comme indice sur le vecteur afin d'obtenir une valeur puis en incrémentant la valeur de cette variable dans une boucle jusqu'à ce qu'elle atteigne le nombre total d'éléments dans le vecteur.

Les itérateurs s'occupent de toute cette logique pour nous, réduisant le code redondant dans lequel nous pourrions potentiellement faire des erreurs. Les itérateurs nous donnent plus de flexibilité pour utiliser la même logique avec de nombreux types de séquences différentes, et pas seulement avec des structures de données avec lesquelles nous pouvons utiliser des indices, telles que les vecteurs. Voyons comment les itérateurs font cela.

Le trait `Iterator` et la méthode `next`

Tous les itérateurs implémentent un trait appelé `Iterator` qui est défini dans la bibliothèque standard. La définition du trait ressemble à ceci :

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // les méthodes avec des implémentations par défaut ont été exclues
}
```

Remarquez que cette définition utilise une nouvelle syntaxe : `type Item` et `Self::Item`, qui définissent un *type associé* à ce trait. Nous verrons ce que sont les types associés au chapitre 19. Pour l'instant, tout ce que vous devez savoir est que ce code dit que l'implémentation du trait `Iterator` nécessite que vous définissiez aussi un type `Item`, et ce type `Item` est utilisé dans le type de retour de la méthode `next`. En d'autres termes, le type `Item` sera le type retourné par l'itérateur.

Le trait `Iterator` exige la définition d'une seule méthode par les développeurs : la méthode `next`, qui retourne un élément de l'itérateur à la fois intégré dans un `Some`, et lorsque l'itération est terminée, il retourne `None`.

On peut appeler la méthode `next` directement sur les itérateurs ; l'encart 13-15 montre quelles valeurs sont retournées par des appels répétés à `next` sur l'itérateur créé à partir du vecteur.

Fichier : src/lib.rs

```
#[test]
fn demo_iterateur() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

Encart 13-15 : appel de la méthode `next` sur un itérateur

Remarquez que nous avons eu besoin de rendre mutable `v1_iter` : appeler la méthode `next` sur un iterator change son état interne qui garde en mémoire l'endroit où il en est dans la séquence. En d'autres termes, ce code *consomme*, ou utilise, l'itérateur. Chaque appel à `next` consomme un élément de l'itérateur. Nous n'avions pas eu besoin de rendre mutable `v1_iter` lorsque nous avions utilisé une boucle `for` parce que la boucle avait pris possession de `v1_iter` et l'avait rendu mutable en coulisses.

Notez également que les valeurs que nous obtenons des appels à `next` sont des références immuables aux valeurs dans le vecteur. La méthode `iter` produit un itérateur pour des références immuables. Si nous voulons créer un itérateur qui prend possession de `v1` et retourne les valeurs possédées, nous pouvons appeler `into_iter` au lieu de `iter`. De même, si nous voulons itérer sur des références mutables, nous pouvons appeler `iter_mut` au lieu de `iter`.

Les méthodes qui consomment un itérateur

Le trait `Iterator` a un certain nombre de méthodes différentes avec des implémentations par défaut que nous fournit la bibliothèque standard ; vous pouvez découvrir ces méthodes en regardant dans la documentation de l'API de la bibliothèque standard pour le trait `Iterator`. Certaines de ces méthodes appellent la méthode `next` dans leur définition, c'est pourquoi nous devons toujours implémenter la méthode `next` lors de l'implémentation du trait `Iterator`.

Les méthodes qui appellent `next` sont appelées des *adaptateurs de consommation*, parce que les appeler consomme l'itérateur. Un exemple est la méthode `sum`, qui prend possession de l'itérateur et itére sur ses éléments en appelant plusieurs fois `next`, consommant ainsi l'itérateur. A chaque étape de l'itération, il ajoute chaque élément à un total en cours et retourne le total une fois l'itération terminée. L'encart 13-16 a un test

illustrant une utilisation de la méthode `sum` :

Fichier : src/lib.rs

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

Encart 13-16 : appel de la méthode `sum` pour obtenir la somme de tous les éléments présents dans l'itérateur

Nous ne sommes pas autorisés à utiliser `v1_iter` après l'appel à `sum` car `sum` a pris possession de l'itérateur avec lequel nous l'appelons.

Méthodes qui produisent d'autres itérateurs

D'autres méthodes définies sur le trait `Iterator` , connues sous le nom *d'adaptateurs d'itération*, nous permettent de transformer un itérateur en un type d'itérateur différent. Nous pouvons enchaîner plusieurs appels à des adaptateurs d'itération pour effectuer des actions complexes de manière compréhensible. Mais comme les itérateurs sont *des évaluations paresseuses*, nous devons faire appel à l'une des méthodes d'adaptation de consommation pour obtenir les résultats des appels aux adaptateurs d'itération.

L'encart 13-17 montre un exemple d'appel à la méthode d'adaptation d'itération `map` , qui prend en paramètre une fermeture qui va s'exécuter sur chaque élément pour produire un nouvel itérateur. La fermeture crée ici un nouvel itérateur dans lequel chaque élément du vecteur a été incrémenté de 1. Cependant, ce code déclenche un avertissement :

Fichier : src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);
```

Encart 13-17 : appel de l'adaptateur d'itération `map` pour créer un nouvel itérateur

Voici l'avertissement que nous obtenons :

```
$ cargo run
   Compiling iterators v0.1.0 (file:///projects/iterators)
warning: unused `Map` that must be used
 --> src/main.rs:4:5
  |
4 |     v1.iter().map(|x| x + 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_must_use)]` on by default
  = note: iterators are lazy and do nothing unless consumed

warning: `iterators` (bin "iterators") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.47s
     Running `target/debug/iterators`
```

Le code dans l'encart 13-17 ne fait rien ; la fermeture que nous avons renseignée n'est jamais exécutée. L'avertissement nous rappelle pourquoi : les adaptateurs d'itération sont des *évaluations paresseuses*, c'est pourquoi nous devons consommer l'itérateur ici.

Pour corriger ceci et consommer l'itérateur, nous utiliserons la méthode `collect`, que vous avez utilisé avec `env::args` dans l'encart 12-1 du chapitre 12. Cette méthode consomme l'itérateur et collecte les valeurs résultantes dans un type de collection de données.

Dans l'encart 13-18, nous recueillons les résultats de l'itération sur l'itérateur qui sont retournés par l'appel à `map` sur un vecteur. Ce vecteur finira par contenir chaque élément du vecteur original incrémenté de 1.

Fichier : src/main.rs

```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

Encart 13-18 : appel de la méthode `map` pour créer un nouvel itérateur, puis appel de la méthode `collect` pour consommer le nouvel itérateur afin de créer un vecteur

Comme `map` prend en paramètre une fermeture, nous pouvons renseigner n'importe quelle opération que nous souhaitons exécuter sur chaque élément. C'est un bon exemple de la façon dont les fermetures nous permettent de personnaliser certains comportements tout en réutilisant le comportement d'itération fourni par le trait `Iterator`.

Utilisation de fermetures capturant leur environnement

Maintenant que nous avons présenté les itérateurs, nous pouvons illustrer une utilisation commune des fermetures qui capturent leur environnement en utilisant l'adaptateur d'itération `filter`. La méthode `filter` appelée sur un itérateur prend en paramètre une fermeture qui s'exécute sur chaque élément de l'itérateur et retourne un booléen pour chacun. Si la fermeture retourne `true`, la valeur sera incluse dans l'itérateur produit par `filter`. Si la fermeture retourne `false`, la valeur ne sera pas incluse dans l'itérateur résultant.

Dans l'encart 13-19, nous utilisons `filter` avec une fermeture qui capture la variable `pointure_chaussure` de son environnement pour itérer sur une collection d'instances de la structure `Chaussure`. Il ne retournera que les chaussures avec la pointure demandée.

Fichier : src/lib.rs

```rust
#[derive(PartialEq, Debug)]
struct Chaussure {
    pointure: u32,
    style: String,
}

fn chaussures_a_la_pointure(chaussures: Vec<Chaussure>, pointure_chaussure: u32)
-> Vec<Chaussure> {
    chaussures.into_iter()
              .filter(|s| s.pointure == pointure_chaussure)
              .collect()
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn filtres_par_pointure() {
        let chaussures = vec![
            Chaussure {
                pointure: 10,
                style: String::from("baskets"),
            },
            Chaussure {
                pointure: 13,
                style: String::from("sandale"),
            },
            Chaussure {
                pointure: 10,
                style: String::from("bottes"),
            },
        ];

        let a_ma_pointure = chaussures_a_la_pointure(chaussures, 10);

        assert_eq!(
            a_ma_pointure,
            vec![
                Chaussure {
                    pointure: 10,
                    style: String::from("baskets")
                },
                Chaussure {
                    pointure: 10,
                    style: String::from("bottes")
                },
            ]
        );
    }
}
```

Encart 13-19 : utilisation de la méthode `filter` avec une fermeture capturant `pointure_chaussure`

La fonction `chaussures_a_la_pointure` prend possession d'un vecteur de chaussures et d'une pointure comme paramètres. Il retourne un vecteur contenant uniquement des chaussures de la pointure demandée.

Dans le corps de `chaussures_a_la_pointure`, nous appelons `into_iter` pour créer un itérateur qui prend possession du vecteur. Ensuite, nous appelons `filter` pour adapter cet itérateur dans un nouvel itérateur qui ne contient que les éléments pour lesquels la fermeture retourne `true`.

La fermeture capture le paramètre `pointure_chaussure` de l'environnement et compare la valeur avec la pointure de chaque chaussure, en ne gardant que les chaussures de la pointure spécifiée. Enfin, l'appel à `collect` retourne un vecteur qui regroupe les valeurs renvoyées par l'itérateur.

Le test confirme que lorsque nous appelons `chaussures_a_la_pointure`, nous n'obtenons que des chaussures qui ont la même pointure que la valeur que nous avons demandée.

Créer nos propres itérateurs avec le trait `Iterator`

Nous avons vu que nous pouvons créer un itérateur en appelant `iter`, `into_iter` ou `iter_mut` sur un vecteur. Nous pouvons créer des itérateurs à partir d'autres types de collections de la bibliothèque standard, comme les tables de hachage. Nous pouvons aussi créer des itérateurs qui font tout ce que nous voulons en implémentant le trait `Iterator` sur nos propres types. Comme nous l'avons mentionné précédemment, la seule méthode pour laquelle nous devons fournir une définition est la méthode `next`. Une fois que nous avons fait cela, nous pouvons utiliser toutes les autres méthodes qui ont des implémentations par défaut fournies par le trait `Iterator` !

Pour preuve, créons un itérateur qui ne comptera que de 1 à 5. D'abord, nous allons créer une structure contenant quelques valeurs. Ensuite nous transformerons cette structure en itérateur en implémentant le trait `Iterator` et nous utiliserons les valeurs de cette implémentation.

L'encart 13-20 montre la définition de la structure `Compteur` et une fonction associée `new` pour créer des instances de `Compteur` :

Fichier : src/lib.rs

```
struct Compteur {
    compteur: u32,
}

impl Compteur {
    fn new() -> Compteur {
        Compteur { compteur: 0 }
    }
}
```

Encart 13-20 : définition de la structure `Compteur` et d'une fonction `new` qui crée des instances de `Compteur` avec une valeur initiale de 0 pour le champ `compteur` .

La structure `Compteur` a un champ `compteur` . Ce champ contient une valeur `u32` qui gardera la trace de l'endroit où nous sommes dans le processus d'itération de 1 à 5. Le champ `compteur` est privé car nous voulons que ce soit l'implémentation de `Compteur` qui gère sa valeur. La fonction `new` impose de toujours démarrer de nouvelles instances avec une valeur de 0 pour le champ `compteur` .

Ensuite, nous allons implémenter le trait `Iterator` sur notre type `Compteur` en définissant le corps de la méthode `next` pour préciser ce que nous voulons qu'il se passe quand cet itérateur est utilisé, comme dans l'encart 13-21 :

Fichier : src/lib.rs

```
impl Iterator for Compteur {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.compteur < 5 {
            self.compteur += 1;
            Some(self.compteur)
        } else {
            None
        }
    }
}
```

Encart 13-21 : implémentation du trait `Iterator` sur notre structure `Compteur`

Nous avons défini le type associé `Item` pour notre itérateur à `u32` , ce qui signifie que l'itérateur renverra des valeurs `u32` . Encore une fois, ne vous préoccupez pas des types associés, nous les aborderons au chapitre 19.

Nous voulons que notre itérateur ajoute 1 à l'état courant, donc nous avons initialisé `compteur` à 0 pour qu'il retourne 1 lors du premier appel à `next` . Si la valeur de `compteur`

est strictement inférieure à 5, `next` va incrémenter `compteur` puis va retourner la valeur courante intégrée dans un `Some`. Une fois que `compteur` vaudra 5, notre itérateur va arrêter d'incrémenter `compteur` et retournera toujours `None`.

Utiliser la méthode `next` de notre Itérateur `Compteur`

Une fois que nous avons implémenté le trait `Iterator`, nous avons un itérateur ! L'encart 13-22 montre un test démontrant que nous pouvons utiliser la fonctionnalité d'itération de notre structure `Compteur` en appelant directement la méthode `next`, comme nous l'avons fait avec l'itérateur créé à partir d'un vecteur dans l'encart 13-15.

Fichier : src/lib.rs

```
#[test]
fn appel_direct_a_next() {
    let mut compteur = Compteur::new();

    assert_eq!(compteur.next(), Some(1));
    assert_eq!(compteur.next(), Some(2));
    assert_eq!(compteur.next(), Some(3));
    assert_eq!(compteur.next(), Some(4));
    assert_eq!(compteur.next(), Some(5));
    assert_eq!(compteur.next(), None);
}
```

Encart 13-22 : test de l'implémentation de la méthode `next`

Ce test crée une nouvelle instance de `Compteur` dans la variable `compteur` et appelle ensuite `next` à plusieurs reprises, en vérifiant que nous avons implémenté le comportement que nous voulions que cet itérateur suive : renvoyer les valeurs de 1 à 5.

Utiliser d'autres méthodes du trait `Iterator`

Maintenant que nous avons implémenté le trait `Iterator` en définissant la méthode `next`, nous pouvons maintenant utiliser les implémentations par défaut de n'importe quelle méthode du trait `Iterator` telles que définies dans la bibliothèque standard, car elles utilisent toutes la méthode `next`.

Par exemple, si pour une raison quelconque nous voulions prendre les valeurs produites par une instance de `Compteur`, les coupler avec des valeurs produites par une autre instance de `Compteur` après avoir sauté la première valeur, multiplier chaque paire ensemble, ne garder que les résultats qui sont divisibles par 3 et additionner toutes les valeurs résultantes

ensemble, nous pourrions le faire, comme le montre le test dans l'encart 13-23 :

Fichier : src/lib.rs

```
#[test]
fn utilisation_des_autres_methodes_du_trait_iterator() {
    let somme: u32 = Compteur::new()
        .zip(Compteur::new().skip(1))
        .map(|(a, b)| a * b)
        .filter(|x| x % 3 == 0)
        .sum();
    assert_eq!(18, somme);
}
```

Encart 13-23 : utilisation d'une gamme de méthodes du trait `Iterator` sur notre itérateur `Compteur`

Notez que `zip` ne produit que quatre paires ; la cinquième paire théorique `(5, None)` n'est jamais produite car `zip` retourne `None` lorsque l'un de ses itérateurs d'entrée retourne `None` .

Tous ces appels de méthode sont possibles car nous avons renseigné comment la méthode `next` fonctionne et la bibliothèque standard fournit des implémentations par défaut pour les autres méthodes qui appellent `next` .

Amélioration de notre projet d'entrée/sortie

Grâce à ces nouvelles connaissances sur les itérateurs, nous pouvons améliorer le projet d'entrée/sortie du chapitre 12 en utilisant des itérateurs pour rendre certains endroits du code plus clairs et plus concis. Voyons comment les itérateurs peuvent améliorer notre implémentation de la fonction `Config::new` et de la fonction `rechercher` .

Supprimer l'appel à `clone` à l'aide d'un itérateur

Dans l'encart 12-6, nous avions ajouté du code qui prenait une *slice* de `String` et qui créait une instance de la structure `Config` en utilisant les indices de la *slice* et en clonant les valeurs, permettant ainsi à la structure `Config` de posséder ces valeurs. Dans l'encart 13-24, nous avons reproduit l'implémentation de la fonction `Config::new` telle qu'elle était dans l'encart 12-23 à la fin du chapitre 12 :

Fichier : src/lib.rs

```
impl Config {
    pub fn new(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("il n'y a pas assez d'arguments");
        }

        let recherche = args[1].clone();
        let nom_fichier = args[2].clone();

        let sensible_casse = env::var("MINIGREP_INSENSIBLE_CASSE").is_err();

        Ok(Config {
            recherche,
            nom_fichier,
            sensible_casse,
        })
    }
}
```

Encart 13-24 : reproduction de la fonction `Config::new` de la fin du chapitre 12

À ce moment-là, nous avions dit de ne pas s'inquiéter des appels inefficaces à `clone` parce que nous les supprimerions à l'avenir. Et bien, ce moment est venu !

Nous avions besoin de `clone` ici parce que nous avons une slice d'éléments `String` dans le paramètre `args` , mais la fonction `new` ne possède pas `args` . Pour renvoyer la propriété d'une instance de `Config` , nous avons dû cloner les valeurs des champs `recherche` et

`nom_fichier` de `Config` afin que cette instance de `Config` puisse prendre possession de ces valeurs.

Avec nos nouvelles connaissances sur les itérateurs, nous pouvons changer la fonction `new` pour prendre possession d'un itérateur passé en argument au lieu d'emprunter une *slice*. Nous utiliserons les fonctionnalités des itérateurs à la place du code qui vérifie la taille de la slice et qui utilise les indices des éléments précis. Cela clarifiera ce que la fonction `Config::new` fait car c'est l'itérateur qui accédera aux valeurs.

Une fois que `Config::new` prend possession de l'itérateur et cesse d'utiliser les opérations avec les indices et d'emprunter les données, nous pouvons déplacer les valeurs `String` de l'iterator dans `Config` plutôt que de faire appel à `clone` et de créer par conséquent de nouvelles allocations.

Utiliser directement l'itérateur retourné

Ouvrez le fichier *src/main.rs* de votre projet d'entrée/sortie, qui devrait ressembler à ceci :

Fichier : src/main.rs

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

    let config = Config::new(&args).unwrap_or_else(|err| {
        eprintln!("Problème rencontré lors de l'interprétation des arguments :
{}", err);
        process::exit(1);
    });

    // -- partie masquée ici --
}
```

Nous allons changer le début de la fonction `main` que nous avions dans l'encart 12-24 pour le code dans l'encart 13-25. Ceci ne compilera pas encore jusqu'à ce que nous mettions également à jour `Config::new` .

Fichier : src/main.rs

```
fn main() {
    let config = Config::new(env::args()).unwrap_or_else(|err| {
        eprintln!("Problème rencontré lors de l'interprétation des arguments :
{}", err);
        process::exit(1);
    });

    // -- partie masquée ici --
}
```

Encart 13-25 : on passe directement la valeur de retour de `env::args` à `Config::new`.

La fonction `env::args` retourne un itérateur ! Plutôt que de collecter les valeurs de l'itérateur dans un vecteur et de passer ensuite une *slice* à `Config::new`, nous passons maintenant la possession de l'itérateur de `env::args` directement à `Config::new`.

Ensuite, nous devons mettre à jour la définition de `Config::new`. Dans le fichier *src/lib.rs* de votre projet d'entrée/sortie, modifions la signature de `Config::new` pour qu'elle ressemble à l'encart 13-26. Ceci ne compilera pas encore car nous devons mettre à jour le corps de la fonction.

Fichier : src/lib.rs

```
impl Config {
    pub fn new(mut args: env::Args) -> Result<Config, &'static str> {
        // -- partie masquée ici --
```

Encart 13-26 : mise à jour de la signature de `Config::new` pour recevoir un itérateur

La documentation de la bibliothèque standard de la fonction `env::args` indique que le type de l'itérateur qu'elle renvoie est `std::env::Args`. Nous avons mis à jour la signature de la fonction `Config::new` pour que le paramètre `args` ait le type `std::env::Args` au lieu de `&[String]`. Etant donné que nous prenons possession de `args` et que nous allons muter `args` en itérant dessus, nous pouvons ajouter le mot-clé `mut` dans la spécification du paramètre `args` pour le rendre mutable.

Utilisation des méthodes du trait `Iterator` au lieu des indices

Corrigeons ensuite le corps de `Config::new`. La documentation de la bibliothèque standard explique aussi que `std::env::Args` implémente le trait `Iterator`, donc nous savons que nous pouvons appeler la méthode `next` dessus ! L'encart 13-27 met à jour le code de l'encart 12-23 afin d'utiliser la méthode `next` :

Fichier : src/lib.rs

```
impl Config {
    pub fn new(mut args: env::Args) -> Result<Config, &'static str> {
        args.next();

        let recherche = match args.next() {
            Some(arg) => arg,
            None => return Err("nous n'avons pas de chaîne de caractères"),
        };

        let nom_fichier = match args.next() {
            Some(arg) => arg,
            None => return Err("nous n'avons pas de nom de fichier"),
        };

        let sensible_casse = env::var("MINIGREP_INSENSIBLE_CASSE").is_err();

        Ok(Config {
            recherche,
            nom_fichier,
            sensible_casse,
        })
    }
}
```

Encart 13-27 : changement du corps de `Config::new` afin d'utiliser les méthodes d'itération

Rappelez-vous que la première valeur de ce qui est retourné par `env::args` est le nom du programme. Nous voulons ignorer cette valeur et passer à la suivante, donc d'abord nous appelons une fois `next` et nous ne faisons rien avec sa valeur de retour. Ensuite, nous appelons `next` pour obtenir la valeur que nous voulons mettre dans le champ `recherche` de `Config`. Si `next` renvoie un `Some`, nous utilisons un `match` pour extraire sa valeur. S'il retourne `None`, cela signifie que pas assez d'arguments ont été fournis, si bien que nous quittons aussitôt la fonction en retournant une valeur `Err`. Nous procédons de même pour la valeur `nom_fichier`.

Rendre le code plus clair avec des adaptateurs d'itération

Nous pouvons également tirer parti des itérateurs dans la fonction `rechercher` de notre projet d'entrée/sortie, qui est reproduite ici dans l'encart 13-28, telles qu'elle était dans l'encart 12-19 à la fin du chapitre 12 :

Fichier : src/lib.rs

```
pub fn rechercher<'a>(recherche: &str, contenu: &'a str) -> Vec<&'a str> {
    let mut resultats = Vec::new();

    for ligne in contenu.lines() {
        if ligne.contains(recherche) {
            resultats.push(ligne);
        }
    }

    resultats
}
```

Encart 13-28 : La mise en oeuvre de la fonction `rechercher` de l'encart 12-19

Nous pouvons écrire ce code de façon plus concise en utilisant des méthodes des adaptateurs d'itération. Ce faisant, nous évitons ainsi d'avoir le vecteur mutable `resultats`. Le style de programmation fonctionnelle préfère minimiser la quantité d'états modifiables pour rendre le code plus clair. Supprimer l'état mutable pourrait nous aider à faire une amélioration future afin que la recherche se fasse en parallèle, car nous n'aurions pas à gérer l'accès concurrent au vecteur `resultats`. L'encart 13-29 montre ce changement :

Fichier : src/lib.rs

```
pub fn rechercher<'a>(recherche: &str, contenu: &'a str) -> Vec<&'a str> {
    contenu
        .lines()
        .filter(|ligne| ligne.contains(recherche))
        .collect()
}
```

Encart 13-29 : utilisation des méthodes des adaptateurs d'itération dans l'implémentation de la fonction `rechercher`

Souvenez-vous que le but de la fonction `rechercher` est de renvoyer toutes les lignes dans `contenu` qui contiennent `recherche`. Comme dans l'exemple de `filter` dans l'encart 13-19, nous pouvons utiliser l'adaptateur `filter` pour garder uniquement les lignes pour lesquelles `ligne.contains(recherche)` renvoie `true`. Nous collectons ensuite les lignes correspondantes dans un autre vecteur avec `collect`. C'est bien plus simple ! N'hésitez pas à faire le même changement pour utiliser les méthodes d'itération dans la fonction `rechercher_insensible_casse`.

Logiquement la question suivante est de savoir quel style utiliser dans votre propre code et pourquoi : l'implémentation originale de l'encart 13-28 ou la version utilisant l'itérateur dans l'encart 13-29. La plupart des développeurs Rust préfèrent utiliser le style avec l'itérateur. C'est un peu plus difficile à comprendre au début, mais une fois que vous avez compris les

différents adaptateurs d'itération et ce qu'ils font, les itérateurs peuvent devenir plus faciles à comprendre. Au lieu de jongler avec différentes boucles et de construire de nouveaux vecteurs, ce code se concentre sur l'objectif de haut niveau de la boucle. Cette abstraction permet d'éliminer une partie du code trivial, de sorte qu'il soit plus facile de dégager les concepts propres à ce code, comme le filtrage de chaque élément de l'itérateur qui est appliqué.

Mais ces deux implémentations sont-elles réellement équivalentes ? L'hypothèse intuitive pourrait être que la boucle de plus bas niveau sera plus rapide. Intéressons nous donc maintenant à leurs performances.

Comparaison des performances : les boucles et les itérateurs

Pour déterminer s'il faut utiliser des boucles ou des itérateurs, nous devons savoir quelle implémentation est la plus rapide : la version de la fonction `rechercher` avec une boucle `for` explicite, ou la version avec des itérateurs.

Nous avons lancé un benchmark en chargeant tout le contenu de *The Adventures of Sherlock Holmes* de Sir Arthur Conan Doyle dans une `String` et en cherchant le mot "the" dans le contenu. Voici les résultats du benchmark sur la version de `rechercher` avec une boucle `for` et avec un itérateur :

```
test benchmark_rechercher_for  ... bench:  19,620,300 ns/iter (+/- 915,700)
test benchmark_rechercher_iter ... bench:  19,234,900 ns/iter (+/- 657,200)
```

La version avec l'itérateur était un peu plus rapide ! Nous n'expliquerons pas le code du benchmark ici, car il ne s'agit pas de prouver que les deux versions sont équivalentes, mais d'avoir une idée générale de la différence de performances entre les deux.

Pour un benchmark plus complet, nous vous conseillons d'utiliser des textes de différentes tailles pour `contenu`, des mots différents et de différentes longueurs pour `recherche`, ainsi que tout autre type de variation que vous pourriez trouver. Le point important est le suivant : les itérateurs, bien qu'il s'agisse d'une abstraction de haut niveau, sont compilés à peu près comme si vous aviez écrit vous-même le code un niveau plus bas. Les itérateurs sont l'une des abstractions à *coût zéro* de Rust, c'est-à-dire que l'utilisation de l'abstraction n'impose aucun surcoût lors de l'exécution. C'est la même notion que celle que Bjarne Stroustrup, le concepteur et développeur original de C++, définit en tant que *coût zéro* dans “Foundations of C++” (2012) :

> En général, les implémentations de C++ obéissent au principe du coût zéro : ce que vous n'utilisez pas, vous ne le payez pas. Et plus encore : ce que vous utilisez, vous ne pourrez pas le coder mieux à la main.

Comme autre exemple, le code suivant est tiré d'un décodeur audio. L'algorithme de décodage utilise l'opération mathématique de prédiction linéaire pour estimer les valeurs futures à partir d'une fonction linéaire des échantillons précédents. Ce code utilise une chaîne d'itérateurs pour faire quelques calculs sur trois variables dans la portée : une slice de données `tampon`, un tableau de 12 `coefficients` et une valeur de décalage des données dans `decalage`. Nous avons déclaré les variables dans cet exemple, mais nous ne leur avons pas donné de valeurs ; bien que ce code n'ait pas beaucoup de signification en

dehors de son contexte, c'est toutefois un exemple concis et concret de la façon dont Rust traduit des idées de haut niveau en code de plus bas niveau.

```
let tampon: &mut [i32];
let coefficients: [i64; 12];
let decalage: i16;

for i in 12..tampon.len() {
    let prediction = coefficients.iter()
                                 .zip(&tampon[i - 12..i])
                                 .map(|(&c, &s)| c * s as i64)
                                 .sum::<i64>() >> decalage;
    let delta = tampon[i];
    tampon[i] = prediction as i32 + delta;
}
```

Pour calculer la valeur de `prediction`, ce code itère sur chacune des 12 valeurs dans `coefficients` et utilise la méthode `zip` pour appairer la valeur de coefficient avec les 12 valeurs précédentes, présentes dans `tampon`. Ensuite, pour chaque paire, nous multiplions les valeurs ensemble, nous additionnons tous les résultats et nous décalons les bits de l'addition de la valeur de `decalage` vers la droite.

Les calculs dans des applications comme les décodeurs audio donnent souvent la priorité aux performances. Ici, nous créons un itérateur à l'aide de deux adaptateurs, puis nous en consommons la valeur. A quel code d'assemblage ce code Rust ressemblera-t-il une fois compilé ? Et bien, à l'heure où nous écrivons ces lignes, il donne le même code assembleur que vous écririez à la main. Il n'y a pas du tout de boucle correspondant à l'itération sur les valeurs dans `coefficients` : Rust sait qu'il y a 12 itérations, donc il “déroule” la boucle. Le *déroulage* est une optimisation qui supprime la surcharge du code de contrôle de boucle et génère à la place du code répété pour chaque itération de la boucle.

Tous les coefficients sont stockés dans des registres, ce qui signifie qu'il est très rapide d'accéder à ces valeurs. Il n'y a pas de vérification des bornes sur les accès au tableau à l'exécution. Toutes ces optimisations que Rust est capable d'appliquer rendent le code produit extrêmement efficace. Maintenant que vous savez cela, vous pouvez utiliser des itérateurs et des fermetures sans crainte ! Ils font en sorte que le code soit de haut niveau, mais n'entraînent pas de pénalité de performance à l'exécution.

Résumé

Les fermetures et les itérateurs sont des fonctionnalités de Rust inspirées par des idées des langages de programmation fonctionnels. Ils contribuent à la capacité de Rust d'exprimer

clairement des idées de haut niveau avec des performances dignes d'un langage de bas niveau. Les implémentations des fermetures et des itérateurs sont telles que les performances à l'exécution n'en sont pas affectées. Cela fait partie de l'objectif de Rust de s'efforcer à fournir des abstractions à coût zéro.

Maintenant que nous avons amélioré l'expressivité de notre projet d'entrée/sortie, regardons d'autres fonctionnalités fournies par `cargo` qui nous aideront à partager notre projet avec le monde entier.

En savoir plus sur cargo et crates.io

Précédemment, nous avons utilisé les fonctionnalités les plus basiques de cargo pour compiler, exécuter et tester notre code, mais il peut faire bien plus. Dans ce chapitre, nous allons voir d'autres fonctionnalités avancées pour vous apprendre à faire ceci :

- Personnaliser votre compilation grâce aux profils de publication
- Publier des bibliothèques sur crates.io
- Organiser des gros projets avec les espaces de travail
- Installer des binaires à partir de crates.io
- Améliorer cargo en utilisant des commandes personnalisées

Cargo peut faire encore plus de choses que ce que nous allons voir dans ce chapitre, donc pour une explication plus complète vous avez à votre disposition sa documentation.

Personnaliser les compilations avec les profils de publication

Dans Rust, les *profils de publication* sont des profils prédéfinis et personnalisables avec différentes configurations qui permettent au développeur d'avoir plus de contrôle sur différentes options de compilation du code. Chaque profil est configuré indépendamment des autres.

Cargo a deux profils principaux : le profil `dev` que cargo utilise lorsque vous lancez `cargo build` et le profil `release` (NdT : publication) que cargo utilise lorsque vous lancez `cargo build --release`. Le profil `dev` est défini avec de bons réglages par défaut pour le développement, et le profil `release` a de bons réglages par défaut de compilation pour la publication.

Ces noms de profils vous rappellent peut-être quelque chose sur la sortie standard de vos compilations :

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
$ cargo build --release
    Finished release [optimized] target(s) in 0.0s
```

Les profils `dev` et `release` sont mentionnés dans cette sortie de compilation, pour indiquer les différents profils qu'utilise le compilateur.

Cargo a des réglages par défaut pour chacun des profils qui s'appliquent lorsqu'il n'y a pas de section `[profile.*]` dans le fichier *Cargo.toml* du projet. En ajoutant les sections `[profile.*]` pour chaque profil que vous souhaitez personnaliser, vous pouvez remplacer n'importe quel paramètre par défaut. Par exemple, voici les valeurs par défaut pour le paramètre `opt-level` des profils `dev` et `release` :

Fichier : Cargo.toml

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

Le paramètre `opt-level` contrôle le nombre d'optimisations que Rust va appliquer à votre code, sur une échelle allant de 0 à 3. L'application d'un niveau plus haut d'optimisation signifie un allongement de la durée de compilation, donc si vous êtes en train de développer et que vous compilez souvent votre code, vous préférerez certainement avoir une

compilation rapide même si le code qui en résulte s'exécute plus lentement. C'est la raison pour laquelle la valeur par défaut de `opt-level` pour `dev` est à `0`. Lorsque vous serez prêt à publier votre code, il sera préférable de passer un peu plus de temps à le compiler. Vous ne compilerez en mode publication (NdT : release) qu'une seule fois, mais vous exécuterez le programme compilé plusieurs fois, donc le mode publication opte pour un temps de compilation plus long afin que le code s'exécute plus rapidement. C'est pourquoi le paramètre `opt-level` par défaut pour le profil `release` est à `3`.

Vous pouvez remplacer n'importe quel paramètre par défaut en ajoutant une valeur différente dans *Cargo.toml*. Par exemple, si nous voulons utiliser le niveau 1 d'optimisation dans le profil de développement, nous pouvons ajouter ces deux lignes à notre fichier *Cargo.toml* :

Fichier : Cargo.toml

```
[profile.dev]
opt-level = 1
```

Ce code remplace le paramètre par défaut à `0`. Maintenant, lorsque nous lançons `cargo build`, cargo va utiliser les réglages par défaut du profil `dev` ainsi que notre valeur personnalisée de `opt-level`. Comme nous avons réglé `opt-level` à `1`, Cargo va appliquer plus d'optimisation que par défaut, mais pas autant que dans une compilation de publication.

Pour la liste complète des options de configuration et leurs valeurs par défaut pour chaque profil, référez-vous à la documentation de cargo.

Publier une crate sur crates.io

Nous avons déjà utilisé des paquets provenant de crates.io comme dépendance de notre projet, mais vous pouvez aussi partager votre code avec d'autres personnes en publiant vos propres paquets. Le registre des crates disponible sur crates.io distribue le code source de vos paquets, donc il héberge principalement du code qui est open source.

Rust et cargo ont des fonctionnalités qui aident les développeurs à trouver et utiliser les paquets que vous publiez. Nous allons voir certaines de ces fonctionnalités puis nous allons expliquer comment publier un paquet.

Créer des commentaires de documentation utiles

Documenter correctement vos paquets aidera les autres utilisateurs à savoir comment et quand les utiliser, donc ça vaut la peine de consacrer du temps à la rédaction de la documentation. Dans le chapitre 3, nous avons vu comment commenter du code Rust en utilisant deux barres obliques `//`. Rust a aussi un type particulier de commentaire pour la documentation, aussi connu sous le nom de *commentaire de documentation*, qui va générer de la documentation en HTML. Le HTML affiche le contenu des commentaires de documentation pour les éléments public de votre API à destination des développeurs qui s'intéressent à la manière *d'utiliser* votre crate et non pas à la manière dont elle est *implémentée*.

Les commentaires de documentation utilisent trois barres obliques `///` au lieu de deux et prend en charge la notation Markdown pour mettre en forme le texte. Placez les commentaires de documentation juste avant l'élément qu'ils documentent. L'encart 14-1 montre des commentaires de documentation pour une fonction `ajouter_un` dans une crate nommée `ma_crate`.

Fichier : src/lib.rs

```
/// Ajoute 1 au nombre donné.
///
/// # Exemples
///
/// ```
/// let argument = 5;
/// let reponse = ma_crate::ajouter_un(argument);
///
/// assert_eq!(6, reponse);
/// ```
pub fn ajouter_un(x: i32) -> i32 {
    x + 1
}
```

Encart 14-1 : un commentaire de documentation pour une fonction

Ici nous avons écrit une description de ce que fait la fonction `ajouter_un`, débuté une section avec le titre `Exemples` puis fourni du code qui montre comment utiliser la fonction `ajouter_un`. Nous pouvons générer la documentation HTML à partir de ces commentaires de documentation en lançant `cargo doc`. Cette commande lance l'outil `rustdoc` qui est distribué avec Rust et place la documentation HTML générée dans le dossier *target/doc*.

Pour plus de facilité, lancer `cargo doc --open` va générer le HTML pour la documentation de votre crate courante (ainsi que la documentation pour toutes les dépendances de la crate) et ouvrir le résultat dans un navigateur web. Rendez-vous à la fonction `ajouter_one` et vous découvrirez comment le texte dans les commentaires de la documentation a été interprété, ce qui devrait ressembler à l'illustration 14-1 :

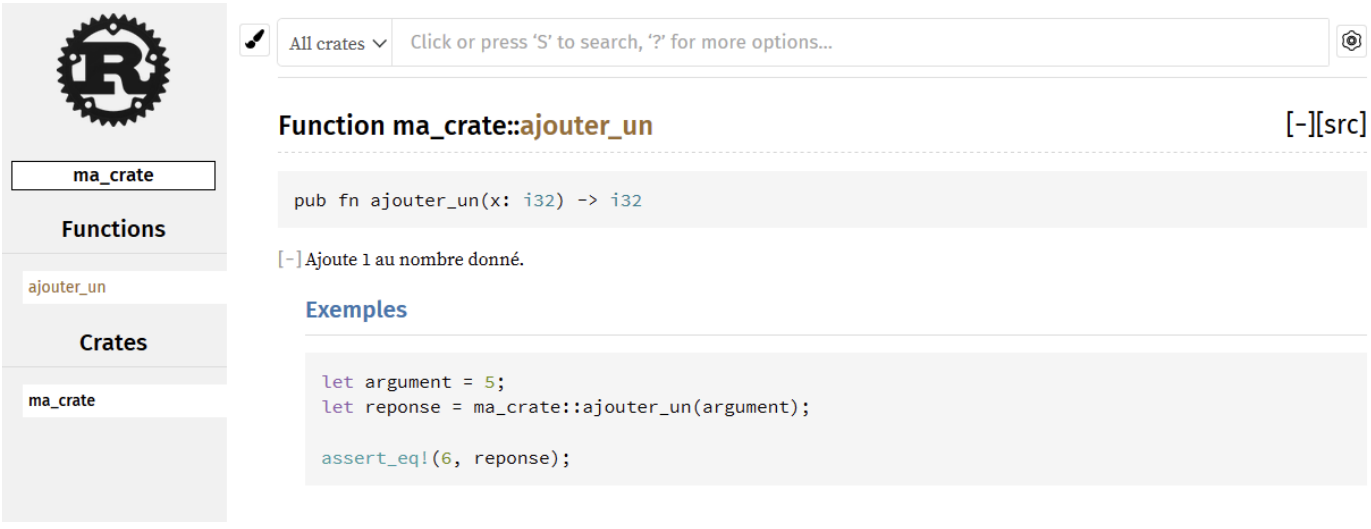


Illustration 14-1 : documentation HTML pour la fonction `ajouter_un`

Les sections utilisées fréquemment

Nous avons utilisé le titre en Markdown `# Exemples` dans l'encart 14-1 afin de créer une section dans le HTML avec le titre “Exemples”. Voici d'autres sections que les auteurs de crate utilisent fréquemment dans leur documentation :

- **Panics** : les scénarios dans lesquels la fonction qui est documentée peut paniquer. Ceux qui utilisent la fonction et qui ne veulent pas que leur programme panique doivent s'assurer qu'ils n'appellent pas la fonction dans ce genre de situation.
- **Errors** : si la fonction retourne un `Result`, documenter les types d'erreurs qui peuvent survenir ainsi que les conditions qui mènent à ces erreurs sera très utile pour ceux qui utilisent votre API afin qu'ils puissent écrire du code pour gérer ces différents types d'erreurs de manière à ce que cela leur convienne.
- **Safety** : si la fonction fait un appel à `unsafe` (que nous verrons au chapitre 19), il devrait exister une section qui explique pourquoi la fonction fait appel à *unsafe* et quels sont les paramètres que la fonction s'attend à recevoir des utilisateurs de l'API.

La plupart des commentaires sur la documentation n'ont pas besoin de ces sections, mais c'est une bonne liste de vérifications à avoir pour vous rappeler les éléments importants à signaler aux utilisateurs.

Les commentaires de documentation pour faire des tests

L'ajout des blocs de code d'exemple dans vos commentaires de documentation peut vous aider à montrer comment utiliser votre bibliothèque, et faire ceci apporte un bonus supplémentaire : l'exécution de `cargo test` va lancer les codes d'exemples présents dans votre documentation comme étant des tests ! Il n'y a rien de mieux que de la documentation avec des exemples. Mais il n'y a rien de pire que des exemples qui ne fonctionnent plus car le code a changé depuis que la documentation a été écrite. Si nous lançons `cargo test` avec la documentation de la fonction `ajouter_un` de l'encart 14-1, nous verrons une section dans les résultats de tests comme celle-ci :

```
   Doc-tests ma_crate

running 1 test
test src/lib.rs - ajouter_un (line 5) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.27s
```

Maintenant, si nous changeons la fonction ou l'exemple de telle sorte que le `assert_eq!` de l'exemple panique et que nous lançons `cargo test` à nouveau, nous verrons que les tests de documentation vont découvrir que l'exemple et le code sont désynchronisés l'un de l'autre !

Commenter l'élément qui contient l'élément courant

Un autre style de commentaire de documentation, `//!`, ajoute de la documentation à l'élément qui contient ce commentaire plutôt que d'ajouter la documentation à l'élément qui suit ce commentaire. Nous utilisons habituellement ces commentaires de documentation dans le fichier de la crate racine (qui est *src/lib.rs* par convention) ou à l'intérieur d'un module afin de documenter la crate ou le module dans son ensemble.

Par exemple, si nous souhaitons ajouter de la documentation qui décrit le rôle de la crate `ma_crate` qui contient la fonction `ajouter_un`, nous pouvons ajouter des commentaires de documentation qui commencent par `//!` au début du fichier *src/lib.rs*, comme dans l'encart 14-2 :

Fichier : src/lib.rs

```
//! # Ma crate
//!
//! `ma_crate` est un regroupement d'utilitaires pour rendre plus pratique
//! certains calculs.

/// Ajoute 1 au nombre donné.
// -- partie masquée ici --
```

Encart 14-2 : documentation portant sur la crate `ma_crate`

Remarquez qu'il n'y a pas de code après la dernière ligne qui commence par `//!`. Comme nous commençons les commentaires par `//!` au lieu de `///`, nous documentons l'élément qui contient ce commentaire plutôt que l'élément qui suit ce commentaire. Dans notre cas, l'élément qui contient ce commentaire est le fichier *src/lib.rs*, qui est la racine de la crate. Ces commentaires vont décrire l'intégralité de la crate.

Lorsque nous lançons `cargo doc --open`, ces commentaires vont s'afficher sur la page d'accueil de la documentation de `ma_crate`, au-dessus de la liste des éléments publics de la crate, comme montré dans l'illustration 14-2 :

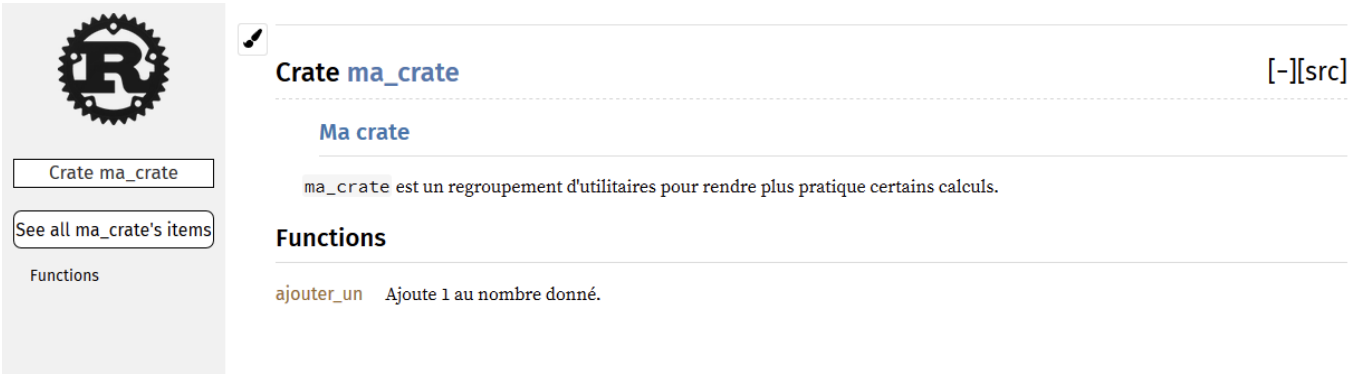


Illustration 14-2 : Documentation générée pour `ma_crate`, qui contient le commentaire qui décrit l'intégralité de la crate

Les commentaires de la documentation placés à l'intérieur des éléments sont particulièrement utiles pour décrire les crates et les modules. Utilisez-les pour expliquer globalement le rôle du conteneur pour aider vos utilisateurs à comprendre l'organisation de votre crate.

Exporter une API publique conviviale avec `pub use`

Dans le chapitre 7, nous avons vu comment organiser notre code en modules en utilisant le mot-clé `mod`, comment faire pour rendre des éléments publics en utilisant le mot-clé `pub`, et comment importer des éléments dans la portée en utilisant le mot-clé `use`. Cependant, la structure qui a un sens pour vous pendant que vous développez une crate peut ne pas être pratique pour vos utilisateurs. Vous pourriez vouloir organiser vos structures dans une hiérarchie qui a plusieurs niveaux, mais les personnes qui veulent utiliser un type que vous avez défini dans un niveau profond de la hiérarchie pourraient rencontrer des difficultés pour savoir que ce type existe. Ils peuvent aussi être agacés d'avoir à écrire `use ma_crate::un_module::un_autre_module::TypeUtile;` plutôt que `use ma_crate::TypeUtile;`.

La structure de votre API publique est une question importante lorsque vous publiez une crate. Les personnes qui utilisent votre crate sont moins familiers avec la structure que vous l'êtes et pourraient avoir des difficultés à trouver les éléments qu'ils souhaitent utiliser si votre crate a une hiérarchie de module imposante.

La bonne nouvelle est que si la structure *n'est pas* pratique pour ceux qui l'utilisent dans une autre bibliothèque, vous n'avez pas à réorganiser votre organisation interne : à la place, vous pouvez ré-exporter les éléments pour créer une structure publique qui est différente de votre structure privée en utilisant `pub use`. Ré-exporter prend un élément public d'un endroit et le rend public dans un autre endroit, comme s'il était défini dans l'autre endroit.

Par exemple, disons que nous avons créé une bibliothèque `art` pour modéliser des concepts artistiques. A l'intérieur de cette bibliothèque nous avons deux modules : un module `types` qui contient deux énumérations `CouleurPrimaire` et `CouleurSecondaire`, et un module `utilitaires` qui contient une fonction `mixer`, comme dans l'encart 14-3 :

Fichier : src/lib.rs

```
//! # Art
//!
//! Une bibliothèque pour modéliser des concepts artistiques.

pub mod types {
    /// Les couleurs primaires du modèle RJB.
    pub enum CouleurPrimaire {
        Rouge,
        Jaune,
        Bleu,
    }

    /// Les couleurs secondaires du modèle RJB.
    pub enum CouleurSecondaire {
        Orange,
        Vert,
        Violet,
    }
}

pub mod utilitaires {
    use crate::types::*;

    /// Combine deux couleurs primaires dans les mêmes quantités pour
    /// créer une couleur secondaire.
    pub fn mixer(c1: CouleurPrimaire, c2: CouleurPrimaire) -> CouleurSecondaire
{
        // -- partie masquée ici --
    }
}
```

Encart 14-3 : une bibliothèque `art` avec des éléments organisés selon les modules `types` et `utilitaires`

L'illustration 14-3 montre la page d'accueil de la documentation de cette crate générée par `cargo doc` qui devrait ressembler à cela :

Illustration 14-3 : Page d'accueil de la documentation de `art` qui liste les modules `types` et

`utilitaires`

Notez que les types `CouleurPrimaire` et `CouleurSecondaire` ne sont pas listés sur la page d'accueil, pas plus que la fonction `mixer`. Nous devons cliquer sur `types` et `utilitaires` pour les voir.

Une autre crate qui dépend de cette bibliothèque va avoir besoin d'utiliser l'instruction `use` pour importer les éléments de `art` dans sa portée, en suivant la structure du module qui est actuellement définie. L'encart 14-4 montre un exemple d'une crate qui utilise les éléments `CouleurPrimaire` et `mixer` de la crate `art` :

Fichier : src/main.rs

```
use art::types::CouleurPrimaire;
use art::utilitaires::mixer;

fn main() {
    let rouge = CouleurPrimaire::Rouge;
    let jaune = CouleurPrimaire::Jaune;
    mixer(rouge, jaune);
}
```

Encart 14-4 : une crate qui utilise les éléments de la crate `art` avec sa structure interne exportée

L'auteur du code de l'encart 14-4, qui utilise la crate `art`, doit comprendre que `CouleurPrimaire` est dans le module `types` et que `mixer` est dans le module `utilitaires`. La structure du module de la crate `art` est bien plus pratique pour les développeurs qui travaillent sur la crate `art` que pour les développeurs qui utilisent la crate `art`. La structure interne qui divise les éléments de la crate dans le module `types` et le module `utilitaires` ne contient aucune information utile à quelqu'un qui essaye de comprendre comment utiliser la crate `art`. Au lieu de cela, la structure du module de la crate `art` génère de la confusion car les développeurs doivent découvrir où trouver les éléments, et la structure n'est pas pratique car les développeurs doivent renseigner les noms des modules dans les instructions `use`.

Pour masquer l'organisation interne de l'API publique, nous pouvons modifier le code de la crate `art` de l'encart 14-3 pour ajouter l'instruction `pub use` pour ré-exporter les éléments au niveau supérieur, comme montré dans l'encart 14-5 :

Fichier : src/lib.rs

```
//! # Art
//!
//! Une bibliothèque pour modéliser des concepts artistiques.

pub use self::types::CouleurPrimaire;
pub use self::types::CouleurSecondaire;
pub use self::utilitaires::mixer;

pub mod types {
    // -- partie masquée ici --
}

pub mod utilitaires {
    // -- partie masquée ici --
}
```

Encart 14-5 : ajout de l'instruction `pub use` pour ré-exporter les éléments

La documentation de l'API que `cargo doc` a générée pour cette crate va maintenant lister et lier les ré-exportations sur la page d'accueil, comme dans l'illustration 14-4, ce qui rend les types `CouleurPrimaire` et `CouleurSecondaire` plus faciles à trouver.

Illustration 14-4 : la page d'accueil de la documentation pour `art` qui liste les ré-exports

Les utilisateurs de la crate `art` peuvent toujours voir et utiliser la structure interne de l'encart 14-3 comme ils l'utilisaient dans l'encart 14-4, mais ils peuvent maintenant utiliser la structure plus pratique de l'encart 14-5, comme montré dans l'encart 14-6 :

Fichier : src/main.rs

```
use art::mixer;
use art::CouleurPrimaire;

fn main() {
    // -- partie masquée ici --
}
```

Encart 14-6 : un programme qui utilise les éléments ré-exportés de la crate `art`

Dans les cas où il y a de nombreux modules imbriqués, ré-exporter les types au niveau le plus haut avec `pub use` peut faire une différence significative dans l'expérience utilisateur de ceux qui utilisent cette crate.

Créer une structure d'API publique utile est plus un art qu'une science, et vous pouvez itérer plusieurs fois pour trouver une API qui fonctionne mieux pour vos utilisateurs. Choisir `pub use` vous donne de la flexibilité pour l'organisation interne de votre crate et découple la structure interne de ce que vous présentez aux utilisateurs. N'hésitez pas à regarder le code source des crates que vous avez installées pour voir si leur structure interne est différente de leur API publique.

Mise en place d'un compte crates.io

Avant de pouvoir publier une crate, vous devez créer un compte sur crates.io et obtenir un jeton d'API. Pour pouvoir faire cela, visitez la page d'accueil de crates.io et connectez-vous avec votre compte GitHub (le compte GitHub est actuellement une obligation, mais crates.io pourra permettre de créer un compte d'une autre manière un jour). Une fois identifié, consultez les réglages de votre compte à l'adresse https://crates.io/me/ et récupérez votre jeton d'API (NdT : *API key*). Ensuite, lancez la commande `cargo login` avec votre clé d'API, comme ceci :

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

Cette commande informera cargo de votre jeton d'API et l'enregistrera localement dans *~/.cargo/credentials*. Notez que ce jeton est un *secret* : ne le partagez avec personne d'autre. Si vous le donnez à quelqu'un pour une quelconque raison, vous devriez le révoquer et générer un nouveau jeton sur crates.io.

Ajouter des métadonnées à une nouvelle crate

Maintenant que vous avez un compte, imaginons que vous avez une crate que vous souhaitez publier. Avant de la publier, vous aurez besoin d'ajouter quelques métadonnées à

votre crate en les ajoutant à la section `[package]` du fichier *Cargo.toml* de votre crate.

Votre crate va avoir besoin d'un nom unique. Tant que vous travaillez en local, vous pouvez nommer un crate comme vous le souhaitez. Cependant, les noms des crates sur crates.io sont accordés selon le principe du *premier arrivé, premier servi*. Une fois qu'un nom de crate est accordé, personne d'autre ne peut publier une crate avec ce nom. Avant d'essayer de publier une crate, recherchez sur le site le nom que vous souhaitez utiliser. Si le nom a été utilisé par une autre crate, vous allez devoir trouver un autre nom et modifier le champ `name` dans le fichier *Cargo.toml* sous la section `[package]` pour utiliser le nouveau nom pour la publication, comme ceci :

Fichier : Cargo.toml

```
[package]
name = "jeu_du_plus_ou_du_moins"
```

Même si vous avez choisi un nom unique, lorsque vous lancez `cargo publish` pour publier la crate à ce stade, vous allez avoir un avertissement suivi par une erreur :

```
$ cargo publish
    Updating crates.io index
warning: manifest has no description, license, license-file, documentation,
homepage or repository.
See https://doc.rust-lang.org/cargo/reference/manifest.html#package-metadata for
more info.
-- partie masquée ici --
error: failed to publish to registry at https://crates.io

Caused by:
  the remote server responded with an error: missing or empty metadata fields:
description, license. Please see https://doc.rust-lang.org/cargo/reference/
manifest.html for how to upload metadata
```

La raison est qu'il manque quelques informations essentielles : une description et une licence sont nécessaires pour que les gens puissent savoir ce que fait votre crate et sous quelles conditions ils peuvent l'utiliser. Pour corriger cette erreur, vous devez rajouter ces informations dans le fichier *Cargo.toml*.

Ajoutez une description qui ne fait qu'une phrase ou deux, car elle va s'afficher à proximité de votre crate dans les résultats de recherche. Pour le champ `license`, vous devez donner une *valeur d'identification de la licence*. La Linux Foundation's Software Package Data Exchange (SPDX) liste les identifications que vous pouvez utiliser pour cette valeur. Par exemple, pour stipuler que votre crate est sous la licence MIT, ajoutez l'identifiant `MIT` :

Fichier : Cargo.toml

```
[package]
name = "jeu_du_plus_ou_du_moins"
license = "MIT"
```

Si vous voulez utiliser une licence qui n'apparaît pas dans le SPDX, vous devez placer le texte de cette licence dans un fichier, inclure ce fichier dans votre projet puis utiliser `licence-file` pour renseigner le nom de ce fichier plutôt que d'utiliser la clé `licence`.

Les conseils sur le choix de la licence appropriée pour votre projet sortent du cadre de ce livre. De nombreuses personnes dans la communauté Rust appliquent à leurs projets la même licence que Rust qui utilise la licence double `MIT OR Apache-2.0`. Cette pratique montre que vous pouvez également indiquer plusieurs identificateurs de licence séparés par `OR` pour avoir plusieurs licences pour votre projet.

Une fois le nom unique, la version, la description et la licence ajoutés, le fichier *Cargo.toml* de ce projet qui est prêt à être publié devrait ressembler à ceci :

Fichier : Cargo.toml

```
[package]
name = "jeu_du_plus_ou_du_moins"
version = "0.1.0"
edition = "2021"
description = "Un jeu où vous devez deviner quel nombre l'ordinateur a choisi."
license = "MIT OR Apache-2.0"

[dependencies]
```

La documentation de cargo décrit d'autres métadonnées que vous pouvez renseigner pour vous assurer que les autres développeurs puissent découvrir et utiliser votre crate plus facilement.

Publier sur crates.io

Maintenant que vous avez créé un compte, sauvegardé votre jeton de clé, choisi un nom pour votre crate, et précisé les métadonnées requises, vous êtes prêt à publier ! Publier une crate téléverse une version précise sur crates.io pour que les autres puissent l'utiliser.

Faites attention lorsque vous publiez une crate car une publication est *permanente*. La version ne pourra jamais être remplacée, et le code ne pourra jamais être effacé. Le but majeur de crates.io est de fournir une archive durable de code afin que les compilations de tous les projets qui dépendent des crates de crates.io puissent toujours continuer à fonctionner. Si la suppression de version était autorisée, cela rendrait ce but impossible.

Cependant, il n'y a pas de limites au nombre de versions de votre crate que vous pouvez publier.

Lancez la commande `cargo publish` à nouveau. Elle devrait fonctionner à présent :

```
$ cargo publish
    Updating crates.io index
   Packaging jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
   Verifying jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
   Compiling jeu_du_plus_ou_du_moins v0.1.0
(file:///projects/jeu_du_plus_ou_du_moins/target/package/
jeu_du_plus_ou_du_moins-0.1.0)
    Finished dev [unoptimized + debuginfo] target(s) in 0.19s
   Uploading jeu_du_plus_ou_du_moins v0.1.0 (file:///projects/
jeu_du_plus_ou_du_moins)
```

Félicitations ! Vous venez de partager votre code avec la communauté Rust, et désormais tout le monde peut facilement ajouter votre crate comme une dépendance de son projet.

Publier une nouvelle version d'une crate existante

Lorsque vous avez fait des changements sur votre crate et que vous êtes prêt à publier une nouvelle version, vous devez changer la valeur de `version` renseignée dans votre fichier *Cargo.toml* et la publier à nouveau. Utilisez les règles de versionnage sémantique pour choisir quelle sera la prochaine version la plus appropriée en fonction des changements que vous avez faits. Lancez ensuite `cargo publish` pour téléverser la nouvelle version.

Retirer des versions de crates.io avec `cargo yank`

Bien que vous ne puissiez pas enlever des versions précédentes d'une crate, vous pouvez prévenir les futurs projets de ne pas l'ajouter comme une nouvelle dépendance. Cela s'avère pratique lorsqu'une version de crate est défectueuse pour une raison ou une autre. Dans de telles circonstances, cargo permet de *déprécier* une version de crate.

Déprécier une version évite que les nouveaux projets ajoutent une dépendance à cette version tout en permettant à tous les projets existants de continuer à en dépendre en leur permettant toujours de télécharger et dépendre de cette version. En gros, une version dépréciée permet à tous les projets avec un *Cargo.lock* de ne pas échouer, mais tous les futurs fichiers *Cargo.lock* générés n'utiliseront pas la version dépréciée.

Pour déprécier une version d'une crate, lancez `cargo yank` et renseignez quelle version vous voulez déprécier :

```
$ cargo yank --vers 1.0.1
```

Si vous ajoutez `--undo` à la commande, vous pouvez aussi annuler une dépréciation et permettre à nouveaux aux projets de dépendre de cette version :

```
$ cargo yank --vers 1.0.1 --undo
```

Une dépréciation *ne supprime pas* du code. Par exemple, la fonctionnalité de dépréciation n'est pas conçue pour supprimer des *secrets* téléversés par mégarde. Si cela arrive, vous devez régénérer immédiatement ces secrets.

Les espaces de travail de cargo

Dans le chapitre 12, nous avons construit un paquet qui comprenait une crate binaire et une crate de bibliothèque. Au fur et à mesure que votre projet se développe, vous pourrez constater que la crate de bibliothèque continue de s'agrandir et vous voudriez alors peut-être diviser votre paquet en plusieurs crates de bibliothèque. Pour cette situation, cargo a une fonctionnalité qui s'appelle *les espaces de travail* qui peuvent aider à gérer plusieurs paquets liés qui sont développés en tandem.

Créer un espace de travail

Un *espace de travail* est un jeu de paquets qui partagent tous le même *Cargo.lock* et le même dossier de sortie. Créons donc un projet en utilisant un espace de travail — nous allons utiliser du code trivial afin de nous concentrer sur la structure de l'espace de travail. Il existe plusieurs façons de structurer un espace de travail ; nous allons vous montrer une manière commune d'organisation. Nous allons avoir un espace de travail contenant un binaire et deux bibliothèques. Le binaire, qui devrait fournir les fonctionnalités principales, va dépendre des deux bibliothèques. Une bibliothèque va fournir une fonction `ajouter_un`, et la seconde bibliothèque, une fonction `ajouter_deux`. Ces trois crates feront partie du même espace de travail. Nous allons commencer par créer un nouveau dossier pour cet espace de travail :

```
$ mkdir ajout
$ cd ajout
```

Ensuite, dans le dossier *ajout*, nous créons le fichier *Cargo.toml* qui va configurer l'intégralité de l'espace de travail. Ce fichier n'aura pas de section `[package]` ou les métadonnées que nous avons vues dans les autres fichiers *Cargo.toml*. A la place, il commencera par une section `[workspace]` qui va nous permettre d'ajouter des membres à l'espace de travail en renseignant le chemin vers le paquet qui contient notre crate binaire ; dans ce cas, ce chemin est *additioneur* :

Fichier : Cargo.toml

```
[workspace]

members = [
    "additioneur",
]
```

Ensuite, nous allons créer la crate binaire `additioneur` en lançant `cargo new` dans le

dossier *ajout* :

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

A partir de ce moment, nous pouvons compiler l'espace de travail en lançant `cargo build`. Les fichiers dans votre dossier *ajout* devraient ressembler à ceci :

```
├── Cargo.lock
├── Cargo.toml
├── additioneur
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

L'espace de travail a un dossier *target* au niveau le plus haut pour y placer les artefacts compilés ; le paquet `additioneur` n'a pas son propre dossier *target*. Même si nous lancions `cargo build` à l'intérieur du dossier *additioneur*, les artefacts compilés finirons toujours dans *ajout/target* plutôt que dans *ajout/additioneur/target*. Cargo organise ainsi le dossier *target* car les crates d'un espace de travail sont censés dépendre l'une de l'autre. Si chaque crate avait son propre dossier *target*, chaque crate devrait recompiler chacune des autres crates présentes dans l'espace de travail pour avoir les artefacts dans son propre dossier *target*. En partageant un seul dossier *target*, les crates peuvent éviter des re-compilations inutiles.

Créer le second paquet dans l'espace de travail

Ensuite, créons un autre paquet, membre de l'espace de travail et appelons-le `ajouter_un`. Changeons le *Cargo.toml* du niveau le plus haut pour renseigner le chemin vers *ajouter_un* dans la liste `members` :

Fichier : Cargo.toml

```
[workspace]

members = [
    "additioneur",
    "ajouter_un",
]
```

Ensuite, générons une nouvelle crate de bibliothèque `ajouter_un` :

```
$ cargo new ajouter_un --lib
     Created library `ajouter_un` package
```

Votre dossier *ajout* devrait maintenant avoir ces dossiers et fichiers :

```
├── Cargo.lock
├── Cargo.toml
├── ajouter_un
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── additioneur
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└── target
```

Dans le fichier *ajouter_un/src/lib.rs*, ajoutons une fonction `ajouter_un` :

Fichier : ajouter_un/src/lib.rs

```
pub fn ajouter_un(x: i32) -> i32 {
    x + 1
}
```

Maintenant que nous avons un autre paquet dans l'espace de travail, nous pouvons faire en sorte que le paquet `additioneur` qui contient notre binaire dépende du paquet `ajouter_un`, qui contient notre bibliothèque. D'abord, nous devons ajouter un chemin de dépendance à `ajouter_un` dans *additioneur/Cargo.toml*.

Fichier : additioneur/Cargo.toml

```
[dependencies]
ajouter_un = { path = "../ajouter_un" }
```

Cargo ne fait pas la supposition que les crates d'un espace de travail dépendent l'une de l'autre, donc vous devez être explicites sur les relations de dépendance entre les crates.

Ensuite, utilisons la fonction `ajouter_un` de la crate `ajouter_un` dans la crate `additioneur`. Ouvrez le fichier *additioneur/src/main.rs* et ajoutez une ligne `use` tout en haut pour importer la bibliothèque `ajouter_un` dans la portée. Changez ensuite la fonction `main` pour appeler la fonction `ajouter_un`, comme dans l'encart 14-7.

Fichier : additioneur/src/main.rs

```
use ajouter_un;

fn main() {
    let nombre = 10;
    println!(
        "Hello, world ! {} plus un vaut {} !",
        nombre,
        ajouter_un::ajouter_un(nombre)
    );
}
```

Encart 14-7 : utilisation de la bibliothèque `ajouter_un` dans la crate `additioneur`

Compilons l'espace de travail en lançant `cargo build` dans le niveau le plus haut du dossier *ajout* !

```
$ cargo build
   Compiling ajouter_un v0.1.0 (file:///projects/ajout/ajouter_un)
   Compiling additioneur v0.1.0 (file:///projects/ajout/additioneur)
    Finished dev [unoptimized + debuginfo] target(s) in 0.68s
```

Pour lancer la crate binaire à partir du dossier *ajout*, nous pouvons préciser quel paquet nous souhaitons exécuter dans l'espace de travail en utilisant l'argument `-p` suivi du nom du paquet avec `cargo run` :

```
$ cargo run -p additioneur
    Finished dev [unoptimized + debuginfo] target(s) in 0.0s
     Running `target/debug/additioneur`
Hello, world ! 10 plus un vaut 11 !
```

Cela exécute le code de *additioneur/src/main.rs*, qui dépend de la crate `ajouter_un` .

Dépendre d'un paquet externe dans un espace de travail

Notez que l'espace de travail a un seul fichier *Cargo.lock* dans le niveau le plus haut de l'espace de travail plutôt que d'avoir un *Cargo.lock* dans chaque dossier de chaque crate. Cela garantit que toutes les crates utilisent la même version de toutes les dépendances. Si nous ajoutons le paquet `rand` aux fichiers *additioneur/Cargo.toml* et *ajouter_un/Cargo.toml*, cargo va réunir les deux en une seule version de `rand` et enregistrer cela dans un seul *Cargo.lock*. Faire en sorte que toutes les crates de l'espace de travail utilisent la même dépendance signifie que les crates dans l'espace de travail seront toujours compatibles l'une avec l'autre. Ajoutons la crate `rand` à la section `[dependencies]` du fichier *ajouter_un/Cargo.toml* pour pouvoir utiliser la crate `rand` dans la crate `ajouter_un` :

Fichier : ajouter_un/Cargo.toml

```
[dependencies]
rand = "0.8.3"
```

Nous pouvons maintenant ajouter `use rand;` au fichier *ajouter_un/src/lib.rs* et compiler l'ensemble de l'espace de travail en lançant `cargo build` dans le dossier *ajout*, ce qui va importer et compiler la crate `rand`. Nous devrions avoir un avertissement car nous n'avons pas utilisé le `rand` que nous avons introduit dans la portée :

```
$ cargo build
    Updating crates.io index
  Downloaded rand v0.8.3
   -- partie masquée ici --
   Compiling rand v0.8.3
   Compiling ajouter_un v0.1.0 (file:///projects/ajout/ajouter_un)
warning: unused import: `rand`
 --> ajouter_un/src/lib.rs:1:5
  |
1 | use rand;
  |     ^^^^
  |
  = note: `#[warn(unused_imports)]` on by default

warning: 1 warning emitted

   Compiling additioneur v0.1.0 (file:///projects/ajout/additioneur)
    Finished dev [unoptimized + debuginfo] target(s) in 10.18s
```

Le *Cargo.lock* du niveau le plus haut contient maintenant les informations de dépendance à `rand` pour `ajouter_un`. Cependant, même si `rand` est utilisé quelque part dans l'espace de travail, nous ne pouvons pas l'utiliser dans d'autres crates de l'espace de travail tant que nous n'ajoutons pas `rand` dans leurs fichiers *Cargo.toml*. Par exemple, si nous ajoutons `use rand;` dans le fichier *additioneur/src/main.rs* pour le paquet `additioneur`, nous allons avoir une erreur :

```
$ cargo build
  -- partie masquée ici --
   Compiling additioneur v0.1.0 (file:///projects/ajout/additioneur)
error[E0432]: unresolved import `rand`
 --> additioneur/src/main.rs:2:5
  |
2 | use rand;
  |     ^^^^ no external crate `rand`
```

Pour corriger cela, modifiez le fichier *Cargo.toml* pour le paquet `additioneur` et indiquez que `rand` est une dépendance de cette crate aussi. La compilation du paquet `additioneur` va rajouter `rand` à la liste des dépendances pour `additioneur` dans *Cargo.lock*, mais

aucune copie supplémentaire de `rand` ne sera téléchargée. Cargo s'est assuré que toutes les crates de chaque paquet de l'espace de travail qui utilise le paquet `rand` seraient de la même version. Utiliser la même version de `rand` dans les espaces de travail économise de l'espace car nous n'avons pas à multiplier les copies, ni à nous assurer que les crates dans l'espace de travail sont compatibles les unes avec les autres.

Ajouter un test à l'espace de travail

Afin de procéder à une autre amélioration, ajoutons un test de la fonction `ajouter_un::ajouter_un` dans la crate `ajouter_un` :

Fichier : add_one/src/lib.rs

```
pub fn ajouter_un(x: i32) -> i32 {
    x + 1
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn cela_fonctionne() {
        assert_eq!(3, ajouter_un(2));
    }
}
```

Lancez maintenant `cargo test` dans le niveau le plus haut du dossier *ajout* :

```
$ cargo test
   Compiling ajouter_un v0.1.0 (file:///projects/ajout/ajouter_un)
   Compiling additioneur v0.1.0 (file:///projects/ajout/additioneur)
    Finished test [unoptimized + debuginfo] target(s) in 0.27s
     Running target/debug/deps/ajouter_un-f0253159197f7841

running 1 test
test tests::cela_fonctionne ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

     Running target/debug/deps/additioneur-49979ff40686fa8e

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests ajouter_un

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

La première section de la sortie indique que le test `cela_fonctionne` de la crate `ajouter_un` a réussi. La section suivante indique qu'aucun test n'a été trouvé dans la crate `additioneur`, puis la dernière section indique elle aussi qu'aucun test de documentation n'a été trouvé dans la crate `ajouter_un`. Lancer `cargo test` dans un espace de travail structuré comme celui-ci va exécuter les tests pour toutes les crates de cet espace de travail.

Nous pouvons aussi lancer des tests pour une crate en particulier dans un espace de travail à partir du dossier du plus haut niveau en utilisant le drapeau `-p` et en renseignant le nom de la crate que nous voulons tester :

```
$ cargo test -p ajouter_un
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
     Running target/debug/deps/ajouter_un-b3235fea9a156f74

running 1 test
test tests::cela_fonctionne ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests ajouter_un

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

Cette sortie montre que `cargo test` a lancé les tests uniquement pour la crate `ajouter_un` et n'a pas lancé les tests de la crate `additioneur` .

Si vous publiez les crates présentes dans l'espace de travail sur crates.io, chaque crate de l'espace de travail va avoir besoin d'être publiée de manière séparée. La commande `cargo publish` n'a pas de drapeau `--all` ou `-p` , donc vous devrez vous rendre dans chaque dossier de chaque crate et lancer `cargo publish` sur chaque crate présente dans l'espace de travail pour publier les crates.

En guise d'entrainement supplémentaire, ajoutez une crate `ajouter_deux` dans cet espace de travail de la même manière que nous l'avons fait pour la crate `ajouter_un` !

Au fur et à mesure que votre projet se développe, pensez à utiliser un espace de travail : il est plus facile de comprendre des composants individuels, plus petits, plutôt qu'un gros tas de code. De plus, garder les crates dans un espace de travail peut améliorer la coordination entre elles si elles sont souvent modifiées ensemble.

Installer des binaires à partir de crates.io avec `cargo install`

La commande `cargo install` vous permet d'installer et utiliser des crates de binaires localement. Cela n'est pas conçu pour remplacer les paquets systèmes ; c'est plutôt un moyen pratique pour les développeurs Rust d'installer des outils que les autres ont partagé sur crates.io. Notez que vous ne pouvez installer que des paquets qui ont des destinations binaires. Une *destination binaire* est le programme exécutable qui est créé si la crate a un fichier *src/main.rs* ou un autre fichier désigné comme binaire, par opposition à une destination de bibliothèque qui n'est pas exécutable en tant que telle mais qu'il est possible d'intégrer à d'autres programmes. Habituellement, l'information permettant de savoir si une crate est une bibliothèque, possède plutôt une destination binaire ou les deux à la fois figure dans le fichier *README*.

Tous les binaires installés avec `cargo install` sont stockés dans le dossier *bin* de la racine. Si vous installez Rust avec *rustup.rs* et que vous n'avez pas personnalisé la configuration, ce dossier sera *$HOME/.cargo/bin*. Assurez-vous que ce dossier est dans votre `$PATH` pour pouvoir exécuter des programmes que vous avez installés avec `cargo install` .

Par exemple, dans le chapitre 12, nous avions mentionné le fait qu'il existait une implémentation de l'outil `grep` en Rust qui s'appelait `ripgrep` et qui permettait de rechercher dans des fichiers. Si nous voulons installer `ripgrep` , nous pouvons faire comme ceci :

```
$ cargo install ripgrep
    Updating crates.io index
  Downloaded ripgrep v11.0.2
  Downloaded 1 crate (243.3 KB) in 0.88s
  Installing ripgrep v11.0.2
-- partie masquée ici --
   Compiling ripgrep v11.0.2
    Finished release [optimized + debuginfo] target(s) in 3m 10s
  Installing ~/.cargo/bin/rg
   Installed package `ripgrep v11.0.2` (executable `rg`)
```

L'avant-dernière ligne de la sortie nous montre l'emplacement et le nom du binaire installé, qui est `rg` dans le cas de `ripgrep` . Tel que mentionné précédemment, du moment que le dossier d'installation est dans votre `$PATH` , vous pouvez ensuite lancer `rg --help` et commencer à utiliser un outil en Rust plus rapide pour rechercher dans des fichiers !

Etendre les fonctionnalités de cargo avec des commandes personnalisées

Cargo est conçu pour que vous puissiez étendre ses fonctionnalités avec des nouvelles sous-commandes sans avoir à modifier cargo. Si un binaire dans votre `$PATH` est nommé selon `cargo-quelquechose`, vous pouvez le lancer comme s'il était une sous-commande de cargo en lançant `cargo quelquechose`. Les commandes personnalisées comme celle-ci sont aussi listées lorsque vous lancez `cargo --list`. Pouvoir utiliser `cargo install` pour installer des extensions et ensuite les lancer comme étant un outil intégré à cargo est un avantage super pratique de la conception de cargo !

Résumé

Le partage de code avec cargo et crates.io fait partie de ce qui rend l'écosystème de Rust très utile pour de nombreuses tâches. La bibliothèque standard de Rust est compacte et stable, et les crates sont faciles à partager, à utiliser et à améliorer à un rythme différent de celui du langage. N'hésitez pas à partager du code qui vous est utile sur crates.io ; il est fort probable qu'il sera aussi utile à quelqu'un d'autre !

Les pointeurs intelligents

Un *pointeur* est un concept général pour une variable qui contient une adresse vers la mémoire. Cette adresse pointe vers d'autres données. Le type de pointeur le plus courant en Rust est la référence, que vous avez appris au chapitre 4. Les références sont marquées par le symbole `&` et empruntent la valeur sur laquelle elles pointent. Elles n'ont pas d'autres fonctionnalités que celle de pointer sur une donnée. De plus, elles n'ont aucun coût sur les performances et c'est le type de pointeur que nous utilisons le plus souvent.

Les *pointeurs intelligents*, d'un autre côté, sont des structures de données qui, non seulement se comportent comme un pointeur, mais ont aussi des fonctionnalités et métadonnées supplémentaires. Le concept de pointeur intelligent n'est pas propre à Rust : les pointeurs intelligents sont originaires du C++ et existent aussi dans d'autres langages. En Rust, les différents pointeurs intelligents définis dans la bibliothèque standard fournissent des fonctionnalités supplémentaires à celles des références. Un exemple que nous allons explorer dans ce chapitre est le type de pointeur intelligent *compteur de références*. Ce pointeur vous permet d'avoir plusieurs propriétaires d'une donnée tout en gardant une trace de leur nombre et, lorsqu'il n'y en a plus, de nettoyer cette donnée.

En Rust, qui utilise le concept de propriété et d'emprunt, une différence supplémentaire entre les références et les pointeurs intelligents est que les références sont des pointeurs qui empruntent seulement la donnée ; alors qu'au contraire, dans de nombreux cas, les pointeurs intelligents sont *propriétaires* des données sur lesquelles ils pointent.

Nous avons déjà rencontré quelques pointeurs intelligents au cours de ce livre, comme `String` et `Vec<T>` au chapitre 8, même si nous ne les avons pas désignés comme étant des pointeurs intelligents à ce moment-là. Ces deux types sont considérés comme des pointeurs intelligents car ils sont propriétaires de données et vous permettent de les manipuler. Ils ont aussi des métadonnées (comme leur capacité) et certaines fonctionnalités ou garanties (comme `String` qui s'assure que ses données soient toujours en UTF-8 valide).

Les pointeurs intelligents sont souvent implémentés en utilisant des structures. Les caractéristiques qui distinguent un pointeur intelligent d'une structure classique est que les pointeurs intelligents implémentent les traits `Deref` et `Drop`. Le trait `Deref` permet à une instance d'un pointeur intelligent de se comporter comme une référence afin que vous puissiez écrire du code qui fonctionne aussi bien avec des références qu'avec des pointeurs intelligents. Le trait `Drop` vous permet de personnaliser le code qui est exécuté lorsqu'une instance d'un pointeur intelligent sort de la portée. Dans ce chapitre, nous verrons ces deux traits et expliquerons pourquoi ils sont importants pour les pointeurs intelligents.

Vu que le motif des pointeurs intelligents est un motif de conception général fréquemment

utilisé en Rust, ce chapitre ne couvrira pas tous les pointeurs intelligents existants. De nombreuses bibliothèques ont leurs propres pointeurs intelligents, et vous pouvez même écrire le vôtre. Nous allons voir les pointeurs intelligents les plus courants de la bibliothèque standard :

- `Box<T>` pour l'allocation de valeurs sur le tas
- `Rc<T>` , un type comptant les références, qui permet d'avoir plusieurs propriétaires
- `Ref<T>` et `RefMut<T>` , auxquels on accède via `RefCell<T>` , un type qui permet d'appliquer les règles d'emprunt au moment de l'exécution plutôt qu'au moment de la compilation

En outre, nous allons voir le motif de *mutabilité interne* dans lequel un type immuable propose une API pour modifier une valeur interne. Nous allons aussi parler des *boucles de références* : comment elles peuvent provoquer des fuites de mémoire et comment les éviter.

Allons-y !

Utiliser `Box<T>` pour pointer sur des données présentes sur le tas

Le pointeur intelligent le plus simple est la *boite*, dont le type s'écrit `Box<T>`. Les boites vous permettent de stocker des données sur le tas plutôt que sur la pile. La seule chose qui reste sur la pile est le pointeur vers les données sur le tas. Revenez au chapitre 4 pour vous rappeler la différence entre la pile et le tas.

Les boites ne provoquent pas de surcharge au niveau des performances, si ce n'est le stockage de leurs données sur le tas plutôt que sur la pile. Mais elles n'ont pas non plus beaucoup plus de fonctionnalités. Vous allez les utiliser principalement dans les situations suivantes :

- Lorsque vous avez un type dont la taille ne peut pas être connu au moment de la compilation et que vous souhaitez une valeur d'un certain type dans un contexte qui nécessite de savoir exactement sa taille
- Lorsque vous avez une grosse quantité de données et que vous souhaitez transférer la possession tout en assurant que les données ne seront pas copiées lorsque vous le ferez
- Lorsque vous voulez prendre possession d'une valeur et que vous souhaitez seulement qu'elle soit d'un type qui implémente un trait particulier plutôt que d'être d'un type spécique

Nous allons expérimenter la première situation dans la section “Pouvoir utiliser des types récursifs grâce aux boites”. Pour la seconde situation, le transfert de possession d'une grosse quantité de données peut prendre beaucoup de temps car les données sont recopiées sur la pile. Pour améliorer les performances dans cette situation, nous pouvons stocker ces données sur le tas grâce à une boite. Ainsi, seul le petit pointeur vers les données est copié sur la pile, alors que les données qu'il pointe restent à leur place sur le tas. La troisième situation décris ce qu'on appelle un *objet de trait* et le chapitre 17 dédie une section entière à ce sujet. Donc ce que vous apprenez ici, vous le retrouverez à nouveau au chapitre 17 !

Utiliser une `Box<T>` pour stocker des données sur le tas

Avant de parler de ce cas d'usage de `Box<T>`, nous devons voir sa syntaxe et comment interagir avec les valeurs stockées dans un `Box<T>`.

L'encart 15-1 nous montre comment utiliser une boite pour stocker une valeur `i32` sur le tas :

Fichier : src/main.rs

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

Encart 15-1 : stocker une valeur `i32` sur le tas en utilisant une boîte

Nous avons défini la variable `b` pour avoir la valeur d'une `Box` qui pointe sur la valeur `5`, qui est donc allouée sur le tas. Ce programme va afficher `b = 5` ; dans ce cas, nous pouvons accéder à la donnée présente dans la boite de la même manière que nous le ferrions si elle était sur la pile. Comme toute valeur possédée, lorsque une boite sort de la portée, comme lorsque `b` le fait à la fin du `main`, elle sera désallouée. Ce sera la boite qui sera désallouée en premier (elle est stockée sur la pile), puis ce sera au tour des données sur lesquelles elle pointait (qui sont stockées sur le tas).

Déposer une seule valeur sur le tas n'est pas très utile, donc vous n'utiliserez que très rarement les boites de cette manière. Laisser les valeurs comme des `i32` indépendantes sur la pile, où elles sont stockées par défaut, reste plus approprié dans la majeure partie des situations. Regardons un cas où les boites nous permettent de définir des types que nous ne pourrions pas définir si nous n'avions pas les boites.

Pouvoir utiliser des types récursifs grâce aux boites

Au moment de la compilation, Rust a besoin de savoir combien d'espace prend un type. Un des types dont la taille ne peut pas être connu au moment de la compilation est le *type récursif*, dans lequel une valeur peut avoir une partie de sa définition qui a une valeur du même type qu'elle-même. Comme cet emboîtement de valeurs pourrait théoriquement se poursuivre à l'infini, Rust ne sait pas combien d'espace une valeur d'un type récursif peut avoir besoin. Cependant, les boites ont une taille connue, donc en utilisant une boite dans la définition d'un type récursif, vous pouvez créer des types récursifs.

Découvrons maintenant la *liste de construction* (NdT : cons list), qui est un type de donnée courant dans les langages de programmation fonctionnels, comme étant un exemple de type récursif. Le type liste de construction que nous allons définir est plutôt simple, sauf pour les cas de récursivité ; par conséquent, les concepts dans l'exemple avec lequel nous allons travailler vous seront utiles à chaque fois que vous vous retrouverez dans des situations plus complexes qui impliquent des types récursifs.

En savoir plus sur les listes de construction

Une *liste de construction* est une structure de donnée qui provient du langage de programmation Lisp et de ses dérivés. En Lisp, la fonction `cons` (qui est une forme contractée de “fonction de construction”) construit une nouvelle paire à partir de ses deux arguments, qui sont souvent une valeur individuelle et une autre paire. Ces paires qui contiennent des paires forment des listes.

Le concept de la fonction cons a fait son chemin dans le jargon plus général de la programmation fonctionnelle : "to cons *x* onto *y*" signifie de manière informelle de construire une nouvelle instance de conteneur en mettant l'élément *x* au début de ce nouveau conteneur, suivi du conteneur *y*.

Chaque élément dans une liste de construction contient deux éléments : la valeur de l'élément courant et celle de l'élément suivant. Le dernier élément dans la liste contient seulement une valeur `Nil` sans aucun élément suivant. Une liste de construction est produite de manière récursive en appelant la fonction `cons`. Le nom canonique pour indiquer le cas de base de la récursion est `Nil`. Notez que ce n'est pas la même chose que les concepts “null” ou “nil” du chapitre 6, qui signale une valeur invalide ou absente.

Bien que les langages de programmation fonctionnels utilisent les listes de construction fréquemment, la liste de construction n'est pas une structure de donnée utilisée couramment en Rust. La plupart du temps lorsque vous avez une liste d'éléments en Rust, `Vec<T>` s'avère être un meilleur choix à faire. Autrement, il existe des types de données récursifs plus complexes *qui sont* utiles dans d'autres situations, mais en commençant avec les listes de construction, nous pouvons découvrir comment les boites nous permettent de définir un type de données récursif sans être trop perturbé par la complexité.

L'encart 15-2 propose une définition d'une énumération pour une liste de construction. Notez que ce code ne se compile pas encore car le type `List` n'a pas encore de taille connue, ce que nous allons voir ensuite.

Fichier : src/main.rs

```
enum List {
    Cons(i32, List),
    Nil,
}
```

Encart 15-2 : première tentative de définition d'une énumération pour représenter une structure de données de liste de construction de valeurs `i32`

> Remarque : nous implémentons une liste de construction qui stocke uniquement des valeurs `i32` pour les besoins de cet exemple. Nous aurions pu l'implémenter en

> utilisant des génériques, que nous avons vu chapitre 10, afin de définir une liste de construction qui pourrait stocker n'importe quel type.

L'utilisation du type `List` pour stocker la liste `1, 2, 3` ressemblerait au code dans l'encart 15-3 :

Fichier : src/main.rs

```
use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

Encart 15-3 : utilisation de l'énumération `List` pour stocker la liste `1, 2, 3`

La première valeur `Cons` stocke `1` et une autre valeur de `List`. Cette valeur `List` est une autre valeur `Cons` qui stocke `2` et une autre valeur de `List`. Cette valeur `List` n'est rien d'autre qu'une valeur `Cons` qui stocke `3` et une valeur `List`, qui finalement est `Nil`, la variante non récursive qui signale la fin de la liste.

Si nous essayons de compiler le code de l'encart 15-3, nous avons l'erreur de l'encart 15-4 :

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |               ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List`
representable
  |
2 |     Cons(i32, Box<List>),
  |               ++++    +

error[E0391]: cycle detected when computing drop-check constraints for `List`
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^
  |
  = note: ...which immediately requires computing drop-check constraints for
`List` again
  = note: cycle used when computing dropck types for `Canonical { max_universe:
U0, variables: [], value: ParamEnvAnd { param_env: ParamEnv { caller_bounds: [],
reveal: UserFacing }, value: List } }`

Some errors have detailed explanations: E0072, E0391.
For more information about an error, try `rustc --explain E0072`.
error: could not compile `cons-list` due to 2 previous errors
```

Encart 15-4 : l'erreur que nous obtenons lorsque nous essayons de définir une énumération récursive

L'erreur explique que ce type “a une taille infinie”. La raison est que nous avons défini `List` avec une variante qui est récursive : elle stocke directement une autre valeur d'elle-même. Au final, Rust ne peut pas savoir combien de place il a besoin pour stocker une valeur `List`. Analysons pourquoi nous obtenons cette erreur. D'abord, regardons comment Rust décide de l'espace dont il a besoin pour stocker une valeur d'un type non récursif.

Calculer la taille d'un type non récursif

Rappelez-vous de l'énumération `Message` que nous avons défini dans l'encart 6-2 lorsque nous avons abordé les définitions des énumérations au chapitre 6 :

```
enum Message {
    Quitter,
    Deplacer { x: i32, y: i32 },
    Ecrire(String),
    ChangerCouleur(i32, i32, i32),
}
```

Pour déterminer combien d'espace allouer pour une valeur `Message`, Rust parcourt chaque variante pour voir quelle variante a besoin le plus d'espace. Rust voit que `Message::Quitter` n'a pas besoin d'espace, `Message::Deplacer` a besoin de suffisamment d'espace pour stocker deux valeurs `i32`, et ainsi de suite. Comme une seule variante sera utilisée, le plus grand espace dont une valeur de `Message` aura besoin sera l'espace que cela prendra de stocker la plus grosse de ses variantes.

Comparez cela avec ce qui se passe lorsque Rust essaye de déterminer combien d'espace un type récursif comme l'énumération `List` de l'encart 15-2 aurait besoin. Le compilateur commence par regarder la variante `Cons`, qui stocke une valeur de type `i32` et une valeur de type `List`. Ainsi, `Cons` a besoin d'une quantité d'espace égale à la taille d'un `i32` plus la taille d'une valeur `List`. Pour savoir combien de mémoire le type `List` a besoin, le compilateur va regarder ses variantes, en commençant avec la variante `Cons`. La variante `Cons` stocke une valeur de type `i32` et une valeur de type `List`, et ce processus continue à l'infini, comme l'illustration 15-1.

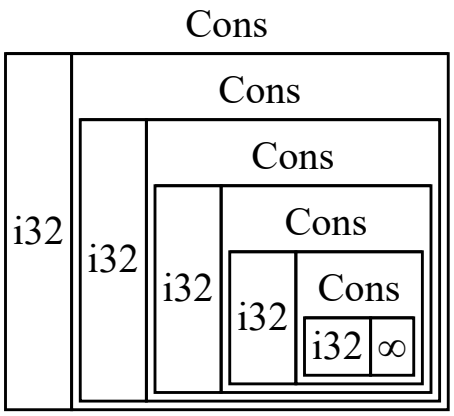


Illustration 15-1 : une `List` infinie qui contient des variantes `Cons` infinies

Utiliser `Box<T>` pour créer un type récursif avec une taille finie

Rust ne peut pas calculer la quantité d'espace à allouer pour les types définis récursivement, donc le compilateur déclenche l'erreur de l'encart 15-4. Mais l'erreur renferme cette suggestion très utile :

```
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List`
representable
  |
2 |     Cons(i32, Box<List>),
  |               ^^^^    ^
```

Dans cette suggestion, “indirection” (NdT : redirection) signifie qu'au lieu de stocker une valeur directement, nous devrions changer la structure des données pour stocker à la place un pointeur vers la valeur.

Comme `Box<T>` est un pointeur, Rust connaît toujours combien d'espace un `Box<T>` a besoin : la taille d'un pointeur ne change pas, peu importe la quantité de données sur lesquelles il pointe. Cela signifie que nous pouvons insérer un `Box<T>` à l'intérieur d'une variante `Cons` au lieu d'y mettre directement une autre valeur `List`. Le `Box<T>` va pointer sur la prochaine valeur `List` qui sera sur le tas plutôt que d'être dans la variante `Cons`. Théoriquement, nous avons toujours une liste, créée avec des listes qui “contiennent” d'autres listes, mais cette implémentation ressemble plus à présent à des éléments placés les uns à côté des autres plutôt que les uns dans les autres.

Nous pouvons changer la définition de l'énumération `List` de l'encart 15-2 et l'utilisation de `List` dans l'encart 15-3 pour le code de l'encart 15-5, qui va se compiler :

Filename : src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
}
```

Encart 15-5 : définition de `List` qui utilise `Box<T>` dans le but d'avoir une taille connue

La variante `Cons` va avoir besoin de l'espace d'un `i32` plus l'espace pour stocker le pointeur vers la donnée de la boite. La variante `Nil` ne stocke pas de valeurs, donc elle a besoin de moins d'espace que la variante `Cons`. Nous savons maintenant que chaque valeur `List` va prendre la taille d'un `i32` plus la taille d'un pointeur vers la donnée de la boite. En utilisant une boite, vous avez arrêté la chaine infinie et récursive, donc le compilateur peut savoir l'espace dont il a besoin pour stocker une valeur `List`. L'illustration 15-2 montre à quoi ressemble maintenant la variante `Cons`.

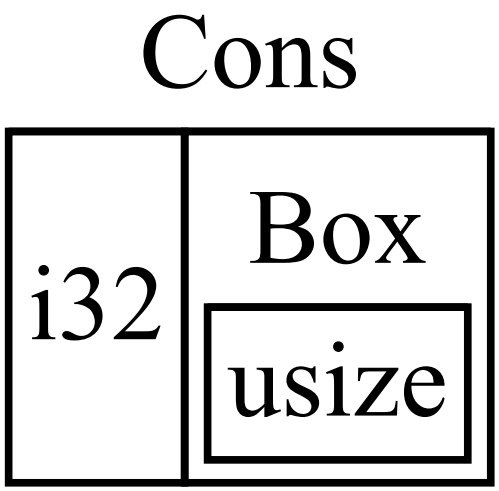


Illustration 15-2 : une `List` qui n'a pas de taille infinie car `Cons` est une `Box`

Les boites fournissent uniquement la redirection et l'allocation sur le tas ; elles n'ont pas d'autres fonctionnalités, comme celles que nous verrons sur d'autres types de pointeurs intelligents. Elles n'ont pas non plus de surcoût sur les performances autre que ce qu'offrent ces capacités spéciales, donc elles sont utiles dans des cas comme les listes de construction où la redirection est la seule fonctionnalité que nous avons besoin. Nous verrons aussi plus de cas d'usages pour les boites dans le chapitre 17.

Le type `Box<T>` est un pointeur intelligent car il implémente le trait `Deref`, qui permet aux valeurs `Box<T>` d'être traitées comme des références. Lorsque une valeur `Box<T>` sort de la portée, les données sur le tas pointées par la boite seront également nettoyées grâce à l'implémentation du trait `Drop`. Explorons plus en détail ces deux traits. Ces deux traits deviendrons encore plus importants pour les fonctionnalités offertes par les autres pointeurs intelligents que nous verrons dans le reste de ce chapitre.

Considérer les pointeurs intelligents comme des références grâce au trait `Deref`

L'implémentation du trait `Deref` vous permet de personnaliser le comportement de *l'opérateur de déréférencement* `*` (qui n'est pas l'opérateur de multiplication ou le joker global). En implémentant `Deref` de manière à ce qu'un pointeur intelligent puisse être considéré comme une référence classique, vous pouvez écrire du code qui fonctionne avec des références mais aussi avec des pointeurs intelligents.

Regardons d'abord comment l'opérateur de déréférencement fonctionne avec des références classiques. Ensuite nous essayerons de définir un type personnalisé qui se comporte comme `Box<T>` et voir pourquoi l'opérateur de déréférencement ne fonctionne pas comme une référence sur notre type fraîchement défini. Nous allons découvrir comment implémenter le trait `Deref` de manière à ce qu'il soit possible que les pointeurs intelligents fonctionnent comme les références. Ensuite nous verrons la fonctionnalité d'*extrapolation de déréférencement* de Rust et comment elle nous permet de travailler à la fois avec des références et des pointeurs intelligents.

> Remarque : il y a une grosse différence entre le type `MaBoite<T>` que nous allons construire et la vraie `Box<T>` : notre version ne va pas stocker ses données sur le tas. Nous allons concentrer cet exemple sur `Deref`, donc l'endroit où est concrètement stocké la donnée est moins important que le comportement similaire aux pointeurs.

Suivre le pointeur vers la valeur grâce à l'opérateur de déréférencement

Une référence classique est un type de pointeur, et une manière de modéliser un pointeur est d'imaginer une flèche pointant vers une valeur stockée autre part. Dans l'encart 15-6, nous créons une référence vers une valeur `i32` et utilisons ensuite l'opérateur de déréférencement pour suivre la référence vers la donnée :

Fichier : src/main.rs

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Encart 15-6 : utiliser l'opérateur de déréférencement pour suivre une référence vers une valeur `i32`

La variable `x` stocke une valeur `i32` : `5`. Nous avons assigné à `y` une référence vers `x`. Nous pouvons faire une `assert` pour vérifier que `x` est égal à `5`. Cependant, si nous souhaitons faire une `assert` sur la valeur dans `y`, nous devons utiliser `*y` pour suivre la référence vers la valeur sur laquelle elle pointe (d'où le *déréférencement*). Une fois que nous avons déréférencé `y`, nous avons accès à la valeur de l'entier sur laquelle `y` pointe afin que nous puissions la comparer avec `5`.

Si nous avions essayé d'écrire `assert_eq!(5, y);` à la place, nous aurions obtenu cette erreur de compilation :

```
$ cargo run
   Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0277]: can't compare `{integer}` with `&{integer}`
 --> src/main.rs:6:5
  |
6 |     assert_eq!(5, y);
  |     ^^^^^^^^^^^^^^^^ no implementation for `{integer} == &{integer}`
  |
  = help: the trait `PartialEq<&{integer}>` is not implemented for `{integer}`
  = note: this error originates in the macro `assert_eq` (in Nightly builds, run
with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0277`.
error: could not compile `deref-example` due to previous error
```

Comparer un nombre et une référence vers un nombre n'est pas autorisé car ils sont de types différents. Nous devons utiliser l'opérateur de déréférencement pour suivre la référence vers la valeur sur laquelle elle pointe.

Utiliser `Box<T>` comme étant une référence

Nous pouvons réécrire le code l'encart 15-6 pour utiliser une `Box<T>` au lieu d'une référence ; l'opérateur de déréférencement devrait fonctionner comme montré dans l'encart 15-7 :

Fichier : src/main.rs

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Encart 15-7 : utilisation de l'opérateur de déréférencement sur un `Box<i32>`

La principale différence entre l'encart 15-7 et l'encart 15-6 est qu'ici nous avons fait en sorte que `y` soit une instance de boite qui pointe sur une copie de la valeur de `x` plutôt qu'avoir une référence vers la valeur de `x`. Dans la dernière assertion, nous pouvons utiliser l'opérateur de déréférencement pour suivre le pointeur de la boite de la même manière que nous l'avons fait lorsque `y` était une référence. Maintenant, nous allons regarder ce qu'il y a de si spécial dans `Box<T>` qui nous permet d'utiliser l'opérateur de déréférencement en définissant notre propre type de boite.

Définir notre propre pointeur intelligent

Construisons un pointeur intelligent similaire au type `Box<T>` fourni par la bibliothèque standard pour apprendre comment les pointeurs intelligents se comportent différemment des références classiques. Ensuite nous regarderons comment lui ajouter la possibilité d'utiliser l'opérateur de déréférencement.

Le type `Box<T>` est essentiellement défini comme étant une structure de tuple d'un seul élément, donc l'encart 15-8 définit un type `MaBoite<T>` de la même manière. Nous allons aussi définir une fonction `new` pour correspondre à la fonction `new` définie sur `Box<T>`.

Fichier : src/main.rs

```
struct MaBoite<T>(T);

impl<T> MaBoite<T> {
    fn new(x: T) -> MaBoite<T> {
        MaBoite(x)
    }
}
```

Encart 15-8 : définition du type `MaBoite<T>`

Nous définissons une structure `MaBoite` et on déclare un paramètre générique `T`, car nous souhaitons que notre type stocke des valeurs de n'importe quel type. Le type `MaBoite` est

une structure de tuple avec un seul élément de type `T`. La fonction `MaBoite::new` prend un paramètre de type `T` et retourne une instance `MaBoite` qui stocke la valeur qui lui est passée.

Essayons d'ajouter la fonction `main` de l'encart 15-7 dans l'encart 15-8 et la modifier pour utiliser le type `MaBoite<T>` que nous avons défini à la place de `Box<T>`. Le code de l'encart 15-9 ne se compile pas car Rust ne sait pas comment déréférencer `MaBoite`.

Fichier : src/main.rs

```
fn main() {
    let x = 5;
    let y = MaBoite::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Encart 15-9 : tentative d'utiliser `MaBoite<T>` de la même manière que nous avions utilisé les références et `Box<T>`

Voici l'erreur de compilation qui en résulte :

```
$ cargo run
   Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0614]: type `MaBoite<{integer}>` cannot be dereferenced
  --> src/main.rs:14:19
   |
14 |     assert_eq!(5, *y);
   |                   ^^

For more information about this error, try `rustc --explain E0614`.
error: could not compile `deref-example` due to previous error
```

Notre type `MaBoite<T>` ne peut pas être déréférencée car nous n'avons pas implémenté cette fonctionnalité sur notre type. Pour permettre le déréférencement avec l'opérateur `*`, nous devons implémenter le trait `Deref`.

Considérer un type comme une référence en implémentant le trait `Deref`

Comme nous l'avons vu dans une section du chapitre 10, pour implémenter un trait, nous devons fournir les implémentations des méthodes nécessaires pour ce trait. Le trait `Deref`, fourni par la bibliothèque standard, nécessite que nous implémentions une méthode `deref` qui emprunte `self` et retourne une référence vers la donnée interne. L'encart 15-10

contient une implémentation de `Deref` à ajouter à la définition de `MaBoite` :

Fichier : src/main.rs

```
use std::ops::Deref;

impl<T> Deref for MaBoite<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

Encart 15-10 : implémentation de `Deref` sur `MaBoite<T>`

La syntaxe `type Target = T;` définit un type associé pour le trait `Deref` à utiliser. Les types associés sont une manière légèrement différente de déclarer un paramètre générique, mais vous n'avez pas à vous préoccuper d'eux pour le moment ; nous les verrons plus en détail au chapitre 19.

Nous renseignons le corps de la méthode `deref` avec `&self.0` afin que `deref` retourne une référence vers la valeur que nous souhaitons accéder avec l'opérateur `*` . Rappellez-vous de la section du chapitre 5 où nous avons appris que le `.0` accède à la première valeur d'une structure tuple. La fonction `main` de l'encart 15-9 qui appelle `*` sur la valeur `MaBoite<T>` se compile désormais, et le `assert` réussit aussi !

Sans le trait `Deref` , le compilateur peut seulement déréférencer des références `&` . La méthode `deref` donne la possibilité au compilateur d'obtenir la valeur de n'importe quel type qui implémente `Deref` en appelant la méthode `deref` pour obtenir une référence `&` qu'il sait comment déréférencer.

Lorsque nous avons précisé `*y` dans l'encart 15-9, Rust fait tourner ce code en coulisses :

```
*(y.deref())
```

Rust remplace l'opérateur `*` par un appel à la méthode `deref` suivi par un simple déréférencement afin que nous n'ayons pas à nous demander si nous devons ou non appeler la méthode `deref` . Cette fonctionnalité de Rust nous permet d'écrire du code qui fonctionne de manière identique que nous ayons une référence classique ou un type qui implémente `Deref` .

La raison pour laquelle la méthode `deref` retourne une référence à une valeur, et que le déréférencement du tout dans les parenthèses externes de `*(y.deref())` reste nécessaire,

est le système de possession. Si la méthode `deref` retournait la valeur directement au lieu d'une référence à cette valeur, la valeur serait *déplacée* à l'extérieur de `self`. Nous ne souhaitons pas prendre possession de la valeur à l'intérieur de `MaBoite<T>` dans ce cas ainsi que la plupart des cas où nous utilisons l'opérateur de déréférencement.

Notez que l'opérateur `*` est remplacé par un appel à la méthode `deref` suivi par un appel à l'opérateur `*` une seule fois, à chaque fois que nous utilisons un `*` dans notre code. Comme la substitution de l'opérateur `*` ne s'effectue pas de manière récursive et infinie, nous récupèrerons une donnée de type `i32`, qui correspond au `5` du `assert_eq!` de l'encart 15-9.

Extrapolation de déréférencement implicite avec les fonctions et les méthodes

L'*extrapolation de déréférencement* est une commodité que Rust applique sur les arguments des fonctions et des méthodes. L'extrapolation de déréférencement fonctionne uniquement avec un type qui implémente le trait `Deref`. L'extrapolation de déréférencement convertit une référence vers ce type en une référence vers un autre type. Par exemple, l'extrapolation de déréférencement peut convertir `&String` en `&str` car `String` implémente le trait `Deref` de sorte qu'il puisse retourner `&str`. L'extrapolation de déréférencement s'applique automatiquement lorsque nous passons une référence vers une valeur d'un type particulier en argument d'une fonction ou d'une méthode qui ne correspond pas à ce type de paramètre dans la définition de la fonction ou de la méthode. Une série d'appels à la méthode `deref` convertit le type que nous donnons dans le type que le paramètre nécessite.

L'extrapolation de déréférencement a été ajoutée à Rust afin de permettre aux développeurs d'écrire des appels de fonctions et de méthodes qui n'ont pas besoin d'indiquer explicitement les références et les déréférencements avec `&` et `*`. La fonctionnalité d'extrapolation de déréférencement nous permet aussi d'écrire plus de code qui peut fonctionner à la fois pour les références et pour les pointeurs intelligents.

Pour voir l'extrapolation de déréférencement en action, utilisons le type `MaBoite<T>` que nous avons défini dans l'encart 15-8 ainsi que l'implémentation de `Deref` que nous avons ajoutée dans l'encart 15-10. L'encart 15-11 montre la définition d'une fonction qui a un paramètre qui est une slice de chaîne de caractères :

Fichier : src/main.rs

```
fn saluer(nom: &str) {
    println!("Salutations, {} !", nom);
}
```

Encart 15-11 : une fonction `saluer` qui prend en paramètre `nom` du type `&str`

Nous pouvons appeler la fonction `saluer` avec une slice de chaîne de caractères en argument, comme par exemple `saluer("Rust");`. L'extrapolation de déréférencement rend possible l'appel de `saluer` avec une référence à une valeur du type `MaBoite<String>`, comme dans l'encart 15-12 :

Fichier : src/main.rs

```
fn main() {
    let m = MaBoite::new(String::from("Rust"));
    saluer(&m);
}
```

Encart 15-12 : appel à `saluer` avec une référence à une valeur du type `MaBoite<String>`, qui fonctionne grâce à l'extrapolation de déréférencement

Ici nous appelons la fonction `saluer` avec l'argument `&m`, qui est une référence vers une valeur de type `MaBoite<String>`. Comme nous avons implémenté le trait `Deref` sur `MaBoite<T>` dans l'encart 15-10, Rust peut transformer le `&MaBoite<String>` en `&String` en appelant `deref`. La bibliothèque standard fournit une implémentation de `Deref` sur `String` qui retourne une slice de chaîne de caractères, comme expliqué dans la documentation de l'API de `Deref`. Rust appelle à nouveau `deref` pour transformer le `&String` en `&str`, qui correspond à la définition de la fonction `saluer`.

Si Rust n'avait pas implémenté l'extrapolation de déréférencement, nous aurions dû écrire le code de l'encart 15-13 au lieu du code de l'encart 15-12 pour appeler `saluer` avec une valeur du type `&MaBoite<String>`.

Fichier : src/main.rs

```
fn main() {
    let m = MaBoite::new(String::from("Rust"));
    saluer(&(*m)[..]);
}
```

Encart 15-13 : le code que nous aurions dû écrire si Rust n'avait pas d'extrapolation de déréférencement

Le `(*m)` déréférence la `MaBoite<String>` en une `String`. Ensuite le `&` et le `[..]` créent

une slice de chaîne de caractères à partir de la `String` qui est égale à l'intégralité du contenu de la `String`, ceci afin de correspondre à la signature de `saluer`. Le code sans l'extrapolation de déréférencement est bien plus difficile à lire, écrire et comprendre avec la présence de tous ces symboles. L'extrapolation de déréférencement permet à Rust d'automatiser ces convertions pour nous.

Lorsque le trait `Deref` est défini pour les types concernés, Rust va analyser les types et utiliser `Deref::deref` autant de fois que nécessaire pour obtenir une référence qui correspond au type du paramètre. Le nombre de fois qu'il est nécessaire d'insérer `Deref::deref` est résolu au moment de la compilation, ainsi il n'y a pas de surcoût au moment de l'exécution pour bénéficier de l'extrapolation de déréférencement !

L'interaction de l'extrapolation de déréférencement avec la mutabilité

De la même manière que vous pouvez utiliser le trait `Deref` pour remplacer le comportement de l'opérateur `*` sur les références immuables, vous pouvez utiliser le trait `DerefMut` pour remplacer le comportement de l'opérateur `*` sur les références mutables.

Rust procède à l'extrapolation de déréférencement lorsqu'il trouve des types et des implémentations de traits dans trois cas :

- Passer de `&T` à `&U` lorsque `T: Deref<Target=U>`
- Passer de `&mut T` à `&mut U` lorsque `T: DerefMut<Target=U>`
- Passer de `&mut T` à `&U` lorsque `T: Deref<Target=U>`

Les deux premiers cas sont exactement les mêmes, sauf pour la mutabilité. Le premier cas signifie que si vous avez un `&T` et que `T` implémente `Deref` pour le type `U`, vous pouvez obtenir un `&U` de manière transparente. Le deuxième cas signifie que la même extrapolation de déréférencement se déroule pour les références mutables.

Le troisième cas est plus ardu : Rust va aussi procéder à une extrapolation de déréférencement d'une référence mutable vers une référence immuable. Mais l'inverse n'est *pas* possible: une extrapolation de déréférencement d'une valeur immuable ne donnera jamais une référence mutable. A cause des règles d'emprunt, si vous avez une référence mutable, cette référence mutable doit être la seule référence vers cette donnée (autrement, le programme ne peut pas être compilé). Convertir une référence mutable vers une référence immuable ne va jamais casser les règles d'emprunt. Convertir une référence immuable vers une référence mutable nécessite que la référence immuable initiale soit la seule référence immuable vers cette donnée, mais les règles d'emprunt ne garantissent pas cela. Rust ne peut donc pas déduire que la conversion d'une référence immuable vers une référence mutable est possible.

Exécuter du code lors du nettoyage avec le trait `Drop`

Le second trait important pour les pointeurs intelligents est `Drop`, qui vous permet de personnaliser ce qui se passe lorsqu'une valeur est en train de sortir d'une portée. Vous pouvez fournir une implémentation du trait `Drop` sur n'importe quel type, et le code que vous renseignez peut être utilisé pour libérer des ressources comme des fichiers ou des connections réseau. Nous présentons `Drop` dans le contexte des pointeurs intelligents car la fonctionnalité du trait `Drop` est quasiment systématiquement utilisée lorsque nous implémentons un pointeur intelligent. Par exemple, lorsqu'une `Box<T>` est libérée, elle va désallouer l'espace occupé sur le tas sur lequel la boite pointe.

Dans certains langages, le développeur doit appeler du code pour libérer la mémoire ou des ressources à chaque fois qu'il finit d'utiliser une instance ou un pointeur intelligent. S'il oublie de le faire, le système peut surcharger et planter. Avec Rust, vous pouvez renseigner du code qui sera exécuté à chaque fois qu'une valeur sort de la portée, et le compilateur va insérer automatiquement ce code. Au final, vous n'avez pas besoin de concentrer votre attention à placer du code de nettoyage à chaque fois qu'une instance d'un type particulier n'est plus utilisée — vous ne risquez pas d'avoir des fuites de ressources !

Vous renseignez le code à exécuter lorsqu'une valeur sort de la portée en implémentant le trait `Drop`. Le trait `Drop` nécessite que vous implémentiez une méthode `drop` qui prend en paramètre une référence mutable à `self`. Pour voir quand Rust appelle `drop`, implémentons `drop` avec une instruction `println!` à l'intérieur, pour le moment.

L'encart 15-14 montre une structure `PointeurPerso` dont la seule fonctionnalité personnalisée est qu'elle va écrire `Nettoyage d'un PointeurPerso !` lorsque l'instance sort de la portée. Cet exemple signale quand Rust exécute la fonction `drop`.

Fichier : src/main.rs

```
struct PointeurPerso {
    donnee: String,
}

impl Drop for PointeurPerso {
    fn drop(&mut self) {
        println!("Nettoyage d'un PointeurPerso avec la donnée `{}` !",
self.donnee);
    }
}

fn main() {
    let c = PointeurPerso {
        donnee: String::from("des trucs"),
    };
    let d = PointeurPerso {
        donnee: String::from("d'autres trucs"),
    };
    println!("PointeurPersos créés.");
}
```

Encart 15-14 : Une structure `PointeurPerso` qui implémente le trait `Drop` dans lequel nous plaçons notre code de nettoyage

Le trait `Drop` est importé dans l'étape préliminaire, donc nous n'avons pas besoin de l'importer dans la portée. Nous implémentons le trait `Drop` sur `PointeurPerso` et nous fournissons une implémentation de la méthode `drop` qui appelle `println!`. Le corps de la fonction `drop` est l'endroit où vous placez la logique que vous souhaitez exécuter lorsqu'une instance du type concerné sort de la portée. Ici nous affichons un petit texte pour voir quand Rust appelle `drop`.

Dans le `main`, nous créons deux instances de `PointeurPerso` et ensuite on affiche `PointeurPersos créés`. A la fin du `main`, nos instances de `PointeurPerso` vont sortir de la portée, et Rust va appeler le code que nous avons placé dans la méthode `drop` et qui va afficher notre message final. Notez que nous n'avons pas besoin d'appeler explicitement la méthode `drop`.

Lorsque nous exécutons ce programme, nous devrions voir la sortie suivante :

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.60s
     Running `target/debug/drop-example`
PointeurPersos créés.
Nettoyage d'un PointeurPerso avec la donnée `d'autres trucs`!
Nettoyage d'un PointeurPerso avec la donnée `des trucs`!
```

Rust a appelé automatiquement `drop` pour nous lorsque nos instances sont sorties de la portée, appelant ainsi le code que nous y avions mis. Les variables sont libérées dans l'ordre inverse de leur création, donc `d` a été libéré avant `c`. Cet exemple vous fournit une illustration de la façon dont la méthode `drop` fonctionne ; normalement vous devriez y mettre le code de nettoyage dont votre type a besoin d'exécuter plutôt que d'afficher simplement un message.

Libérer prématurément une valeur avec `std::mem::drop`

Malheureusement, il n'est pas simple de désactiver la fonctionnalité automatique `drop`. La désactivation de `drop` n'est généralement pas nécessaire ; tout l'intérêt du trait `Drop` est qu'il est pris en charge automatiquement. Occasionnellement, cependant, vous pourriez avoir besoin de nettoyer prématurément une valeur. Un exemple est lorsque vous utilisez des pointeurs intelligents qui gèrent un système de verrouillage : vous pourriez vouloir forcer la méthode `drop` qui libère le verrou afin qu'un autre code dans la même portée puisse prendre ce verrou. Rust ne vous autorise pas à appeler manuellement la méthode `drop` du trait `Drop` ; à la place vous devez appeler la fonction `std::mem::drop`, fournie par la bibliothèque standard, si vous souhaitez forcer une valeur à être libérée avant la fin de sa portée.

Si nous essayons d'appeler manuellement la méthode `drop` du trait `Drop` en modifiant la fonction `main` de l'encart 15-14, comme dans l'encart 15-15, nous aurons une erreur de compilation :

Fichier : src/main.rs

```
fn main() {
    let c = PointeurPerso {
        donnee: String::from("des trucs"),
    };
    println!("PointeurPerso créé.");
    c.drop();
    println!("PointeurPerso libéré avant la fin du main.");
}
```

Encart 15-15 : tentative d'appel manuel de la méthode `drop` du trait `Drop` afin de nettoyer prématurément

Lorsque nous essayons de compiler ce code, nous obtenons l'erreur suivante :

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
error[E0040]: explicit use of destructor method
  --> src/main.rs:16:7
   |
16 |     c.drop();
   |     --^^^^--
   |     | |
   |     | explicit destructor calls not allowed
   |     help: consider using `drop` function: `drop(c)`

For more information about this error, try `rustc --explain E0040`.
error: could not compile `drop-example` due to previous error
```

Ce message d'erreur signifie que nous ne sommes pas autorisés à appeler explicitement `drop`. Le message d'erreur utilise le terme de *destructeur* (`destructor`) qui est un terme général de programmation qui désigne une fonction qui nettoie une instance. Un *destructeur* est analogue à un *constructeur*, qui construit une instance. La fonction `drop` en Rust est un destructeur particulier.

Rust ne nous laisse pas appeler explicitement `drop` car Rust appellera toujours automatiquement `drop` sur la valeur à la fin du `main`. Cela serait une erreur de *double libération* car Rust essayerait de nettoyer la même valeur deux fois.

Nous ne pouvons pas désactiver l'ajout automatique de `drop` lorsqu'une valeur sort de la portée, et nous ne pouvons pas désactiver explicitement la méthode `drop`. Donc, si nous avons besoin de forcer une valeur à être nettoyée prématurément, nous pouvons utiliser la fonction `std::mem::drop`.

La fonction `std::mem::drop` est différente de la méthode `drop` du trait `Drop`. Nous pouvons l'appeler en lui passant en argument la valeur que nous souhaitons libérer prématurément. La fonction est présente dans l'étape préliminaire, donc nous pouvons modifier `main` de l'encart 15-15 pour appeler la fonction `drop`, comme dans l'encart 15-16 :

Fichier : src/main.rs

```
fn main() {
    let c = PointeurPerso {
        donnee: String::from("des trucs"),
    };
    println!("PointeurPerso créé.");
    drop(c);
    println!("PointeurPerso libéré avant la fin du main.");
}
```

Encart 15-16 : appel à `std::mem::drop` pour libérer explicitement une valeur avant qu'elle

sorte de la portée

L'exécution de code va afficher ceci :

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/drop-example`
PointeurPerso créé.
Nettoyage d'un PointeurPerso avec la donnée `des trucs` !
PointeurPerso libéré avant la fin du main.
```

Le texte `Nettoyage d'un PointeurPerso avec la donnée `des trucs` !` est affiché entre `PointeurPerso créé` et `PointeurPerso libéré avant la fin du main`, ce qui démontre que la méthode `drop` a été appelée pour libérer `c` à cet endroit.

Vous pouvez utiliser le code renseigné dans une implémentation du trait `Drop` de plusieurs manières afin de rendre le nettoyage pratique et sûr : par exemple, vous pouvez l'utiliser pour créer votre propre alloueur de mémoire ! Grâce au trait `Drop` et le système de possession de Rust, vous n'avez pas à vous souvenir de nettoyer car Rust le fait automatiquement.

Vous n'avez pas non plus à vous soucier des problèmes résultant du nettoyage accidentel de valeurs toujours utilisées : le système de possession garantit que les références restent toujours en vigueur, et garantit également que `drop` n'est appelée qu'une seule fois lorsque la valeur n'est plus utilisée.

Maintenant que nous avons examiné `Box<T>` et certaines des caractéristiques des pointeurs intelligents, découvrons d'autres pointeurs intelligents définis dans la bibliothèque standard.

`Rc<T>`, le pointeur intelligent qui compte les références

Dans la majorité des cas, la possession est claire : vous savez exactement quelle variable possède une valeur donnée. Cependant, il existe des cas où une valeur peut être possédée par plusieurs propriétaires. Par exemple, dans des structures de données de graphes, plusieurs extrémités peuvent pointer vers le même noeud, et ce noeud est par conception possédé par toutes les extrémités qui pointent vers lui. Un noeud ne devrait pas être nettoyé, à moins qu'il n'ait plus d'extrémités qui pointent vers lui.

Pour permettre la possession multiple, Rust dispose du type `Rc<T>`, qui est une abréviation pour `Reference Counting` *(compteur de références)*. Le type `Rc<T>` assure le suivi du nombre de références vers une valeur, afin de déterminer si la valeur est toujours utilisée ou non. S'il y a zéro références vers une valeur, la valeur peut être nettoyée sans qu'aucune référence ne devienne invalide.

Imaginez que `Rc<T>` est comme une télévision dans une salle commune. Lorsqu'une personne entre pour regarder la télévision, elle l'allume. Une autre entre dans la salle et regarde la télévision. Lorsque la dernière personne quitte la salle, elle éteint la télévision car elle n'est plus utilisée. Si quelqu'un éteint la télévision alors que d'autres continuent à la regarder, cela va provoquer du chahut !

Nous utilisons le type `Rc<T>` lorsque nous souhaitons allouer une donnée sur le tas pour que plusieurs éléments de notre programme puissent la lire et que nous ne pouvons pas déterminer au moment de la compilation quel élément cessera de l'utiliser en dernier. Si nous savions quel élément finirait en dernier, nous pourrions simplement faire en sorte que cet élément prenne possession de la donnée, et les règles de possession classiques qui s'appliquent au moment de la compilation prendraient effet.

Notez que `Rc<T>` fonctionne uniquement dans des scénarios à un seul processus. Lorsque nous verrons la concurrence au chapitre 16, nous verrons comment procéder au comptage de références dans des programmes multi-processus.

Utiliser `Rc<T>` pour partager une donnée

Retournons à notre exemple de liste de construction de l'encart 15-5. Souvenez-vous que nous l'avons défini en utilisant `Box<T>`. Cette fois-ci, nous allons créer deux listes qui partagent toutes les deux la propriété d'une troisième liste. Théoriquement, cela ressemblera à l'illustration 15-3 :

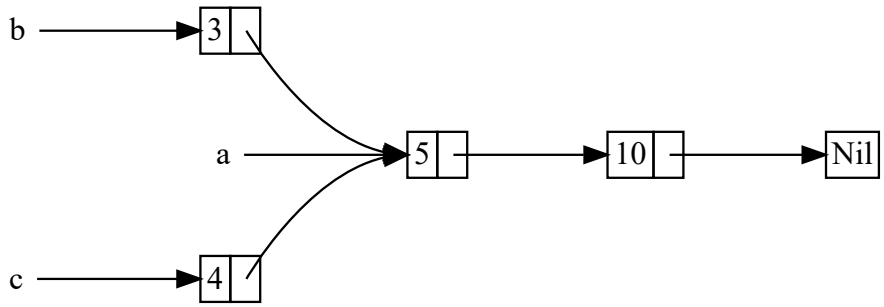


Illustration 15-3 : deux listes, `b` et `c`, qui se partagent la possession d'une troisième liste, `a`

Nous allons créer une liste `a` qui contient `5` et ensuite `10`. Ensuite, nous allons créer deux autres listes : `b` qui démarre avec `3` et `c` qui démarre avec `4`. Les deux listes `b` et `c` vont ensuite continuer sur la première liste `a` qui contient déjà `5` et `10`. Autrement dit, les deux listes vont se partager la première liste contenant `5` et `10`.

Si nous essayons d'implémenter ce scénario en utilisant les définitions de `List` avec `Box<T>`, comme dans l'encart 15-17, cela ne va pas fonctionner :

Fichier : src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

Encart 15-17 : démonstration que nous ne sommes pas autorisés à avoir deux listes qui utilisent `Box<T>` pour partager la propriété d'une troisième liste

Lorsque nous compilons ce code, nous obtenons cette erreur :

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0382]: use of moved value: `a`
  --> src/main.rs:11:30
   |
9  |     let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
   |         - move occurs because `a` has type `List`, which does not implement
the `Copy` trait
10 |     let b = Cons(3, Box::new(a));
   |                              - value moved here
11 |     let c = Cons(4, Box::new(a));
   |                              ^ value used here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `cons-list` due to previous error
```

Les variantes `Cons` prennent possession des données qu'elles obtiennent, donc lorsque nous avons créé la liste `b`, `a` a été déplacée dans `b` et `b` possède désormais `a`. Ensuite, lorsque nous essayons d'utiliser `a` à nouveau lorsque nous créons `c`, nous ne sommes pas autorisés à le faire car `a` a été déplacé.

Nous pourrions changer la définition de `Cons` pour stocker des références à la place, mais ensuite nous aurions besoin de renseigner des paramètres de durée de vie. En renseignant les paramètres de durée de vie, nous devrions préciser que chaque élément dans la liste vivra au moins aussi longtemps que la liste entière. C'est le cas pour les éléments et les listes dans l'encart 15-17, mais pas dans tous les cas.

A la place, nous allons changer la définition de `List` pour utiliser `Rc<T>` à la place de `Box<T>`, comme dans l'encart 15-18. Chaque variante `Cons` va maintenant posséder une valeur et un `Rc<T>` pointant sur une `List`. Lorsque nous créons `b`, au lieu de prendre possession de `a`, nous allons cloner le `Rc<List>` que `a` possède, augmentant ainsi le nombre de références de un à deux et permettant à `a` et `b` de partager la propriété des données dans `Rc<List>`. Nous allons aussi cloner `a` lorsque nous créons `c`, augmentant le nombre de références de deux à trois. Chaque fois que nous appelons `Rc::clone`, le compteur de références des données présentes dans le `Rc<List>` va augmenter, et les données ne seront pas nettoyées tant qu'il n'y aura pas zéro référence vers elles.

Filename : src/main.rs

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

Encart 15-18 : une définition de `List` qui utilise `Rc<T>`

Nous devons ajouter une instruction `use` pour importer `Rc<T>` dans la portée car il n'est pas présent dans l'étape préliminaire. Dans le `main`, nous créons la liste qui stocke `5` et `10` et la stocke dans une nouvelle `Rc<List>` dans `a`. Ensuite, lorsque nous créons `b` et `c`, nous appelons la fonction `Rc::clone` et nous passons une référence vers le `Rc<List>` de `a` en argument.

Nous aurions pu appeler `a.clone()` plutôt que `Rc::clone(&a)`, mais la convention en Rust est d'utiliser `Rc::clone` dans cette situation. L'implémentation de `Rc::clone` ne fait pas une copie profonde de toutes les données comme le fait la plupart des implémentations de `clone`. L'appel à `Rc:clone` augmente uniquement le compteur de références, ce qui ne prend pas beaucoup de temps. Les copies profondes des données peuvent prendre beaucoup de temps. En utilisant `Rc::clone` pour les compteurs de références, nous pouvons distinguer visuellement un clonage qui fait une copie profonde d'un clonage qui augmente uniquement le compteur de références. Lorsque vous enquêtez sur des problèmes de performances dans le code, vous pouvez ainsi écarter les appels à `Rc::clone` pour ne vous intéresser qu'aux clonages à copie profonde que vous recherchez probablement.

Cloner une `Rc<T>` augmente le compteur de référence

Changeons notre exemple de l'encart 15-18 pour que nous puissions voir le compteur de références changer au fur et à mesure que nous créons et libérons des références dans le `Rc<List>` présent dans `a`.

Dans l'encart 15-19, nous allons changer le `main` afin qu'il ait une portée en son sein autour de `c` ; ainsi nous pourrons voir comment le compteur de références change lorsque `c` sort de la portée.

Fichier : src/main.rs

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("compteur après la création de a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("compteur après la création de b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("compteur après la création de c = {}", Rc::strong_count(&a));
    }
    println!("compteur après que c est sorti de la portée = {}",
Rc::strong_count(&a));
}
```

Encart 15-19 : affichage du compteur de références

A chaque étape du programme où le compteur de références change, nous affichons le compteur de références, que nous pouvons obtenir en faisant appel à la fonction `Rc::strong_count`. Cette fonction s'appelle `strong_count` plutôt que `count` car le type `Rc<T>` a aussi un `weak_count` ; nous verrons à quoi sert ce `weak_count` dans la dernière section de ce chapitre.

Ce code affiche ceci :

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.45s
     Running `target/debug/cons-list`
compteur après la création de a = 1
compteur après la création de b = 2
compteur après la création de c = 3
compteur après que c est sorti de la portée = 2
```

Nous pouvons voir clairement que le `Rc<List>` dans `a` a un compteur de références initial à `1` ; puis à chaque fois que nous appelons `clone`, le compteur augmente de 1. Nous n'avons pas à appeler une fonction pour réduire le compteur de références, comme nous avons dû le faire avec `Rc::clone` pour augmenter compteur : l'implémentation du trait `Drop` réduit le compteur de références automatiquement lorsqu'une valeur de `Rc<T>` sort de la portée.

Ce que nous ne voyons pas dans cet exemple, c'est que lorsque `b` et `a` sortent de la portée à la fin du `main`, le compteur vaut alors 0 et que le `Rc<List>` est nettoyé complètement à ce moment. L'utilisation de `Rc<T>` permet à une valeur d'avoir plusieurs propriétaires, et le compteur garantit que la valeur reste en vigueur tant qu'au moins un propriétaire existe encore.

Grâce aux références immuables, `Rc<T>` vous permet de partager des données entre plusieurs éléments de votre programme pour uniquement les lire. Si `Rc<T>` vous avait aussi permis d'avoir des références mutables, vous auriez alors violé une des règles d'emprunt vues au chapitre 4 : les emprunts mutables multiples à une même donnée peuvent causer des accès concurrents et des incohérences. Cependant, pouvoir modifier des données reste très utile ! Dans la section suivante, nous allons voir le motif de mutabilité interne et le type `RefCell<T>` que vous pouvez utiliser conjointement avec un `Rc<T>` pour pouvoir travailler avec cette contrainte d'immuabilité.

`RefCell<T>` et le motif de mutabilité interne

La *mutabilité interne* est un motif de conception en Rust qui vous permet de muter une donnée même s'il existe des références immuables ; normalement, cette action n'est pas autorisée par les règles d'emprunt. Pour muter des données, le motif utilise du code `unsafe` dans une structure de données pour contourner les règles courantes de Rust qui gouvernent la mutation et l'emprunt. Nous n'avons pas encore parlé du code unsafe ; nous le ferons au chapitre 19. Nous pouvons utiliser des types qui utilisent le motif de mutabilité interne lorsque nous pouvons être sûr que les règles d'emprunt seront suivies au moment de l'exécution, même si le compilateur ne peut pas en être sûr. Le code `unsafe` concerné est ensuite incorporé dans une API sûre, et le type externe reste immuable.

Découvrons ce concept en examinant le type `RefCell<T>` qui applique le motif de mutabilité interne.

Appliquer les règles d'emprunt au moment de l'exécution avec `RefCell<T>`

Contrairement à `Rc<T>`, le type `RefCell<T>` représente une propriété unique de la donnée qu'il contient. Qu'est-ce qui rend donc `RefCell<T>` différent d'un type comme `Box<T>` ? Souvenez-vous des règles d'emprunt que vous avez apprises au chapitre 4 :

- A un instant donné, vous pouvez avoir *soit* (mais pas les deux) une référence mutable, soit n'importe quel nombre de références immuables
- Les références doivent toujours être en vigueur.

Avec les références et `Box<T>`, les règles d'emprunt obligatoires sont appliquées au moment de la compilation. Avec `RefCell<T>`, ces obligations sont appliquées *au moment de l'exécution*. Avec les références, si vous ne respectez pas ces règles, vous allez obtenir une erreur de compilation. Avec `RefCell<T>`, si vous ne les respectez pas, votre programme va paniquer et se fermer.

Les avantages de vérifier les règles d'emprunt au moment de la compilation est que les erreurs vont se produire plus tôt dans le processus de développement et qu'il n'y a pas d'impact sur les performances à l'exécution car toute l'analyse a déjà été faite au préalable. Pour ces raisons, la vérification des règles d'emprunt au moment de compilation est le meilleur choix à faire dans la majorité des cas, ce qui explique pourquoi c'est le choix par défaut de Rust.

L'avantage de vérifier les règles d'emprunt plutôt à l'exécution est que cela permet certains scénarios qui restent sûrs pour la mémoire, bien qu'interdits à cause des vérifications à la

compilation. L'analyse statique, comme le compilateur Rust, est de nature prudente. Certaines propriétés du code sont impossibles à détecter en analysant le code : l'exemple le plus connu est le *problème de l'arrêt*, qui dépasse le cadre de ce livre mais qui reste un sujet intéressant à étudier.

Comme certaines analyses sont impossibles, si le compilateur Rust ne peut pas s'assurer que le code respecte les règles d'emprunt, il risque de rejeter un programme valide ; dans ce sens, il est prudent. Si Rust accepte un programme incorrect, les utilisateurs ne pourront pas avoir confiance dans les garanties qu'apporte Rust. Cependant, si Rust rejette un programme valide, le développeur sera importuné, mais rien de catastrophique ne va se passer. Le type `RefCell<T>` est utile lorsque vous êtes sûr que votre code suit bien les règles d'emprunt mais que le compilateur est incapable de comprendre et de garantir cela.

De la même manière que `Rc<T>`, `RefCell<T>` sert uniquement pour des scénarios à une seule tâche et va vous donner une erreur à la compilation si vous essayez de l'utiliser dans un contexte multitâches. Nous verrons comment bénéficier des fonctionnalités de `RefCell<T>` dans un programme multi-processus au chapitre 16.

Voici un résumé des raisons de choisir `Box<T>`, `Rc<T>` ou `RefCell<T>` :

- `Rc<T>` permet d'avoir plusieurs propriétaires pour une même donnée ; `Box<T>` et `RefCell<T>` n'ont qu'un seul propriétaire.
- `Box<T>` permet des emprunts immuables ou mutables à la compilation ; `Rc<T>` permet uniquement des emprunts immuables, vérifiés à la compilation ; `RefCell<T>` permet des emprunts immuables ou mutables, vérifiés à l'exécution.
- Comme `RefCell<T>` permet des emprunts mutables, vérifiés à l'exécution, vous pouvez muter la valeur à l'intérieur du `RefCell<T>` même si le `RefCell<T>` est immuable.

Modifer une valeur à l'intérieur d'une valeur immuable est ce qu'on appelle le motif de *mutabilité interne*. Découvrons une situation pour laquelle la mutabilité interne s'avère utile, puis examinons comment cela est rendu possible.

Mutabilité interne : un emprunt mutable d'une valeur immuable

Une des conséquences des règles d'emprunt est que lorsque vous avez une valeur immuable, vous ne pouvez pas emprunter sa mutabilité. Par exemple, ce code ne va pas se compiler :

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

Si vous essayez de compiler ce code, vous allez obtenir l'erreur suivante :

```
$ cargo run
   Compiling borrowing v0.1.0 (file:///projects/borrowing)
error[E0596]: cannot borrow `x` as mutable, as it is not declared as mutable
 --> src/main.rs:3:13
  |
2 |     let x = 5;
  |         - help: consider changing this to be mutable: `mut x`
3 |     let y = &mut x;
  |             ^^^^^^ cannot borrow as mutable

For more information about this error, try `rustc --explain E0596`.
error: could not compile `borrowing` due to previous error
```

Cependant, il existe des situations pour lesquelles il serait utile qu'une valeur puisse se modifier elle-même dans ses propres méthodes mais qui semble être immuable pour le reste du code. Le code à l'extérieur des méthodes de la valeur n'est pas capable de modifier la valeur. L'utilisation de `RefCell<T>` est une manière de pouvoir procéder à des mutations internes. Mais `RefCell<T>` ne contourne pas complètement les règles d'emprunt : le vérificateur d'emprunt du compilateur permet cette mutabilité interne, et les règles d'emprunt sont plutôt vérifiées à l'exécution. Si vous violez les règles, vous allez provoquer un `panic!` plutôt que d'avoir une erreur de compilation.

Voyons un exemple pratique dans lequel nous pouvons utiliser `RefCell<T>` pour modifier une valeur immuable et voir en quoi cela est utile.

Un cas d'utilisation de la mutabilité interne : le mock object

Un *double de test* est un concept général de programmation consistant à utiliser un type à la place d'un autre pendant des tests. Un *mock object* est un type particulier de double de test qui enregistre ce qui se passe lors d'un test afin que vous puissiez vérifier que les actions se sont passées correctement.

Rust n'a pas d'objets au sens où l'entendent les autres langages qui en ont, et Rust n'offre pas non plus de fonctionnalité de mock object dans la bibliothèque standard comme le font d'autres langages. Cependant, vous pouvez très bien créer une structure qui va répondre aux mêmes besoins qu'un mock object.

Voici le scénario que nous allons tester : nous allons créer une bibliothèque qui surveillera la

proximité d'une valeur par rapport à une valeur maximale et enverra des messages en fonction de cette limite. Par exemple, cette bibliothèque peut être utilisée pour suivre le quota d'un utilisateur afin de suivre le nombre d'appels aux API qu'il est autorisé à faire.

Notre bibliothèque fournira uniquement la fonctionnalité de suivi en fonction de la proximité d'une valeur avec la maximale et définiera quels seront les messages associés. Les applications qui utiliseront notre bibliothèque devront fournir un mécanisme pour envoyer les messages : l'application peut afficher le message dans l'application, l'envoyer par email, l'envoyer par SMS ou autre chose. La bibliothèque n'a pas à se charger de ce détail. Tout ce que ce mécanisme doit faire est d'implémenter un trait `Messager` que nous allons fournir. L'encart 15-20 propose le code pour cette bibliothèque :

Fichier : src/lib.rs

```
pub trait Messager {
    fn envoyer(&self, msg: &str);
}

pub struct TraqueurDeLimite<'a, T: Messager> {
    messager: &'a T,
    valeur: usize,
    max: usize,
}

impl<'a, T> TraqueurDeLimite<'a, T>
where
    T: Messager,
{
    pub fn new(messager: &T, max: usize) -> TraqueurDeLimite<T> {
        TraqueurDeLimite {
            messager,
            valeur: 0,
            max,
        }
    }

    pub fn set_valeur(&mut self, valeur: usize) {
        self.valeur = valeur;

        let pourcentage_du_maximum = self.valeur as f64 / self.max as f64;

        if pourcentage_du_maximum >= 1.0 {
            self.messager.envoyer("Erreur : vous avez dépassé votre quota !");
        } else if pourcentage_du_maximum >= 0.9 {
            self.messager
                .envoyer("Avertissement urgent : vous avez utilisé 90% de votre
quota !");
        } else if pourcentage_du_maximum >= 0.75 {
            self.messager
                .envoyer("Avertissement : vous avez utilisé 75% de votre
quota !");
        }
    }
}
```

Encart 15-20 : une bibliothèque qui suit la proximité d'une valeur avec une valeur maximale et avertit lorsque cette valeur atteint un certain seuil

La partie la plus importante de ce code est celle où le trait `Messager` a une méthode qui fait appel à `envoyer` en prenant une référence immuable à `self` ainsi que le texte du message. Ce trait est l'interface que notre mock object doit implémenter afin que le mock puisse être utilisé de la même manière que l'objet réel. L'autre partie importante est lorsque nous souhaitons tester le comportement de la méthode `set_valeur` sur le `TraqueurDeLimite`. Nous pouvons changer ce que nous envoyons dans le paramètre `valeur`, mais `set_valeur`

ne nous retourne rien qui nous permettrait de le vérifier. Nous voulons pouvoir dire que si nous créons un `TraqueurDeLimite` avec quelque chose qui implémente le trait `Messager` et une valeur précise pour `max`, lorsque nous passons différents nombres pour `valeur`, le messager reçoit bien l'instruction d'envoyer les messages correspondants.

Nous avons besoin d'un mock object qui, au lieu d'envoyer un email ou un SMS lorsque nous faisons appel à `envoyer`, va seulement enregistrer les messages qu'on lui demande d'envoyer. Nous pouvons créer une nouvelle instance du mock object, créer un `TraqueurDeLimite` qui utilise le mock object, faire appel à la méthode `set_value` sur le `TraqueurDeLimite` et ensuite vérifier que le mock object a bien les messages que nous attendions. L'encart 15-21 montre une tentative d'implémentation d'un mock object qui fait ceci, mais le vérificateur d'emprunt ne nous autorise pas à le faire :

Fichier : src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;

    struct MessagerMock {
        messages_envoyes: Vec<String>,
    }

    impl MessagerMock {
        fn new() -> MessagerMock {
            MessagerMock {
                messages_envoyes: vec![],
            }
        }
    }

    impl Messager for MessagerMock {
        fn envoyer(&self, message: &str) {
            self.messages_envoyes.push(String::from(message));
        }
    }

    #[test]
    fn envoi_d_un_message_d_avertissement_superieur_a_75_pourcent() {
        let messager_mock = MessagerMock::new();
        let mut traqueur = TraqueurDeLimite::new(&messager_mock, 100);

        traqueur.set_valeur(80);

        assert_eq!(messager_mock.messages_envoyes.len(), 1);
    }
}
```

Encart 15-21 : une tentative d'implémentation d'un `MessagerMock` qui n'est pas autorisée par le vérificateur d'emprunt

Ce code de test définit une structure `MessagerMock` qui a un champ `messages_envoyes` qui est un `Vec` de valeurs `String`, afin d'y enregistrer les messages qui lui sont envoyés. Nous définissons également une fonction associée `new` pour faciliter la création de valeurs `MessagerMock` qui commencent avec une liste vide de messages. Nous implémentons ensuite le trait `Messager` sur `MessagerMock` afin de donner un `MessagerMock` à un `TraqueurDeLimite`. Dans la définition de la méthode `envoyer`, nous prenons le message envoyé en paramètre et nous le stockons dans la liste `messages_envoyes` du `MessagerMock`.

Dans le test, nous vérifions ce qui se passe lorsque le `TraqueurDeLimite` doit atteindre une valeur qui est supérieure à 75 pourcent de la valeur `max`. D'abord, nous créons un nouveau `MessagerMock`, qui va démarrer avec une liste vide de messages. Ensuite, nous créons un nouveau `TraqueurDeLimite` et nous lui donnons une référence vers ce `MessagerMock` et une valeur `max` de 100. Nous appelons la méthode `set_valeur` sur le `TraqueurDeLimite` avec une valeur de 80, qui est plus grande que 75 pourcents de 100. Enfin, nous vérifions que la liste de messages qu'a enregistrée le `MessagerMock` contient bien désormais un message.

Cependant, il reste un problème avec ce test, problème qui est montré ci-dessous :

```
$ cargo test
   Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
error[E0596]: cannot borrow `self.messages_envoyes` as mutable, as it is behind
a `&` reference
  --> src/lib.rs:58:13
   |
2  |     fn envoyer(&self, message: &str);
   |                ----- help: consider changing that to be a mutable
reference: `&mut self`
...
58 |             self.messages_envoyes.push(String::from(message));
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `self` is a
`&` reference, so the data it refers to cannot be borrowed as mutable

For more information about this error, try `rustc --explain E0596`.
error: could not compile `limit-tracker` due to previous error
warning: build failed, waiting for other jobs to finish...
error: build failed
```

Nous ne pouvons pas modifier le `MessagerMock` pour enregistrer les messages, car la méthode `envoyer` utilise une référence immuable à `self`. Nous ne pouvons pas non plus suivre la suggestion du texte d'erreur pour utiliser `&mut self` à la place, car ensuite la signature de `envoyer` ne va pas correspondre à la signature de la définition du trait

`Messager` (essayez et vous constaterez le message d'erreur que vous obtiendrez).

C'est une situation dans laquelle la mutabilité interne peut nous aider ! Nous allons stocker `messages_envoyes` dans une `RefCell<T>`, et ensuite la méthode `envoyer` pourra modifier `messages_envoyes` pour stocker les messages que nous avons avons vus. L'encart 15-22 montre à quoi cela peut ressembler :

Fichier : src/lib.rs

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::cell::RefCell;

    struct MessagerMock {
        messages_envoyes: RefCell<Vec<String>>,
    }

    impl MessagerMock {
        fn new() -> MessagerMock {
            MessagerMock {
                messages_envoyes: RefCell::new(vec![]),
            }
        }
    }

    impl Messager for MessagerMock {
        fn envoyer(&self, message: &str) {
            self.messages_envoyes.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn envoi_d_un_message_d_avertissement_superieur_a_75_pourcent() {
        // -- partie masquée ici --

        assert_eq!(messager_mock.messages_envoyes.borrow().len(), 1);
    }
}
```

Encart 15-22 : utilisation du `RefCell<T>` pour muter une valeur interne que les valeurs externes considèrent comme immuable

Le champ `messages_envoyes` est maintenant du type `RefCell<Vec<String>>` au lieu de `Vec<String>`. Dans la fonction `new`, nous créons une nouvelle instance de `RefCell<Vec<String>>` autour du vecteur vide.

En ce qui concerne l'implémentation de la méthode `envoyer`, le premier paramètre est

toujours un emprunt immuable de `self`, ce qui correspond à la définition du trait. Nous appelons la méthode `borrow_mut` sur le `RefCell<Vec<String>>` présent dans `self.messages_envoyes` pour obtenir une référence mutable vers la valeur présente dans le `RefCell<Vec<String>>`, qui correspond au vecteur. Ensuite, nous appelons `push` sur la référence mutable vers le vecteur pour enregistrer le message envoyé pendant le test.

Le dernier changement que nous devons appliquer se trouve dans la vérification : pour savoir combien d'éléments sont présents dans le vecteur, nous faisons appel à `borrow` de `RefCell<Vec<String>>` pour obtenir une référence immuable vers le vecteur.

Maintenant que vous avez appris à utiliser `RefCell<T>`, regardons comment il fonctionne !

Suivre les emprunts à l'exécution avec `RefCell<T>`

Lorsque nous créons des références immuables et mutables, nous utilisons respectivement les syntaxes `&` et `&mut`. Avec `RefCell<T>`, nous utilisons les méthodes `borrow` et `borrow_mut`, qui font partie de l'API stable de `RefCell<T>`. La méthode `borrow` retourne un pointeur intelligent du type `Ref<T>` et `borrow_mut` retourne un pointeur intelligent du type `RefMut<T>`. Les deux implémentent `Deref`, donc nous pouvons les considérer comme des références classiques.

Le `RefCell<T>` suit combien de pointeurs intelligents `Ref<T>` et `RefMut<T>` sont actuellement actifs. A chaque fois que nous faisons appel à `borrow`, le `RefCell<T>` augmente son compteur du nombre d'emprunts immuables qui existent. Lorsqu'une valeur `Ref<T>` sort de la portée, le compteur d'emprunts immuables est décrémenté de un. A tout moment `RefCell<T>` nous permet d'avoir plusieurs emprunts immuables ou bien un seul emprunt mutable, tout comme le font les règles d'emprunt au moment de la compilation.

Si nous ne respectons pas ces règles, l'implémentation de `RefCell<T>` va paniquer à l'exécution plutôt que de provoquer une erreur de compilation comme nous l'aurions eu en utilisant des références classiques. L'encart 15-23 nous montre une modification apportée à l'implémentation de `envoyer` de l'encart 15-22. Nous essayons délibérément de créer deux emprunts mutables actifs dans la même portée pour montrer que `RefCell<T>` nous empêche de faire ceci à l'exécution.

Fichier : src/lib.rs

```
    impl Messager for MessagerMock {
        fn envoyer(&self, message: &str) {
            let mut premier_emprunt = self.messages_envoyes.borrow_mut();
            let mut second_emprunt = self.messages_envoyes.borrow_mut();

            premier_emprunt.push(String::from(message));
            second_emprunt.push(String::from(message));
        }
    }
```

Encart 15-23 : création de deux références mutables dans la même portée pour voir si `RefCell<T>` va paniquer

Nous créons une variable `premier_emprunt` pour le pointeur intelligent `RefMut<T>` retourné par `borrow_mut`. Ensuite nous créons un autre emprunt de la même manière, qui s'appelle `second_emprunt`. Cela fait deux références mutables dans la même portée, ce qui n'est pas autorisé. Lorsque nous lançons les tests sur notre bibliothèque, le code de l'encart 15-23 va se compiler sans erreur, mais les tests vont échouer :

```
$ cargo test
   Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
    Finished test [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests (target/debug/deps/limit_tracker-e599811fa246dbde)

running 1 test
test tests::envoi_d_un_message_d_avertissement_superieur_a_75_pourcent ...
FAILED

failures:

---- tests::envoi_d_un_message_d_avertissement_superieur_a_75_pourcent stdout
----
thread 'main' panicked at 'already borrowed: BorrowMutError', src/lib.rs:60:53
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::envoi_d_un_message_d_avertissement_superieur_a_75_pourcent

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Remarquez que le code a paniqué avec le message `already borrowed: BorrowMutError` (NdT : `déjà emprunté`). C'est ainsi que `RefCell<T>` gère les violations des règles d'emprunt à l'exécution.

La détection des erreurs d'emprunt à l'exécution plutôt qu'à la compilation signifie que vous pourriez découvrir une erreur dans votre code plus tard dans le processus de développement et peut-être même pas avant que votre code ne soit déployé en production. De plus, votre code va subir une petite perte de performances à l'exécution en raison du contrôle des emprunts à l'exécution plutôt qu'à la compilation. Cependant, l'utilisation de `RefCell<T>` rend possible l'écriture d'un mock object qui peut se modifier lui-même afin d'enregistrer les messages qu'il a vu passer alors que vous l'utilisez dans un contexte où seules les valeurs immuables sont permises. Vous pouvez utiliser `RefCell<T>` malgré ses inconvénients pour obtenir plus de fonctionnalités que celles qu'offre une référence classique.

Permettre plusieurs propriétaires de données mutables en combinant `Rc<T>` et `RefCell<T>`

Il est courant d'utiliser `RefCell<T>` en tandem avec `Rc<T>`. Rappelez-vous que `Rc<T>` vous permet d'avoir plusieurs propriétaires d'une même donnée, mais qu'il ne vous donne qu'un seul accès immuable à cette donnée. Si vous avez un `Rc<T>` qui contient un `RefCell<T>`, vous pouvez obtenir une valeur qui peut avoir plusieurs propriétaires *et* que vous pouvez modifier !

Souvenez-vous de l'exemple de la liste de construction de l'encart 15-18 où nous avions utilisé `Rc<T>` pour permettre à plusieurs listes de se partager la possession d'une autre liste. Comme `Rc<T>` stocke seulement des valeurs immuables, nous ne pouvons changer aucune valeur dans la liste une fois que nous l'avons créée. Ajoutons un `RefCell<T>` pour pouvoir changer les valeurs dans les listes. L'encart 15-24 nous montre ceci en ajoutant un `RefCell<T>` dans la définition de `Cons`, nous pouvons ainsi modifier les valeurs stockées dans n'importe quelle liste :

Fichier : src/main.rs

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let valeur = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&valeur), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *valeur.borrow_mut() += 10;

    println!("a après les opérations = {:?}", a);
    println!("b après les opérations = {:?}", b);
    println!("c après les opérations = {:?}", c);
}
```

Encart 15-24 : utilisation de `Rc<RefCell<i32>>` pour créer une `List` que nous pouvons modifier

Nous créons une valeur qui est une instance de `Rc<RefCell<i32>>` et nous la stockons dans une variable `valeur` afin que nous puissions y avoir accès plus tard. Ensuite, nous créons une `List` dans `a` avec une variante de `Cons` qui utilise `valeur`. Nous devons utiliser clone sur `valeur` afin que `a` et `valeur` soient toutes les deux propriétaires de la valeur interne `5` plutôt que d'avoir à transférer la possession de `valeur` à `a` ou avoir `a` qui emprunte `valeur`.

Nous insérons la liste `a` dans un `Rc<T>` pour que, lorsque nous créons `b` et `c`, elles puissent toutes les deux utiliser `a`, ce que nous avions déjà fait dans l'encart 15-18.

Après avoir créé les listes dans `a`, `b`, et `c`, nous ajoutons 10 à la valeur dans `valeur`. Nous faisons cela en appelant `borrow_mut` sur `valeur`, ce qui utilise la fonctionnalité de déréférencement automatique que nous avons vue au chapitre 5 (voir la section “Où est l'opérateur -> ?”) pour déréférencer le `Rc<T>` dans la valeur interne `RefCell<T>`. La méthode `borrow_mut` retourne un pointeur intelligent `RefMut<T>`, et nous utilisons l'opérateur de déréférencement sur lui pour changer sa valeur interne.

Lorsque nous affichons `a`, `b` et `c`, nous pouvons constater qu'elles ont toutes la valeur

modifiée de 15 au lieu de 5 :

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.63s
     Running `target/debug/cons-list`
a après les opérations = Cons(RefCell { value: 15 }, Nil)
b après les opérations = Cons(RefCell { value: 3 }, Cons(RefCell { value: 15 },
Nil))
c après les opérations = Cons(RefCell { value: 4 }, Cons(RefCell { value: 15 },
Nil))
```

Cette technique est plutôt ingénieuse ! En utilisant `RefCell<T>`, nous avons une valeur `List` qui est immuable de l'extérieur. Mais nous pouvons utiliser les méthodes de `RefCell<T>` qui nous donne accès à sa mutabilité interne afin que nous puissions modifier notre donnée lorsque nous en avons besoin. Les vérifications des règles d'emprunt à l'exécution nous protègent des accès concurrents, et il est parfois intéressant de sacrifier un peu de vitesse pour cette flexibilité dans nos structures de données.

La bibliothèque standard a d'autres types qui fournissent de la mutabilité interne, comme `Cell<T>`, qui est similaire sauf qu'au lieu de fournir des références à la valeur interne, la valeur est copiée à l'intérieur et à l'extérieur du `Cell<T>`. Il existe aussi `Mutex<T>` qui offre de la mutabilité interne qui est sécurisée pour une utilisation partagée entre plusieures tâches ; nous allons voir son utilisation au chapitre 16. Plongez-vous dans la documentation de la bibliothèque standard pour plus de détails entre ces différents types.

Les boucles de références qui peuvent provoquer des fuites de mémoire

Les garanties de sécurité de la mémoire de Rust rendent difficile, mais pas impossible, la création accidentelle de mémoire qui n'est jamais nettoyée (aussi appelée *fuite de mémoire*). Eviter totalement les fuites de mémoire n'est pas une des garanties de Rust, en tout cas pas comme pour l'accès concurrent au moment de la compilation, ce qui signifie que les fuites de mémoire sont sans risque pour la mémoire avec Rust. Nous pouvons constater que Rust permet les fuites de mémoire en utilisant `Rc<T>` et `RefCell<T>` : il est possible de créer des références où les éléments se réfèrent entre eux de manière cyclique. Cela crée des fuites de mémoire car le compteur de références de chaque élément dans la boucle de références ne vaudra jamais 0, et les valeurs ne seront jamais libérées.

Créer une boucle de références

Voyons comment une boucle de références peut exister et comment l'éviter, en commençant par la définition de l'énumération `List` et la méthode `parcourir` de l'encart 15-25 :

Fichier : src/main.rs

```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn parcourir(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}

fn main() {}
```

Encart 15-25 : une liste de construction qui stocke une `RefCell<T>` pour que nous puissions modifier ce sur quoi une variante `Cons` pointe

Nous utilisons une autre variation de la définition de `List` de l'encart 15-5. Le second élément dans la variante `Cons` est maintenant un `RefCell<Rc<List>>`, ce qui signifie qu'au lieu de pouvoir modifier la valeur `i32` comme nous l'avions fait dans l'encart 15-24, nous modifions ce sur quoi une variante `Cons` pointe (qui reste une valeur `List`). Nous ajoutons également une méthode `parcourir` pour nous faciliter l'accès au second élément si nous avons une variante `Cons`.

Dans l'encart 15-26, nous ajoutons une fonction `main` qui utilise les définitions de l'encart 15-25. Ce code crée une liste dans `a` et une liste dans `b` qui pointe sur la liste de `a`. Ensuite, on modifie la liste de `a` pour pointer sur `b`, ce qui crée une boucle de références. Il y a aussi des instructions `println!` tout du long pour montrer la valeur des compteurs de références à différents endroits du processus.

Fichier : src/main.rs

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("compteur initial de a = {}", Rc::strong_count(&a));
    println!("prochain élément de a = {:?}", a.parcourir());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("compteur de a après création de b = {}", Rc::strong_count(&a));
    println!("compteur initial de b = {}", Rc::strong_count(&b));
    println!("prochain élément de b = {:?}", b.parcourir());

    if let Some(lien) = a.parcourir() {
        *lien.borrow_mut() = Rc::clone(&b);
    }

    println!("compteur de b après avoir changé a = {}", Rc::strong_count(&b));
    println!("compteur de a après avoir changé a = {}", Rc::strong_count(&a));

    // Décommentez la ligne suivante pour constater que nous sommes dans
    // une boucle de références, cela fera déborder la pile
    // println!("prochain élément de a = {:?}", a.parcourir());
}
```

Encart 15-26 : création d'une boucle de références de deux valeurs `List` qui se pointent mutuellement dessus

Nous créons une instance `Rc<List>` qui stocke une valeur `List` dans la variable `a` avec une valeur initiale de `5, Nil`. Nous créons ensuite une instance `Rc<List>` qui stocke une autre valeur `List` dans la variable `b` qui contient la valeur 10 et pointe vers la liste dans `a`.

Nous modifions `a` afin qu'elle pointe sur `b` au lieu de `Nil`, ce qui crée une boucle. Nous faisons ceci en utilisant la méthode `parcourir` pour obtenir une référence au `RefCell<Rc<List>>` présent dans `a`, que nous plaçons dans la variable `lien`. Ensuite nous utilisons la méthode `borrow_mut` sur le `RefCell<Rc<List>>` pour remplacer la valeur actuellement présente en son sein, la `Rc<List>` contenant `Nil`, par la `Rc<List>` présente dans `b`.

Lorsque nous exécutons ce code, en gardant le dernier `println!` commenté pour le moment, nous obtenons ceci :

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.53s
     Running `target/debug/cons-list`
compteur initial de a = 1
prochain élément de a = Some(RefCell { value: Nil })
compteur de a après création de b = 2
compteur initial de b = 1
prochain élément de b = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
compteur de b après avoir changé a = 2
compteur de a après avoir changé a = 2
```

Les compteurs de références des instances de `Rc<List>` valent tous les deux 2 pour `a` et `b` après avoir modifié `a` pour qu'elle pointe sur `b`. A la fin du `main`, Rust nettoie d'abord la variable `b`, ce qui décrémente le compteur de références dans l'instance `Rc<List>` de 2 à 1. La mémoire utilisée sur le tas par `Rc<List>` ne sera pas libérée à ce moment, car son compteur de références est à 1, et non pas 0. Puis, Rust libère `a`, ce qui décrémente le compteur `a` de références `Rc<List>` de 2 à 1, également. La mémoire de cette instance ne peut pas non plus être libérée car l'autre instance `Rc<List>` y fait toujours référence. La mémoire alouée à la liste ne sera jamais libérée. Pour représenter cette boucle de références, nous avons créé un diagramme dans l'illustration 15-4.

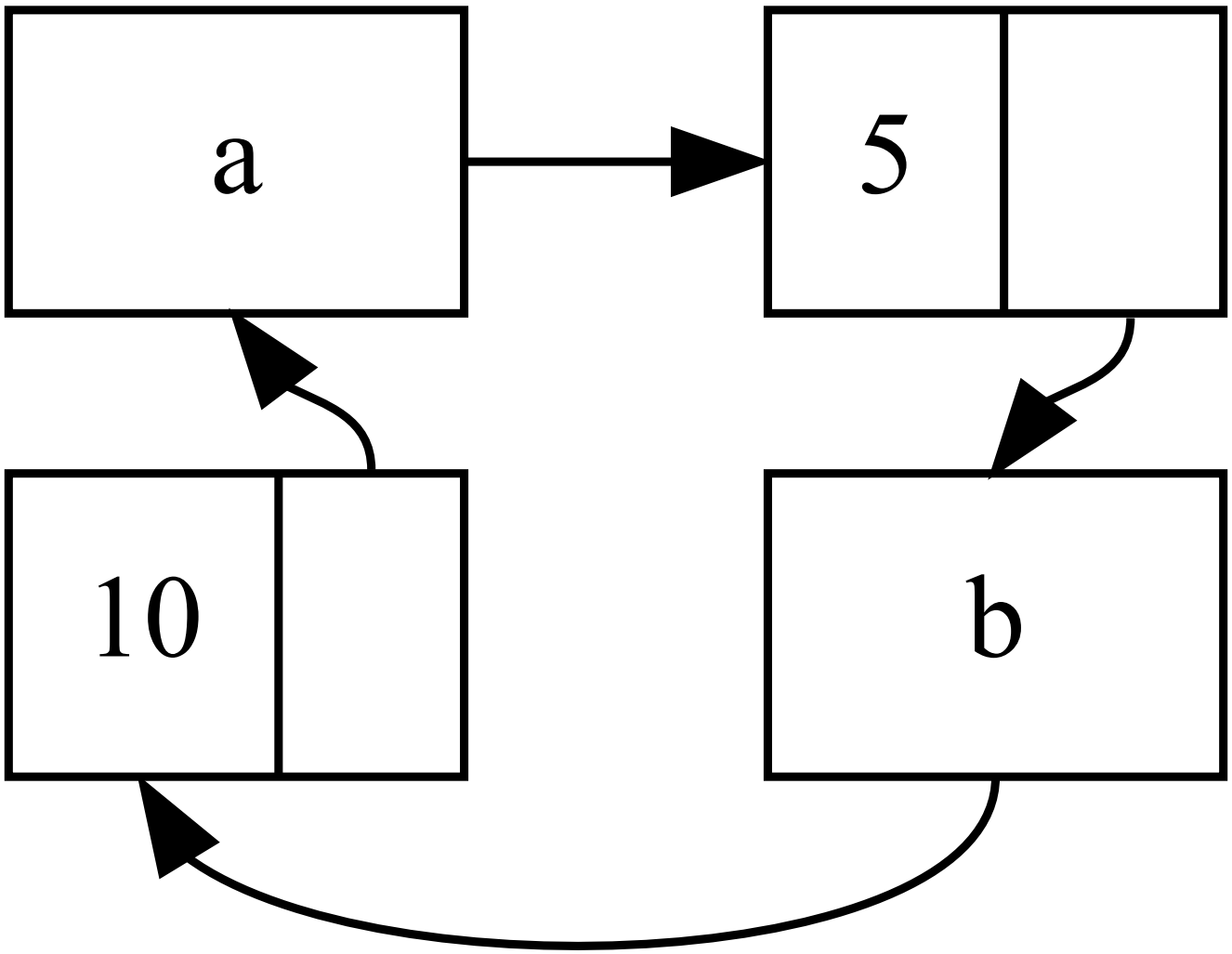


Illustration 15-4 : une boucle de références entre les listes `a` et `b` qui se pointent mutuellement dessus

Si vous décommentez le dernier `println!` et que vous exécutez le programme, Rust va essayer d'afficher cette boucle avec `a` qui pointe sur `b` qui pointe sur `a` ... et ainsi de suite jusqu'à ce que cela fasse déborder la pile.

Dans ce cas, juste après que nous avons créé la boucle de références, le programme se termine. Les conséquences de cette boucle ne sont pas désastreuses. Cependant, si un programme plus complexe alloue beaucoup de mémoire dans une boucle de références et la garde pendant longtemps, le programme va utiliser bien plus de mémoire qu'il n'en a besoin et pourrait surcharger le système en consommant ainsi toute la mémoire disponible.

La création de boucles de références n'est pas facile à réaliser, mais n'est pas non plus impossible. Si vous avez des valeurs `RefCell<T>` qui contiennent des valeurs `Rc<T>` ou des combinaisons similaires de types emboîtés avec de la mutabilité interne et du comptage de références, vous devez vous assurer que vous ne créez pas de boucles ; vous ne pouvez pas

compter sur Rust pour les détecter. La création de boucle de références devrait être un bogue de logique de votre programme dont vous devriez réduire le risque en pratiquant des tests automatisés, des revues de code, ainsi que d'autres pratiques de développement.

Une autre solution pour éviter les boucles de références est de réorganiser vos structures de données afin que certaines références prennent possession et d'autres non. Par conséquent, vous pouvez obtenir des boucles de certaines références qui prennent possession ou d'autres références qui ne prennent pas possession, et seules celles qui prennent possession décident si oui ou non une valeur peut être libérée. Dans l'encart 15-25, nous voulons toujours que les variantes `Cons` possèdent leur propre liste, donc il est impossible de réorganiser la structure des données. Voyons maintenant un exemple qui utilise des graphes constitués de nœuds parents et de nœuds enfants pour voir quand des relations sans possessions constituent un moyen approprié d'éviter les boucles de références.

Eviter les boucles de références : transformer un `Rc<T>` en `Weak<T>`

Précédemment, nous avons démontré que l'appel à `Rc::clone` augmente le `strong_count` d'une instance de `Rc<T>` , et une instance `Rc<T>` est nettoyée seulement si son `strong_count` est à 0. Vous pouvez aussi créer une *référence faible* (NdT : d'où le `weak`) vers la valeur présente dans une instance `Rc<T>` en appelant `Rc::downgrade` et en lui passant une référence vers le `Rc<T>` . Lorsque vous faites appel à `Rc::downgrade` , vous obtenez un pointeur intelligent du type `Weak<T>` . Plutôt que d'augmenter le `strong_count` de l'instance de 1, l'appel à `Rc::downgrade` augmente le `weak_count` de 1. Le type `Rc<T>` utilise le `weak_count` pour compter combien de références `Weak<T>` existent, de la même manière que `strong_count` . La différence réside dans le fait que `weak_count` n'a pas besoin d'être à 0 pour que l'instance `Rc<T>` soit nettoyée.

Les références fortes désignent la manière de partager la propriété d'une instance `Rc<T>` . Les références faibles n'expriment pas de relation de possession. Ils ne provoqueront pas de boucle de références car n'importe quelle boucle impliquant des références faibles sera détruite une fois que le compteur de références fortes des valeurs impliquées vaudra 0.

Comme la valeur contenue dans une référence `Weak<T>` peut être libérée, pour pouvoir faire quelque chose avec cette valeur, vous devez vous assurer qu'elle existe toujours. Vous pouvez faire ceci en appelant la méthode `upgrade` sur une instance `Weak<T>` , qui va retourner une `Option<Rc<T>>` . Ce résultat retournera `Some` si la valeur `Rc<T>` n'a pas encore été libérée, et un `None` si la valeur `Rc<T>` a été libérée. Comme `upgrade` retourne une `Option<Rc<T>>` , Rust va s'assurer que les cas de `Some` et de `None` sont bien gérés, et qu'il n'existe pas de pointeur invalide.

Par exemple, plutôt que d'utiliser une liste dont les éléments ne connaissent que les éléments suivants, nous allons créer un arbre dont les éléments connaissent les éléments enfants *et* leurs éléments parents.

Créer une structure d'arbre de données : un `Noeud` avec des nœuds enfants

Pour commencer, nous allons créer un arbre avec des nœuds qui connaissent leurs nœuds enfants. Nous allons créer une structure `Noeud` qui contient sa propre valeur ainsi que les références vers ses `Noeud` enfants :

Fichier : src/main.rs

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct Noeud {
    valeur: i32,
    enfants: RefCell<Vec<Rc<Noeud>>>,
}
```

Nous souhaitons qu'un `Noeud` prenne possession de ses enfants, et nous souhaitons partager la possession avec des variables afin d'accéder directement à chaque `Noeud` de l'arbre. Pour pouvoir faire ceci, nous définissons les éléments du `Vec<T>` comme étant des valeurs du type `Rc<Noeud>`. Nous souhaitons également pouvoir modifier le fait que tel nœud soit enfant de tel autre, donc, dans `enfants`, nous englobons le `Vec<Rc<Noeud>>` dans un `RefCell<T>`.

Ensuite, nous allons utiliser notre définition de structure et créer une instance de `Noeud` qui s'appellera `feuille` avec la valeur `3` et sans enfant, comme dans l'encart 15-27 :

Filename : src/main.rs

```
fn main() {
    let feuille = Rc::new(Noeud {
        valeur: 3,
        enfants: RefCell::new(vec![]),
    });

    let branche = Rc::new(Noeud {
        valeur: 5,
        enfants: RefCell::new(vec![Rc::clone(&feuille)]),
    });
}
```

Encart 15-27 : création d'un nœud `feuille` sans aucun enfant et un nœud `branche` avec `feuille` comme enfant

Nous créons un clone du `Rc<Noeud>` dans `feuille` et nous le stockons dans `branche`, ce qui signifie que le `Noeud` dans `feuille` a maintenant deux propriétaires : `feuille` et `branche`. Nous pouvons obtenir `feuille` à partir de `branche` en utilisant `branche.feuille`, mais il n'y a pas de moyen d'obtenir `branche` à partir de `feuille`. La raison est que `feuille` n'a pas de référence vers `branche` et ne sait pas s'ils sont liés. Nous voulons que `feuille` sache quelle `branche` est son parent. C'est ce que nous allons faire dès maintenant.

Ajouter une référence à un enfant vers son parent

Pour que le nœud enfant connaisse son parent, nous devons ajouter un champ `parent` vers notre définition de structure `Noeud`. La difficulté ici est de choisir quel sera le type de `parent`. Nous savons qu'il ne peut pas contenir de `Rc<T>`, car cela créera une boucle de référence avec `feuille.parent` qui pointe sur `branche` et `branche.enfant` qui pointe sur `feuille`, ce qui va faire que leurs valeurs `strong_count` ne seront jamais à 0.

En concevant le lien d'une autre manière, un nœud parent devrait prendre possession de ses enfants : si un nœud parent est libéré, ses nœuds enfants devraient aussi être libérés. Cependant, un enfant ne devrait pas prendre possession de son parent : si nous libérons un nœud enfant, le parent doit toujours exister. C'est donc un cas d'emploi pour les références faibles !

Donc, plutôt qu'un `Rc<T>`, nous allons faire en sorte que le type de `parent` soit un `Weak<T>`, plus précisément un `RefCell<Weak<Noeud>>`. Maintenant, la définition de notre structure `Noeud` devrait ressembler à ceci :

Fichier : src/main.rs

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Noeud {
    valeur: i32,
    parent: RefCell<Weak<Noeud>>,
    enfants: RefCell<Vec<Rc<Noeud>>>,
}
```

Un nœud devrait pouvoir avoir une référence vers son nœud parent, mais il ne devrait pas prendre possession de son parent. Dans l'encart 15-28, nous mettons à jour cette nouvelle

définition pour que le nœud `feuille` puisse avoir un moyen de pointer vers son parent, `branche` :

Fichier : src/main.rs

```
fn main() {
    let feuille = Rc::new(Noeud {
        valeur: 3,
        parent: RefCell::new(Weak::new()),
        enfants: RefCell::new(vec![]),
    });

    println!("parent de la feuille = {:?}", feuille.parent.borrow().upgrade());

    let branche = Rc::new(Noeud {
        valeur: 5,
        parent: RefCell::new(Weak::new()),
        enfants: RefCell::new(vec![Rc::clone(&feuille)]),
    });

    *feuille.parent.borrow_mut() = Rc::downgrade(&branche);

    println!("parent de la feuille = {:?}", feuille.parent.borrow().upgrade());
}
```

Encart 15-28 : un nœud `feuille` avec une référence faible vers son nœud parent, `branche`

La création du nœud `feuille` semble être identique à la création du nœud `feuille` de l'encart 15-27, sauf pour le champ `parent` : `feuille` commence sans parent, donc nous créons une nouvelle instance de référence de type `Weak<Noeud>`, qui est vide.

A ce moment-là, lorsque nous essayons d'obtenir une référence vers le parent de `feuille` en utilisant la méthode `upgrade`, nous obtenons une valeur `None`. Nous constatons cela dans la première instruction `println!` sur la sortie :

```
parent de la feuille = None
```

Lorsque nous créons le nœud `branche`, il va aussi avoir une nouvelle référence `Weak<Noeud>` dans le champ `parent`, car `branche` n'a pas de nœud parent. Nous avons néanmoins `feuille` dans `enfants` de `branche`. Une fois que nous avons l'instance de `Noeud` dans `branche`, nous pouvons modifier `feuille` pour lui donner une référence `Weak<Noeud>` vers son parent. Nous utilisons la méthode `borrow_mut` sur la `RefCell<Weak<Noeud>>` du champ `parent` de `feuille`, et ensuite nous utilisons la fonction `Rc::downgrade` pour créer une référence de type `Weak<Node>` vers `branche` à partir du `Rc<Noeud>` présent dans `branche`.

Lorsque nous affichons à nouveau le parent de `feuille`, cette fois nous obtenons la variante `Some` qui contient `branche` : désormais, `feuille` peut accéder à son parent ! Lorsque nous affichons `feuille`, nous avons aussi évité la boucle qui aurait probablement fini en débordement de pile comme nous l'avions expérimenté dans l'encart 15-26 ; les références `Weak<Noeud>` s'écrivent `(Weak)` :

```
parent de la feuille = Some(Noeud { valeur: 5, parent: RefCell { value: (Weak)
},
enfants: RefCell { value: [Noeud { valeur: 3, parent: RefCell { value: (Weak) },
enfants: RefCell { value: [] } }] } })
```

L'absence d'une sortie infinie nous confirme que ce code ne crée pas de boucle de références. Nous pouvons aussi le constater en affichant les valeurs que nous pouvons obtenir en faisant appel à `Rc::strong_count` et `Rc::weak_count`.

Visualiser les modifications de `strong_count` et `weak_count`

Regardons comment changent les valeurs `strong_count` et `weak_count` des instances de `Rc<Noeud>` en créant une portée interne et en déplaçant la création de `branche` dans cette portée. En faisant ceci, nous pourrons constater ce qui se passe lorsque `branche` est créée et lorsqu'elle sera libérée lorsqu'elle sortira de la portée. Ces modifications sont présentées dans l'encart 15-29 :

Fichier : src/main.rs

```
fn main() {
    let feuille = Rc::new(Noeud {
        valeur: 3,
        parent: RefCell::new(Weak::new()),
        enfants: RefCell::new(vec![]),
    });

    println!(
        "feuille strong = {}, weak = {}",
        Rc::strong_count(&feuille),
        Rc::weak_count(&feuille),
    );

    {
        let branche = Rc::new(Noeud {
            valeur: 5,
            parent: RefCell::new(Weak::new()),
            enfants: RefCell::new(vec![Rc::clone(&feuille)]),
        });

        *feuille.parent.borrow_mut() = Rc::downgrade(&branche);

        println!(
            "branche strong = {}, weak = {}",
            Rc::strong_count(&branche),
            Rc::weak_count(&branche),
        );

        println!(
            "feuille strong = {}, weak = {}",
            Rc::strong_count(&feuille),
            Rc::weak_count(&feuille),
        );
    }

    println!("parent de la feuille = {:?}", feuille.parent.borrow().upgrade());
    println!(
        "feuille strong = {}, weak = {}",
        Rc::strong_count(&feuille),
        Rc::weak_count(&feuille),
    );
}
```

Encart 15-29 : création de `branche` dans une portée interne et vérification des compteurs de références strong et weak

Après la création de `feuille`, son `Rc<Noeud>` a le compteur strong à 1 et le compteur weak à 0. Dans la portée interne, nous créons `branche` et l'associons à `feuille`, et à partir de là, lorsque nous affichons les compteurs, le `Rc<Noeud>` dans `branche` aura le compteur strong à 1 et le compteur weak à 1 (pour que `feuille.parent` pointe sur `branche` avec un

`Weak<Noeud>`). Lorsque nous affichons les compteurs dans `feuille` nous constatons qu'il a le compteur strong à 2, car `branche` a maintenant un clone du `Rc<Noeud>` de `feuille` stocké dans `branche.enfants`, mais a toujours le compteur weak à 0.

Lorsque la portée interne se termine, `branche` sort de la portée et le compteur strong de `Rc<Noeud>` décroît à 0, donc son `Noeud` est libéré. Le compteur weak à 1 de `feuille.parent` n'a aucune répercussion suite à la libération ou non du `Noeud`, donc nous ne sommes pas dans une situation de fuite de mémoire !

Si nous essayons d'accéder au parent de `feuille` après la fin de la portée, nous allons à nouveau obtenir `None`. A la fin du programme, le `Rc<Noeud>` dans `feuille` a son compteur strong à 1 et son compteur weak à 0, car la variable `feuille` est à nouveau la seule référence au `Rc<Noeud>`.

Toute cette logique qui gère les compteurs et les libérations des valeurs est intégrée dans `Rc<T>` et `Weak<T>` et leurs implémentations du trait `Drop`. En précisant dans la définition de `Noeud` que le lien entre un enfant et son parent doit être une référence `Weak<T>`, vous pouvez avoir des nœuds parents qui pointent sur des nœuds enfants et vice versa sans risquer de créer des boucles de références et des fuites de mémoire.

Résumé

Ce chapitre a expliqué l'utilisation des pointeurs intelligents pour appliquer des garanties et des compromis différents de ceux qu'applique Rust par défaut avec les références classiques. Le type `Box<T>` a une taille connue et pointe sur une donnée allouée sur le tas. Le type `Rc<T>` compte le nombre de références vers une donnée présente sur le tas afin que cette donnée puisse avoir plusieurs propriétaires. Le type `RefCell<T>` nous permet de l'utiliser lorsque nous avons besoin d'un type immuable mais que nous avons besoin de changer une valeur interne à ce type, grâce à sa fonctionnalité de mutabilité interne ; elle nous permet aussi d'appliquer les règles d'emprunt à l'exécution plutôt qu'à la compilation.

Nous avons aussi vu les traits `Deref` et `Drop`, qui offrent des fonctionnalités très importantes aux pointeurs intelligents. Nous avons expérimenté les boucles de références qui peuvent causer des fuites de mémoire et nous avons vu comment les éviter en utilisant `Weak<T>`.

Si ce chapitre a éveillé votre curiosité et que vous souhaitez mettre en œuvre vos propres pointeurs intelligents, visitez “The Rustonomicon” pour en savoir plus.

Au chapitre suivant, nous allons parler de concurrence en Rust. Vous découvrirez peut-être

même quelques nouveaux pointeurs intelligents ...

La concurrence sans craintes

Le développement sécurisé et efficace dans des contextes de concurrence est un autre objectif majeur de Rust. La *programmation concurrente*, dans laquelle différentes parties d'un programme s'exécutent de manière indépendante, et le *parallélisme*, dans lequel différentes parties d'un programme s'exécutent en même temps, sont devenus des pratiques de plus en plus importantes au fur et à mesure que les ordinateurs tirent parti de leurs processeurs multiples. Historiquement, le développement dans ces contextes était difficile et favorisait les erreurs : Rust compte bien changer la donne.

Au début, l'équipe de Rust pensait que garantir la sécurité de la mémoire et éviter les problèmes de concurrence étaient deux challenges distincts qui devaient être résolus de manières différentes. Avec le temps, l'équipe a découvert que les systèmes de possession et de type sont des jeux d'outils puissants qui aident à sécuriser la mémoire *et* à régler des problèmes de concurrence ! En exploitant la possession et la vérification de type, de nombreuses erreurs de concurrence deviennent des erreurs à la compilation en Rust plutôt que des erreurs à l'exécution. Ainsi, plutôt que d'avoir à passer beaucoup de votre temps à tenter de reproduire les circonstances exactes dans lesquelles un bogue de concurrence s'est produit à l'exécution, le code incorrect va refuser de se compiler et va vous afficher une erreur expliquant le problème. Au final, vous pouvez corriger votre code pendant que vous travaillez dessus plutôt que d'avoir à le faire a posteriori après qu'il ait potentiellement été livré en production. Nous avons surnommé cet aspect de Rust la *concurrence sans craintes*. La concurrence sans craintes vous permet d'écrire du code dépourvu de bogues subtils et qu'il sera facile de remanier sans risquer d'introduire de nouveaux bogues.

> Remarque : pour des raisons de simplicité, nous allons désigner la plupart des problèmes par *des problèmes de concurrence* plutôt que d'être trop précis en disant *des problèmes de concurrence et/ou de parallélisme*. Si ce livre traitait spécifiquement de concurrence et/ou de parallélisme, nous serions plus précis. Pour ce chapitre, veuillez garder à l'esprit que nous parlons de *concurrence et/ou de parallélisme* à chaque fois que nous parlerons de *concurrence*.

De nombreux langages sont dogmatiques sur les solutions qu'ils offrent pour gérer les problèmes de concurrence. Par exemple, Erlang a une fonctionnalité élégante de passage de messages pour la concurrence mais a une façon étrange de partager un état entre les tâches. Ne proposer qu'un sous-ensemble de solutions possibles est une stratégie acceptable pour les langages de haut niveau, car un langage de haut niveau offre des avantages en sacrifiant certains contrôles afin d'être plus accessible. Cependant, les langages de bas niveau sont censés fournir la solution la plus performante dans n'importe

quelle situation donnée et proposer moins d'abstraction vis-à-vis du matériel. C'est pourquoi Rust offre toute une gamme d'outils pour répondre aux problèmes de modélisation quelle que soit la manière qui est adaptée à la situation et aux exigences.

Voici les sujets que nous allons aborder dans ce chapitre :

- Comment créer des tâches pour exécuter plusieurs parties de code en même temps
- Le *passage de message* en concurrence, qui permet à plusieurs tâches d'accéder à la même donnée
- Les traits `Sync` et `Send`, qui étendent les garanties de Rust sur la concurrence tant aux types définis par les utilisateurs qu'à ceux fournis par la bibliothèque standard

Utiliser les tâches pour exécuter simultanément du code

Dans la plupart des systèmes d'exploitation actuels, le code d'un programme est exécuté dans un *processus*, et le système d'exploitation gère plusieurs processus à la fois. Dans votre programme, vous pouvez vous aussi avoir des parties indépendantes qui s'exécutent simultanément. Les éléments qui font fonctionner ces parties indépendantes sont appelés les *tâches*.

Le découpage des calculs de votre programme dans plusieurs tâches peut améliorer sa performance car le programme fait plusieurs choses à la fois, mais cela rajoute aussi de la complexité. Comme les tâches peuvent s'exécuter de manière simultanée, il n'y a pas de garantie absolue sur l'ordre d'exécution des différentes parties de votre code. Cela peut poser des problèmes, tels que :

- Les situations de concurrence, durant lesquelles les tâches accèdent à des données ou des ressources dans un ordre incohérent
- Des interblocages, durant lesquels deux tâches attendent mutuellement que l'autre finisse d'utiliser une ressource que l'autre tâche utilise, bloquant la progression des deux tâches
- Des bogues qui surgissent uniquement dans certaines situations et qui sont difficiles à reproduire et corriger durablement

Rust cherche à atténuer les effets indésirables de l'utilisation des tâches, mais le développement dans un contexte multitâches exige toujours une attention particulière et nécessite une structure de code différente de celle des programmes qui s'exécutent dans une seule tâche.

Les langages de programmation implémentent les tâches de différentes manières. De nombreux systèmes d'exploitation offrent des API pour créer de nouvelles tâches. L'appel à cette API du système d'exploitation pour créer des tâches par un langage est parfois qualifié de *1:1*, ce qui signifie une tâche du système d'exploitation par tâche dans le langage de programmation. La bibliothèque standard de Rust fournit une seule implémentation 1:1 ; il existe des crates qui implémentent d'autres modèles qui font des choix différents.

Créer une nouvelle tâche avec `spawn`

Pour créer une nouvelle tâche, nous appelons la fonction `thread::spawn` et nous lui passons une fermeture (nous avons vu les fermetures au chapitre 13) qui contient le code que nous souhaitons exécuter dans la nouvelle tâche. L'exemple dans l'encart 16-1 affiche du texte à partir de la tâche principale et un autre texte à partir d'une nouvelle tâche :

Fichier : src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("Bonjour n°{} à partir de la nouvelle tâche !", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("Bonjour n°{} à partir de la tâche principale !", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Encart 16-1 : création d'une nouvelle tâche pour afficher une chose pendant que la tâche principale affiche autre chose

Remarquez qu'avec cette fonction, la nouvelle tâche s'arrêtera lorsque la tâche principale s'arrêtera, qu'elle ait fini ou non de s'exécuter. La sortie de ce programme peut être différente à chaque fois, mais elle devrait ressembler à ceci :

```
Bonjour n°1 à partir de la tâche principale !
Bonjour n°1 à partir de la nouvelle tâche !
Bonjour n°2 à partir de la tâche principale !
Bonjour n°2 à partir de la nouvelle tâche !
Bonjour n°3 à partir de la tâche principale !
Bonjour n°3 à partir de la nouvelle tâche !
Bonjour n°4 à partir de la tâche principale !
Bonjour n°4 à partir de la nouvelle tâche !
Bonjour n°5 à partir de la nouvelle tâche !
```

L'appel à `thread::sleep` force une tâche à mettre en pause son exécution pendant une petite durée, permettant à une autre tâche de s'exécuter. Les tâches se relaieront probablement, mais ce n'est pas garanti : cela dépend de comment votre système d'exploitation agence les tâches. Lors de cette exécution, la tâche principale a écrit en premier, même si l'instruction d'écriture de la nouvelle tâche apparaissait d'abord dans le code. Et même si nous avons demandé à la nouvelle tâche d'écrire jusqu'à ce que `i` vaille `9`, elle ne l'a fait que jusqu'à `5`, moment où la tâche principale s'est arrêtée.

Si vous exécutez ce code et que vous ne voyez que du texte provenant de la tâche principale, ou que vous ne voyez aucun chevauchement, essayez d'augmenter les nombres dans les intervalles pour donner plus d'opportunités au système d'exploitation pour basculer entre

les tâches.

Attendre que toutes les tâches aient fini en utilisant `join`

Le code dans l'encart 16-1 non seulement stoppe la nouvelle tâche prématurément la plupart du temps à cause de la fin de la tâche principale, mais il ne garantit pas non plus que la nouvelle tâche va s'exécuter ne serait-ce qu'une seule fois. La raison à cela est qu'il n'y a pas de garantie sur l'ordre dans lequel les tâches vont s'exécuter !

Nous pouvons régler le problème des nouvelles tâches qui ne s'exécutent pas, ou pas complètement, en sauvegardant la valeur de retour de `thread::spawn` dans une variable. Le type de retour de `thread::spawn` est `JoinHandle`. Un `JoinHandle` est une valeur possédée qui, lorsque nous appelons la méthode `join` sur elle, va attendre que ses tâches finissent. L'encart 16-2 montre comment utiliser le `JoinHandle` de la tâche que nous avons créée dans l'encart 16-1 en appelant la méthode `join` pour s'assurer que la nouvelle tâche finit bien avant que `main` ne se termine :

Fichier : src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let manipulateur = thread::spawn(|| {
        for i in 1..10 {
            println!("Bonjour n°{} à partir de la nouvelle tâche !", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("Bonjour n°{} à partir de la tâche principale !", i);
        thread::sleep(Duration::from_millis(1));
    }

    manipulateur.join().unwrap();
}
```

Encart 16-2 : sauvegarde d'un `JoinHandle` d'un `thread::spawn` pour garantir que la tâche est exécutée jusqu'à la fin

L'appel à `join` sur le manipulateur bloque la tâche qui s'exécute actuellement jusqu'à ce que la tâche représentée par le manipulateur se termine. *Bloquer* une tâche signifie que cette tâche est empêchée d'accomplir un quelconque travail ou de se terminer. Comme

nous avons inséré l'appel à `join` après la boucle `for` de la tâche principale, l'exécution de l'encart 16-2 devrait produire un résultat similaire à celui-ci :

```
Bonjour n°1 à partir de la tâche principale !
Bonjour n°2 à partir de la tâche principale !
Bonjour n°1 à partir de la nouvelle tâche !
Bonjour n°3 à partir de la tâche principale !
Bonjour n°2 à partir de la nouvelle tâche !
Bonjour n°4 à partir de la tâche principale !
Bonjour n°3 à partir de la nouvelle tâche !
Bonjour n°4 à partir de la nouvelle tâche !
Bonjour n°5 à partir de la nouvelle tâche !
Bonjour n°6 à partir de la nouvelle tâche !
Bonjour n°7 à partir de la nouvelle tâche !
Bonjour n°8 à partir de la nouvelle tâche !
Bonjour n°9 à partir de la nouvelle tâche !
```

Les deux tâches continuent à alterner, mais la tâche principale attend à cause de l'appel à `manipulateur.join()` et ne se termine pas avant que la nouvelle tâche ne soit finie.

Mais voyons maintenant ce qui se passe lorsque nous déplaçons le `manipulateur.join()` avant la boucle `for` du `main` comme ceci :

Fichier : src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let manipulateur = thread::spawn(|| {
        for i in 1..10 {
            println!("Bonjour n°{} à partir de la nouvelle tâche !", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    manipulateur.join().unwrap();

    for i in 1..5 {
        println!("Bonjour n°{} à partir de la tâche principale !", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

La tâche principale va attendre que la nouvelle tâche se finisse et ensuite exécuter sa boucle `for`, ainsi la sortie ne sera plus chevauchée, comme ci-dessous :

```
Bonjour n°1 à partir de la nouvelle tâche !
Bonjour n°2 à partir de la nouvelle tâche !
Bonjour n°3 à partir de la nouvelle tâche !
Bonjour n°4 à partir de la nouvelle tâche !
Bonjour n°5 à partir de la nouvelle tâche !
Bonjour n°6 à partir de la nouvelle tâche !
Bonjour n°7 à partir de la nouvelle tâche !
Bonjour n°8 à partir de la nouvelle tâche !
Bonjour n°9 à partir de la nouvelle tâche !
Bonjour n°1 à partir de la tâche principale !
Bonjour n°2 à partir de la tâche principale !
Bonjour n°3 à partir de la tâche principale !
Bonjour n°4 à partir de la tâche principale !
```

Des petits détails, comme l'endroit où `join` est appelé, peuvent déterminer si vos tâches peuvent être exécutées ou non en même temps.

Utiliser les fermetures `move` avec les tâches

Le mot-clé `move` est souvent utilisé avec des fermetures passées à `thread::spawn` car la fermeture va alors prendre possession des valeurs de son environnement qu'elle utilise, ce qui transfère la possession des valeurs d'une tâche à une autre. Dans une section du chapitre 13, nous avons présenté `move` dans le contexte des fermetures. A présent, nous allons plus nous concentrer sur l'interaction entre `move` et `thread::spawn`.

Remarquez dans l'encart 16-1 que la fermeture que nous donnons à `thread::spawn` ne prend pas d'arguments : nous n'utilisons aucune donnée de la tâche principale dans le code de la nouvelle tâche. Pour utiliser des données de la tâche principale dans la nouvelle tâche, la fermeture de la nouvelle tâche doit capturer les valeurs dont elle a besoin. L'encart 16-3 montre une tentative de création d'un vecteur dans la tâche principale et son utilisation dans la nouvelle tâche. Cependant, cela ne fonctionne pas encore, comme vous allez le constater dans un moment.

Fichier : src/main.rs

```
use std::thread;

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

    let manipulateur = thread::spawn(|| {
        println!("Voici un vecteur : {:?}", v);
    });

    manipulateur.join().unwrap();
}
```

Encart 16-3 : tentative d'utilisation d'un vecteur créé par la tâche principale dans une autre tâche

La fermeture utilise `v`, donc elle va capturer `v` et l'intégrer dans son environnement. Comme `thread::spawn` exécute cette fermeture dans une nouvelle tâche, nous devrions pouvoir accéder à `v` dans cette nouvelle tâche. Mais lorsque nous compilons cet exemple, nous obtenons l'erreur suivante :

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0373]: closure may outlive the current function, but it borrows `v`,
which is owned by the current function
 --> src/main.rs:6:32
  |
6 |     let manipulateur = thread::spawn(|| {
  |                                      ^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {:?}", v);
  |                                           - `v` is borrowed here
  |
note: function requires argument type to outlive `'static`
 --> src/main.rs:6:18
  |
6 |       let manipulateur = thread::spawn(|| {
  |  ________________________^
7 | |         println!("Here's a vector: {:?}", v);
8 | |     });
  | |______^
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
  |
6 |     let handle = thread::spawn(move || {
  |                                ++++

For more information about this error, try `rustc --explain E0373`.
error: could not compile `threads` due to previous error
```

Rust *déduit* comment capturer `v`, et comme `println!` n'a besoin que d'une référence à `v`,

la fermeture essaye d'emprunter `v`. Cependant, il y a un problème : Rust ne peut pas savoir combien de temps la tâche va s'exécuter, donc il ne peut pas savoir si la référence à `v` sera toujours valide.

L'encart 16-4 propose un scénario qui est a plus de chance d'avoir une référence à `v` qui ne sera plus valide :

Fichier : src/main.rs

```
use std::thread;

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

    let manipulateur = thread::spawn(|| {
        println!("Voici un vecteur : {:?}", v);
    });

    drop(v); // oh, non !

    manipulateur.join().unwrap();
}
```

Encart 16-4 : une tâche dont la fermeture essaye de capturer une référence à `v` à partir de la tâche principale, qui va ensuite libérer `v`

Si nous étions autorisés à exécuter ce code, il y aurait une possibilité que la nouvelle tâche soit immédiatement placée en arrière-plan sans être exécutée du tout. La nouvelle tâche a une référence à `v` en son sein, mais la tâche principale libère immédiatement `v`, en utilisant la fonction `drop` que nous avons vue au chapitre 15. Ensuite, lorsque la nouvelle tâche commence à s'exécuter, `v` n'est plus en vigueur, donc une référence à cette dernière est elle aussi invalide !

Pour corriger l'erreur de compilation de l'encart 16-3, nous pouvons appliquer le conseil du message d'erreur :

```
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
  |
6 |     let manipulateur = thread::spawn(move || {
  |                                      ++++
```

En ajoutant le mot-clé `move` avant la fermeture, nous forçons la fermeture à prendre possession des valeurs qu'elle utilise au lieu de laisser Rust déduire qu'il doit emprunter les valeurs. Les modifications à l'encart 16-3 proposées dans l'encart 16-5 devraient se compiler et s'exécuter comme prévu :

Fichier : src/main.rs

```
use std::thread;

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

    let manipulateur = thread::spawn(move || {
        println!("Voici un vecteur : {:?}", v);
    });

    manipulateur.join().unwrap();
}
```

Encart 16-5 : utilisation du mot-clé `move` pour forcer une fermeture à prendre possession des valeurs qu'elle utilise

Qu'est-ce qui arriverait au code de l'encart 16-4 dans lequel la tâche principale fait appel à `drop` si nous utilisions la fermeture avec `move` ? Est-ce que le `move` résoudrait le problème ? Malheureusement, non ; nous obtiendrions une erreur différente parce que ce que l'encart 16-4 essaye de faire n'est pas autorisé pour une raison différente de la précédente. Si nous ajoutions `move` à la fermeture, nous déplacerions `v` dans l'environnement de la fermeture, et nous ne pourrions plus appeler `drop` sur `v` dans la tâche principale. Nous obtiendrons à la place cette erreur de compilation :

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0382]: use of moved value: `v`
  --> src/main.rs:10:10
   |
4  |     let v = vec![1, 2, 3];
   |         - move occurs because `v` has type `std::vec::Vec<i32>`, which does
not implement the `Copy` trait
5  |
6  |     let manipulateur = thread::spawn(move || {
   |                                      ------- value moved into closure here
7  |         println!("Voici un vecteur : {:?}", v);
   |                                             - variable moved due to use in
closure
...
10 |     drop(v); // oh, non !
   |          ^ value used here after move

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
error: could not compile `threads`.

To learn more, run the command again with --verbose.
```

```
error[E0382]: use of moved value: `v`
  -- > src/main.rs:10:10
    |
6   |     let manipulateur = thread::spawn(move || {
    |                                      ------- value moved (into closure)
here
...
10  |     drop(v); // oh non, le vecteur est libéré !
    |          ^ value used here after move
    |
    = note: move occurs because `v` has type `std::vec::Vec<i32>`, which does
    not implement the `Copy` trait
```

Les règles de possession de Rust nous ont encore sauvé la mise ! Nous obtenions une erreur avec le code de l'encart 16-3 car Rust a été conservateur et a juste emprunté `v` pour la tâche, ce qui signifie que la tâche principale pouvait théoriquement neutraliser la référence de la tâche créée. En demandant à Rust de déplacer la possession de `v` à la nouvelle tâche, nous avons garanti à Rust que la tâche principale n'utiliserait plus `v`. Si nous changeons l'encart 16-4 de la même manière, nous violons les règles de possession lorsque nous essayons d'utiliser `v` dans la tâche principale. Le mot-clé `move` remplace le comportement d'emprunt conservateur par défaut de Rust; il ne nous laisse pas enfreindre les règles de possession.

Armé de cette connaissance de base des tâches et de leur API, découvrons ce que nous pouvons *faire* avec les tâches.

Utiliser l'envoi de messages pour transférer des données entre les tâches

Une approche de plus en plus populaire pour garantir la sécurité de la concurrence est l'*envoi de message*, avec lequel les tâches ou les acteurs communiquent en envoyant aux autres des messages contenant des données. Voici l'idée résumée, tirée d'un slogan provenant de la documentation du langage Go : “Ne communiquez pas en partageant la mémoire ; partagez plutôt la mémoire en communiquant”.

Un des outils majeurs que Rust a pour accomplir l'envoi de messages pour la concurrence est le *canal*, un concept de programmation dont la bibliothèque standard de Rust fournit une implémentation. Vous pouvez imaginer un canal de programmation comme étant un canal d'eau, comme un ruisseau ou une rivière. Si vous posez quelque chose comme un canard en plastique ou un bateau sur un ruisseau, il se déplacera en descendant le long de la voie d'eau.

Un canal de programmation est divisé en deux parties : un transmetteur et un receveur. La partie du transmetteur est le lieu en amont où vous déposez les canards en plastique sur la rivière et la partie du receveur est celle où les canards en plastique finissent leur voyage. Une partie de votre code appelle des méthodes du transmetteur en lui passant les données que vous souhaitez envoyer, tandis qu'une autre partie attend que des messages arrivent. Un canal est déclaré *fermé* lorsque l'une des parties, le transmetteur ou le récepteur, est libérée.

Ici, nous allons concevoir un programme qui a une tâche pour générer des valeurs et les envoyer dans un canal, et une autre tâche qui va recevoir les valeurs et les afficher. Nous allons envoyer de simples valeurs entre les tâches en utilisant un canal pour illustrer cette fonctionnalité. Une fois que vous serez familier avec cette technique, vous pourrez utiliser les canaux pour créer un système de dialogue en ligne ou un système où de nombreuses tâches font chacune une partie d'un gros calcul et envoient leur résultat à une tâche chargée de les agréger.

Pour commencer, dans l'encart 16-6, nous allons créer un canal mais nous n'allons rien faire avec. Remarquez qu'il ne se compilera pas encore car Rust ne peut pas savoir le type de valeurs que nous souhaitons envoyer dans le canal.

Fichier : src/main.rs

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```

Encart 16-6 : création d'un canal et assignation de ses deux parties à `tx` et `rx`

Nous créons un nouveau canal en utilisant la fonction `mpsc::channel` ; `mpsc` signifie *multiple producer, single consumer*, c'est-à-dire *plusieurs producteurs, un seul consommateur*. En bref, la façon dont la bibliothèque standard de Rust a implémenté ces canaux permet d'avoir plusieurs extrémités *émettrices* qui produisent des valeurs, mais seulement une seule extrémité *réceptrice* qui consomme ces valeurs. Imaginez plusieurs ruisseaux qui se rejoignent en une seule grosse rivière : tout ce qui est déposé sur les ruisseaux va finir dans une seule rivière à la fin. Nous allons commencer avec un seul producteur pour le moment, mais nous allons ajouter d'autres producteurs lorsque notre exemple fonctionnera.

La fonction `mpsc::channel` retourne un tuple, le premier élément est celui qui permet d'envoyer et le second est celui qui reçoit. Les abréviations `tx` et `rx` sont utilisés traditionnellement dans de nombreux domaines pour signifier respectivement *transmetteur* et *récepteur*, nous avons donc nommé nos variables ainsi pour indiquer clairement le rôle de chaque élément. Nous utilisons une instruction `let` avec un motif qui déstructure les tuples ; nous verrons l'utilisation des motifs dans les instructions `let` et la déstructuration au chapitre 18. L'utilisation d'une instruction `let` est une façon d'extraire facilement les éléments du tuple retourné par `mpsc::channel` .

Déplaçons maintenant l'élément de transmission dans une nouvelle tâche et faisons-lui envoyer une chaîne de caractères afin que la nouvelle tâche communique avec la tâche principale, comme dans l'encart 16-7. C'est comme poser un canard en plastique sur l'amont de la rivière ou envoyer un message instantané d'une tâche à une autre.

Fichier : src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let valeur = String::from("salut");
        tx.send(valeur).unwrap();
    });
}
```

Encart 16-7 : déplacement de `tx` dans la nouvelle tâche et envoi de “salut”

Nous utilisons à nouveau `thread::spawn` pour créer une nouvelle tâche et ensuite utiliser `move` pour déplacer `tx` dans la fermeture afin que la nouvelle tâche possède désormais `tx` . La nouvelle tâche a besoin de posséder la partie émettrice du canal pour être en

capacité d'envoyer des messages dans ce canal.

La partie émettrice a une méthode `send` qui prend en argument la valeur que nous souhaitons envoyer. La méthode `send` retourne un type `Result<T, E>`, donc si la partie réceptrice a déjà été libérée et qu'il n'y a nulle part où envoyer la valeur, l'opération d'envoi va retourner une erreur. Dans cet exemple, nous faisons appel à `unwrap` pour paniquer en cas d'erreur. Mais dans un vrai programme, nous devrions gérer ce cas correctement : retournez au chapitre 9 pour revoir les stratégies permettant de gérer correctement les erreurs.

Dans l'encart 16-8, nous allons obtenir la valeur de l'extrémité réceptrice du canal dans la tâche principale. C'est comme récupérer le canard en plastique dans l'eau à la fin de la rivière, ou récupérer un message instantané.

Fichier : src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let valeur = String::from("salut");
        tx.send(valeur).unwrap();
    });

    let recu = rx.recv().unwrap();
    println!("On a reçu : {}", recu);
}
```

Encart 16-8 : réception de la valeur “salut” dans la tâche principale pour l'afficher

La partie réception d'un canal a deux modes intéressants : `recv` et `try_recv`. Nous avons utilisé `recv`, un raccourci pour *recevoir*, qui va bloquer l'exécution de la tâche principale et attendre jusqu'à ce qu'une valeur soit envoyée dans le canal. Une fois qu'une valeur est envoyée, `recv` va la retourner dans un `Result<T, E>`. Lorsque la partie transmission du canal se ferme, `recv` va retourner une erreur pour signaler qu'il n'y aura plus de valeurs qui arriveront.

La méthode `try_recv` ne bloque pas, mais va plutôt retourner immédiatement un `Result<T, E>` : une valeur `Ok` qui contiendra un message s'il y en a un de disponible, et une valeur `Err` s'il n'y a pas de message cette fois-ci. L'utilisation de `try_recv` est pratique si cette tâche à d'autres choses à faire pendant qu'elle attend les messages : nous pouvons ainsi écrire une boucle qui appelle régulièrement `try_recv`, gère le message s'il y en a un,

et sinon fait d'autres choses avant de vérifier à nouveau.

Nous avons utilisé `recv` dans cet exemple pour des raisons de simplicité ; nous n'avons rien d'autres à faire dans la tâche principale que d'attendre les messages, donc bloquer la tâche principale est acceptable.

Lorsque nous exécutons le code de l'encart 16-8, nous allons voir la valeur s'afficher grâce à la tâche principale :

```
On a reçu : salut
```

C'est parfait ainsi !

Les canaux et le transfert de possession

Les règles de possession jouent un rôle vital dans l'envoi de messages car elles vous aident à écrire du code sûr et concurrent. Réfléchir à la possession avec vos programmes Rust vous offre l'avantage d'éviter des erreurs de développement avec la concurrence. Faisons une expérience pour montrer comment la possession et les canaux fonctionnent ensemble pour éviter les problèmes : nous allons essayer d'utiliser la `valeur` dans la nouvelle tâche *après* que nous l'avons envoyée dans le canal. Essayez de compiler le code de l'encart 16-9 pour découvrir pourquoi ce code n'est pas autorisé :

Fichier : src/main.rs

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let valeur = String::from("hi");
        tx.send(valeur).unwrap();
        println!("valeur vaut {}", valeur);
    });

    let recu = rx.recv().unwrap();
    println!("On a reçu : {}", recu);
}
```

Encart 16-9 : tentative d'utiliser `valeur` après que nous l'avons envoyée dans le canal

Ici, nous essayons d'afficher `valeur` après que nous l'avons envoyée dans le canal avec `tx.send`. Ce serait une mauvaise idée de permettre cela : une fois que la valeur a été

envoyée à une autre tâche, cette tâche peut la modifier ou la libérer avant que nous essayions de l'utiliser à nouveau. Il est possible que des modifications faites par l'autre tâche puissent causer des erreurs ou des résultats inattendus à cause de données incohérentes ou manquantes. Toutefois, Rust nous affiche une erreur si nous essayons de compiler le code de l'encart 16-9 :

```
$ cargo run
   Compiling message-passing v0.1.0 (file:///projects/message-passing)
error[E0382]: borrow of moved value: `valeur`
  --> src/main.rs:10:31
   |
8  |         let valeur = String::from("salut");
   |             ------ move occurs because `valeur` has type `String`, which
does not implement the `Copy` trait
9  |         tx.send(valeur).unwrap();
   |                 ------ value moved here
10 |         println!("valeur vaut {}", valeur);
   |                                    ^^^^^^ value borrowed here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `message-passing` due to previous error
```

Notre erreur de concurrence a provoqué une erreur à la compilation. La fonction `send` prend possession de ses paramètres, et lorsque la valeur est déplacée, le récepteur en prend possession. Cela nous évite d'utiliser à nouveau accidentellement la valeur après l'avoir envoyée ; le système de possession vérifie que tout est en ordre.

Envoyer plusieurs valeurs et voir le récepteur les attendre

Le code de l'encart 16-8 s'est compilé et exécuté, mais il ne nous a pas clairement indiqué que deux tâches séparées communiquaient entre elles via le canal. Dans l'encart 16-10 nous avons fait quelques modifications qui prouvent que le code de l'encart 16-8 est exécuté avec de la concurrence : la nouvelle tâche va maintenant envoyer plusieurs messages et faire une pause d'une seconde entre chaque message.

Fichier : src/main.rs

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let valeurs = vec![
            String::from("salutations"),
            String::from("à partir"),
            String::from("de la"),
            String::from("nouvelle tâche"),
        ];

        for valeur in valeurs {
            tx.send(valeur).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for recu in rx {
        println!("On a reçu : {}", recu);
    }
}
```

Encart 16-10 : envoi de plusieurs messages en faisant une pause entre chacun

Cette fois-ci, la nouvelle tâche a un vecteur de chaînes de caractères que nous souhaitons envoyer à la tâche principale. Nous itérons sur celui-ci, on envoie les chaînes une par une en faisant une pause entre chaque envoi en appelant la fonction `thread::sleep` avec une valeur `Duration` de 1 seconde.

Dans la tâche principale, nous n'appelons plus explicitement la fonction `recv` : à la place, nous utilisons `rx` comme un itérateur. Pour chaque valeur reçue, nous l'affichons. Lorsque le canal se fermera, l'itération se terminera.

Lorsque nous exécutons le code de l'encart 16-10, nous devrions voir la sortie suivante, avec une pause de 1 seconde entre chaque ligne :

```
On a reçu : salutations
On a reçu : à partir
On a reçu : de la
On a reçu : nouvelle tâche
```

Comme nous n'avons pas de code qui met en pause ou retarde la boucle `for` de la tâche principale, nous pouvons dire que la tâche principale est en attente de réception des valeurs de la part de la nouvelle tâche.

Créer plusieurs producteurs en clonant le transmetteur

Précédemment, nous avions évoqué que `mpsc` était un acronyme pour *multiple producer, single consumer*. Mettons `mpsc` en œuvre en élargissant le code de l'encart 16-10 pour créer plusieurs tâches qui vont toutes envoyer des valeurs au même récepteur. Nous pouvons faire ceci en clonant la partie émettrice du canal, comme dans l'encart 16-11 :

Fichier : src/main.rs

```
// -- partie masquée ici --

let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();
thread::spawn(move || {
    let valeurs = vec![
        String::from("salutations"),
        String::from("à partir"),
        String::from("de la"),
        String::from("nouvelle tâche"),
    ];

    for valeur in valeurs {
        tx1.send(valeur).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

thread::spawn(move || {
    let valeurs = vec![
        String::from("encore plus"),
        String::from("de messages"),
        String::from("pour"),
        String::from("vous"),
    ];

    for valeur in valeurs {
        tx.send(valeur).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

for recu in rx {
    println!("On a reçu : {}", recu);
}

// -- partie masquée ici --
```

Encart 16-11 : envoi de plusieurs messages à partir de plusieurs producteurs

Cette fois-ci, avant de créer la première nouvelle tâche, nous appelons `clone` sur la partie émettrice du canal. Cela va nous donner un nouveau transmetteur que nous pourrons passer à la première nouvelle tâche. Nous passons ensuite le transmetteur original à une seconde nouvelle tâche. Cela va nous donner deux tâches, chacune envoyant des messages différents à la partie réceptrice du canal.

Lorsque vous exécuterez ce code, votre sortie devrait ressembler à ceci :

```
On a reçu : salutations
On a reçu : encore plus
On a reçu : de messages
On a reçu : pour
On a reçu : à partir
On a reçu : de la
On a reçu : nouvelle tâche
On a reçu : pour vous
```

Vous pourrez peut-être constater que les valeurs sont dans un autre ordre chez vous ; cela dépend de votre système. C'est ce qui rend la concurrence aussi intéressante que difficile. Si vous jouez avec la valeur de `thread::sleep` en lui donnant différentes valeurs dans différentes tâches, chaque exécution sera encore moins déterministe et créera une sortie différente à chaque fois.

Maintenant que nous avons découvert le fonctionnement des canaux, examinons un autre genre de concurrence.

Le partage d'état en concurrence

L'envoi de messages est un assez bon moyen de gestion de la concurrence, mais il n'y en a pas qu'un seul. Repensons à cette partie du slogan de la documentation du langage Go : “ne communiquez pas en partageant la mémoire”.

A quoi ressemble la communication par partage de mémoire ? De plus, pourquoi les partisans de l'envoi de messages ne devraient-ils pas l'utiliser et faire plutôt le contraire ?

De manière générale, les canaux dans les langages de programmation ressemblent à la possession exclusive, car une fois que vous avez transféré une valeur dans un canal, vous ne pouvez plus utiliser cette valeur. Le partage de mémoire en concurrence est comme de la possession multiple : plusieurs tâches peuvent accéder au même endroit de la mémoire en même temps. Comme vous l'avez vu au chapitre 15, dans lequel les pointeurs intelligents la rendent possible, la possession multiple peut ajouter de la complexité car ses différents propriétaires ont besoin d'être gérés. Le système de type de Rust et les règles de possession aident beaucoup à les gérer correctement. Par exemple, découvrons les mutex, une des primitives les plus courantes pour partager la mémoire.

Utiliser les mutex pour permettre l'accès à la donnée à une seule tâche à la fois

Mutex est une abréviation pour *mutual exclusion*, ce qui veut dire qu'un mutex ne permet qu'à une seule tâche d'accéder à une donnée à un instant donné. Pour accéder à la donnée dans un mutex, une tâche doit d'abord signaler qu'elle souhaite y accéder en demandant l'obtention du *verrou* du mutex. Le verrou est une structure de donnée qui fait partie du mutex et qui assure le suivi de qui a actuellement accès à la donnée. Par conséquent, le mutex est qualifié de *gardien* de la donnée qu'il renferme via le système de verrou.

Les mutex ont la réputation d'être difficiles à utiliser car vous devez veiller à deux règles :

- Vous devez obtenir le verrou avant d'utiliser la donnée.
- Lorsque vous avez fini avec la donnée que le mutex garde, vous devez déverrouiller la donnée afin que d'autres tâches puissent obtenir le verrou.

Pour faire une métaphore de la vie courante d'un mutex, imaginez une table ronde lors d'une conférence avec un seul microphone. Avant qu'un participant ne puisse parler, il doit demander ou signaler qu'il veut utiliser le micro. Lorsqu'il obtient le micro, il peut parler aussi longtemps qu'il le souhaite et ensuite passer le micro au prochain participant qui a demandé à pouvoir parler. Si un participant oublie de rendre le micro après avoir fini de parler, personne d'autre ne peut parler. Si la gestion du micro partagé se passe mal, la table

ronde ne fonctionnera pas comme prévu !

La gestion des mutex peut devenir incroyablement compliquée, c'est pourquoi tant de personnes sont partisanes des canaux. Cependant, grâce au système de type de Rust et aux règles de possession, vous ne pouvez pas vous tromper dans le verrouillage et déverrouillage.

L'API des `Mutex<T>`

Pour illustrer l'utilisation d'un mutex, commençons par utiliser un mutex dans le contexte d'une seule tâche, comme dans l'encart 16-12 :

Fichier : src/main.rs

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut nombre = m.lock().unwrap();
        *nombre = 6;
    }

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

Encart 16-12 : découverte de l'API de `Mutex<T>` dans le contexte d'une seule tâche pour raison de simplicité

Comme avec beaucoup de types, nous créons un `Mutex<T>` en utilisant la fonction associée `new`. Pour accéder à la donnée dans le mutex, nous utilisons la méthode `lock` pour obtenir le verrou. Cela va bloquer la tâche courante, donc elle ne s'exécutera plus tant que ce ne sera pas à notre tour d'avoir le verrou.

L'appel à `lock` échouera si une autre tâche qui avait le verrou a paniqué. Dans ce cas, personne ne pourra obtenir le verrou, donc nous avons choisi d'utiliser `unwrap` pour que notre tâche panique si nous nous retrouvons dans une telle situation.

Après avoir obtenu le verrou, nous pouvons utiliser la valeur de retour comme une référence mutable vers la donnée, qui s'appellera `nombre` dans ce cas. Le système de type s'assure que nous obtenons le verrou avant d'utiliser la valeur présente dans `m` : le `Mutex<i32>` n'est pas un `i32`, donc nous *devons* obtenir le verrou pour pouvoir utiliser la valeur `i32`. Nous ne pouvons pas l'oublier ; le système de type ne nous laissera pas accéder

au `i32` à l'intérieur de toute façon.

Comme vous pouvez vous en douter, `Mutex<T>` est un pointeur intelligent. Plus précisément, l'appel à `lock` *retourne* un pointeur intelligent `MutexGuard`, intégré dans un `LockResult` que nous avons géré en faisant appel à `unwrap`. Le pointeur intelligent `MutexGuard` implémente `Deref` pour pouvoir pointer sur la donnée interne ; ce pointeur intelligent implémente aussi `Drop` qui libère le verrou automatiquement lorsqu'un `MutexGuard` sort de la portée, ce qui arrive à la fin de la portée interne dans l'encart 16-12. Au final, nous ne risquons pas d'oublier de rendre le verrou et ainsi bloquer l'utilisation du mutex pour les autres tâches car la libération du verrou se produit automatiquement.

Après avoir libéré le verrou, nous pouvons afficher la valeur dans le mutex et constater que nous avons pu changer la valeur interne du `i32` à `6`.

Partager un `Mutex<T>` entre plusieurs tâches

Essayons maintenant de partager une valeur entre plusieurs tâches en utilisant `Mutex<T>`. Nous allons faire fonctionner 10 tâches et faire en sorte que chacune augmente la valeur du compteur de 1, donc le compteur va passer de 0 à 10. Le prochain exemple dans l'encart 16-13 débouchera sur une erreur de compilation, et nous allons utiliser cette erreur pour en apprendre plus sur l'utilisation de `Mutex<T>` et sur la façon dont Rust nous aide à l'utiliser correctement.

Fichier : src/main.rs

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let compteur = Mutex::new(0);
    let mut manipulateurs = vec![];

    for _ in 0..10 {
        let manipulateur = thread::spawn(move || {
            let mut nombre = compteur.lock().unwrap();

            *nombre += 1;
        });
        manipulateurs.push(manipulateur);
    }

    for manipulateur in manipulateurs {
        manipulateur.join().unwrap();
    }

    println!("Resultat : {}", *compteur.lock().unwrap());
}
```

Encart 16-13 : dix tâches qui augmentent chacune un compteur gardé par un `Mutex<T>`

Nous avons créé une variable `compteur` pour stocker un `i32` dans un `Mutex<T>`, comme nous l'avons fait dans l'encart 16-12. Ensuite, nous créons 10 tâches en itérant sur un intervalle de nombres. Nous utilisons `thread::spawn` et nous donnons à toutes les tâches la même fermeture, qui déplace le compteur dans la tâche, obtient le verrou sur le `Mutex<T>` en faisant appel à la méthode `lock` et ajoute ensuite 1 à la valeur présente dans le mutex. Lorsqu'une tâche finit d'exécuter sa fermeture, `nombre` va sortir de la portée et va libérer le verrou afin qu'une autre tâche puisse l'obtenir.

Dans la tâche principale, nous collectons tous les manipulateurs. Ensuite, comme nous l'avions fait dans l'encart 16-2, nous faisons appel à `join` sur chaque manipulateur pour s'assurer que toutes les tâches ont fini. Une fois que c'est le cas, la tâche principale va obtenir le verrou et afficher le résultat de ce programme.

Nous avions annoncé que cet exemple ne se compilerait pas. Découvrons maintenant pourquoi !

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0382]: use of moved value: `compteur`
  --> src/main.rs:9:36
   |
5  |     let compteur = Mutex::new(0);
   |         -------- move occurs because `compteur` has type `Mutex<i32>`,
which does not implement the `Copy` trait
...
9  |         let manipulateur = thread::spawn(move || {
   |                                          ^^^^^^^ value moved into closure
here, in previous iteration of loop
10 |             let mut nombre = compteur.lock().unwrap();
   |                              -------- use occurs due to use in closure

For more information about this error, try `rustc --explain E0382`.
error: could not compile `shared-state` due to previous error
```

Le message d'erreur signale que la valeur `compteur` a été déplacée dans l'itération précédente de la boucle. Donc Rust nous explique qu'il ne peut pas déplacer la possession du verrou de `compteur` dans plusieurs tâches. Corrigeons cette erreur de compilation avec une méthode permettant d'avoir plusieurs propriétaires et que nous avons vue au chapitre 15.

Plusieurs propriétaires avec plusieurs tâches

Dans le chapitre 15, nous avons assigné plusieurs propriétaires à une valeur en utilisant le pointeur intelligent `Rc<T>` pour créer un compteur de référence. Faisons la même chose ici et voyons ce qui se passe. Nous allons intégrer le `Mutex<T>` dans un `Rc<T>` dans l'encart 16-14 et cloner le `Rc<T>` avant de déplacer sa possession à la tâche.

Fichier : src/main.rs

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let compteur = Rc::new(Mutex::new(0));
    let mut manipulateurs = vec![];

    for _ in 0..10 {
        let compteur = Rc::clone(&compteur);
        let manipulateur = thread::spawn(move || {
            let mut nombre = compteur.lock().unwrap();

            *nombre += 1;
        });
        manipulateurs.push(manipulateur);
    }

    for manipulateur in manipulateurs {
        manipulateur.join().unwrap();
    }

    println!("Résultat : {}", *compteur.lock().unwrap());
}
```

Encart 16-14 : tentative d'utilisation d'un `Rc<T>` pour nous permettre d'utiliser plusieurs tâches qui posséderont le `Mutex<T>`

A nouveau, nous compilons et nous obtenons … une erreur différente ! Le compilateur nous en apprend beaucoup.

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
   --> src/main.rs:11:22
    |
11  |           let manipulateur = thread::spawn(move || {
    |  _____________________________^^^^^^^^^^^^^_-
    | |                            |
    | |                            `Rc<Mutex<i32>>` cannot be sent between
threads safely
12  | |             let mut nombre = compteur.lock().unwrap();
13  | |
14  | |             *nombre += 1;
15  | |         });
    | |_________- within this `[closure@src/main.rs:11:36: 15:10]`
    |
    = help: within `[closure@src/main.rs:11:36: 15:10]`, the trait `Send` is not
implemented for `Rc<Mutex<i32>>`
    = note: required because it appears within the type `[closure@src/
main.rs:11:36: 15:10]`
note: required by a bound in `spawn`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `shared-state` due to previous error
```

Ouah, ce message d'erreur est très verbeux ! Voici la partie la plus importante sur laquelle se concentrer : `` `Rc<Mutex<i32>>` cannot be sent between threads safely ``. Le compilateur nous indique aussi pour quelle raison : `` the trait `Send` is not implemented for `Rc<Mutex<i32>>` ``. Nous allons voir `Send` dans la prochaine section : c'est l'un des traits qui garantissent que le type que nous utilisons avec les tâches est prévu pour être utilisé dans des situations de concurrence.

Malheureusement l'utilisation de `Rc<T>` n'est pas sure lorsqu'il est partagé entre plusieurs tâches. Lorsque `Rc<T>` gère le compteur de références, il incrémente le compteur autant de fois que nous avons fait appel à `clone` et décrémente le compteur à chaque fois qu'un clone est libéré. Mais il n'utilise pas de primitives de concurrence pour s'assurer que les changements faits au compteur ne peuvent pas être interrompus par une autre tâche. Cela pourrait provoquer des bogues subtils induisant une mauvaise gestion du compteur, ce qui pourrait provoquer des fuites de mémoire ou faire qu'une valeur soit libérée avant que nous ayions fini de l'utiliser. Nous avons besoin d'un type exactement comme `Rc<T>` mais qui procède aux changements du compteur de références de manière sure en situation de concurrence.

Compteur de référence atomique avec `Arc<T>`

Heureusement, `Arc<T>` *est* un type comme `Rc<T>` qui est sûr en situation de concurrence.

Le *A* signifie *atomique*, ce qui signifie que c'est un type *compteur de références atomique*. L'atome est une sorte de primitive concurrente que nous n'allons pas aborder en détails ici : rendez-vous dans la documentation de la bibliothèque standard sur `std::sync::atomic` pour en savoir plus. Pour le moment, vous avez juste besoin de retenir que les atomes fonctionnent comme les types primitifs mais qui sont sûrs à partager entre plusieurs tâches.

Vous vous demandez pourquoi tous les types primitifs ne sont pas atomiques et pourquoi les types de la bibliothèque standard ne sont pas implémentés en utilisant `Arc<T>` par défaut. La raison à cela est que la sécurité entre les tâches a un coût sur les performances que vous n'êtes prêt à payer que lorsque vous en avez besoin. Si vous procédez à des opérations sur des valeurs uniquement dans une seule tâche, votre code va s'exécuter plus vite car il n'a pas besoin d'appliquer les garanties fournies par les types atomiques.

Retournons à notre exemple : `Arc<T>` et `Rc<T>` ont la même API, donc corrigeons notre programme en changeant la ligne `use` , l'appel à `new` et l'appel à `clone` . Le code dans l'encart 16-15 va finalement se compiler et s'exécuter :

Fichier : src/main.rs

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let compteur = Arc::new(Mutex::new(0));
    let mut manipulateurs = vec![];

    for _ in 0..10 {
        let compteur = Arc::clone(&compteur);
        let manipulateur = thread::spawn(move || {
            let mut nombre = compteur.lock().unwrap();

            *nombre += 1;
        });
        manipulateurs.push(manipulateur);
    }

    for manipulateur in manipulateurs {
        manipulateur.join().unwrap();
    }

    println!("Résultat : {}", *compteur.lock().unwrap());
}
```

Encart 16-15 : utilisation d'un `Arc<T>` pour englober le `Mutex<T>` afin de partager la possession entre plusieurs tâches

Ce code va finalement afficher ceci :

```
Resultat : 10
```

Nous y sommes arrivés ! Nous avons compté de 0 à 10, ce qui ne semble pas très impressionnant, mais cela nous a appris beaucoup sur `Mutex<T>` et la sûreté des tâches. Vous pouvez aussi utiliser cette structure de programme pour procéder à des opérations plus complexes que simplement incrémenter un compteur. En utilisant cette stratégie, vous pouvez diviser un calcul en différentes parties, répartir ces parties sur des tâches, et ensuite utiliser un `Mutex<T>` pour faire en sorte que chaque tâche mette à jour le résultat final avec sa propre partie.

Similarités entre `RefCell<T>/Rc<T>` et `Mutex<T>/Arc<T>`

Vous avez peut-être constaté que `compteur` est immuable mais que nous pouvons obtenir une référence mutable vers la valeur qu'il renferme ; cela signifie que `Mutex<T>` a une mutabilité interne, comme le fait la famille des `Cell`. De la même manière que nous avons utilisé `RefCell<T>` au chapitre 15 pour nous permettre de changer le contenu dans un `Rc<T>`, nous utilisons `Mutex<T>` pour modifier le contenu d'un `Arc<T>`.

Un autre détail à souligner est que Rust ne peut pas vous protéger de tous les genres d'erreurs de logique lorsque vous utilisez `Mutex<T>`. Souvenez-vous que le chapitre 15 utilisait `Rc<T>` avec le risque de créer des boucles de références, dans lesquelles deux valeurs `Rc<T>` se référeraient l'une à l'autre, ce qui provoquait des fuites de mémoire. De la même manière, l'utilisation de `Mutex<T>` risque de créer des *interblocages*. Cela se produit lorsqu'une opération nécessite de verrouiller deux ressources et que deux tâches ont chacune un des deux verrous, ce qui fait qu'elles s'attendent mutuellement pour toujours. Si vous êtes intéressés par les interblocages, essayez de créer un programme Rust qui a un interblocage ; recherchez ensuite des stratégies pour remédier aux interblocages dans n'importe quel langage et implémentez-les en Rust. La documentation de l'API de la bibliothèque standard pour `Mutex<T>` et `MutexGuard` offre des informations précieuses à ce sujet.

Nous allons terminer ce chapitre en parlant des traits `Send` et `Sync` et voir comment nous pouvons les utiliser sur des types personnalisés.

Etendre la concurrence avec les traits `Sync` et `Send`

Curieusement, le langage Rust a *très* peu de fonctionnalités de concurrence. La plupart des fonctionnalités de concurrence que nous avons vues précédemment dans ce chapitre font partie de la bibliothèque standard, pas du langage. Vos options pour gérer la concurrence ne sont pas limitées à celles du langage ou de la bibliothèque standard ; vous pouvez aussi écrire vos propres fonctionnalités de concurrence ou utiliser celles qui ont été écrites par d'autres.

Cependant, deux concepts de concurrence sont intégrés dans le langage : les traits `Sync` et `Send` de `std::marker` .

Permettre le transfert de possession entre les tâches avec `Send`

Le trait `Send` indique que la possession des valeurs du type qui implémente `Send` peut être transféré entre plusieurs tâches. Presque tous les types de Rust implémentent `Send` , mais il subsiste quelques exceptions, comme `Rc<T>` : il ne peut pas implémenter `Send` car si vous clonez une valeur `Rc<T>` et que vous essayez de transférer la possession de ce clone à une autre tâche, les deux tâches peuvent modifier le compteur de référence en même temps. Pour cette raison, `Rc<T>` n'est prévu que pour une utilisation dans des situations qui ne nécessitent qu'une seule tâche et pour lesquelles vous n'avez pas besoin de payer le surcoût sur la performance induit par la sureté de fonctionnement multi tâches.

Toutefois, le système de type et de traits liés de Rust garantit que vous ne pourrez jamais envoyer accidentellement en toute insécurité une valeur `Rc<T>` entre des tâches. Lorsque nous avons essayé de faire cela dans l'encart 16-14, nous avons obtenu l'erreur `the trait Send is not implemented for Rc<Mutex<i32>>` . Lorsque nous l'avons changé pour un `Arc<T>` , qui implémente `Send` , le code s'est compilé.

Tous les types composés entièrement d'autres types qui implémentent `Send` sont automatiquement marqués comme `Send` eux-aussi. Presque tous les types primitifs sont `Send` , à part les pointeurs bruts, ce que nous verrons au chapitre 19.

Permettre l'accès à plusieurs tâches avec `Sync`

Le trait `Sync` indique qu'il est sûr d'avoir une référence dans plusieurs tâches vers le type qui implémente `Sync` . Autrement dit, n'importe quel type `T` implémente `Sync` si `&T` (une référence immuable vers `T`) implémente `Send` , ce qui signifie que la référence peut être envoyée en toute sécurité à une autre tâche. De la même manière que `Send` , les types

primitifs implémentent `Sync`, et les types composés entièrement d'autres types qui implémentent `Sync` sont eux-mêmes `Sync`.

Le pointeur intelligent `Rc<T>` n'implémente pas non plus `Sync` pour les mêmes raisons qu'il n'implémente pas `Send`. Le type `RefCell<T>` (que nous avons vu au chapitre 15) et la famille liée aux types `Cell<T>` n'implémentent pas `Sync`. L'implémentation du vérificateur d'emprunt que `RefCell<T>` met en oeuvre à l'exécution n'est pas sûre pour le multi tâches. Le pointeur intelligent `Mutex<T>` implémente `Sync` et peut être utilisé pour partager l'accès entre plusieurs tâches, comme vous l'avez vu dans la section précédente.

Implémenter manuellement `Send` et `Sync` n'est pas sûr

Comme les types qui sont constitués de types implémentant les traits `Send` et `Sync` sont automatiquement des `Send` et `Sync`, nous n'avons pas à implémenter manuellement ces traits. Comme ce sont des traits de marquage, ils n'ont même pas de méthodes à implémenter. Ils sont uniquement utiles pour appliquer les règles de concurrence.

L'implémentation manuelle de ces traits implique de faire du code Rust non sécurisé. Nous allons voir le code Rust non sécurisé dans le chapitre 19 ; pour l'instant l'information à retenir est que construire de nouveaux types pour la concurrence constitués d'éléments qui n'implémentent pas `Send` et `Sync` nécessite une réflexion approfondie pour respecter les garanties de sécurité. “The Rustonomicon” contient plus d'informations à propos de ces garanties et de la façon de les faire appliquer.

Résumé

Ce n'est pas la dernière fois que vous allez rencontrer de la concurrence dans ce livre : le projet du chapitre 20 va utiliser les concepts de ce chapitre dans une situation plus réaliste que les petits exemples que nous avons utilisés ici.

Nous l'avons dit précédemment, comme les outils pour gérer la concurrence de Rust ne sont pas directement intégrés dans le langage, de nombreuses solutions pour de la concurrence sont implémentées dans des crates. Elles évoluent plus rapidement que la bibliothèque standard, donc assurez-vous de rechercher en ligne des crates modernes et à la pointe de la technologie à utiliser dans des situations multitâches.

La bibliothèque standard de Rust fournit les canaux pour l'envoi de messages et les types de pointeurs intelligents, comme `Mutex<T>` et `Arc<T>`, qui sont sûrs à utiliser en situation de concurrence. Le système de type et le vérificateur d'emprunt sont là pour s'assurer que le

code utilisé dans ces solutions ne vont pas conduire à des situations de concurrence ou utiliser des références qui ne sont plus en vigueur. Une fois que votre code se compile, vous pouvez être assuré qu'il fonctionnera bien sur plusieurs tâches sans avoir les genres de bogues *difficiles à traquer* qui sont monnaie courante dans les autres langages. Le développement en concurrence est un domaine qui ne devrait plus faire peur : lancez-vous et utilisez la concurrence dans vos programmes sans crainte !

Au chapitre suivant, nous allons voir des techniques adaptées pour modéliser des problèmes et structurer votre solution au fur et à mesure que vos programmes en Rust grandissent. De plus, nous analyserons les liens qui peuvent exister entre les idées de Rust et celles avec lesquelles vous êtes peut-être familier en programmation orientée objet.

Les fonctionnalités orientées objet de Rust

La programmation orientée objet (POO) est une façon de concevoir des programmes. Les objets sont apparus dans Simula dans les années 1960. Ces objets ont influencé l'architecture de programmation d'Alan Kay dans laquelle les objets s'envoient des messages. Il a inventé le terme *programmation orientée objet* en 1967 pour décrire cette architecture. Plusieurs définitions de la POO s'opposent ; Rust est considéré comme orienté objet selon certaines définitions mais pas par d'autres. Dans ce chapitre, nous examinerons certaines caractéristiques généralement considérées comme orientées objet et nous verrons comment ces caractéristiques se traduisent en code Rust traditionnel. Puis nous vous montrerons comment implémenter un patron de conception orienté objet en Rust et nous comparerons les avantages et inconvénients de faire cela plutôt que d'implémenter une solution qui utilise quelques points forts de Rust.

Les caractéristiques des langages orientés objet

Les développeurs ne se sont jamais entendus sur les fonctionnalités qu'un langage doit avoir pour être considéré orienté objet. Rust est influencé par de nombreux paradigmes de programmation, y compris la POO ; par exemple, nous avons examiné les fonctionnalités issues de la programmation fonctionnelle au chapitre 13. On peut vraisemblablement dire que les langages orientés objet ont plusieurs caractéristiques en commun, comme les objets, l'encapsulation et l'héritage. Examinons chacune de ces caractéristiques et regardons si Rust les supporte.

Les objets contiennent des données et suivent un comportement

Le livre *Design Patterns: Elements of Reusable Object-Oriented Software* d'Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (Addison-Wesley Professional, 1994) que l'on surnomme le livre du *Gang of Four* est un catalogue de patrons de conception orientés objet. Il définit la POO ainsi :

> Les programmes orientés objet sont constitués d'objets. Un *objet* regroupe des données ainsi que les procédures qui opèrent sur ces données. Ces procédures sont typiquement appelées *méthodes* ou *opérations*.

Si l'on s'en tient à cette définition, Rust est orienté objet : les structures et les énumérations ont des données et les blocs `impl` leur fournissent des méthodes. Bien que les structures et les énumérations dotées de méthodes ne soient pas qualifiées d'objets, elles en ont les fonctionnalités selon la définition des objets faite par le *Gang of Four*.

L'encapsulation qui masque les détails d'implémentation

Un autre aspect qu'on associe souvent à la POO est l'idée d'*encapsulation*, ce qui signifie que les détails d'implémentation d'un objet ne sont pas accessibles au code utilisant cet objet. Ainsi, la seule façon d'interagir avec un objet est via son API publique ; le code qui utilise l'objet ne devrait pas pouvoir accéder aux éléments internes d'un objet et changer directement ses données ou son comportement. Cela permet au développeur de changer et remanier les éléments internes d'un objet sans avoir à changer le code qui utilise cet objet.

Nous avons abordé la façon de contrôler l'encapsulation au chapitre 7 : on peut utiliser le mot-clé `pub` pour décider quels modules, types, fonctions et méthodes de notre code devraient être publics ; par défaut, tout le reste est privé. Par exemple, nous pouvons définir

une structure `CollectionMoyennee` qui a un champ contenant un vecteur de valeurs `i32`. La structure peut aussi avoir un champ qui contient la moyenne des valeurs dans le vecteur de sorte qu'il ne soit pas nécessaire de recalculer la moyenne à chaque fois que quelqu'un en a besoin. En d'autres termes, `CollectionMoyennee` va mettre en cache la moyenne calculée pour nous. L'encart 17-1 contient la définition de la structure `CollectionMoyennee` :

Fichier : src/lib.rs

```
pub struct CollectionMoyennee {
    liste: Vec<i32>,
    moyenne: f64,
}
```

Encart 17-1 : Une structure `CollectionMoyennee` qui contient une liste d'entiers et la moyenne des éléments de la collection

La structure est marquée `pub` de façon à ce qu'elle puisse être utilisée par du code externe, mais les champs au sein de la structure restent privés. C'est important dans ce cas puisque nous voulons nous assurer que lorsqu'une valeur est ajoutée ou retirée dans la liste, la moyenne soit aussi mise à jour. Nous le faisons en implémentant les méthodes `ajouter`, `retirer` et `moyenne` sur la structure, comme le montre l'encart 17-2 :

Fichier : src/lib.rs

```
impl CollectionMoyennee {
    pub fn ajouter(&mut self, valeur: i32) {
        self.liste.push(valeur);
        self.mettre_a_jour_moyenne();
    }

    pub fn retirer(&mut self) -> Option<i32> {
        let resultat = self.liste.pop();
        match resultat {
            Some(valeur) => {
                self.mettre_a_jour_moyenne();
                Some(valeur)
            }
            None => None,
        }
    }

    pub fn moyenne(&self) -> f64 {
        self.moyenne
    }

    fn mettre_a_jour_moyenne(&mut self) {
        let total: i32 = self.liste.iter().sum();
        self.moyenne = total as f64 / self.liste.len() as f64;
    }
}
```

Encart 17-2 : Implémentations des méthodes publiques `ajouter`, `retirer` et `moyenne` sur `CollectionMoyennee`

Les méthodes publiques `ajouter`, `retirer` et `moyenne` sont les seules façons d'accéder ou de modifier les données d'une instance de `CollectionMoyennee`. Lorsqu'un élément est ajouté à `liste` en utilisant la méthode `ajouter` ou retiré en utilisant la méthode `retirer`, l'implémentation de chacune de ces méthodes appelle la méthode privée `mettre_a_jour_moyenne` qui met à jour le champ `moyenne` également.

Nous laissons les champs `liste` et `moyenne` privés pour qu'il soit impossible pour du code externe d'ajouter ou de retirer des éléments dans notre champ `liste` directement ; sinon, le champ `moyenne` pourrait ne plus être synchronisé lorsque la liste change. La méthode `moyenne` renvoie la valeur du champ `moyenne`, ce qui permet au code externe de lire le champ `moyenne` mais pas de le modifier.

Puisque nous avons encapsulé les détails d'implémentation de la structure `CollectionMoyennee`, nous pourrons aisément en changer plus tard quelques aspects, tels que la structure de données. Par exemple, nous pourrions utiliser un `HashSet<i32>` plutôt qu'un `Vec<i32>` pour le champ `liste`. Du moment que les signatures des méthodes

publiques `ajouter`, `retirer` et `moyenne` restent les mêmes, du code qui utilise `CollectionMoyennee` n'aurait pas besoin de changer. En revanche, si nous avions fait en sorte que `liste` soit publique, cela n'aurait pas été forcément le cas : `HashSet<i32>` et `Vec<i32>` ont des méthodes différentes pour ajouter et retirer des éléments, donc il aurait vraisemblablement fallu changer le code externe s'il modifiait directement `liste`.

Si l'encapsulation est une condition nécessaire pour qu'un langage soit considéré orienté objet, alors Rust satisfait cette condition. La possibilité d'utiliser `pub` ou non pour différentes parties de notre code permet d'encapsuler les détails d'implémentation.

L'héritage comme système de type et comme partage de code

L'*héritage* est un mécanisme selon lequel un objet peut hériter de la définition d'un autre objet, acquérant ainsi les données et le comportement de l'objet père sans que l'on ait besoin de les redéfinir.

Si un langage doit avoir de l'héritage pour être un langage orienté objet, alors Rust n'en est pas un. Il est impossible de définir une structure qui hérite des champs et de l'implémentation des méthodes de la structure mère. Cependant, si vous avez l'habitude d'utiliser l'héritage dans vos programmes, vous pouvez utiliser d'autres solutions en Rust, en fonction de la raison qui vous a conduit en premier lieu à vous tourner vers l'héritage.

Il y a deux principales raisons de choisir l'héritage. La première raison est la réutilisation de code : vous pouvez implémenter un comportement particulier pour un type, et l'héritage vous permet de réutiliser cette implémentation sur un autre type. À la place, vous pouvez partager du code Rust en utilisant des implémentations de méthodes de trait par défaut, comme nous l'avons vu dans l'encart 10-14 lorsque nous avons ajouté une implémentation par défaut de la méthode `resumer` sur le trait `Resumable`. La méthode `resumer` serait alors disponible sur tout type implémentant le trait `Resumable` sans avoir besoin de rajouter du code. C'est comme si vous aviez une classe mère avec l'implémentation d'une méthode et une classe fille avec une autre implémentation de cette méthode. On peut aussi remplacer l'implémentation par défaut de la méthode `resumer` quand on implémente le trait `Resumable`, un peu comme une classe fille qui remplace l'implémentation d'une méthode héritée d'une classe mère.

L'autre raison d'utiliser l'héritage concerne le système de types : pour permettre à un type fils d'être utilisé à la place d'un type père. Cela s'appelle le *polymorphisme*, ce qui veut dire qu'on peut substituer plusieurs objets entre eux à l'exécution s'ils partagent certaines caractéristiques.

Polymorphisme

Pour beaucoup de gens, le polymorphisme est synonyme d'héritage. Mais il s'agit en fait d'un principe plus général qui se rapporte au code manipulant des données de divers types. Pour l'héritage, ces types sont généralement des classes filles (ou *sous-classes*).

À la place, Rust utilise la généricité pour construire des abstractions des différents types et traits liés possibles pour imposer des contraintes sur ce que ces types doivent fournir. Cela est parfois appelé *polymorphisme paramétrique borné*.

L'héritage est récemment tombé en disgrâce en tant que solution de conception dans plusieurs langages de programmation parce qu'il conduit souvent à partager plus de code que nécessaire. Les classes mères ne devraient pas toujours partager toutes leurs caractéristiques avec leurs classes filles, mais elles y sont obligées avec l'héritage. Cela peut rendre la conception d'un programme moins flexible. De plus, cela introduit la possibilité d'appeler des méthodes sur des classes filles qui n'ont aucun sens ou qui entraînent des erreurs parce que les méthodes ne s'appliquent pas à la classe fille. De plus, certains langages ne permettront à une classe fille d'hériter que d'une seule classe, ce qui restreint d'autant plus la flexibilité de la conception d'un programme.

Voilà pourquoi Rust suit une autre approche, en utilisant des objets traits plutôt que l'héritage. Jetons un œil à la façon dont les objets traits permettent le polymorphisme en Rust.

Utiliser les objets traits qui permettent des valeurs de types différents

Au chapitre 8, nous avions mentionné qu'une limite des vecteurs est qu'ils ne peuvent stocker des éléments que d'un seul type. Nous avions contourné le problème dans l'encart 8-10 en définissant une énumération `Cellule` avec des variantes pouvant contenir des entiers, des flottants et du texte. Ainsi, on pouvait stocker différents types de données dans chaque cellule et quand même avoir un vecteur qui représentait une rangée de cellules. C'est une très bonne solution quand nos éléments interchangeables ne possèdent qu'un ensemble bien déterminé de types que nous connaissons lors de la compilation de notre code.

Cependant, nous avons parfois envie que l'utilisateur de notre bibliothèque puisse étendre l'ensemble des types valides dans une situation donnée. Pour montrer comment nous pourrions y parvenir, créons un exemple d'outil d'interface graphique (GUI) qui itère sur une liste d'éléments et appelle une méthode `afficher` sur chacun d'entre eux pour l'afficher à l'écran — une technique courante pour les outils d'interface graphique. Créons une *crate* de bibliothèque appelée `gui` qui contient la structure d'une bibliothèque d'interface graphique. Cette *crate* pourrait inclure des types que les usagers pourront utiliser, tels que `Bouton` ou `ChampDeTexte`. De plus, les utilisateurs de `gui` voudront créer leurs propres types qui pourront être affichés : par exemple, un développeur pourrait ajouter une `Image` et un autre pourrait ajouter une `ListeDeroulante`.

Nous n'implémenterons pas une véritable bibliothèque d'interface graphique pour cet exemple, mais nous verrons comment les morceaux pourraient s'assembler. Au moment d'écrire la bibliothèque, nous ne pouvons pas savoir ni définir tous les types que les autres développeurs auraient envie de créer. Mais nous savons que `gui` doit gérer plusieurs valeurs de types différents et qu'elle doit appeler la méthode `afficher` sur chacune de ces valeurs de types différents. Elle n'a pas besoin de savoir exactement ce qui arrivera quand on appellera la méthode `afficher`, mais seulement de savoir que la valeur disposera de cette méthode que nous pourrons appeler.

Pour faire ceci dans un langage avec de l'héritage, nous pourrions définir une classe `Composant` qui a une méthode `afficher`. Les autres classes, telles que `Bouton`, `Image` et `ListeDeroulante` hériteraient de `Composant` et hériteraient ainsi de la méthode `afficher`. Elles pourraient toutes redéfinir la méthode `afficher` avec leur comportement personnalisé, mais l'environnement de développement pourrait considérer tous les types comme des instances de `Composant` et appeler `afficher` sur chacun d'entre eux. Mais puisque Rust n'a pas d'héritage, il nous faut un autre moyen de structurer la bibliothèque `gui` pour permettre aux utilisateurs de l'enrichir avec de nouveaux types.

Définir un trait pour du comportement commun

Pour implémenter le comportement que nous voulons donner à `gui`, nous définirons un trait nommé `Affichable` qui aura une méthode nommée `afficher`. Puis nous définirons un vecteur qui prend un *objet trait*. Un objet trait pointe à la fois vers une instance d'un type implémentant le trait indiqué ainsi que vers une table utilisée pour chercher les méthodes de trait de ce type à l'exécution. Nous créons un objet trait en indiquant une sorte de pointeur, tel qu'une référence `&` ou un pointeur intelligent `Box<T>`, puis le mot-clé `dyn` et enfin le trait en question. (Nous expliquerons pourquoi les objets traits doivent utiliser un pointeur dans une section du chapitre 19.) Nous pouvons utiliser des objets traits à la place d'un type générique ou concret. Partout où nous utilisons un objet trait, le système de types de Rust s'assurera à la compilation que n'importe quelle valeur utilisée dans ce contexte implémentera le trait de l'objet trait. Ainsi, il n'est pas nécessaire de connaître tous les types possibles à la compilation.

Nous avons mentionné qu'en Rust, nous nous abstenons de qualifier les structures et énumérations d'*objets* pour les distinguer des objets des autres langages. Dans une structure ou une énumération, les données dans les champs de la structure et le comportement dans les blocs `impl` sont séparés, alors que dans d'autres langages, les données et le comportement se combinent en un concept souvent qualifié d'objet. En revanche, les objets traits ressemblent davantage aux objets des autres langages dans le sens où ils combinent des données et du comportement. Mais les objets traits diffèrent des objets traditionnels dans le sens où on ne peut pas ajouter des données à un objet trait. Les objets traits ne sont généralement pas aussi utiles que les objets des autres langages : leur but spécifique est de permettre de construire des abstractions de comportements communs.

L'encart 17-3 illustre la façon de définir un trait nommé `Affichable` avec une méthode nommée `afficher` :

Fichier : src/lib.rs

```
pub trait Affichable {
    fn afficher(&self);
}
```

Encart 17-3 : définition du trait `Affichable`

Cette syntaxe devrait vous rappeler nos discussions sur comment définir des traits au chapitre 10. Puis vient une nouvelle syntaxe : l'encart 17-4 définit une structure `Ecran` qui contient un vecteur `composants`. Ce vecteur est du type `Box<dyn Affichable>`, qui est un objet trait ; c'est un bouche-trou pour n'importe quel type au sein d'un `Box` qui implémente

le trait `Affichable`.

Fichier : src/lib.rs

```
pub struct Ecran {
    pub composants: Vec<Box<dyn Affichable>>,
}
```

Encart 17-4 : définition de la structure `Ecran` avec un champ `composants` contenant un vecteur d'objets traits qui implémentent le trait `Affichable`

Sur la structure `Ecran`, nous allons définir une méthode nommée `executer` qui appellera la méthode `afficher` sur chacun de ses `composants`, comme l'illustre l'encart 17-5 :

Fichier : src/lib.rs

```
impl Ecran {
    pub fn executer(&self) {
        for composant in self.composants.iter() {
            composant.afficher();
        }
    }
}
```

Encart 17-5 : une méthode `executer` sur `Ecran` qui appelle la méthode `afficher` sur chaque composant

Cela ne fonctionne pas de la même manière que d'utiliser une structure avec un paramètre de type générique avec des traits liés. Un paramètre de type générique ne peut être remplacé que par un seul type concret à la fois, tandis que les objets traits permettent à plusieurs types concrets de remplacer l'objet trait à l'exécution. Par exemple, nous aurions pu définir la structure `Ecran` en utilisant un type générique et un trait lié comme dans l'encart 17-6 :

Fichier : src/lib.rs

```
pub struct Ecran<T: Affichable> {
    pub composants: Vec<T>,
}

impl<T> Ecran<T>
where
    T: Affichable,
{
    pub fn executer(&self) {
        for composant in self.composants.iter() {
            composant.afficher();
        }
    }
}
```

Encart 17-6 : une implémentation différente de la structure `Ecran` et de sa méthode `executer` en utilisant la généricité et les traits liés

Cela nous restreint à une instance de `Ecran` qui a une liste de composants qui sont soit tous de type `Bouton`, soit tous de type `ChampDeTexte`. Si vous ne voulez que des collections homogènes, il est préférable d'utiliser la généricité et les traits liés parce que les définitions seront monomorphisées à la compilation pour utiliser les types concrets.

D'un autre côté, en utilisant des objets traits, une instance de `Ecran` peut contenir un `Vec<T>` qui contient à la fois un `Box<Bouton>` et un `Box<ChampDeTexte>`. Regardons comment cela fonctionne, puis nous parlerons ensuite du coût en performances à l'exécution.

Implémenter le trait

Ajoutons maintenant quelques types qui implémentent le trait `Affichable`. Nous fournirons le type `Bouton`. Encore une fois, implémenter une vraie bibliothèque d'interface graphique dépasse la portée de ce livre, alors la méthode `afficher` n'aura pas d'implémentation utile dans son corps. Pour imaginer à quoi pourrait ressembler l'implémentation, une structure `Bouton` pourrait avoir des champs `largeur`, `hauteur` et `libelle`, comme l'illustre l'encart 17-7 :

Fichier : src/lib.rs

```
pub struct Bouton {
    pub largeur: u32,
    pub hauteur: u32,
    pub libelle: String,
}

impl Affichable for Bouton {
    fn afficher(&self) {
        // code servant à afficher vraiment un bouton
    }
}
```

Encart 17-7 : une structure `Bouton` qui implémente le trait `Affichable`

Les champs `largeur`, `hauteur` et `libelle` de `Bouton` pourront ne pas être les mêmes que ceux d'autres composants, comme un type `ChampDeTexte`, qui pourrait avoir ces champs plus un champ `texte_de_substitution` à la place. Chacun des types que nous voudrons afficher à l'écran implémentera le trait `Affichable` mais utilisera du code différent dans la méthode `afficher` pour définir comment afficher ce type en particulier, comme c'est le cas de `Bouton` ici (sans le vrai code d'implémentation, qui dépasse le cadre de ce chapitre). Le type `Bouton`, par exemple, pourrait avoir un bloc `impl` supplémentaire contenant des méthodes en lien à ce qui arrive quand un utilisateur clique sur le bouton. Ce genre de méthodes ne s'applique pas à des types comme `ChampDeTexte`.

Si un utilisateur de notre bibliothèque décide d'implémenter une structure `ListeDeroulante` avec des champs `largeur`, `hauteur` et `options`, il implémentera également le trait `Affichable` sur le type `ListeDeroulante`, comme dans l'encart 17-8 :

Fichier : src/main.rs

```
use gui::Affichable;

struct ListeDeroulante {
    largeur: u32,
    hauteur: u32,
    options: Vec<String>,
}

impl Affichable for ListeDeroulante {
    fn afficher(&self) {
        // code servant à afficher vraiment une liste déroulante
    }
}
```

Encart 17-8 : une autre *crate* utilisant `gui` et implémentant le trait `Affichable` sur une structure `ListeDeroulante`

L'utilisateur de notre bibliothèque peut maintenant écrire sa fonction `main` pour créer une instance de `Ecran`. Il peut ajouter à l'instance de `Ecran` une `ListeDeroulante` ou un `Bouton` en les mettant chacun dans un `Box<T>` pour en faire des objets traits. Il peut ensuite appeler la méthode `executer` sur l'instance de `Ecran`, qui appellera `afficher` sur chacun de ses composants. L'encart 17-9 montre cette implémentation :

Fichier : src/main.rs

```
use gui::{Bouton, Ecran};

fn main() {
    let ecran = Ecran {
        composants: vec![
            Box::new(ListeDeroulante {
                largeur: 75,
                hauteur: 10,
                options: vec![
                    String::from("Oui"),
                    String::from("Peut-être"),
                    String::from("Non"),
                ],
            }),
            Box::new(Bouton {
                largeur: 50,
                hauteur: 10,
                libelle: String::from("OK"),
            }),
        ],
    };

    ecran.executer();
}
```

Encart 17-9 : utilisation d'objets traits pour stocker des valeurs de types différents qui implémentent le même trait

Quand nous avons écrit la bibliothèque, nous ne savions pas que quelqu'un pourrait y ajouter le type `ListeDeroulante`, mais notre implémentation de `Ecran` a pu opérer sur le nouveau type et l'afficher parce que `ListeDeroulante` implémente le trait `Affichable`, ce qui veut dire qu'elle implémente la méthode `afficher`.

Ce concept — se préoccuper uniquement des messages auxquels une valeur répond plutôt que du type concret de la valeur — est similaire au concept du *duck typing* (“typage canard”) dans les langages typés dynamiquement : si ça marche comme un canard et que ça fait coin-coin comme un canard, alors ça doit être un canard ! Dans l'implémentation de `executer` sur `Ecran` dans l'encart 17-5, `executer` n'a pas besoin de connaître le type concret de chaque composant. Elle ne vérifie pas si un composant est une instance de `Bouton` ou de

`ListeDeroulante`, elle ne fait qu'appeler la méthode `afficher` sur le composant. En spécifiant `Box<dyn Affichable>` comme type des valeurs dans le vecteur `composants`, nous avons défini que `Ecran` n'avait besoin que de valeurs sur lesquelles on peut appeler la méthode `afficher`.

L'avantage d'utiliser les objets traits et le système de types de Rust pour écrire du code semblable à celui utilisant le *duck typing* est que nous n'avons jamais besoin de vérifier si une valeur implémente une méthode en particulier à l'exécution, ni de nous inquiéter d'avoir des erreurs si une valeur n'implémente pas une méthode mais qu'on l'appelle quand même. Rust ne compilera pas notre code si les valeurs n'implémentent pas les traits requis par les objets traits.

Par exemple, l'encart 17-10 montre ce qui arrive si on essaie de créer un `Ecran` avec une `String` comme composant :

Fichier : src/main.rs

```
use gui::Ecran;

fn main() {
    let ecran = Ecran {
        composants: vec![Box::new(String::from("Salutations"))],
    };

    ecran.run();
}
```

Encart 17-10 : tentative d'utiliser un type qui n'implémente pas le trait de l'objet trait

Nous aurons cette erreur parce que `String` n'implémente pas le trait `Affichable` :

```
$ cargo run
   Compiling gui v0.1.0 (file:///projects/gui)
error[E0277]: the trait bound `String: Affichable` is not satisfied
 --> src/main.rs:5:26
  |
5 |         composants: vec![Box::new(String::from("Salutations"))],
  |                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait
`Affichable` is not implemented for `String`
  |
  = note: required for the cast to the object type `dyn Affichable`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `gui` due to previous error
```

L'erreur nous fait savoir que soit nous passons quelque chose à `Ecran` que nous ne voulions pas lui passer et que nous devrions lui passer un type différent, soit nous devrions

implémenter `Affichable` sur `String` de sorte que `Ecran` puisse appeler `afficher` dessus.

Les objets traits effectuent de la répartition dynamique

Rappelez-vous de notre discussion dans une section du chapitre 10 à propos du processus de monomorphisation effectué par le compilateur quand nous utilisons des traits liés sur des génériques : le compilateur génère des implémentations non génériques de fonctions et de méthodes pour chaque type concret que nous utilisons à la place d'un paramètre de type générique. Le code résultant de la monomorphisation effectue du *dispatch statique* (*répartition statique*), qui peut être mis en place quand le compilateur sait, au moment de la compilation, quelle méthode vous appelez. Cela s'oppose au *dispatch dynamique* (*répartition dynamique*), qui est mis en place quand le compilateur ne peut pas déterminer à la compilation quelle méthode vous appelez. Dans le cas de la répartition dynamique, le compilateur produit du code qui devra déterminer à l'exécution quelle méthode appeler.

Quand nous utilisons des objets traits, Rust doit utiliser de la répartition dynamique. Le compilateur ne connaît pas tous les types qui pourraient être utilisés avec le code qui utilise des objets traits, donc il ne sait pas quelle méthode implémentée sur quel type il doit appeler. À la place, lors de l'exécution, Rust utilise les pointeurs à l'intérieur de l'objet trait pour savoir quelle méthode appeler. Il y a un coût à l'exécution lors de la recherche de cette méthode qui n'a pas lieu avec la répartition statique. La répartition dynamique empêche en outre le compilateur de choisir de remplacer un appel de méthode par le code de cette méthode, ce qui empêche par ricochet certaines optimisations. Cependant, cela a permis de rendre plus flexible le code que nous avons écrit dans l'encart 17-5 et que nous avons pu gérer dans l'encart 17-9, donc c'est un compromis à envisager.

Implémenter un patron de conception orienté-objet

Le *patron état* est un patron de conception orienté objet. Le point essentiel de ce patron est qu'une valeur possède un état interne qui est représenté par un ensemble *d'objets état*, et le comportement de la valeur change en fonction de son état interne. Les objets état partagent des fonctionnalités : en Rust, bien sûr, nous utilisons des structures et des traits plutôt que des objets et de l'héritage. Chaque objet état est responsable de son propre comportement et décide lorsqu'il doit changer pour un autre état. La valeur contenue dans un objet état ne sait rien sur les différents comportements des états et ne sait pas quand il va changer d'état.

L'utilisation du patron état signifie que lorsque les exigences métier du programme ont changé, nous n'avons pas besoin de changer le code à l'intérieur de l'objet état ou le code qui utilise l'objet. Nous avons juste besoin de modifier le code dans un des objets état pour changer son fonctionnement ou pour ajouter d'autres objets état. Voyons un exemple du patron état et comment l'utiliser en Rust.

Nous allons implémenter un processus de publication de billets de blogs de manière incrémentale. Les fonctionnalités finales du blog seront les suivantes :

1. Un billet de blog commence par un brouillon vide.
2. Lorsque le brouillon est terminé, une relecture du billet est demandée.
3. Lorsqu'un billet est approuvé, il est publié.
4. Seuls les billets de blog publiés retournent du contenu à afficher si bien que les billets non approuvés ne peuvent pas être publiés accidentellement.

Tous les autres changements effectués sur un billet n'auront pas d'effet. Par exemple, si nous essayons d'approuver un brouillon de billet de blog avant d'avoir demandé une relecture, le billet devrait rester à l'état de brouillon non publié.

L'encart 17-11 présente ce processus de publication sous forme de code : c'est un exemple d'utilisation de l'API que nous allons implémenter dans une crate de bibliothèque `blog`. Elle ne va pas encore se compiler car nous n'avons pas encore implémenté la crate `blog`.

Fichier : src/main.rs

```
use blog::Billet;

fn main() {
    let mut billet = Billet::new();

    billet.ajouter_texte("J'ai mangé une salade au déjeuner aujourd'hui");
    assert_eq!("", billet.contenu());

    billet.demander_relecture();
    assert_eq!("", billet.contenu());

    billet.approuver();
    assert_eq!("J'ai mangé une salade au déjeuner aujourd'hui",
billet.contenu());
}
```

Encart 17-11 : du code qui montre le comportement attendu de notre crate `blog`

Nous voulons permettre à l'utilisateur de créer un nouveau brouillon de billet de blog avec `Billet::new`. Nous voulons qu'il puisse ajouter du texte au billet de blog. Si nous essayons d'obtenir immédiatement le contenu du billet, avant qu'il ne soit relu, nous n'obtiendrons aucun texte car le billet est toujours un brouillon. Nous avons ajouté des `assert_eq!` dans le code pour les besoins de la démonstration. Un excellent test unitaire pour cela serait de vérifier qu'un brouillon de billet de blog retourne bien une chaîne de caractères vide à partir de la méthode `contenu`, mais nous n'allons pas écrire de tests pour cet exemple.

Ensuite, nous voulons permettre de demander une relecture du billet, et nous souhaitons que `contenu` retourne toujours une chaîne de caractères vide pendant que nous attendons la relecture. Lorsque la relecture du billet est approuvée, il doit être publié, ce qui signifie que le texte du billet doit être retourné lors de l'appel à `contenu`.

Remarquez que le seul type avec lequel nous interagissons avec la crate est le type `Billet`. Ce type va utiliser le patron état et va héberger une valeur qui sera un des trois objets état représentant les différents états par lesquels passe un billet : brouillon, en attente de relecture ou publié. Le changement d'un état à un autre sera géré en interne du type `Billet`. Les états vont changer en réponse à l'appel des méthodes de l'instance de `Billet` par les utilisateurs de notre bibliothèque qui n'auront donc pas à les gérer directement. Ainsi les utilisateurs ne peuvent pas faire d'erreur avec les états, comme celle de publier un billet avant qu'il ne soit relu par exemple.

Définir `Billet` et créer une nouvelle instance à l'état de brouillon

Commençons l'implémentation de la bibliothèque ! Nous savons que nous aurons besoin

d'une structure publique `Billet` qui héberge du contenu, donc nous allons commencer par définir cette structure ainsi qu'une fonction publique `new` qui lui est associée pour créer une instance de `Billet`, comme dans l'encart 17-12. Nous allons aussi créer un trait privé `Etat`. Ensuite `Billet` devra avoir un champ privé `etat` pour y loger une `Option<T>` contenant un objet trait de `Box<dyn Etat>`. Nous verrons plus tard l'intérêt du `Option<T>`.

Fichier : src/lib.rs

```
pub struct Billet {
    etat: Option<Box<dyn Etat>>,
    contenu: String,
}

impl Billet {
    pub fn new() -> Billet {
        Billet {
            etat: Some(Box::new(Brouillon {})),
            contenu: String::new(),
        }
    }
}

trait Etat {}

struct Brouillon {}

impl Etat for Brouillon {}
```

Encart 17-12 : définition d'une structure `Billet` et d'une fonction `new` qui crée une nouvelle instance de `Billet`, un trait `Etat` et une structure `Brouillon`

Le trait `Etat` définit le comportement partagé par plusieurs états de billet, et les états `Brouillon`, `EnRelecture` et `Publier` vont tous implémenter ce trait `Etat`. Pour l'instant, le trait n'a pas de méthode, et nous allons commencer par définir uniquement l'état `Brouillon` car c'est l'état dans lequel nous voulons que soit un nouveau billet lorsqu'il est créé.

Lorsque nous créons un nouveau `Billet`, nous assignons à son champ `etat` une valeur `Some` qui contient une `Box`. Cette `Box` pointe sur une nouvelle instance de la structure `Brouillon`. Cela garantira qu'à chaque fois que nous créons une nouvelle instance de `Billet`, elle commencera à l'état de brouillon. Comme le champ `etat` de `Billet` est privé, il n'y a pas d'autre manière de créer un `Billet` dans un autre état ! Dans la fonction `Billet::new`, nous assignons une nouvelle `String` vide au champ `contenu`.

Stocker le texte du contenu du billet

L'encart 17-11 a montré que nous souhaitons appeler une méthode `ajouter_texte` et lui passer un `&str` qui est ensuite ajouté au contenu textuel du billet de blog. Nous implémentons ceci avec une méthode plutôt que d'exposer publiquement le champ `contenu` avec `pub`. Cela signifie que nous pourrons implémenter une méthode plus tard qui va contrôler comment le champ `contenu` sera lu. La méthode `ajouter_texte` est assez simple, donc ajoutons son implémentation dans le bloc `Billet` de l'encart 17-13 :

Fichier : src/lib.rs

```
impl Billet {
    // -- partie masquée ici --
    pub fn ajouter_texte(&mut self, texte: &str) {
        self.contenu.push_str(texte);
    }
}
```

Encart 17-13 : implémentation de la méthode `ajouter_texte` pour ajouter du texte au `contenu` d'un billet

La méthode `ajouter_texte` prend en argument une référence mutable vers `self`, car nous changeons l'instance `Billet` sur laquelle nous appelons `ajouter_texte`. Nous faisons ensuite appel à `push_str` sur le `String` dans `contenu` et nous y envoyons l'argument `texte` pour l'ajouter au `contenu` déjà stocké. Ce comportement ne dépend pas de l'état dans lequel est le billet, donc cela ne fait pas partie du patron état. La méthode `ajouter_texte` n'interagit pas du tout avec le champ `etat`, mais c'est volontaire.

S'assurer que le contenu d'un brouillon est vide

Même si nous avons appelé `ajouter_texte` et ajouté du contenu dans notre billet, nous voulons que la méthode `contenu` retourne toujours une slice de chaîne de caractères vide car le billet est toujours à l'état de brouillon, comme le montre la ligne 7 de l'encart 17-11. Implémentons maintenant la méthode `contenu` de la manière la plus simple qui réponde à cette consigne : toujours retourner un slice de chaîne de caractères vide. Nous la changerons plus tard lorsque nous implémenterons la capacité de changer l'état d'un billet afin qu'il puisse être publié. Pour l'instant, les billets ne peuvent qu'être à l'état de brouillon, donc le contenu du billet devrait toujours être vide. L'encart 17-14 montre l'implémentation de ceci :

Fichier : src/lib.rs

```
impl Billet {
    // -- partie masquée ici --
    pub fn contenu(&self) -> &str {
        ""
    }
}
```

Encart 17-14 : ajout d'une implémentation de la méthode `contenu` sur `Billet` qui va toujours retourner une slice de chaîne de caractères vide

Avec cette méthode `contenu` ajoutée, tout ce qu'il y a dans l'encart 17-11 fonctionne comme prévu jusqu'à la ligne 7.

Demander une relecture du billet va changer son état

Ensuite, nous avons besoin d'ajouter une fonctionnalité pour demander la relecture d'un billet, qui devrait changer son état de `Brouillon` à `EnRelecture`. L'encart 17-15 montre ce code :

Fichier : src/lib.rs

```
impl Billet {
    // -- partie masquée ici --
    pub fn demander_relecture(&mut self) {
        if let Some(s) = self.etat.take() {
            self.etat = Some(s.demander_relecture())
        }
    }
}

trait Etat {
    fn demander_relecture(self: Box<Self>) -> Box<dyn Etat>;
}

struct Brouillon {}

impl Etat for Brouillon {
    fn demander_relecture(self: Box<Self>) -> Box<dyn Etat> {
        Box::new(EnRelecture {})
    }
}

struct EnRelecture {}

impl Etat for EnRelecture {
    fn demander_relecture(self: Box<Self>) -> Box<dyn Etat> {
        self
    }
}
```

Encart 17-15 : implémentation des méthodes `demander_relecture` sur `Billet` et le trait `Etat`

Nous installons la méthode publique `demander_relecture` sur `Billet` qui va prendre en argument une référence mutable à `self`. Ensuite nous appelons la méthode interne `demander_relecture` sur l'état interne de `Billet`, et cette deuxième méthode `demander_relecture` consomme l'état en cours et applique un nouvel état.

Nous avons ajouté la méthode `demander_relecture` sur le trait `Etat` ; tous les types qui implémentent le trait vont maintenant devoir implémenter la méthode `demander_relecture`. Remarquez qu'au lieu d'avoir `self`, `&self`, ou `&mut self` en premier paramètre de la méthode, nous avons `self: Box<Self>`. Cette syntaxe signifie que la méthode est valide uniquement lorsqu'on l'appelle sur une `Box` qui contient ce type. Cette syntaxe prend possession de `Box<Self>`, ce qui annule l'ancien état du `Billet` qui peut changer pour un nouvel état.

Pour consommer l'ancien état, la méthode `demander_relecture` a besoin de prendre possession de la valeur d'état. C'est ce à quoi sert le `Option` dans le champ `etat` de

`Billet` : nous faisons appel à la méthode `take` pour obtenir la valeur dans le `Some` du champ `etat` et le remplacer par `None`, car Rust ne nous permet pas d'avoir des champs non renseignés dans des structures. Cela nous permet d'extraire la valeur de `etat` d'un `Billet`, plutôt que de l'emprunter. Ensuite, nous allons réaffecter le résultat de cette opération à `etat` du `Billet` concerné.

Nous devons assigner temporairement `None` à `etat` plutôt que de lui donner directement avec du code tel que `self.etat = self.etat.demander_relecture();` car nous voulons prendre possession de la valeur `etat`. Cela garantit que `Billet` ne peut pas utiliser l'ancienne valeur de `etat` après qu'on ait changé cet état.

La méthode `demander_relecture` sur `Brouillon` doit retourner une nouvelle instance d'une structure `EnRelecture` dans une `Box`, qui représente l'état lorsqu'un billet est en attente de relecture. La structure `EnRelecture` implémente elle aussi la méthode `demander_relecture` mais ne fait aucune modification. A la place, elle se retourne elle-même, car lorsque nous demandons une relecture sur un billet déjà à l'état `EnRelecture`, il doit rester à l'état `EnRelecture`.

Désormais nous commençons à voir les avantages du patron état : la méthode `demander_relecture` sur `Billet` est la même peu importe la valeur de son `etat`. Chaque état est maître de son fonctionnement.

Nous allons conserver la méthode `contenu` sur `Billet` comme elle est, elle va donc continuer à retourner une slice de chaîne de caractères vide. Nous pouvons maintenant avoir un `Billet` à l'état `Brouillon` ou `EnRelecture`, mais nous voulons qu'il suive le même comportement lorsqu'il est dans l'état `EnRelecture`. L'encart 17-11 fonctionne maintenant jusqu'à la ligne 10 !

Ajouter une méthode `approuver` qui change le comportement de `contenu`

La méthode `approuver` ressemble à la méthode `demander_relecture` : elle va changer `etat` pour lui donner la valeur que l'état courant retournera lorsqu'il sera approuvé, comme le montre l'encart 17-16 :

Fichier : src/lib.rs

```
impl Billet {
    // -- partie masquée ici --
    pub fn approuver(&mut self) {
        if let Some(s) = self.etat.take() {
            self.etat = Some(s.approuver())
        }
    }
}

trait Etat {
    fn demander_relecture(self: Box<Self>) -> Box<dyn Etat>;
    fn approuver(self: Box<Self>) -> Box<dyn Etat>;
}

struct Brouillon {}

impl Etat for Brouillon {
    // -- partie masquée ici --
    fn approuver(self: Box<Self>) -> Box<dyn Etat> {
        self
    }
}

struct EnRelecture {}

impl Etat for EnRelecture {
    // -- partie masquée ici --
    fn approuver(self: Box<Self>) -> Box<dyn Etat> {
        Box::new(Publier {})
    }
}

struct Publier {}

impl Etat for Publier {
    fn demander_relecture(self: Box<Self>) -> Box<dyn Etat> {
        self
    }

    fn approuver(self: Box<Self>) -> Box<dyn Etat> {
        self
    }
}
```

Encart 17-16 : implémentation de la méthode `approuver` sur `Billet` et sur le trait `Etat`

Nous avons ajouté la méthode `approuver` au trait `Etat` et ajouté une nouvelle structure `Publier`, qui implémente `Etat`.

Comme pour la façon de fonctionner de `demander_relecture` sur `EnRelecture`, si nous faisons appel à la méthode `approuver` sur un `Brouillon`, cela n'aura pas d'effet car

`approuver` va retourner `self`. Lorsque nous appellerons `approuver` sur `EnRelecture`, elle va retourner une nouvelle instance de la structure `Publier` dans une instance de `Box`. La structure `Publier` implémente le trait `Etat`, et pour chacune des méthodes `demander_relecture` et `approuver`, elle va retourner elle-même, car le billet doit rester à l'état `Publier` dans ce cas-là.

Nous devons maintenant modifier la méthode `contenu` sur `Billet`. Nous souhaitons que la valeur retournée par `contenu` dépende de l'état actuel du `Billet`, donc nous allons faire en sorte que le `Billet` délègue sa logique à une méthode `contenu` défini sur son `etat`, comme dans l'encart 17-17 :

Fichier : src/lib.rs

```
impl Billet {
    // -- partie masquée ici --
    pub fn contenu(&self) -> &str {
        self.etat.as_ref().unwrap().contenu(self)
    }
    // -- partie masquée ici --
}
```

Encart 17-17 : correction de la méthode `contenu` de `Billet` afin qu'elle délègue à la méthode `contenu` de `Etat`

Comme notre but est de conserver toutes ces règles dans les structures qui implémentent `Etat`, nous appelons une méthode `contenu` sur la valeur de `etat` et nous lui passons en argument l'instance du billet (avec le `self`). Nous retournons ensuite la valeur retournée par la méthode `contenu` sur la valeur de `etat`.

Nous faisons appel à la méthode `as_ref` sur `Option` car nous voulons une référence vers la valeur dans `Option` plutôt que d'en prendre possession. Comme `etat` est un `Option<Box<dyn Etat>>`, lorsque nous faisons appel à `as_ref`, une `Option<&Box<dyn Etat>>` est retournée. Si nous n'avions pas fait appel à `as_ref`, nous aurions obtenu une erreur car nous ne pouvons pas déplacer `etat` de `&self`, lui-même est emprunté et provenant des paramètres de la fonction.

Nous faisons ensuite appel à la méthode `unwrap`, mais nous savons qu'elle ne va jamais paniquer, car nous savons que les méthodes sur `Billet` vont garantir que `etat` contiendra toujours une valeur `Some` lorsqu'elles seront utilisées. C'est un des cas dont nous avons parlé dans une section du chapitre 9 lorsque nous savions qu'une valeur `None` ne serait jamais possible, même si le compilateur n'est pas capable de le comprendre.

A partir de là, lorsque nous faisons appel à `contenu` sur `&Box<dyn Etat>`, l'extrapolation de

déréférencement va s'appliquer sur le `&` et le `Box` pour que la méthode `contenu` puisse finalement être appelée sur le type qui implémente le trait `Etat`. Cela signifie que nous devons ajouter `contenu` à la définition du trait `Etat`, et que c'est ici que nous allons placer la logique pour le contenu à retourner en fonction de l'état nous avons, comme le montre l'encart 17-18 :

Fichier : src/lib.rs

```
trait Etat {
    // -- partie masquée ici --
    fn contenu<'a>(&self, billet: &'a Billet) -> &'a str {
        ""
    }
}

// -- partie masquée ici --
struct Publier {}

impl Etat for Publier {
    // -- partie masquée ici --
    fn contenu<'a>(&self, billet: &'a Billet) -> &'a str {
        &billet.contenu
    }
}
```

Encart 17-18 : ajout de la méthode `contenu` sur le trait `Etat`

Nous avons ajouté une implémentation par défaut pour la méthode `contenu` qui retourne une slice de chaîne de caractères vide. Cela nous permet de ne pas avoir à implémenter `contenu` sur les structures `Brouillon` et `EnRelecture`. La structure `Publier` va remplacer la méthode `contenu` et retourner la valeur présente dans `billet.contenu`.

Remarquez aussi que nous devons annoter des durées de vie sur cette méthode, comme nous l'avons vu au chapitre 10. Nous allons prendre en argument une référence au `billet` et retourner une référence à une partie de ce `billet`, donc la durée de vie retournée par la référence est liée à la durée de vie de l'argument `billet`.

Et nous avons maintenant terminé, tout le code de l'encart 17-11 fonctionne désormais ! Nous avons implémenté le patron état avec les règles de notre processus de publication définies pour notre blog. La logique des règles est intégrée dans les objets état plutôt que d'être dispersée un peu partout dans `Billet`.

Les inconvénients du patron état

Nous avons démontré que Rust est capable d'implémenter le patron état qui est orienté objet pour regrouper les différents types de comportement qu'un billet doit avoir à chaque état. Les méthodes sur `Billet` ne savent rien des différents comportements. De la manière dont nous avons organisé le code, nous n'avons qu'à regarder à un seul endroit pour connaître les différents comportements qu'un billet publié va suivre : l'implémentation du trait `Etat` sur la structure `Publier`.

Si nous avions utilisé une autre façon d'implémenter ces règles sans utiliser le patron état, nous aurions dû utiliser des expressions `match` dans les méthodes de `Billet` ou même dans le code du `main` qui vérifie l'état du billet et les comportements associés aux changements d'états. Cela aurait eu pour conséquence d'avoir à regarder à différents endroits pour comprendre toutes les conséquences de la publication d'un billet ! Et ce code grossirait au fur et à mesure que nous ajouterions des états : chaque expression `match` devrait avoir des nouvelles branches pour ces nouveaux états.

Avec le patron état, les méthodes de `Billet` et les endroits où nous utilisons `Billet` n'ont pas besoin d'expressions `match`, et pour ajouter un nouvel état, nous avons seulement besoin d'ajouter une nouvelle structure et d'implémenter les méthodes du trait sur cette structure.

L'implémentation qui utilise le patron état est facile à améliorer pour ajouter plus de fonctionnalités. Pour découvrir la simplicité de maintenance du code qui utilise le patron état, essayez d'accomplir certaines de ces suggestions :

- Ajouter une méthode `rejeter` qui fait retourner l'état d'un billet de `EnRelecture` à `Brouillon`.
- Attendre deux appels à `approuver` avant que l'état puisse être changé en `Publier`.
- Permettre aux utilisateurs d'ajouter du contenu textuel uniquement lorsqu'un billet est à l'état `Brouillon`. Indice : rendre l'objet état responsable de ce qui peut changer dans le contenu mais pas responsable de la modification de `Billet`.

Un inconvénient du patron état est que comme les états implémentent les transitions entre les états, certains des états sont couplés entre eux. Si nous ajoutons un nouvel état entre `EnRelecture` et `Publier`, `Planifier` par exemple, nous devrons alors changer le code dans `EnRelecture` pour qu'il passe ensuite à l'état `Planifier` au lieu de `Publier`. Cela représenterait moins de travail si `EnRelecture` n'avait pas besoin de changer lorsqu'on ajoute un nouvel état, mais cela signifierait alors qu'il faudrait changer de patron.

Un autre inconvénient est que nous avons de la logique en double. Pour éviter ces doublons, nous devrions essayer de faire en sorte que les méthodes `demander_relecture` et `approuver` qui retournent `self` deviennent les implémentations par défaut sur le trait `Etat` ; cependant, cela violerait la sûreté des objets, car le trait ne sait pas ce qu'est

exactement `self`. Nous voulons pouvoir utiliser `Etat` en tant qu'objet trait, donc nous avons besoin que ses méthodes soient sûres pour les objets.

Nous avons aussi des doublons dans le code des méthodes `demander_relecture` et `approuver` sur `Billet`. Ces deux méthodes délèguent leur travail à la même méthode de la valeur du champ `etat` de type `Option` et assignent la nouvelle valeur du même champ `etat` à la fin. Si nous avions beaucoup de méthodes sur `Billet` qui suivaient cette logique, nous devrions envisager de définir une macro pour éviter cette répétition (voir la section dédiée dans le chapitre 19).

En implémentant le patron état exactement comme il est défini pour les langages orientés-objet, nous ne profitons pas pleinement des avantages de Rust. Voyons voir si nous pouvons faire quelques changements pour que la crate `blog` puisse lever des erreurs dès la compilation lorsqu'elle aura détecté des états ou des transitions invalides.

Implémenter les états et les comportements avec des types

Nous allons vous montrer comment repenser le patron état pour qu'il offre des compromis différents. Plutôt que d'encapsuler complètement les états et les transitions, faisant que le code externe ne puissent pas les connaître, nous allons coder ces états sous forme de différents types. En conséquence, le système de vérification de type de Rust va empêcher toute tentative d'utilisation des brouillons de billets là où seuls des billets publiés sont autorisés, en provoquant une erreur de compilation.

Considérons la première partie du `main` de l'encart 17-11 :

Fichier : src/main.rs

```
fn main() {
    let mut billet = Billet::new();

    billet.ajouter_texte("J'ai mangé une salade au déjeuner aujourd'hui");
    assert_eq!("", billet.contenu());
}
```

Nous pouvons toujours créer de nouveaux billets à l'état de brouillon en utilisant `Billet::new` et ajouter du texte au contenu du billet. Mais au lieu d'avoir une méthode `contenu` sur un brouillon de billet qui retourne une chaîne de caractères vide, nous faisons en sorte que les brouillons de billets n'aient même pas de méthode `contenu`. Ainsi, si nous essayons de récupérer le contenu d'un brouillon de billet, nous obtenons une erreur de compilation qui nous informera que la méthode n'existe pas. Finalement, il nous sera impossible de publier le contenu d'un brouillon de billet en production, car ce code ne se compilera même pas. L'encart 17-19 nous propose les définitions d'une structure `Billet` et

d'une structure `BrouillonDeBillet` ainsi que leurs méthodes :

Fichier : src/lib.rs

```
pub struct Billet {
    contenu: String,
}

pub struct BrouillonDeBillet {
    contenu: String,
}

impl Billet {
    pub fn new() -> BrouillonDeBillet {
        BrouillonDeBillet {
            contenu: String::new(),
        }
    }

    pub fn contenu(&self) -> &str {
        &self.contenu
    }
}

impl BrouillonDeBillet {
    pub fn ajouter_texte(&mut self, texte: &str) {
        self.contenu.push_str(texte);
    }
}
```

Encart 17-19 : un `Billet` avec une méthode `contenu` et un `BrouillonDeBillet` sans méthode `contenu`

Les deux structures `Billet` et `BrouillonDeBillet` ont un champ privé `contenu` qui stocke le texte du billet de blog. Les structures n'ont plus le champ `etat` car nous avons déplacé la signification de l'état directement dans le nom de ces types de structures. La structure `Billet` représente un billet publié et possède une méthode `contenu` qui retourne le `contenu`.

Nous avons toujours la fonction `Billet::new`, mais au lieu de retourner une instance de `Billet`, elle va retourner une instance de `BrouillonDeBillet`. Comme `contenu` est privé et qu'il n'y a pas de fonction qui retourne `Billet`, il ne sera pas possible pour le moment de créer une instance de `Billet`.

La structure `BrouillonDeBillet` a une méthode `ajouter_texte`, donc nous pouvons ajouter du texte à `contenu` comme nous le faisions avant, mais remarquez toutefois que `BrouillonDeBillet` n'a pas de méthode `contenu` de définie ! Donc pour l'instant le

programme s'assure que tous les billets démarrent à l'état de brouillon et que les brouillons ne proposent pas de contenu à publier. Toute tentative d'outre-passer ces contraintes va déclencher une erreur de compilation.

Implémenter les changements d'état en tant que changement de type

Donc, comment publier un billet ? Nous voulons renforcer la règle qui dit qu'un brouillon de billet doit être relu et approuvé avant de pouvoir être publié. Un billet à l'état de relecture doit continuer à ne pas montrer son contenu. Implémentons ces contraintes en introduisant une nouvelle structure, `BilletEnRelecture`, en définissant la méthode `demander_relecture` sur `BrouillonDeBillet` retournant un `BilletEnRelecture`, et en définissant une méthode `approuver` sur `BilletEnRelecture` pour qu'elle retourne un `Billet`, comme le propose l'encart 17-20 :

Fichier : src/lib.rs

```
impl BrouillonDeBillet {
    // -- partie masquée ici --
    pub fn demander_relecture(self) -> BilletEnRelecture {
        BilletEnRelecture {
            contenu: self.contenu,
        }
    }
}

pub struct BilletEnRelecture {
    contenu: String,
}

impl BilletEnRelecture {
    pub fn approuver(self) -> Billet {
        Billet {
            contenu: self.contenu,
        }
    }
}
```

Encart 17-20 : ajout d'un `BilletEnRelecture` qui est créé par l'appel à `demander_relecture` sur `BrouillonDeBillet`, ainsi qu'une méthode `approuver` qui transforme un `BilletEnRelecture` en `Billet` publié

Les méthodes `demander_relecture` et `approuver` prennent possession de `self`, ce qui consomme les instances de `BrouillonDeBillet` et de `BilletEnRelecture` pour les transformer respectivement en `BilletEnRelecture` et en `Billet`. Ainsi, il ne restera plus d'instances de `BrouillonDeBillet` après avoir appelé `approuver` sur elles, et ainsi de suite.

La structure `BilletEnRelecture` n'a pas de méthode `contenu` qui lui est définie, donc si on essaye de lire son contenu, on obtient une erreur de compilation, comme avec `BrouillonDeBillet`. Comme la seule manière d'obtenir une instance de `Billet` qui a une méthode `contenu` de définie est d'appeler la méthode `approuver` sur un `BilletEnRelecture`, et que la seule manière d'obtenir un `BilletEnRelecture` est d'appeler la méthode `demander_relecture` sur un `BrouillonDeBillet`, nous avons désormais intégré le processus de publication des billets de blog avec le système de type.

Mais nous devons aussi faire quelques petits changements dans le `main`. Les méthodes `demander_relecture` et `approuver` retournent des nouvelles instances au lieu de modifier la structure sur laquelle elles ont été appelées, donc nous devons ajouter des assignations de masquage `let billet =` pour stocker les nouvelles instances retournées. Nous ne pouvons pas non plus vérifier que le contenu des brouillons de billets et de ceux en cours de relecture sont bien vides, donc nous n'avons plus besoin des vérifications associées : en effet, nous ne pouvons plus compiler du code qui essaye d'utiliser le contenu d'un billet dans ces états. Le code du `main` mis à jour est présenté dans l'encart 17-21 :

Fichier : src/main.rs

```
use blog::Billet;

fn main() {
    let mut billet = Billet::new();

    billet.ajouter_texte("J'ai mangé une salade au déjeuner aujourd'hui");

    let billet = billet.demander_relecture();

    let billet = billet.approuver();

    assert_eq!("J'ai mangé une salade au déjeuner aujourd'hui",
billet.contenu());
}
```

Encart 17-21 : modification de `main` pour utiliser la nouvelle implémentation du processus de publication de billet de blog

Les modifications que nous avons eu besoin de faire à `main` pour réassigner `billet` impliquent que cette implémentation ne suit plus exactement le patron état orienté-objet : les changements d'états ne sont plus totalement intégrés dans l'implémentation de `Billet`. Cependant, nous avons obtenu que les états invalides sont désormais impossibles grâce au système de types et à la vérification de type qui s'effectue à la compilation ! Cela garantit que certains bogues, comme l'affichage du contenu d'un billet non publié, seront détectés avant d'arriver en production.

Essayez d'implémenter les exigences fonctionnelles supplémentaires suggérées dans la liste présente au début de cette section, sur la crate `blog` dans l'état où elle était après l'encart 17-20, afin de vous faire une idée sur cette façon de concevoir le code. Notez aussi que certaines de ces exigences pourraient déjà être implémentées implicitement du fait de cette conception.

Nous avons vu que même si Rust est capable d'implémenter des patrons de conception orientés-objet, d'autres patrons, tel qu'intégrer l'état dans le système de type, sont également possibles en Rust. Ces patrons présentent différents avantages et inconvénients. Bien que vous puissiez être très familier avec les patrons orientés-objet, vous gagnerez à repenser les choses pour tirer avantage des fonctionnalités de Rust, telles que la détection de certains bogues à la compilation. Les patrons orientés-objet ne sont pas toujours la meilleure solution en Rust à cause de certaines de ses fonctionnalités, comme la possession, que les langages orientés-objet n'ont pas.

Résumé

Que vous pensiez ou non que Rust est un langage orienté-objet après avoir lu ce chapitre, vous savez maintenant que vous pouvez utiliser les objets trait pour pouvoir obtenir certaines fonctionnalités orienté-objet en Rust. La répartition dynamique peut offrir de la flexibilité à votre code en échange d'une perte de performances à l'exécution. Vous pouvez utiliser cette flexibilité pour implémenter des patrons orientés-objet qui facilitent la maintenance de votre code. Rust offre d'autres fonctionnalités, comme la possession, que les langages orientés-objet n'ont pas. L'utilisation d'un patron orienté-objet n'est pas toujours la meilleure manière de tirer parti des avantages de Rust, mais cela reste une option disponible.

Dans le chapitre suivant, nous allons étudier les motifs, qui constituent une autre des fonctionnalités de Rust et apportent beaucoup de flexibilité. Nous les avons abordés brièvement dans le livre, mais nous n'avons pas encore vu tout leur potentiel. C'est parti !

Les motifs et le filtrage par motif

Les motifs sont une syntaxe spéciale de Rust permettant de filtrer selon la structure des types, qu'elle soit simple ou complexe. L'utilisation de motifs conjointement avec des expressions `match` et d'autres constructions vous donne davantage de maîtrise sur le flux de contrôle de votre programme. Un motif est constitué d'une combinaison de :

- littéraux
- tableaux de structures, énumérations, structures ou tuples
- variables
- jokers
- espaces réservés

Ces composants décrivent la forme de la donnée avec laquelle nous travaillons, que nous comparons alors à différents motifs de valeurs pour déterminer si notre programme dispose de la donnée appropriée pour exécuter une partie spécifique de code.

Pour utiliser un motif, nous le comparons à une certaine valeur. Si le motif correspond à la valeur, nous utilisons les éléments présents dans la valeur pour notre code. Rappelez-vous que les expressions `match` du chapitre 6 utilisaient les motifs, comme pour la machine à trier la monnaie par exemple. Si la valeur correspondait à la forme d'un motif, nous pouvions utiliser le nom de la pièce. Sinon, le code associé au motif n'était pas exécuté.

Ce chapitre sert de référence pour tout ce qui concerne les motifs. Nous allons voir les moments appropriés pour utiliser les motifs, les différences entre les motifs réfutables et irréfutables ainsi que les différentes syntaxes de motifs que vous pouvez rencontrer. A la fin de ce chapitre, vous saurez comment utiliser les motifs pour exprimer clairement de nombreux concepts.

Tous les endroits où les motifs peuvent être utilisés

Les motifs apparaissent dans de nombreux endroits en Rust, et vous en avez utilisé beaucoup sans vous en rendre compte ! Cette section va présenter les différentes situations où l'utilisation des motifs est appropriée.

Les branches des `match`

Comme nous l'avons vu au chapitre 6, nous utilisons les motifs dans les branches des expressions `match`. Techniquement, les expressions `match` sont définies avec le mot-clé `match`, une valeur sur laquelle procéder et une ou plusieurs branches qui constituent un motif, chacune associée à une expression à exécuter si la valeur correspond au motif de la branche, comme ceci :

```
match VALEUR {
    MOTIF => EXPRESSION,
    MOTIF => EXPRESSION,
    MOTIF => EXPRESSION,
}
```

L'une des conditions à respecter pour les expressions `match` est qu'elles doivent être *exhaustives* dans le sens où toutes les valeurs possibles de la valeur présente dans l'expression `match` doivent être prises en compte. Une façon de s'assurer que vous avez couvert toutes les possibilités est d'avoir un motif passe-partout pour la dernière branche : par exemple, une valeur quelconque ne pourra jamais échouer car la dernière branche permet de couvrir tous les autres cas possibles.

Le motif spécifique `_` va correspondre à tout, mais il ne fournira jamais de variable, donc il est souvent utilisé dans la dernière branche. Le motif `_` peut par exemple être utile lorsque vous souhaitez ignorer toutes les autres valeurs qui n'ont pas été listées. Nous allons voir plus en détail le motif `_` dans une section plus tard dans ce chapitre.

Les expressions conditionnelles `if let`

Au chapitre 6, nous avons vu comment utiliser les expressions `if let`, principalement pour pouvoir écrire l'équivalent d'un `match` qui ne correspond qu'à un seul cas. Accessoirement, `if let` peut avoir un `else` correspondant au code à exécuter si le motif du `if let` ne correspond pas au premier critère.

L'encart 18-1 montre qu'il est aussi possible de conjuguer les expressions `if let`, `else if`

et `else if let`. Faire ceci nous donne plus de flexibilité qu'une expression `match` dans laquelle nous ne pouvons fournir qu'une seule valeur à comparer avec les motifs. De plus, dans une série de branches `if let`, `else if` et `else if let`, les conditions n'ont pas besoin d'être en rapport les unes avec les autres.

Le code de l'encart 18-1 montre une série de vérifications pour quelques conditions qui décident quelle devrait être la couleur de fond. Pour cet exemple, nous avons créé les variables avec des valeurs codées en dur qu'un vrai programme devrait recevoir d'une saisie d'un utilisateur.

Fichier : src/main.rs

```
fn main() {
    let couleur_favorite: Option<&str> = None;
    let on_est_mardi = false;
    let age: Result<u8, _> = "34".parse();

    if let Some(couleur) = couleur_favorite {
        println!("Utilisation de votre couleur favorite, {}, comme couleur de
fond", couleur);
    } else if on_est_mardi {
        println!("Mardi, c'est le jour du vert !");
    } else if let Ok(age) = age {
        if age > 30 {
            println!("Utilisation du violet comme couleur de fond");
        } else {
            println!("Utilisation de l'orange comme couleur de fond");
        }
    } else {
        println!("Utilisation du bleu comme couleur de fond");
    }
}
```

Encart 18-1 : mélange de `if let`, `else if`, `else if let`, et `else`

Si l'utilisateur renseigne une couleur favorite, c'est cette couleur qui devient la couleur de fond. Sinon, si nous sommes mardi, la couleur de fond sera le vert. Sinon, si l'utilisateur a renseigné son âge dans une chaîne de caractères et que nous pouvons l'interpréter comme un nombre avec succès, la couleur de fond sera soit le violet, soit l'orange en fonction de la valeur de ce nombre. Enfin, si aucune de ces conditions ne s'applique, la couleur de fond sera le bleu.

Cette structure conditionnelle nous permet de répondre à des conditions complexes. Avec les valeurs codées en dur que nous avons ici, cet exemple devrait afficher `Utilisation du violet comme couleur de fond`.

Vous pouvez constater que le `if let` nous permet d'utiliser les variables masquées de la

même manière que le font les branches `match` : la ligne `if let Ok(age) = age` crée une nouvelle variable masquée `age` qui contient la valeur présente dans la variante `Ok`. Cela signifie que nous devons placer la condition `if age > 30` à l'intérieur de ce bloc : nous ne pouvons pas combiner ces deux conditions dans une seule `if let Ok(age) = age && age > 30`. La variable masquée `age` que nous souhaitons comparer à 30 n'est pas encore en vigueur tant que la nouvelle portée entre les accolades n'a pas commencée.

Le désavantage de l'utilisation des expressions `if let` est que le compilateur ne vérifie pas l'exhaustivité contrairement à une expression `match`. Si nous avions enlevé le dernier bloc `else`, oubliant ainsi de gérer certains cas, le compilateur n'aurait pas pu nous prévenir d'un possible bogue de logique.

les boucles conditionelles `while let`

Comme les constructions `if let`, les boucles conditionnelles `while let` permettent à une boucle `while` de s'exécuter aussi longtemps qu'un motif continue à correspondre.
L'exemple dans l'encart 18-2 montre une boucle `while let` qui utilise un vecteur comme une pile et affiche les valeurs du vecteur dans l'ordre opposé à celui dans lequel elles ont été insérées.

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

pile.push(1);
pile.push(2);
pile.push(3);

while let Some(donnee_du_haut) = pile.pop() {
    println!("{}", donnee_du_haut);
}
```

Encart 18-2 : utilisation d'une boucle `while let` pour afficher les valeurs aussi longtemps que `pile.pop()` retourne une `Some`

Cet exemple affiche 3, 2 puis ensuite 1. La méthode `pop` sort le dernier élément du vecteur et retourne `Some(valeur)`. Si le vecteur est vide, `pop` retourne alors `None`. La boucle `while` continue à exécuter le code de son bloc aussi longtemps que `pop` retourne un `Some`. Lorsque `pop` retournera `None`, la boucle s'arrêtera. Nous pouvons utiliser `while let` pour extraire tous les éléments de la pile.

Les boucles `for`

Au chapitre 3, nous avions mentionné que la boucle `for` était la construction de boucle la plus utilisée dans du code Rust, mais nous n'avons pas encore abordé le motif que prend `for`. Dans une boucle `for`, le motif est la valeur qui suit directement le mot-clé `for`, de sorte que `x` est le motif dans `for x in y`.

L'encart 18-3 montre comment utiliser un motif dans une boucle `for` pour déstructurer, ou décomposer, un tuple faisant partie de la boucle `for`.

```
let v = vec!['a', 'b', 'c'];

for (indice, valeur) in v.iter().enumerate() {
    println!("{} est à l'indice {}", valeur, indice);
}
```

Encart 18-3 : utilisation d'un motif dans une boucle `for` pour déstructurer un tuple

Le code de l'encart 18-3 va afficher ceci :

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
    Finished dev [unoptimized + debuginfo] target(s) in 0.52s
     Running `target/debug/patterns`
a est à l'indice 0
b est à l'indice 1
c est à l'indice 2
```

Nous avons utilisé la méthode `enumerate` pour produire une valeur et son indice à partir d'un itérateur que nous avons placé dans un tuple. La premiere valeur produite est le tuple `(0, 'a')`. Comme cette valeur correspond au motif `(indice, valeur)`, `indice` se voit affecter `0`, `valeur` se voit affecter `'a'`, provoquant l'affichage de la première ligne sur la sortie.

Les instructions `let`

Avant d'arriver à ce chapitre, nous n'avions abordé explicitement l'utilisation des motifs qu'avec `match` et `if let`, mais en réalité, nous avions utilisé les motifs dans d'autres endroits, y compris dans les instructions `let`. Par exemple, considérons l'assignation de la variable suivante avec `let` :

```
let x = 5;
```

Tout au long de ce livre, nous avons utilisé `let` de cette manière des centaines de fois, et

malgré tout vous ne vous êtes probablement pas rendu compte que vous utilisiez les motifs ! Plus formellement, une instruction `let` ressemble à ceci :

```
let MOTIF = EXPRESSION;
```

Dans des instructions telles que `let x = 5;` avec un nom de variable dans l'emplacement `MOTIF`, le nom de la variable n'est juste qu'une forme particulièrement simple de motif. Rust compare l'expression avec le motif et assigne tous les noms qu'il trouve. Dans l'exemple `let x = 5;`, `x` est un motif qui signifie “relie ce qui correspond ici à la variable `x`”. Puisque le nom `x` constitue un motif complet, il signifie exactement “relie tout ce qui suit à la variable `x`, quelle qu'en soit la valeur”.

Pour comprendre plus clairement l'aspect filtrage par motif de `let`, examinons l'encart 18-4, qui utilise un motif `let` pour destructurer un tuple.

```
let (x, y, z) = (1, 2, 3);
```

Encart 18-4 : utilisation d'un motif pour destructurer un tuple et créer trois variables à la fois

Ici, nous avons fait correspondre un tuple à un motif. Rust compare la valeur `(1, 2, 3)` avec le motif `(x, y, z)` et constate que la valeur correspond au motif, donc Rust relie `1` à `x`, `2` à `y` et `3` à `z`. Vous pouvez ainsi considérer que ce motif de tuple encapsule trois variables individuelles.

Si le nombre d'éléments dans le motif ne correspond pas au nombre d'éléments dans le tuple, le type global ne va pas correspondre et nous allons obtenir une erreur de compilation. Par exemple, l'encart 18-5 montre une tentative de déstructurer un tuple avec trois éléments dans deux variables, ce qui ne va pas fonctionner.

```
let (x, y) = (1, 2, 3);
```

Encart 18-5 : construction incorrecte d'un motif dont les variables ne vont pas correspondre au nombre d'éléments présents dans le tuple

Si vous essayez de compiler ce code, vous obtiendrez cette erreur de type :

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error[E0308]: mismatched types
 --> src/main.rs:2:9
  |
2 |     let (x, y) = (1, 2, 3);
  |         ^^^^^^   --------- this expression has type `({integer}, {integer},
{integer})`
  |         |
  |         expected a tuple with 3 elements, found one with 2 elements
  |
  = note: expected tuple `({integer}, {integer}, {integer})`
             found tuple `(_, _)`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `patterns` due to previous error
```

Si nous souhaitons ignorer une ou plusieurs valeurs dans un tuple, nous pouvons utiliser `_` ou `..`, comme vous allez le voir à la dernière section de ce chapitre. Si le problème est que nous avons trop de variables dans le motif, la solution pour faire correspondre les types consiste à enlever des variables de façon à ce que le nombre de variables corresponde au nombre d'éléments présents dans le tuple.

Les paramètres de fonctions

Les paramètres de fonctions peuvent aussi être des motifs. Le code de l'encart 18-6 déclare une fonction `foo` qui prend un paramètre `x` de type `i32`.

```
fn fonction(x: i32) {
    // le code se place ici
}
```

Encart 18-6 : une signature de fonction qui utilise des motifs dans ses paramètres

La partie `x` est un motif ! Comme nous l'avons dit pour `let`, nous pouvons faire correspondre le motif avec un tuple dans les arguments de la fonction. L'encart 18-7 déstructure les valeurs d'un tuple que nous passons en argument d'une fonction.

Fichier : src/main.rs

```
fn afficher_coordonnees(&(x, y): &(i32, i32)) {
    println!("Coordonnées actuelles : ({}, {})", x, y);
}

fn main() {
    let point = (3, 5);
    afficher_coordonnees(&point);
}
```

Encart 18-7 : une fonction avec des paramètres qui déstructurent un tuple

Ce code affiche `Coordonées actuelles : (3, 5)`. Les valeurs `&(3, 5)` correspondent au motif `&(x, y)`, donc `x` a la valeur `3` et `y` a la valeur `5`.

Nous pouvons aussi utiliser les motifs dans la liste des paramètres d'une fermeture de la même manière que dans la liste des paramètres d'une fonction, car les fermetures sont similaires aux fonctions, comme nous l'avons dit au chapitre 13.

A présent, vous avez vu plusieurs façons d'utiliser les motifs, mais les motifs ne fonctionnent pas de la même manière dans toutes les situations où nous les utilisons. Des fois, le motif sera irréfutable ; d'autres fois, il sera réfutable. C'est ce que nous allons voir tout de suite.

La réfutabilité : lorsqu'un motif peut échouer à correspondre

Les motifs se divisent en deux catégories : réfutables et irréfutables. Les motifs qui vont correspondre à n'importe quelle valeur qu'on lui passe sont *irréfutables*. Un exemple serait le `x` dans l'instruction `let x = 5;` car `x` correspond à tout ce qui est possible de sorte que la correspondance ne puisse pas échouer. Les motifs pour lesquels la correspondance peut échouer pour certains valeurs sont *réfutables*. Un exemple serait `Some(x)` dans l'expression `if let Some(x) = une_valeur` car si la valeur dans la variable `une_valeur` est `None` au lieu de `Some`, le motif `Some(x)` ne correspondra pas.

Les paramètres de fonctions, les instructions `let` et les boucles `for` ne peuvent accepter que des motifs irréfutables, car le programme ne peut rien faire d'autre lorsque les valeurs ne correspondent pas. Les expressions `if let` et `while let` acceptent les motifs réfutables et irréfutables, mais dans le second cas, le compilateur affichera une mise en garde car, par définition, ces expressions sont destinées à gérer un problème éventuel : le but des conditions est de se comporter différemment en fonction de la réussite ou de l'échec.

De manière générale, vous ne devriez pas avoir à vous soucier des différences entre les motifs réfutables et irréfutables ; en revanche, vous devez vous familiariser avec le concept de réfutabilité afin que vous puissiez comprendre ce qui se passe lorsque vous le verrez apparaître dans un message d'erreur. Dans ce cas, vous allez avoir besoin de changer soit le motif, soit la construction avec laquelle vous l'utilisez, selon le comportement attendu du code.

Examinons un exemple de ce qu'il se passe lorsque nous essayons d'utiliser un motif réfutable lorsque Rust prévoit d'utiliser un motif irréfutable, et vice-versa. L'encart 18-8 montre une instruction `let`, mais comme le motif nous avons indiqué `Some(x)`, un motif réfutable. Comme vous pouvez vous en douter, ce code ne va pas se compiler.

```
let Some(x) = une_option_quelconque;
```

Encart 18-8 : tentative d'utilisation d'un motif réfutable avec `let`

Si `une_option_quelconque` était une valeur `None`, elle ne correspondrait pas au motif `Some(x)`, ce qui signifie que le motif est réfutable. Cependant, l'instruction `let` ne peut accepter qu'un motif irréfutable car il n'existe pas d'instructions valides à exécuter avec une valeur `None`. A la compilation, Rust s'y opposera en expliquant que nous avons essayé d'utiliser un motif réfutable là où un motif irréfutable est nécessaire :

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error[E0005]: refutable pattern in local binding: `None` not covered
   --> src/main.rs:3:9
    |
3   |     let Some(x) = une_option_quelconque;
    |         ^^^^^^^ pattern `None` not covered
    |
    = note: `let` bindings require an "irrefutable pattern", like a `struct` or
an `enum` with only one variant
    = note: for more information, visit https://doc.rust-lang.org/book/ch18-02-
refutability.html
    = note: the matched value is of type `Option<i32>`
help: you might want to use `if let` to ignore the variant that isn't matched
    |
3   |     if let Some(x) = une_option_quelconque { /* */ }
    |

For more information about this error, try `rustc --explain E0005`.
error: could not compile `patterns` due to previous error
```

Comme nous n'avons pas couvert (et nous ne pouvons pas le faire !) chaque valeur possible avec le motif `Some(x)`, Rust génère une erreur de compilation, à juste titre.

Pour corriger le problème lorsque nous avons un motif réfutable là où un motif irréfutable est nécessaire, nous pouvons modifier le code qui utilise ce motif : au lieu d'utiliser `let`, nous pouvons utiliser `if let`. Dans ce cas, si le motif ne correspond pas, le programme va simplement sauter le code entre les accolades, ce qui lui permet de poursuivre son exécution sans rencontrer d'erreur. L'encart 18-9 montre comment corriger le code de l'encart 18-8.

```
if let Some(x) = une_option_quelconque {
    println!("{}", x);
}
```

Encart 18-9 : utilisation de `if let` et d'un bloc avec un motif réfutable plutôt qu'un `let`

Nous avons donné au code une porte de sortie. Ce code est parfaitement valide, cependant il implique que nous ne pouvons pas utiliser un motif irréfutable sans provoquer une erreur. Si nous donnons au `if let` un motif qui correspond toujours, tel que `x`, comme montré dans l'encart 18-10, le compilateur va lever un avertissement.

```
if let x = 5 {
    println!("{}", x);
};
```

Encart 18-10 : tentative d'utiliser un motif irréfutable avec `if let`

Rust explique que cela ne fait aucun sens d'utiliser `if let` avec un motif irréfutable :

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
warning: irrefutable `if let` pattern
 --> src/main.rs:2:8
  |
2 |     if let x = 5 {
  |        ^^^^^^^^^
  |
  = note: `#[warn(irrefutable_let_patterns)]` on by default
  = note: this pattern will always match, so the `if let` is useless
  = help: consider replacing the `if let` with a `let`

warning: `patterns` (bin "patterns") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.39s
     Running `target/debug/patterns`
5
```

C'est pourquoi les branches de `match` doivent utiliser des motifs réfutables, sauf pour la dernière branche, qui devrait correspondre à n'importe quelle valeur grâce à un motif irréfutable. Rust nous permet d'utiliser un motif irréfutable dans un `match` ne possédant qu'une seule branche, mais cette syntaxe n'est pas particulièrement utile et devrait être remplacée par une instruction `let` plus simple.

Maintenant que vous savez où utiliser les motifs et que vous connaissez la différence entre les motifs réfutables et irréfutables, voyons toutes les syntaxes que nous pouvons utiliser pour créer des motifs.

La syntaxe des motifs

Tout au long de ce livre, vous avez rencontré de nombreux types de motifs. Dans cette section, nous allons rassembler toutes les syntaxes valides des motifs et examiner les raisons pour lesquelles vous devriez utiliser chacune d'entre elles.

Correspondre aux littéraux

Comme vous l'avez vu chapitre 6, vous pouvez faire directement correspondre des motifs avec des littéraux. Le code suivant vous donne quelques exemples :

```
let x = 1;

match x {
    1 => println!("un"),
    2 => println!("deux"),
    3 => println!("trois"),
    _ => println!("n'importe quoi"),
}
```

Ce code affiche `un` car la valeur dans `x` est `1`. Cette syntaxe est très utile lorsque vous souhaitez que votre code fasse quelque chose s'il obtient une valeur précise.

Correspondre à des variables nommées

Les variables nommées sont des motifs irréfutables qui correspondent à n'importe quelle valeur, et nous les avons utilisées de nombreuses fois dans le livre. Cependant, il subsiste un problème lorsque vous utilisez les variables nommées dans les expressions `match`. Comme `match` débute une nouvelle portée, les variables utilisées comme faisant partie du motif de la construction `match` vont masquer celles ayant le même nom et provenant de l'extérieur de la construction `match`, comme c'est le cas avec toutes les variables. Dans l'encart 18-11, nous déclarons une variable `x` avec la valeur `Some(5)` et une variable `y` avec la valeur `10`. Nous créons alors une expression `match` sur la valeur `x`. Observez les motifs sur les branches du `match` et du `println!` à la fin, et essayez de deviner ce qui sera écrit avant d'exécuter ce code ou de lire la suite.

Fichier : src/main.rs

```
let x = Some(5);
let y = 10;

match x {
    Some(50) => println!("On a 50"),
    Some(y) => println!("Correspondance, y = {:?}", y),
    _ => println!("Cas par défaut, x = {:?}", x),
}

println!("A la fin : x = {:?}, y = {:?}", x, y);
```

Encart 18-11 : une expression `match` avec une branche qui crée une variable masquée `y`

Voyons ce qui se passe lorsque l'expression `match` est utilisée. Le motif présent dans la première branche du `match` ne correspond pas à la valeur actuelle de `x`, donc le code passe à la branche suivante.

Le motif dans la deuxième branche du `match` ajoute une nouvelle variable `y` qui va correspondre à n'importe quelle valeur logée dans une valeur `Some`. Comme nous sommes dans une nouvelle portée à l'intérieur de l'expression `match`, c'est une nouvelle variable `y`, et pas le `y` que nous avons déclaré au début avec la valeur 10. Cette nouvelle correspondance `y` va correspondre à n'importe quelle valeur à l'intérieur d'un `Some`, ce qui est la situation présente actuellement dans `x`. Ainsi, ce nouveau `y` correspondra à la valeur interne du `Some` présent dans `x`. Cette valeur est `5`, donc l'expression de cette branche s'exécute et affiche `Correspondance, y = 5`.

En supposant maintenant que `x` ait la valeur `None` plutôt que `Some(5)`, les motifs présents dans les deux premières branches ne correspondront pas, donc la valeur qui correspondra sera celle avec le tiret du bas. Comme nous n'avons pas introduit de nouvelle variable `x` dans la branche du motif, le `x` de l'expression associée désigne toujours la variable `x` en dehors et qui n'a pas été masquée. Le `match` va donc afficher `Cas par défaut, x = None`.

Lorsque l'expression `match` est terminée, sa portée se termine également, et avec elle la portée de la variable interne `y`. Le dernier `println!` affiche donc `A la fin : x = Some(5), y = 10`.

Pour créer une expression `match` qui compare les valeurs de la variable externe `x` avec `y`, plutôt que d'utiliser une variable masquée, nous aurions besoin d'utiliser à la place un contrôle de correspondance. Nous verrons les contrôles de correspondance dans une des sections suivantes.

Plusieurs motifs

Dans les expressions `match`, vous pouvez faire correspondre une même branche à plusieurs motifs en utilisant la syntaxe `|`, qui signifie *ou*. Par exemple, dans le code suivant appliquant un `match` sur la valeur de `x`, la première des branches possède une option *ou*, ce qui signifie que si la valeur de `x` correspond à l'un ou l'autre des motifs de cette branche, le code associé sera exécuté :

```
let x = 1;

match x {
    1 | 2 => println!("un ou deux"),
    3 => println!("trois"),
    _ => println!("quelque chose d'autre"),
}
```

Ce code va afficher `un ou deux`.

Faire correspondre un intervalle de valeurs avec `..=`

La syntaxe `..=` nous permet de faire correspondre un intervalle inclusif de valeurs. Dans le code suivant, lorsqu'un motif correspond à une des valeurs présentes dans l'intervalle, cette branche va s'exécuter :

```
let x = 5;

match x {
    1..=5 => println!("de un à cinq"),
    _ => println!("quelque chose d'autre"),
}
```

Si `x` vaut 1, 2, 3, 4 ou 5, la première branche va correspondre. Cette syntaxe est plus pratique à utiliser que d'avoir à utiliser l'opérateur `|` pour exprimer la même idée ; à la place de `1..=5` nous aurions dû écrire `1 | 2 | 3 | 4 | 5` si nous avions utilisé `|`. Renseigner un intervalle est bien plus court, en particulier si nous souhaitons avoir une correspondance avec les valeurs comprises entre 1 et 1000 par exemple !

Les intervalles peuvent être des nombres ou des `char` (caractères), car le compilateur vérifie que l'intervalle n'est pas vide au moment de la compilation et les seuls types pour lesquels Rust peut dire si un intervalle est vide ou non sont ceux constitués de nombres ou de `char`.

Voici un exemple d'utilisation d'intervalles de `char` :

```
    let x = 'c';

    match x {
        'a'..='j' => println!("lettre ASCII du début"),
        'k'..='z' => println!("lettre ASCII de la fin"),
        _ => println!("autre chose"),
    }
```

Rust peut nous dire que `c` est dans le premier intervalle du premier motif et afficher `lettre ASCII du début`.

Destructurer pour séparer les valeurs

Nous pouvons aussi utiliser les motifs pour destructurer les structures, les énumérations, et les tuples pour utiliser différentes parties de ces valeurs. Passons en revue chacun des cas.

Destructurer les structures

L'encart 18-12 montre une structure `Point` avec deux champs, `x` et `y`, que nous pouvons séparer en utilisant un motif avec une instruction `let`.

Fichier : src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

Encart 18-12 : déstructuration des champs d'une structure dans des variables séparées

Ce code crée les variables `a` et `b` qui correspondent aux valeurs des champs `x` et `y` de la structure `p`. Cet exemple montre que les noms des variables du motif n'ont pas à correspondre aux noms des champs de la structure. Mais il est courant de vouloir faire correspondre le nom des variables avec le nom des champs pour se rappeler plus facilement quelle variable provient de quel champ.

Comme faire correspondre les noms des variables avec ceux des champs est une pratique courante et qu'écrire `let Point { x: x, y: y } = p;` est inutilement redondant, il existe un raccourci pour les motifs qui correspondent aux champs des structures : il vous suffit de lister simplement le nom des champs de la structure pour que les variables créées à partir du motif aient les mêmes noms. L'encart 18-12 montre du code qui se comporte de la même manière que le code de l'encart 18-12, mais dans lequel les variables créées dans le motif du `let` sont `x` et `y` au lieu de `a` et `b`.

Fichier : src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
    assert_eq!(7, y);
}
```

Encart 18-13 : déstructuration des champs d'une structure en utilisant le raccourci pour les champs des structures

Ce code crée les variables `x` et `y` qui correspondent aux champs `x` et `y` de la variable `p`. Il en résulte que les variables `x` et `y` contiennent les valeurs correspondantes de la structure `p`.

Nous pouvons aussi destructurer en utilisant des valeurs littérales faisant partie du motif de la structure plutôt que d'avoir à créer les variables pour tous les champs. Ceci nous permet de tester que certains champs possèdent des valeurs particulières tout en créant des variables pour destructurer les autres champs.

L'encart 18-14 montre une expression `match` qui sépare les valeurs `Point` en trois catégories : les points qui sont sur l'axe `x` (ce qui est vrai lorsque `y = 0`), ceux sur l'axe `y` (`x = 0`) et ceux qui ne sont sur aucun de ces deux axes.

Fichier : src/main.rs

```
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("Sur l'axe x à la position {}", x),
        Point { x: 0, y } => println!("Sur l'axe y à la position {}", y),
        Point { x, y } => println!("Sur aucun des axes : ({}, {})", x, y),
    }
}
```

Encart 18-14 : déstructurer et faire correspondre des valeurs littérales grâce à un seul motif

La première branche va correspondre avec tous les points qui se trouvent sur l'axe `x` en précisant que le champ `y` correspond au littéral `0`. Le motif va systématiquement créer une variable `x` que nous pourrons utiliser dans le code de cette branche.

De la même manière, la deuxième branche correspondra avec tous les points sur l'axe `y` en précisant que le champ `x` correspondra uniquement si sa valeur est `0` et créera une variable `y` pour la valeur du champ `y`. La troisième branche n'a pas besoin d'un littéral en particulier, donc elle correspondra à n'importe quel autre `Point` et créera les variables pour les champs `x` et `y`.

Dans cet exemple, la valeur `p` correspond avec la deuxième branche car son `x` vaut `0`, donc ce code va afficher `Sur l'axe y à la position 7`.

Destructurer une énumération

Nous avons déjà destructuré des énumérations précédemment dans ce livre, par exemple lorsque nous avions destructuré `Option<i32>` dans l'encart 6-5 du chapitre 6. Un détail que nous n'avions pas précisé explicitement était que le motif pour destructurer une énumération doit correspondre à la façon dont sont définies les données dans l'énumération. Par exemple, dans l'encart 18-15 nous utilisons l'énumération `Message` de l'encart 6-2 et nous ajoutons un `match` avec des motifs qui devraient destructurer chaque valeur interne.

Fichier : src/main.rs

```
enum Message {
    Quitter,
    Deplacer { x: i32, y: i32 },
    Ecrire(String),
    ChangerCouleur(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangerCouleur(0, 160, 255);

    match msg {
        Message::Quitter => {
            println!("La variante Quitter n'a pas de données à déstructurer.")
        }
        Message::Deplacer { x, y } => {
            println!(
                "Déplacement de {} sur l'axe x et de {} sur l'axe y",
                x, y
            );
        }
        Message::Ecrire(text) => println!("Message textuel : {}", text),
        Message::ChangerCouleur(r, g, b) => println!(
            "Changement des taux de rouge à {}, de vert à {} et de bleu à {}",
            r, g, b
        ),
    }
}
```

Encart 18-15 : déstructuration des variantes d'une énumération qui stocke différents types de valeurs

Ce code va afficher `Changement des taux de rouge à 0, de vert à 160 et de bleu à 255` . Essayez de changer la valeur de `message` pour voir le code qu'exécute les autres branches.

Pour les variantes d'énumération sans aucune donnée, telle que `Message::Quitter` , nous ne pouvons pas destructurer de valeurs. Nous pouvons uniquement correspondre à la valeur littérale `Message::Quitter` et il n'y a pas de variable dans ce motif.

Pour les variantes d'énumération qui ressemblent aux structures, comme `Message::Deplacer` , nous pouvons utiliser un motif similaire aux motifs que nous utilisons pour correspondre aux structures. Après le nom de la variante, nous utilisons des accolades puis nous listons les champs avec des variables afin de diviser les éléments à utiliser dans le code de cette branche. Ici nous utilisons la forme raccourcie comme nous l'avons fait à l'encart 18-13.

Pour les variantes d'énumérations qui ressemblent à des tuples, telles que `Message::Ecrire` qui stocke un tuple avec un seul élément, ou `Message::ChangerCouleur`

qui stocke un tuple avec trois éléments, le motif est semblable à celui que nous renseignons pour correspondre aux tuples. Le nombre de variables dans le motif doit correspondre au nombre d'éléments dans la variante qui correspond.

Destructurer des structures et des énumérations imbriquées

Jusqu'à présent, tous nos exemples avaient des correspondances avec des structures ou des énumérations qui n'avaient qu'un seul niveau de profondeur. Les correspondances fonctionnent aussi sur les éléments imbriqués !

Par exemple, nous pouvons remanier le code de l'encart 18-15 pour pouvoir utiliser des couleurs RVB et TSV dans le message `ChangerCouleur`, comme dans l'encart 18-16.

```
enum Couleur {
    Rvb(i32, i32, i32),
    Tsv(i32, i32, i32),
}

enum Message {
    Quitter,
    Deplacer { x: i32, y: i32 },
    Ecrire(String),
    ChangerCouleur(Couleur),
}

fn main() {
    let msg = Message::ChangerCouleur(Couleur::Tsv(0, 160, 255));

    match msg {
        Message::ChangerCouleur(Couleur::Rvb(r, v, b)) => println!(
            "Changement des taux de rouge à {}, de vert à {} et de bleu à {}",
            r, v, b
        ),
        Message::ChangerCouleur(Couleur::Tsv(t, s, v)) => println!(
            "Changement des taux de teinte à {}, de saturation à {} et de valeur
à {}",
            t, s, v
        ),
        _ => (),
    }
}
```

Encart 18-16 : correspondance avec des énumérations imbriquées

Le motif de la première branche dans l'expression `match` correspond à la variante d'énumération `Message::ChangerCouleur` qui contient une variante `Couleur::Rvb` ; ensuite le motif fait correspondre des variables aux trois valeurs `i32` que cette dernière contient. Le

motif de la seconde branche correspond aussi à une variante de l'énumération de `Message::ChangerCouleur`, mais la valeur interne correspond plutôt à la variante `Couleur::Tsv`. Nous pouvons renseigner ces conditions complexes dans une seule expression `match`, bien que deux énumérations différentes soient impliquées.

Destructurer des structures et des tuples

Nous pouvons mélanger les correspondances et les motifs pour déstructurer des éléments imbriqués de manière bien plus complexe. L'exemple suivant montre une déstructuration complexe dans laquelle nous imbriquons des structures et des tuples à l'intérieur d'un tuple et nous y destructurons toutes les valeurs primitives :

```
let ((pieds, pouces), Point { x, y }) = ((3, 10), Point { x: 3, y: -10 });
```

Ce code nous permet de décomposer les parties qui composent des types complexes pour pouvoir utiliser séparément les valeurs qui nous intéressent.

La déstructuration avec les motifs est un moyen efficace d'utiliser des parties de valeurs, comme par exemple la valeur de chaque champ d'une structure, indépendamment les unes des autres.

Ignorer des valeurs dans un motif

Vous avez pu constater qu'il est parfois utile d'ignorer des valeurs dans un motif, comme celle dans la dernière branche d'un `match`, pour obtenir un joker qui ne fait rien mis à part qu'il représente toutes les autres valeurs possibles. Il existe plusieurs façons d'ignorer totalement ou en partie des valeurs dans un motif : en utilisant le motif `_` (que vous avez déjà vu), le motif `_` à l'intérieur d'un autre motif, un nom qui commence avec un tiret bas, ou enfin `..` pour ignorer les parties restantes d'une valeur. Voyons comment et pourquoi utiliser ces différents motifs.

Ignorer complètement une valeur avec _

Nous avons utilisé le tiret bas (_) comme un motif joker qui correspondra avec n'importe quelle valeur mais ne l'assignera pas. Bien que le motif du tiret bas `_` soit particulièrement utile dans la dernière branche d'une expression `match`, nous pouvons aussi l'utiliser dans n'importe quel motif, y compris dans les paramètres de fonctions, comme montré dans l'encart 18-17.

Fichier : src/main.rs

```
fn fonction(_: i32, y: i32) {
    println!("Ce code utilise uniquement le paramètre y : {}", y);
}

fn main() {
    fonction(3, 4);
}
```

Encart 18-17 : utilisation d'un `_` dans la signature d'une fonction

Ce code va complètement ignorer la valeur envoyée en premier argument, `3`, et va afficher `Ce code utilise uniquement le paramètre y : 4`.

Dans la plupart des cas lorsque vous n'avez pas besoin d'un paramètre d'une fonction, vous pouvez changer la signature pour qu'elle n'inclut pas le paramètre non utilisé. Ignorer un paramètre de fonction peut être particulièrement utile dans certains cas, comme par exemple, lors de l'implémentation d'un trait lorsque vous avez besoin d'un certain type de signature mais que le corps de la fonction dans votre implémentation n'a pas besoin d'un des paramètres. Le compilateur ne vous avertira plus que ces paramètres de fonction ne sont pas utilisés, ce qui serait le cas si vous utilisiez un nom à la place.

Ignorer des parties d'une valeur en utilisant un _ imbriqué

Nous pouvons aussi utiliser `_` au sein d'un autre motif pour ignorer uniquement une partie d'une valeur, par exemple, si nous ne souhaitons tester qu'une seule partie d'une valeur mais que nous n'utilisons pas les autres parties dans le code que nous souhaitons exécuter. L'encart 18-18 montre du code qui s'occupe de gérer la valeur d'un réglage. Les règles métier sont que l'utilisateur ne doit pas pouvoir modifier un réglage existant mais peut annuler le réglage ou lui donner une valeur s'il n'en a pas encore.

```
    let mut valeur_du_reglage = Some(5);
    let nouvelle_valeur_du_reglage = Some(10);

    match (valeur_du_reglage, nouvelle_valeur_du_reglage) {
        (Some(_), Some(_)) => {
            println!("Vous ne pouvez pas écraser une valeur déjà existante");
        }
        _ => {
            valeur_du_reglage = nouvelle_valeur_du_reglage;
        }
    }

    println!("Le réglage vaut {:?}", valeur_du_reglage);
```

Encart 18-18 : utilisation d'un tiret bas dans des motifs qui correspondent avec des variantes

`Some` lorsque nous n'avons pas besoin d'utiliser la valeur à l'intérieur du `Some`

Ce code va afficher `Vous ne pouvez pas écraser une valeur déjà existante` et ensuite `Le réglage vaut Some(5)`. Dans la première branche, nous n'avons pas besoin de récupérer ou d'utiliser les valeurs à l'intérieur de chacune des variantes `Some`, mais nous avons besoin de tester les situations où `valeur_du_reglage` et `nouvelle_valeur_du_reglage` sont toutes deux des variantes `Some`. Dans ce cas, nous écrivons que nous n'allons pas changer `valeur_du_reglage` et elle ne changera pas.

Dans tous les autres cas (lorsque soit `valeur_du_reglage`, soit `nouvelle_valeur_du_reglage` vaut `None`) qui correspondront avec le motif `_` de la seconde branche, nous voulons permettre à la valeur de `nouvelle_valeur_du_reglage` de remplacer celle de `valeur_du_reglage`.

Nous pouvons aussi utiliser les tirets bas à plusieurs endroits dans un même motif pour ignorer des valeurs précises. L'encart 18-19 montre un exemple qui ignore la deuxième et la quatrième valeur dans un tuple de cinq éléments.

```
    let nombres = (2, 4, 8, 16, 32);

    match nombres {
        (premier, _, troisieme, _, cinquieme) => {
            println!("Voici quelques nombres : {}, {}, {}", premier, troisieme,
cinquieme)
        }
    }
```

Encart 18-19 : on ignore plusieurs éléments d'un tuple

Ce code va afficher `Voici quelques nombres : 2, 8, 32` tandis que les valeurs 4 et 16 sont ignorées.

Ignorer une variable non utilisée en préfixant son nom avec un _

Si vous créez une variable mais que vous ne l'utilisez nulle part, Rust va lancer un avertissement car cela pourrait être un bogue. Mais parfois il est utile de créer une variable que vous n'utilisez pas encore, ce qui peut arriver lorsque vous créez un prototype ou un projet. Dans ce genre de situation, vous pouvez demander à Rust de ne pas vous avertir que la variable n'est pas utilisée en préfixant son nom avec un tiret bas. Dans l'encart 18-20, nous créons deux variables non utilisées, mais lorsque nous compilerons ce code, nous n'aurons d'avertissement que pour une seule d'entre elles.

Fichier : src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

Encart 18-20 : préfixer le nom d'une variable avec un tiret bas pour éviter d'avoir des avertissements signalant une variable non utilisée

Ici nous avons un avertissement qui nous prévient que nous n'utilisons pas la variable `y`, mais nous n'avons pas d'avertissement concernant la variable dont le nom est préfixé par un tiret bas.

Notez qu'il existe une différence subtile entre utiliser uniquement `_` et préfixer un nom avec un tiret bas. La syntaxe `_x` continue à associer la valeur à une variable, alors que `_` ne le fait pas du tout. Pour montrer un cas où cette différence est importante, l'encart 18-21 va nous donner une erreur.

```
    let s = Some(String::from("Salutations !"));

    if let Some(_s) = s {
        println!("j'ai trouvé une chaine de caractères");
    }

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

Encart 18-21 : une variable non utilisée préfixée par un tiret bas continue à assigner la valeur, ce qui pourrait entraîner une prise de possession de la valeur

Nous allons obtenir une erreur car la valeur `s` est toujours déplacée dans `_s`, ce qui nous empêche d'utiliser `s` ensuite. A l'inverse, l'utilisation du tiret bas tout seul n'assigne jamais la valeur à quelque chose. Par conséquent, l'encart 18-22 va se compiler sans aucune erreur car `s` n'est pas déplacé dans `_`.

```
    let s = Some(String::from("Salutations !"));

    if let Some(_) = s {
        println!("j'ai trouvé une chaine de caractères");
    }

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

Encart 18-22 : l'utilisation d'un tiret bas n'assigne pas la valeur

Ce code fonctionne correctement car nous n'assignons jamais `s` à quelque chose ; elle n'est jamais déplacée.

Ignorer les éléments restants d'une valeur avec `..`

Avec les valeurs qui ont de nombreux éléments, nous pouvons utiliser la syntaxe `..` pour n'utiliser que quelques éléments et ignorer les autres, ce qui évite d'avoir à faire une liste de tirets bas pour chacune des valeurs ignorées. Le motif `..` ignore tous les éléments d'une valeur qui ne correspondent pas explicitement au reste du motif. Dans l'encart 18-23, nous avons une structure `Point` qui stocke des coordonnées dans un espace tridimensionnel. Dans l'expression `match`, nous souhaitons utiliser uniquement la coordonnée `x` et ignorer les valeurs des champs `y` et `z`.

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}

let origine = Point { x: 0, y: 0, z: 0 };

match origine {
    Point { x, .. } => println!("x vaut {}", x),
}
```

Encart 18-23 : on ignore tous les champs d'un `Point` à l'exception de `x` en utilisant `..`

Nous ajoutons la valeur `x` puis nous insérons simplement le motif `..`. C'est plus rapide que d'avoir à ajouter `y: _` et `z: _`, en particulier lorsque nous travaillons avec des structures qui ont beaucoup de champs alors qu'un seul champ ou deux nous intéressent.

La syntaxe `..` va s'étendre à toutes les valeurs qu'elle devra couvrir. L'encart 18-24 montre comment utiliser `..` avec un tuple.

Fichier : src/main.rs

```
fn main() {
    let nombres = (2, 4, 8, 16, 32);

    match nombres {
        (premier, .., dernier) => {
            println!("Voici quelques nombres : {}, {}", premier, dernier);
        }
    }
}
```

Encart 18-24 : on correspond uniquement avec la première et la dernière valeur d'un tuple en ignorant toutes les autres valeurs

Dans ce code, la première et la dernière valeur correspondent à `premier` et `dernier`. Le `..` va correspondre et ignorer tout ce qui se trouve entre les deux.

Cependant, l'utilisation de `..` peut être ambigu. S'il n'est pas possible de déterminer clairement quelles valeurs doivent correspondre et quelles valeurs doivent être ignorées, Rust va nous retourner une erreur. L'encart 18-25 nous montre un exemple d'utilisation ambigu de `..` qui, par conséquent, ne se compilera pas.

Fichier : src/main.rs

```
fn main() {
    let nombres = (2, 4, 8, 16, 32);

    match nombres {
        (.., second, ..) => {
            println!("Voici quelques nombres : {}", second)
        },
    }
}
```

Encart 18-25 : une tentative d'utilisation de `..` de manière ambigüe

Lorsque nous compilons cet exemple, nous obtenons l'erreur suivante :

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error: `..` can only be used once per tuple pattern
 --> src/main.rs:5:22
  |
5 |         (.., second, ..) => {
  |          --          ^^ can only be used once per tuple pattern
  |          |
  |          previously used here

error: could not compile `patterns` due to previous error
```

Il est impossible pour Rust de déterminer combien de valeurs doivent être ignorées dans le tuple avant de faire correspondre une valeur avec `second` et ensuite combien d'autres doivent être ignorées après. Ce code pourrait signifier que nous voulons ignorer `2`, faire correspondre `second` avec `4`, puis ignorer ensuite `8`, `16` et `32` ; ou que nous souhaitons ignorer `2` et `4`, faire correspondre `second` à `8`, puis ignorer ensuite `16` et `32` ; et ainsi de suite. Le nom de la variable `second` ne signifie pas grand-chose pour Rust, donc nous obtenons une erreur de compilation à cause de l'utilisation de `..` à deux endroits qui rendent la situation ambigüe.

Plus de conditions avec les contrôles de correspondance

Un *contrôle de correspondance* est une condition `if` supplémentaire renseignée après le motif d'une branche d'un `match` qui doit elle aussi correspondre en même temps que le filtrage par motif, pour que cette branche soit choisie. Les contrôles de correspondance sont utiles pour exprimer des idées plus complexes que celles permises uniquement par les motifs.

La condition peut utiliser des variables créées dans le motif. L'encart 18-26 montre un `match` dans lequel la première branche a le motif `Some(x)` et procède aussi au contrôle de correspondance `if x < % 2 == 0` (qui sera vrai si le nombre est pair).

```
let nombre = Some(4);

match nombre {
    Some(x) if x % 2 == 0 => println!("Le nombre {} est pair", x),
    Some(x) => println!("Le nombre {} est impair", x),
    None => (),
}
```

Encart 18-26 : ajout d'un contrôle de correspondance à un motif

Cet exemple va afficher `Le nombre 4 est pair`. Lorsque `nombre` est comparé au motif de la première branche, il va correspondre, car `Some(4)` correspond à `Some(x)`. Ensuite, le contrôle de correspondance vérifie si le reste de la division de `x` par 2 vaut 0, et comme c'est le cas, la première branche est choisie.

Si `nombre` avait été plutôt `Some(5)`, le contrôle de correspondance de la première branche aurait été faux car le reste de la division de 5 par 2 est 1, ce qui n'est pas égal à 0. Rust serait donc allé à la deuxième branche, qui devrait être choisie car cette deuxième branche correspond à n'importe quelle variante `Some` et n'a pas de contrôle de correspondance.

Comme il n'existe pas d'autre moyen d'exprimer la condition `if x % 2 == 0` dans un motif, le contrôle de correspondance nous donne la possibilité d'exprimer une telle logique. L'inconvénient de cette expressivité renforcée est que le compilateur n'essaie pas de vérifier l'exhaustivité lorsqu'on utilise les contrôles de correspondance.

Dans l'encart 18-11, nous avions mentionné le fait que nous pouvions utiliser des contrôles de correspondance pour résoudre notre problème de masquage dans le motif. Souvenez-vous qu'une nouvelle variable avait été créée à l'intérieur du motif dans l'expression `match` au lieu d'utiliser la variable située à l'extérieur du `match`. Cette nouvelle variable implique que nous ne pouvons pas comparer avec la variable qui se situe à l'extérieur. L'encart 18-27 nous montre comment nous pouvons utiliser un contrôle de correspondance pour répondre

à ce besoin.

Fichier : src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Nous obtenons 50"),
        Some(n) if n == y => println!("Nous avons une correspondance, n = {}",
n),
        _ => println!("Cas par défaut, x = {:?}", x),
    }

    println!("Au final : x = {:?}, y = {}", x, y);
}
```

Encart 18-27 : utilisation d'un contrôle de correspondance pour vérifier l'égalité avec une variable externe au bloc

Ce code va maintenant afficher `Cas par défaut, x = Some(5)`. Le motif de la deuxième branche du `match` ne crée pas de nouvelle variable `y` qui masquerait le `y` externe, ce qui signifie que nous pouvons utiliser le `y` externe dans le contrôle de correspondance. Au lieu de renseigner le motif comme étant `Some(y)`, ce qui aurait masqué le `y` externe, nous renseignons `Some(n)`. Cela va créer une nouvelle variable `n` qui ne masque rien car il n'y a pas de variable `n` à l'extérieur du `match`.

Le contrôle de correspondance `if n == y` n'est pas un motif et donc il n'introduit pas de nouvelle variable. Ce `y` *est* la variable externe `y` au lieu d'être une nouvelle variable masquée `y`, et nous pouvons comparer une valeur qui a la même valeur que le `y` externe en comparant `n` à `y`.

Vous pouvez aussi utiliser l'opérateur *ou* `|` dans un contrôle de correspondance pour y renseigner plusieurs motifs ; la condition du contrôle de correspondance s'effectuera alors sur tous les motifs. L'encart 18-28 montre la priorité de combinaison d'un contrôle de correspondance sur un motif qui utilise `|`. La partie importante de cet exemple est que le contrôle de correspondance `if y` s'applique sur `4`, `5` *et* `6`, même si `if y` semble s'appliquer uniquement à `6`.

```
let x = 4;
let y = false;

match x {
    4 | 5 | 6 if y => println!("yes"),
    _ => println!("no"),
}
```

Encart 18-28 : combinaison de plusieurs motifs avec un contrôle de correspondance

La condition de correspondance signifie que la branche correspond uniquement si la valeur de `x` vaut `4`, `5` ou `6` *et* que `y` vaut `true`. Lorsque ce code s'exécute, le motif de la première branche correspond car `x` vaut 4, mais le contrôle de correspondance `if y` est faux, donc ce programme affiche `no`. La raison est que la condition `if` s'applique à tout le motif `4 | 5 | 6` et pas seulement à la dernière valeur `6`. Autrement dit, la priorité d'un contrôle de correspondance avec un motif se comporte comme ceci :

```
(4 | 5 | 6) if y => ...
```

et pas comme ceci :

```
4 | 5 | (6 if y) => ...
```

Après avoir exécuté le code, le fonctionnement des priorités devient évident : si le contrôle de correspondance était seulement appliqué à la dernière valeur renseignée avec l'opérateur `|`, la branche correspondrait et le programme aurait affiché `yes`.

Capturer des valeurs avec @

L'opérateur `@` nous permet de créer une variable qui stocke une valeur en même temps que nous testons cette valeur pour vérifier si elle correspond à un motif. L'encart 18-29 montre un exemple dans lequel nous souhaitons tester qu'un champ `id` d'un `Message::Hello` est dans un intervalle `3..=7`. Mais nous voulons aussi associer la valeur à la variable `id_variable` pour que nous puissions l'utiliser dans le code associé à la branche. Nous aurions pu nommer cette variable avec le même nom que le champ `id`, mais pour cet exemple nous allons utiliser un nom différent.

```
    enum Message {
        Hello { id: i32 },
    }

    let msg = Message::Hello { id: 5 };

    match msg {
        Message::Hello {
            id: id_variable @ 3..=7,
        } => println!("Nous avons trouvé un id dans l'intervalle : {}",
id_variable),
        Message::Hello { id: 10..=12 } => {
            println!("Nous avons trouvé un id dans un autre intervalle")
        }
        Message::Hello { id } => println!("Nous avons trouvé un autre id : {}",
id),
    }
```

Encart 18-29 : utilisation de `@` pour lier une valeur d'un motif à une variable pendant qu'on la teste

Cet exemple va afficher `Nous avons trouvé un id dans l'intervalle : 5`. En renseignant `id_variable @` avant l'intervalle `3..=7`, nous capturons la valeur qui correspond à l'intervalle pendant que nous vérifions que la valeur correspond au motif de l'intervalle.

Dans la deuxième branche, où nous avons uniquement un intervalle renseigné dans le motif, le code associé à la branche n'a pas besoin d'une variable qui contienne la valeur actuelle du champ `id`. La valeur du champ `id` aurait pu être 10, 11 ou 12, mais le code associé à ce motif ne la connaîtra pas. Le code du motif n'est pas capable d'utiliser la valeur du champ `id`, car nous n'avons pas enregistré `id` dans une variable.

Dans la dernière branche, nous avons renseigné une variable sans intervalle, nous avons donc dans la variable `id` la valeur qui peut être utilisée dans le code de la branche. La raison à cela est que nous avons utilisé la syntaxe raccourcie pour les champs des structures. Mais, dans cette branche, nous n'avons pas appliqué de tests à la valeur sur le champ `id`, comme nous l'avions fait avec les deux premières branches : n'importe quelle valeur correspondra à ce motif.

L'utilisation de `@` nous permet de tester une valeur et de l'enregistrer dans une variable au sein d'un seul et même motif.

Résumé

Les motifs de Rust sont très utiles lorsque nous devons distinguer différents types de données. Lorsque nous les avions utilisés dans les expressions `match`, Rust s'est assuré que vos motifs couvraient l'intégralité de toutes valeurs possibles, et, dans le cas contraire, votre programme ne se compilait pas. Les motifs dans les instructions `let` et les paramètres de fonction rendent ces constructions encore plus utiles, permettant de déstructurer les valeurs en parties plus petites tout en les assignant à des variables. Nous pouvons créer des motifs très simples ou alors plus complexes pour répondre à nos besoins.

Dans le chapitre suivant, qui sera l'avant-dernier du livre, nous allons découvrir quelques aspects avancés de l'éventail de fonctionnalités de Rust.

Les fonctionnalités avancées

Jusqu'ici, vous avez appris les fonctionnalités les plus utilisées du langage de programmation Rust. Avant de commencer le nouveau projet du chapitre 20, nous allons regarder quelques aspects du langage que vous pourriez rencontrer de temps à autre. Vous pouvez utiliser ce chapitre comme référence à consulter lorsque vous rencontrerez des éléments de Rust qui vous sont inconnus. Les fonctionnalités que vous allez découvrir dans ce chapitre sont utiles dans des situations très spécifiques. Même si vous n'allez pas les rencontrer très souvent, nous voulons nous assurer que vous comprenez bien toutes les fonctionnalités que Rust peut offrir.

Dans ce chapitre, nous allons voir :

- Le *unsafe* de Rust : comment désactiver certaines garanties de Rust et prendre la responsabilité de veiller vous-même manuellement à les assurer
- Les traits avancés : les types associés, les types de paramètres par défaut, la syntaxe entièrement détaillée, les supertraits et le motif newtype en lien avec les traits
- Les types avancés : en savoir plus sur le motif newtype, les alias de type, le type never et les types à taille dynamique
- Les fonctions et fermetures avancées : les pointeurs de fonctions et la façon de retourner des fermetures
- Les macros : une manière de définir du code qui produit encore plus de code au moment de la compilation

Voilà pléthore de fonctionnalités de Rust dans lesquelles chacun y trouvera son compte ! Commençons tout de suite !

Le Rust non sécurisé (`unsafe`)

Tout le code Rust que nous avons abordé jusqu'à présent a bénéficié des garanties de sécurité de la mémoire, vérifiées à la compilation. Cependant Rust possède un second langage caché en son sein qui n'applique pas ces vérifications de sécurité de la mémoire : il s'appelle le *Rust non sécurisé* et fonctionne comme le Rust habituel, mais fournit quelques super-pouvoirs supplémentaires.

Le Rust non sécurisé existe car, par nature, l'analyse statique est conservative. Lorsque le compilateur essaye de déterminer si le code respecte ou non les garanties, il vaut mieux rejeter quelques programmes valides plutôt que d'accepter quelques programmes invalides. Bien que le code *puisse* être correct, si le compilateur Rust n'a pas assez d'information pour être sûr, il va refuser ce code. Dans ce cas, vous pouvez utiliser du code non sécurisé pour dire au compilateur “fais-moi confiance, je sais ce que je fait”. Le prix à payer pour cela est que vous l'utilisez à vos risques et périls : si vous écrivez du code non sécurisé de manière incorrecte, des problèmes liés à la sécurité de la mémoire peuvent se produire, tel qu'un déréférencement d'un pointeur vide.

Une autre raison pour laquelle Rust embarque son alter-ego non sécurisé est que le matériel des ordinateurs sur lequel il repose n'est pas sécurisé par essence. Si Rust ne vous laissait pas procéder à des opérations non sécurisées, vous ne pourriez pas faire certaines choses. Rust doit pouvoir vous permettre de développer du code bas-niveau, comme pouvoir interagir directement avec le système d'exploitation ou même écrire votre propre système d'exploitation. Pouvoir travailler avec des systèmes bas-niveau est un des objectifs du langage. Voyons ce que nous pouvons faire avec le Rust non sécurisé et comment le faire.

Les super-pouvoirs du code non sécurisé

Pour pouvoir utiliser le Rust non sécurisé, il faut utiliser le mot-clé `unsafe` et ensuite créer un nouveau bloc qui contient le code non sécurisé. Vous pouvez faire cinq actions en Rust non sécurisé, qui s'appellent *les super-pouvoirs du non sécurisé*, actions que vous ne pourriez pas faire en Rust sécurisé. Ces super-pouvoirs permettent de :

- Déréférencer un pointeur brut
- Faire appel à une fonction ou une méthode non sécurisée
- Lire ou modifier une variable statique mutable
- Implémenter un trait non sécurisé
- Accéder aux champs des `union`

Il est important de comprendre que `unsafe` ne désactive pas le vérificateur d'emprunt et ne

désactive pas les autres vérifications de sécurité de Rust : si vous utilisez une référence dans du code non sécurisé, elle sera toujours vérifiée. Le mot-clé `unsafe` vous donne seulement accès à ces cinq fonctionnalités qui ne sont alors pas vérifiées par le compilateur en vue de veiller à la sécurité de la mémoire. Vous conservez donc un certain niveau de sécurité à l'intérieur d'un bloc `unsafe` .

De plus, `unsafe` ne signifie pas que le code à l'intérieur du bloc est obligatoirement dangereux ou qu'il va forcément présenter des problèmes de sécurité mémoire : l'idée étant qu'en tant que développeur, vous vous assuriez que le code à l'intérieur d'un bloc `unsafe` va accéder correctement à la mémoire.

Personne n'est parfait, les erreurs arrivent, et en imposant que ces cinq opérations non sécurisés se trouvent dans des blocs marqués d'un `unsafe` , Rust vous permet de savoir que ces éventuelles erreurs liées à la sécurité de la mémoire se trouveront dans un bloc `unsafe` . Vous devez donc essayer de minimiser la taille des blocs `unsafe` ; vous ne le regretterez pas lorsque vous rechercherez des bogues de mémoire.

Pour isoler autant que possible le code non sécurisé, il vaut mieux intégrer du code non sécurisé dans une abstraction et fournir ainsi une API sécurisée, comme nous le verrons plus tard dans ce chapitre lorsque nous examinerons les fonctions et méthodes non sécurisées. Certaines parties de la bibliothèque standard sont implémentées comme étant des abstractions sécurisées et basées sur du code non sécurisé qui a été audité. Encapsuler du code non sécurisé dans une abstraction sécurisée évite que l'utilisation de `unsafe` ne se propage dans des endroits où vous ou vos utilisateurs souhaiteraient éviter d'utiliser les fonctionnalités du code `unsafe` , car au final utiliser une abstraction sécurisée doit rester sûr.

Analysons ces cinq super-pouvoirs à tour de rôle. Nous allons aussi découvrir quelques abstractions qui fournissent une interface sécurisée pour faire fonctionner du code non sécurisé.

Déréférencer un pointeur brut

Au chapitre 4, dans la section “Les références pendouillantes”, nous avions mentionné que le compilateur s'assure que les références sont toujours valides. Le Rust non sécurisé offre deux nouveaux types qui s'appellent les *pointeurs brut* et qui ressemblent aux références. Comme les références, les pointeurs bruts peuvent être immuables ou mutables et s'écrivent respectivement `*const T` et `*mut T` . L'astérisque n'est pas l'opérateur de déréférencement ; il fait partie du nom du type. Dans un contexte de pointeur brut, *immuable* signifie que le pointeur ne peut pas être affecté directement après avoir été déréférencé.

Par rapport aux références et aux pointeurs intelligents, les pointeurs bruts peuvent :

- ignorer les règles d'emprunt en ayant plusieurs pointeurs tant immuables que mutables ou en ayant plusieurs pointeurs mutables qui pointent vers le même endroit.
- ne pas être obligés de pointer sur un emplacement mémoire valide
- être autorisés à avoir la valeur nulle
- ne pas implémenter de fonctionnalité de nettoyage automatique

En renonçant à ce que Rust fasse respecter ces garanties, vous pouvez sacrifier la sécurité garantie pour obtenir de meilleures performances ou avoir la possibilité de vous interfacer avec un autre langage ou matériel pour lesquels les garanties de Rust ne s'appliquent pas.

L'encart 19-1 montre comment créer un pointeur brut immuable et mutable à partir de références.

```
let mut nombre = 5;

let r1 = &nombre as *const i32;
let r2 = &mut nombre as *mut i32;
```

Encart 19-1 : création de pointeurs bruts à partir de références

Remarquez que nous n'incorporons pas le mot-clé `unsafe` dans ce code. Nous pouvons créer des pointeurs bruts dans du code sécurisé ; nous ne pouvons simplement pas déréférencer les pointeurs bruts à l'extérieur d'un bloc non sécurisé, comme vous allez le constater d'ici peu.

Nous avons créé des pointeurs bruts en utilisant `as` pour transformer les références immuables et mutables en leur type de pointeur brut correspondant. Comme nous les avons créés directement à partir de références qui sont garanties d'être valides, nous savons que ces pointeurs bruts seront valides, mais nous ne pouvons pas faire cette supposition sur tous les pointeurs bruts.

Ensuite, nous allons créer un pointeur brut dont la validité n'est pas certaine. L'encart 19-2 montre comment créer un pointeur brut vers un emplacement arbitraire de la mémoire. Essayer d'utiliser de la mémoire arbitraire va engendrer un comportement incertain : il peut y avoir des données à cette adresse comme il peut ne pas y en avoir, le compilateur pourrait optimiser le code de tel sorte qu'aucun accès mémoire n'aura lieu ou bien le programme pourrait déclencher une erreur de segmentation. Habituellement, il n'y a pas de bonne raison d'écrire du code comme celui-ci, mais c'est possible.

```
let addresse = 0x012345usize;
let r = addresse as *const i32;
```

Encart 19-2 : création d'un pointeur brut vers une adresse mémoire arbitraire

Souvenez-vous que nous pouvons créer des pointeurs bruts dans du code sécurisé, mais que nous ne pouvons pas y *déréférencer* les pointeurs bruts et lire les données sur lesquelles ils pointent. Dans l'encart 19-3, nous utilisons l'opérateur de déréférencement `*` sur un pointeur brut qui nécessite un bloc `unsafe` .

```
let mut nombre = 5;

let r1 = &nombre as *const i32;
let r2 = &mut nombre as *mut i32;

unsafe {
    println!("r1 vaut : {}", *r1);
    println!("r2 vaut : {}", *r2);
}
```

Encart 19-3 : déréférencement d'un pointeur brut à l'intérieur d'un bloc `unsafe`

La création de pointeur ne pose pas de problèmes ; c'est seulement lorsque nous essayons d'accéder aux valeurs sur lesquelles ils pointent qu'on risque d'obtenir une valeur invalide.

Remarquez aussi que dans les encarts 19-1 et 19-3, nous avons créé les pointeurs bruts `*const i32` et `*mut i32` qui pointent tous les deux au même endroit de la mémoire, où `nombre` est stocké. Si nous avions plutôt tenté de créer une référence immuable et une mutable vers `nombre` , le code n'aurait pas compilé à cause des règles de possession de Rust qui ne permettent pas d'avoir une référence mutable en même temps qu'une ou plusieurs références immuables. Avec les pointeurs bruts, nous pouvons créer un pointeur mutable et un pointeur immuable vers le même endroit et changer la donnée via le pointeur mutable, en risquant un accès concurrent. Soyez vigilant !

Avec tous ces dangers, pourquoi vous risquer à utiliser les pointeurs bruts ? Une des utilisations principale consiste à s'interfacer avec du code C, comme vous allez le découvrir dans la section suivante. Une autre utilisation est de nous permettre de créer une abstraction sécurisée que le vérificateur d'emprunt ne comprend pas. Nous allons découvrir les fonctions non sécurisées puis voir un exemple d'une abstraction sécurisée qui utilise du code non sécurisé.

Faire appel à une fonction ou une méthode non sécurisée

Le deuxième type d'opération qui nécessite un bloc `unsafe` est l'appel à des fonctions non sécurisées. Les fonctions et méthodes non sécurisées ressemblent exactement aux méthodes et fonctions habituelles, mais ont un `unsafe` en plus devant le reste de leur

définition. Le mot-clé `unsafe` dans ce cas signifie que la fonction a des exigences que nous devons respecter pour pouvoir y faire appel, car Rust ne pourra pas garantir de son côté que nous les ayons remplies. En faisant appel à une fonction non sécurisée dans un bloc `unsafe`, nous reconnaissons que nous avons lu la documentation de cette fonction et pris la responsabilité de respecter les conditions d'utilisation de la fonction.

Voici une fonction non sécurisée `dangereux`, qui ne fait rien dans son corps :

```
unsafe fn dangereux() {}

unsafe {
    dangereux();
}
```

Nous devons faire appel à la fonction `dangereux` dans un bloc `unsafe` séparé. Si nous essayons d'appeler `dangereux` sans le bloc `unsafe`, nous obtenons une erreur :

```
$ cargo run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0133]: call to unsafe function is unsafe and requires unsafe function or
block
 --> src/main.rs:4:5
  |
4 |     dangereux();
  |     ^^^^^^^^^^ call to unsafe function
  |
  = note: consult the function's documentation for information on how to avoid
undefined behavior

For more information about this error, try `rustc --explain E0133`.
error: could not compile `unsafe-example` due to previous error
```

En ajoutant le bloc `unsafe` autour de notre appel à `dangereux`, nous déclarons à Rust que nous avons lu la documentation de la fonction, que nous comprenons comment l'utiliser correctement et que nous avons vérifié que nous répondons bien aux exigences de la fonction.

Les corps des fonctions non sécurisées sont bel et bien des blocs `unsafe`, donc pour pouvoir procéder à d'autres opérations non sécurisées dans une fonction non sécurisée, nous n'avons pas besoin d'ajouter un autre bloc `unsafe`.

Créer une abstraction sécurisée sur du code non sécurisé

Ce n'est pas parce qu'une fonction contient du code non sécurisé que nous devons forcément marquer l'intégralité de cette fonction comme non sécurisée. En fait, envelopper

du code non sécurisé dans une fonction sécurisée est une abstraction courante. Par exemple, étudions une fonction de la bibliothèque standard, `split_at_mut`, qui nécessite du code non sécurisé, et étudions comment nous devrions l'implémenter. Cette méthode sécurisée est définie sur des slices mutables : elle prend une slice en paramètre et en créée deux autres en divisant la slice à l'indice donné en argument. L'encart 19-4 montre comment utiliser `split_at_mut`.

```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split_at_mut(3);

assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);
```

Encart 19-4 : utilisation de la fonction sécurisée `split_at_mut`

Nous ne pouvons pas implémenter cette fonction en utilisant uniquement du Rust sécurisé. Une tentative en ce sens ressemblerait à l'encart 19-5, qui ne se compilera pas. Par simplicité, nous allons implémenter `split_at_mut` comme une fonction plutôt qu'une méthode et seulement pour des slices de valeurs `i32` au lieu d'un type générique `T`.

```
fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();

    assert!(mid <= len);

    (&mut values[..mid], &mut values[mid..])
}
```

Encart 19-5 : une tentative d'implémentation de `split_at_mut` en utilisant uniquement du Rust sécurisé

Cette fonction commence par obtenir la longueur totale de la slice. Elle vérifie ensuite que l'indice donné en paramètre est bien à l'intérieur de la slice en vérifiant s'il est inférieur ou égal à la longueur. La vérification implique que si nous envoyons un indice qui est plus grand que la longueur de la slice à découper, la fonction va paniquer avant d'essayer d'utiliser cet indice.

Ensuite, nous retournons deux slices mutables dans un tuple : une à partir du début de la slice initiale jusqu'à l'indice `mod` et une autre à partir de l'indice jusqu'à la fin de la slice.

Lorsque nous essayons de compiler le code de l'encart 19-5, nous allons obtenir une erreur.

```
$ cargo run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0499]: cannot borrow `*values` as mutable more than once at a time
 --> src/main.rs:6:30
  |
1 | fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32])
{
  |                         - let's call the lifetime of this reference `'1`
...
6 |     (&mut values[..mid], &mut values[mid..])
  |     --------------------------^^^^^^--------
  |     |     |                   |
  |     |     |                   second mutable borrow occurs here
  |     |     first mutable borrow occurs here
  |     returning this value requires that `*values` is borrowed for `'1`

For more information about this error, try `rustc --explain E0499`.
error: could not compile `unsafe-example` due to previous error
```

Le vérificateur d'emprunt de Rust ne comprend pas que nous empruntons différentes parties de la slice ; il comprend seulement que nous empruntons la même slice à deux reprises. L'emprunt de différentes parties d'une slice ne pose fondamentalement pas de problèmes car les deux slices ne se chevauchent pas, mais Rust n'est pas suffisamment intelligent pour comprendre ceci. Lorsque nous savons que ce code est correct, mais que Rust ne le sait pas, il est approprié d'utiliser du code non sécurisé.

L'encart 19-6 montre comment utiliser un bloc `unsafe`, un pointeur brut, et quelques appels à des fonctions non sécurisées pour construire une implémentation de `split_at_mut` qui fonctionne.

```
use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}
```

Encart 19-6 : utilisation de code non sécurisé dans l'implémentation de la fonction `split_at_mut`

Souvenez-vous de la section “Le type slice” du chapitre 4 dans laquelle nous avions dit qu'une slice est définie par un pointeur vers une donnée ainsi qu'une longueur de la slice. Nous avons utilisé la méthode `len` pour obtenir la longueur d'une slice ainsi que la méthode `as_mut_ptr` pour accéder au pointeur brut d'une slice. Dans ce cas, comme nous avons une slice mutable de valeurs `i32`, `as_mut_ptr` retourne un pointeur brut avec le type `*mut i32` que nous stockons dans la variable `ptr`.

Nous avons conservé la vérification que l'indice `mid` soit dans la slice. Ensuite, nous utilisons le code non sécurisé : la fonction `slice::from_raw_parts_mut` prend en paramètre un pointeur brut et une longueur, et elle créée une slice. Nous utilisons cette fonction pour créer une slice qui débute à `ptr` et qui est longue de `mid` éléments. Ensuite nous faisons appel à la méthode `add` sur `ptr` avec `mid` en argument pour obtenir un pointeur brut qui démarre à `mid`, et nous créons une slice qui utilise ce pointeur et le nombre restant d'éléments après `mid` comme longueur.

La fonction `slice::from_raw_parts_mut` est non sécurisée car elle prend en argument un pointeur brut et doit avoir confiance en la validité de ce pointeur. La méthode `add` sur les pointeurs bruts est aussi non sécurisée, car elle doit croire que l'emplacement décalé est aussi un pointeur valide. Voilà pourquoi nous avons placé un bloc `unsafe` autour de nos appels à `slice::from_raw_parts_mut` et `add` afin que nous puissions les effectuer. En analysant le code et en ayant ajouté la vérification que `mid` doit être inférieur ou égal à `len`, nous pouvons affirmer que tous les pointeurs bruts utilisés dans le bloc `unsafe` sont des pointeurs valides vers les données de la slice. C'est une utilisation acceptable et appropriée de `unsafe`.

Remarquez que nous n'avons pas eu besoin de marquer la fonction résultante `split_at_mut` comme étant `unsafe`, et que nous pouvons faire appel à cette fonction dans du code Rust sécurisé. Nous avons créé une abstraction sécurisée du code non sécurisé avec une implémentation de la fonction qui utilise de manière sécurisée du code non sécurisé, car elle créée uniquement des pointeurs valides à partir des données auxquelles cette fonction a accès.

En contre-partie, l'utilisation de `slice::from_raw_parts_mut` dans l'encart 19-7 peut planter lorsque la slice sera utilisée. Ce code prend un emplacement arbitraire dans la mémoire et crée un slice de 10 000 éléments.

```
use std::slice;

let addresse = 0x01234usize;
let r = addresse as *mut i32;

let valeurs: &[i32] = unsafe { slice::from_raw_parts_mut(r, 10000) };
```

Encart 19-7 : création d'une slice à partir d'un emplacement mémoire arbitraire

Nous ne possédons pas la mémoire à cet emplacement arbitraire, et il n'y a aucune garantie que la slice créée par ce code contiennent des valeurs `i32` valides. Toute tentative d'utilisation de `valeurs` aura un comportement imprévisible bien qu'il s'agisse d'une slice valide.

Utiliser des fonctions `extern` pour faire appel à du code externe

Parfois, votre code Rust peut avoir besoin d'interagir avec du code écrit dans d'autres langages. Dans ce cas, Rust propose un mot-clé, `extern`, qui facilite la création et l'utilisation du *Foreign Function Interface (FFI)*. Le FFI est un outil permettant à un langage de programmation de définir des fonctions auxquelles d'autres langages de programmation pourront faire appel.

L'encart 19-8 montre comment configurer l'intégration de la fonction `abs` de la bibliothèque standard du C. Les fonctions déclarées dans des blocs `extern` sont toujours non sécurisées lorsqu'on les utilise dans du code Rust. La raison à cela est que les autres langages n'appliquent pas les règles et garanties de Rust, Rust ne peut donc pas les vérifier, si bien que la responsabilité de s'assurer de la sécurité revient au développeur.

Fichier : src/main.rs

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("La valeur absolue de -3 selon le langage C : {}", abs(-3));
    }
}
```

Encart 19-8 : déclaration et appel à une fonction externe qui est définie dans un autre langage

Au sein du bloc `extern "C"`, nous listons les noms et les signatures des fonctions externes de l'autre langage que nous souhaitons solliciter. La partie "C" définit quelle est l'*application binary interface (ABI)* que la fonction doit utiliser : l'ABI définit comment faire appel à la fonction au niveau assembleur. L'ABI `"C"` est la plus courante et respecte l'ABI du langage de programmation C.

Faire appel à des fonctions Rust dans d'autres langages

Nous pouvons aussi utiliser `extern` pour créer une interface qui permet à d'autres langages de faire appel à des fonctions Rust. Au lieu d'avoir un bloc `extern`, nous ajoutons le mot-clé `extern` et nous renseignons l'ABI à utiliser juste avant le mot-clé `fn`. Nous avons aussi besoin d'ajouter l'annotation `#[no_mangle]` pour dire au compilateur Rust de ne pas déformer le nom de cette fonction. La *déformation* s'effectue lorsqu'un compilateur change le nom que nous avons donné à une fonction pour un nom qui contient plus d'informations pour d'autres étapes du processus de compilation, mais qui est moins lisible par l'humain. Tous les compilateurs de langages de programmation déforment les noms de façon légèrement différente, donc pour que le nom d'une fonction Rust soit utilisable par d'autres langages, nous devons désactiver la déformation du nom par le compilateur de Rust.

Lire ou modifier une variable statique mutable

Jusqu'à présent, nous n'avons pas parlé des *variables globales*, que Rust accepte mais qui peuvent poser des problèmes avec les règles de possession de Rust. Si deux tâches accèdent en même temps à la même variable globale, cela peut causer un accès concurrent.

En Rust, les variables globales s'appellent des variables *statiques*. L'encart 19-9 montre un exemple de déclaration et d'utilisation d'une variable statique avec une slice de chaîne de caractères comme valeur.

Fichier : src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("Cela vaut : {}", HELLO_WORLD);
}
```

Encart 19-9 : définition et utilisation d'une variable statique immuable

Les variables statiques ressemblent aux constantes, que nous avons vues dans la section “Différences entre les variables et les constantes” du chapitre 3. Les noms des variables statiques sont par convention en `SCREAMING_SNAKE_CASE`. Les variables statiques peuvent uniquement stocker des références ayant la durée de vie `'static`, de façon à ce que le compilateur Rust puisse la déterminer tout seul et que nous n'ayons pas besoin de la renseigner explicitement. L'accès à une variable statique immuable est sécurisé.

Les constantes et les variables statiques immuables se ressemblent, mais leur différence subtile est que les valeurs dans les variables statiques ont une adresse fixe en mémoire. L'utilisation de sa valeur va toujours accéder à la même donnée. Les constantes en revanche, peuvent reproduire leurs données à chaque fois qu'elles sont utilisées.

Une autre différence entre les constantes et les variables statiques est que les variables statiques peuvent être mutables. Lire et modifier des variables statiques mutables est *non sécurisé*. L'encart 19-10 montre comment déclarer, lire et modifier la variable statique mutable `COMPTEUR`.

Fichier : src/main.rs

```
static mut COMPTEUR: u32 = 0;

fn ajouter_au_compteur(valeur: u32) {
    unsafe {
        COMPTEUR += valeur;
    }
}

fn main() {
    ajouter_au_compteur(3);

    unsafe {
        println!("COMPTEUR : {}", COMPTEUR);
    }
}
```

Encart 19-10 : la lecture et l'écriture d'une variable statique mutable est non sécurisé

Comme avec les variables classiques, nous renseignons la mutabilité en utilisant le mot-clé `mut`. Tout code qui lit ou modifie `COMPTEUR` doit se trouver dans un bloc `unsafe`. Ce code se compile et affiche `COMPTEUR : 3` comme nous l'espérions car nous n'avons qu'une seule tâche. Si nous avions plusieurs tâches qui accèdent à `COMPTEUR`, nous pourrions avoir un accès concurrent.

Avec les données mutables qui sont accessibles globalement, il devient difficile de s'assurer qu'il n'y a pas d'accès concurrent, c'est pourquoi Rust considère les variables statiques mutables comme étant non sécurisées. Lorsque c'est possible, il vaut mieux utiliser les techniques de concurrence et les pointeurs intelligents adaptés au multitâche que nous avons vus au chapitre 16, afin que le compilateur puisse vérifier que les données qu'utilisent les différentes tâches sont sécurisées.

Implémenter un trait non sécurisé

Un autre cas d'usage de `unsafe` est l'implémentation d'un trait non sécurisé. Un trait n'est pas sécurisé lorsque au moins une de ses méthodes contient une invariante que le compilateur ne peut pas vérifier. Nous pouvons déclarer un trait qui n'est pas sécurisé en ajoutant le mot-clé `unsafe` devant `trait` et en marquant aussi l'implémentation du trait comme `unsafe`, comme dans l'encart 19-11.

```
unsafe trait Foo {
    // les méthodes vont ici
}

unsafe impl Foo for i32 {
    // les implémentations des méthodes vont ici
}

fn main() {}
```

Encart 19-11 : définition et implémentation d'un trait non sécurisé

En utilisant `unsafe impl`, nous promettons que nous veillons aux invariantes que le compilateur ne peut pas vérifier.

Par exemple, souvenez-vous des traits `Sync` et `Send` que nous avions découverts dans une section du chapitre 16 : le compilateur implémente automatiquement ces traits si nos types sont entièrement composés des types `Send` et `Sync`. Si nous implémentions un type qui contenait un type qui n'était pas `Send` ou `Sync`, tel que les pointeurs bruts, et nous souhaitions marquer ce type comme étant `Send` ou `Sync`, nous aurions dû utiliser `unsafe`. Rust ne peut pas vérifier que notre type respecte les garanties pour que ce type puisse être envoyé en toute sécurité entre des tâches ou qu'il puisse être utilisé par plusieurs tâches ; en conséquence, nous avons besoin de faire ces vérifications manuellement et le signaler avec `unsafe`.

Utiliser des champs d'un Union

La dernière action qui fonctionne uniquement avec `unsafe` est d'accéder aux champs d'un *union*. Un `union` ressemble à une `struct`, mais un seul champ de ceux déclarés est utilisé dans une instance précise au même moment. Les unions sont principalement utilisés pour s'interfacer avec les unions du code C. L'accès aux champs des unions n'est pas sécurisé car Rust ne peut pas garantir le type de la donnée qui est actuellement stockée dans l'instance de l'union. Vous pouvez en apprendre plus sur les unions dans the Rust Reference.

Quand utiliser du code non sécurisé

L'utilisation de `unsafe` pour mettre en oeuvre une des cinq actions (ou super-pouvoirs) que nous venons d'aborder n'est pas une mauvaise chose et ne doit pas être mal vu. Mais il est plus difficile de sécuriser du code `unsafe` car le compilateur ne peut pas aider à garantir la sécurité de la mémoire. Lorsque vous avez une bonne raison d'utiliser du code non sécurisé, vous pouvez le faire, et vous aurez l'annotation explicite `unsafe` pour faciliter la recherche de la source des problèmes lorsqu'ils surviennent.

Les traits avancés

Nous avons vu les traits dans une section du chapitre 10, mais nous n'avons pas abordé certains détails plus avancés. Maintenant que vous en savez plus sur Rust, nous pouvons attaquer les choses sérieuses.

Placer des types à remplacer dans les définitions des traits grâce aux types associés

Les *types associés* connectent un type à remplacer avec un trait afin que la définition des méthodes puisse utiliser ces types à remplacer dans leur signature. Celui qui implémente un trait doit renseigner un type concret pour être utilisé à la place du type à remplacer pour cette implémentation précise. Ainsi, nous pouvons définir un trait qui utilise certains types sans avoir besoin de savoir exactement quels sont ces types jusqu'à ce que ce trait soit implémenté.

Nous avions dit que vous auriez rarement besoin de la plupart des fonctionnalités avancées de ce chapitre. Les types associés sont un entre-deux : ils sont utilisés plus rarement que les fonctionnalités expliquées dans le reste de ce livre, mais on les rencontre plus fréquemment que la plupart des autres fonctionnalités présentées dans ce chapitre.

Un exemple de trait avec un type associé est le trait `Iterator` que fournit la bibliothèque standard. Le type associé `Item` permet de renseigner le type des valeurs que le type qui implémente le trait `Iterator` parcourt. Dans une section du chapitre 13, nous avions mentionné que la définition du trait `Iterator` ressemblait à cet encart 19-12.

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

Encart 19-12 : la définition du trait `Iterator` qui a un type `Item` associé

Le type `Item` est un type à remplacer, et la définition de la méthode `next` informe qu'elle va retourner des valeurs du type `Option<Self::Item>` . Ceux qui implémenterons le trait `Iterator` devront renseigner un type concret pour `Item` , et la méthode `next` va retourner une `Option` qui contiendra une valeur de ce type concret.

Les types associés ressemblent au même concept que les génériques, car ces derniers nous permettent de définir une fonction sans avoir à renseigner les types avec lesquels elle

travaille. Donc pourquoi utiliser les types associés ?

Examinons les différences entre les deux concepts grâce à un exemple du chapitre 13 qui implémente le trait `Iterator` sur la structure `Compteur`. Dans l'encart 13-21, nous avions renseigné que le type `Item` était `u32` :

Fichier : src/lib.rs

```
impl Iterator for Compteur {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // -- partie masquée ici --
```

Cette syntaxe ressemble aux génériques. Donc pourquoi ne pas simplement définir le trait `Iterator` avec les génériques, comme dans l'encart 19-13 ?

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

Encart 19-13 : une définition hypothétique du trait `Iterator` en utilisant des génériques

La différence est que lorsque on utilise les génériques, comme dans l'encart 19-13, on doit annoter les types dans chaque implémentation ; et comme nous pouvons aussi implémenter `Iterator<String> for Compteur` ou tout autre type, nous pourrions alors avoir plusieurs implémentations de `Iterator` pour `Compteur`. Autrement dit, lorsqu'un trait a un paramètre générique, il peut être implémenté sur un type plusieurs fois, en changeant à chaque fois le type concret du paramètre de type générique. Lorsque nous utilisons la méthode `next` sur `Compteur`, nous devons appliquer une annotation de type pour indiquer quelle implémentation de `Iterator` nous souhaitons utiliser.

Avec les types associés, nous n'avons pas besoin d'annoter les types car nous ne pouvons pas implémenter un trait plusieurs fois sur un même type. Dans l'encart 19-12 qui contient la définition qui utilise les types associés, nous ne pouvons choisir quel sera le type de `Item` qu'une seule fois, car il ne peut y avoir qu'un seul `impl Iterator for Compteur`. Nous n'avons pas à préciser que nous souhaitons avoir un itérateur de valeurs `u32` à chaque fois que nous faisons appel à `next` sur `Compteur`.

Les paramètres de types génériques par défaut et la surcharge d'opérateur

Lorsque nous utilisons les paramètres de types génériques, nous pouvons renseigner un type concret par défaut pour le type générique. Cela évite de contraindre ceux qui implémentent ce trait d'avoir à renseigner un type concret si celui par défaut fonctionne bien. La syntaxe pour renseigner un type par défaut pour un type générique est `<TypeARemplacer=TypeConcret>` lorsque nous déclarons le type générique.

Un bon exemple d'une situation pour laquelle cette technique est utile est avec la surcharge d'opérateurs. *La surcharge d'opérateur* permet de personnaliser le comportement d'un opérateur (comme `+`) dans des cas particuliers.

Rust ne vous permet pas de créer vos propres opérateurs ou de surcharger des opérateurs. Mais vous pouvez surcharger les opérations et les traits listés dans `std::ops` en implémentant les traits associés à l'opérateur. Par exemple, dans l'encart 19-14 nous surchargeons l'opérateur `+` pour additionner ensemble deux instances de `Point`. Nous pouvons faire cela en implémentant le trait `Add` sur une structure `Point` :

Fichier : src/main.rs

```
use std::ops::Add;

#[derive(Debug, Copy, Clone, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(
        Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
        Point { x: 3, y: 3 }
    );
}
```

Encart 19-14 : implémentation du trait `Add` pour surcharger l'opérateur `+` pour les instances de `Point`

La méthode `add` ajoute les valeurs `x` de deux instances de `Point` ainsi que les valeurs `y` de deux instances de `Point` pour créer un nouveau `Point`. Le trait `Add` a un type associé `Output` qui détermine le type retourné pour la méthode `add`.

Le type générique par défaut dans ce code est dans le trait `Add`. Voici sa définition :

```
trait Add<Rhs=Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

Ce code devrait vous être familier : un trait avec une méthode et un type associé. La nouvelle partie concerne `Rhs=Self` : cette syntaxe s'appelle les *paramètres de types par défaut*. Le paramètre de type générique `Rhs` (c'est le raccourci de “Right Hand Side”) qui définit le type du paramètre `rhs` dans la méthode `add`. Si nous ne renseignons pas de type concret pour `Rhs` lorsque nous implémentons le trait `Add`, le type de `Rhs` sera par défaut `Self`, qui sera le type sur lequel nous implémentons `Add`.

Lorsque nous avons implémenté `Add` sur `Point`, nous avons utilisé la valeur par défaut de `Rhs` car nous voulions additionner deux instances de `Point`. Voyons un exemple d'implémentation du trait `Add` dans lequel nous souhaitons personnaliser le type `Rhs` plutôt que d'utiliser celui par défaut.

Nous avons deux structures, `Millimetres` et `Metres`, qui stockent des valeurs dans différentes unités. Ce léger enrobage d'un type existant dans une autre structure s'appelle le *motif newtype*, que nous décrivons plus en détail dans la section Utiliser le motif newtype pour la sécurité et l'abstraction des types. Nous voulons pouvoir additionner les valeurs en millimètres avec les valeurs en mètres et appliquer l'implémentation de `Add` pour pouvoir faire la conversion correctement. Nous pouvons implémenter `Add` sur `Millimetres` avec `Metres` comme étant le `Rhs`, comme dans l'encart 19-15.

Fichier : src/lib.rs

```
use std::ops::Add;

struct Millimetres(u32);
struct Metres(u32);

impl Add<Metres> for Millimetres {
    type Output = Millimetres;

    fn add(self, other: Metres) -> Millimetres {
        Millimetres(self.0 + (other.0 * 1000))
    }
}
```

Encart 19-15 : implémentation du trait `Add` sur `Millimetres` pour pouvoir additionner `Millimetres` à `Metres`

Pour additionner `Millimetres` et `Metres`, nous renseignons `impl Add<Metres>` pour régler la valeur du paramètre de type `Rhs` au lieu d'utiliser la valeur par défaut `Self`.

Vous utiliserez les paramètres de types par défaut dans deux principaux cas :

- Pour étendre un type sans casser le code existant
- Pour permettre la personnalisation dans des cas spécifiques que la plupart des utilisateurs n'auront pas

Le trait `Add` de la bibliothèque standard est un exemple du second cas : généralement, vous additionnez deux types similaires, mais le trait `Add` offre la possibilité de personnaliser cela. L'utilisation d'un paramètre de type par défaut dans la définition du trait `Add` signifie que vous n'aurez pas à renseigner de paramètre en plus la plupart du temps. Autrement dit, il n'est pas nécessaire d'avoir recours à des assemblages de code, ce qui facilite l'utilisation du trait.

Le premier cas est similaire au second mais dans le cas inverse : si vous souhaitez ajouter un paramètre de type à un trait existant, vous pouvez lui en donner un par défaut pour permettre l'ajout des fonctionnalités du trait sans casser l'implémentation actuelle du code.

La syntaxe totalement définie pour clarifier les appels à des méthodes qui ont le même nom

Il n'y a rien en Rust qui empêche un trait d'avoir une méthode portant le même nom qu'une autre méthode d'un autre trait, ni ne vous empêche d'implémenter ces deux traits sur un même type. Il est aussi possible d'implémenter directement une méthode avec le même nom que celle présente dans les traits sur ce type.

Lorsque nous faisons appel à des méthodes qui ont un conflit de nom, vous devez préciser à Rust précisément celle que vous souhaitez utiliser. Imaginons le code dans l'encart 19-16 dans lequel nous avons défini deux traits, `Pilote` et `Magicien`, qui ont tous les deux une méthode `voler`. Nous implémentons ensuite ces deux traits sur un type `Humain` qui a déjà lui-aussi une méthode `voler` qui lui a été implémentée. Chaque méthode `voler` fait quelque chose de différent.

Fichier : src/main.rs

```
trait Pilote {
    fn voler(&self);
}

trait Magicien {
    fn voler(&self);
}

struct Humain;

impl Pilote for Humain {
    fn voler(&self) {
        println!("Ici le capitaine qui vous parle.");
    }
}

impl Magicien for Humain {
    fn voler(&self) {
        println!("Décollage !");
    }
}

impl Humain {
    fn voler(&self) {
        println!("*agite frénétiquement ses bras*");
    }
}
```

Encart 19-16 : deux traits qui ont une méthode `voler` et qui sont implémentés sur le type `Humain`, et une méthode `voler` est aussi implémentée directement sur `Humain`

Lorsque nous utilisons `voler` sur une instance de `Humain`, le compilateur fait appel par défaut à la méthode qui est directement implémentée sur le type, comme le montre l'encart 19-17.

Fichier : src/main.rs

```
fn main() {
    let une_personne = Humain;
    une_personne.voler();
}
```

Encart 19-17 : utilisation de `voler` sur une instance de `Humain`

L'exécution de ce code va afficher `*agite frénétiquement ses bras*`, ce qui démontre que Rust a appelé la méthode `voler` implémentée directement sur `Humain`.

Pour faire appel aux méthodes `voler` des traits `Pilote` ou `Magicien`, nous devons utiliser une syntaxe plus explicite pour préciser quelle méthode `voler` nous souhaitons utiliser. L'encart 19-18 montre cette syntaxe.

Fichier : src/main.rs

```
fn main() {
    let une_personne = Humain;
    Pilote::voler(&une_personne);
    Magicien::voler(&une_personne);
    une_personne.voler();
}
```

Encart 19-18 : préciser de quel trait nous souhaitons utiliser la méthode `voler`

Si on renseigne le nom du trait avant le nom de la méthode, cela indique à Rust quelle implémentation de `voler` nous souhaitons utiliser. Nous pouvons aussi écrire `Humain::voler(&une_personne)`, qui est équivalent à `une_personne.voler()` que nous avons utilisé dans l'encart 19-18, mais c'est un peu plus long à écrire si nous n'avons pas besoin de préciser les choses.

L'exécution de ce code affiche ceci :

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.46s
     Running `target/debug/traits-example`
Ici le capitaine qui vous parle.
Décollage !
*agite frénétiquement ses bras*
```

Comme la méthode `voler` prend un paramètre `self`, si nous avions deux *types* qui implémentaient chacun un des deux *traits*, Rust pourrait en déduire quelle implémentation de quel trait utiliser en fonction du type de `self`.

Cependant, les fonctions associées qui ne sont pas des méthodes n'ont pas de paramètre

`self`. Lorsqu'il y a plusieurs types ou traits qui définissent des fonctions qui ne sont pas des méthodes et qui ont le même nom de fonction, Rust ne peut pas toujours savoir quel type vous sous-entendez jusqu'à ce que vous utilisiez la *syntaxe totalement définie*. Par exemple, le trait `Animal` de l'encart 19-19 a une fonction associée `nom_bebe` qui n'est pas une méthode, et le trait `Animal` est implémenté pour la structure `Dog`.Il y a aussi une fonction associée `nom_bebe` qui n'est pas une méthode et qui est définie directement sur `Chien`.

Fichier : src/main.rs

```
trait Animal {
    fn nom_bebe() -> String;
}

struct Chien;

impl Chien {
    fn nom_bebe() -> String {
        String::from("Spot")
    }
}

impl Animal for Chien {
    fn nom_bebe() -> String {
        String::from("chiot")
    }
}

fn main() {
    println!("Un bébé chien s'appelle un {}", Chien::nom_bebe());
}
```

Encart 19-19 : un trait avec une fonction associée et un type avec une autre fonction associée qui porte le même nom et qui implémente aussi ce trait

Ce code a été conçu pour un refuge pour animaux qui souhaite que tous leurs chiots soient nommés Spot, ce qui est implémenté dans la fonction associée `nom_bebe` de `Chien`. Le type `Chien` implémente lui aussi le trait `Animal`, qui décrit les caractéristiques que tous les animaux doivent avoir. Les bébés chiens doivent s'appeler des chiots, et ceci est exprimé dans l'implémentation du trait `Animal` sur `Chien` dans la fonction `nom_bebe` associée au trait `Animal`.

Dans le `main`, nous faisons appel à la fonction `Chien::nom_bebe`, qui fait appel à la fonction associée directement définie sur `Chien`. Ce code affiche ceci :

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.54s
     Running `target/debug/traits-example`
Un bébé chien s'appelle un Spot
```

Ce résultat n'est pas celui que nous souhaitons. Nous voulons appeler la fonction `nom_bebe` qui fait partie du trait `Animal` que nous avons implémenté sur `Chien` afin que le code affiche `Un bébé chien s'appelle un chiot`. La technique pour préciser le nom du trait que nous avons utilisée précédemment ne va pas nous aider ici ; si nous changeons le `main` par le code de l'encart 19-20, nous allons avoir une erreur de compilation.

Fichier : src/main.rs

```
fn main() {
    println!("Un bébé chien s'appelle un {}", Animal::nom_bebe());
}
```

Encart 19-20 : tentative d'appel à la fonction `nom_bebe` du trait `Animal`, mais Rust ne sait pas quelle implémentation utiliser

Comme `Animal::nom_bebe` n'a pas de paramètre `self`, et qu'il peut y avoir d'autres types qui implémentent le trait `Animal`, Rust ne peut pas savoir quelle implémentation de `Animal::nom_bebe` nous souhaitons utiliser. Nous obtenons alors cette erreur de compilation :

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0283]: type annotations needed
  --> src/main.rs:20:43
   |
20 |     println!("Un bébé chien s'appelle un {}", Animal::nom_bebe());
   |                                                ^^^^^^^^^^^^^^^^ cannot infer
type
   |
   = note: cannot satisfy `_: Animal`

For more information about this error, try `rustc --explain E0283`.
error: could not compile `traits-example` due to previous error
```

Pour expliquer à Rust que nous souhaitons utiliser l'implémentation de `Animal` pour `Chien` et non pas l'implémentation de `Animal` pour d'autres types, nous devons utiliser la syntaxe totalement définie. L'encart 19-21 montre comment utiliser la syntaxe totalement définie.

Fichier : src/main.rs

```
fn main() {
    println!("Un bébé chien s'appelle un {}", <Chien as Animal>::nom_bebe());
}
```

Encart 19-21 : utilisation de la syntaxe totalement définie pour préciser que nous souhaitons appeler la fonction `nom_bebe` du trait `Animal` tel qu'il est implémenté sur `Chien`

Nous avons donné à Rust une annotation de type entre des chevrons, ce qui indique que nous souhaitons appeler la méthode `nom_bebe` du trait `Animal` telle qu'elle est implémentée sur `Chien` en indiquant que nous souhaitons traiter le type `Chien` comme étant un `Animal` pour cet appel de fonction. Ce code va désormais afficher ce que nous souhaitons :

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/traits-example`
Un bébé chien s'appelle un chiot
```

De manière générale, une syntaxe totalement définie est définie comme ceci :

```
<Type as Trait>::function(destinataire_si_methode, argument_suivant, ...);
```

Pour les fonctions associées qui ne sont pas des méthodes, il n'y a pas de `destinataire` : il n'y a qu'une liste d'arguments. Vous pouvez utiliser la syntaxe totalement définie à n'importe quel endroit où vous faites appel à des fonctions ou des méthodes. Cependant, vous avez la possibilité de ne pas renseigner toute partie de cette syntaxe que Rust peut déduire à partir d'autres informations présentes dans le code. Vous avez seulement besoin d'utiliser cette syntaxe plus verbeuse dans les cas où il y a plusieurs implémentations qui utilisent le même nom et que Rust doit être aidé pour identifier quelle implémentation vous souhaitez appeler.

Utiliser les supertraits pour utiliser la fonctionnalité d'un trait dans un autre trait

Des fois, vous pourriez avoir besoin d'un trait pour utiliser la fonctionnalité d'un autre trait. Dans ce cas, vous devez pouvoir compter sur le fait que le trait dépendant soit bien implémenté. Le trait sur lequel vous comptez est alors un *supertrait* du trait que vous implémentez.

Par exemple, imaginons que nous souhaitons créer un trait `OutlinePrint` qui offre une méthode `outline_print` affichant une valeur entourée d'astérisques. Ainsi, pour une

structure `Point` qui implémente `Display` pour afficher `(x, y)`, lorsque nous faisons appel à `outline_print` sur une instance de `Point` qui a `1` pour valeur de `x` et `3` pour `y`, cela devrait afficher ceci :

```
**********
*        *
* (1, 3) *
*        *
**********
```

Dans l'implémentation de `outline_print`, nous souhaitons utiliser la fonctionnalité du trait `Display`. De ce fait, nous devons indiquer que le trait `OutlinePrint` fonctionnera uniquement pour les types qui auront également implémenté `Display` et qui fourniront la fonctionnalité dont a besoin `OutlinePrint`. Nous pouvons faire ceci dans la définition du trait en renseignant `OutlinePrint: Display`. Cette technique ressemble à l'ajout d'un trait lié au trait. L'encart 19-22 montre une implémentation du trait `OutlinePrint`.

Fichier : src/main.rs

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let valeur = self.to_string();
        let largeur = valeur.len();
        println!("{}", "*".repeat(largeur + 4));
        println!("*{}*", " ".repeat(largeur + 2));
        println!("* {} *", valeur);
        println!("*{}*", " ".repeat(largeur + 2));
        println!("{}", "*".repeat(largeur + 4));
    }
}
```

Encart 19-22 : implémentation du trait `OutlinePrint` qui nécessite la fonctionnalité offerte par `Display`

Comme nous avons précisé que `OutlinePrint` nécessite le trait `Display`, nous pouvons utiliser la fonction `to_string` qui est automatiquement implémentée pour n'importe quel type qui implémente `Display`. Si nous avions essayé d'utiliser `to_string` sans ajouter un double-point et en renseignant le trait `Display` après le nom du trait, nous aurions alors obtenu une erreur qui nous informerait qu'il n'y a pas de méthode `to_string` pour le type `&Self` dans la portée courante.

Voyons ce qui ce passe lorsque nous essayons d'implémenter `OutlinePrint` sur un type qui n'implémente pas `Display`, comme c'est le cas de la structure `Point` :

Fichier : src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}
```

Nous obtenons une erreur qui dit que `Display` est nécessaire mais n'est pas implémenté :

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0277]: `Point` doesn't implement `std::fmt::Display`
  --> src/main.rs:20:6
   |
20 | impl OutlinePrint for Point {}
   |      ^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Point`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-
print) instead
note: required by a bound in `OutlinePrint`
  --> src/main.rs:3:21
   |
3  | trait OutlinePrint: fmt::Display {
   |                     ^^^^^^^^^^^^ required by this bound in `OutlinePrint`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `traits-example` due to previous error
```

Pour régler cela, nous implémentons `Display` sur `Point` afin de répondre aux besoins de `OutlinePrint`, comme ceci :

Fichier : src/main.rs

```
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

Ceci fait, l'implémentation du trait `OutlinePrint` sur `Point` va se compiler avec succès, et nous pourrons appeler `outline_print` sur une instance de `Point` pour l'afficher dans le cadre constitué d'astérisques.

Utiliser le motif newtype pour implémenter des traits externes sur des types externes

Dans une section du chapitre 10, nous avions mentionné la règle de l'orphelin qui énonçait que nous pouvions implémenter un trait sur un type à condition que le trait ou le type soit local à notre crate. Il est possible de contourner cette restriction en utilisant le *motif newtype*, ce qui implique de créer un nouveau type dans une structure tuple (nous avons vu les structures tuple dans la section "Utilisation de structures tuples sans champ nommé pour créer des types différents" du chapitre 5). La structure tuple aura un champ et sera une petite enveloppe pour le type sur lequel nous souhaitons implémenter le trait. Ensuite, le type enveloppant est local à notre crate, et nous pouvons lui implémenter un trait. *Newtype* est un terme qui provient du langage de programmation Haskell. Il n'y a pas de conséquence sur les performance à l'exécution pour l'utilisation de ce motif, ce qui signifie que le type enveloppant est résolu à la compilation.

Comme exemple, disons que nous souhaitons implémenter `Display` sur `Vec<T>`, ce que la règle de l'orphelin nous empêche de faire directement car le trait `Display` et le type `Vec<T>` sont définis en dehors de notre crate. Nous pouvons construire une structure `Enveloppe` qui possède une instance de `Vec<T>` ; et ensuite nous pouvons implémenter `Display` sur `Enveloppe` et utiliser la valeur `Vec<T>`, comme dans l'encart 19-23.

Fichier : src/main.rs

```
use std::fmt;

struct Enveloppe(Vec<String>);

impl fmt::Display for Enveloppe {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Enveloppe(vec![String::from("hello"), String::from("world")]);
    println!("w = {}", w);
}
```

Encart 19-23 : création d'un type `Enveloppe` autour de `Vec<String>` pour implémenter `Display`

L'implémentation de `Display` utilise `self.0` pour accéder à la valeur de `Vec<T>`, car `Enveloppe` est une structure tuple et `Vec<T>` est l'élément à l'indice 0 du tuple. Ensuite, nous pouvons utiliser la fonctionnalité du type `Display` sur `Enveloppe`.

Le désavantage d'utiliser cette technique est que `Enveloppe` est un nouveau type, donc il n'implémente pas toutes les méthodes de la valeur qu'il possède. Il faudrait implémenter toutes les méthodes de `Vec<T>` directement sur `Enveloppe` de façon à ce qu'elles délèguent aux méthodes correspondantes de `self.0`, ce qui nous permettrait d'utiliser `Enveloppe` exactement comme un `Vec<T>`. Si nous voulions que le nouveau type ait toutes les méthodes du type qu'il possède, l'implémentation du trait `Deref` (que nous avons vu dans une section du chapitre 15) sur `Enveloppe` pour retourner le type interne pourrait être une solution. Si nous ne souhaitons pas que le type `Enveloppe` ait toutes les méthodes du type qu'il possède (par exemple, pour limiter les fonctionnalités du type `Enveloppe`), nous n'avons qu'à implémenter manuellement que les méthodes que nous souhaitons.

Maintenant vous savez comment le motif newtype est utilisé en lien avec les traits ; c'est aussi un motif très utile même lorsque les traits ne sont pas concernés. Changeons de sujet et découvrons d'autres techniques avancées pour interagir avec le système de type de Rust.

Les types avancés

Le système de type de Rust offre quelques fonctionnalités que nous avons mentionnées dans ce livre mais que nous n'avons pas encore étudiées. Nous allons commencer par voir les newtypes en général lorsque nous examinerons pourquoi les newtypes sont des types utiles. Ensuite nous nous pencherons sur les alias de type, une fonctionnalité qui ressemble aux newtypes mais avec quelques différences sémantiques. Nous allons aussi voir le type `!` et les types à taille dynamique.

Utiliser le motif newtype pour la sécurité et l'abstraction des types

> Remarque : cette section suppose que vous avez lu la section précédente

Le motif newtype est utile pour faire des choses qui vont au-delà de ce que nous avons vu jusqu'à présent, notamment pour s'assurer statiquement que des valeurs ne soient jamais confondues ou pour spécifier les unités d'une valeur. Vous avez vu un exemple d'utilisation des newtypes pour indiquer des unités dans l'encart 19-15 : souvenez-vous des structures `Millimetres` et `Metres` qui englobaient des valeurs `u32` dans ces newtypes. Si nous avions écrit une fonction avec un paramètre de type `Millimetres`, nous ne n'aurions pas pu compiler un programme qui aurait accidentellement fait appel à cette fonction avec une valeur du type `Metres` ou `u32` pur.

Une autre utilisation du motif newtype est de permettre d'abstraire certains détails d'implémentation d'un type : le newtype peut exposer une API publique qui est différente de l'API du type interne privé.

Les newtypes peuvent aussi masquer des implémentations internes. Par exemple, nous pouvons fournir un type `Personnes` pour embarquer un `HashMap<i32, String>` qui stocke l'identifiant de personnes associés à leur nom. Le code qui utilisera `Personnes` ne pourra utiliser que l'API publique que nous fournissons, telle qu'une méthode pour ajouter une chaîne de caractères en tant que nom à la collection `Personnes` ; ce code n'aura pas besoin de savoir que nous assignons en interne un identifiant `i32` aux noms. Le motif newtype est une façon allégée de procéder à de l'encapsulation pour masquer des détails d'implémentation, comme nous l'avons vu dans une partie du chapitre 17.

Créer des synonymes de types avec les alias de type

En plus du motif newtype, Rust fournit la possibilité de déclarer un *alias de type* pour donner

un autre nom à un type déjà existant. Pour faire cela, nous utilisons le mot-clé `type`. Par exemple, nous pouvons créer l'alias `Kilometres` pour un `i32`, comme ceci :

```
type Kilometres = i32;
```

Désormais, l'alias `Kilometres` est un *synonyme* de `i32` ; contrairement aux types `Millimetres` et `Metres` que nous avons créés dans l'encart 19-15, `Kilometres` n'est pas un newtype séparé. Les valeurs qui ont le type `Kilometre` seront traitées comme si elles étaient du type `i32` :

```
type Kilometres = i32;

let x: i32 = 5;
let y: Kilometres = 5;

println!("x + y = {}", x + y);
```

Comme `Kilometres` et `i32` sont du même type, nous pouvons additionner les valeurs des deux types et nous pouvons envoyer des valeurs `Kilometres` aux fonctions qui prennent des paramètres `i32`. Cependant, en utilisant cette méthode, nous ne bénéficions pas des bienfaits de la vérification du type que nous avions avec le motif newtype que nous avons vu précédemment.

L'utilisation principale pour les synonymes de types est de réduire la répétition. Par exemple, nous pourrions avoir un type un peu long tel que celui-ci :

```
Box<dyn Fn() + Send + 'static>
```

Ecrire ce type un peu long dans des signatures de fonctions et comme annotations de types tout au long du code peut s'avérer pénible et faciliter les erreurs. Imaginez que vous ayez un projet avec plein de code ressemblant à celui de l'encart 19-24.

```
let f: Box<dyn Fn() + Send + 'static> = Box::new(|| println!("salut"));

fn prend_un_long_type(f: Box<dyn Fn() + Send + 'static>) {
    // -- partie masquée ici --
}

fn retourne_un_long_type() -> Box<dyn Fn() + Send + 'static> {
    // -- partie masquée ici --
}
```

Encart 19-24 : utilisation d'un type long à écrire dans de nombreux endroits

Un alias de type simplifie ce code en réduisant la répétition. Dans l'encart 19-25, nous avons

ajouté un alias `Thunk` pour ce type verbeux, alias plus court qui peut le remplacer partout où il est utilisé.

```
type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("salut"));

fn prend_un_long_type(f: Thunk) {
    // -- partie masquée ici --
}

fn retourne_un_long_type() -> Thunk {
    // -- partie masquée ici --
}
```

Encart 19-25 : ajout et utilisation d'un alias `Thunk` pour réduire les répétitions

Ce code est plus facile à lire et écrire ! Choisir un nom plus explicite pour un alias peut aussi vous aider à communiquer ce que vous voulez faire (*thunk* est un terme désignant du code qui doit être évalué plus tard, donc c'est un nom approprié pour une fermeture qui est stockée).

Les alias de type sont couramment utilisés avec le type `Result<T, E>` pour réduire la répétition. Regardez le module `std::io` de la bibliothèque standard. Les opérations d'entrée/sortie retournent souvent un `Result<T, E>` pour gérer les situations où les opérations échouent. Cette bibliothèque a une structure `std::io::Error` qui représente toutes les erreurs possibles d'entrée/sortie. De nombreuses fonctions dans `std::io` vont retourner un `Result<T, E>` avec `E` qui est un alias pour `std::io::Error`, comme par exemple ces fonctions sont dans le trait `Write` :

```
use std::fmt;
use std::io::Error;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

Le `Result<..., Error>` est répété plein de fois. C'est pourquoi `std::io` possède cette déclaration d'alias de type :

```
type Result<T> = std::result::Result<T, std::io::Error>;
```

Comme cette déclaration est dans le module `std::io`, nous pouvons utiliser l'alias `std::io::Result<T>` — qui est un `Result<T, E>` avec le `E` qui est déjà renseigné comme étant un `std::io::Error`. Les fonctions du trait `Write` ressemblent finalement à ceci :

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<()>;
}
```

L'alias de type nous aide sur deux domaines : il permet de faciliter l'écriture du code *et* il nous donne une interface cohérente pour tout `std::io`. Comme c'est un alias, c'est simplement un autre `Result<T, E>`, ce qui signifie que nous pouvons utiliser n'importe quelle méthode qui fonctionne avec `Result<T, E>`, ainsi que les syntaxes spéciales telle que l'opérateur `?`.

Le type "jamais", qui ne retourne jamais de valeur

Rust a un type spécial qui s'appelle `!` qui est connu dans le vocabulaire de la théorie des types comme étant le *type vide* car il n'a pas de valeur. Nous préférons appeler cela le *type jamais* car il remplace le type de retour lorsqu'une fonction ne va jamais retourner quelque chose. Voici un exemple :

```
fn bar() -> ! {
    // -- partie masquée ici --
}
```

Ce code peut être interprété comme “la fonction `bar` qui ne retourne pas de valeur”. Les fonctions qui ne retournent pas de valeur s'appellent des *fonctions divergentes*. Nous ne pouvons pas créer de valeurs de type `!` donc `bar` ne pourra jamais retourner de valeur.

Mais à quoi sert un type dont on ne peut jamais créer de valeurs ? Souvenez-vous du code de l'encart 2-5 ; nous avons reproduit une partie de celui-ci dans l'encart 19-26.

```
        let supposition: u32 = match supposition.trim().parse() {
            Ok(nombre) => nombre,
            Err(_) => continue,
        };
```

Encart 19-26 : un `match` avec une branche qui finit par un `continue`

A l'époque, nous avions sauté quelques détails dans ce code. Dans la section "La structure de contrôle `match`" du chapitre 6, nous avons vu que les branches d'un `match` doivent toutes retourner le même type. Donc, par exemple, le code suivant ne fonctionne pas :

```
let supposition = match supposition.trim().parse() {
    Ok(_) => 5,
    Err(_) => "salut",
};
```

Le type de `supposition` dans ce code devrait être un entier *et* une chaîne de caractères, et Rust nécessite que `supposition` n'ait qu'un seul type possible. Donc que retourne `continue` ? Pourquoi pouvons-nous retourner un `u32` dans une branche et avoir une autre branche qui finit avec un `continue` dans l'encart 19-26 ?

Comme vous l'avez deviné, `continue` a une valeur `!`. Ainsi, lorsque Rust calcule le type de `supposition`, il regarde les deux branches, la première avec une valeur `u32` et la seconde avec une valeur `!`. Comme `!` ne peut jamais retourner de valeur, Rust décide alors que le type de `supposition` est `u32`.

Une façon classique de décrire ce comportement est de dire que les expressions du type `!` peuvent être transformées dans n'importe quel type. Nous pouvons finir cette branche de `match` avec `continue` car `continue` ne retourne pas de valeur ; à la place, il retourne le contrôle en haut de la boucle, donc dans le cas d'un `Err`, nous n'assignons jamais de valeur à `supposition`.

Ce type "jamais" est tout aussi utile avec la macro `panic!`. Vous souvenez-vous que la fonction `unwrap` que nous appelons sur les valeurs `Option<T>` fournit une valeur ou panique ? Voici sa définition :

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

Dans ce code, il se passe la même chose que l'encart 19-26 : Rust constate que `val` est du type `T` et que `panic!` est du type `!`, donc le résultat de l'ensemble de l'expression `match` est `T`. Ce code fonctionne car `panic!` ne produit pas de valeur ; il termine le programme. Dans le cas d'un `None`, nous ne retournons pas une valeur de `unwrap`, donc ce code est valide.

Une des expressions qui sont du type `!` est le `loop` :

```
print!("pour toujours ");

loop {
    print!("et toujours ");
}
```

Ici, la boucle ne se termine jamais, donc `!` est la valeur de cette expression. En revanche, cela ne sera pas vrai si nous utilisons un `break`, car la boucle va s'arrêter lorsqu'elle rencontrera le `break`.

Les types à taille dynamique et le trait `Sized`

Vu qu'il est nécessaire pour Rust de connaître certains détails, comme la quantité d'espace à allouer à une valeur d'un type donné, il y a un aspect de ce système de type qui peut être déroutant : le concept des *types à taille dynamique*. Parfois appelés *DST* (Dynamically Sized Types) ou *types sans taille*, ces types nous permettent d'écrire du code qui utilise des valeurs dont la taille ne peut être connue qu'à l'exécution.

Voyons les détails d'un type à taille dynamique qui s'appelle `str`, que nous avons utilisé dans ce livre. Notez bien que ce n'est pas `&str`, mais bien `str` qui est un DST. Nous ne pouvons connaître la longueur de la chaîne de caractère qu'à l'exécution, ce qui signifie que nous ne pouvons ni créer une variable de type `str`, ni prendre en argument un type `str`. Imaginons le code suivant, qui ne fonctionnera pas :

```
let s1: str = "Salut tout le monde !";
let s2: str = "Comment ça va ?";
```

Rust a besoin de savoir combien de mémoire allouer pour chaque valeur d'un type donné, et toutes les valeurs de ce type doivent utiliser la même quantité de mémoire. Si Rust nous avait autorisé à écrire ce code, ces deux valeurs `str` devraient occuper la même quantité de mémoire. Mais elles ont deux longueurs différentes : `s1` prend 21 octets en mémoire alors que `s2` en a besoin de 15. C'est pourquoi il est impossible de créer une variable qui stocke un type à taille dynamique.

Donc qu'est-ce qu'on peut faire ? Dans ce cas, vous connaissez déjà la réponse : nous faisons en sorte que le type de `s1` et `s2` soit `&str` plutôt que `str`. Souvenez-vous que dans la section "Les slices de chaînes de caractères" du chapitre 4, nous avions dit que la structure de données slice stockait l'emplacement de départ et la longueur de la slice.

Aussi, bien qu'un `&T` soit une valeur unique qui stocke l'adresse mémoire à laquelle se

trouve le `T`, un `&str` est constitué de *deux* valeurs : l'adresse du `str` et sa longueur. Ainsi, nous pouvons connaître la taille d'une valeur `&str` à la compilation : elle vaut deux fois la taille d'un `usize`. Ce faisant, nous connaissons toujours la taille d'un `&str`, peu importe la longueur de la chaîne de caractères sur laquelle il pointe. Généralement, c'est comme cela que les types à taille dynamique sont utilisés en Rust : ils ont des métadonnées supplémentaires qui stockent la taille des informations dynamiques. La règle d'or des types à taille dynamique est que nous devons toujours placer les valeurs à types à taille dynamique derrière un pointeur d'une certaine sorte.

Nous pouvons combiner `str` avec n'importe quel type de pointeur : par exemple, `Box<str>` ou `Rc<str>`. En fait, vous avez vu cela déjà auparavant mais avec un autre type à taille dynamique : les traits. Chaque trait est un type à taille dynamique auquel nous pouvons nous référer en utilisant le nom du trait. Dans une section du chapitre 17, nous avions mentionné que pour utiliser les traits comme des objets traits, nous devions les utiliser via un pointeur, tel que `&dyn Trait` ou `Box<dyn Trait>` (`Rc<dyn Trait>` fonctionnera également).

Pour pouvoir travailler avec les DST, Rust dispose d'un trait particulier `Sized` pour déterminer si oui ou non la taille d'un type est connue à la compilation. Ce trait est automatiquement implémenté sur tout ce qui a une taille connue à la compilation. De plus, Rust ajoute implicitement le trait lié `Sized` sur chaque fonction générique. Ainsi, la définition d'une fonction générique telle que celle-ci :

```
fn generique<T>(t: T) {
    // -- partie masquée ici --
}
```

... est en réalité traitée comme si nous avions écrit ceci :

```
fn generique<T: Sized>(t: T) {
    // -- partie masquée ici --
}
```

Par défaut, les fonctions génériques vont fonctionner uniquement sur des types qui ont une taille connue à la compilation. Cependant, vous pouvez utiliser la syntaxe spéciale suivante pour éviter cette restriction :

```
fn generique<T: ?Sized>(t: &T) {
    // -- partie masquée ici --
}
```

Le trait lié `?Sized` signifie que "`T` peut être ou ne pas être `Sized`" et cette notation prévaut sur le comportement par défaut qui dit que les types génériques doivent avoir une taille

connue au moment de la compilation. La syntaxe `?Trait` avec ce comportement n'est disponible que pour `Sized`, et pour aucun autre trait.

Remarquez aussi que nous avons changé le type du paramètre `t` de `T` en `&T`. Comme ce type pourrait ne pas être un `Sized`, nous devons l'utiliser via un pointeur d'une sorte ou d'une autre. Dans ce cas, nous avons choisi une référence.

Dans la partie suivante, nous allons parler des fonctions et des fermetures !

Les fonctions et fermetures avancées

Dans cette section, nous allons explorer quelques fonctionnalités avancées liées aux fonctions et aux fermetures, y compris les pointeurs de fonctions et la capacité de retourner des fermetures.

Pointeurs de fonctions

Nous avons déjà vu comment envoyer des fermetures dans des fonctions ; mais vous pouvez aussi envoyer des fonctions classiques dans d'autres fonctions ! Cette technique est utile lorsque vous souhaitez envoyer une fonction que vous avez déjà définie plutôt que de définir une nouvelle fermeture. Vous pouvez faire ceci avec des pointeurs de fonctions, qui vous permettent d'utiliser des fonctions en argument d'autres fonctions. Les fonctions nécessitent le type `fn` (avec un f minuscule), à ne pas confondre avec le trait de fermeture `Fn`. Le type `fn` s'appelle un *pointeur de fonction*. La syntaxe pour indiquer qu'un paramètre est un pointeur de fonction ressemble à celle des fermetures, comme vous pouvez le voir dans l'encart 19-27.

Fichier : src/main.rs

```
fn ajouter_un(x: i32) -> i32 {
    x + 1
}

fn le_faire_deux_fois(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let reponse = le_faire_deux_fois(ajouter_un, 5);

    println!("La réponse est : {}", reponse);
}
```

Encart 19-27 : utiliser le type `fn` pour accepter un pointeur de fonction en argument

Ce code affiche `La réponse est : 12`. Nous avons précisé que le paramètre `f` dans `le_faire_deux_fois` est une `fn` qui prend en argument un paramètre du type `i32` et retourne un `i32`. Nous pouvons ensuite appeler `f` dans le corps de `le_faire_deux_fois`. Dans `main`, nous pouvons envoyer le nom de la fonction `ajouter_un` dans le premier argument de `le_faire_deux_fois`.

Contrairement aux fermetures, `fn` est un type plutôt qu'un trait, donc nous indiquons `fn`

directement comme type de paramètre plutôt que de déclarer un paramètre de type générique avec un des traits `Fn` comme trait lié.

Les pointeurs de fonctions implémentent simultanément les trois traits de fermeture (`Fn` , `FnMut` et `FnOnce`) afin que vous puissiez toujours envoyer un pointeur de fonction en argument d'une fonction qui attendait une fermeture. Il vaut mieux écrire des fonctions qui utilisent un type générique et un des traits de fermeture afin que vos fonctions puissent accepter soit des fonctions, soit des fermetures.

Une situation dans laquelle vous ne voudrez accepter que des `fn` et pas des fermetures, est lorsque vous vous interfacez avec du code externe qui n'a pas de fermetures : les fonctions C peuvent accepter des fonctions en argument, mais le C n'a pas fermetures.

Comme exemple d'une situation dans laquelle vous pouvez utiliser soit une fermeture définie directement ou le nom d'une fonction, prenons l'utilisation de `map` . Pour utiliser la fonction `map` pour transformer un vecteur de nombres en vecteur de chaînes de caractères, nous pouvons utiliser une fermeture, comme ceci :

```
let liste_de_nombres = vec![1, 2, 3];
let liste_de_chaines: Vec<String> =
    liste_de_nombres.iter().map(|i| i.to_string()).collect();
```

Ou alors nous pouvons utiliser le nom d'une fonction en argument de `map` plutôt qu'une fermeture, comme ceci :

```
let liste_de_nombres = vec![1, 2, 3];
let liste_de_chaines: Vec<String> =
    liste_de_nombres.iter().map(ToString::to_string).collect();
```

Notez que nous devons utiliser la syntaxe complète que nous avons vue précédemment dans la section précédente car il existe plusieurs fonctions disponibles qui s'appellent `to_string` . Ici, nous utilisons la fonction `to_string` définie dans le trait `ToString` que la bibliothèque standard a implémenté sur chaque type qui implémente `Display` .

Rappelez-vous qu'à la section “Les valeurs d'énumérations” du chapitre 6, nous apprenions que le nom de chaque variante d'énumération que nous déclarons devient aussi une fonction d'initialisation. Nous pouvons utiliser ces fonctions d'initialisation en tant que pointeurs de fonctions qui implémentent les traits de fermetures, ce qui signifie que nous pouvons utiliser les fonctions d'initialisation comme paramètre des méthodes qui acceptent des fermetures, comme ceci :

```
    enum Statut {
        Valeur(u32),
        Stop,
    }

    let liste_de_statuts: Vec<Statut> =
(0u32..20).map(Statut::Valeur).collect();
```

Nous avons ici créé des instances de `Statut::Valeur` en utilisant chacune des valeurs `u32` présentes dans l'intervalle sur laquelle nous appelons `map` en utilisant la fonction d'initialisation de `Statut::Valeur`. Certaines personnes préfèrent ce style, et d'autres préfèrent utiliser des fermetures. Ces deux approches se compilent et produisent le même code, vous pouvez donc utiliser le style qui est le plus clair pour vous.

Retourner des fermetures

Les fermetures sont représentées par des traits, ce qui signifie que vous ne pouvez pas retourner directement des fermetures. Dans la plupart des situations où vous auriez voulu retourner un trait, vous pouvez utiliser à la place le type concret qui implémente le trait comme valeur de retour de la fonction. Mais vous ne pouvez pas faire ceci avec les fermetures car elles n'ont pas de type concret qu'elles peuvent retourner ; vous n'êtes pas autorisé à utiliser le pointeur de fonction `fn` comme type de retour, par exemple.

Le code suivant essaye de retourner directement une fermeture, mais ne peut pas se compiler :

```
fn retourne_une_fermeture() -> dyn Fn(i32) -> i32 {
    |x| x + 1
}
```

Voici l'erreur de compilation :

```
$ cargo build
   Compiling functions-example v0.1.0 (file:///projects/functions-example)
error[E0746]: return type cannot have an unboxed trait object
 --> src/lib.rs:1:25
  |
1 | fn retourne_une_fermeture() -> dyn Fn(i32) -> i32 {
  |                                ^^^^^^^^^^^^^^^^^^ doesn't have a size known
at compile-time
  |
  = note: for information on `impl Trait`, see <https://doc.rust-lang.org/book/
ch10-02-traits.html#returning-types-that-implement-traits>
help: use `impl Fn(i32) -> i32` as the return type, as all return paths are of
type `[closure@src/lib.rs:2:5: 2:14]`, which implements `Fn(i32) -> i32`
  |
1 | fn retourne_une_fermeture() -> impl Fn(i32) -> i32 {
  |                                ~~~~~~~~~~~~~~~~~~

For more information about this error, try `rustc --explain E0746`.
error: could not compile `functions-example` due to previous error
```

Une nouvelle fois l'erreur du trait `Sized` ! Rust ne sait pas combien de mémoire sera nécessaire pour stocker la fermeture. Nous avons vu une solution à ce problème précédemment. Nous pouvons utiliser un objet trait :

```
fn retourne_une_fermeture() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

Ce code va se compiler à merveille. Pour en savoir plus sur les objets trait, rendez-vous à la section du chapitre 17.

Maintenant, penchons-nous sur les macros !

Les macros

Nous avons déjà utilisé des macros tout au long de ce livre, comme `println!`, mais nous n'avons pas examiné en profondeur ce qu'est une macro et comment elles fonctionnent. Le terme *macro* renvoie à une famille de fonctionnalités de Rust : les macros *déclaratives* avec `macro_rules!` et trois types de macros *procédurales* :

- Des macros `#[derive]` personnalisées qui renseigne du code ajouté grâce à l'attribut `derive` utilisé sur les structures et les énumérations
- Les macros qui ressemblent à des attributs qui définissent des attributs personnalisés qui sont utilisables sur n'importe quel élément
- Les macros qui ressemblent à des fonctions mais qui opèrent sur les éléments renseignés en argument

Nous allons voir chacune d'entre elles à leur tour, mais avant, posons-nous la question de pourquoi nous avons besoin de macros alors que nous avons déjà les fonctions.

La différence entre les macros et les fonctions

Essentiellement, les macros sont une façon d'écrire du code qui écrit un autre code, ce qui s'appelle la *métaprogrammation*. Dans l'annexe C, nous verrons l'attribut `derive`, qui génère une implémentation de différents traits pour vous. Nous avons aussi utilisé les macros `println!` et `vec!` dans ce livre. Toutes ces macros *se déploient* pour produire plus de code que celui que vous avez écrit manuellement.

La métaprogrammation est utile pour réduire la quantité de code que vous avez à écrire et à maintenir, ce qui est aussi un des rôles des fonctions. Cependant, les macros ont quelques pouvoirs en plus que les fonctions n'ont pas.

La signature d'une fonction doit déclarer le nombre et le type de paramètres qu'a cette fonction. Les macros, à l'inverse, peuvent prendre un nombre variable de paramètres : nous pouvons appeler `println!("salut")` avec un seul paramètre, ou `println!("salut {}", nom)` avec deux paramètres. De plus, les macros sont déployées avant que le compilateur n'interprète la signification du code, donc une macro peut, par exemple, implémenter un trait sur un type donné. Une fonction ne peut pas le faire, car elle est exécutée à l'exécution et un trait doit être implémenté à la compilation.

Le désavantage d'implémenter une macro par rapport à une fonction est que les définitions de macros sont plus complexes que les définitions de fonction car vous écrivez du code Rust qui écrit lui-même du code Rust. A cause de cette approche, les définitions de macro sont généralement plus difficiles à lire, à comprendre et à maintenir que les définitions de

fonctions.

Une autre différence importante entre les macros et les fonctions est que vous devez définir les macros ou les importer dans la portée *avant* de les utiliser dans le fichier, contrairement aux fonctions que vous pouvez définir n'importe où et y faire appel n'importe où.

Les macros déclaratives avec `macro_rules!` pour la métaprogrammation générale

La forme la plus utilisée de macro en Rust est la *macro déclarative*. Elles sont parfois appelées “macros définies par un exemple”, “macros `macro_rules!` ” ou simplement “macros”. Fondamentalement, les macros déclaratives vous permettent d'écrire quelque chose de similaire à une expression `match` de Rust. Comme nous l'avons vu au chapitre 6, les expressions `match` sont des structures de contrôle qui prennent en argument une expression, comparent la valeur qui en résulte avec les motifs et ensuite exécutent le code associé au motif qui correspond. Les macros comparent elles aussi une valeur avec des motifs qui sont associés à code particulier : dans cette situation, la valeur est littéralement le code source Rust envoyé à la macro ; les motifs sont comparés avec la structure de ce code source ; et le code associé à chaque motif vient remplacer le code passé à la macro, lorsqu'il correspond. Tout ceci se passe lors de la compilation.

Pour définir une macro, il faut utiliser la construction `macro_rules!` . Explorons l'utilisation de `macro_rules!` en observant comment la macro `vec!` est définie. Le chapitre 8 nous a permis de comprendre comment utiliser la macro `vec!` pour créer un nouveau vecteur avec des valeurs précises. Par exemple, la macro suivante crée un nouveau vecteur qui contient trois entiers :

```
let v: Vec<u32> = vec![1, 2, 3];
```

Nous aurions pu aussi utiliser la macro `vec!` pour créer un vecteur de deux entiers ou un vecteur de cinq slices de chaînes de caractères. Nous n'aurions pas pu utiliser une fonction pour faire la même chose car nous n'aurions pas pu connaître le nombre ou le type des valeurs au départ.

L'encart 19-28 montre une définition légèrement simplifiée de la macro `vec!` .

Fichier : src/lib.rs

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Encart 19-28 : une version simplifiée de la définition de la macro `vec!`

> Remarque : la définition actuelle de la macro `vec!` de la bibliothèque standard embarque du code pour pré-allouer la bonne quantité de mémoire en amont. Ce code est une optimisation que nous n'allons pas intégrer ici pour simplifier l'exemple.

L'annotation `#[macro_export]` indique que cette macro doit être disponible à chaque fois que la crate dans laquelle la macro est définie est importée dans la portée. Sans cette annotation, la macro ne pourrait pas être importée dans la portée.

Ensuite, nous commençons la définition de la macro avec `macro_rules!` suivi du nom de la macro que nous définissons *sans* le point d'exclamation. Le nom, qui dans ce cas est `vec`, est suivi par des accolades indiquant le corps de la définition de la macro.

La structure dans le corps de `vec!` ressemble à la structure d'une expression `match`. Ici nous avons une branche avec le motif `( $( $x:expr ), * )`, suivie par `=>` et le code du bloc associé à ce motif. Si le motif correspond, le bloc de code associé sera déployé. Etant donné que c'est le seul motif dans cette macro, il n'y a qu'une seule bonne façon d'y correspondre ; tout autre motif va déboucher sur une erreur. Des macros plus complexes auront plus qu'une seule branche.

La syntaxe correcte pour un motif dans les définitions de macros est différente de la syntaxe de motif que nous avons vue au chapitre 18 car les motifs de macros sont comparés à des structures de code Rust plutôt qu'à des valeurs. Examinons la signification des éléments du motif de l'encart 19-28 ; pour voir l'intégralité de la syntaxe du motif de la macro, référez-vous à la documentation.

Premièrement, un jeu de parenthèses englobent l'intégralité du motif. Ensuite vient le symbole dollar (`$`), suivi par un jeu de parenthèses qui capturent les valeurs qui correspondent au motif entre les parenthèses pour les utiliser dans le code de

remplacement. A l'intérieur du `$()` nous avons `$x:expr`, qui correspond à n'importe quelle expression Rust et donne le nom `$x` à l'expression.

La virgule qui suit le `$()` signifie que cette virgule littérale comme caractère littéral de séparation peut optionnellement apparaître après le code qui correspond au code du `$()`. Le `*` informe que ce motif correspond à zéro ou plus éléments répétés correspondant à ce qui précède ce `*`.

Lorsque nous faisons appel à cette macro avec `vec![1, 2, 3];`, le motif `$x` correspond à trois reprises avec les trois expressions `1`, `2`, et `3`.

Maintenant, penchons-nous sur le motif dans le corps du code associé à cette branche : `temp_vec.push()` dans le `$()*` est généré pour chacune des parties qui correspondent au `$()` dans le motif pour zéro ou plus de fois, en fonction de combien de fois le motif correspond. Le `$x` est remplacé par chaque expression qui correspond. Lorsque nous faisons appel à cette macro avec `vec![1, 2, 3];`, le code généré qui remplace cet appel de macro ressemblera à ceci :

```
{
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
}
```

Nous avons défini une macro qui peut prendre n'importe quel nombre d'arguments de n'importe quel type et qui peut générer du code pour créer un vecteur qui contient les éléments renseignés.

Il subsiste quelques cas limites étranges avec `macro_rules!`. Bientôt, Rust rajoutera un second type de macro déclarative qui fonctionnera de la même manière mais qui corrigera ces cas limites. Après cette mise à jour, `macro_rules!` sera dépréciée. En sachant cela, ainsi que le fait que la plupart des développeurs Rust vont davantage *utiliser* les macros qu'en *écrire*, nous arrêtons là la discussion sur `macro_rules!`. Pour en apprendre plus sur l'écriture des macros, consultez la documentation en ligne ou d'autres ressources comme "The Little Book of Rust Macros", débuté par Daniel Keep et continué par Lukas Wirth.

Les macros procédurales pour générer du code à partir des attributs

La seconde forme de macro est la *macro procédurale*, qui se comporte davantage comme une fonction (et est un type de procédure). Les macros procédurales prennent du code en

entrée, travaillent sur ce code et produisent du code en sortie plutôt que de faire des correspondances sur des motifs et remplacer du code avec un autre code, comme le font les macros déclaratives.

Les trois types de macros procédurales (les dérivées personnalisées, celles qui ressemblent aux attributs, et celles qui ressemblent à des fonctions) fonctionnent toutes de la même manière.

Lorsque vous créez une macro procédurale, les définitions doivent être rangées dans leur propre crate avec un type spécial de crate. Ceci pour des raisons techniques complexes que nous espérons supprimer dans l'avenir. La déclaration des macros procédurales ressemble au code de l'encart 19-29, dans lequel `un_attribut_quelconque` est un emplacement pour l'utilisation d'une macro spécifique.

Fichier : src/lib.rs

```
use proc_macro;

#[un_attribut_quelconque]
pub fn un_nom_quelconque(entree: TokenStream) -> TokenStream {
}
```

Encart 19-29 : un exemple de déclaration d'une macro procédurale

La fonction qui définit une macro procédurale prend un `TokenStream` en entrée et produit un `TokenStream` en sortie. Le type `TokenStream` est défini par la crate `proc_macro` qui est fournie par Rust et représente une séquence de jetons. C'est le cœur de la macro : le code source sur lequel la macro opère compose l'entrée `TokenStream`, et le code que la macro produit est la sortie `TokenStream`. La fonction a aussi un attribut qui lui est rattaché et qui indique quel genre de macro procédurale nous créons. Nous pouvons avoir différents types de macros procédurales dans la même crate.

Voyons maintenant les différents types de macros procédurales. Nous allons commencer par une macro dérivée personnalisée et nous expliquerons ensuite les petites différences avec les autres types.

Comment écrire une macro dérivée personnalisée

Créons une crate `hello_macro` qui définit un trait qui s'appelle `HelloMacro` avec une fonction associée `hello_macro`. Plutôt que de contraindre les utilisateurs de notre crate à implémenter le trait `HelloMacro` sur chacun de leurs types, nous allons fournir une macro procédurale qui permettra aux utilisateurs de pouvoir annoter leur type avec

`#[derive(HelloMacro)]` afin d'obtenir une implémentation par défaut de la fonction `hello_macro`. L'implémentation par défaut affichera `Hello, Macro ! Mon nom est TypeName !`, dans lequel `TypeName` est le nom du type sur lequel ce trait a été défini. Autrement dit, nous allons écrire une crate qui permet à un autre développeur d'écrire du code comme l'encart 19-30 en utilisant notre crate.

Fichier : src/main.rs

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

Encart 19-30 : le code qu'un utilisateur de notre crate pourra écrire lorsqu'il utilisera notre macro procédurale

Ce code va afficher `Hello, Macro ! Mon nom est Pancakes !` lorsque vous en aurez fini. La première étape consiste à créer une nouvelle crate de bibliothèque, comme ceci :

```
$ cargo new hello_macro --lib
```

Ensuite, nous allons définir le trait `HelloMacro` et sa fonction associée :

Fichier : src/lib.rs

```
pub trait HelloMacro {
    fn hello_macro();
}
```

Nous avons maintenant un trait et sa fonction. A partir de là, notre utilisateur de la crate peut implémenter le trait pour accomplir la fonctionnalité souhaitée, comme ceci :

```
use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro ! Mon nom est Pancakes !");
    }
}

fn main() {
    Pancakes::hello_macro();
}
```

Cependant, l'utilisateur doit écrire le bloc d'implémentation pour chacun des types qu'il souhaite utiliser avec `hello_macro` ; nous souhaitons lui épargner ce travail.

De plus, nous ne pouvons pas encore fournir la fonction `hello_macro` avec l'implémentation par défaut qui va afficher le nom du type du trait sur lequel nous l'implémentons : Rust n'est pas réflexif, donc il ne peut pas connaître le nom du type à l'exécution. Nous avons besoin d'une macro pour générer le code à la compilation.

La prochaine étape consiste à définir la macro procédurale. Au moment de l'écriture de ces lignes, les macros procédurales ont besoin d'être placées dans leur propre crate. Cette restriction sera levée plus tard. La convention pour structurer les crates et les crates de macros est la suivante : pour une crate `foo`, une crate de macro procédurale personnalisée de dérivée doit s'appeler `foo_derive`. Créons une nouvelle crate `hello_macro_derive` au sein de notre projet `hello_macro` :

```
$ cargo new hello_macro_derive --lib
```

Nos deux crates sont étroitement liées, donc nous créons la crate de macro procédurale à l'intérieur du dossier de notre crate `hello_macro`. Si nous changeons la définition du trait dans `hello_macro`, nous aurons aussi à changer l'implémentation de la macro procédurale dans `hello_macro_derive`. Les deux crates vont devoir être publiées séparément, et les développeurs qui vont utiliser ces crates vont avoir besoin d'ajouter les deux dépendances et les importer dans la portée. Nous pourrions plutôt faire en sorte que la crate `hello_macro` utilise `hello_macro_derive` comme dépendance et ré-exporter le code de la macro procédurale. Cependant, la façon dont nous avons structuré le projet donne la possibilité aux développeurs d'utiliser `hello_macro` même s'ils ne veulent pas la fonctionnalité `derive`.

Nous devons déclarer la crate `hello_macro_derive` comme étant une crate de macro procédurale. Nous allons aussi avoir besoin des fonctionnalités des crates `syn` et `quote`,

comme vous allez le constater bientôt, donc nous allons les ajouter comme dépendances. Ajoutez ceci dans le fichier *Cargo.toml* de `hello_macro_derive` :

Fichier : hello_macro_derive/Cargo.toml

```
[lib]
proc-macro = true

[dependencies]
syn = "1.0"
quote = "1.0"
```

Pour commencer à définir la macro procédurale, placez le code de l'encart 19-31 dans votre fichier *src/lib.rs* de la crate `hello_macro_derive`. Notez que ce code ne se compilera pas tant que nous n'ajouterons pas une définition pour la fonction `impl_hello_macro`.

Fichier : hello_macro_derive/src/lib.rs

```
use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construit une représentation du code Rust en arborescence
    // syntaxique que nous pouvons manipuler
    let ast = syn::parse(input).unwrap();

    // Construit l'implémentation du trait
    impl_hello_macro(&ast)
}
```

Encart 19-31 : du code dont la plupart des macros procédurales auront besoin pour travailler avec du code Rust

Remarquez que nous avons séparé le code de la fonction `hello_macro_derive`, qui est responsable de parcourir le `TokenStream`, de celui de la fonction `impl_hello_macro`, qui est responsable de transformer l'arborescence syntaxique : cela facilite l'écriture de la macro procédurale. Le code dans la fonction englobante (qui est `hello_macro_derive` dans notre cas) sera le même pour presque toutes les crates de macro procédurales que vous allez voir ou créer. Le code que vous renseignez dans le corps de la fonction (qui est `impl_hello_macro` dans notre cas) diffèrera en fonction de ce que fait votre macro procédurale.

Nous avons ajouté trois nouvelles crates : `proc_macro`, `syn` et `quote`. La crate `proc_macro` est fournie par Rust, donc nous n'avons pas besoin de l'ajouter aux dépendances dans

Cargo.toml. La crate `proc_macro` fournit une API du compilateur qui nous permet de lire et manipuler le code Rust à partir de notre code.

La crate `syn` transforme le code Rust d'une chaîne de caractères en une structure de données sur laquelle nous pouvons procéder à des opérations. La crate `quote` re-transforme les structures de données de `syn` en code Rust. Ces crates facilite le parcours de toute sorte de code Rust que nous aurions besoin de gérer : l'écriture d'un interpréteur complet de code Rust n'a jamais été aussi facile.

La fonction `hello_macro_derive` va être appelée lorsqu'un utilisateur de notre bibliothèque utilisera `#[derive(HelloMacro)]` sur un type. Cela sera possible car nous avons annoté notre fonction `hello_macro_derive` avec `proc_macro_derive` et nous avons indiqué le nom, `HelloMacro`, qui correspond au nom de notre trait ; c'est la convention que la plupart des macros procédurales suivent.

La fonction `hello_macro_derive` commence par convertir le `input` qui est un `TokenStream` en une structure de données que nous pouvons ensuite interpréter et sur laquelle faire des opérations. C'est là que `syn` entre en jeu. La fonction `parse` de `syn` prend un `TokenStream` et retourne une structure `DeriveInput` qui représente le code Rust. L'encart 19-32 montre les parties intéressantes de la structure `DeriveInput` que nous obtenons en convertissant la chaîne de caractères `struct Pancakes;` :

```
DeriveInput {
    // -- partie masquée ici --

    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

Encart 19-32 : l'instance de `DeriveInput` que nous obtenons lorsque nous analysons le code qui est décoré par l'attribut de la macro dans l'encart 19-30

Les champs de cette structure montrent que ce code Rust que nous avons converti est une structure unitaire avec l' `ident` (raccourci de `identifier`, qui désigne le nom) `Pancakes`. Il y

a d'autres champs sur cette structure décrivant toutes sortes de codes Rust ; regardez la documentation de `syn` pour `DeriveInput` pour en savoir plus.

Bientôt, nous définirons la fonction `impl_hello_macro`, qui nous permettra de construire le nouveau code Rust que nous souhaitons injecter. Mais avant de faire cela, remarquez que la sortie de notre macro `derive` est aussi un `TokenStream`. Le `TokenStream` retourné est ajouté au code que les utilisateurs de notre crate ont écrit, donc lorsqu'ils compilent leur crate, ils récupéreront la fonctionnalité additionnelle que nous injectons dans le `TokenStream` modifié.

Vous avez peut-être remarqué que nous faisons appel à `unwrap` pour faire paniquer la fonction `hello_macro_derive` si l'appel à la fonction `syn::parse` que nous faisons échoue. Il est nécessaire de faire paniquer notre macro procédurale si elle rencontre des erreurs car les fonctions `proc_macro_derive` doivent retourner un `TokenStream` plutôt qu'un `Result` pour se conformer à l'API de la macro procédurale. Nous avons simplifié cet exemple en utilisant `unwrap` ; dans du code en production, vous devriez renseigner des messages d'erreur plus précis sur ce qui s'est mal passé en utilisant `panic!` ou `expect`.

Maintenant que nous avons le code pour transformer le code Rust annoté d'un `TokenStream` en une instance de `DeriveInput`, créons le code qui implémente le trait `HelloMacro` sur le type annoté, comme montré dans l'encart 19-33.

Fichier : hello_macro_derive/src/lib.rs

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let nom = &ast.ident;
    let generation = quote! {
        impl HelloMacro for #nom {
            fn hello_macro() {
                println!("Hello, Macro ! Mon nom est {}", stringify!(#nom));
            }
        }
    };
    generation.into()
}
```

Encart 19-33 : implémentation du trait `HelloMacro` en utilisant le code Rust interprété

Nous obtenons une instance de structure `Ident` qui contient le nom (`identifier`) du type annoté en utilisant `ast.ident`. La structure de l'encart 19-32 montre que lorsque nous exécutons la fonction `impl_hello_macro` sur le code de l'encart 19-30, le `ident` que nous obtenons aura le champ `ident` avec la valeur `"Pancakes"`. Ainsi, la variable `nom` de l'encart 19-33 contiendra une instance de la structure `Ident` qui, une fois affichée, sera la chaîne de caractères `"Pancakes"`, le nom de la structure de l'encart 19-30.

La macro `quote!` nous permet de définir le code Rust que nous souhaitons retourner. Le compilateur attend quelque chose de différent que le résultat direct produit par l'exécution de `quote!`, donc nous devons convertir ce dernier en `TokenStream`. Nous faisons ceci en faisant appel à la méthode `into`, qui utilise cette représentation intermédiaire et retourne une valeur du type attendu, le type `TokenStream` ici.

La macro `quote!` fournit aussi quelques mécaniques de gabarit intéressantes : nous pouvons entrer `#nom`, et `quote!` va le remplacer avec la valeur présente dans la variable `nom`. Vous pouvez même exécuter des répétitions d'une façon similaire à celle des macros classiques. Regardez dans la documentation de `quote` pour une présentation plus détaillée.

Nous souhaitons que notre macro procédurale génère une implémentation de notre trait `HelloMacro` pour le type que l'utilisateur a annoté, que nous pouvons obtenir en utilisant `#nom`. L'implémentation du trait utilise une fonction, `hello_macro`, dont le corps contient la fonctionnalité que nous souhaitons fournir : l'affichage de `Hello, Macro ! Mon nom est` suivi par le nom du type annoté.

La macro `stringify!` utilisée ici est écrite en Rust. Elle prend en argument une expression Rust, comme `1 + 2`, et à la compilation transforme l'expression en une chaîne de caractères littérale, comme `"1 + 2"`. Cela est différent de `format!` ou de `println!`, des macros qui évaluent l'expression et retourne ensuite le résultat dans une `String`. Il est possible que l'entrée `#nom` soit une expression à écrire littéralement, donc nous utilisons `stringify!`. L'utilisation de `stringify!` évite aussi une allocation en convertissant `#nom` en une chaine de caractères littérale à la compilation.

Maintenant, `cargo build` devrait fonctionner correctement pour `hello_macro` et `hello_macro_derive`. Relions maintenant ces crates au code de l'encart 19-30 pour voir les macros procédurales à l'oeuvre ! Créez un nouveau projet binaire dans votre dossier *projects* en utilisant `cargo new pancakes`. Nous avons besoin d'ajouter `hello_macro` et `hello_macro_derive` comme dépendances dans le *Cargo.toml* de la crate `pancakes`. Si vous publiez vos versions de `hello_macro` et de `hello_macro_derive` sur crates.io, ce seront des dépendances classiques ; sinon, vous pouvez les indiquer en tant que dépendances locales avec `path` comme ci-après :

```
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

Renseignez le code l'encart 19-30 dans *src/main.rs*, puis lancez `cargo run` : cela devrait afficher `Hello, Macro ! Mon nom est Pancakes !`. L'implémentation du trait `HelloMacro` à l'aide de la macro procédurale a été incluse sans que la crate `pancakes` n'ait eu besoin de l'implémenter ; le `#[derive(HelloMacro)]` a ajouté automatiquement l'implémentation du

trait.

Maintenant, découvrons comment les autres types de macros procédurales se distinguent des macros `derive` personnalisées.

Les macros qui ressemblent à des attributs

Les macros qui ressemblent à des attributs ressemblent aux macros `derive` personnalisées, mais au lieu de générer du code pour l'attribut `derive`, elles vous permettent de créer des nouveaux attributs. Elles sont aussi plus flexibles : `derive` fonctionne uniquement pour les structures et les énumérations ; les attributs peuvent être aussi appliqués aux autres éléments, comme les fonctions. Voici un exemple d'utilisation d'une macro qui ressemble à un attribut : imaginons que vous avez un attribut `chemin` qui est une annotation pour des fonctions lorsque vous utilisez un environnement de développement d'application web :

```
#[chemin(GET, "/")]
fn index() {
```

Cet attribut `#[chemin]` sera défini par l'environnement de développement comme étant une macro procédurale. La signature de la fonction de définition de la macro ressemblera à ceci :

```
#[proc_macro_attribute]
pub fn chemin(attribut: TokenStream, element: TokenStream) -> TokenStream {
```

Maintenant, nous avons deux paramètres de type `TokenStream`. Le premier correspond au contenu de l'attribut : la partie `GET, "/"`. Le second est le corps de l'élément sur lequel cet attribut sera appliqué : dans notre cas, `fn index() {}` et le reste du corps de la fonction.

Mis à part cela, les macros qui ressemblent à des attributs fonctionnent de la même manière que les macros `derive` personnalisées : vous générez une crate avec le type de la crate `proc-macro` et vous implémentez une fonction qui génèrera le code que vous souhaitez !

Les macros qui ressemblent à des fonctions

Les macros qui ressemblent à des fonctions définissent des macros qui ressemblent à des appels de fonction. De la même manière que les macros `macro_rules!`, elles sont plus flexibles que les fonctions ; par exemple, elles peuvent prendre une quantité non finie

d'arguments. Cependant, les macros `macro_rules!` peuvent être définies uniquement en utilisant la syntaxe qui ressemble à `match` et que nous avons vue dans une section précédente. Les macros qui ressemblent à des fonctions prennent en paramètre un `TokenStream` et leurs définitions manipulent ce `TokenStream` en utilisant du code Rust comme le font les deux autres types de macros procédurales. Voici un exemple d'une macro qui ressemble à une fonction qui est une macro `sql!` qui devrait être utilisée comme ceci :

```
let sql = sql!(SELECT * FROM publications WHERE id=1);
```

Cette macro devrait interpréter l'instruction SQL qu'on lui envoie et vérifier si elle est syntaxiquement correcte, ce qui est un procédé bien plus complexe que ce qu'une macro `macro_rules!` peut faire. La macro `sql!` sera définie comme ceci :

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
```

Cette définition ressemble à la signature de la macro `derive` personnalisée : nous récupérons les éléments entre parenthèses et retournons le code que nous souhaitons générer.

Résumé

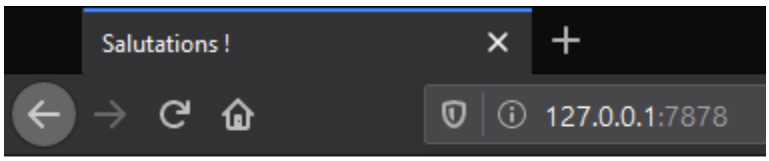
Ouah ! Maintenant vous avez quelques fonctionnalités de Rust supplémentaires dans votre boite à outils que vous n'utiliserez probablement que rarement, mais vous savez maintenant qu'elles pourront vous aider dans certaines situations très particulières. Nous avons introduits plusieurs sujets complexes afin que vous puissiez les reconnaître, ainsi que la syntaxe associée, lorsque vous les rencontrerez dans des messages de suggestions dans des erreurs ou dans le code de quelqu'un d'autre. Utilisez ce chapitre comme référence pour vous guider vers ces solutions.

Au chapitre suivant, nous allons mettre en pratique tout ce que nous avons appris dans ce livre en l'appliquant à un nouveau projet !

Projet final : construire un serveur web multitâches

Ce fut un long voyage, mais nous avons atteint la fin de ce livre. Dans ce chapitre, nous allons construire un nouveau projet ensemble pour mettre en application certains concepts que nous avons vus dans les derniers chapitres, et aussi pour récapituler quelques leçons précédentes.

Pour notre projet final, nous allons construire un serveur web qui dit “salutations” et qui ressemble dans un navigateur web à l'illustration 20-1.



Salut !

Bonjour de la part de Rust

Illustration 20-1 : notre dernier projet en commun

Voici le plan de construction du serveur web :

1. En savoir plus sur TCP et HTTP.
2. Ecouter les connections TCP sur un port.
3. Interpréter une petite quantité de requêtes HTTP.
4. Créer une réponse HTTP adéquate.
5. Augmenter le débit de notre serveur avec un groupe de tâches.

Mais avant de commencer, nous devons signaler une chose : les méthodes que nous allons utiliser ne sont pas les meilleures pour construire un serveur web avec Rust. Un certain nombre de crates éprouvées en production et disponibles sur crates.io fourniront des serveurs web et des implémentations de groupe de tâches plus complets que ce que nous allons construire.

Toutefois, notre intention dans ce chapitre est de vous aider à apprendre, et ne pas de se laisser aller à la facilité. Comme Rust est un langage de programmation système, nous pouvons choisir le niveau d'abstraction avec lequel nous souhaitons travailler et nous pouvons descendre à un niveau plus bas que ce qui est possible ou pratique dans d'autres langages. Nous allons écrire manuellement le serveur HTTP basique et le groupe de tâches

afin que vous puissiez apprendre les idées et techniques générales qui se cachent derrière les crates que vous serez peut-être amenés à utiliser à l'avenir.

Développer un serveur web monotâche

Nous allons commencer par faire fonctionner un serveur web monotâche. Avant de commencer, faisons un survol rapide des protocoles utilisés dans les serveurs web. Les détails de ces protocoles ne sont pas le sujet de ce livre, mais un rapide aperçu vous donnera les informations dont vous avez besoin.

Les deux principaux protocoles utilisés dans les serveurs web sont le *Hypertext Transfer Protocol (HTTP)* et le *Transmission Control Protocol (TCP)*. Ces deux protocoles sont des protocoles de type *requête-réponse*, ce qui signifie qu'un *client* initie des requêtes tandis que le *serveur* écoute les requêtes et fournit une réponse au client. Le contenu de ces requêtes et de ces réponses est défini par les protocoles.

TCP est le protocole le plus bas-niveau qui décrit les détails de comment une information passe d'un serveur à un autre mais ne précise pas ce qu'est cette information. HTTP est construit sur TCP en définissant le contenu des requêtes et des réponses. Il est techniquement possible d'utiliser HTTP avec d'autres protocoles, mais dans la grande majorité des cas, HTTP envoie ses données via TCP. Nous allons travailler avec les octets bruts des requêtes et des réponses de TCP et HTTP.

Ecouter les connexions TCP

Notre serveur web a besoin d'écouter les connexions TCP, donc cela sera la première partie sur laquelle nous travaillerons. La bibliothèque standard offre un module `std::net` qui nous permet de faire ceci. Créons un nouveau projet de manière habituelle :

```
$ cargo new salutations
     Created binary (application) `salutations` project
$ cd salutations
```

Maintenant, saisissez le code de l'encart 20-1 dans *src/main.rs* pour commencer. Ce code va écouter les flux TCP entrants à l'adresse `127.0.0.1:7878`. Lorsqu'il obtiendra un flux entrant, il va afficher `Connexion établie !`.

Fichier : src/main.rs

```
use std::net::TcpListener;

fn main() {
    let ecouteur = TcpListener::bind("127.0.0.1:7878").unwrap();

    for flux in ecouteur.incoming() {
        let flux = flux.unwrap();

        println!("Connexion établie !");
    }
}
```

Encart 20-1 : écoute des flux entrants et affichage d'un message lorsque nous recevons un flux

En utilisant `TcpListener`, nous pouvons écouter les connexions TCP à l'adresse `127.0.0.1:7878`. Dans cette adresse, la partie avant les double-points est une adresse IP qui représente votre ordinateur (c'est la même sur chaque ordinateur et ne représente pas spécifiquement l'ordinateur de l'auteur), et `7878` est le port. Nous avons choisi ce port pour deux raisons : HTTP n'est pas habituellement accepté sur ce port et 7878 correspond aux touches utilisées sur un clavier de téléphone pour écrire *Rust*.

La fonction `bind` dans ce scénario fonctionne comme la fonction `new` dans le sens où elle retourne une nouvelle instance de `TcpListener`. La raison pour laquelle cette fonction s'appelle `bind` *(NdT : signifie "lier")* est que dans le domaine des réseaux, se connecter à un port se dit se “lier à un port”.

La fonction `bind` retourne un `Result<T, E>`, ce qui signifie que la création de lien peut échouer. Par exemple, la connexion au port 80 nécessite d'être administrateur (les utilisateurs non-administrateur ne peuvent écouter que sur les ports supérieurs à 1023), donc si nous essayons de connecter un port 80 sans être administrateur, le lien ne va pas fonctionner. Pour donner un autre exemple, le lien ne va pas fonctionner si nous exécutons deux instances de notre programme et que nous avons deux programmes qui écoutent sur le même port. Comme nous écrivons un serveur basique uniquement à but pédagogique, nous n'avons pas à nous soucier de la gestion de ce genre d'erreur ; c'est pourquoi nous utilisons `unwrap` pour arrêter l'exécution du programme si des erreurs surviennent.

La méthode `incoming` d'un `TcpListener` retourne l'itérateur qui nous donne une séquence de flux (plus précisément, des flux de type `TcpStream`). Un seul *flux* représente une connexion entre le client et le serveur. Une *connexion* est le nom qui désigne le processus complet de requête et de réponse, durant lequel le client se connecte au serveur, le serveur génère une réponse puis le serveur ferme la connexion. Ainsi, `TcpStream` va se lire lui-même pour voir ce que le client a envoyé et nous permettre ensuite d'écrire notre réponse dans le flux. De manière générale, cette boucle `for` traitera l'une après l'autre chaque

connexion dans l'ordre et produira une série de flux que nous devrons gérer.

Pour l'instant, notre gestion des flux consiste à appeler `unwrap` pour arrêter notre programme si le flux rencontre une erreur ; s'il n'y a pas d'erreurs, le programme affiche un message. Nous ajouterons davantage de fonctionnalités en cas de succès dans le prochain encart. La raison pour laquelle nous pourrions recevoir des erreurs de la méthode `incoming` lorsqu'un client se connecte au serveur est qu'en réalité nous n'itérons pas sur les connexions. En effet, nous itérons sur des *tentatives de connexion*. La connexion peut échouer pour de nombreuses raisons, beaucoup d'entre elles sont spécifiques au système d'exploitation. Par exemple, de nombreux systèmes d'exploitation ont une limite sur le nombre de connexions ouvertes simultanément qu'ils peuvent supporter ; les tentatives de nouvelles connexions une fois ce nombre dépassé produiront une erreur jusqu'à ce que certaines des connexions soient fermées.

Essayons d'exécuter ce code ! Saisissez `cargo run` dans le terminal et ensuite ouvrez *127.0.0.1:7878* dans un navigateur web. Le navigateur devrait afficher un message d'erreur tel que “La connexion a été réinitialisée”, car le serveur ne renvoie pas de données pour le moment. Mais si vous regardez le terminal, vous devriez voir quelques messages qui se sont affichés lorsque le navigateur s'est connecté au serveur !

```
     Running `target/debug/salutations`
Connexion établie !
Connexion établie !
Connexion établie !
```

Des fois, vous pourriez voir plusieurs messages s'afficher pour une seule requête du navigateur ; la raison à cela est peut-être que le navigateur fait une requête pour la page ainsi que des requêtes pour d'autres ressources, comme l'icone *favicon.ico* qui s'affiche dans l'onglet du navigateur.

Peut-être que le navigateur essaie aussi de se connecter plusieurs fois au serveur car le serveur ne renvoie aucune donnée dans sa réponse. Lorsque `flux` sort de la portée et est nettoyé à la fin de la boucle, la connexion est fermée car cela est implémenté dans le `drop`. Les navigateurs réagissent à ces connexions fermées en ré-essayant, car le problème peut être temporaire. La partie importante est que nous avons obtenu avec succès un manipulateur de connexion TCP !

Pensez à arrêter le programme en appuyant sur ctrl-c lorsque vous avez fini d'exécuter une version donnée du code. Relancez ensuite `cargo run` après avoir appliqué une série de modifications afin d'être sûr que vous exécutez bien la toute dernière version du code.

Lire la requête

Commençons à implémenter la fonctionnalité permettant de lire la requête du navigateur ! Pour séparer les parties où nous obtenons une connexion de celle où nous agissons avec la connexion, nous allons créer une nouvelle fonction pour traiter les connexions. Dans cette nouvelle fonction `gestion_connexion`, nous allons lire des données provenant du flux TCP et les afficher afin que nous puissions voir les données envoyées par le navigateur. Changez le code pour qu'il ressemble à l'encart 20-2.

Fichier : src/main.rs

```
use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;

fn main() {
    let ecouteur = TcpListener::bind("127.0.0.1:7878").unwrap();

    for flux in ecouteur.incoming() {
        let flux = flux.unwrap();

        gestion_connexion(flux);
    }
}

fn gestion_connexion(mut flux: TcpStream) {
    let mut tampon = [0; 1024];

    flux.read(&mut tampon).unwrap();

    println!("Requête : {}", String::from_utf8_lossy(&tampon[..]));
}
```

Encart 20-2 : lecture du `TcpStream` et affichage des données

Nous avons importé `std::io::prelude` dans la portée pour accéder à certains traits qui nous permettent de lire et d'écrire dans le flux. Dans la boucle `for` de la fonction `main`, au lieu d'afficher un message qui dit que nous avons établi une connexion, nous faisons maintenant appel à `gestion_connexion` et nous lui passons le `flux`.

Dans la fonction `gestion_connexion`, nous avons fait en sorte que le paramètre `flux` soit mutable. La raison à cela est que l'instance de `TcpStream` garde en mémoire interne le suivi des données qu'il nous a retournées. Il peut lire plus de données que nous en avons demandées et les conserver pour la prochaine fois que nous en redemanderons. Il doit donc être `mut` car son état interne doit pouvoir changer ; d'habitude, nous n'avons pas besoin que la “lecture” nécessite d'être mutable, mais dans ce cas nous avons besoin du mot-clé `mut`.

Ensuite, nous devons lire les données du flux. Nous faisons cela en deux temps : d'abord, nous déclarons un `tampon` sur la pile pour y stocker les données qui seront lues. Nous avons fait en sorte que le tampon fasse 1024 octets, ce qui est suffisamment grand pour stocker les données d'une requête basique, ce qui est suffisant pour nos besoins dans ce chapitre. Si nous avions voulu gérer des requêtes de taille arbitraire, cette gestion du tampon aurait été plus complexe ; nous allons la garder simpliste pour l'instant. Nous envoyons le tampon dans `flux.read` qui va lire les octets provenant du `TcpStream` et les ajouter dans le tampon.

Ensuite, nous convertissons les octets présents dans le tampon en chaînes de caractères et nous affichons cette chaîne de caractères. La fonction `String::from_utf8_lossy` prend en paramètre un `&[u8]` et le transforme en une `String`. La partie “lossy” du nom indique le comportement de cette fonction lorsqu'elle rencontre une séquence UTF-8 invalide : elle va remplacer la séquence invalide par `�`, le caractère `U+FFFD REPLACEMENT CHARACTER`. Vous devriez voir ces caractères de remplacement à la place des caractères du tampon qui n'ont pas été renseignés par des données de requête.

Essayons ce code ! Démarrez le programme et faites à nouveau une requête dans un navigateur web. Notez que nous obtenons toujours une page d'erreur dans le navigateur web, mais que la sortie de notre programme dans le terminal devrait ressembler à ceci :

```
$ cargo run
   Compiling salutations v0.1.0 (file:///projects/salutations)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/salutations`
Requête : GET / HTTP/1.1
Host: 127.0.0.1:7878
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101
Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
������������������������������������������
```

En fonction de votre navigateur, vous pourriez voir une sortie légèrement différente. Maintenant que nous affichons les données des requêtes, nous pouvons constater pourquoi nous obtenons plusieurs connexions pour un seul chargement de page dans le navigateur web en analysant le chemin après le `Requête : GET`. Si les connexions répétées sont toutes vers */*, nous pouvons constater que le navigateur essaye d'obtenir */* à répétition car il n'obtient pas de réponse de la part de notre programme.

Décomposons les données de cette requête pour comprendre ce que le navigateur demande à notre programme.

Une analyse plus poussée d'une requête HTTP

HTTP est un protocole basé sur du texte, et une requête doit suivre cette forme :

```
Méthode URI-Demandée Version-HTTP CRLF
entêtes CRLF
corps-du-message
```

La première ligne est la *ligne de requête* qui contient les informations sur ce que demande le client. La première partie de la ligne de requête indique la *méthode* utilisée, comme `GET` ou `POST`, qui décrit comment le client fait sa requête. Notre client a utilisé une requête `GET`.

La partie suivante de la ligne de requête est */*, qui indique *l'URI (Uniform Resource Identifier)* que demande le client : une URI est presque, mais pas complètement, la même chose qu'une *URL (Uniform Resource Locator)*. La différence entre les URI et les URL n'est pas très importante pour nous dans ce chapitre, mais la spécification de HTTP utilise le terme URI, donc, ici, nous pouvons simplement lire URL là où URI est écrit.

La dernière partie est la version HTTP que le client utilise, puis la ligne de requête termine avec une *séquence CRLF* (CRLF signifie *Carriage Return, retour chariot*, et *Line Feed, saut de ligne* qui sont des termes qui remontent à l'époque des machines à écrire !). La séquence CRLF peut aussi être écrite `\r\n`, dans laquelle `\r` est un retour chariot et `\n` est un saut de ligne. La séquence CRLF sépare la ligne de requête du reste des données de la requête. Notez toutefois que lorsqu'un CRLF est affiché, nous voyons une nouvelle ligne plutôt qu'un `\r\n`.

D'après la ligne de requête que nous avons reçue après avoir exécuté notre programme précédemment, nous constatons que la méthode est `GET`, `/` est l'URI demandée et `HTTP/1.1` est la version.

Après la ligne de requête, les lignes suivant celle où nous avons `Host:` sont des entêtes. Les requêtes `GET` n'ont pas de corps.

Essayez de faire une requête dans un navigateur différent ou de demander une adresse différente, telle que *127.0.0.1:7878/test*, afin d'observer comment les données de requête changent.

Maintenant que nous savons ce que demande le navigateur, envoyons-lui quelques données !

Ecrire une réponse

Maintenant, nous allons implémenter l'envoi d'une réponse à une requête client. Les réponses suivent le format suivant :

```
Version-HTTP Code-Statut Phrase-De-Raison CRLF
entêtes CRLF
corps-message
```

La première ligne est une *ligne de statut* qui contient la version HTTP utilisée dans la réponse, un code numérique de statut qui résume le résultat de la requête et une phrase de raison qui fournit une description textuelle du code de statut. Après la séquence CRLF viennent tous les entêtes, une autre séquence CRLF et enfin le corps de la réponse.

Voici un exemple de réponse qui utilise HTTP version 1.1, a un code de statut de 200, une phrase de raison à OK, pas d'entêtes, et pas de corps :

```
HTTP/1.1 200 OK\r\n\r\n
```

Le code de statut 200 est la réponse standard de succès. Le texte est une toute petite réponse HTTP de succès. Ecrivons ceci dans le flux de notre réponse à une requête avec succès ! Dans la fonction `gestion_connexion`, enlevez le `println!` qui affiche les données de requête et remplacez-le par le code de l'encart 20-3.

Fichier : src/main.rs

```
fn gestion_connexion(mut flux: TcpStream) {
    let mut tampon = [0; 1024];

    flux.read(&mut tampon).unwrap();

    let reponse = "HTTP/1.1 200 OK\r\n\r\n";

    flux.write(reponse.as_bytes()).unwrap();
    flux.flush().unwrap();
}
```

Encart 20-3 : écriture d'une toute petite réponse HTTP de réussite dans le flux

La première ligne définit la variable `reponse` qui contient les données du message de réussite. Ensuite, nous faisons appel à `as_bytes` sur notre `reponse` pour convertir la chaîne de caractères en octets. La méthode `write` sur le `flux` prend en argument un `&[u8]` et envoie ces octets directement dans la connexion.

Comme l'opération `write` peut échouer, nous utilisons `unwrap` sur toutes les erreurs, comme précédemment. Encore une fois, dans un véritable application, vous devriez gérer les cas d'erreur ici. Enfin, `flush` va attendre et empêcher le programme de continuer à

s'exécuter jusqu'à ce que tous les octets soient écrits dans la connexion ; `TcpStream` contient un tampon interne pour réduire les appels au système d'exploitation concerné.

Avec ces modifications, exécutons à nouveau notre code et lançons une requête dans le navigateur. Nous n'affichons plus les données dans le terminal, donc nous ne voyons plus aucune sortie autre que celle de Cargo. Lorsque vous chargez *127.0.0.1:7878* dans un navigateur web, vous devriez obtenir une page blanche plutôt qu'une erreur. Vous venez de coder en dur une réponse à une requête HTTP !

Retourner du vrai HTML

Implémentons la fonctionnalité permettant de retourner plus qu'une simple page blanche. Créez un nouveau fichier, *hello.html*, à la racine de votre dossier de projet, et pas dans le dossier *src*. Vous pouvez ajouter le HTML que vous souhaitez ; l'encart 20-4 vous montre une possibilité.

Fichier : hello.html

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Salutations !</title>
  </head>
  <body>
    <h1>Salut !</h1>
    <p>Bonjour de la part de Rust</p>
  </body>
</html>
```

Encart 20-4 : un exemple de fichier HTML à retourner dans une réponse

Ceci est un document HTML5 minimal avec des entêtes et un peu de texte. Pour retourner ceci à partir d'un serveur lorsqu'une requête est reçue, nous allons modifier `gestion_connexion` comme proposé dans l'encart 20-5 pour lire le fichier HTML, l'ajouter dans la réponse comme faisant partie de son corps, et l'envoyer.

Fichier : src/main.rs

```
use std::fs;
// -- partie masquée ici --

fn gestion_connexion(mut flux: TcpStream) {
    let mut tampon = [0; 1024];
    flux.read(&mut tampon).unwrap();

    let contenu = fs::read_to_string("hello.html").unwrap();

    let reponse = format!(
        "HTTP/1.1 200 OK\r\nContent-Length: {}\r\n\r\n{}",
        contenu.len(),
        contenu
    );

    flux.write(reponse.as_bytes()).unwrap();
    flux.flush().unwrap();
}
```

Encart 20-5 : envoi du contenu de *hello.html* dans le corps de la réponse

Nous avons ajouté une ligne en haut pour importer le module de système de fichiers de la bibliothèque standard. Le code pour lire le contenu d'un fichier dans une `String` devrait vous être familier ; nous l'avons utilisé dans le chapitre 12 lorsque nous lisions le contenu d'un fichier pour notre projet d'entrée/sortie, dans l'encart 12-4.

Ensuite, nous avons utilisé `format!` pour ajouter le contenu du fichier comme étant le corps de la réponse avec succès. Pour garantir que ce soit une réponse HTTP valide, nous avons ajouté l'entête `Content-Length` qui définit la taille du corps de notre réponse, qui dans ce cas est la taille de `hello.html` .

Exécutez ce code avec `cargo run` et ouvrez *127.0.0.1:7878* dans votre navigateur web ; vous devriez voir le résultat de votre HTML !

Pour le moment, nous ignorons les données de la requête présentes dans `tampon` et nous renvoyons sans conditions le contenu du fichier HTML. Cela signifie que si vous essayez de demander *127.0.0.1:7878/autre-chose* dans votre navigateur web, vous obtiendrez la même réponse HTML. Notre serveur est très limité, et ne correspond pas à ce que font la plupart des serveurs web. Nous souhaitons désormais personnaliser nos réponses en fonction de la requête et ne renvoyer le fichier HTML que pour une requête bien formatée faite à */*.

Valider la requête et répondre de manière sélective

Jusqu'à présent, notre serveur web retourne le HTML du fichier peu importe ce que demande le client. Ajoutons une fonctionnalité pour vérifier que le navigateur demande bien

/ avant de retourner le fichier HTML et retournons une erreur si le navigateur demande autre chose. Pour cela, nous devons modifier `gestion_connexion` comme dans l'encart 20-6. Ce nouveau code compare le contenu de la requête que nous recevons à la requête que nous attendrions pour / et ajoute des blocs `if` et `else` pour traiter les requêtes de manière différenciée.

Fichier : src/main.rs

```
// -- partie masquée ici --

fn gestion_connexion(mut flux: TcpStream) {
    let mut tampon = [0; 1024];
    flux.read(&mut tampon).unwrap();

    let get = b"GET / HTTP/1.1\r\n";

    if tampon.starts_with(get) {
        let contenu = fs::read_to_string("hello.html").unwrap();

        let reponse = format!(
            "HTTP/1.1 200 OK\r\nContent-Length: {}\r\n\r\n{}",
            contenu.len(),
            contenu
        );

        flux.write(reponse.as_bytes()).unwrap();
        flux.flush().unwrap();
    } else {
        // autres requêtes
    }
}
```

Encart 20-6 : détection et gestion des requêtes vers / de manière différenciée des autres requêtes

D'abord, nous codons en dur les données correspondant à la requête / dans la variable `get`. Comme nous lisons des octets bruts provenant du tampon, nous transformons `get` en une chaîne d'octets en ajoutant la syntaxe de chaîne d'octets `b""` au début des données du contenu. Ensuite, nous vérifions que le `tampon` commence par les mêmes octets que ceux présents dans `get`. Si c'est le cas, cela signifie que nous avons reçu une requête vers / correctement formatée, qui est le cas de succès que nous allons gérer dans le bloc `if` qui retourne le contenu de notre fichier HTML.

Si `tampon` ne *commence pas* avec les octets présents dans `get`, cela signifie que nous avons reçu une autre requête. Nous allons bientôt ajouter du code au bloc `else` pour répondre à toutes ces autres requêtes.

Exécutez ce code maintenant et demandez *127.0.0.1:7878* ; vous devriez obtenir le HTML de *hello.html*. Si vous faites n'importe quelle autre requête, comme *127.0.0.1:7878/autre-chose*, vous allez obtenir une erreur de connexion comme celle que vous avez vue lorsque vous exécutiez le code l'encart 20-1 et de l'encart 20-2.

Maintenant ajoutons le code de l'encart 20-7 au bloc `else` pour retourner une réponse avec le code de statut 404, qui signale que le contenu demandé par cette requête n'a pas été trouvé. Nous allons aussi retourner du HTML pour qu'une page s'affiche dans le navigateur, indiquant la réponse à l'utilisateur final.

Fichier : src/main.rs

```
// -- partie masquée ici --
} else {
    let ligne_statut = "HTTP/1.1 404 NOT FOUND";
    let contenu = fs::read_to_string("404.html").unwrap();

    let reponse = format!(
        "{}\r\nContent-Length: {}\r\n\r\n{}",
        ligne_statut,
        contenu.len(),
        contenu
    );

    flux.write(reponse.as_bytes()).unwrap();
    flux.flush().unwrap();
}
```

Encart 20-7 : répondre un code de statut 404 et une page d'erreur lorsqu'autre chose que */* a été demandé

Ici notre réponse possède une ligne de statut avec le code de statut 404 et la phrase de raison `NOT FOUND` . Le corps de la réponse sera le HTML présent dans le fichier *404.html*. Nous aurons besoin de créer un fichier `404.html` au même endroit que *hello.html* pour la page d'erreur; de nouveau, n'hésitez pas à utiliser le HTML que vous souhaitez ou, à défaut, utilisez le HTML d'exemple présent dans l'encart 20-8.

Fichier : 404.html

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Salutations !</title>
  </head>
  <body>
    <h1>Oups !</h1>
    <p>Désolé, je ne connaît pas ce que vous demandez.</p>
  </body>
</html>
```

Encart 20-8 : contenu d'exemple pour la page à renvoyer avec les réponses 404

Une fois ces modifications appliquées, exécutez à nouveau votre serveur. Les requêtes vers *127.0.0.1:7878* devraient retourner le contenu de *hello.html* et toutes les autres requêtes, telle que *127.0.0.1:7878/autre-chose*, devraient retourner le HTML d'erreur présent dans *404.html*.

Un peu de remaniement

Pour l'instant, les blocs `if` et `else` contiennent beaucoup de code répété : ils lisent tous les deux des fichiers et écrivent le contenu de ces fichiers dans le flux. La seule différence entre eux sont la ligne de statut et le nom du fichier. Rendons le code plus concis en isolant ces différences dans des lignes `if` et `else` qui vont assigner les valeurs de la ligne de statut et du nom de fichier à des variables ; nous pourrons ensuite utiliser ces variables sans avoir à nous préoccuper du contexte dans le code qui va lire le fichier et écrire la réponse. L'encart 20-9 montre le code résultant après remplacement des gros blocs `if` et `else` .

Fichier : src/main.rs

```
// -- partie masquée ici--

fn gestion_connexion(mut flux: TcpStream) {
    // -- partie masquée ici--

    let (ligne_statut, nom_fichier) = if tampon.starts_with(get) {
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contenu = fs::read_to_string(nom_fichier).unwrap();

    let reponse = format!(
        "{}\r\nContent-Length: {}\r\n\r\n{}",
        ligne_statut,
        contenu.len(),
        contenu
    );

    flux.write(reponse.as_bytes()).unwrap();
    flux.flush().unwrap();
}
```

Encart 20-9 : remaniement des blocs `if` et `else` pour qu'ils contiennent uniquement le code qui différencie les deux cas

Maintenant que les blocs `if` et `else` retournent uniquement les valeurs correctes pour la ligne de statut et le nom du fichier dans un tuple, nous pouvons utiliser la déstructuration pour assigner ces deux valeurs à `ligne_statut` et `nom_fichier` en utilisant un motif dans l'instruction `let` , comme nous l'avons vu dans le chapitre 18.

Le code précédent qui était en double se trouve maintenant à l'extérieur des blocs `if` et `else` et utilise les variables `ligne_statut` et `nom_fichier` . Cela permet de mettre en évidence plus facilement les différences entre les deux cas, et cela signifie que nous n'avons qu'un seul endroit du code à modifier si nous souhaitons changer le fonctionnement de lecture du fichier et d'écriture de la réponse. Le comportement du code de l'encart 20-9 devrait être identique à celui de l'encart 20-8.

Super ! Nous avons maintenant un serveur web simple qui tient dans environ 40 lignes de code, qui répond à une requête précise par une page de contenu et répond à toutes les autres avec une réponse 404.

Actuellement, notre serveur fonctionne dans une seule tâche, ce qui signifie qu'il ne peut répondre qu'à une seule requête à la fois. Examinons maintenant à quel point cela peut être un problème en simulant des réponses lentes à des requêtes. Ensuite, nous corrigerons notre serveur pour qu'il puisse gérer plusieurs requêtes à la fois.

Transformer notre serveur monotâche en serveur multitâches

Pour le moment, le serveur va traiter chaque requête l'une après l'autre, ce qui signifie qu'il ne traitera pas une deuxième connexion tant que la première n'a pas fini d'être traitée. Si le serveur reçoit encore plus de requêtes, cette exécution en série sera de moins en moins adaptée. Si le serveur reçoit une requête qui prend longtemps à traiter, les demandes suivantes devront attendre que la longue requête à traiter soit terminée, même si les nouvelles requêtes peuvent être traitées rapidement. Nous devons corriger cela, mais d'abord, observons le problème se produire pour de vrai.

Simuler une longue requête à traiter avec l'implémentation actuelle du serveur

Nous allons voir comment une requête longue à traiter peut affecter le traitement des autres requêtes avec l'implémentation actuelle de notre serveur. L'encart 20-10 rajoute le traitement d'une requête pour */pause* qui va simuler une longue réponse qui va faire en sorte que le serveur soit en pause pendant 5 secondes avant de pouvoir répondre à nouveau.

Fichier : src/main.rs

```
use std::thread;
use std::time::Duration;
// -- partie masquée ici--

fn gestion_connexion(mut flux: TcpStream) {
    // -- partie masquée ici--

    let get = b"GET / HTTP/1.1\r\n";
    let pause = b"GET /pause HTTP/1.1\r\n";

    let (ligne_statut, nom_fichier) = if tampon.starts_with(get) {
        ("HTTP/1.1 200 OK", "hello.html")
    } else if tampon.starts_with(pause) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    // -- partie masquée ici--
}
```

Encart 20-10 : simulation d'une requête provoquant un long traitement en détectant */pause* et en faisant une pause de 5 secondes

Ce code est peu brouillon, mais est suffisant pour nos besoins de simulation. Nous avons créé une deuxième possibilité de requête `pause` avec des données que notre serveur peut reconnaître. Nous avons ajouté un `else if` après le bloc `if` pour tester les requêtes destinées à */pause*. Lorsque cette requête est reçue, le serveur va se mettre en pause pendant 5 secondes avant de générer la page HTML de succès.

Vous pouvez constater à quel point notre serveur est primitif : une bibliothèque digne de ce nom devrait gérer la détection de différents types de requêtes de manière bien moins verbeuse !

Démarrez le serveur en utilisant `cargo run`. Ouvrez ensuite deux fenêtres de navigateur web : une pour *http://127.0.0.1:7878/* et l'autre pour *http://127.0.0.1:7878/pause*. Si vous demandez l'URI */* plusieurs fois, comme vous l'avez fait précédemment, vous constaterez que le serveur répond rapidement. Mais lorsque vous saisirez */pause* et que vous chargerez ensuite */*, vous constaterez que */* attend que `pause` ait fini sa pause de 5 secondes avant de se charger.

Il y a plusieurs manières de changer le fonctionnement de notre serveur web pour éviter d'accumuler des requêtes après une requête dont le traitement est long ; celle que nous allons implémenter est un groupe de tâches.

Améliorer le débit avec un groupe de tâches

Un *groupe de tâches* est un groupe constitué de tâches qui ont été créées et qui attendent des missions. Lorsque le programme reçoit une nouvelle mission, il assigne une des tâches du groupe pour cette mission, et cette tâche va traiter la mission. Les tâches restantes dans le groupe restent disponibles pour traiter d'autres missions qui peuvent arriver pendant que la première tâche est en cours de traitement. Lorsque la première tâche en a fini avec sa mission, elle retourne dans le groupe de tâches inactives, prête à gérer une nouvelle tâche. Un groupe de tâches vous permet de traiter plusieurs connexions en simultané, ce qui augmente le débit de votre serveur.

Nous allons limiter le nombre de tâches dans le groupe à un petit nombre pour nous protéger d'attaques par déni de service (Denial of Service, DoS) ; si notre programme créait une nouvelle tâche à chaque requête qu'il reçoit, quelqu'un qui ferait 10 millions de requêtes à notre serveur pourrait faire des ravages en utilisant toutes les ressources de notre serveur et bloquer ainsi le traitement de toute nouvelle requête.

Plutôt que de générer des tâches en quantité illimitée, nous allons faire en sorte qu'il y ait un

nombre fixe de tâches qui seront en attente dans le groupe. Lorsqu'une requête arrive, une tâche sera choisie dans le groupe pour procéder au traitement. Le groupe gèrera une file d'attente pour les requêtes entrantes. Chaque tâche dans le groupe va récupérer une requête dans cette liste d'attente, la traiter puis demander une autre requête à la file d'attente. Avec ce fonctionnement, nous pouvons traiter `N` requêtes en concurrence, où `N` est le nombre de tâches. Si toutes les tâches répondent chacune à une requête longue à traiter, les requêtes suivantes vont se stocker dans la file d'attente, mais nous aurons quand même augmenté le nombre de requêtes longues que nous pouvons traiter avant d'en arriver là.

Cette technique n'est qu'une des nombreuses manières d'améliorer le débit d'un serveur web. D'autres options que vous devriez envisager sont le modèle fork/join et le modèle d'entrée-sortie asynchrone monotâche. Si vous êtes intéressés par ce sujet, vous pouvez aussi en apprendre plus sur ces autres solutions et essayer de les implémenter en Rust ; avec un langage bas niveau comme Rust, toutes les options restent possibles.

Avant que nous ne commencions l'implémentation du groupe de tâches, parlons de l'utilisation du groupe. Lorsque vous essayez de concevoir du code, commencer par écrire l'interface client peut vous aider à vous guider dans la conception. Ecrivez l'API du code afin qu'il soit structuré de la manière dont vous souhaitez l'appeler ; puis implémentez ensuite la fonctionnalité au sein de cette structure, plutôt que d'implémenter la fonctionnalité puis de concevoir l'API publique.

De la même manière que nous avons utilisé le développement piloté par les tests dans le projet du chapitre 12, nous allons utiliser ici le développement orienté par le compilateur. Nous allons écrire le code qui appelle les fonctions que nous souhaitons, et ensuite nous analyserons les erreurs du compilateur pour déterminer ce qu'il faut ensuite corriger pour que le code fonctionne.

La structure du code si nous pouvions créer une tâche pour chaque requête

Pour commencer, voyons à quoi ressemblerait notre code s'il créait une nouvelle tâche pour chaque connexion. Comme nous l'avons évoqué précédemment, cela ne sera pas notre solution finale à cause des problèmes liés à la création potentielle d'un nombre illimité de tâches, mais c'est un début. L'encart 20-11 montre les changements à apporter au `main` pour créer une nouvelle tâche pour gérer chaque flux avec une boucle `for` .

Fichier : src/main.rs

```
fn main() {
    let ecouteur = TcpListener::bind("127.0.0.1:7878").unwrap();

    for flux in ecouteur.incoming() {
        let flux = flux.unwrap();

        thread::spawn(|| {
            gestion_connexion(flux);
        });
    }
}
```

Encart 20-11 : création d'une nouvelle tâche pour chaque flux

Comme vous l'avez appris au chapitre 16, `thread::spawn` va créer une nouvelle tâche puis exécuter dans cette nouvelle tâche le code présent dans la fermeture. Si vous exécutez ce code et chargez */pause* dans votre navigateur, et que vous ouvrez */* dans deux nouveaux onglets, vous constaterez en effet que les requêtes vers */* n'aurons pas à attendre que */pause* se finisse. Mais comme nous l'avons mentionné, cela peut potentiellement surcharger le système si vous créez des nouvelles tâches sans aucune limite.

Créer une interface similaire pour un nombre fini de tâches

Nous souhaitons faire en sorte que notre groupe de tâches fonctionne de la même manière, donc passer des tâches à un groupe de tâches ne devrait pas nécessiter de gros changements au code qui utilise notre API. L'encart 20-12 montre une interface possible pour une structure `GroupeTaches` que nous souhaitons utiliser à la place de `thread::spawn`.

Fichier : src/main.rs

```
fn main() {
    let ecouteur = TcpListener::bind("127.0.0.1:7878").unwrap();
    let groupe = GroupeTaches::new(4);

    for flux in ecouteur.incoming() {
        let flux = flux.unwrap();

        groupe.executer(|| {
            gestion_connexion(flux);
        });
    }
}
```

Encart 20-12 : Notre interface idéale `GroupeTaches`

Nous avons utilisé `GroupeTaches::new` pour créer un nouveau groupe de tâches avec un nombre configurable de tâches, dans notre cas, quatre. Ensuite, dans la boucle `for`, `groupe.executer` a une interface similaire à `thread::spawn` qui prend une fermeture que le groupe devra exécuter pour chaque flux. Nous devons implémenter `groupe.executer` pour qu'il prenne la fermeture et la donne à une tâche dans le groupe pour qu'elle l'exécute. Ce code ne se compile pas encore, mais nous allons faire comme si c'était le cas pour que le compilateur puisse nous guider dans la résolution des problèmes.

Construire la structure `GroupeTaches` en utilisant le développement orienté par le compilateur

Faites les changements de l'encart 20-12 dans votre *src/main.rs*, et utilisez ensuite les erreurs du compilateur lors du `cargo check` pour orienter votre développement. Voici la première erreur que nous obtenons :

```
$ cargo check
    Checking salutations v0.1.0 (file:///projects/salutations)
error[E0433]: failed to resolve: use of undeclared type `GroupeTaches`
  --> src/main.rs:10:16
   |
10 |     let groupe = GroupeTaches::new(4);
   |                  ^^^^^^^^^^^^ use of undeclared type `GroupeTaches`

For more information about this error, try `rustc --explain E0433`.
error: could not compile `hello` due to previous error
```

Bien ! Cette erreur nous informe que nous avons besoin d'un type ou d'un module qui s'appelle `GroupeTaches`, donc nous allons le créer. Notre implémentation de `GroupeTaches` sera indépendante du type de travail qu'accomplira notre serveur web. Donc, transformons la crate binaire `salutations` en crate de bibliothèque pour y implémenter notre `GroupeTaches`. Après l'avoir changé en crate de bibliothèque, nous pourrons utiliser ensuite cette bibliothèque de groupe de tâches dans n'importe quel projet où nous aurons besoin d'un groupe de tâches, et pas seulement pour servir des requêtes web.

Créez un *src/lib.rs* qui contient ce qui suit et qui est la définition la plus simple d'une structure `GroupeTaches` que nous pouvons avoir pour le moment :

Fichier : src/lib.rs

```
pub struct GroupeTaches;
```

Créez ensuite un nouveau dossier, *src/bin*, et déplacez-y la crate binaire *src/main.rs* qui sera donc désormais *src/bin/main.rs*. Ceci va faire que la crate de bibliothèque sera la crate

principale dans le dossier *salutations* ; nous pouvons quand même continuer à exécuter le binaire dans *src/bin/main.rs* en utilisant `cargo run`. Après avoir déplacé le fichier *main.rs*, modifiez-le pour importer la crate de bibliothèque et importer `GroupeTaches` dans la portée en ajoutant le code suivant en haut de *src/bin/main.rs* :

Fichier : src/bin/main.rs

```
use salutations::GroupeTaches;
```

Ce code ne fonctionne toujours pas, mais vérifions-le à nouveau pour obtenir l'erreur que nous devons maintenant résoudre :

```
$ cargo check
    Checking salutations v0.1.0 (file:///projects/salutations)
error[E0599]: no function or associated item named `new` found for struct
`GroupeTaches` in the current scope
  --> src/bin/main.rs:11:28
   |
11 |     let groupe = GroupeTaches::new(4);
   |                                ^^^ function or associated item not found in
`GroupeTaches`

For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` due to previous error
```

Cette erreur indique que nous devons ensuite créer une fonction associée `new` pour `GroupeTaches`. Nous savons aussi que `new` nécessite d'avoir un paramètre qui peut accepter `4` comme argument et doit retourner une instance de `GroupeTaches`. Implémentons la fonction `new` la plus simple possible qui aura ces caractéristiques :

Fichier : src/lib.rs

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

Nous avons choisi `usize` comme type du paramètre `taille`, car nous savons qu'un nombre négatif de tâches n'a pas de sens. Nous savons également que nous allons utiliser ce 4 comme étant le nombre d'éléments dans une collection de tâches, ce qui est à quoi sert le type `usize`, comme nous l'avons vu dans la section “Types de nombres entiers” du chapitre 3.

Vérifions à nouveau le code :

```
$ cargo check
    Checking salutations v0.1.0 (file:///projects/salutations)
error[E0599]: no method named `executer` found for struct `GroupeTaches` in the
current scope
  --> src/bin/main.rs:16:14
   |
16 |         groupe.executer(|| {
   |                ^^^^^^^^ method not found in `GroupeTaches`

For more information about this error, try `rustc --explain E0599`.
error: could not compile `hello` due to previous error
```

Désormais, nous obtenons une erreur car nous n'avons pas implémenté la méthode `executer` sur `GroupeTaches`. Souvenez-vous que nous avions décidé dans la section "Créer une interface similaire pour un nombre fini de tâches" que notre groupe de tâches devrait avoir une interface similaire à `thread::spawn`. C'est pourquoi nous allons implémenter la fonction `executer` pour qu'elle prenne en argument la fermeture qu'on lui donne et qu'elle la passe à une tâche inactive du groupe pour qu'elle l'exécute.

Nous allons définir la méthode `executer` sur `GroupeTaches` pour prendre en paramètre une fermeture. Souvenez-vous que nous avions vu dans une section du chapitre 13 que nous pouvions prendre en paramètre les fermetures avec trois types de traits différents : `Fn`, `FnMut`, et `FnOnce`. Nous devons décider quel genre de fermeture nous allons utiliser ici. Nous savons que nous allons faire quelque chose de sensiblement identique à l'implémentation du `thread::spawn` de la bibliothèque standard, donc nous pouvons nous inspirer de ce qui lie la signature de `thread::spawn` à son paramètre. La documentation nous donne ceci :

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T,
        F: Send + 'static,
        T: Send + 'static,
```

Le paramètre de type `F` est celui qui nous intéresse ici ; le paramètre de type `T` est lié à la valeur de retour, et ceci ne nous intéresse pas ici. Nous pouvons constater que `spawn` utilise le trait `FnOnce` lié à `F`. C'est probablement ce dont nous avons besoin, parce que nous allons sûrement passer cet argument dans le `execute` de `spawn`. Nous pouvons aussi être sûr que `FnOnce` est le trait dont nous avons besoin car la tâche qui va traiter une requête ne va le faire qu'une seule fois, ce qui correspond à la partie `Once` dans `FnOnce`.

Le paramètre de type `F` a aussi le trait lié `Send` et la durée de vie liée `'static`, qui sont

utiles dans notre situation : nous avons besoin de `Send` pour transférer la fermeture d'une tâche vers une autre et de `'static` car nous ne connaissons pas la durée d'exécution de la tâche. Créons donc une méthode `executer` sur `GroupeTaches` qui va utiliser un paramètre générique de type `F` avec les liens suivants :

Fichier : src/lib.rs

```
impl GroupeTaches {
    // -- partie masquée ici--
    pub fn executer<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
    }
}
```

Nous utilisons toujours le `()` après `FnOne` car ce `FnOnce` représente une fermeture qui ne prend pas de paramètres et retourne le type unité `()`. Exactement comme les définitions de fonctions, le type de retour peut être omis de la signature, mais même si elle ne contient pas de paramètre, nous avons tout de même besoin des parenthèses.

A nouveau, c'est l'implémentation la plus simpliste de la méthode `executer` : elle ne fait rien, mais nous essayons seulement de faire en sorte que notre code se compile. Vérifions-le à nouveau :

```
$ cargo check
    Checking salutations v0.1.0 (file:///projects/salutations)
    Finished dev [unoptimized + debuginfo] target(s) in 0.24s
```

Cela se compile ! Mais remarquez que si vous lancez `cargo run` et faites la requête dans votre navigateur web, vous verrez l'erreur dans le navigateur que nous avions tout au début du chapitre. Notre bibliothèque n'exécute pas encore la fermeture envoyée à `executer` !

> Remarque : un dicton que vous avez probablement déjà entendu à propos des compilateurs stricts, comme Haskell et Rust, est que “si le code se compile, il fonctionne”. Mais ce dicton n'est pas toujours vrai. Notre projet se compile, mais il ne fait absolument rien ! Si nous construisions un vrai projet, complexe, il serait bon de commencer à écrire des tests unitaires pour vérifier que ce code compile *et* qu'il suit le comportement que nous souhaitons.

Valider le nombre de tâches envoyé à `new`

Nous ne faisons rien avec les paramètres passés à `new` et `executer`. Implémentons le corps de ces fonctions avec le comportement que nous souhaitons. Pour commencer, réfléchissons à `new`. Précédemment, nous avions choisi un type sans signe pour le paramètre `taille`, car un groupe avec un nombre négatif de tâches n'a pas de sens. Cependant, un groupe avec aucune tâche n'a pas non plus de sens, alors que zéro est une valeur parfaitement valide pour `usize`. Nous allons ajouter du code pour vérifier que `taille` est plus grand que zéro avant de retourner une instance de `GroupeTaches` et faire en sorte que le programme panique s'il reçoit un zéro, en utilisant la macro `assert!` comme dans l'encart 20-13.

Filename : src/lib.rs

```
impl GroupeTaches {
    /// Crée un nouveau GroupeTaches.
    ///
    /// La taille est le nom de tâches présentes dans le groupe.
    ///
    /// # Panics
    ///
    /// La fonction `new` devrait paniquer si la taille vaut zéro.
    pub fn new(taille: usize) -> GroupeTaches {
        assert!(taille > 0);

        GroupeTaches
    }

    // -- partie masquée ici --
}
```

Encart 20-13 : implémentation de `GroupeTaches::new` qui devrait paniquer si `taille` vaut zéro

Nous avons ajouté un peu de documentation pour notre `GroupeTaches` avec des commentaires de documentation. Remarquez que nous avons suivi les pratiques de bonne documentation en ajoutant une section qui liste les situations pour lesquelles notre fonction peut paniquer, comme nous l'avons vu dans le chapitre 14. Essayez de lancer `cargo doc --open` et de cliquer sur la structure `GroupeTaches` pour voir à quoi ressemble la documentation générée pour `new` !

Au lieu d'ajouter la macro `assert!` comme nous venons de le faire, nous aurions pu faire en sorte que `new` retourne un `Result` comme nous l'avions fait avec `Config::new` dans le projet d'entrée/sortie dans l'encart 12-9. Mais nous avons décidé que dans le cas présent, la création d'un groupe de tâches sans aucune tâche devait être une erreur irrécupérable. Si vous en sentez l'envie, essayez d'écrire une version de `new` avec la signature suivante, pour comparer les deux versions :

```
pub fn new(taille: usize) -> Result<GroupeTaches, ErreurGroupeTaches> {
```

Créer l'espace de rangement des tâches

Maintenant que nous avons une manière de savoir si nous avons un nombre valide de tâches à stocker dans le groupe, nous pouvons créer ces tâches et les stocker dans la structure `GroupeTaches` avant de la retourner. Mais comment “stocker” une tâche ? Regardons à nouveau la signature de `thread::spawn` :

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    where
        F: FnOnce() -> T,
        F: Send + 'static,
        T: Send + 'static,
```

La fonction `spawn` retourne un `JoinHandle<T>`, où `T` est le type que retourne notre fermeture. Essayons d'utiliser nous aussi `JoinHandle` pour voir ce qu'il va se passer. Dans notre cas, les fermetures que nous passons dans le groupe de tâches vont traiter les connexions mais ne vont rien retourner, donc `T` sera le type unité, `()`.

Le code de l'encart 20-14 va se compiler mais ne va pas encore créer de tâches pour le moment. Nous avons changé la définition de `GroupeTaches` pour qu'elle possède un vecteur d'instances `thread::JoinHandle<()>`, nous avons initialisé le vecteur avec une capacité de la valeur de `taille`, mis en place une boucle `for` qui va exécuter du code pour créer les tâches puis nous avons retourné une instance de `GroupeTaches` qui les contient.

Fichier : src/lib.rs

```
use std::thread;

pub struct GroupeTaches {
    taches: Vec<thread::JoinHandle<()>>,
}

impl GroupeTaches {
    // -- partie masquée ici --
    pub fn new(taille: usize) -> GroupeTaches {
        assert!(taille > 0);

        let mut taches = Vec::with_capacity(taille);

        for _ in 0..taille {
            // on crée quelques tâches ici et on les stocke dans le vecteur
        }

        GroupeTaches { taches }
    }

    // -- partie masquée ici --
}
```

Encart 20-14 : création d'un vecteur pour `GroupeTaches` pour stocker les tâches

Nous avons importé `std::thread` dans la portée de la crate de bibliothèque, car nous utilisons `thread::JoinHandle` comme étant le type des éléments du vecteur dans `GroupeTaches`.

Une fois qu'une taille valide est reçue, notre `GroupeTaches` crée un nouveau vecteur qui peut stocker `taille` éléments. Nous n'avons pas encore utilisé la fonction `with_capacity` dans ce livre, qui fait la même chose que `Vec::new` mais avec une grosse différence : elle pré-alloue l'espace dans le vecteur. Comme nous savons que nous avons besoin de stocker `taille` éléments dans le vecteur, faire cette allocation en amont est bien plus efficace que d'utiliser `Vec::new` qui va se redimensionner lorsque des éléments lui seront ajoutés.

Lorsque vous lancez à nouveau `cargo check`, vous devriez avoir quelques avertissements en plus, mais cela devrait être un succès.

Une structure `Operateur` chargée d'envoyer le code de `GroupeTaches` à une tâche

Nous avions laissé un commentaire dans la boucle `for` dans l'encart 20-14 qui concernait la création des tâches. Maintenant, nous allons voir comment créer ces tâches. La bibliothèque standard fournit un moyen de créer des tâches avec `thread::spawn` à qui il faut passer le code que la tâche doit exécuter dès qu'elle est créée. Cependant, dans notre cas, nous souhaitons créer des tâches et faire en sorte qu'elles *attendent* du code que nous leur

enverrons plus tard. L'implémentation des tâches de la bibliothèque standard n'offre aucun moyen de faire ceci ; nous devons donc implémenter cela nous-même.

Nous allons implémenter ce comportement en introduisant une nouvelle structure de données entre le `GroupeTaches` et les tâches qui va gérer ce nouveau comportement. Nous allons appeler cette structure `Operateur`, nom qui lui est traditionnellement donné avec `Worker` dans les implémentations de groupe de tâches. Imaginez des personnes qui travaillent dans la cuisine d'un restaurant : les opérateurs attendent les commandes des clients puis sont chargés de prendre en charge ces commandes et d'y répondre.

Au lieu de stocker un vecteur d'instances `JoinHandle<()>` dans le groupe de tâches, nous allons stocker des instances de structure `Operateur`. Chaque `Operateur` va stocker une seule instance de `JoinHandle<()>`. Ensuite nous implémenterons une méthode sur `Operateur` qui va prendre en argument une fermeture de code à exécuter et l'envoyer à la tâche qui fonctionne déjà pour exécution. Nous allons aussi donner à chacun des opérateurs un identifiant `id` afin que nous puissions distinguer les différents opérateurs dans le groupe dans les journaux ou lors de débogages.

Appliquons ces changements à l'endroit où nous créons un `GroupeTaches`. Nous allons implémenter le code de `Operateur` qui envoie la fermeture à la tâche en suivant ces étapes :

1. Définir une structure `Operateur` qui possède un `id` et un `JoinHandle<()>`.
2. Modifier le `GroupeTaches` afin qu'il possède un vecteur d'instances de `Operateur`.
3. Définir une fonction `Operateur::new` qui prend en argument un numéro d'`id` et retourne une instance de `Operateur` qui contient l'`id` et une tâche créée avec une fermeture vide.
4. Dans `GroupeTaches::new`, utiliser le compteur de la boucle `for` pour générer un `id`, créer un nouveau `Operateur` avec cet `id` et stocker l'opérateur dans le vecteur.

Si vous vous sentez prêt(e) à relever le défi, essayez de faire ces changements de votre côté avant de regarder le code de l'encart 20-15.

Vous êtes prêt(e) ? Voici l'encart 20-15 qui propose une solution pour procéder aux changements listés précédemment.

Fichier : src/lib.rs

```
use std::thread;

pub struct GroupeTaches {
    operateurs: Vec<Operateur>,
}

impl GroupeTaches {
    // -- partie masquée ici --
    pub fn new(taille: usize) -> GroupeTaches {
        assert!(taille > 0);

        let mut operateurs = Vec::with_capacity(taille);

        for id in 0..taille {
            operateurs.push(Operateur::new(id));
        }

        GroupeTaches { operateurs }
    }
    // -- partie masquée ici --
}

struct Operateur {
    id: usize,
    tache: thread::JoinHandle<()>,
}

impl Operateur {
    fn new(id: usize) -> Operateur {
        let tache = thread::spawn(|| {});

        Operateur { id, tache }
    }
}
```

Encart 20-15 : modification de `GroupeTaches` pour stocker des instances de `Operateur` plutôt que de stocker directement des tâches

Nous avons changé le nom du champ `taches` de `GroupeTaches` en `operateurs` car il stocke maintenant des instances de `Operateur` plutôt que des instances de `JoinHandle<()>`. Nous utilisons le compteur de la boucle `for` comme argument de `Operateur::new` et nous stockons chacun des nouveaux `Operateur` dans le vecteur `operateurs`.

Le code externe (comme celui de notre serveur dans *src/bin/main.rs*) n'a pas besoin de connaître les détails de l'implémentation qui utilise une structure `Operateur` dans `GroupeTaches`, donc nous faisons en sorte que la structure `Operateur` et sa fonction `new` soient privées. La fonction `Operateur::new` utilise l' `id` que nous lui donnons et stocke une

instance de `JoinHandle<()>` qui est créée en instanciant une nouvelle tâche utilisant une fermeture vide.

Ce code va se compiler et stocker le nombre d'instances de `Operateur` que nous avons renseigné en argument de `GroupeTaches::new`. Mais nous n'exécutons *toujours pas* la fermeture que nous obtenons de `executer`. Voyons maintenant comment faire cela.

Envoyer des requêtes à des tâches via des canaux

Maintenant nous allons nous attaquer au problème qui fait que les fermetures passées à `thread::spawn` ne font absolument rien. Actuellement, nous obtenons la fermeture que nous souhaitons exécuter dans la méthode `executer`. Mais nous avons besoin de donner une fermeture à `thread::spawn` à exécuter lorsque nous créons chaque `Operateur` lors de la création de `GroupeTaches`.

Nous souhaitons que les structures `Operateur` que nous venons de créer récupèrent du code à exécuter dans une liste d'attente présente dans le `GroupeTaches` et renvoient ce code à leur tâche pour l'exécuter.

Dans le chapitre 16, vous avez appris les *canaux* (une manière simple de communiquer entre deux tâches) qui seront parfaits pour ce cas d'emploi. Nous allons utiliser un canal pour les fonctions pour créer la liste d'attente des missions, et `executer` devrait envoyer une mission de `GroupeTaches` vers les instances `Operateur`, qui vont passer la mission à leurs tâches. Voici le plan :

1. Le `GroupeTaches` va créer un canal et se connecter à la partie émettrice de ce canal.
2. Chaque `Operateur` va se connecter à la partie réceptrice du canal.
3. Nous allons créer une nouvelle structure `Mission` qui va stocker les fermetures que nous souhaitons envoyer dans le canal.
4. La méthode `executer` va envoyer la mission qu'elle souhaite executer à la partie émettrice du canal.
5. Dans sa propre tâche, l' `Operateur` va vérifier en permanence la partie réceptrice du canal et exécuter les fermetures des missions qu'il va recevoir.

Commençons par créer un canal dans `GroupeTaches::new` et stocker la partie émettrice dans l'instance de `GroupeTaches`, comme dans l'encart 20-16. La structure `Mission` ne contient rien pour le moment mais sera le type d'éléments que nous enverrons dans le canal.

Fichier : src/lib.rs

```
// -- partie masquée ici --
use std::sync::mpsc;

pub struct GroupeTaches {
    operateurs: Vec<Operateur>,
    envoi: mpsc::Sender<Mission>,
}

struct Mission;

impl GroupeTaches {
    // -- partie masquée ici --
    pub fn new(taille: usize) -> GroupeTaches {
        assert!(taille > 0);

        let (envoi, reception) = mpsc::channel();

        let mut operateurs = Vec::with_capacity(taille);

        for id in 0..taille {
            operateurs.push(Operateur::new(id));
        }

        GroupeTaches { operateurs, envoi }
    }
    // -- partie masquée ici --
}
```

Encart 20-16 : modification de `GroupeTaches` pour stocker la partie émettrice du canal qui émet des instances de `Mission`

Dans `GroupeTaches::new`, nous créons notre nouveau canal et faisons en sorte que le groupe stocke la partie émettrice. Cela devrait pouvoir se compiler, mais il subsiste des avertissements.

Essayons de donner la partie réceptrice du canal à chacun des opérateurs lorsque le groupe de tâches crée le canal. Nous savons que nous voulons utiliser la partie réceptrice dans la tâche que l'opérateur utilise, donc nous allons créer une référence vers le paramètre `reception` dans la fermeture. Le code de l'encart 20-17 ne se compile pas encore.

Fichier : src/lib.rs

```
impl GroupeTaches {
    // -- partie masquée ici --
    pub fn new(taille: usize) -> GroupeTaches {
        assert!(taille > 0);

        let (envoi, reception) = mpsc::channel();

        let mut operateurs = Vec::with_capacity(taille);

        for id in 0..taille {
            operateurs.push(Operateur::new(id, reception));
        }

        GroupeTaches { operateurs, envoi }
    }
    // -- partie masquée ici --
}

// -- partie masquée ici --

impl Operateur {
    fn new(id: usize, reception: mpsc::Receiver<Mission>) -> Operateur {
        let tache = thread::spawn(|| {
            reception;
        });

        Operateur { id, tache }
    }
}
```

Encart 20-17 : envoi de la partie réceptrice du canal aux opérateurs

Nous avons juste fait de petites modifications simples : nous envoyons la partie réceptrice du canal dans `Operateur::new` puis nous l'utilisons dans la fermeture.

Lorsque nous essayons de vérifier ce code, nous obtenons cette erreur :

```
$ cargo check
    Checking salutations v0.1.0 (file:///projects/salutations)
error[E0382]: use of moved value: `reception`
  --> src/lib.rs:27:42
   |
22 |         let (envoi, reception) = mpsc::channel();
   |                     --------- move occurs because `reception` has type
`std::sync::mpsc::Receiver<Job>`, which does not implement the `Copy` trait
...
27 |             operateurs.push(Worker::new(id, reception));
   |                                          ^^^^^^^^^ value moved here, in
previous iteration of loop

For more information about this error, try `rustc --explain E0382`.
error: could not compile `hello` due to previous error
```

Le code essaye d'envoyer `reception` dans plusieurs instances de `Operateur`. Ceci ne fonctionne pas, comme vous l'avez appris au chapitre 16 : l'implémentation du canal que fournit Rust est du type plusieurs *producteurs*, un seul *consommateur*. Cela signifie que nous ne pouvons pas simplement cloner la partie réceptrice du canal pour corriger ce code. Même si nous aurions pu le faire, ce n'est pas la solution que nous souhaitons utiliser ; nous voulons plutôt distribuer les missions entre les tâches en partageant la même réception entre tous les opérateurs.

De plus, obtenir une mission de la file d'attente du canal implique de modifier la `reception`, donc les tâches ont besoin d'une méthode sécurisée pour partager et modifier `reception` ; autrement, nous risquons de nous trouver dans des situations de concurrence (comme nous l'avons vu dans le chapitre 16).

Souvenez-vous des pointeurs intelligents conçus pour les échanges entre les tâches que nous avons vus au chapitre 16 : pour partager la possession entre plusieurs tâches et permettre aux tâches de modifier la valeur, nous avons besoin d'utiliser `Arc<Mutex<T>>`. Le type `Arc` va permettre à plusieurs opérateurs de posséder la réception tandis que `Mutex` va s'assurer qu'un seul opérateur obtienne une mission dans la réception à un moment donné. L'encart 20-18 montre les changements que nous devons apporter.

Fichier : src/lib.rs

```
use std::sync::Arc;
use std::sync::Mutex;
// -- partie masquée ici --

impl GroupeTaches {
    // -- partie masquée ici --
    pub fn new(taille: usize) -> GroupeTaches {
        assert!(taille > 0);

        let (envoi, reception) = mpsc::channel();

        let reception = Arc::new(Mutex::new(reception));

        let mut operateurs = Vec::with_capacity(taille);

        for id in 0..taille {
            operateurs.push(Operateur::new(id, Arc::clone(&reception)));
        }

        GroupeTaches { operateurs, envoi }
    }

    // -- partie masquée ici --
}

// -- partie masquée ici --

impl Operateur {
    fn new(id: usize, reception: Arc<Mutex<mpsc::Receiver<Mission>>>) ->
Operateur {
        // -- partie masquée ici --
    }
}
```

Encart 20-18 : partage de la partie réceptrice du canal entre les opérateurs en utilisant `Arc` et `Mutex`

Dans `GroupeTaches::new`, nous installons la partie réceptrice du canal dans un `Arc` et un `Mutex`. Pour chaque nouvel opérateur, nous clonons le `Arc` pour augmenter le compteur de références afin que les opérateurs puissent se partager la possession de la partie réceptrice.

Grâce à ces changements, le code se compile ! Nous touchons au but !

Implémenter la méthode `executer`

Finissons en implémentant la méthode `executer` de `GroupeTaches`. Nous allons également modifier la structure `Mission` pour la transformer en un alias de type pour un objet trait qui

contiendra le type de fermeture que `executer` recevra. Comme nous l'avons vu dans une section du chapitre 19, les alias de type nous permettent de raccourcir les types un peu trop longs. Voyez cela dans l'encart 20-19.

Fichier : src/lib.rs

```
// -- partie masquée ici --

type Mission = Box<dyn FnOnce() + Send + 'static>;

impl GroupeTaches {
    // -- partie masquée ici --

    pub fn executer<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let mission = Box::new(f);

        self.envoi.send(mission).unwrap();
    }
}

// -- partie masquée ici --
```

Encart 20-19 : création d'un alias de type `Mission` pour une `Box` qui contient chaque fermeture et qui transportera la mission dans le canal

Après avoir créé une nouvelle instance `Mission` en utilisant la fermeture que nous obtenons dans `executer`, nous envoyons cette mission dans le canal via la partie émettrice. Nous utilisons `unwrap` sur `send` pour les cas où l'envoi échoue. Cela peut arriver si, par exemple, nous stoppons l'exécution de toutes les tâches, ce qui signifiera que les parties réceptrices auront finis de recevoir des nouveaux messages. Pour le moment, nous ne pouvons pas stopper l'exécution de nos tâches : nos tâches continuerons à s'exécuter aussi longtemps que le groupe existe. La raison pour laquelle nous utilisons `unwrap` est que nous savons que le cas d'échec ne va pas se produire, mais le compilateur ne le sait pas.

Mais nous n'avons pas encore tout à fait fini ! Dans l'opérateur, notre fermeture envoyée à `thread::spawn` ne fait que *référencer* la partie réception du canal. Au lieu de ça, nous avons besoin que la fermeture boucle à l'infini, demandant une mission à la partie réceptrice du canal et l'exécutant quand elle en obtient une. Appliquons les changements montrés dans l'encart 20-20 à `Operateur::new`.

Fichier : src/lib.rs

```
// -- partie masquée ici --

impl Operateur {
    fn new(id: usize, reception: Arc<Mutex<mpsc::Receiver<Mission>>>) ->
Operateur {
        let tache = thread::spawn(move || loop {
            let mission = reception.lock().unwrap().recv().unwrap();

            println!("L'opérateur {} a obtenu une mission ; il l'exécute.", id);

            mission();
        });

        Operateur { id, tache }
    }
}
```

Encart 20-20 : réception et exécution des missions dans la tâche de l'opérateur

Ici, nous faisons d'abord appel à `lock` sur `reception` pour obtenir le mutex, puis nous faisons appel à `unwrap` pour paniquer dès qu'il y a une erreur. L'acquisition d'un verrou peut échouer si le mutex est dans un état *empoisonné*, ce qui peut arriver si d'autres tâches ont paniqué pendant qu'elles avaient le verrou au lieu de le rendre. Dans cette situation, l'appel à `unwrap` fera paniquer la tâche, ce qui est la bonne chose à faire. Vous pouvez aussi changer ce `unwrap` en un `expect` avec un message d'erreur qui sera plus explicite pour vous.

Si nous obtenons le verrou du mutex, nous faisons appel à `recv` pour recevoir une `Mission` provenant du canal. Un `unwrap` final s'occupe lui aussi des cas d'erreurs qui peuvent se produire si la tâche qui est connectée à la partie émettrice du canal se termine, de la même manière que la méthode `send` enverrait `Err` si la partie réceptrice se fermerait.

L'appel à `recv` bloque l'exécution, donc s'il n'y a pas encore de mission, la tâche courante va attendre jusqu'à ce qu'une mission soit disponible. Le `Mutex<T>` s'assure qu'une seule tâche d'`Operateur` essaie d'obtenir une mission à un instant donné.

Notre groupe de tâches est désormais en état de fonctionner ! Faites un `cargo run` et faites quelques requêtes :

```
$ cargo run
   Compiling salutations v0.1.0 (file:///projects/salutations)
warning: field is never read: `operateurs`
 -- > src/lib.rs:7:5
  |
7 |     operateurs: Vec<Operateur>,
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(dead_code)]` on by default

warning: field is never read: `id`
  -- > src/lib.rs:48:5
   |
48 |     id: usize,
   |     ^^^^^^^^^

warning: field is never read: `tache`
  -- > src/lib.rs:49:5
   |
49 |     tache: thread::JoinHandle<()>,
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warning: 3 warnings emitted

    Finished dev [unoptimized + debuginfo] target(s) in 1.40s
     Running `target/debug/main`
L'opérateur 0 a obtenu une mission ; il l'exécute.
L'opérateur 2 a obtenu une mission ; il l'exécute.
L'opérateur 1 a obtenu une mission ; il l'exécute.
L'opérateur 3 a obtenu une mission ; il l'exécute.
L'opérateur 0 a obtenu une mission ; il l'exécute.
L'opérateur 2 a obtenu une mission ; il l'exécute.
L'opérateur 1 a obtenu une mission ; il l'exécute.
L'opérateur 3 a obtenu une mission ; il l'exécute.
L'opérateur 0 a obtenu une mission ; il l'exécute.
L'opérateur 2 a obtenu une mission ; il l'exécute.
```

Parfait ! Nous avons maintenant un groupe de tâches qui exécute des connexions de manière asynchrone. Il n'y a jamais plus de quatre tâches qui sont créées, donc notre système ne sera pas surchargé si le serveur reçoit beaucoup de requêtes. Si nous faisons une requête vers */pause*, le serveur sera toujours capable de servir les autres requêtes grâce aux autres tâches qui pourront les exécuter.

> Remarque : si vous ouvrez */pause* dans plusieurs fenêtres de navigation en simultané, elles peuvent parfois être chargées une par une avec 5 secondes d'intervalle. Certains navigateurs web exécutent plusieurs instances de la même requête de manière séquentielle pour des raisons de mise en cache. Cette limitation n'est pas imputable à notre serveur web.

Ayant appris la boucle `while let` dans le chapitre 18, vous pourriez vous demander pourquoi nous n'avons pas écrit le code des tâches des opérateurs comme dans l'encart 20-21.

Fichier : src/lib.rs

```
// -- partie masquée ici --

impl Operateur {
    fn new(id: usize, reception: Arc<Mutex<mpsc::Receiver<Mission>>>) ->
Operateur {
        let tache = thread::spawn(move || {
            while let Ok(mission) = reception.lock().unwrap().recv() {
                println!("L'opérateur {} a obtenu une mission ; il l'exécute.",
id);

                mission();
            }
        });

        Operateur { id, tache }
    }
}
```

Encart 20-21 : une implémentation alternative de `Operateur::new` qui utilise `while let`

Ce code se compile et s'exécute mais ne se produit pas le comportement des tâches que nous souhaitons : une requête lente à traiter va continuer à mettre en attente de traitement les autres requêtes. La raison à cela est subtile : la structure `Mutex` n'a pas de méthode publique `unlock` car la propriété du verrou se base sur la durée de vie du `MutexGuard<T>` au sein du `LockResult<MutexGuard<T>>` que retourne la méthode `lock` . A la compilation, le vérificateur d'emprunt peut ensuite vérifier la règle qui dit qu'une ressource gardée par un `Mutex` ne peut être accessible que si nous avons ce verrou. Mais cette implémentation peut aussi conduire à ce que nous gardions le verrou plus longtemps que prévu si nous ne réfléchissons pas avec attention à la durée de vie du `MutexGuard<T>` .

Le code de l'encart 20-20 qui utilise `let mission = reception.lock().unwrap().recv().unwrap();` fonctionne, car avec `let` , toute valeur

temporaire utilisée dans la partie droite du signe égal est libérée immédiatement lorsque l'instruction `let` se termine. Cependant, `while let` (ainsi que `if let` et `match`) ne libèrent pas les valeurs temporaires avant la fin du bloc associé. Dans l'encart 20-21, le verrou continue à être maintenu pendant toute la durée de l'appel à `mission()` , ce qui veut dire que les autres opérateurs ne peuvent pas recevoir de tâches.

Arrêt propre et nettoyage

Le code de l'encart 20-20 répond aux requêtes de manière asynchrone grâce à l'utilisation du groupe de tâches, comme nous l'espérions. Nous avons quelques avertissements sur les champs `operateurs`, `id` et `tâche` que nous n'utilisons pas directement et qui nous rappellent que nous ne nettoyons rien. Lorsque nous arrêtons brutalement la tâche principale en appuyant sur ctrl-c, toutes les autres tâches sont également immédiatement stoppées, même si elles sont en train de servir une requête.

Maintenant, nous allons implémenter le trait `Drop` afin d'appeler `join` sur chacune des tâches du groupe afin qu'elles puissent finir les requêtes qu'elles sont en train de traiter avant de s'arrêter. Ensuite, nous allons implémenter un moyen de demander aux tâches d'arrêter d'accepter de nouvelles requêtes et de s'arrêter. Pour voir ce code en action, nous allons modifier notre serveur pour n'accepter que deux requêtes avant d'arrêter proprement son groupe de tâches.

Implémenter le trait `Drop` sur `GroupeTaches`

Commençons par implémenter `Drop` sur notre groupe de tâches. Lorsque le groupe est nettoyé, nos tâches doivent toutes faire appel à `join` pour s'assurer qu'elles finissent leur travail. L'encart 20-22 montre une première tentative d'implémentation de `Drop` ; ce code ne fonctionne pas encore tout à fait.

Fichier : src/lib.rs

```
impl Drop for GroupeTaches {
    fn drop(&mut self) {
        for operateur in &mut self.operateurs {
            println!("Arrêt de l'opérateur {}", operateur.id);

            operateur.tache.join().unwrap();
        }
    }
}
```

Encart 20-22 : utilisation de `join` sur chaque tâche lorsque le groupe de tâches sort de la portée

D'abord, nous faisons une boucle sur tous les `operateurs` du groupe de tâches. Pour ce faire, nous utilisons `&mut` car `self` n'est qu'une référence mutable du groupe de tâches mais nous aurons également besoin de pouvoir muter chaque `operateur`. Pour chaque opérateur, nous affichons un message qui indique qu'il s'arrête puis nous faisons appel à

`join` sur la tâche de cet opérateur. Si l'appel à `join` échoue, nous utilisons `unwrap` pour faire paniquer Rust et ainsi procéder à un arrêt brutal.

Voici l'erreur que nous obtenons lorsque nous compilons ce code :

```
$ cargo check
    Checking salutations v0.1.0 (file:///projects/salutations)
error[E0507]: cannot move out of `operateur.tache` which is behind a mutable
reference
  --> src/lib.rs:52:13
   |
52 |             operateur.tache.join().unwrap();
   |             ^^^^^^^^^^^^^^^^ move occurs because `operateur.tache` has type
`JoinHandle<()>`, which does not implement the `Copy` trait

For more information about this error, try `rustc --explain E0507`.
error: could not compile `hello` due to previous error
```

L'erreur nous informe que nous ne pouvons pas faire appel à `join` car nous avons seulement fait un emprunt mutable pour chacun des `operateur` alors que `join` prend possession de son argument. Pour résoudre ce problème, nous devons sortir la `tache` de l'instance de `Operateur` qui la possède afin que `join` puisse la consommer. Nous faisons ceci dans l'encart 17-15 : comme `Operateur` contient désormais un `Option<thread::JoinHandle<()>>` , nous pouvons utiliser la méthode `take` sur `Option` pour sortir la valeur de la variante `Some` et y mettre à la place une variante `None` . Autrement dit, un `Operateur` qui est en cours d'exécution aura une variante `Some` dans `tache` , et lorsque nous souhaiterons nettoyer `Operateur` , nous remplacerons `Some` par `None` afin que `Operateur` n'ait pas de tâche à exécuter.

Donc nous savons que nous voulons modifier la définition de `Operateur` comme ceci :

Fichier : src/lib.rs

```
struct Operateur {
    id: usize,
    tache: Option<thread::JoinHandle<()>>,
}
```

Maintenant, aidons-nous du compilateur pour trouver les autres endroits qui ont besoin de changer. En vérifiant ce code, nous obtenons deux erreurs :

```
$ cargo check
    Checking salutations v0.1.0 (file:///projects/salutations)
error[E0599]: no method named `join` found for enum `Option` in the current
scope
  --> src/lib.rs:52:27
   |
52 |             operateur.tache.join().unwrap();
   |                           ^^^^ method not found in
`Option<JoinHandle<()>>`

error[E0308]: mismatched types
  --> src/lib.rs:72:22
   |
72 |         Operateur { id, tache }
   |                     ^^^^^ expected enum `Option`, found struct
`JoinHandle`
   |
   = note: expected enum `Option<JoinHandle<()>>`
           found struct `JoinHandle<_>`
help: try wrapping the expression in `Some`
   |
72 |         Operateur { id, Some(tache) }
   |                     +++++     +

Some errors have detailed explanations: E0308, E0599.
For more information about an error, try `rustc --explain E0308`.
error: could not compile `hello` due to 2 previous errors
```

Corrigeons la seconde erreur, qui se situe dans le code à la fin de `Operateur::new` : nous devons intégrer la valeur de `tache` dans un `Some` lorsque nous créons un nouvel `Operateur` . Faites les changements suivants pour corriger cette erreur :

Fichier : src/lib.rs

```
impl Operateur {
    fn new(id: usize, reception: Arc<Mutex<mpsc::Receiver<Mission>>>) ->
Operateur {
        // -- partie masquée ici --

        Operateur {
            id,
            tache: Some(tache),
        }
    }
}
```

La première erreur se situe dans notre implémentation de `Drop` . Nous avions mentionné plus tôt que nous voulions faire appel à `take` sur la valeur de `Option` pour déplacer `tache` en dehors de `operateur` . Voici les changements à apporter pour ceci :

Fichier : src/lib.rs

```
impl Drop for GroupeTaches {
    fn drop(&mut self) {
        for operateur in &mut self.operateurs {
            println!("Arrêt de l'opérateur {}", operateur.id);

            if let Some(tache) = operateur.tache.take() {
                tache.join().unwrap();
            }
        }
    }
}
```

Comme nous l'avons vu au chapitre 17, la méthode `take` sur `Option` sort la variante `Some` et laisse un `None` à la place. Nous utilisons `if let` pour destructurer le `Some` et obtenir la tâche ; ensuite nous faisons appel à `join` sur cette tâche. Si la tâche d'un opérateur est déjà un `None`, nous savons qu'il a déjà nettoyé sa tâche et que dans ce cas nous n'avons rien à faire.

Demander aux tâches d'arrêter d'attendre des missions

Avec tous ces changements, notre code se compile désormais sans aucun avertissement. Mais la mauvaise nouvelle est que pour l'instant ce code ne fonctionne comme nous le souhaitons. La cause se situe dans la logique des fermetures qui sont exécutées par les tâches des instances de `Operateur` : pour le moment, nous faisons appel à `join`, mais cela ne va pas arrêter les tâches car elles font une boucle infinie avec `loop` pour attendre des missions. Si nous essayons de nettoyer notre `GroupeTaches` avec l'implémentation actuelle de `drop`, la tâche principale va se bloquer pour toujours en attendant en vain que la première tâche se termine.

Pour corriger ce problème, nous allons modifier les tâches pour qu'elles attendent soit une `Mission` à exécuter, soit le signal qui leur dit qu'elles doivent arrêter d'attendre des missions et sortir de la boucle infinie. Notre canal va envoyer une de ces deux variantes d'énumération au lieu d'instances de `Mission`.

Fichier : src/lib.rs

```
enum Message {
    NouvelleMission(Mission),
    Extinction,
}
```

Cette énumération `Message` aura pour valeurs une variante `NouvelleMission` qui contiendra la `Mission` que la tâche devra exécuter, ou la variante `Extinction` qui va faire en sorte que la tâche sorte de sa boucle et se termine.

Nous devons corriger le canal pour utiliser les valeurs du type `Message` à la place du type `Mission`, comme dans l'encart 20-23.

Fichier : src/lib.rs

```
pub struct GroupeTaches {
    operateurs: Vec<Operateur>,
    envoi: mpsc::Sender<Message>,
}

// -- partie masquée ici --

impl GroupeTaches {
    // -- partie masquée ici --

    pub fn executer<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let mission = Box::new(f);

        self.envoi.send(Message::NouvelleMission(mission)).unwrap();
    }
}

// -- partie masquée ici --

impl Operateur {
    fn new(id: usize, reception: Arc<Mutex<mpsc::Receiver<Message>>>) ->
Operateur {
        let tache = thread::spawn(move || loop {
            let message = reception.lock().unwrap().recv().unwrap();

            match message {
                Message::NouvelleMission(mission) => {
                    println!("L'opérateur {} a reçu une mission ; il
l'exécute.", id);

                    mission();
                }
                Message::Extinction => {
                    println!("L'opérateur {} a reçu l'instruction d'arrêt.",
id);

                    break;
                }
            }
        });

        Operateur {
            id,
            tache: Some(tache),
        }
    }
}
```

Encart 20-23 : envoi et réception de valeurs de `Message` et sortie de la boucle si un

`Operateur` reçoit `Message:Extinction`

Pour intégrer l'énumération `Message`, nous devons changer `Mission` par `Message` à deux endroits : dans la définition de `GroupeTaches` et dans la signature de `Operateur::new`. La méthode `executer` de `GroupeTaches` doit envoyer des missions encapsulées dans des variantes de `Message::NouvelleTache`. Ensuite, dans `Operateur::new` où nous recevons des `Message` du canal, la mission sera traitée si la variante `NouvelleTache` est reçue, ou bien la tâche arrêtera la boucle si la variante `Extinction` est reçue.

Grâce à ces changements, le code va se compiler et continuer de fonctionner de la même manière qu'il le faisait après l'encart 20-20. Mais nous allons obtenir un avertissement car nous ne créons aucun message de la variante `Extinction`. Corrigeons cet avertissement en modifiant notre implémentation de `Drop` pour qu'elle ressemble à l'encart 20-24.

Fichier : src/lib.rs

```
impl Drop for GroupeTaches {
    fn drop(&mut self) {
        println!("Envoi du message d'extinction à tous les opérateurs.");

        for _ in &self.operateurs {
            self.envoi.send(Message::Extinction).unwrap();
        }

        println!("Arrêt de tous les opérateurs.");

        for operateur in &mut self.operateurs {
            println!("Arrêt de l'opérateur {}", operateur.id);

            if let Some(tache) = operateur.tache.take() {
                tache.join().unwrap();
            }
        }
    }
}
```

Encart 20-24 : envoi de `Message::Extinction` aux opérateurs avant de d'appeler `join` sur toutes les tâches de ces opérateurs

Nous itérons deux fois sur les opérateurs : une fois pour envoyer un message `Extinction` pour chaque opérateur, et une seconde fois pour utiliser `join` sur leur tâche. Si nous avions essayé d'envoyer le message et d'utiliser immédiatement `join` dans la même boucle, nous n'aurions pas pu garantir que l'opérateur de l'itération en cours serait celui qui obtiendrait le message envoyé dans le canal.

Pour mieux comprendre pourquoi nous avons besoin de deux boucles distinctes, imaginez

un scénario avec deux opérateurs. Si nous avions utilisé une seule boucle pour itérer sur chacun des opérateurs, dans la première itération un message d'extinction aurait été envoyé dans le canal et `join` aurait été utilisé sur la tâche du premier opérateur. Si ce premier opérateur était occupé à traiter une requête à ce moment-là, le second opérateur aurait alors récupéré le message d'extinction dans le canal et se serait arrêté. Nous serions alors restés à attendre que le premier opérateur s'arrête, mais cela ne se serait jamais produit car c'est la seconde tâche qui aurait obtenu le message d'extinction. Nous serions alors dans une situation d'interblocage !

Pour éviter ce scénario, nous allons commencer par émettre tous nos messages `Extinction` dans le canal en utilisant une boucle ; puis nous utilisons `join` sur toutes les tâches dans une seconde boucle. Chaque opérateur va arrêter de recevoir de nouvelles requêtes du canal dès qu'il aura reçu le message d'extinction. Donc, nous sommes maintenant assurés que si nous envoyons autant de messages d'extinction qu'il y a d'opérateurs, chaque opérateur recevra un message d'extinction avant que `join` ne soit utilisé sur leur tâche.

Pour observer ce code en action, modifions notre `main` pour accepter uniquement deux requêtes avant d'arrêter proprement le serveur, comme dans l'encart 20-25.

Fichier : src/bin/main.rs

```
fn main() {
    let ecouteur = TcpListener::bind("127.0.0.1:7878").unwrap();
    let groupe = GroupeTaches::new(4);

    for flux in ecouteur.incoming().take(2) {
        let flux = flux.unwrap();

        groupe.executer(|| {
            gestion_connexion(flux);
        });
    }

    println!("Arrêt complet.");
}
```

Encart 20-25 : arrêt du serveur après avoir servi deux requêtes en sortant de la boucle

Dans la réalité on ne voudrait pas qu'un serveur web s'arrête après avoir servi seulement deux requêtes. Ce code sert uniquement à montrer que l'arrêt et le nettoyage s'effectuent bien proprement.

La méthode `take` est définie dans le trait `Iterator` et limite l'itération aux deux premiers éléments au maximum. Le `GroupeTaches` va sortir de la portée à la fin du `main` et

l'implémentation de `drop` va s'exécuter.

Démarrez le serveur avec `cargo run` et faites trois requêtes. La troisième requête devrait renvoyer une erreur tandis que dans votre terminal vous devriez avoir une sortie similaire à ceci :

```
$ cargo run
   Compiling salutations v0.1.0 (file:///projects/salutations)
    Finished dev [unoptimized + debuginfo] target(s) in 1.0s
     Running `target/debug/main`
L'opérateur 0 a reçu une mission ; il l'exécute.
L'opérateur 3 a reçu une mission ; il l'exécute.
Arrêt.
Envoi du message d'extinction à tous les opérateurs.
Arrêt de tous les opérateurs.
Arrêt de l'opérateur 0
L'opérateur 1 a reçu l'instruction d'arrêt.
L'opérateur 2 a reçu l'instruction d'arrêt.
L'opérateur 0 a reçu l'instruction d'arrêt.
L'opérateur 3 a reçu l'instruction d'arrêt.
Arrêt de l'opérateur 1
Arrêt de l'opérateur 2
Arrêt de l'opérateur 3
```

Vous pourriez avoir un ordre différent entre les opérateurs et les messages affichés. Nous pouvons constater la façon dont ce code fonctionne grâce aux messages : les opérateurs 0 et 3 obtiennent les deux premières requêtes puis, à la troisième requête, le serveur arrête d'accepter des connexions. Lorsque le `GroupeTaches` sort de la portée à la fin du `main`, son implémentation de `Drop` entre en action et le groupe demande à tous les opérateurs de s'arrêter. Chaque opérateur va afficher un message lorsqu'il recevra le message d'extinction puis le groupe de tâche utilisera `join` pour arrêter la tâche de chaque opérateur.

Remarquez un aspect intéressant spécifique à cette exécution : le `GroupeTaches` a envoyé les messages d'extinction dans le canal, et avant que tous les opérateurs aient reçu les messages, nous avons essayé d'utiliser `join` sur l'opérateur 0. L'opérateur 0 n'avait pas encore reçu le message d'extinction, donc la tâche principale a attendu que l'opérateur 0 finisse. Pendant ce temps, tous les autres opérateurs ont reçu les messages d'extinction. Lorsque l'opérateur 0 a fini, la tâche principale a attendu que les autres opérateurs se terminent. A ce stade, ils avaient alors tous reçu le message d'extinction et étaient en mesure de s'arrêter.

Félicitations ! Nous avons maintenant terminé notre projet ; nous avons un serveur web basique qui utilise un groupe de tâches pour répondre de manière asynchrone. Nous pouvons demander un arrêt propre du serveur qui va nettoyer toutes les tâches du groupe.

Voici le code complet afin que vous puissiez vous y référer :

Fichier : src/bin/main.rs

```
use salutations::GroupeTaches;
use std::fs;
use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;
use std::thread;
use std::time::Duration;

fn main() {
    let ecouteur = TcpListener::bind("127.0.0.1:7878").unwrap();
    let groupe = GroupeTaches::new(4);

    for flux in ecouteur.incoming() {
        let flux = flux.unwrap();

        groupe.executer(|| {
            gestion_connexion(flux);
        });
    }

    println!("Shutting down.");
}

fn gestion_connexion(mut flux: TcpStream) {
    let mut tampon = [0; 1024];
    flux.read(&mut tampon).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let pause = b"GET /pause HTTP/1.1\r\n";

    let (ligne_statut, nom_fichier) = if tampon.starts_with(get) {
        ("HTTP/1.1 200 OK", "hello.html")
    } else if tampon.starts_with(pause) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contenu = fs::read_to_string(nom_fichier).unwrap();

    let reponse = format!(
        "{}\r\nContent-Length: {}\r\n\r\n{}",
        ligne_statut,
        contenu.len(),
        contenu
    );

    flux.write(reponse.as_bytes()).unwrap();
    flux.flush().unwrap();
}
```

Fichier : src/lib.rs

```rust
use std::sync::mpsc;
use std::sync::Arc;
use std::sync::Mutex;
use std::thread;

pub struct GroupeTaches {
    operateurs: Vec<Operateur>,
    envoi: mpsc::Sender<Message>,
}

type Mission = Box<dyn FnOnce() + Send + 'static>;

enum Message {
    NouvelleMission(Mission),
    Extinction,
}

impl GroupeTaches {
    /// Crée un nouveau GroupeTaches.
    ///
    /// La taille est le nom de tâches présentes dans le groupe.
    ///
    /// # Panics
    ///
    /// La fonction `new` devrait paniquer si la taille vaut zéro.
    pub fn new(taille: usize) -> GroupeTaches {
        assert!(taille > 0);

        let (envoi, reception) = mpsc::channel();

        let reception = Arc::new(Mutex::new(reception));

        let mut operateurs = Vec::with_capacity(taille);

        for id in 0..taille {
            operateurs.push(Operateur::new(id, Arc::clone(&reception)));
        }

        GroupeTaches { operateurs, envoi }
    }

    pub fn executer<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let mission = Box::new(f);

        self.envoi.send(Message::NouvelleMission(mission)).unwrap();
    }
}

impl Drop for GroupeTaches {
```

```
    fn drop(&mut self) {
        println!("Envoi du message d'extinction à tous les opérateurs.");

        for _ in &self.operateurs {
            self.envoi.send(Message::Extinction).unwrap();
        }

        println!("Arrêt de tous les opérateurs.");

        for operateur in &mut self.operateurs {
            println!("Arrêt de l'opérateur {}", operateur.id);

            if let Some(tache) = operateur.tache.take() {
                tache.join().unwrap();
            }
        }
    }
}

struct Operateur {
    id: usize,
    tache: Option<thread::JoinHandle<()>>,
}

impl Operateur {
    fn new(id: usize, reception: Arc<Mutex<mpsc::Receiver<Message>>>) ->
Operateur {
        let tache = thread::spawn(move || loop {
            let message = reception.lock().unwrap().recv().unwrap();

            match message {
                Message::NouvelleMission(mission) => {
                    println!("L'opérateur {} a reçu une mission ; il
l'exécute.", id);

                    mission();
                }
                Message::Extinction => {
                    println!("L'opérateur {} a reçu l'instruction d'arrêt.",
id);

                    break;
                }
            }
        });

        Operateur {
            id,
            tache: Some(tache),
        }
    }
}
```

Nous aurions pu faire bien plus ! Si vous souhaitez continuer à améliorer ce projet, voici quelques idées :

- Ajouter de la documentation à `GroupeTaches` et aux méthodes publiques.
- Ajouter des tests sur les fonctionnalités de la bibliothèque.
- Remplacer les appels à `unwrap` pour fournir une meilleure gestion des erreurs.
- Utiliser `GroupeTaches` pour exécuter d'autres tâches que de répondre à des requêtes web.
- Trouver une crate de groupe de tâches *(NdT : thread pool)* sur crates.io et implémenter un serveur web similaire en l'utilisant. Comparer ensuite son API et sa robustesse au groupe de tâches que nous avons implémenté.

Résumé

Bravo ! Vous êtes arrivé à la fin du livre ! Nous tenons à vous remercier chaleureusement de nous avoir accompagné pendant cette présentation de Rust. Vous êtes maintenant fin prêt(e) à créer vos propres projets Rust et aider les projets des autres développeurs. Rappelez-vous qu'il existe une communauté accueillante de Rustacés qui adorerait vous aider à relever tous les défis que vous rencontrerez dans votre aventure avec Rust.

Annexes

Les sections suivantes contiennent des informations de référence que vous pourriez trouver utile pour votre exploration de Rust.

Annexe A : les mots-clés

La liste suivante contient des mots-clés réservés qui sont actuellement utilisés dans le langage Rust ou qui pourraient l'être à l'avenir. De ce fait, ils ne peuvent pas être utilisés comme identificateurs (sauf comme identificateurs bruts, ce que nous allons voir dans la section “les identificateurs bruts”), y compris pour les noms de fonctions, de variables, de paramètres, de champs de structures, de modules, de crates, de constantes, de macros, de valeurs statiques, d'attributs, de types, de traits ou de durées de vie.

Les mots-clés actuellement utilisés

Les mots-clés suivants ont actuellement la fonction décrite.

- `as` - effectue une transformation de type primitive, précise le trait qui contient un élément ou renomme des éléments dans les instructions `use` et `extern crate`
- `async` - retourne un `Future` plutôt que de bloquer la tâche en cours
- `await` - met en pause l'exécution jusqu'à ce que le résultat d'un `Future` soit disponible
- `break` - sort immédiatement d'une boucle
- `const` - définit des éléments constants ou des pointeurs bruts constants
- `continue` - passe directement à la prochaine itération de la boucle en cours
- `crate` - crée un lien vers une crate externe ou une variable de macro qui représente la crate dans laquelle la macro est définie
- `dyn` - utilisation dynamique d'un objet trait
- `else` - une branche de repli pour les structures de contrôle de flux `if` et `if let`
- `enum` - définit une énumération
- `extern` - crée un lien vers une crate, une fonction ou une variable externe
- `false` - le littéral qui vaut “faux” pour un booléen
- `fn` - définit une fonction ou le type pointeur de fonction
- `for` - crée une boucle sur les éléments d'un itérateur, implémente un trait, ou renseigne une durée de vie de niveau supérieur
- `if` - une branche liée au résultat d'une expression conditionnelle
- `impl` - implémente des fonctionnalités propres à un élément ou à un trait
- `in` - fait partie de la syntaxe de la boucle `for`
- `let` - lie une valeur à une variable
- `loop` - fait une boucle sans condition (théoriquement infinie)
- `match` - compare une valeur à des motifs
- `mod` - définit un module
- `move` - fait en sorte qu'une fermeture prenne possession de tout ce qu'elle utilise

- `mut` - autorise la mutabilité sur des références, des pointeurs bruts ou des éléments issus de motifs
- `pub` - autorise la visibilité publique sur des champs de structures, des blocs `impl` ou des modules
- `ref` - lie une valeur avec une référence
- `return` - retourne une valeur depuis une fonction
- `Self` - un alias de type pour le type que nous définissons ou implémentons
- `self` - désigne le sujet d'une méthode ou du module courant
- `static` - une variable globale ou une durée de vie qui persiste tout au long de l'exécution du programme
- `struct` - définit une structure
- `super` - le module parent du module courant
- `trait` - définit un trait
- `true` - le littéral qui vaut “vrai” pour un booléen
- `type` - définit un alias de type ou un type associé
- `union` - définit une union mais n'est un mot-clé que lorsqu'il est utilisé dans la déclaration d'une union
- `unsafe` - autorise du code, des fonctions, des traits ou des implémentations non sécurisées
- `use` - importe des éléments dans la portée
- `where` - indique des conditions pour contraindre un type
- `while` - crée une boucle en fonction des résultats d'une expression

Les mots-clés réservés pour une utilisation future

Les mots-clés suivants n'offrent actuellement aucune fonctionnalité mais sont réservés par Rust pour une potentielle utilisation future.

- `abstract`
- `become`
- `box`
- `do`
- `final`
- `macro`
- `override`
- `priv`
- `try`
- `typeof`
- `unsized`

- `virtual`
- `yield`

Les identificateurs bruts

Un *identificateur brut* est une syntaxe qui vous permet d'utiliser des mots-clés là où ils ne devraient pas pouvoir l'être. Vous pouvez utiliser un identificateur brut en faisant précéder un mot-clé par un `r#`.

Par exemple, `match` est un mot-clé. Si vous essayez de compiler la fonction suivante qui utilise `match` comme nom :

Fichier : src/main.rs

```
fn match(aiguille: &str, botte_de_foin: &str) -> bool {
    botte_de_foin.contains(aiguille)
}
```

... vous allez obtenir l'erreur suivante :

```
error: expected identifier, found keyword `match`
 -- > src/main.rs:4:4
  |
4 | fn match(aiguille: &str, botte_de_foin: &str) -> bool {
  |    ^^^^^ expected identifier, found keyword
```

L'erreur montre que vous ne pouvez pas utiliser le mot-clé `match` comme identificateur de la fonction. Pour utiliser `match` comme nom de fonction, vous devez utiliser la syntaxe d'identificateur brut, comme ceci :

Fichier : src/main.rs

```
fn r#match(aiguille: &str, botte_de_foin: &str) -> bool {
    botte_de_foin.contains(aiguille)
}

fn main() {
    assert!(r#match("rem", "lorem ipsum"));
}
```

Ce code va se compiler sans erreur. Remarquez le préfixe `r#` sur le nom de la fonction dans sa définition mais aussi lorsque cette fonction est appelée dans `main`.

Les identificateurs bruts vous permettent d'utiliser n'importe quel mot de votre choix

comme identificateur, même si ce mot est un mot-clé réservé. De plus, les identificateurs bruts vous permettent d'utiliser des bibliothèques écrites dans des éditions de Rust différentes de celle qu'utilise votre crate. Par exemple, `try` n'est pas un mot-clé dans l'édition 2015, mais il l'est dans l'édition 2018. Si vous dépendez d'une bibliothèque qui était écrite avec l'édition 2015 et qui avait une fonction `try`, vous allez avoir besoin dans ce cas d'utiliser la syntaxe d'identificateur brut `r#try` pour faire appel à cette fonction à partir de code écrit avec l'édition 2018. Voir l'annexe E pour en savoir plus sur les éditions.

🚧 Attention, peinture fraîche !

Cette page a été traduite par une seule personne et n'a pas été relue et vérifiée par quelqu'un d'autre ! Les informations peuvent par exemple être erronées, être formulées maladroitement, ou contenir d'autres types de fautes.

Vous pouvez contribuer à l'amélioration de cette page sur sa Pull Request.

Annexe B : les opérateurs et les symboles

Cette annexe contient un glossaire de syntaxes Rust, comprenant les opérateurs et les autres symboles qui s'utilisent tout seuls ou alors dans le cadre de chemins, de génériques, de traits liés, de macros, d'attributs, de commentaires, de tuples, de crochets ou d'accolades.

Opérateurs

Le tableau B-1 contient une liste d'opérateurs en Rust, un exemple de comment l'opérateur devrait être utilisé dans ce contexte, une petite explication, et si cet opérateur est surchargeable. Si un opérateur est surchargeable, le trait concerné à utiliser pour la surcharge est indiqué.

Tableau B-1 : les opérateurs

Opérateur	Exemple	Explication	Surchargeable ?
`!`	`ident!(...)`, `ident!{...}`, `ident![...]`	Identificateur de macro	
`!`	`!expr`	Négation binaire ou logique	`Not`
`!=`	`var != expr`	Comparaison de non-égalité	`PartialEq`
`%`	`expr % expr`	Reste arithmétique	`Rem`

Opérateur	Exemple	Explication	Surchargeable ?
`%=`	`var %= expr`	Reste arithmétique et assignation	`RemAssign`
`&`	`&expr`, `&mut expr`	Emprunt	
`&`	`&type`, `&mut type`, `&'a type`, `&'a mut type`	Type de pointeur emprunté	
`&`	`expr & expr`	ET binaire	`BitAnd`
`&=`	`var &= expr`	ET binaire et assignation	`BitAndAssign`
`&&`	`expr && expr`	ET logique	
`*`	`expr * expr`	Multiplication arithmétique	`Mul`
`*=`	`var *= expr`	Multiplication arithmétique et assignation	`MulAssign`
`*`	`*expr`	Déréférencement	
`*`	`*const type`, `*mut type`	Pointeur brut	
`+`	`trait + trait`, `'a + trait`	Contrainte de type composé	
`+`	`expr + expr`	Addition arithmétique	`Add`
`+=`	`var += expr`	Addition arithmétique et assignation	`AddAssign`
`,`	`expr, expr`	Séparateur d'arguments et d'éléments	
`-`	`- expr`	Négation arithmétique	`Neg`
`-`	`expr - expr`	Soustraction arithmétique	`Sub`
`-=`	`var -= expr`	Soustraction arithmétique et assignation	`SubAssign`
`->`	`fn(...) -> type`, `|...| -> type`	Type de retour de fonction et de fermeture	
`.`	`expr.ident`	Accès à un membre	

Opérateur	Exemple	Explication	Surchargeable ?
`..`	`..`, `expr..`, `..expr`, `expr..expr`	Littéral d'intervalle d'exclusion	
`..=`	`..=expr`, `expr..=expr`	Littéral d'intervalle d'inclusion	
`..`	`..expr`	Syntaxe de mise à jour de litéraux de structure	
`..`	`variant(x, ..)`, `struct_type { x, .. }`	Motif “ainsi que la suite”	
`...`	`expr...expr`	Dans un motif : motif d'intervalle inclusif	
`/`	`expr / expr`	Division arithmétique	`Div`
`/=`	`var /= expr`	Division arithmétique et assignation	`DivAssign`
`:`	`pat: type`, `ident: type`	Contrainte	
`:`	`ident: expr`	Initialisateur de champ de structure	
`:`	`'a: loop {...}`	Une identification de boucle	
`;`	`expr;`	Fin d'élément et d'instruction	
`;`	`[...; len]`	Syntaxe désignant une partie d'un tableau à taille finie	
`<<`	`expr << expr`	Décalage à gauche	`Shl`
`<<=`	`var <<= expr`	Décalage à gauche et assignation	`ShlAssign`
`<`	`expr < expr`	Comparaison "inférieur à"	`PartialOrd`
`<=`	`expr <= expr`	Comparaison "inférieur ou égal à"	`PartialOrd`
`=`	`var = expr`, `ident = type`	Assignation ou équivalence	

Opérateur	Exemple	Explication	Surchargeable ?
`==`	`expr == expr`	Comparaison d'égalité	`PartialEq`
`=>`	`pat => expr`	Syntaxe d'une partie d'une branche correspondante	
`>`	`expr > expr`	Comparaison "supérieur à"	`PartialOrd`
`>=`	`expr >= expr`	Comparaison "supérieur ou égal à"	`PartialOrd`
`>>`	`expr >> expr`	Décalage à droite	`Shr`
`>>=`	`var >>= expr`	Décalage à droite et assignation	`ShrAssign`
`@`	`ident @ pat`	Création d'un identificateur à partir du motif	
`^`	`expr ^ expr`	OU exclusif binaire	`BitXor`
`^=`	`var ^= expr`	OU exclusif binaire et assignation	`BitXorAssign`
`|`	`pat | pat`	Alternatives à un motif	
`|`	`expr | expr`	OU binaire	`BitOr`
`|=`	`var |= expr`	OU binaire et assignation	`BitOrAssign`
`||`	`expr || expr`	OU logique	
`?`	`expr?`	Propagation d'erreur	

Les symboles non-opérateurs

La liste suivante contient tout ce qui n'est pas une lettre et qui ne fonctionne pas comme un opérateur ; autrement dit tout ce qui ne se comporte pas comme un appel de fonction ou de méthode.

Le tableau B-2 montre des symboles qui s'utilisent tout seuls et qui sont valables dans plusieurs situations.

Tableau B-2 : syntaxes autonomes

Symbole	Explication

Symbole	Explication
`'ident`	Nom d'une durée de vie ou nom boucle
`...u8`, `...i32`, `...f64`, `...usize`, etc.	Nombre littéral d'un type spécifique
`"..."`	Chaîne de caractère littérale
`r"..."`, `r#"..."#`, `r##"..."##`, etc.	Chaîne de caractères brute littérale, les caractères d'échappement ne sont pas traités
`b"..."`	Chaîne d'octet littéral ; construit un `[u8]` au lieu d'une chaîne de caractères
`br"..."`, `br#"..."#`, `br##"..."##`, etc.	Chaîne d'octets brute littérale, combinaison de la chaîne d'octets brute et de la chaîne d'octets littérale
`'...'`	Caractère littéral
`b'...'`	Octet ASCII littéral
`|...| expr`	Une fermeture
`!`	Le type “jamais", toujours vide pour les fonctions divergentes
`_`	Le motif “ignoré" ; aussi utilisé pour rendre lisibles les nombres entiers littéraux

Le tableau B-3 montre des symboles qui s'utilisent dans le contexte d'un chemin dans une structure de modules pour obtenir un élément.

Tableau B-3 : syntaxes utilisés pour les chemins

Symbole	Explication
`ident::ident`	Chemin d'un espace de nom
`::path`	Chemin relatif à la crate racine (c'est à dire un chemin explicitement absolu)
`self::path`	Chemin relatif au module courant (c'est à dire un chemin explicitement relatif)
`super::path`	Chemin relatif au parent du module courant
`type::ident`, `<type as trait>::ident`	Des constantes, fonctions et types associées
`<type>::...`	Un élément associé pour un type qui ne peut pas être directement nommé (par exemple, `<&T>::...`, `<[T]>::...`, etc)
`trait::method(...)`	Clarifier l'appel d'une méthode en nommant le trait qui le définit

Symbole	Explication
`type::method(...)`	Clarifier l'appel d'une fonction en nommant le type pour laquelle elle est définie
`<type as trait>::method(...)`	Clarifier l'appel d'une méthode en nommant le trait et le type

Le tableau B-4 montre des symboles qui apparaissent dans le contexte d'utilisation de paramètres de type génériques.

Tableau B-4 : génériques

Symbole	Explication
`path<...>`	Précise des paramètres sur un type générique utilisé dans un type (par exemple, `Vec<u8>`)
`path::<...>`, `method::<...>`	Précise des paramètres sur un type générique, une fonction, ou une méthode dans une expression ; parfois appelé turbofish (par exemple, `"42".parse::<i32>()`)
`fn ident<...> ...`	Définit une fonction générique
`struct ident<...> ...`	Définit une structure générique
`enum ident<...> ...`	Définit une énumération générique
`impl<...> ...`	Définit une implémentation générique
`for<...> type`	Augmente la durée de vie
`type<ident=type>`	Un type générique sur lequel un ou plusieurs types associés ont des affectations spécifiques (par exemple, `Iterator<Item=T>`)

Le tableau B-5 montre des symboles qui s'utilisent pour contraindre des paramètres de type génériques avec des traits liés.

Tableau B-5 : contraintes de trait lié

Symbole	Explication
`T: U`	Paramètre générique `T` contraint aux types qui implémentent `U`
`T: 'a`	Type générique `T` doit vivre aussi longtemps que la durée de vie `'a` (ce qui signifie que le type ne peut pas contenir temporairement de références avec une durée de vie plus petite que `'a`)

Symbole	Explication
`T : 'static`	Type générique `T` qui ne contient pas d'autres références empruntées autres que des `'static`
`'b: 'a`	La durée de vie générique `'b` doit vivre aussi longtemps que `'a`
`T: ?Sized`	Permet aux paramètres de type génériques d'être de type à taille dynamique
`'a + trait, trait + trait`	Contrainte de type composé

Le tableau B-6 montre des symboles qui s'utilisent lors de l'appel ou de la définition de macros et pour spécifier des attributs sur un élément.

Tableau B-6 : macros et attributs

Symbole	Explication
`#[meta]`	Attribut externe
`#![meta]`	Attribut interne
`$ident`	Substitution de macro
`$ident:kind`	Capture de macro
`$(…)…`	Répétition de macro
`ident!(...)`, `ident!{...}`, `ident![...]`	Appel d'une macro

Le tableau B-7 montre des symboles pour créer des commentaires.

Tableau B-7 : commentaires

Symbole	Explication
`//`	Ligne commentée
`//!`	Commentaire de documentation sur l'élément contenant actuel
`///`	Commentaire de documentation sur l'élément suivant ce commentaire
`/*...*/`	Bloc de commentaire
`/*!...*/`	Bloc de commentaire de documentation sur l'élément contenant actuel
`/**...*/`	Bloc de commentaire de documentation sur l'élément suivant ce commentaire

Le tableau B-8 montre des symboles utilisés avec les tuples.

Tableau B-8 : les tuples

Symbole	Explication
`()`	Un tuple vide (aussi appelé unitaire), à la fois un type et un litéral
`(expr)`	Une expression entre parenthèses
`(expr,)`	Un tuple d'un seul élement qui est une expression
`(type,)`	Un tuple d'un seul élement qui est un type
`(expr, ...)`	Une expression dans un tuple
`(type, ...)`	Un type dans un tuple
`expr(expr, ...)`	Une expression d'appel à une fonction ; aussi utilisé pour initialiser une structure tuple ou une variante d'énumération tuple
`expr.0`, `expr.1`, etc.	Utilisation d'indices sur un tuple

Le tableau B-9 montre les contextes d'utilisation des accolades.

Tableau B-9 : accolades

Symbole	Explication
`{...}`	Bloc d'expression
`Type {...}`	Un littéral de `struct`

Le tableau B-10 montre les contextes d'utilisation des crochets.

Tableau B-10 : crochets

Symbole	Explication
`[...]`	Un littéral de tableau
`[expr; len]`	Un littéral de tableau qui contient `len` copies de `expr`
`[type; len]`	Un type de tableau qui contient `len` instances de `type`
`expr[expr]`	Une collection indexée. C'est surchargeable (via `Index` et `IndexMut`)
`expr[..]`, `expr[a..]`, `expr[..b]`, `expr[a..b]`	Une collection indexée qui se comporte comme une slice de collection, grâce à l'utilisation de `Range`, `RangeFrom`, `RangeTo`, ou de `RangeFull` comme “indice”

🚧 Attention, peinture fraîche !

Cette page a été traduite par une seule personne et n'a pas été relue et vérifiée par quelqu'un d'autre ! Les informations peuvent par exemple être erronées, être formulées maladroitement, ou contenir d'autres types de fautes.

Vous pouvez contribuer à l'amélioration de cette page sur sa Pull Request.

Annexe C : les traits dérivables

Dans de nombreux endroits du livre, nous avons vu l'attribut `derive`, que vous pouvez appliquer à une définition de structure ou d'énumération. L'attribut `derive` génère du code qui va implémenter un trait avec sa propre implémentation par défaut sur le type que vous avez annoté avec la syntaxe `derive`.

Dans cette annexe, nous allons produire une référence de tous les traits de la bibliothèque standard que vous pouvez utiliser avec `derive`. Chaque section va donner :

- Quels opérateurs et méthodes seront activés en dérivant de ce trait
- Ce que fait l'implémentation du trait appliqué par le `derive`
- Ce que l'implémentation du trait implique sur le type concerné
- Les conditions dans lesquelles vous pouvez ou non implémenter le trait
- Des exemples d'opérations qui nécessitent que le trait soit implémenté

Si vous souhaitez appliquer un comportement différent de celui fourni par l'attribut `derive`, consultez la documentation de la bibliothèque standard pour le trait concerné afin d'en savoir plus sur son implémentation manuelle.

Le reste des traits définis dans la bibliothèque standard ne peuvent pas être implémentés sur des types en utilisant `derive`. Ces traits n'ont pas de comportement logique par défaut, donc c'est à vous de les implémenter de la façon la plus appropriée pour ce que vous souhaitez accomplir.

Un exemple de trait qui ne peut pas être dérivé est `Display`, qui permet de formater la donnée pour les utilisateurs finaux. Vous devez toujours réfléchir au formatage du type le plus approprié pour un utilisateur final. Quelles parties d'un type un utilisateur final devrait pouvoir voir ? Sous quelle forme les données devraient être les plus intéressantes pour eux ? Le compilateur de Rust n'a pas cette intuition, donc il ne peut pas fournir un

comportement par défaut à votre place.

La liste des traits dérivables fournis dans cette annexe n'est pas exhaustive : les bibliothèques peuvent implémenter `derive` pour leurs propres traits, étendant potentiellement à l'infini la liste de traits que vous pouvez utiliser avec `derive`. L'implémentation de `derive` implique l'utilisation d'une macro procédurale, que nous avons vu dans une section du chapitre 19.

`Debug` pour l'affichage au développeur

Le trait `Debug` permet le formatage de déboguage pour mettre en forme en tant que chaînes de caractères, que vous pouvez utiliser en ajoutant `:?` dans un espace réservé `{}`.

Le trait `Debug` vous permet d'afficher des instances d'un type pour des besoins de déboguage, afin que vous et les autres développeurs qui utilisent votre type puissent inspecter une de ses instances à un endroit précis de l'exécution du programme.

Le trait `Debug` est nécessaire, par exemple, pour l'utilisation de la macro `assert_eq!`. Cette macro affiche les valeurs des instances passées en argument dans le cas où l'affirmation échoue afin que le développeur puisse voir pourquoi les deux instances ne sont pas égales.

`PartialEq` et `Eq` pour comparer l'égalité

Le trait `PartialEq` vous permet de comparer des instances d'un type pour vérifier leur égalité et permet l'utilisation des opérateurs `==` et `!=`.

L'application de `derive` avec `PartialEq` implémente la méthode `eq`. Lorsque `PartialEq` est dérivé sur une structure, deux instances ne peuvent être égales seulement si *tous* leurs champs sont égaux, et les instances ne sont pas égales si un des champs n'est pas égal. Lorsque ce trait est dérivé sur une énumération, chaque variante est égale à elle-même et n'est pas égale aux autres variantes.

Le trait `Eq` est nécessaire, par exemple, pour utiliser la macro `assert_eq!`, qui nécessite de pouvoir comparer l'égalité de deux instances d'un type.

Le trait `Eq` n'a pas de méthode. Son rôle est de signaler que pour chaque valeur du type annoté, la valeur est égale à elle-même. Le trait `Eq` peut seulement être appliqué sur des types qui implémentent `PartialEq`, bien que tous les types qui implémentent `PartialEq` ne puissent pas implémenter `Eq`. Un exemple de ceci sont les types de nombres à virgule flottante : l'implémentation des nombres à virgule flottante stipule que deux instances ayant

la valeur “not-a-number” (`NaN` , c'est-à-dire “ceci n'est pas un nombre”) ne sont pas égales entre elles.

Par exemple, `Eq` est nécessaire est pour les clés dans un `HashMap<K, V>` afin que le `HashMap<K, V>` puisse déterminer si deux clés sont identiques.

`PartialOrd` et `Ord` pour comparer les ordres de grandeur

Le trait `PartialOrd` vous permet de comparer des instances d'un type pour pouvoir les trier. Un type qui implémente `PartialOrd` peut être utilisé avec les opérateurs `<` , `>` , `<=` , et `>=` . Vous pouvez appliquer uniquement le trait `PartialOrd` aux types qui implémentent aussi `PartialEq` .

L'application de `derive` avec `PartialOrd` implémente la méthode `partial_cmp` , qui retourne un `Option<Ordering>` qui vaudra `None` lorsque les valeurs fournies ne fournissent pas un ordre. Un exemple de valeur qui ne produit pas d'ordre, même si la plupart des valeurs de ce type peuvent être comparées, est la valeur “not-a-number” (`NaN`) des virgules flottantes. L'appel à `partial_cmp` entre n'importe quel nombre à virgule flottante et la valeur `NaN` de virgule flottante va retourner `None` .

Lorsqu'il est dérivé sur une structure, `PartialOrd` compare deux instances en comparant les valeurs de chaque champ dans l'ordre dans lequel les champs apparaissent dans la définition de la structure. Lorsqu'il est dérivé sur des énumérations, les variantes de l'énumération déclarées plus tôt dans la définition de l'énumération sont considérées inférieures aux variantes déclarées ensuite.

Le trait `PartialOrd` est nécessaire, par exemple, pour la méthode `gen_range` de la crate `rand` qui génère une valeur aléatoire dans l'intervalle contrainte par une valeur minimale et une valeur maximale.

Le trait `Ord` vous permet de savoir si un ordre valide existe toujours entre deux valeurs du type annoté. Le trait `Ord` implémente la méthode `cmp` , qui retourne un `Ordering` plutôt qu'une `Option<Ordering>` car un ordre valide sera toujours possible. Vous pouvez appliquer le trait `Ord` uniquement sur les types qui implémentent aussi `PartialOrd` et `Eq` (et `Eq` nécessite `PartialEq`). Lorsqu'il est dérivé sur des structures et des énumérations, `cmp` se comporte de la même manière que l'implémentation de `partial_cmp` dérivée de `PartialOrd` .

Par exemple, `Ord` doit être implémenté sur le type de valeurs que nous stockons dans un `BTreeSet<T>` , qui est une structure de donnée qui stocke des données en fonction de

l'ordre de tri de ces valeurs.

`Clone` et `Copy` pour dupliquer des valeurs

Le trait `Clone` vous permet de créer explicitement une copie profonde d'une valeur, et le processus de duplication peut impliquer l'exécution d'un code arbitraire pour copier les données stockées dans le tas. Rendez-vous à la section "Les interactions entre les variables et les données : le déplacement" du chapitre 4 pour plus d'informations sur `Clone` .

Utiliser `derive` avec `Clone` implémente la méthode `clone` , qui, lorsqu'elle est implémentée sur tout le type, fait appel à `clone` sur chaque constituant du type. Cela signifie que tous les champs ou les valeurs dans le type doivent aussi implémenter `Clone` pour dériver de `Clone` .

`Clone` est par exemple nécessaire lorsque nous appelons la méthode `to_vec` sur une slice. La slice ne prend pas possession des instances du type qu'il contient, mais le vecteur retourné par `to_vec` va avoir besoin de prendre possession de ses instances, donc `to_vec` fait appel à `clone` sur chaque élément. C'est pourquoi le type stocké dans la slice doit implémenter `Clone` .

Le trait `Copy` vous permet de dupliquer une valeur en copiant uniquement les éléments stockés sur la pile ; il n'est pas nécessaire d'avoir de code arbitraire. Rendez-vous à la section "Données uniquement sur la pile : la copie" du chapitre 4 pour plus d'informations sur `Copy` .

Le trait `Copy` ne définit pas de méthode, volontairement pour empêcher les développeurs de surcharger ces méthodes et ainsi violer l'affirmation qu'aucun code arbitraire est exécuté à la copie. Ainsi, tous les développeurs peuvent compter sur le fait qu'une copie de valeur est très rapide.

Vous pouvez utiliser `derive` avec `Copy` sur n'importe quel type constitué d'éléments qui implémentent aussi `Copy` . Vous ne pouvez appliquer le trait `Copy` que sur des types qui implémentent aussi `Clone` , car un type qui implémente `Copy` a aussi une implémentation triviale de `Clone` qui procède aux mêmes actions que `Copy` .

Le trait `Copy` est rarement nécessaire ; les types qui implémentent `Copy` peuvent être optimisés, ce qui veut dire que vous n'avez pas à appeler `clone` , ce qui rend le code plus concis.

Tout ce que vous pouvez accomplir avec `Copy` , vous pouvez le faire avec `Clone` , mais le code risque d'être plus lent ou doit parfois utiliser `clone` .

`Hash` pour faire correspondre une valeur avec une valeur de taille fixe

Le trait `Hash` vous permet d'obtenir une valeur à taille fixe en utilisant une fonction de hachage sur une instance d'un type d'une taille quelconque. Utiliser `derive` avec `Hash` implémente la méthode `hash`. L'implémentation dérive de la méthode `hash` combine le résultat de l'appel de `hash` sur chaque élément du type, ce qui signifie que tous ses champs ou valeurs doivent aussi implémenter `Hash` pour pouvoir lui appliquer le trait `Hash`.

Pour stocker des clés efficacement dans un `HashMap<K, V>`, les clés doivent nécessairement implémenter `Hash`.

`Default` pour des valeurs par défaut

Le trait `Default` vous permet de créer une valeur par défaut pour un type. Implémenter `Default` avec `derive` ajoute la fonction `default`. Cette fonction `default` fait elle-même appel à la fonction `default` sur chaque élément du type, ce qui signifie que tous les champs ou les valeurs dans le type doit aussi implémenter `Default` pour que ce type puisse dériver de `Default`.

La fonction `Default::default` est couramment utilisé en association avec la syntaxe de modification de structures que nous avons vu dans la section "Créer des instances à partir d'autres instances avec la syntaxe de mise à jour de structure" du chapitre 5. Vous pouvez personnaliser quelques champs d'une structure et ensuite définir et utiliser une valeur par défaut pour le reste des champs en utilisant `..Default::default()`.

Le trait `Default` est nécessaire lorsque vous utilisez la méthode `unwrap_or_default` sur les instances de `Option<T>`, par exemple. Si le `Option<T>` vaut `None`, la méthode `unwrap_or_default` va retourner le résultat de `Default::default` sur le type `T` provenant du `Option<T>`.

🚧 Attention, peinture fraîche !

Cette page a été traduite par une seule personne et n'a pas été relue et vérifiée par quelqu'un d'autre ! Les informations peuvent par exemple être erronées, être formulées maladroitement, ou contenir d'autres types de fautes.

Vous pouvez contribuer à l'amélioration de cette page sur sa Pull Request.

Annexe D - Des outils de développement utiles

Dans cette annexe, nous allons découvrir quelques outils de développement utiles que propose le projet Rust. Nous allons voir le formatage automatique, des moyens rapides pour corriger des avertissements, un analyseur statique, et l'intégration avec un IDE.

Le formatage automatique avec `rustfmt`

L'outil `rustfmt` reformate votre code suivant le style de code de la communauté. De nombreux projets collaboratifs utilisent `rustfmt` pour éviter des désaccords sur le style à utiliser lorsqu'ils écrivent du code Rust : tout le monde formate leur code en utilisant l'outil.

Pour installer `rustfmt`, saisissez ceci :

```
$ rustup component add rustfmt
```

Cette commande vous offre `rustfmt` et `cargo-fmt`, de la même manière que Rust vous installe `rustc` et `cargo`. Pour formater un projet Cargo, saisissez ceci :

```
$ cargo fmt
```

L'exécution de cette commande reformate tout le code Rust dans la crate courante. Cela va uniquement changer le style de code, pas sa sémantique. Pour plus d'informations sur `rustfmt`, voyez sa documentation.

Corriger votre code avec `rustfix`

L'outil rustfix est inclus lors de l'installation de Rust et peut automatiquement corriger certains avertissements de compilateur. Si vous avez déjà écrit du code en Rust, vous avez probablement vu des avertissements du compilateur. Par exemple, avec le code suivant :

Fichier : src/main.rs

```
fn fait_quelquechose() {}

fn main() {
    for i in 0..100 {
        fait_quelquechose();
    }
}
```

Ici, nous appelons la fonction `fait_quelquechose` 100 fois, mais nous n'utilisons jamais la variable `i` dans le corps de la boucle `for` . Rust nous avertit de cela :

```
$ cargo build
   Compiling mon_programme v0.1.0 (file:///projects/mon_programme)
warning: unused variable: `i`
 -- > src/main.rs:4:9
  |
4 |     for i in 1..100 {
  |         ^ help: consider using `_i` instead
  |
  = note: #[warn(unused_variables)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 0.50s
```

L'avertissement indique que nous devrions utiliser `_i` comme nom à sa place : le tiret bas indique que nous avons l'intention de ne pas utiliser cette variable. Nous pouvons appliquer automatiquement cette suggestion en utilisant l'outil `rustfix` en lançant la commande `cargo fix` :

```
$ cargo fix
    Checking mon_programme v0.1.0 (file:///projects/mon_programme)
      Fixing src/main.rs (1 fix)
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

Lorsque nous regardons à nouveau *src/main.rs*, nous pouvons constater que `cargo fix` a changé le code :

Fichier : src/main.rs

```
fn fait_quelquechose() {}

fn main() {
    for _i in 0..100 {
        fait_quelquechose();
    }
}
```

La variable de la boucle `for` s'appelle maintenant `_i`, et l'avertissement ne s'affiche plus.

Vous pouvez aussi utiliser la commande `cargo fix` pour corriger votre code entre différentes éditions de Rust. Les éditions sont abordées à l'annexe E.

Une analyse statique plus complète avec Clippy

L'outil Clippy est une collection d'analyses statiques pour analyser votre code afin que vous puissiez débusquer certaines erreurs courantes et ainsi améliorer votre code.

Pour installer Clippy, saisissez ceci :

```
$ rustup component add clippy
```

Pour lancer l'analyse statique de Clippy sur un projet Cargo, saisissez ceci :

```
$ cargo clippy
```

Par exemple, imaginons que vous écrivez un programme qui utilise une approximation d'une constante mathématique, comme pi, comme le fait ce programme :

Fichier : src/main.rs

```
fn main() {
    let x = 3.1415;
    let r = 8.0;
    println!("l'aire du cercle vaut {}", x * r * r);
}
```

L'exécution de `cargo clippy` sur ce projet va afficher cette erreur :

```
error: approximate value of `f{32, 64}::consts::PI` found. Consider using it
directly
 -- > src/main.rs:2:13
  |
2 |     let x = 3.1415;
  |             ^^^^^^
  |
  = note: #[deny(clippy::approx_constant)] on by default
  = help: for further information visit https://rust-lang-nursery.github.io/
rust-clippy/master/index.html#approx_constant
```

Cette erreur vous fais savoir que Rust a cette constante qui est définie plus précisément et que votre programme serait plus pertinent si vous utilisiez à la place la constante. Vous changeriez alors votre code pour utiliser la constante `PI`. Le code suivant ne donne pas d'erreur ou d'avertissement avec Clippy :

Fichier : src/main.rs

```
fn main() {
    let x = std::f64::consts::PI;
    let r = 8.0;
    println!("l'aire du cercle vaut {}", x * r * r);
}
```

Pour en savoir plus Clippy, voyez sa documentation.

L'intégration aux IDE en utilisant le Rust Language Server

Pour aider l'intégration aux IDE, le projet Rust distribue le *Rust Language Server* (`rls`). Cet outil suit le Language Server Protocol, qui est une spécification entre les IDE et les langages pour communiquer entre eux. Différents clients peuvent utiliser le `rls`, comme le plug-in Rust pour Visual Studio Code.

Pour installer le `rls`, saisissez ceci :

```
$ rustup component add rls
```

Installez ensuite le système du *language server* dans votre IDE ; vous devriez obtenir des capacités supplémentaires comme l'auto-complétion, pouvoir se rendre à la définition de l'élément, et la mise en valeur d'erreurs sur la ligne concernée.

Pour plus d'information sur `rls`, rendez-vous à sa documentation.

🚧 Attention, peinture fraîche !

Cette page a été traduite par une seule personne et n'a pas été relue et vérifiée par quelqu'un d'autre ! Les informations peuvent par exemple être erronées, être formulées maladroitement, ou contenir d'autres types de fautes.

Vous pouvez contribuer à l'amélioration de cette page sur sa Pull Request.

Annexe E - Les éditions

Au chapitre 1, vous avez constaté que `cargo new` ajoutait une petite métadonnée à propos d'une édition dans votre fichier *Cargo.toml*. Cette annexe vous explique ce que cela signifie !

Le langage Rust et son compilateur suivent un cycle de publication de six semaines, ce qui signifie que leurs utilisateurs suivent un flux constant de nouvelles fonctionnalités. Les autres langages de programmation publient moins souvent des changements mais qui sont plus gros ; Rust a fait le choix de publier des petits changements plus fréquemment. Au bout d'un certain moment, tous ces petits changements s'accumulent. Mais de mise à jour en mise à jour, il devient difficile de regarder en arrière et de dire : “Ouah, Rust a beaucoup changé entre Rust 1.10 et Rust 1.31”.

Tous les deux ou trois ans, l'équipe Rust produit une nouvelle *édition* de Rust. Chaque édition rassemble des fonctionnalités qui ont convergé en un ensemble clair, avec une documentation et des outils complètement à jour. Les nouvelles éditions sont livrées comme faisant partie du cycle habituel de publication toutes les six semaines.

Les éditions apportent différentes choses pour différentes personnes :

- Pour les utilisateurs actifs de Rust, une nouvelle édition regroupe les différents changements progressifs dans un ensemble clair.
- Pour ceux qui n'utilisent pas Rust, une nouvelle édition signale la livraison d'avancées majeures, qui pourrait être le signal que Rust mériterait un nouveau coup d'œil.
- Pour ceux qui développent Rust, une nouvelle édition est un point de ralliement pour l'ensemble du projet.

Au moment de cette écriture, deux éditions de Rust sont disponibles : Rust 2015 et Rust 2018. Ce livre est écrit selon les termes de l'édition Rust 2018.

La clé `edition` dans *Cargo.toml* indique quelle édition le compilateur doit utiliser dans votre code. Si la clé n'existe pas, Rust utilise `2015` comme valeur de l'édition, pour des raisons de rétro-compatibilité.

Chaque projet peut opter pour une autre édition que l'édition 2015 par défaut. Les éditions peuvent impliquer des changements incompatibles, comme l'introduction d'un nouveau mot-clé qui rentre en conflit avec des identificateurs (noms de variables, de fonctions, ...) utilisés dans le code. Cependant, à moins que vous ne décidiez d'opter pour ces changements, votre code va continuer à se compiler même si vous augmentez la version du compilateur Rust que vous utilisez.

Toutes les versions du compilateur Rust supporte toutes les éditions qui ont existé avant la publication courante du compilateur, et ils peuvent lier ensemble les crates de n'importe quelle édition supportée. Les changements de chaque édition changent uniquement la façon dont le compilateur interprète initialement le code. Par conséquent, si vous utilisez Rust 2015 et qu'une de vos dépendances utilise Rust 2018, votre programme va se compiler et être capable d'utiliser cette dépendance. La situation inverse, dans laquelle votre projet utilise Rust 2018 et qu'une dépendance utilise Rust 2015, va aussi fonctionner.

En clair : la plupart des fonctionnalités seront disponibles sur toutes les versions. Les développeurs qui utilisent n'importe quelle édition de Rust vont continuer à constater des améliorations au fur et à mesure que des nouvelles éditions stables sont publiées. Cependant, dans certains cas, principalement lorsque des nouveaux mot-clés serons rajoutés, certaines nouvelles fonctionnalités ne seront disponibles que dans les nouvelles éditions. Vous aurez alors besoin de changer d'édition si vous souhaitez profiter des avantages de ces fonctionnalités.

Pour en savoir plus, le *Edition Guide* est un livre complet sur les éditions, qui énumère les différences entre les éditions et qui explique comment mettre à jour automatiquement votre code vers une nouvelle édition via `cargo fix`.

🚧 Attention, peinture fraîche !

Cette page a été traduite par une seule personne et n'a pas été relue et vérifiée par quelqu'un d'autre ! Les informations peuvent par exemple être erronées, être formulées maladroitement, ou contenir d'autres types de fautes.

Vous pouvez contribuer à l'amélioration de cette page sur sa Pull Request.

Annexe F : les traductions de ce livre

Voici des ressources dans d'autres langages qu'en Français. Certaines sont toujours en cours de construction ; consultez le drapeau Translation pour les aider, ou nous informer de la création d'une nouvelle traduction !

- English
- Português (BR)
- Português (PT)
- 简体中文
- Українська
- Español, alternative
- Italiano
- Русский
- 한국어
- 日本語
- Polski
- עברית
- Cebuano
- Tagalog
- Esperanto
- ελληνική
- Svenska
- Farsi
- Deutsch
- Turkish, online

🚧 Attention, peinture fraîche !

Cette page a été traduite par une seule personne et n'a pas été relue et vérifiée par quelqu'un d'autre ! Les informations peuvent par exemple être erronées, être formulées maladroitement, ou contenir d'autres types de fautes.

Vous pouvez contribuer à l'amélioration de cette page sur sa Pull Request.

Annexe G - Comment Rust est construit, et “Nightly Rust”

Cette annexe va expliquer comment Rust est construit et comment cela vous impacte en tant que développeur Rust.

La stabilité sans stagnation

En tant que langage, Rust se soucie *beaucoup* de la stabilité de votre code. Nous voulons que Rust soit une solide fondation sur laquelle vous pouvez construire, et si les choses changent constamment, cela serait impossible. En même temps, si nous ne pouvions pas expérimenter de nouvelles fonctionnalités, nous ne pourrions pas découvrir les défauts importants avant leur publication, ce qui serait trop tard pour changer les choses.

Notre solution à ce problème est ce que nous appelons la “stabilité sans stagnation”, et notre ligne directrice est la suivante : vous ne devriez jamais craindre de passer à nouvelle version de Rust stable. Chaque mise à jour devrait être facile, et devrait aussi vous apporter de nouvelles fonctionnalités, moins de bogues et un temps de compilation plus rapide.

Les canaux de diffusion et sauter dans le train

Le développement de Rust suit un *planning ferroviaire*. Ce que cela veut dire, c'est que tout le développement est fait sur la branche `master` du dépôt de Rust. Les publications suivent le modèle de trains de publication de programmes, qui a été popularisé par Cisco IOS et d'autres projets logiciels. Il y a trois *canaux de diffusion* pour Rust :

- Nightly
- Beta

- Stable

La plupart des développeurs Rust utilisent principalement le canal stable, mais ceux qui souhaitent essayer les nouvelles fonctionnalités expérimentales utilisent nightly ou beta.

Voici un exemple du fonctionnement du processus de développement et de publication : supposons que l'équipe de Rust travaille sur la publication de Rust 1.5. Cette publication a été faite en décembre 2015, et nous permet de nous appuyer sur des numéros de version réalistes. Une nouvelle fonctionnalité a été ajoutée à Rust : un nouveau commit est arrivé sur la branche `master`. Chaque nuit, une nouvelle version nightly de Rust est produite. Chaque jour voit une nouvelle publication, et ces publications sont créées automatiquement par l'infrastructure de publication. Ainsi, les publications ressemblent à ceci, une fois par nuit :

```
nightly: * - - * - - *
```

Tous les six semaines, c'est le moment de préparer une nouvelle publication ! La branche `beta` du dépôt Rust est alors dérivée de la branche `master` utilisée par nightly. Ainsi, il y a deux canaux de publications :

```
nightly: * - - * - - *
                     |
beta:                *
```

La plupart des utilisateurs Rust n'utilisent pas activement les publications en beta, mais les tests en beta sur leur système d'Intégration Continue aident à découvrir des potentielles régressions. Pendant ce temps, il continue à avoir une publication nightly chaque nuit :

```
nightly: * - - * - - * - - * - - *
                     |
beta:                *
```

Imaginons qu'une régression soit trouvée. C'est alors une bonne chose que nous ayons du temps pour tester la publication beta avant que la régression se retrouve dans une publication stable ! La correction est alors appliquée sur `master`, ainsi nightly est corrigé, et ensuite la correction est reportée sur la branche `beta`, et une nouvelle publication de beta est produite :

```
nightly: * - - * - - * - - * - - * - - *
                     |
beta:                * - - - - - - - - *
```

Six semaines après que la première beta soit créée, c'est le moment de publier une version stable ! La branche `stable` est produite à partir de la branche `beta` :

```
nightly: * - - * - - * - - * - - * - - * - * - *
                  |
beta:             * - - - - - - - - - *
                                      |
stable:                               *
```

Youpi ! Rust 1.5 est sorti ! Cependant, nous avons oublié quelque chose : comme les six semaines sont passées, nous devons aussi publier une nouvelle beta de la version *suivante* de Rust, la 1.6. Donc après que la branche `stable` soit dérivée de la `beta`, la prochaine version de la branche `beta` doit à nouveau être dérivée de nightly :

```
nightly: * - - * - - * - - * - - * - - * - * - *
                  |                           |
beta:             * - - - - - - - - - *       *
                                      |
stable:                               *
```

C'est appelé le “modèle ferroviaire” car toutes les six semaines, une nouvelle publication “quitte la gare”, mais doit encore voyager dans la voie de la beta avant d'arriver en gare de la publication stable.

Rust publie régulièrement toutes les six semaines, réglée comme une montre. Si vous savez la date d'une publication Rust, vous savez la date de la suivante : elle aura toujours lieu six semaines plus tard. Un des avantages d'avoir des publications planifiées toutes les six semaines est que le train suivant arrive rapidement après. Si une fonctionnalité n'est pas intégrée à une publication, il n'y a pas à s'inquiéter : une autre arrive bientôt ! Cela aide à réduire la pression pour faire passer en toute discrétion des fonctionnalités éventuellement inachevées à l'approche de la date limite de diffusion.

Grâce à ce processus, vous pouvez toujours découvrir la prochaine compilation de Rust et constater par vous-même qu'il est facile de mettre à jour : si une publication en beta ne fonctionne pas comme prévu, vous pouvez signaler cela à l'équipe et cela sera corrigé avant que la prochaine publication stable soit produite ! La dégradation d'une version bêta est plutôt rare, mais `rustc` reste un logiciel, et les bogues peuvent exister malgré tout.

Les fonctionnalités instables

Il reste une surprise avec ce modèle de publication : les fonctionnalités instables. Rust utilise une technique qui s'appelle les “drapeaux de fonctionnalités” pour déterminer quelles fonctionnalités sont activées dans une publication donnée. Si une nouvelle fonctionnalité est en développement actif, elle va atterrir sur `master`, et ainsi, dans nightly, mais derrière un *drapeau de fonctionnalités*. Si vous, en tant qu'utilisateur, souhaitez essayer la fonctionnalité

en cours de développement, vous pouvez, mais vous devez utiliser une publication nightly de Rust et annoter votre code source avec le drapeau approprié pour l'activer.

Si vous utilisez une publication beta ou stable de Rust, vous ne pouvez pas utiliser de drapeaux de fonctionnalités. C'est la clé qui permet d'obtenir une utilisation pratique avec les nouvelles fonctionnalités avant que nous les déclarions stables pour toujours. Ceux qui souhaitent activer ces fonctionnalités expérimentales peuvent le faire, et ceux qui souhaitent avoir une expérience plus solide peuvent s'en tenir au canal stable et leur code ne sera pas cassé. C'est la stabilité sans stagnation.

Ce livre contient uniquement des informations sur des fonctionnalités stables, car les fonctionnalités en cours de développement sont toujours en train de changer, et elles seront sûrement différentes entre le moment où ce livre sera écrit et lorsqu'elles seront activées dans les compilations stables. Vous pouvez trouver la documentation pour les fonctionnalités uniquement pour nightly en ligne.

Rustup et le role de Rust nightly

Rustup facilite les changements entre les différents canaux de publication de Rust, de manière globale ou par projet. Par défaut, vous avez Rust stable d'installé. Pour installer nightly, vous pouvez saisir, par exemple :

```
$ rustup toolchain install nightly
```

Vous pouvez aussi voir avec `rustup` toutes les *toolchains* (les publications de Rust et leurs composants associés) que vous avez d'installées. Voici un exemple d'un ordinateur sous Windows d'un des auteurs du livre :

```
> rustup toolchain list
stable-x86_64-pc-windows-msvc (default)
beta-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
```

Comme vous pouvez le constater, la toolchain stable est celle par défaut. La plupart des utilisateurs Rust utilisent celle qui est stable la plupart du temps. Il est possible que vous souhaitiez utiliser celle qui est stable la plupart du temps, mais que vous souhaitiez utiliser nightly sur un projet particulier, car parce que vous vous intéressez à une fonctionnalité expérimentale. Pour pouvoir faire cela, vous pouvez utiliser `rustup override` dans le dossier de ce projet pour régler `rustup` pour qu'il utilise la toolchain nightly lorsque vous vous trouvez dans ce dossier :

```
$ cd ~/projets/necessite-nightly
$ rustup override set nightly
```

Maintenant, à chaque fois que vous faites appel à `rustc` ou `cargo` à l'intérieur de *~/projets/necessite-nightly*, `rustup` va s'assurer que vous utilisez Rust nightly, plutôt que votre Rust stable par défaut. C'est très utile lorsque vous avez beaucoup de projets Rust !

Le processus de RFC et les équipes

Donc, comment en apprendre plus ces nouvelles fonctionnalités ? Le modèle de développement de Rust suit le *processus de Request For Comments (RFC)*. Si vous souhaitez avoir une amélioration de Rust, vous pouvez rédiger une proposition, qu'on appelle une RFC.

N'importe qui peut écrire de RFC pour améliorer Rust, et les propositions sont examinées et débattues par l'équipe de Rust, qui est composée de nombreuses sous-équipes spécialisées dans différents domaines. Voici une liste complète des équipes sur le site web de Rust, qui comprend des équipes pour chaque aspect du projet : la conception du langage, l'implémentation du compilateur, de l'infrastructure, de la documentation, et plus encore. L'équipe appropriée lit la proposition et les commentaires, écrit quelques commentaires la concernant, et finalement, un consensus se crée pour accepter ou rejeter la fonctionnalité.

Si la fonctionnalité est acceptée, un ticket est ouvert sur le dépôt de Rust, et quelqu'un peut l'implémenter. La personne qui l'implémente ne peut pas être celle qui a proposé la fonctionnalité ! Lorsque l'implémentation est prête, elle atterrit sur la branche `master` derrière un drapeau de fonctionnalité, comme nous l'avons vu dans la section “Les fonctionnalités instables”.

Au bout d'un moment, une fois que les développeurs Rust qui utilisent les publications nightly ont pu tester la nouvelle fonctionnalité, les membres de l'équipe vont discuter de la fonctionnalité, de voir comment elle a fonctionné sur nightly, et vont décider si elle doit être publiée sur Rust stable ou non. Si la décision est d'avancer, le drapeau de fonctionnalité est enlevé, et la fonctionnalité est maintenant considérée comme stable ! Elle saute alors dans le train en direction d'une nouvelle publication stable de Rust.