Konfigurowanie

W tym samouczku założono, że masz zainstalowanego lokalnie Rusta. Na szczęście jest to bardzo proste.

$ curl https://sh.rustup.rs -sSf | sh

$ rustup component add rust-docs

Polecam pobranie domyślnej stabilnej wersji; łatwo jest później pobrać niestabilne wersje i przełączać się między nimi. To pobiera kompilator, menedżera pakietów Cargo, dokumentację API i Rust Book. Podróż tysiąca mil zaczyna się od jednego kroku, a ten pierwszy krok jest bezbolesny. rustup to polecenie, którego używasz do zarządzania instalacją Rusta. Kiedy pojawi się nowa stabilna wersja, wystarczy powiedzieć rustup update, aby zaktualizować. rustup doc otworzy dokumentację offline w Twojej przeglądarce. Prawdopodobnie masz już edytor, który ci się podoba, a podstawowa obsługa Rusta jest dobra. Sugerowałbym, abyś najpierw zaczął od podstawowego podświetlania składni i rozwijał się, gdy twoje programy stają się większe. Osobiście jestem fanem Geany'ego, który jest jednym z niewielu edytorów z gotową obsługą Rusta; jest to szczególnie łatwe w Linuksie, ponieważ jest dostępne +przez menedżera pakietów, ale działa dobrze na innych platformach. Najważniejszą rzeczą jest wiedzieć, jak edytować, kompilować i uruchamiać programy Rusta. Uczysz się programować palcami; wpisz kod samodzielnie i naucz się efektywnie zmieniać kolejność rzeczy za pomocą edytora. Czy Rust jest językiem, którym byłbyś zainteresowany? Przeanalizujmy kilka małych przykładów kodu, które demonstrują niektóre z jego mocnych stron. Główną koncepcją, która sprawia, że Rust jest wyjątkowy, jest „własność”. Rozważ ten mały przykład:

```
fn main ( ) {

let mut x = vec! [ "Hello" , "world" ];

}
```

Ten program tworzy zmienną o nazwie x . Wartość tej zmiennej to Vec <T> , „wektor”, który tworzymy za pomocą makra zdefiniowanego w bibliotece standardowej. To makro nazywa się vec , makra są wywoływane z ! . Wszystko to zgodnie z ogólną zasadą w Rust: wyjaśniaj. Makra mogą robić znacznie bardziej złożone rzeczy niż wywołania funkcji, dlatego różnią się wizualnie. ! Pomaga także analizować, ułatwiając narzędzia do pisania, co również jest ważne. Użyliśmy mut, aby uczynić x zmiennym: w Rust zmienne są domyślnie niezmienne. W dalszej części tego przykładu będziemy mutować ten wektor. Należy wspomnieć, że nie potrzebujemy tutaj adnotacji typu: podczas gdy Rust jest pisany statycznie, nie musimy jawnie opisywać typu. Rust ma wnioskowanie o typie, aby zrównoważyć moc pisania statycznego z szczegółowością adnotacji typu. Rust preferuje alokację pamięci ze stosu niż z kopca: x jest umieszczane bezpośrednio na stosie. Jednak typ Vec <T> przydziela miejsce na elementy wektorowe na kopcu. Jeśli nie jesteś zaznajomiony z tym rozróżnieniem, możesz je na razie zignorować lub zajrzeć do „The Stack and the Mound”. Rust jako systemowy język programowania daje możliwość kontrolowania alokacji pamięci, ale to, jak zaczynamy, nie jest tak istotne. Wcześniej wspomnieliśmy, że „członkostwo” jest nową kluczową koncepcją w Rust. W terminologii Rusta x jest „właścicielem” wektora. Oznacza to, że gdy x wyjdzie poza zakres, pamięć przydzielona wektorowi zostanie zwolniona. Odbywa się to w sposób deterministyczny przez kompilator Rust, bez potrzeby stosowania mechanizmu takiego jak Garbage Collector. Innymi słowy, w Ruście nie wywołuje się jawnie funkcji takich jak malloc i free: kompilator statycznie określa, kiedy pamięć musi zostać przydzielona lub zwolniona, i wstawia te wywołania za ciebie. Błądzić jest rzeczą ludzką, ale kompilatory nigdy nie zapominają. Dodajmy kolejną linię do naszego przykładu:

```
fn main ( ) {

let mut x = vec! [ "Hello" , "world" ];

let y = & x [ 0 ];

}
```

Wprowadziliśmy kolejną zmienną i . W tym przypadku y jest „odniesieniem” do pierwszego elementu wektora. Referencje w Rust są podobne do wskaźników w innych językach, ale z dodatkowymi kontrolami bezpieczeństwa w czasie kompilacji. Referencje wchodzą w interakcję z systemem członkostwa poprzez „pożyczanie”, pożyczają to, na co są skierowane, zamiast tego posiadać. Różnica polega na tym, że gdy odwołanie wykracza poza zakres, podstawowa pamięć nie zostanie zwolniona. W takim przypadku zwalnialibyśmy tę samą pamięć dwa razy, co jest złe. Dodajmy trzecią linię. Ta linia wygląda niewinnie, ale powoduje błąd kompilacji:

```
fn main ( ) {

let mut x = vec! [ "Hello", "world"];

let y = & x [ 0];

x.push ("foo");

}
```

push to metoda w wektorach, która dodaje element na końcu wektora. Kiedy próbujemy skompilować program, pojawia się błąd:

```
błąd: nie można pożyczyć `x` jako zmiennego, ponieważ jest również pożyczony jako niezmienny

x.push ("foo");

^

uwaga: tutaj występuje poprzednie wypożyczenie `x`; niezmienna pożyczka zapobiega kolejnym ruchom lub zmiennym pożyczkom `x` aż do zakończenia pożyczki

let y = & x [ 0];

^

uwaga: poprzednie wypożyczenie kończy się tutaj

fn main ( ) {

}

^
```

wow! Kompilator Rust może czasami podać bardzo szczegółowe błędy i tym razem jeden z nich. Jak wyjaśnia błąd, podczas gdy zmieniamy zmienną, nie możemy wywoływać push . Dzieje się tak, ponieważ mamy już odniesienie do elementu wektora, y . Mutacja czegoś, gdy istnieje odniesienie do tego, jest niebezpieczne, ponieważ możemy unieważnić odniesienie. W tym konkretnym przypadku, kiedy tworzymy wektor, przydzieliliśmy miejsce tylko na dwa elementy. Dodanie strony trzeciej oznaczałoby przydzielenie nowego segmentu pamięci dla wszystkich elementów, skopiowanie wszystkich poprzednich wartości i zaktualizowanie wewnętrznego wskaźnika do tej pamięci. Wszystko

to jest w porządku. Problem polega na tym, że i nie zostałby zaktualizowany, generując „wiszący wskaźnik”. Co jest złe. Jakiekolwiek użycie y byłoby w tym przypadku błędem, a kompilator ostrzega nas przed tym. Jak więc rozwiązać ten problem? Istnieją dwa podejścia, które możemy przyjąć. Pierwszym z nich jest wykonanie kopii zamiast odniesienia:

```
fn main ( ) {

let mut x = vec! [ "Hello" , "world" ];

let y = x [ 0 ] .clone ();

x.push ( "foo" );

}
```

Rust ma domyślnie semantykę ruchu, więc jeśli chcemy zrobić kopię niektórych danych, wywołujemy metodę clone() . W tym przykładzie y nie jest już odwołaniem do wektora przechowywanego w x , ale kopią jego pierwszego elementu, "Hello" . Ponieważ nie mamy referencji, nasze naciśnięcie ( ) działa doskonale. Jeśli naprawdę chcemy odniesienia, potrzebujemy innej opcji: upewnij się, że nasze odniesienie wykracza poza zakres, zanim spróbujemy wykonać mutację. W ten sposób:

```
fn main ( ) {

let mut x = vec! [ "Hello" , "world" ];

{

let y = & x [ 0 ];

}

x.push ( "foo" );

}
```

Dzięki dodatkowej parze kluczy stworzyliśmy zakres wewnętrzny. i wyjdzie poza zakres, zanim wywołamy push() , więc nie ma problemu. Ta koncepcja członkostwa nie tylko dobrze zapobiega zawieszaniu się wskaźników, ale także całemu zestawowi problemów, takich jak unieważnianie iteratora, współbieżność i inne.

Pierwsze kroki

Ta pierwsza część przeprowadzi cię przez Rusta i jego narzędzia. Najpierw zainstalujemy Rusta. Potem klasyczny program „Hello World”. Na koniec porozmawiamy o Cargo, menedżerze pakietów Rust i systemie kompilacji.

Hello World!

Teraz, kiedy już zainstalowałeś Rusta, napiszmy twój pierwszy program. Tradycją jest, że pierwszym programem w dowolnym języku jest ten, który wypisuje tekst „Witaj, świecie!” do ekranu. Zaletą rozpoczynania od tak prostego programu jest to, że sprawdzasz nie tylko, czy kompilator jest zainstalowany, ale także czy działa. Drukowanie informacji na ekranie to bardzo powszechna rzecz. Pierwszą rzeczą, którą musimy zrobić, to utworzyć plik, w którym możemy umieścić nasz kod. Lubię tworzyć katalog projektów w moim katalogu użytkownika i przechowywać tam wszystkie moje projekty. Rust nie dba o to, gdzie znajduje się kod. Prowadzi to do wyjaśnienia kolejnej kwestii: ten przewodnik zakłada, że masz podstawową znajomość wiersza poleceń. Rust nie żąda niczego w

odniesieniu do narzędzi do edycji ani miejsca zamieszkania kodu. Jeśli wolisz IDE od interfejsu wiersza poleceń, prawdopodobnie powinieneś rzucić okiem na SolidOak lub inne wtyczki do twojego ulubionego IDE. Istnieje wiele rozszerzeń o różnej jakości, które są opracowywane przez społeczność. Zespół stojący za Rust publikuje również wtyczki dla różnych wydawców. Konfigurowanie edytora lub środowiska IDE wykracza poza cele tego samouczka. Sprawdź dokumentację dotyczącą konkretnej konfiguracji. Powiedziawszy to, utwórzmy katalog w naszym katalogu projektu.

$ mkdir ~ / projects

$ cd ~ / projects

$ mkdir hello_world

$ cd hello_world

Jeśli korzystasz z systemu Windows i nie używasz programu PowerShell, znak ~ może nie działać. Aby uzyskać więcej informacji, zapoznaj się z dokumentacją konkretnego terminala. Utwórzmy nowy plik kodu źródłowego. Nazwiemy nasz plik main.rs. Pliki Rust kończą się rozszerzeniem .rs . Jeśli używasz więcej niż jednego słowa w nazwie pliku, użyj łącznika: hello_world.rs zamiast helloworld.rs . Teraz, gdy masz otwarty plik, wpisz to w:

fn main ( ) {

println! ("Hello, world!");

}

Zapisz zmiany w pliku i wpisz następujące polecenie w oknie terminala:

$ rustc main.rs

$. / main # lub main.exe w systemie Windows

Hello, world!

Powodzenie! Zobaczmy teraz szczegółowo, co się stało.

fn main ( ) {

}

Linie te definiują funkcję w Rust. Główna funkcja jest szczególna: jest początkiem każdego programu Rusta. Pierwszy wiersz mówi: „Deklaruję funkcję o nazwie main, która nie otrzymuje żadnych argumentów i nic nie zwraca”. Gdyby istniały argumenty, byłyby one ujęte w nawiasy ((i)), a ponieważ nie zwracamy niczego z tej funkcji, możemy całkowicie pominąć zwracany typ. Dojdziemy do tego później. Zauważysz również, że funkcja jest ujęta w nawiasy klamrowe ({ i }). Rust wymaga tych kluczy ograniczających wszystkie ciała funkcji. Za dobry styl uważa się również umieszczenie nawiasu otwierającego w tym samym wierszu co deklaracja funkcji, ze spacją pośrednią. Poniżej znajduje się ta linia:

println! ("Hello, world!");

Ta linia wykonuje całą pracę w naszym małym programie. Istnieje wiele szczegółów, które są tutaj ważne. Po pierwsze, jest wcięty czterema spacjami, a nie tabulatorami. Skonfiguruj swój edytor, aby wstawiać cztery spacje za pomocą klawisza tabulacji. Udostępniamy przykładowe konfiguracje dla różnych edytorów. Drugi punkt to println! ( ) Część . To jest wywołanie makra Rust, czyli jak

metaprogramowanie odbywa się w Rust. Gdyby zamiast tego była to funkcja, wyglądałaby tak: println() .Dla naszych celów nie musimy się martwić tą różnicą. Po prostu wiedz, że czasami zobaczysz! , a to oznacza, że wywołujesz makro zamiast normalnej funkcji. Rust implementuje println! jako makro zamiast funkcji nie bez powodu, ale to zaawansowany temat. Ostatnia rzecz, o której warto wspomnieć: makra Rust różnią się od makr C, jeśli ich używałeś. Nie bój się używać makr. Do szczegółów dojdziemy w końcu, na razie musisz nam po prostu zaufać. Następnie „Witaj, świecie!" jest to ciąg znaków. Ciągi znaków są zaskakująco złożonym tematem w systemowych językach programowania, a konkretnie jest to ciąg znaków „przypisany statycznie". Jeśli chcesz przeczytać o alokacji pamięci, spójrz na stos i kopiec , ale na razie nie musisz, jeśli nie chcesz. Przekazujemy ten napis jako argument do println! który z kolei drukuje ciąg znaków na ekranie. Łatwy! Na koniec linia kończy się średnikiem (;). Rust jest językiem zorientowanym na wyrażenia, co oznacza, że większość rzeczy to wyrażenia, a nie instrukcje. ; służy do wskazania, że wyrażenie się zakończyło i że następne jest gotowe do rozpoczęcia. Większość linii kodu w Rust kończy się na ; . Na koniec skompiluj i uruchom nasz program. Możemy skompilować z naszym kompilatorem rustc, przekazując mu nazwę naszego pliku źródłowego:

$ rustc main.rs

Jest to podobne do gcc lub clang , jeśli pochodzisz z C lub C . Rust wyświetli wykonywalny plik binarny. Możesz to zobaczyć za pomocą ls :

$ ls

main main.rs

lub w Windowsie:

$ dir

main.exe main.rs

Istnieją dwa pliki: nasz kod źródłowy z rozszerzeniem .rs oraz plik wykonywalny (main.exe w Windows, main w innych)

$. / main # lub main.exe w systemie Windows

Powyżej drukuje nasz tekst Witaj, świecie! do naszego terminala. Jeśli zaczynasz od języka dynamicznego, takiego jak Ruby, Python lub JavaScript, prawdopodobnie nie jesteś przyzwyczajony do rozdzielenia tych dwóch kroków. Rust jest językiem kompilowanym, co oznacza, że możesz skompilować swój program i przekazać go komuś innemu, a on nie musi mieć zainstalowanego Rusta. Jeśli dasz komuś plik .rb, .py lub .js, musi on mieć implementację Ruby/Python/JavaScript, ale potrzebujesz tylko jednego polecenia dla obu, skompiluj i uruchom swój program. Chodzi o zrównoważenie zalet i wad w projektowaniu języka, a Rust wybrał.

Gratulacje, oficjalnie napisałeś program w języku Rust. To czyni cię programistą Rusta! Witamy. Następnie chciałbym przedstawić inne narzędzie, Cargo, które służy do pisania programów Rusta dla świata rzeczywistego. Po prostu użycie rustc jest dobre do prostych rzeczy, ale w miarę rozwoju projektu będziesz potrzebować czegoś, co pomoże ci zarządzać wszystkimi dostępnymi opcjami, a także ułatwi dzielenie się kodem z innymi ludźmi i projektami.

Witaj Cargo!

Cargo to narzędzie, którego Rusteros używa do zarządzania projektami Rust. Cargo jest obecnie w stanie przed wersją 1.0, w związku z czym wciąż trwają prace nad nim. Jednak jest już wystarczająco

dobry dla wielu projektów Rust i zakłada się, że projekty Rust będą używać Cargo od samego początku. Cargo zarządza trzema rzeczami: kompilacją twojego kodu, pobieraniem zależności potrzebnych twojemu kodowi i kompilacją tych zależności. W pierwszej kolejności Twój kod nie będzie miał żadnej zależności, dlatego będziemy używać tylko pierwszej części funkcjonalności Cargo. W końcu dodamy więcej. Ponieważ zaczęliśmy korzystać z Cargo od samego początku, łatwo będzie dodać później. Jeśli zainstalowałeś Rusta za pośrednictwem oficjalnych instalatorów, powinieneś mieć Cargo. Jeśli zainstalowałeś Rust w jakikolwiek inny sposób, możesz zajrzeć do pliku README Cargo, aby uzyskać szczegółowe instrukcje, jak go zainstalować.

Migracja w Charge

Przekonwertujmy Hello World w Charge. Aby wczytać nasz projekt, potrzebujemy dwóch rzeczy: Stworzyć plik konfiguracyjny Cargo.toml i umieścić nasz plik źródłowy we właściwym miejscu. Zróbmy najpierw tę część:

$ mkdir src

$ mv main.rs src / main.rs

Zauważ, że ponieważ tworzymy plik wykonywalny, użyj main.rs . Gdybyśmy chcieli stworzyć bibliotekę, powinniśmy użyć lib.rs . Dostosowane lokalizacje punktu wejścia można określić za pomocą klucza [[[lib]] lub [[bin]]] crates-custom w pliku TOML opisanym poniżej. Cargo oczekuje, że pliki kodu źródłowego będą znajdować się w katalogu src. Pozostawia to poziom główny dla innych rzeczy, takich jak pliki README, informacje licencyjne, i wszystko niezwiązane z twoim kodem. Cargo pomaga nam utrzymać ład i porządek w naszych projektach. Miejsce na wszystko i wszystko na swoim miejscu. Oto nasz plik konfiguracyjny:

$ edytor Cargo.toml

Upewnij się, że masz tę poprawną nazwę: potrzebujesz dużej litery C! Włóż to do środka:

[package]

name = "hello_world"

version = "0.0.1"

authors = ["Your name <your@example.com> "]

Ten plik jest w formacie TOML. Wyjaśnijmy to samo:

Celem TOML jest stworzenie minimalnego formatu konfiguracji, który jest łatwy do odczytania ze względu na oczywistą semantykę. TOML jest przeznaczony do jednoznacznego mapowania do tabeli skrótów. TOML powinien być łatwy do konwersji na struktury danych w wielu różnych językach. TOML jest bardzo podobny do INI, ale ma kilka dodatkowych subtelności. Kiedy już masz ten plik na swoim miejscu, powinniśmy być gotowi do kompilacji! Spróbuj tego:

$ build charge

Compiling hello_world v0. 0.1 (file: /// home / yourname / projects /

hello_world)

$. / target / debug / hello_world

Hello World!

Bam! Budujemy nasz projekt za pomocą opłaty za kompilację i uruchamiamy go za pomocą . / target/ debug /hello_world . Możemy wykonać dwa kroki w jednym z uruchomieniem ładowania:

$ charge run

Running `target / debug / hello_world`

Hello World!

Zauważ, że tym razem nie przebudowaliśmy projektu. Cargo ustalił, że nie zmieniliśmy pliku źródłowego, więc po prostu uruchomiłem plik binarny. Gdybyśmy dokonali modyfikacji, powinniśmy zobaczyć, jak wykonuje ona dwa kroki:

$ charge run

Compiling hello_world v0. 0.1 (file: /// home / yourname / projects / hello_world)

Running `target / debug / hello_world`

Hello World!

Nie przyniosło nam to wiele poza prostym użyciem rustc , ale pomyśl o przyszłości: kiedy nasze projekty staną się bardziej skomplikowane, będziemy musieli zrobić więcej, aby wszystkie części poprawnie się skompilowały. Dzięki Cargo, w miarę rozwoju naszego projektu, po prostu uruchamiamy cargo build i wszystko będzie działać poprawnie. Kiedy nasz projekt będzie wreszcie gotowy do wydania, możesz użyć cargo build --release, aby skompilować go z optymalizacjami. Być może zauważyłeś również, że Cargo utworzyło nowy plik: Cargo.lock

[root]

name = "hello_world"

version = "0.0.1"

Ten plik jest używany przez Cargo do śledzenia zależności używanych w Twojej aplikacji. Na razie nie mamy żadnych i jest to trochę rozproszone. Nigdy nie powinieneś sam dotykać tego pliku, po prostu pozwól Cargo się tym zająć. Otóż to! Z powodzeniem zbudowaliśmy hello_world z Cargo. Chociaż nasz program jest prosty, wykorzystuje wiele prawdziwych narzędzi, których będziesz używać przez resztę swojej kariery w Rust. Możesz założyć, że aby rozpocząć praktycznie każdy projekt Rust, wykonaj następujące czynności:

$ git clone someturl.com/foo

$ cd foo

$ build chargé

Nowy projekt

Nie musisz przechodzić przez cały ten proces za każdym razem, gdy chcesz rozpocząć nowy projekt! Cargo ma możliwość stworzenia katalogu szablonów, w którym można od razu przystąpić do tworzenia. Aby rozpocząć nowy projekt z Cargo, używamy cargo new :

$ cargo nowy hello_world –bin

Podajemy opcję --bin, ponieważ tworzymy program binarny: gdybyśmy tworzyli bibliotekę, pominęlibyśmy ją. Rzućmy okiem na to, co wygenerował dla nas Cargo:

$ cd hello_world

$ tree.

.}-- Cargo.toml

└── src

└── main.rs

1 directory, 2 files

Jeśli nie masz zainstalowanego polecenia tree, prawdopodobnie możesz je pobrać za pomocą programu obsługi pakietów swojej dystrybucji. Nie jest to konieczne, ale na pewno przydatne. To wszystko, czego potrzebujesz, aby zacząć. Najpierw spójrzmy na nasz Cargo.toml:

[package]

name = "hello_world"

version = "0.0.1"

authors = ["Your Name <your@example.com>"]

Załadowałem ten plik z wartościami domyślnymi na podstawie podanych argumentów i globalnych ustawień git. Możesz również zauważyć, że Cargo zainicjował katalog hello_world jako repozytorium git. Oto zawartość src / main.rs :

fn main ( ) {

println! ("Hello, world!");

}

Ładunek wygenerował komunikat „Witaj, świecie!” dla nas jesteś gotowy, aby zacząć pocierać łokcie. Cargo ma swój własny przewodnik, który obejmuje wszystkie jego funkcje w znacznie głębszy sposób. Teraz, gdy nauczyliśmy się narzędzi, zacznijmy uczyć się więcej o Rust jako języku. To jest baza, która będzie ci dobrze służyć przez resztę czasu z Rust. Masz dwie możliwości: Zanurz się w projekcie dzięki „Naucz się Rusta” lub zacznij od dołu i pracuj w górę dzięki „Składni i semantyce”. Bardziej doświadczeni programiści systemowi prawdopodobnie woleliby „Learn Rust”, podczas gdy ci z języków dynamicznych również byliby zachwyceni. Różni ludzie uczą się inaczej! Wybierz to, co jest dla Ciebie najlepsze.

Naucz się Rusta

Powitanie! Ta sekcja zawiera kilka samouczków, które nauczą Cię tworzenia projektu w języku Rust. Otrzymasz ogólny zarys, ale przyjrzymy się też szczegółom. Jeśli wolisz bardziej oddolne doświadczenie, sprawdź składnię i semantykę .

Gra w zgadywanie

W naszym pierwszym projekcie zaimplementujemy klasyczny problem programistyczny dla początkujących: grę w zgadywanie. Jak działa gra: Nasz program wygeneruje losową liczbę całkowitą z przedziału od jednego do stu. Poprosi nas o przedstawienie przeczucia. Po podaniu naszego numeru powie nam, czy byliśmy dużo poniżej, czy daleko powyżej. Gdy odgadniemy prawidłową liczbę, pogratulujesz nam. Brzmi dobrze?

Początkowe ustawienia

Stwórzmy nowy projekt. Przejdź do katalogu swojego projektu. Pamiętasz, jak stworzyliśmy naszą strukturę katalogów i Cargo.toml dla hello_world? Mam polecenie, które robi to za nas. Spróbujmy:

$ cd ~ / projects

$ charge new riddles --bin

$ cd riddles

Przekazujemy nazwę naszego projektu new wraz z flagą --bin , ponieważ tworzymy plik binarny zamiast biblioteki. Spójrz na wygenerowany Cargo.toml :

[package]

name = "riddles"

version = "0.1.0"

authors = ["Your Name <your@example.com>"]

Cargo uzyskuje te informacje z Twojego otoczenia. Jeśli nie jest poprawne, popraw je. Wreszcie Cargo wygenerowało komunikat „Witaj, świecie!” dla nas. Spójrz na src / main.rs :

fn main () {

println! ("Hello, world!");

}

Spróbujmy skompilować to, co dostarczył nam Cargo:

$ build charge

Compiling riddles v0.1.0 (file: /// home / you / projects / riddles)

Doskonały! Ponownie otwórz plik src / main.rs. Będziemy pisać cały nasz kod w tym pliku. Zanim przejdziemy dalej, pokażę jeszcze jedno polecenie z Cargo: uruchom . cargo run jest swego rodzaju buildem cargo, ale z tą różnicą, że wykonuje również binarny wyprodukowanie. Spróbujmy:

$ charge run

Compiling riddles v0. 1.0 (file: /// home / you / projects / riddles)

Running `target / debug / riddles`

Hello World!

Świetnie! Polecenie run jest bardzo przydatne, gdy trzeba szybko przejrzeć projekt. Nasza gra jest jednym z tych projektów, będziemy musieli szybko przetestować każdą iterację, zanim przejdziemy do następnej.

Przetwarzanie próby zagadki

Spróbujmy! Pierwszą rzeczą, którą musimy zrobić w naszej grze w zgadywanie, jest umożliwienie naszemu graczowi wprowadzenia próby zgadywania. Umieść to w swoim src / main.rs :

use std :: io;

```
fn main () {

println! ("Guess the number!");

println! ("Please enter your hunch.");

let mut hunch = String :: new ();

io :: stdin (). read_line (& mut hunch)

.okay()

.expect ("Failed to read line");

println! ("Your hunch was: {}", hunch);

}
```

Jest tu dużo! Spróbujmy przejść przez to, kawałek po kawałku.

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

Będziemy musieli otrzymać dane wejściowe od użytkownika, a następnie wydrukować wynik jako dane wyjściowe. Z tego powodu potrzebujemy biblioteki io z biblioteki standardowej. Rust ma znaczenie tylko dla kilku rzeczy we wszystkich programach, ten zestaw rzeczy nazywa się „preludium”. Jeśli nie ma go w preludium, będziesz musiał wywołać go bezpośrednio przez użycie.

```
 fn main() {
```

Jak widzieliśmy wcześniej, funkcja main() jest punktem wejścia do programu. Składnia fn deklaruje nową funkcję, znaki () wskazują, że nie ma argumentów, a { rozpoczyna się treść funkcji. Ponieważ nie uwzględniamy zwracanego typu, zakłada się, że jest to () pusta krotka .

```
println! ("Guess the number!");

println! ("Please enter your hunch.");
```

Wcześniej dowiedzieliśmy się, że println! () Jest makrem, które wyświetla ciąg znaków na ekranie.v

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

Teraz robimy się ciekawi! W tej małej kolejce dzieje się bardzo dużo. Pierwszą rzeczą, na którą należy zwrócić uwagę, jest to, że jest to instrukcja let używana do tworzenia zmiennych. Ma postać:v

```
let foo = bar;
```

Spowoduje to utworzenie nowej zmiennej o nazwie foo i powiązanie jej z paskiem wartości. W wielu językach nazywa się to „zmienną”, ale zmienne Rusta mają kilka asów w rękawie. Na przykład są one domyślnie niezmienne. Właśnie dlatego w naszym przykładzie użyto mut : powoduje to zmienne wiązanie, a nie niezmienne. let nie tylko bierze nazwę z lewej strony, ale akceptuje „wzór”. Wzory wykorzystamy nieco później. Na razie wystarczy użyć:

```
let foo = 5 ; // immutable.

let mut bar = 5 ; // mutable
```

Ach, // rozpocznij komentarz, aż do końca linii. Rust ignoruje wszystko w komentarzach

Wiemy więc, że let mut przeczucie wprowadzi mutowalne powiązanie o nazwie przeczucie , ale musimy spojrzeć na drugą stronę = , aby wiedzieć, z czym jest ono powiązane: String :: new () .

String to typ ciągu znaków, dostarczany przez standardową bibliotekę. Ciąg to segment tekstu zakodowany w UTF-8, który może rosnąć. Składnia :: new () wykorzystuje :: ponieważ jest to „funkcja powiązana" określonego typu. Innymi słowy, jest powiązany z samym String, a nie z konkretną instancją String . Niektóre języki nazywają to „metodą statyczną".

Ta funkcja nazywa się new() , ponieważ tworzy nowy pusty String . Funkcję new () znajdziesz na wielu typach, ponieważ jest to popularna nazwa tworzenia nowej wartości pewnego typu. Kontynuujmy:

io :: stdin (). read_line (& mut hunch)

.okay()

.expect ("Line reading failure");

Kolejna kupa! Idźmy kawałek po kawałku. Pierwsza linia składa się z dwóch części. Oto pierwszy:

io :: stdin ()

Pamiętasz, jak używamy use w std :: io w pierwszej linii naszego programu? Teraz wywołujemy powiązaną funkcję w std :: io . Gdybyś nie użył use std :: io , moglibyśmy zapisać tę linię jako std :: io :: stdin () . Ta konkretna funkcja zwraca uchwyt do standardowego wejścia twojego terminala. Dokładniej, std :: io :: Stdin . Następna część użyje tego uchwytu, aby uzyskać dane wejściowe użytkownika:

.read_line (& mut hunch)

Tutaj wywołujemy metodę read_line() na naszym uchwycie. Metody są podobne do powiązanych funkcji, ale są dostępne tylko w konkretnym wystąpieniu typu, a nie w samym typie. Przekazujemy również argument do read_line() : & mut hunch . Pamiętasz, kiedy stworzyliśmy hunch? Powiedzieliśmy, że jest zmienny. Jednak read_line nie akceptuje String jako argumentu: akceptuje & mut String . Rust ma funkcję zwaną „referencjami", która pozwala na posiadanie wielu odniesień do fragmentu danych, zmniejszając w ten sposób potrzebę kopiowania. Referencje to złożona funkcja, ponieważ jedną z najmocniejszych zalet Rusta jest to, jak łatwe i bezpieczne jest korzystanie z referencji. Na razie nie musimy wiedzieć zbyt wiele o tych szczegółach, aby zakończyć nasz program. Wszystko, co musimy wiedzieć w tej chwili, to to, że po prostu podobnie jak wiązania let, referencje są domyślnie niezmienne. W konsekwencji musimy napisać & mut przeczucie zamiast &hunch. Ponieważ read_line () akceptuje zmienne odwołanie do ciągu znaków. Twoim zadaniem jest wzięcie tego, co użytkownik wprowadzi w standardowym wejściu, i umieszczenie tego w ciągu znaków. Z tego powodu przyjmuje ten ciąg jako argument, a ponieważ musi dodać dane wejściowe użytkownika, musi być zmienny. Jeszcze nie skończyliśmy z tą linią. Chociaż jest to pojedyncza linia tekstu, jest to tylko pierwsza część kompletnej logicznej linii kodu:

.okay()

.expect ("Line reading failure");

Gdy wywołujesz metodę ze składnią .foo(), możesz wprowadzić znak końca wiersza i kolejną spację. Pomaga to w dzieleniu długich linii. Mogliśmy napisać:

io :: stdin (). read_line (& mut hunched) .ok (). expect ("Line reading failed");

Ale to jest trudniejsze do odczytania. Podzieliliśmy go więc na trzy linie dla trzech wywołań metod. Mówiliśmy już o read_line() , ale co z ok() i expect() ? Wspomnieliśmy już, że read_line() umieszcza dane wejściowe użytkownika w dostarczonym przez nas łańcuchu & mut. Ale zwraca również wartość: w tym przypadku io :: Result . Rust ma wiele typów zwanych Result w swojej standardowej bibliotece: ogólną Result i specyficzne wersje dla podbibliotek, takie jak io :: Result .

Celem tych wyników jest zakodowanie informacji o obsłudze błędów. Wartości typu Result mają w sobie zdefiniowane metody. W tym przypadku io :: Result ma metodę ok(), co przekłada się na „chcemy założyć, że ta wartość jest pomyślna". Jeśli nie, odrzuć informację o błędzie. ' Po co odrzucać informację o błędzie? W przypadku programu podstawowego chcemy po prostu wydrukować ogólny błąd, dowolny problem, który oznacza, że nie możemy kontynuować. Metoda ok () zwraca wartość, która ma inną metodę zdefiniowaną w: seek() . Metoda spodziewaj się () przyjmuje wartość, przy której została wywołana, a jeśli nie jest to pomyślna wartość, panikuje! z przekazaną przez nas wiadomością. Panika! jak to spowoduje nagłe zakończenie programu (awarię), wyświetlając ten komunikat. Jeśli usuniemy wywołania tych dwóch metod, nasz program się skompiluje, ale otrzymamy ostrzeżenie:

$ build charge

Compiling riddles v0. 1.0 (file: /// home / you / projects / riddles)

src / main.rs: 10 : 5 : 10 : 44 warning: unused result which must be used,

[warn (unused_must_use)] on by default

src / main.rs: 10 io :: stdin (). read_line (& mut hunch);

Rust informuje nas, że nie użyliśmy wartości Result . To ostrzeżenie pochodzi ze specjalnej adnotacji, że io :: Result ma . Rust próbuje ci powiedzieć, że nie poradziłeś sobie z możliwym błędem. Właściwym sposobem wyeliminowania błędu jest napisanie kodu obsługującego błędy. Na szczęście, jeśli chcemy zakończyć wykonywanie programu tylko w przypadku wystąpienia problemu, możemy skorzystać z tych dwóch małych metod. Gdybyśmy mogli jakoś naprawić błąd, zrobilibyśmy coś innego, ale zostawmy to na przyszły projekt. Pozostała nam tylko jedna linia z tego pierwszego przykładu:v

println! ("Your hunch was: {}", hunch);

}

Ta linia drukuje ciąg znaków, w którym zapisujemy nasz wpis. {} s to symbole zastępcze, dlatego jako argument przekazujemy zagadkę. Gdyby było wiele {} s, powinniśmy przekazać wiele argumentów:

let x = 5 ;

let y = 10 ;

println! ("xyy: {} y {}" , x, y);

Łatwe. Tak czy inaczej, to jest wycieczka. Możemy go uruchomić za pomocą opłaty za uruchomienie:

$ charge run

Compiling guessing_game v0. 1.0 (file: /// home / you / projects / riddles)

Running `target / debug / riddles`

Guess the number!

Please enter your hunch.

6

Your hunch was: 6

Gratulacje! Nasza pierwsza część dobiegła końca: możemy pobrać dane z klawiatury i wydrukować je z powrotem.

Generowanie tajnego numeru

Następnie musimy wygenerować tajny numer. Rust nie zawiera jeszcze funkcji liczb losowych w standardowej bibliotece. Jednak zespół Rusta zapewnia skrzynię rand . „Skrzynia" to pakiet kodu Rusta. Budujemy „skrzynię binarną", która jest plikiem wykonywalnym. Rand to „biblioteka skrzynek", która zawiera kod do wykorzystania przez inne programy. Używanie skrzyń zewnętrznych to miejsce, w którym Cargo naprawdę błyszczy. Zanim będziemy mogli napisać kod używający rand, musimy zmodyfikować nasz plik Cargo.toml. Otwórz go i dodaj te linie na końcu:

[dependencies]

rand = "0.3.0"

Sekcja [dependencies] w Cargo.toml przypomina sekcję [package]: wszystko, co następuje, jest jej częścią, aż do rozpoczęcia następnej sekcji. Cargo korzysta z sekcji zależności, aby dowiedzieć się, od których zewnętrznych skrzynek jesteśmy uzależnieni, a także od wymaganych wersji. W tym przypadku użyliśmy wersji 0.3.0 . Cargo obsługuje wersjonowanie semantyczne, które jest standardem zapisywania numerów wersji. Gdybyśmy chcieli użyć najnowszej wersji, moglibyśmy użyć * lub szeregu wersji. Dokumentacja Cargo zawiera więcej szczegółów. Teraz, nie zmieniając niczego w naszym kodzie, zbudujmy nasz projekt:

$ build charge

Updating registry `https: // github.com / rust-lang / crates.io-index`

Downloading rand v0. 3.8

Downloading libc v0. 1.6

Compiling libc v0. 1.6

Compiling rand v0. 3.8

Compiling riddles v0. 1.0 (file: /// home / you / projects / riddles)

(Oczywiście można zobaczyć różne wersje.)

Dużo więcej produkcji! Teraz, gdy mamy zewnętrzną zależność, Cargo pobiera najnowsze wersje wszystkiego z rejestru, który może kopiować dane z Crates.io. Crates.io to miejsce, w którym ludzie w ekosystemie Rust publikują swoje projekty open source, z których mogą korzystać inni. Po zaktualizowaniu rejestru Cargo sprawdza nasze zależności (w [dependencies]) i pobiera je, jeśli jeszcze ich nie mają. W tym przypadku właśnie powiedzieliśmy, że chcemy polegać na rand i otrzymaliśmy również kopię libc. Dzieje się tak dlatego, że sam Rand zależy od libc do działania. Po pobraniu zależności Cargo je kompiluje, aby później skompilować nasz kod. Jeśli uruchomimy cargo build , otrzymamy inny wynik:

$ build charge

To prawda, nie ma wyjścia! Cargo wie, że nasz projekt został zbudowany, a także wszystkie jego zależności, więc nie ma powodu, aby robić cały proces od nowa. Nie mając nic do roboty, po prostu biegnij. Jeśli ponownie otworzymy src/main.rs, dokonamy banalnej zmiany, zapiszemy zmiany, zobaczymy tylko jedną linię:

$ build chargé

Compiling riddles v0. 1.0 (file: /// home / you / projects / riddles)

Powiedzieliśmy Cargo, że chcemy dowolnej wersji 0.3.x rand , a on pobrał najnowszą wersję w momencie pisania tego samouczka, v0.3.8 . Ale co się stanie, gdy zostanie wydana następna wersja v0.3.9 z poważnymi poprawkami błędów? Chociaż otrzymywanie poprawek błędów jest ważne, co się stanie, jeśli wersja 0.3.9 zawiera regresję, która psuje nasz kod? Odpowiedzią na ten problem jest plik Cargo.lock, który znajdziesz w katalogu swojego projektu. Kiedy tworzysz swój projekt po raz pierwszy, cargo określa wszystkie wersje spełniające Twoje kryteria i zapisuje je w pliku Cargo.lock . Kiedy będziesz budować swój projekt w przyszłości, Cargo zauważy, że istnieje plik Cargo.lock i użyje określonych w nim wersji, zamiast wykonywać całą pracę polegającą na ponownym określaniu wersji. Pozwala to na uzyskanie automatycznie powtarzalnej konstrukcji. Innymi słowy, pozostaniemy przy wersji 0.3.8, dopóki jawnie nie prześlemy wersji, w ten sam sposób zrobią to ludzie, którym udostępniliśmy nasz kod, dzięki plikowi Cargo.lock. Ale co się dzieje, gdy chcemy użyć wersji 0.3.9? Cargo ma inne polecenie, update , które tłumaczy się jako „zignoruj blokadę i określ wszystkie najnowsze wersje, które pasują do tego, co określiliśmy”. Jeśli to zadziała, zapisz te wersje w pliku blokady Cargo.lock . Jednak domyślnie Cargo będzie wyszukiwać tylko wersje większe niż 0.3.0 i mniejsze niż 0.4.0 . Jeśli chcielibyśmy przejść na wersję 0.4.x , musielibyśmy bezpośrednio zaktualizować plik Cargo.toml. Gdy to zrobimy, następnym razem, gdy uruchomimy kompilację ładunku, Cargo zaktualizuje indeks i ponownie oceni nasze wymagania dotyczące randów. O Cargo i jego ekosystemie można powiedzieć znacznie więcej, ale na razie to wszystko, co musimy wiedzieć. Cargo naprawdę ułatwia ponowne wykorzystanie bibliotek, a Russetters mają tendencję do pisania małych projektów, które są zbudowane z mniejszego zestawu pakietów. Użyjmy teraz rand . Oto nasz kolejny krok:

```
extern crate rand;

use std :: io;

use rand :: Rng;

fn main () {

println! ("Guess the number!");

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

println! ("The secret number is: {}", secret_number);

println! ("Please enter your hunch.");

let mut hunch = String :: new ();

io :: stdin (). read_line (& mut hunch)

.okay()

.expect ("Failed to read line");
```

println! ("Your hunch was: {}", hunch);

}

Pierwszą rzeczą, którą zrobiliśmy, jest zmiana pierwszej linii. Teraz mówi extern crate Rand . Ponieważ deklarujemy rand w naszej sekcji [dependencies] , możemy użyć extern crate , aby Rust wiedział, że będziemy używać rand . Jest to równoważne użyciu rand; , abyśmy mogli wykorzystać wszystko, co znajduje się w skrzyni rand poprzez przedrostek rand :: . Następnie dodaliśmy kolejną linię use: use rand :: Rng . Za kilka chwil będziemy używać metody, a to wymaga dostępności Rng, aby mogła działać. Podstawowa idea jest taka: metody są wewnątrz czegoś, co nazywa się „cechami", a aby metoda działała, potrzebujesz, aby cecha była dostępna. Aby uzyskać więcej informacji, przejdź do sekcji Cechy. W środku znajdują się jeszcze dwie linie:

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

println! ("The secret number is: {}", secret_number);

Używamy funkcji rand :: thread_rng(), aby uzyskać kopię generatora liczb losowych, która jest lokalna dla wątku wykonania, w którym się znajdujemy. Ponieważ udostępniliśmy rand :: Rng poprzez użycie rand :: Rng , dostępna jest metoda gen_range (). Ta metoda przyjmuje dwa argumenty i generuje losową liczbę między nimi. Dolna granica jest inkluzywna, ale górna jest wyłączna, więc potrzebujemy 1 i 101, aby otrzymać liczbę od jednego do stu. Druga linia drukuje tylko tajny numer. Jest to przydatne, gdy rozwijamy nasz program, abyśmy mogli go przetestować. Usuniemy tę linię w ostatecznej wersji. To nie jest gra, jeśli wydrukujesz odpowiedź zaraz po uruchomieniu! Spróbuj uruchomić program kilka razy:

$ charge run

Compiling riddles v0. 1.0 (file: /// home / you / projects / riddles)

Running `target / debug / riddles`

Guess the number!

The secret number is: 7

Please enter your hunch.

4

Your hunch was: 4

$ charge run

Running `target / debug / riddles`

Guess the number!

The secret number is: 83

Please enter your hunch.

5

Your hunch was: 5

Dalej: porównajmy naszą zagadkę z sekretną liczbą. Porównanie zagadek. Teraz, gdy mamy dane wprowadzone przez użytkownika, porównajmy zagadkę z naszym tajnym numerem. Oto nasz kolejny krok, chociaż nadal nie działa całkowicie:

```
extern crate rand;

use std :: io;

use std :: cmp :: Ordering;

use rand :: Rng;

fn main () {

println! ("Guess the number!");

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

println! ("The secret number is: {}", secret_number);

println! ("Please enter your hunch.");

let mut hunch = String :: new ();

io :: stdin (). read_line (& mut hunch)

.okay()

.expect ("Failed to read line");

println! ("Your hunch was: {}", hunch);

match hunch.cmp (& secret_number) {

Ordering :: Less => println! ("Very small!"),

Ordering :: Greater => println! ("Very big!"),

Ordering :: Equal => println! ("You won!"),

}

}
```

Kilka kawałków tutaj. Pierwszy to inne zastosowanie. Udostępniliśmy typ o nazwie std :: cmp :: Ordering . Następnie pięć nowych dolnych linii, które go używają:

```
match hunch.cmp (& secret_number) {

Ordering :: Less => println! ("Very small!"),

Ordering :: Greater => println! ("Very big!"),

Ordering :: Equal => println! ("You won!"),

}
```

Metodę cmp () można wywołać do wszystkiego, co można porównać, wymaga odniesienia do rzeczy, z którą chcesz porównać. Zwraca udostępniony wcześniej typ Ordering. Użyliśmy instrukcji

dopasowania, aby dokładnie określić, jaki to rodzaj Zamówienia. Ordering to enum , skrót od „enumeration”, który wygląda tak:

enum foo {

Pub,

Baz,

}

Przy tej definicji wszystko typu Foo może być albo Foo :: Bar albo Foo :: Baz . Używamy ::, aby wskazać przestrzeń nazw dla określonego wariantu enum. Wyliczenie Ordering ma trzy możliwe warianty: Less , Equal i Greater (odpowiednio minor, equal i major). Instrukcja dopasowania przyjmuje wartość typu i umożliwia utworzenie „ramienia” dla każdej możliwej wartości. Ponieważ mamy trzy możliwe typy Orderingu, mamy trzy ramiona:

match guess.cmp (& secret_number) {

Ordering :: Less => println! ("Very small!"),

Ordering :: Greater => println! ("Very big!"),

Ordering :: Equal => println! ("You won!"),

}

Jeśli jest Mniej, drukujemy za mały! , jeśli jest większy, za duży! , a jeśli jest równe , wygrałeś! . match jest naprawdę przydatne i jest często używane w Rust. Wcześniej wspomniałem, że nadal nie działa. Przetestujmy to:

$ build charge

Compiling riddles v0. 1.0 (file: /// home / you / projects / riddles)

src / main.rs: 28 : 21 : 28 : 35 error: mismatched types:

expected `& collections :: string :: String`,

found `& _`

(expected struct `collections :: string :: String`, found integral variable) [E0308]

src / main.rs: 28 match hunch.cmp (& secret_number) {

error: aborting due to previous error

Could not compile `riddles`.

Ups! Duży błąd. Najważniejsze w tym jest to, że mamy „niedopasowane typy”. Rust ma silny, statyczny system stawek. Jednak ma również wnioskowanie o typie. Kiedy wpisaliśmy let przeczucie = String :: nowy () , Rust był w stanie wywnioskować, że przeczucie musi być typu String , więc nie kazał nam pisać tego typu. W przypadku naszego tajnego numeru istnieje kilka typów, które mogą mieć wartość od jednego do stu: i32 , liczba trzydziestodwubitowa, u32 , liczba trzydziestodwubitowa bez znaku lub i64 liczba sześćdziesięcioczterobitowa numer lub inne. Jak dotąd nie miało to znaczenia, ponieważ Rust domyślnie używa i32 . Jednak w tym przypadku Rust nie wie, jak porównać przeczucie z secret_number. Oba muszą być tego samego typu. Na koniec chcemy przekonwertować ciąg, który odczytujemy jako

dane wejściowe, na liczbę rzeczywistą, dla celów porównawczych. Możemy to zrobić za pomocą trzech dodatkowych linii. Oto nasz nowy program:

```
extern crate rand;

use std :: io;

use std :: cmp :: Ordering;

use rand :: Rng;

fn main () {

println! ("Guess the number!");

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

println! ("The secret number is: {}", secret_number);

println! ("Please enter your hunch.");

let mut hunch = String :: new ();

io :: stdin (). read_line (& mut hunch)

.okay()

.expect ("Failed to read line");

let hunch: u32 = hunch.trim (). parse ()

.okay()

.expect ("Please enter a number!");

println! ("Your hunch was: {}", hunch);

match hunch.cmp (& secret_number) {

Ordering :: Less => println! ("Very small!"),

Ordering :: Greater => println! ("Very big!"),

Ordering :: Equal => println! ("You won!"),

}

}
```

Trzy nowe linie:

```
let hunch: u32 = hunch.trim (). parse ()

.okay()

.expect ("Please enter a number!");
```

Chwileczkę, myślałem, że już mieliśmy przeczucie? Robimy, ale Rust pozwala nam zastąpić poprzednie przeczucie nowym. Jest to często używane w tej samej sytuacji, gdzie przeczucie to String , ale chcemy przekonwertować go na u32 . To cieniowanie pozwala nam ponownie użyć nazwy przeczucie zamiast

zmuszać nas do wymyślenia dwóch unikalnych nazw, takich jak przeczucie_str i przeczucie lub innych. Łączymy przeczucie z wyrażeniem, które wygląda jak coś, co napisaliśmy wcześniej: zgadnij.przytnij (). parse () Po którym następuje wywołanie ok (). Oczekiwać () . Tutaj przeczucie odnosi się do starej wersji, która była łańcuchem znaków zawierającym dane wprowadzone przez użytkownika. Metoda trim () w String s usuwa wszystkie spacje na początku i na końcu naszych ciągów znaków. Jest to ważne, ponieważ musieliśmy nacisnąć klawisz „return", aby spełnić wymagania read_line() . Oznacza to, że jeśli wpiszemy 5 i naciśniemy przeczucie „return", to wygląda to tak: 5 \ n . \ n oznacza „nową linię", klawisz Enter. trim() pozbywa się tego, pozostawiając nasz ciąg tylko na 5 . Metoda parse () na ciągach znaków przetwarza ciąg znaków na pewien typ liczb. Ponieważ może analizować różne liczby, musimy dać Rustowi wskazówkę co do dokładnego typu liczby, której potrzebujemy. Stąd część przeczucia niech: u32 . Dwukropek (:) po przeczuciu powie Rustowi, że zanotujemy typ. u32 jest trzydziestodwubitową liczbą całkowitą bez znaku. Rust ma szereg wbudowane typy liczb , ale wybraliśmy u32 . Jest to dobra opcja domyślna dla małej liczby dodatniej. Podobnie jak read_line() , nasze wywołanie parse() może spowodować błąd. Co jeśli nasz ciąg znaków zawiera A ≃ ? Nie byłoby sposobu, aby przekonwertować to na liczbę. Dlatego zrobimy to samo, co zrobiliśmy z read_line (): użyjemy metod ok () i expect(), aby nagle się zakończyły, jeśli wystąpią jakieś błędy. Wypróbujmy nasz program!

$ charge run

Compiling riddles v0. 1.0 (file: /// home / you / projects / riddles)

Running `target / riddles`

Guess the number!

The secret number is: 58

Please enter your hunch.

76

Your hunch was: 76

Very big!

Doskonały! Możesz zobaczyć, że nawet dodałem spacje przed moją próbą, a mimo to program stwierdził, że próbuję 76. Uruchom program kilka razy i sprawdź, czy zgadywanie liczby działa, podobnie jak próba bardzo małej liczby. Teraz większość gry działa, ale możemy spróbować zgadnąć tylko raz. Spróbujmy to zmienić dodając cykle!

Iteracja

Słowo kluczowe loop zapewnia nieskończoną pętlę. Dodajmy: Zgadnij numer! Sekretny numer to: 58 Podaj swoją zagadkę. 76 Twoje przeczucie było: 76 Bardzo duże!

extern crate rand;

use std :: io;

use std :: cmp :: Ordering;

use rand :: Rng;

fn main () {

```
println! ("Guess the number!");

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

println! ("The secret number is: {}", secret_number);

loop {

println! ("Please enter your hunch.");

let mut hunch = String :: new ();

io :: stdin (). read_line (& mut hunch)

.okay()

.expect ("Failed to read line");

let hunch: u32 = hunch.trim (). parse ()

.okay()

.expect ("Please enter a number!");

println! ("Hunch: {}", hunch);

match hunch.cmp (& secret_number) {

Ordering :: Less => println! ("Very small!"),

Ordering :: Greater => println! ("Very big!"),

Ordering :: Equal => println! ("You won!"),

}

}

}
```

Sprawdź to. Ale czekaj, czy nie dodaliśmy właśnie nieskończonej pętli? Tak. Pamiętasz naszą dyskusję o parse()? Jeśli udzielimy odpowiedzi nieliczbowej, zwrócimy (return) i zakończymy wykonanie. Przestrzegać:

```
charge run

Compiling riddles v0. 1.0 (file: /// home / you / projects / riddles)

Running `target / riddles`

Guess the number!

The secret number is: 59

Please enter your hunch.

Four. Five

Your hunch was: 45

Very small!
```

Please enter your hunch.

60

Your hunch was: 60

Very big!

Please enter your hunch.

59

Your hunch was: 59

You won!

Please enter your hunch.

quit

thread '<main>' panicked at 'Please type a number!'

Ha! quit w efekcie kończy wykonanie. Jak również każdy inny wpis, który nie jest liczbą. Cóż, jest to co najmniej nieoptymalne. Wyjdźmy pierwsi, gdy wygramy:

```
extern crate rand;

use std :: io;

use std :: cmp :: Ordering;

use rand :: Rng;

fn main () {

println! ("Guess the number!");

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

println! ("The secret number is: {}", secret_number);

loop {

println! ("Please enter your hunch.");

let mut hunch = String :: new ();

io :: stdin (). read_line (& mut hunch)

.okay()

.expect ("Failed to read line");

let hunch: u32 = hunch.trim (). parse ()

.okay()

.expect ("Please enter a number!");

println! ("Your hunch was: {}", hunch);
```

```
match hunch.cmp (& secret_number) {

Ordering :: Less => println! ("Very small!"),

Ordering :: Greater => println! ("Very big!"),

Ordering :: Equal => {

println! ("You won!");

break;

}

}

}

}
```

Dodając linię przerwy po „Wygrałeś!”, Przerwiemy cykl, gdy wygramy. Wyjście z pętli oznacza również wyjście z programu, ponieważ jest to ostatnia rzecz w main() . Pozostało nam tylko jedno ulepszenie: gdy ktoś wprowadzi wartość nienumeryczną, nie chcemy kończyć biegu, chcemy go po prostu zignorować. Możemy to zrobić w następujący sposób:

```
extern crate rand;

use std :: io;

use std :: cmp :: Ordering;

use rand :: Rng;

fn main () {

println! ("Guess the number!");

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

println! ("The secret number is: {}", secret_number);

loop {

println! ("Please enter your hunch.");

let mut hunch = String :: new ();

io :: stdin (). read_line (& mut hunch)

.okay()

.expect ("Failed to read line");

let hunch: u32 = match hunch.trim (). parse () {

Ok (num) => num,

Err (_) => continue,

};
```

```
println! ("Your hunch was: {}", hunch);

match hunch.cmp (& secret_number) {

Ordering :: Less => println! ("Very small!"),

Ordering :: Greater => println! ("Very big!"),

Ordering :: Equal => {

println! ("You won!");

break;

}

}

}

}
```

Oto linie, które uległy zmianie:

```
let hunch: u32 = match hunch.trim (). parse () {

Ok (num) => num,

Err (_) => continue,

};
```

W ten sposób przeszliśmy od „nagłego zakończenia błędu” do „efektywnej obsługi błędu”, poprzez zmianę ok (). Expect() do instrukcji dopasowania . Result zwrócony przez parse() jest enum podobnie jak Ordering , ale w tym przypadku każdy wariant ma powiązane dane: Ok jest pomyślny, a Err jest niepowodzeniem. Każda zawiera więcej informacji: przeanalizowaną liczbę całkowitą w pomyślnym przypadku lub typ błędu. W tym przypadku dopasowujemy Ok (num) , co przypisuje wewnętrzną wartość Ok do name num , a następnie powraca po prawej stronie. W przypadku Err nie obchodzi nas jaki to błąd, dlatego zamiast nazwy używamy _. To ignoruje błąd i kontynuuje przenosi nas do następnej iteracji pętli. Teraz powinno być dobrze! Spróbujmy:

```
$ charge run

Compiling riddles v0. 1.0 (file: /// home / you / projects / riddles)

Running `target / riddles`

Guess the number!

The secret number is: 61

Please enter your hunch.

10

Your hunch was: 10

Very small!

Please enter your hunch.
```

99

Your hunch was: 99

Very small!

Please enter your hunch.

foo

Please enter your hunch.

61

Your hunch was: 61

You won!

Świetnie! Ostatnim ulepszeniem kończymy grę logiczną. Czy możesz sobie wyobrazić, co to jest? To prawda, nie chcemy drukować tajnego numeru. To było dobre do testów, ale rujnuje naszą grę. Oto nasz ostateczny kod źródłowy:

```
extern crate rand;

use std :: io;

use std :: cmp :: Ordering;

use rand :: Rng;

fn main () {

println! ("Guess the number!");

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

loop {

println! ("Please enter your hunch.");

let mut hunch = String :: new ();

io :: stdin (). read_line (& mut hunch)

.okay()

.expect ("Failed to read line");

let hunch: u32 = match hunch.trim (). parse () {

Ok (num) => num,

Err (_) => continue,

};

println! ("Your hunch was: {}", hunch);

match hunch.cmp (& secret_number) {

Ordering :: Less => println! ("Very small!"),
```

```
Ordering :: Greater => println! ("Very big!"),

Ordering :: Equal => {

println! ("You won!");

break;

}

}

}

}
```

Zakończone!

W tym momencie pomyślnie ukończyłeś grę w zgadywanie! Gratulacje! Ten pierwszy projekt wiele Cię nauczył: let , match , metody, powiązane funkcje, korzystanie z zewnętrznych skrzynek i nie tylko. Nasz następny projekt pokaże jeszcze więcej.

Obiad filozofów

W naszym drugim projekcie przyjrzyjmy się klasycznemu problemowi współbieżności. Nazywa się „Obiad Filozofów”. Został pierwotnie wymyślony przez Dijkstrę w 1965 r., ale użyjemy nieco zmodyfikowanej wersji tego artykułu Tony'ego Hoare'a z 1985 r. W starożytności bogaty filantrop założył uniwersytet, w którym mieszkało pięciu wybitnych filozofów. Każdy filozof miał pokój, w którym mógł wykonywać swoją zawodową działalność myślową: była też wspólna jadalnia, wyposażona w okrągły stół otoczony pięcioma krzesłami, z których każdy był identyfikowany z imieniem siedzącego na nim filozofa. Filozofowie usiedli wokół stołu w kierunku przeciwnym do ruchu wskazówek zegara. Po lewej stronie każdego filozofa leżał złoty widelec, a pośrodku miska spaghetti, którą ciągle uzupełniano. Oczekiwano, że filozof spędza większość czasu na myśleniu; ale kiedy byli głodni, idź do jadalni, weź widelec po lewej stronie i zanurz go w spaghetti. Ale taka była splątana natura spaghetti, że drugi widelec był wymagany, aby doprowadzić go do ust. Filozof musiał więc również wziąć widelec po swojej prawej stronie. Kiedy skończyli, musieli opuścić oba widelce, wstać z krzesła i dalej myśleć. Oczywiście widelec może być używany tylko przez jednego filozofa na raz. Jeśli chce tego inny filozof, musi poczekać, aż widelec będzie znów dostępny. Ten klasyczny problem wykazuje pewne elementy współbieżności. Powodem tego jest to, że jest to faktycznie trudne rozwiązanie do wdrożenia: prosta implementacja może spowodować zakleszczenie. Rozważmy na przykład prosty algorytm, który mógłby rozwiązać ten problem:

1. Filozof bierze widelec po swojej lewej stronie.

2. Następnie weź widelec po prawej stronie.

3. Jedz.

4. Opuść widelec.

A teraz wyobraźmy sobie taką sekwencję zdarzeń:

1. Filozof 1 rozpoczyna algorytm, biorąc widelec po swojej lewej stronie.

2. Filozof 2 rozpoczyna algorytm, biorąc rozwidlenie po swojej lewej stronie.

3. Filozof 3 rozpoczyna algorytm, biorąc rozwidlenie po swojej lewej stronie.

4. Filozof 4 rozpoczyna algorytm, biorąc rozwidlenie po swojej lewej stronie.

5. Filozof 5 rozpoczyna algorytm, biorąc rozwidlenie po swojej lewej stronie.

6. ...? Wszystkie widelce zostały zabrane, ale nikt nie może jeść!

Istnieją różne sposoby rozwiązania tego problemu. Poprowadzimy Cię przez rozwiązanie tego samouczka. Na razie zacznijmy od modelowania problemu. Zacznijmy od filozofów:

```
struct Philosopher {

name: String ,

}

impl Filosofo {

fn new (name: & str ) -> Philosopher {

Philosopher {

name: name.to_string (),

}

}

}

fn main () {

let f1 = Philosopher :: new ( "Judith Butler" );

let f2 = Philosopher :: new ( "Gilles Deleuze" );

let f3 = Philosopher :: new ( "Karl Marx" );

let f4 = Philosopher :: new ( "Emma Goldman" );

let f5 = Philosopher :: new ( "Michel Foucault" );

}
```

Tutaj tworzymy strukturę (struct) reprezentującą filozofa. Na razie wystarczy nazwa. Wybieramy typ String dla nazwy zamiast & str . Ogólnie rzecz biorąc, praca z typem, który jest właścicielem (posiada) twoje dane, jest łatwiejsza niż praca z typem, który używa referencji. Kontynuujmy:

```
# struct Philosopher {

# name: String ,

#}

impl Filosofo {

fn new (name: & str ) -> Philosopher {

Philosopher {
```

name: name.to_string (),

}

}

}

Ten blok impl pozwala nam definiować rzeczy w strukturach Philosopher. W tym przypadku definiujemy „powiązaną funkcję” o nazwie new . Pierwsza linia wygląda następująco:

struct Philosopher {

name: String ,

#}

impl Philosopher {

fn new (name: & str) -> Philosopher {

Philosopher {

name: name.to_string (),

#}

#}

#}

Otrzymujemy argument name , typu & str . Odwołanie do innego ciągu znaków. Spowoduje to zwrócenie instancji naszej struktury Philosopher.

struct Philosopher {

name: String ,

#}

impl Philosopher {

fn new (name: & str) -> Philosopher {

Philosopher {

name: name.to_string (),

}

#}

#}

Powyższe tworzy nowego Filozofa i przypisuje nasz argument name do pola name . Nie sam argument, ponieważ wywołujemy na nim .to_string(). Który tworzy kopię łańcucha wskazującego na nasz & str i daje nam nowy String , który jest typem nazwy pola Filosofo . Dlaczego nie zaakceptować ciągu znaków bezpośrednio? Łatwiej jest zadzwonić. Gdybyśmy otrzymali String, ale dzwoniący miał & str, byłby zmuszony wywołać .to_string() po swojej stronie. Wadą tej elastyczności jest to, że zawsze robimy kopię. W przypadku tego małego programu nie jest to szczególnie ważne, a wiemy, że i tak będziemy

używać krótkich łańcuchów. Ostatnia rzecz, którą być może zauważyłeś: definiujemy tylko Filozofa i wydaje się, że nic z tym nie robimy. Rust jest językiem „opartym na wyrażeniach”, co oznacza, że prawie wszystko w Rust jest wyrażeniem, które zwraca wartość. Dotyczy to również funkcji, ostatnie wyrażenie jest zwracane automatycznie. Ponieważ tworzymy nowego Filozofa jako ostatnie wyrażenie tej funkcji, ostatecznie zwracamy go. Nazwa new() nie jest niczym specjalnym dla Rusta, ale jest konwencją dla funkcji, które tworzą nowe instancje struktur. Zanim porozmawiamy o tym, dlaczego, przyjrzyjmy się ponownie funkcji main():

```
# struct Philosopher {
# name: String ,
#}
#
# impl Philosopher {
# fn new (name: & str ) -> Philosopher {
# Philosopher {
# name: name.to_string (),
#}
#}
#}
#
fn main () {
let f1 = Philosopher :: new ( "Judith Butler" );
let f2 = Philosopher :: new ( "Gilles Deleuze" );
let f3 = Philosopher :: new ( "Karl Marx" );
let f4 = Philosopher :: new ( "Emma Goldman" );
let f5 = Philosopher :: new ( "Michel Foucault" );
}
```

Tutaj tworzymy pięć zmiennych z pięcioma nowymi filozofami. To jest moja piątka ulubionych, ale możesz ich zastąpić kimkolwiek wolisz. Gdybyś nie zdefiniował funkcji new (), funkcja main() wyglądałaby tak:

```
# struct Philosopher {
# name: String ,
#}
fn main () {
let f1 = Philosopher {name: "Judith Butler" .to_string ()};
```

```
let f2 = Philosopher {name: "Gilles Deleuze" .to_string ()};

let f3 = Philosopher {name: "Karl Marx" .to_string ()};

let f4 = Philosopher {name: "Emma Goldman" .to_string ()};

let f5 = Philosopher {name: "Michel Foucault" .to_string ()};

}
```

Trochę głośniej. Korzystanie z new ma również inne zalety, ale nawet w tym prostym przypadku okazuje się lepsze. Teraz, gdy mamy już podstawy, istnieje wiele sposobów na zaatakowanie większego problemu. Lubię zaczynać od końca: stwórzmy sposób, w jaki każdy filozof skończy jeść. Jako mały krok stwórzmy metodę, a następnie przejrzyjmy wszystkich filozofów, którzy ją nazywają:

```
struct Philosopher {

name: String ,

}

impl Filosofo {

fn new (name: & str ) -> Philosopher {

Philosopher {

name: name.to_string (),

}

}

fn eat (& self ) {

println! ( "{} has finished eating." , self .name);

}

}

fn main () {

let philosophers = vec! [

Philosopher :: new ( "Judith Butler" ),

Philosopher :: new ( "Gilles Deleuze" ),

Philosopher :: new ( "Karl Marx" ),

Philosopher :: new ( "Emma Goldman" ),

Philosopher :: new ( "Michel Foucault" ),

];

for f in & philosophers {

f.comer ();
```

}

}

Najpierw spójrzmy na main() . Zamiast mieć pięć indywidualnych zmiennych dla naszych filozofów, tworzymy Vec < T > . Vec < T > jest również nazywany „wektorem” i jest tablicą zdolną do wzrostu. Następnie używamy pętli [for] [for] do iteracji po wektorze, uzyskując jednocześnie odniesienie do każdego filozofa. W ciele pętli wywołujemy f.comer (); , który jest zdefiniowany jako:

fn eat (& self) {

println! ("{} has finished eating.", self.name);

}

W Rust metody otrzymują jawny parametr self . Właśnie dlatego jedzenie () jest metodą, a new jest powiązaną funkcją: new () nie ma self . W naszej pierwszej wersji jedzenia () wydrukowaliśmy tylko imię filozofa i wspomnieliśmy, że skończył jeść. Uruchomienie tego programu powinno wygenerować następujące dane wyjściowe:

Judith Butler has finished eating.

Gilles Deleuze has finished eating.

Karl Marx has finished eating.

Emma Goldman has finished eating.

Michel Foucault has finished eating.

Bardzo proste, wszyscy skończyli jeść! Ale nie zaimplementowaliśmy jeszcze prawdziwego problemu, więc jeszcze nie skończyliśmy! Następnie nie tylko chcemy po prostu skończyć jedzenie, ale faktycznie jemy. Oto następna wersja:

use std :: thread;

struct Philosopher {

name: String ,

}

impl Filosofo {

fn new (name: & str) -> Philosopher {

Philosopher {

name: name.to_string (),

}

}

fn eat (& self) {

println! ("{} is eating." , self .name);

thread :: sleep_ms (1000);

```
println! ( "{} has finished eating." , self .name);
}
}
fn main () {
let philosophers = vec! [
Philosopher :: new ( "Judith Butler" ),
Philosopher :: new ( "Gilles Deleuze" ),
Philosopher :: new ( "Karl Marx" ),
Philosopher :: new ( "Emma Goldman" ),
Philosopher :: new ( "Michel Foucault" ),
];
for f in & philosophers {
f.comer ();
}
}
```

Tylko kilka zmian. Przeanalizujmy je część po części.

```
use std :: thread;
```

use udostępnia nazwy w naszym zakresie. Zaczniemy korzystać z modułu wątku -ze standardowej biblioteki i dlatego musimy go użyć.

```
fn eat (& self) {
println! ("{} is eating.", self.name);
thread :: sleep_ms (1000);
println! ("{} has finished eating.", self.name);
}
```

Teraz drukujemy dwie wiadomości, z sleep_ms () pośrodku. Co symuluje czas potrzebny filozofowi na zjedzenie posiłku. Jeśli uruchomisz ten program, powinieneś zobaczyć, jak każdy filozof je na raz:

```
Judith Butler is eating.
Judith Butler has finished eating.
Gilles Deleuze is eating.
Gilles Deleuze has finished eating.
Karl Marx is eating.
Karl Marx has finished eating.
```

Emma Goldman is eating.

Emma Goldman has finished eating.

Michel Foucault is eating.

Michel Foucault has finished eating.

Doskonały! Idziemy do przodu. Jest tylko jeden szczegół: nie działamy równolegle, co jest kluczowe dla naszego problemu! Aby nasi filozofowie jedli w tym samym czasie, musimy wprowadzić małą zmianę. Oto następna iteracja:

```
use std :: thread;

struct Philosopher {

name: String ,

}

impl Filosofo {

fn new (name: & str ) -> Philosopher {

Philosopher {

name: name.to_string (),

}

}

fn eat (& self ) {

println! ( "{} is eating." , self .name);

thread :: sleep_ms ( 1000 );

println! ( "{} has finished eating." , self .name);

}

}

fn main () {

let philosophers = vec! [

Philosopher :: new ( "Judith Butler" ),

Philosopher :: new ( "Gilles Deleuze" ),

Philosopher :: new ( "Karl Marx" ),

Philosopher :: new ( "Emma Goldman" ),

Philosopher :: new ( "Michel Foucault" ),

];

let handles: Vec <_> = philosophers.into_iter (). map (| f | {
```

```
thread :: spawn (move || {
f.comer ();
})
}). collect ();
for h in handles {
h.join (). unwrap ();
}
}
```

Wszystko, co zrobiliśmy, to zmieniliśmy pętlę w main() i dodaliśmy drugą! To pierwsza zmiana:

```
let handles: Vec <_> = philosophers.into_iter (). map (| f | {
thread :: spawn (move || {
f.comer ();
})
}). collect ();
```

Mimo to jest to tylko pięć linii, pięć gęstych linii. Przeanalizujmy według części.

```
let handles: Vec < _ > =
```

Wprowadzamy nową zmienną, zwaną uchwytami. Nadaliśmy tę nazwę, ponieważ stworzymy kilka nowych wątków, w wyniku czego do tych wątków dojdą pewne uchwyty (uchwyty, uchwyty), które pozwolą nam kontrolować ich działanie. Musimy jawnie opisać typ ze względu na coś, do czego odniesiemy się później. _ jest symbolem zastępczym dla typu. Mówimy, że „rączki są wektorem czegoś, ale ty, Rust, możesz określić, czym to coś jest".

```
philosophers.into_iter (). map (| f | {
```

Bierzemy naszą listę filozofów i wywołujemy na niej into_iter(). Tworzy to iterator, który przejmuje (należy do) każdego filozofa. Musimy to zrobić, aby przekazać filozofom nasze wątki. Następnie bierzemy ten iterator i wywołujemy na nim mapę, metodę, która przyjmuje zamknięcie jako argument i wywołuje to zamknięcie dla każdego z elementów jednocześnie.

```
thread :: spawn (move || {
f.comer ();
})
```

W tym miejscu występuje współbieżność. Funkcja thread :: spawn przyjmuje zamknięcie jako argument i wykonuje to zamknięcie w nowym wątku. Zamknięcie wymaga dodatkowej adnotacji, move , aby wskazać, że zamknięcie przejmie wartości, które przechwytuje. Głównie zmienna f funkcji mapy . Wewnątrz wątku jedyne, co robimy, to wywołujemy eat ();

```
(); in f .
}). collect ();
```

Na koniec bierzemy wynik wszystkich tych wywołań, aby zmapować i zebrać je. collect() przekonwertuje je na kolekcję pewnego typu, dlatego zwracamy uwagę na zwracany typ: chcemy Vec <T> . Elementy to zwracane wartości wywołań wątku :: spawn , które są uchwytami do tych wątków. Uff! dla h w uchwytach {

h.join (). unwrap ();

}

Na końcu main() przechodzimy przez uchwyty, wywołując na nich join(), co blokuje wykonywanie do momentu zakończenia wykonywania wątku. Zapewnia to zakończenie wykonywania wątku przed zakończeniem programu. Jeśli uruchomisz ten program, zobaczysz, że filozofowie jedzą bez porządku! Mamy wielowątkowe!

Gilles Deleuze is eating.

Gilles Deleuze has finished eating.

Emma Goldman is eating.

Emma Goldman has finished eating.

Michel Foucault is eating.

Judith Butler is eating.

Judith Butler has finished eating.

Karl Marx is eating.

Karl Marx has finished eating.

Michel Foucault has finished eating.

Ale jeśli chodzi o widelce, nie wymodelowaliśmy ich jeszcze w pełni. W tym celu utwórzmy nową strukturę:

use std :: sync :: Mutex;

struct Table {

forks: Vec <Mutex <() >>,

}

Ta tabela zawiera wektor Mutex. Mutex to sposób kontrolowania współbieżności, tylko jeden wątek może uzyskiwać dostęp do treści w danym momencie. Właśnie takiej właściwości potrzebujemy dla naszych posiadaczy. Używamy pustej pary, () , wewnątrz muteksu, ponieważ nie będziemy używać tej wartości, po prostu będziemy ją trzymać. Zmodyfikujmy program, aby używał Mesa:

use std :: thread;

use std :: sync :: {Mutex, Arc};

struct Philosopher {

name: String ,

left: usize,

```
right: usize,
}
impl Filosofo {
fn new (name: & str , left: usize, right: usize) -> Philosopher {
Philosopher {
name: name.to_string (),
left: left,
right: right,
}
}
fn eat (& self , table: & table) {
let _left = table.forks [ self. left] .lock (). unwrap ();
let _right = table.forks [ self. right] .lock (). unwrap ();
println! ( "{} is eating." , self .name);
thread :: sleep_ms ( 1000 );
println! ( "{} has finished eating." , self .name);
}
}
struct Table {
forks: Vec <Mutex < () >>,
}
fn main () {
let table = Arc :: new (Table {forks: vec! [
Mutex :: new (()),
Mutex :: new (()),
Mutex :: new (()),
Mutex :: new (()),
Mutex :: new (()),
]});
let philosophers = vec! [
Philosopher :: new ( "Judith Butler" , 0 , 1 ),
```

```
Philosopher :: new ( "Gilles Deleuze" , 1 , 2 ),
Philosopher :: new ( "Karl Marx" , 2 , 3 ),
Philosopher :: new ( "Emma Goldman" , 3 , 4 ),
Philosopher :: new ( "Michel Foucault" , 0 , 4 ),
];
let handles: Vec <_> = philosophers.into_iter (). map (| f | {
let table = table.clone ();
thread :: spawn (move || {
f.comer (& table);
})
}). collect ();
for h in handles {
h.join (). unwrap ();
}
}
```

Wiele zmian! Jednak dzięki tej iteracji uzyskaliśmy program funkcjonalny. Zobaczmy szczegóły:

```
use std :: sync :: {Mutex, Arc};
```

Użyjemy innej struktury pakietu std :: sync : Arc < T > .Porozmawiamy o tym więcej, gdy go użyjemy.

```
struct Philosopher {
name: String,
left: usize,
right: usize,
}
```

Będziemy musieli dodać jeszcze dwa pola do naszej struktury Philosopher. Każdy filozof będzie miał dwa widelce: ten po lewej i ten po prawej. Do ich wskazania użyjemy typu usize, ponieważ jest to typ, za pomocą którego wektory są indeksowane. Te dwie wartości będą indeksami w posiadaczach, które posiada nasza Tabela.

```
fn new (name: & str, left: usize, right: usize) -> Philosopher {
Philosopher {
name: name.to_string (),
left: left,
right: right,
```

```
}

}
```

Teraz musimy zbudować te wartości w lewo iw prawo, abyśmy mogli dodać je do new() .

```
fn eat (& self, table: & table) {

let _left = table.forks [self.left] .lock (). unwrap ();

let _right = table.forks [self.right] .lock (). unwrap ();

println! ("{} is eating.", self.name);

thread :: sleep_ms (1000);

println! ("{} has finished eating.", self.name);

}
```

Mamy dwie nowe linie, dodaliśmy też argument table . Uzyskaj dostęp do listy posiadaczy tabeli , a następnie użyj self.izquierda i self.derecha, aby uzyskać dostęp do rozwidlenia w określonym indeksie. To daje nam dostęp do Mutex w tym indeksie, gdzie wywołujemy lock() . Jeśli muteks jest obecnie używany przez kogoś innego, zablokujemy go, dopóki nie będzie dostępny. Wywołanie lock() może się nie powieść, a jeśli tak się stanie, chcemy je nagle zakończyć. W tym przypadku błąd, który może wystąpić, polega na tym, że muteks to „zatrucie" („zatrucie"), co dzieje się, gdy wątek wpada w panikę, trzymając blokadę. Ponieważ nie powinno to mieć miejsca, po prostu używamy metody unwrap() . Kolejna dziwna rzecz dotycząca tych linii: nazwaliśmy wyniki _ left i _ right . A co z tym dopiskiem? Cóż, tak naprawdę nie planujemy użyć wartości wewnątrz blokady. Po prostu chcemy go zdobyć. W rezultacie Rust ostrzeże nas, że nigdy nie używamy wartości. Za pomocą indeksu dolnego mówimy Rustowi, czego chcemy, aby nie generował ostrzeżenia. A co ze zwolnieniem blokady? Stanie się tak, gdy _left i _right automatycznie wyjdą poza zakres.

```
let table = Arc :: new (Table {forks: vec! [

Mutex :: new (()),

Mutex :: new (()),

Mutex :: new (()),

Mutex :: new (()),

Mutex :: new (()),

]});
```

Następnie w main() tworzymy nową tabelę i zawijamy ją w Arc <T> . „arc" pochodzi od „atomowej liczby referencji", musimy udostępnić naszą tabelę między wieloma wątkami. W miarę jak będziemy to udostępniać, liczba referencji będzie rosła, a kiedy każdy wątek się skończy, będzie spadać.

```
let philosophers = vec! [

Philosopher :: new ("Judith Butler", 0, 1),

Philosopher :: new ("Gilles Deleuze", 1, 2),

Philosopher :: new ("Karl Marx", 2, 3),
```

```
Philosopher :: new ("Emma Goldman", 3, 4),

Philosopher :: new ("Michel Foucault", 0, 4),

];
```

Musimy przekazać nasze wartości na lewo i prawo konstruktorom naszych Filozofów. Ale jest tu jeszcze jeden szczegół i to bardzo ważny. Jeśli spojrzysz na wzór, jest on spójny do końca, pan Foucault musi mieć 4, 0 jako argumenty, ale zamiast tego ma 0, 4 . Właśnie to zapobiega impasowi: jeden z filozofów jest leworęczny! To jeden ze sposobów rozwiązania problemu i moim zdaniem najprostszy.

```
let handles: Vec <_> = philosophers.into_iter (). map (| f | {

let table = table.clone ();

thread :: spawn (move || {

f.comer (& table);

})

}). collect ();
```

Na koniec wewnątrz naszej pętli map() / collect() wywołujemy mesa.clone() . Metoda clone () w Arc < T > zwiększa liczbę odwołań, a gdy wykracza poza zakres, zmniejsza ją. Zauważysz, że możemy wprowadzić nową zmienną tabelaryczną, która nadpisze starą. Jest to często używane, aby nie trzeba było wymyślać dwóch unikalnych nazw. Dzięki temu nasz program działa! Tylko dwóch filozofów może jeść w danym momencie, w związku z czym będziesz mieć wyjście, które będzie wyglądać tak:

```
Gilles Deleuze is eating.

Emma Goldman is eating.

Emma Goldman has finished eating.

Gilles Deleuze has finished eating.

Judith Butler is eating.

Karl Marx is eating.

Judith Butler has finished eating.

Michel Foucault is eating.

Karl Marx has finished eating.

Michel Foucault has finished eating.
```

Gratulacje! Zaimplementowałeś klasyczny problem współbieżności w Rust.

Rust w innych językach

W naszym trzecim projekcie wybierzemy coś, co zademonstruje jedną z największych zalet Rusta: brak środowiska wykonawczego. W miarę rozwoju organizacje stopniowo wykorzystują wiele języków programowania. Różne języki programowania mają różne mocne i słabe strony, a architektura poliglotyczna pozwala na użycie określonego języka tam, gdzie jego mocne strony mają sens, a inny język jest słaby. Bardzo częstym obszarem, w którym wiele języków programowania jest słabych, jest

wydajność w czasie wykonywania. Często używanie języka, który jest powolny, ale oferuje programistom zwiększoną produktywność, jest cenną równowagą. Aby temu zaradzić, języki te umożliwiają napisanie części systemu w C, a następnie wywołanie tego kodu tak, jakby był napisany w języku wyższego poziomu. Ta funkcja jest nazywana „interfejsem funkcji obcych", powszechnie skracanym do „FFI". Rust obsługuje FFI w obu kierunkach: może z łatwością wywoływać kod C, ale co najważniejsze, można go wywoływać równie łatwo jak C. W połączeniu z brakiem modułu wyrzucania elementów bezużytecznych i niskimi wymaganiami dotyczącymi czasu wykonywania, Rust jest kandydatem do osadzenia w innych językach kiedy potrzebujesz tych dodatkowych ooKmh.

Problem

Jest wiele problemów, które mogliśmy wybrać, ale my wybierzemy przykład, w którym Rust ma wyraźną przewagę nad innymi językami: obliczenia numeryczne i wątki. Wiele języków, na cześć spójności, umieszcza numery na kopcu, a nie na stosie. Szczególnie w językach skupionych na programowaniu obiektowym i używaniu modułu wyrzucania elementów bezużytecznych, alokacja pamięci z kopca jest zachowaniem domyślnym. Czasami optymalizacje mogą umieścić pewne liczby na stosie, ale zamiast polegać na optymalizatorze do wykonania tej pracy, możemy chcieć upewnić się, że zawsze używamy liczb pierwotnych zamiast jakiegoś obiektu. Po drugie, wiele języków ma „globalną blokadę interpretera" (GIL), która ogranicza współbieżność w wielu sytuacjach. Odbywa się to w imię bezpieczeństwa, co jest pozytywnym efektem, ale ogranicza ilość pracy, którą można wykonać jednocześnie, co jest ogromnym minusem. Aby podkreślić te 2 aspekty, stworzymy mały projekt, który w dużym stopniu wykorzystuje te dwa aspekty. Ponieważ celem przykładu jest osadzenie Rusta w innych językach, zamiast samego problemu, użyjemy zabawkowego przykładu: Rozpocznij dziesięć wątków. W każdym wątku liczy się od jednego do pięciu milionów. Po zakończeniu wszystkich wątków wypisz „completed!". Wybrałem pięć milionów na podstawie mojego konkretnego komputera. Oto przykład tego kodu Ruby:

```
threads = []

10 .times do

threads << Thread .new do

count = 0

5_000_000 .times do

count = 1

end

end

end

threads.each {| t | t.join}

puts "completed!"
```

Spróbuj uruchomić ten przykład i wybierz liczbę, która działa przez kilka sekund. W zależności od sprzętu komputera będziesz musiał zwiększyć lub zmniejszyć liczbę. W moim systemie uruchomienie tego programu zajmuje 2156 . Jeśli używam jakiegoś narzędzia do monitorowania procesów, takiego jak top , widzę, że używa ono tylko jednego jądra na moim komputerze. Prezent GIL wykonuje swoją pracę. Chociaż prawdą jest, że jesteś w programie syntetycznym, można sobie wyobrazić wiele

problemów podobnych do tego w prawdziwym świecie. Dla naszych celów wybranie kilku wątków i zajęcie ich reprezentuje rodzaj równoległych i kosztownych obliczeń.

Biblioteka Rust

Napiszmy ten problem w Rust. Najpierw utwórzmy nowy projekt w Cargo:

$ charge new embed

$ cd embed

Ten program jest łatwy do napisania w Rust:

```
use std :: thread;

fn process () {

let handles: Vec <_> = ( 0 .. 10 ) .map (| _ | {

thread :: spawn (|| {

let mut _x = 0 ;

for _ in ( 0 .. 5_000_000 ) {

_x = 1

}

})

}). collect ();

for h in handles {

h.join (). ok (). expect ( "A thread could not be joined!" );

}

}
```

Niektóre z nich powinny wyglądać znajomo z poprzednich przykładów. Rozpoczynamy dziesięć wątków, zbierając je w uchwyty wektorowe. W każdym wątku iterujemy pięć milionów razy, dodając jeden do _x przy każdej iteracji. Dlaczego indeks dolny? Cóż, jeśli go usuniemy, a następnie skompilujemy:

```
$ build charge

Compiling embed v0. 1.0 (file: /// Users / goyox86 / Code / rust / embed)

src / lib.rs: 3 : 1 : 16 : 2 warning: function is never used: ` process` , #

[warn (dead_code)] on by default

src / lib.rs: 3 fn process () {

src / lib.rs: 4 let handles: Vec <_> = ( 0 .. 10 ) .map (| _ | {

src / lib.rs: 5 thread :: spawn (|| {
```

src / lib.rs: 6 let mut x = 0 ;

src / lib.rs: 7 for _ in (0 .. 5 _000_000) {

src / lib.rs: 8 x = 1

…

src / lib.rs: 6 : 17 : 6 : 22 warning: variable `x` is assigned to, but never

used, # [warn (unused_variables)] on by default

src / lib.rs: 6 let mut x = 0 ;

Pierwsze ostrzeżenie dotyczy budowy biblioteki. Gdybyśmy mieli test dla tej funkcji, ostrzeżenie zniknęłoby. Ale na razie nigdy się nie nazywa. Drugi jest związany z x kontra _x . Jako iloczyn faktycznego braku działania z x otrzymujemy ostrzeżenie. W naszym przypadku jest to całkowicie w porządku, ponieważ chcemy marnować cykle procesora. Używając łącznika przedrostka usuwamy ostrzeżenie. Na koniec łączymy każdy z wątków. Na razie jest to jednak biblioteka Rusta i nie ujawnia niczego, co można wywołać z C. Gdybyśmy chcieli połączyć ją z innym językiem, w obecnym stanie, nie zadziałałaby. Musimy tylko wprowadzić kilka drobnych zmian, aby to naprawić. Pierwszą rzeczą jest zmodyfikowanie zasady naszego kodu:

[no_mangle]

pub extern fn process () {

Musimy dodać nowy atrybut no_mangle . Kiedy tworzymy bibliotekę Rust, zmienia ona nazwę funkcji w skompilowanym wyjściu. Przyczyny tego są poza zakresem tego samouczka, ale aby inne języki wiedziały, jak wywołać tę funkcję, musimy uniemożliwić kompilatorowi zmianę nazwy w skompilowanym wyjściu. Ten atrybut wyłącza to zachowanie. Inną zmianą jest pub extern . Pub oznacza, że tę funkcję można wywołać spoza tego modułu, a extern mówi, że można ją wywołać z C. To wszystko! Niewiele zmian. Drugą rzeczą, którą musimy zrobić, to zmienić ustawienie w naszym Cargo.toml . Dodaj to na koniec:

[lib]

name = "embed"

crate-type = ["dylib"]

Linie te informują Rusta, że chcemy skompilować naszą bibliotekę do standardowej biblioteki dynamicznej. Rust kompiluje „rlib”, specyficzny format z Rusta. Teraz zbudujmy projekt:

$ charge build --release

Compiling embed v0. 1.0 (file: /// Users / goyox86 / Code / rust / embed)

Wybraliśmy charge build --release , z którym budujemy projekt. Chcemy, żeby to było jak najszybciej! Dane wyjściowe biblioteki można znaleźć w celu/wydaniu:

$ ls target / release /

zbuduj przykłady deps libembeber.dylib natywny. Ten libembeber.dylib to nasza biblioteka „obiektów współdzielonych”. Możemy używać tej biblioteki jak każdej biblioteki współdzielonych obiektów

napisanej w C! Uwaga: może to być libembeber.so lub libembeber.dll , w zależności od platformy. Teraz, gdy mamy już naszą bibliotekę Rust, użyjmy jej z Ruby.

Ruby

Utwórz plik embeber.rb w naszym projekcie i umieść go w nim:

```
require 'ffi'

Hello module

extend FFI :: Library

ffi_lib 'target / release / libembeber.dylib'

attach_function : process , [] ,: void

end

Hello .process

puts 'completed!'
```

Zanim będziemy mogli go uruchomić, musimy zainstalować klejnot ffi:

```
$ gem install ffi # this may need sudo

Fetching: ffi- 1.9 . 8 .gem ( 100 %)

Tworzenie rozszerzeń natywnych. To może trochę potrwać&hellip;

Successfully installed ffi- 1.9 . 8

Parsing documentation for ffi- 1.9 . 8

Installing ri documentation for ffi- 1.9 . 8

Done installing documentation for ffi after 0 seconds

1 gem installed
```

Finally, let's try running it:

```
$ ruby embeber.rb

completed!

$
```

Wow, to było szybkie! W moim systemie zajmuje to 0,086 sekundy, w przeciwieństwie do dwóch sekund niż w przypadku czystej wersji Ruby. Przeanalizujmy ten kod Ruby:

```
require 'ffi'
```

Najpierw musimy wymagać klejnotu ffi . Pozwala nam na interakcję z biblioteką Rust, taką jak biblioteka C.

```
Hello module

extend FFI :: Library
```

ffi_lib 'target / release / libembeber.dylib'

Moduł Hello służy do dołączania natywnych funkcji biblioteki współdzielonej. Wewnątrz rozszerzamy moduł FFI :: Library, a następnie wywołujemy metodę ffi_lib, aby załadować naszą współdzieloną bibliotekę obiektów. Po prostu przekazujemy ścieżkę, w której przechowywana jest nasza biblioteka, która, jak widzieliśmy wcześniej, to target / release / libembeber.dylib .

attach_function: proces, [],: void

Metoda attach_function jest dostarczana przez klejnot FFI. To właśnie łączy naszą funkcję process() w Rust z metodą w Ruby o tej samej nazwie. Ponieważ proces () nie otrzymuje żadnych argumentów, drugi parametr jest pustą tablicą, a ponieważ nic nie zwraca, przekazujemy : void jako ostatni argument.

Hello .process

To wezwanie do Rusta. Połączenie naszego modułu i wywołania funkcji attach_function skonfigurowało wszystko. Wygląda jak metoda Ruby, ale tak naprawdę jest to kod Rusta!

puts 'completed!'

Wreszcie, jako wymóg naszego projektu, wyświetlamy ukończony! . Otóż to! Jak widzieliśmy, połączenie tych dwóch języków jest naprawdę łatwe i zapewnia nam dużo wydajności. Następnie wypróbujmy Pythona!

Python

Utwórz plik embeber.py w tym katalogu i umieść go w:

```
from ctypes import cdll

lib = cdll.LoadLibrary ( "target / release / libembeber.dylib" )

lib.process ()

print ( "completed!" )
```

Jeszcze łatwiej! Używamy cdll z modułu ctypes. Szybkie wywołanie LoadLibrary później, a potem możemy wywołać process() . W moim systemie zajmuje to 0,017 sekundy. Szybko!

Node.js

Node nie jest językiem, ale obecnie jest dominującą implementacją Javascript po stronie serwera. Aby stworzyć FFI z Node, musimy najpierw zainstalować bibliotekę:

$ npm install ffi

After it is installed, we can use it:

```
var ffi = require ( 'ffi' );

var lib = ffi.Library ( 'target / release / libembeber' , {

'process' : [ 'void' , []]

});

lib.process ();
```

console .log ("completed!");

Wygląda bardziej jak przykład Ruby niż przykład Pythona. Używamy modułu ffi, aby uzyskać dostęp do ffi.Library() , co pozwala nam załadować naszą współdzieloną bibliotekę obiektów. Musimy zwrócić uwagę na zwracany typ i typy argumentów funkcji, które są puste dla zwrotu i pustą tablicę reprezentującą brak argumentów. Stamtąd po prostu wywołujemy funkcję process() i drukujemy wynik. W moim systemie ten przykład zajmuje szybkie 0,092 sekundy.

Rust Cash

Więc nauczyłeś się pisać kod w Rust. Istnieje jednak różnica między pisaniem dowolnego kodu w Rust a pisaniem dobrego kodu w Rust. Ta sekcja składa się ze stosunkowo niezależnych samouczków, które pokazują, jak przenieść Rusta na wyższy poziom. Przedstawione zostaną wspólne wzorce i cechy charakterystyczne biblioteki standardowej. Możesz przeczytać te sekcje w preferowanej kolejności.

Stos i kopiec

Podobnie jak język systemowy, Rust działa na niskim poziomie. Jeśli pochodzisz z języka wysokiego poziomu, istnieją pewne aspekty języków programowania systemowego, których możesz nie znać. Najważniejsze jest funkcjonowanie pamięci, z baterią i kopcem. Jeśli wiesz, jak języki takie jak C używają mapowania ze stosu, ten rozdział będzie przeglądem. Jeśli nie, poznasz tę ogólną koncepcję, ale z podejściem Rustero. Zarządzanie pamięcią

Te dwa terminy odnoszą się do zarządzania pamięcią. Stos i kopiec to abstrakcje, które pomagają określić, kiedy przydzielić i zwolnić pamięć. Oto porównanie na wysokim poziomie: Stos jest bardzo szybki i tam domyślnie przydzielana jest pamięć w Rust. Ale przypisanie jest lokalne dla wywołania funkcji i ma ograniczony rozmiar. Z drugiej strony kopiec jest wolniejszy i jest przypisany przez twój program. Ale ma praktycznie nieograniczony rozmiar i jest dostępny na całym świecie.

Bateria

Porozmawiajmy o tym programie Rust:

fn main () {

let x = 42 ;

}

Ten program ma zmienną (powiązanie zmiennej), x . Pamięć trzeba skądś przydzielać. Rust przydziela z domyślnego stosu, co oznacza, że podstawowe wartości „idą na stos”. Ale co to oznacza? Zobaczmy, kiedy wywoływana jest funkcja, część pamięci jest przydzielana na jej zmienne lokalne i inne dodatkowe informacje. Ta pamięć jest nazywana „rejestrem aktywacji” („ramka stosu”). Na potrzeby tego samouczka zignorujemy dodatkowe informacje i weźmiemy pod uwagę tylko zmienne lokalne, którym przydzielamy pamięć. Tak więc w tym przypadku, gdy wykonywana jest funkcja main(), przypisujemy do naszego rejestru aktywacji 32-bitową liczbę całkowitą. Wszystko to jest obsługiwane automatycznie, jak widać, nie musieliśmy pisać żadnego specjalnego kodu Rusta ani niczego innego. Po zakończeniu funkcji rekord aktywacji zostaje zwolniony lub cofnięty. Dzieje się to automatycznie, nie musieliśmy tu robić nic specjalnego. To wszystko w tym prostym programie. Kluczową rzeczą do zrozumienia jest to, że alokacja pamięci ze stosu jest bardzo, bardzo szybka. Ponieważ znamy z góry wszystkie zmienne lokalne, możemy uzyskać całą pamięć na raz. A ponieważ wyrzucimy to wszystko, możemy się tego pozbyć bardzo szybko. Wadą jest to, że nie możemy przechowywać wartości w kółko,

jeśli są nam potrzebne przez okres dłuższy niż czas życia funkcji. Nie rozmawialiśmy również o tym, co oznacza ta nazwa, „stos”. Aby to zrobić, potrzebujemy nieco bardziej złożonego przykładu:

```
fn foo () {

let y = 5 ;

let z = 100 ;

}

fn main () {

let x = 42 ;

foo ();

}
```

Ten program ma w sumie trzy zmienne: dwie w foo() , jedną w main() . Tak jak poprzednio, przy wywołaniu funkcji main() przypisywana jest pojedyncza liczba całkowita dla jej rekordu aktywacji. Zanim jednak zademonstrujemy, co się dzieje, gdy wywołane zostanie foo(), musimy zwizualizować, co dzieje się w pamięci. Twój system operacyjny przedstawia programowi bardzo prostą wizję: ogromną listę adresów, od 0 do bardzo dużej liczby, która reprezentuje ilość pamięci RAM maszyny. Na przykład, jeśli masz gigabajt pamięci RAM, twoje adresy będą się zmieniać od 0 do 1 073 741 824 , liczba pochodząca od 2 30 , liczby bajtów w gigabajcie. Ta pamięć jest rodzajem gigantycznego układu: adresy zaczynają się od zera i rosną do liczby końcowej. Oto schemat naszego pierwszego rekordu aktywacji:

AddressNameValue

0 x 42

Umieściliśmy x w adresie 0 , o wartości 42

Po wywołaniu foo () przypisywany jest nowy rekord aktywacji:

AddressNameValue

two z 100

one and 5

0 x 42

Ponieważ 0 było zarezerwowane dla pierwszej ramki, 1 i 2 są używane do zapisu aktywacji foo() . Stos rośnie w górę, gdy wywołujemy więcej funkcji. Należy tutaj zwrócić uwagę na kilka ważnych rzeczy. Liczby 0, 1 i 2 istnieją tylko w celach ilustracyjnych i nie mają żadnego związku z liczbami, których faktycznie używałby komputer. W szczególności serie adresów są oddzielone liczbą bajtów, a odstęp ten może nawet przekraczać rozmiar przechowywanej wartości. Po zakończeniu foo () Twój rekord aktywacji zostaje zwolniony:

AddressNameValue

0 x 42

A potem po zakończeniu funkcji main() ta ostatnia wartość znika. Łatwy! Nazywa się to stosem, ponieważ działa jak stos talerzy: pierwszy talerz, który umieścisz, będzie ostatnim, który usuniesz. Stosy

są czasami nazywane kolejkami „ostatnie weszło, pierwsze wyszło”, z tych powodów ostatnia wartość, którą umieścisz na stosie, będzie pierwszą, którą z niego uzyskasz. Spróbujmy na przykładzie trzech poziomów:

```
fn bar () {

let i = 6 ;

}

fn foo () {

let a = 5 ;

let b = 100 ;

let c = 1 ;

Pub();

}

fn main () {

let x = 42 ;

foo ();

}
```

Cóż, w pierwszej kolejności wywołujemy main() :

AddressNameValue

0 x 42

Następnie main() wywołuje foo():

AddressNameValue

3 c one

two b 100

one to 5

0 x 42

Następnie foo () wywołuje bar ():

AddressNameValue

4 i 6

3 c one

two b 100

one to 5

0 x 42

Uff! Nasz stos rośnie.

Po zakończeniu paska () jego rekord aktywacji jest zwalniany, pozostawiając tylko foo() i main() :

AddressNameValue

3 c one

two b 100

one to 5

0 x 42

Następnie foo () kończy się, pozostawiając tylko main ()

AddressNameValue

0 x 42

W takim razie skończyliśmy. to jest zrozumiałe? To jest jak układanie talerzy w stos: dodajesz na górę i zdejmujesz z niej.

Kopiec

Teraz wszystko to działa dobrze, ale nie wszystko działa w ten sposób. Czasami trzeba przekazywać pamięć między różnymi funkcjami lub utrzymywać pamięć przy życiu przez dłuższy czas niż wykonanie funkcji. Do tego używamy kopca. W Rust możesz przydzielić pamięć z kopca za pomocą typu Box <T> . Oto przykład:

fn main () {

let x = Box :: new (5);

let y = 42 ;

}

Oto, co się dzieje, gdy wywoływany jest main():

AddressNameValue

one and 42

0 x ??????

Przydzielamy miejsce na dwie zmienne na stosie. y wynosi 42 , jak wiemy do tej pory, ale co z x? Cóż, x to Box <i32> , a pola przydzielają pamięć z kopca. Wartość omawianego pudełka to struktura, która ma wskaźnik do „kopca”. Kiedy rozpoczyna się wykonywanie funkcji i wywoływane jest Box :: new (), przydziela ona trochę pamięci dla kopca i umieszcza tam 5. Pamięć wygląda teraz tak:

AddressNameValue

2 30 5

... ...

one and 42

0 x 2 30

Mamy 2 30 na naszym hipotetycznym komputerze z 1 GB pamięci RAM. A ponieważ nasz stos rośnie od zera, najłatwiejszym sposobem przydzielenia pamięci jest drugi koniec. Tak więc nasza pierwsza wartość znajduje się na najwyższym miejscu w pamięci. A wartość struktury w x ma płaski wskaźnik (wskaźnik surowy) do miejsca, które przypisaliśmy na kopcu, a następnie wartość x EN 2 30 , adres pamięci, o który prosiliśmy. Nie rozmawialiśmy zbyt wiele o tym, co tak naprawdę oznacza alokacja i zwalnianie pamięci w tych kontekstach. Zagłębianie się w szczegóły wykracza poza zakres tego samouczka. Należy zauważyć, że kopiec nie jest zwykłym stosem rosnącym z przeciwnej strony. W dalszej części książki będziemy mieli tego przykład, ale ponieważ kopiec można przydzielać i zwalniać w dowolnej kolejności, może się on kończyć „lukami". Oto diagram rozkładu pamięci programu, który działa od jakiegoś czasu:

AddressName Value

2 30 5

(2 30) - 1

(2 30) - 2

(2 30) - 3 42

... ...

3 and (2 30) - 3

two and 42

one and 42

0 x 2 30

W tym przypadku przypisaliśmy cztery rzeczy na kopcu, ale zwolniliśmy dwie z nich. Istnieje luka między 2 30 a (2 30) - 3, która nie jest obecnie wykorzystywana. Konkretne szczegóły dotyczące tego, jak i dlaczego tak się dzieje, zależą od strategii zastosowanej do zarządzania kopcem. Różne programy mogą używać różnych „alokatorów pamięci", czyli bibliotek obsługujących alokację pamięci. Programy w Rust używają do tego celu jemalloc. W każdym razie, wracając do naszego przykładu. Ponieważ ta pamięć jest na kopcu, może pozostać żywa dłużej niż funkcja, która tworzy pudełko. Jednak w tym przypadku tak się nie dzieje. poruszając się po zakończeniu funkcji, musimy zwolnić rekord aktywacji z main() . Box <T> ma jednak asa w rękawie: Drop . Implementacja Drop for Box zwalnia pamięć, która została przydzielona podczas tworzenia skrzynki. Świetnie! Więc kiedy x opuszcza (opuszcza kontekst), najpierw zwalnia przydzieloną pamięć z kopca:

AddressNameValue

one and 42

0 x ??????

Możemy sprawić, że pamięć pozostanie żywa dłużej, przenosząc własność, czasami nazywaną „wyjmowaniem z pudełka". Bardziej złożone przykłady zostaną omówione później. ↩ Następnie zapis aktywacji znika, uwalniając całą naszą pamięć.

Argumenty i pożyczki

Zrobiliśmy kilka podstawowych przykładów ze stosem i kopcem, ale co z argumentami funkcji i pożyczaniem? Oto mały program Rust:

```
fn foo (i: & i32 ) {

let z = 42 ;

}

fn main () {

let x = 5 ;

let y = & x;

foo (y);

}
```

Gdy wpiszemy main() , pamięć wygląda tak:

Address Name Value

one and 0

0 x 5

x to prosta liczba 5 , a y to odniesienie do x . Tak więc wartość y to adres pamięci, w której znajduje się x, czyli w tym przypadku 0 . Co się stanie, gdy wywołamy foo () przekazując do y jako argument?

Address Name Value

3 z 42

two i 0

one and 0

0 x 5

Rekordy aktywacji służą nie tylko do zmiennych lokalnych, ale także do argumentów. W tym przypadku musimy mieć zarówno i , nasz argument, jak i z naszą zmienną lokalną. i jest kopią argumentu, a . Ponieważ wartość y wynosi 0, to jest to wartość i . Jest to jeden z powodów, dla których pożyczanie zmiennej nie zwalnia pamięci: wartość referencyjna jest tylko wskaźnikiem do adresu pamięci. Gdybyśmy pozbyli się ukrytej pamięci, sprawy nie potoczyłyby się całkiem dobrze.

Złożony przykład

Dobra, przejdźmy przez ten złożony program krok po kroku:

```
fn foo (x: & i32 ) {

let y = 10 ;

let z = & y;

baz (z);

bar (x, z);
```

```
}
fn bar (a: & i32 , b: & i32 ) {
let c = 5 ;
let d = Box :: new ( 5 );
let e = & d;
baz (e);
}
fn baz (f: & i32 ) {
let g = 100 ;
}
fn main () {
let h = 3 ;
let i = Box :: new ( 20 );
let j = & h;
foo (j);
}
```

Najpierw wywołujemy funkcję main():

AddressName Value

2 30 twenty

... ...

two j 0

one i 2 30

0 h 3

Przydzielamy pamięć dla j , i , oraz h . i jest na kopcu, dlatego jego wartość na niego wskazuje. Następnie na końcu main() wywoływane jest foo():

AddressName Value

2 30 twenty

... ...

5 z 4

4 and 10

3 x 0

two j 0

one i 2 30

0 h 3

Przestrzeń jest przydzielana dla x , y i z . Argument x ma taką samą wartość jak j , ponieważ właśnie to podaliśmy funkcji. Jest to wskaźnik do adresu 0 , ponieważ j wskazuje na h . Następnie foo() wywołuje baz() , przekazując go z :

AddressName Value

2 30 twenty

... ...

7 g 100

6 F 4

5 z 4

4 and 10

3 x 0

two j 0

one i 2 30

0 h 3

Przydzieliliśmy pamięć dla f i g . baz() jest bardzo krótki, więc kiedy się kończy, pozbywamy się twojego rekordu aktywacji:

AddressName Value

2 30 twenty

... ...

5 z 4

4 and 10

3 x 0

two j 0

one i 2 30

0 h 3

Następnie foo () wywołuje bar () z x i z :

AddressName Value

2 30 twenty

(2 30) - 1 5

... ...

10 and 9

9 d (2 30) - 1

8 c 5

7 b 4

6 to 0

5 z 4

4 and 10

3 x 0

two j 0

one i 2 30

0 h 3

W końcu przypisujemy kopcowi inną wartość, więc musimy odjąć jeden od 2 30 . Łatwiej to napisać niż 1 073 741 823 . W każdym razie ustawiamy zmienne jak zwykle. Na końcu bar () wywołuje to baz () :

AddressName Value

2 30 twenty

(2 30) - 1 5

... ...

12 g 100

eleven F 9

10 and 9

9 d (2 30) - 1

8 c 5

7 b 4

6 to 0

5 z 4

4 and 10

3 x 0

two j 0

one i 2 30

0 h 3

Dzięki temu jesteśmy w naszym najgłębszym punkcie! Wow! Gratuluję, że śledziłeś to wszystko i doszedłeś tak daleko. Wtedy baz() się kończy, pozbywamy się f i g :

AddressName Value

2 30 twenty

(2 30) - 1 5

... ...

...

10 and 9

9 d (2 30) - 1

8 c 5

7 b 4

6 to 0

5 z 4

4 and 10

3 x 0

two j 0

one i 2 30

0 h 3

Następnie wracamy z bar () . d w tym przypadku to Box <T> , więc uwalnia również to, na co wskazuje: (2 30) - 1.

Address Name Value

2 30 twenty

... ...

5 z 4

4 and 10

3 x 0

two j 0

one i 2 30

0 h 3

Następnie foo() zwraca:

Address Name Value

2 30 twenty

… … …

two j 0

one i 2 30

0 h 3

W końcu main() powraca, co czyści resztę. Kiedy i zostanie zwolnione (przez Drop), wyczyści również resztę na kopcu.

Co robią inne języki?

Większość języków z domyślnie mapą Garbage Collectora z kopca. Oznacza to, że wszystkie wartości znajdują się w ramkach (w ramkach). Istnieje wiele powodów, dla których odbywa się to w ten sposób, ale wykraczają one poza zakres tego samouczka. Ponadto istnieją pewne optymalizacje, które sprawiają, że nie jest to w 100% prawdziwe przez cały czas. Zamiast polegać na stosie i upuszczaniu w celu wyczyszczenia pamięci, śmieciarz jest odpowiedzialny za zarządzanie kopcem. Którego użyć Jeśli stos jest szybszy i łatwiejszy w użyciu, po co nam kopiec? Świetnym powodem jest to, że alokacja ze stosu oznacza, że masz tylko semantykę LIFO do odzyskania pamięci. Alokacja z kopca jest ściśle bardziej ogólna, umożliwiając pobieranie i zwracanie pamięci do puli w dowolnej kolejności, ale kosztem złożoności. Generalnie powinieneś preferować alokację ze stosu, dlatego Rust przypisuje ze stosu domyślnego. Model stosu LIFO jest prostszy na podstawowym poziomie. Ma to dwa główne skutki: wydajność środowiska wykonawczego i wpływ semantyczny.

Efektywność czasu wykonania.

Zarządzanie pamięcią dla stosu jest trywialne: maszyna po prostu zwiększa pojedynczą wartość, tak zwany „wskaźnik stosu”. Zarządzanie pamięcią dla kopca nie jest: Pamięć przydzielona z kopca jest zwalniana w dowolnych punktach, a każdy blok pamięci przydzielonej z kopca może mieć dowolny rozmiar, menedżer pamięci generalnie musi pracować znacznie ciężej, aby zidentyfikować pamięć, którą można ponownie wykorzystać. Jeśli chcesz zagłębić się w ten temat bardziej szczegółowo, ten artykuł jest bardzo dobrym wprowadzeniem.

Wpływ semantyczny

Alokacja stosu wpływa na sam język Rust, a tym samym na model mentalny programisty. Semantyka LIFO jest tym, co napędza sposób, w jaki język Rust obsługuje automatyczne zarządzanie pamięcią. Nawet cofnięcie alokacji pudła przydzielonego do sterty, którego właściciel jest unikalny, może być sterowane semantyką LIFO opartą na stosie, jak omówiono w tym rozdziale. Elastyczność (tj. wyrazistość) semantyki innej niż LIFO oznacza, że generalnie kompilator nie może automatycznie wywnioskować w czasie kompilacji, gdzie pamięć powinna zostać zwolniona; musi polegać na protokołach dynamicznych, potencjalnie spoza samego języka, w celu kierowania cofnięciem alokacji (zliczanie referencji, używane przez Rc i Arc, jest tego przykładem). Mapowanie ze stosu wpływa na Rust jako język, a wraz z nim na model mentalny dewelopera. Semantyka LIFO decyduje o tym, jak język Rust obsługuje automatyczną obsługę pamięci. Nawet zwolnienie przypisanego pudełka z kopca z jednym właścicielem może być obsługiwane przez semantykę LIFO, jak omówiono w tym rozdziale. Elastyczność (np. wyrazistość) semantyki innej niż LIFO oznacza, że generalnie kompilator nie może automatycznie iw czasie kompilacji wywnioskować, gdzie pamięć powinna zostać zwolniona; musi polegać na protokołach dynamicznych, potencjalnie zewnętrznych w stosunku do języka, w celu zwolnienia pamięci (przykładem jest zliczanie referencji, takie jak używane w Rc <T> i Arc <T>).

Doprowadzona do skrajności większa ekspresyjna moc alokacji z kopca odbywa się kosztem albo znacznego wsparcia czasu wykonywania (np. weryfikacja nie zapewniana przez kompilator Rusta).

Testy

Testowanie programów może być skutecznym sposobem wykazania obecności błędów, ale wykazanie ich braku jest beznadziejnie niewystarczające. Edsger W. Dijkstra, „Skromny programista" (1972) Porozmawiamy o testowaniu kodu Rust. To, o czym nie będziemy mówić, to właściwy sposób testowania kodu Rusta. Istnieje wiele szkół myślenia dotyczących prawidłowego i nieprawidłowego sposobu pisania testów. Wszystkie te podejścia wykorzystują te same podstawowe narzędzia, w tej sekcji pokażemy składnię, aby z nich skorzystać.

Atrybut testowy

Zasadniczo test w Rust jest funkcją opatrzoną adnotacją o atrybucie test . Zamierzamy stworzyć nowy projekt o nazwie adder z Cargo:

$ new adder charge

$ cd adder

Cargo automatycznie wygeneruje prosty test podczas tworzenia nowego projektu. Oto zawartość src / lib.rs:

#[test]

fn it_works () {

}

Zwróć uwagę na # [test] . Ten atrybut wskazuje, że jest to funkcja testowa. Obecnie nie ma ciała. Ale to wystarczy, żeby się stało! Możemy przeprowadzić testy z ładunkiem testowym:

$ test charge

Compiling adder v0. 1.0 (file: /// Users / goyox86 / Code / rust / adder)

Running target / debug / adder-ba17f4f6708ca3b9

running 1 test

test it_works ... ok

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

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

Charge I kompiluje i uruchamia nasze testy. Istnieją tutaj dwa zestawy danych wyjściowych: jeden dla testów, które piszemy, i jeden dla testów dokumentacji. Porozmawiamy o nich później. Na razie zobaczmy tę linię:

test it_works ... ok

Zwróć uwagę na plik it_works . Pochodzi od nazwy naszej funkcji:

fn it_works () {

#}

Otrzymujemy również linię podsumowującą:

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

Dlaczego więc nasze puste testy przechodzą pomyślnie? Każdy test, który nie panikuje! przechodzi i wszelkie testy, które panikują! awaria. Oblewamy nasz test:

#[test]

fn it_works () {

assert! (false);

}

assert! jest to makro dostarczone przez Rust, które przyjmuje argument: jeśli argument jest prawdziwy, nic się nie dzieje. Jeśli argument jest fałszywy, potwierdź! temu panika! . Ponownie uruchommy nasze testy:

$ test charge

Compiling adder v0. 1.0 (file: /// Users / goyox86 / Code / rust / adder)

Running target / debug / adder-ba17f4f6708ca3b9

running 1 test

test it_works ... FAILED

failures:

---- it_works stdout ----

thread 'it_works' panicked at 'assertion failed: false' , src / lib.rs: 3

failures:

it_works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

thread '<main>' panicked at 'Some tests failed' ,

/Users/rustbuild/src/rust-buildbot/slave/stable-dist-rustcmac/

build/src/libtest/lib.rs: 259

Rust tells us that our test has failed:

test it_works ... FAILED

And it is reflected in the summary line:

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

Otrzymujemy również niezerową wartość zwracaną:

$ echo $?

101

Jest to bardzo przydatne do zintegrowania testu ładunku z innymi narzędziami. Możemy odwrócić niepowodzenie testu za pomocą innego atrybutu: should_panic :

#[test]

[should_panic]

fn it_works () {

assert! (false);

}

Te testy zakończą się sukcesem, jeśli wpadniemy w panikę! i zakończy się niepowodzeniem, jeśli zostanie ukończony. Spróbujmy:

$ test charge

Compiling adder v0. 1.0 (file: /// Users / goyox86 / Code / rust / adder)

Running target / debug / adder-ba17f4f6708ca3b9

running 1 test

test it_works ... ok

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

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

Rust udostępnia kolejne makro, assert_eq! , która porównuje dwa argumenty w celu sprawdzenia równości:

#[test]

[should_panic]

fn it_works () {

assert_eq! ("Hello" , "world");

}

Czy ten test przechodzi, czy nie? Ze względu na obecność atrybutu should_panic przekazuje:

$ test charge

Compiling adder v0. 1.0 (file: /// Users / goyox86 / Code / rust / adder)

Running target / debug / adder-ba17f4f6708ca3b9

running 1 test

test it_works ... ok

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

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured

Testy Should_Panic mogą być kruche, trudno jest zagwarantować, że test nie zawiedzie z nieoczekiwanego powodu. Aby w tym pomóc, do atrybutu should_panic można dodać opcjonalny oczekiwany parametr . Test zapewni, że komunikat o błędzie zawiera podany komunikat. Bezpieczniejszą wersją testu byłoby:

```
#[test]

# [should_panic (expected = "assertion failed")]

fn it_works () {

assert_eq! ( "Hello" , "world" );

}
```

To było tyle jeśli chodzi o podstawy! Napiszmy „prawdziwy” test:

```
pub fn sum_two (a: i32) -> i32 {

a + 2

}

#[test]

fn it_works () {

assert_eq! (4, sum_two (2));

}
```

Jest to bardzo powszechne użycie assert_eq! : wywołaj jakąś funkcję z pewnymi znanymi argumentami i porównaj wynik tego wywołania z oczekiwanym wynikiem.

Moduł testów

Jest sposób, w jaki nasz przykład nie jest idiomatyczny: brakuje w nim modułu testów. Idiomatyczny sposób zapisania naszego przykładu wygląda następująco:

```
pub fn sum_two (a: i32) -> i32 {

a + 2

}

# [cfg (test)]

mod tests {

use super :: sum_two;
```

```
#[test]

fn it_works () {

assert_eq! (4, sum_two (2));

}

}
```

Jest tu kilka zmian. Pierwszym z nich jest włączenie testów modów z atrybutem cfg. Moduł pozwala nam pogrupować wszystkie nasze testy, a także pozwala nam zdefiniować funkcje wsparcia w razie potrzeby, wszystko to nie jest częścią naszej skrzynki. Atrybut cfg kompiluje nasz kod testowy tylko wtedy, gdy próbowaliśmy uruchomić testy. Może to zaoszczędzić czas kompilacji, zapewnia również, że nasze testy są całkowicie wykluczone z normalnej kompilacji. Druga zmiana to instrukcja użycia. Ponieważ jesteśmy w wewnętrznym module, musimy udostępnić nasz test w naszym obecnym zakresie. Może to być irytujące, jeśli masz duży moduł i dlatego powszechne jest korzystanie z funkcji globu. Zmieńmy nasz src / lib.rs, aby z niego skorzystać:

```
pub fn sum_two (a: i32) -> i32 {

a + 2

}

# [cfg (test)]

mod tests {

use super :: *;

#[test]

fn it_works () {

assert_eq! (4, sum_two (2));

}

}
```

Zwróć uwagę, że linia jest inna. Teraz przeprowadzamy nasze testy:

```
$ test charge

Compiling adder v0. 1.0 (file: /// Users / goyox86 / Code / rust / adder)

Running target / debug / adder-ba17f4f6708ca3b9

running 1 test

test tests :: it_works ... ok

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

Doc-tests adder

running 0 tests

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

Działa!

Obecna konwencja polega na używaniu modułu testów do przechowywania testów „w stylu jednostkowym”. Wszystko, co tylko testuje mały fragment funkcjonalności, trafia tutaj. Ale co z testowaniem „w stylu integracji”? Dla nich mamy katalog testów .

Katalog testów

Aby napisać test integracyjny, utwórzmy katalog testów i umieśćmy w nim plik testy / lib.rs o następującej zawartości:

```
extern crate adder;

#[test]

fn it_works () {

assert_eq! (4, adder :: sum_two (2));

}
```

Wygląda podobnie do naszych poprzednich testów, ale nieco inaczej. Teraz na początku mamy zewnętrzny dodatek do skrzynek. Dzieje się tak dlatego, że testy w katalogu to osobna skrzynia, więc musimy zaimportować naszą bibliotekę. Właśnie dlatego testy są doskonałym miejscem do pisania testów integracyjnych: te testy korzystają z biblioteki tak, jak zrobiłby to każdy inny konsument. Uruchommy je:

```
$ test charge

Compiling adder v0. 1.0 (file: /// Users / goyox86 / Code / rust / adder)

Running target / debug / lib -f 71036151ee98b04

running 1 test

test it_works ... ok

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

Running target / debug / adder-ba17f4f6708ca3b9

running 1 test

test tests :: it_works ... ok

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

Doc-tests adder

running 0 tests

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

Mamy teraz trzy sekcje: Uruchomiono również nasze poprzednie testy wraz z nowym testem integracyjnym. To było na tyle, jeśli chodzi o katalog testów. Moduł testów nie jest tutaj potrzebny, ponieważ cały moduł jest przeznaczony do testów. Na koniec przyjrzyjmy się trzeciej sekcji: testom dokumentacji.

Testy dokumentacji

Nie ma nic lepszego niż dokumentacja z przykładami. Nie ma nic gorszego niż przykłady, które nie działają, ponieważ kod zmienił się od czasu napisania dokumentacji. W związku z tym Rust obsługuje automatyczne wykonywanie przykładów obecnych w Twojej dokumentacji. Oto dopracowany src / lib.rs z przykładami:

//! The `adder` crate provides functions that add numbers to other numbers.

//!

//! # Examples

//!

//!

//! assert_eq! (4, adder :: add_two (2)); //! `` ''

/// This function adds two to its argument. /// /// # Examples /// /// /// use adder ::

add_two; /// /// assert_eq! (4, add_two (2)); /// pub fn add_two (a: i32) -> i32 {a + 2}

[cfg (test)]

mod tests {use super :: *;

#[test]

fn it_works () {

assert_eq! (4, add_two (2));

}

}

Zwróć uwagę na dokumentację na poziomie modułu z `//!` I dokumentację na poziomie funkcji z `///`. Dokumentacja Rust obsługuje Markdown w komentarzach i wysokich (\ `\` \ `) rozdzielających bloki kodu. Tradycyjnie dołącza się sekcję `# Examples`, dokładnie tak jak ta, po której następują przykłady. Ponownie uruchommy testy:

bash

$ test charge

Compiling adder v0.1.0 (file: /// Users / goyox86 / Code / rust / adder)

Running target / debug / lib-f71036151ee98b04

running 1 test

test it_works ... ok

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

Running target / debug / adder-ba17f4f6708ca3b9

running 1 test

test tests :: it_works ... ok

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

Doc-tests adder

running 2 tests

test _0 ... ok

test sum_two_0 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured

Teraz mamy uruchomione wszystkie trzy rodzaje testów! Zanotuj nazwy testów dokumentacji: _0 jest generowane dla testu modułu, a sum_two_0 dla testu funkcji. Liczby te będą automatycznie zwiększane o nazwy takie jak sum_two_1 w miarę dodawania kolejnych przykładów.

Kompilacja warunkowa

Rust ma specjalny atrybut # [cfg] , który umożliwia kompilację kodu w oparciu o opcję udostępnioną kompilatorowi. Ma dwie formy:

[cfg (foo)]

fn foo () {}

[cfg (bar = "baz")]

fn bar () {}

It also has some helpers:

[cfg (any (unix, windows))]

fn foo () {}

[cfg (all (unix, target_pointer_width = "32"))]

fn bar () {}

[cfg (not (foo))]

fn not_foo () {}

Które można dowolnie zagnieżdżać:

[cfg (any (nie (unix), all (target_os = "macos", target_arch = "powerpc")))]

fn foo () {}

Aby aktywować lub dezaktywować te przełączniki, jeśli korzystasz z Cargo, są one skonfigurowane w sekcji [features] Twojego Cargo.toml

[features]

No features by default

default = []

The “secure-password” feature depends on the bcrypt package.

secure-password = ["bcrypt"]

Kiedy to robimy, Cargo przekazuje opcję do rustc :

--cfg feature = "$ {feature_name}"

Suma tych opcji cfg określi, które z nich są aktywowane, a co za tym idzie, który kod zostanie skompilowany. Weźmy ten kod:

[cfg (feature = "foo")]

mod foo {

}

Jeśli skompilujemy go z cargo build --features "foo" , Cargo wyśle opcję --cfg feature = "foo" do rustc , a wyjście będzie miało mod foo . Jeśli skompilujemy z normalną opłatą za kompilację, nie będzie dostępna żadna dodatkowa opcja iz tego powodu żaden moduł foo nie będzie istniał.

cfg_attr

Możesz także skonfigurować inny atrybut oparty na zmiennej cfg za pomocą cfg_attr :

[cfg_attr (a, b)]

fn foo () {}

Będzie to to samo co # [b], jeśli a jest skonfigurowane przez atrybut cfg i nic więcej.

cfg!

Rozszerzenie składni pozwala również na użycie tego typu opcji w dowolnym innym miejscu kodu:

if cfg! (target_os = "macos") || cfg! (target_os = "ios") {

println! ("Think Different!");

}

Zostaną one zastąpione przez prawda lub fałsz w czasie kompilacji, w zależności od opcji konfiguracji.

Dokumentacja

Dokumentacja jest ważną częścią każdego projektu oprogramowania i obywatelem pierwszej klasy w Rust. Porozmawiajmy o narzędziach, które Rust zapewnia do dokumentowania twoich projektów.

O rustdocu

Dystrybucja Rust zawiera narzędzie rustdoc odpowiedzialne za generowanie dokumentacji. rustdoc jest również używany przez Cargo poprzez cargo doc . Dokumentację można wygenerować na dwa sposoby: z kodu źródłowego lub z plików Markdown.

Dokumentowanie kodu źródłowego

Głównym sposobem dokumentowania projektu Rust jest adnotacja kodu źródłowego. W tym celu można skorzystać z komentarzy do dokumentacji:

/// Build a new `Rc`.

///

/// # Examples

///

///

/// use std :: rc :: Rc; /// /// let five = Rc :: new (5); /// `` `pub fn new

(value: T) -> Rc {// the implementation goes here}

Powyższy kod generuje dokumentację, która wygląda następująco: [this] [rc-new] (angielski). Pominąłem implementację, zamiast tego dodałem zwykły komentarz. To pierwsza rzecz, na którą należy zwrócić uwagę w tej adnotacji:

użyj `///` zamiast `//`. Potrójny ukośnik wskazuje, że jest to komentarz do dokumentacji. Komentarze do dokumentacji są zapisywane w formacie Markdown. Rust prowadzi rejestr takich komentarzy, zapis, którego używa podczas generowania dokumentacji. Jest to ważne podczas dokumentowania rzeczy takich jak wyliczenia:

`` 'rust

/// The type `Option`. See [module level documentation] (../) for more

information.

enum Option <T> {

/// No value

None

/// Some `T` value

Some (T),

}

Powyższe działa, ale to nie:

/// The type `Option`. See [module level documentation] (../) for more

information.

enum Option {

None, /// No value

Some (T), /// Some value `T`

}

You will get an error:

hello.rs:4:1: 4: 2 error: expected ident, found `}`

hello.rs:4}

Pisanie komentarzy do dokumentacji

Tak czy inaczej, omówmy szczegółowo każdą część tego komentarza:

/// Build a new `Rc <T>`.

fn foo () {}

Pierwsza linia komentarza do dokumentacji powinna zawierać krótkie podsumowanie jej funkcjonalności. Sentencja. Tylko podstawy. Wysoki poziom.

///

/// Other details about the construction of `Rc <T>` s, perhaps describing semantics

/// complicated, maybe additional options, anything extra.

///

fn foo () {}

Nasz oryginalny przykład miał tylko jedną linię podsumowującą, ale gdybyśmy mieli więcej do powiedzenia, moglibyśmy dodać więcej wyjaśnień w nowej książce. Sekcje specjalne

/// # Examples

fn foo () {}

Poniżej znajdują się sekcje specjalne. Są one oznaczone nagłówkiem # . Powszechnie stosowane są trzy typy nagłówków. Nie jest to na razie specjalna składnia, tylko konwencja.

/// # Panics

fn foo () {}

Złe i nieodwracalne użycie funkcji (np. błędy programistyczne) w Rust są zwykle sygnalizowane paniką, która przynajmniej zabija bieżący wątek. Jeśli twoja rola ma nietrywialny kontrakt, taki jak ten, który jest panikowany/egzekwowany, udokumentowanie tego jest bardzo ważne.

/// # Failures

fn foo () {}

Jeśli twoja funkcja lub metoda zwraca Result <T, E> , to dobrze jest opisać warunki, w których Err (E) zwraca. Jest to nieco mniej ważne niż Panics , ponieważ jest zakodowane w systemie typów, ale wciąż jest coś do zrobienia.

/// # Safety

fn foo () {}

Jeśli twoja funkcja jest niebezpieczna (niepewna), powinieneś wyjaśnić, jakie są niezmienniki, które musi obsługiwać wywołujący.

/// # Examples

///

///

/// use std :: rc :: Rc; /// /// let five = Rc :: new (5); /// `` `

fn foo () {}

Po trzecie, „Przykłady”, podaj jeden lub więcej przykładów użycia Twojej funkcji lub metody, a Twoi użytkownicy Cię pokochają. Te przykłady znajdują się w adnotacjach bloków kodu, o których powiemy za chwilę, mogą mieć więcej niż jedną sekcję:

`` 'rust

/// # Examples

///

/// Simple `& str` patterns:

///

///

/// let v: Vec <& str> = "Mary had a little lamb" .split ('') .collect (); ///

assert_eq! (v, vec! ["Mary", "had", "a", "little lamb"]); /// /// /// More complex patterns with lambdas: /// /// /// let v: Vec <& str> = "abc1def2ghi" .split (| c: char | c.is_numeric ()). collect (); /// assert_eq! (v, vec! ["abc", "def", "ghi"]); /// `` `

fn foo () {}

Omówmy szczegóły tych bloków kodu.

Adnotacje bloków kodu

Aby napisać kod Rusta w komentarzu, użyj potrójnego basu:

`` 'rust

///

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

fn foo () {}

Jeśli chcesz kod inny niż Rust, możesz dodać adnotację:

`` 'rust

/// `` `c

/// printf ("Hello, world \ n");

///

fn foo () {}

Składnia tej sekcji zostanie podświetlona zgodnie z wyświetlanym językiem. Jeśli wyświetlasz tylko zwykły tekst, użyj `text`. Tutaj ważne jest, aby wybrać odpowiednią adnotację, ponieważ `rustdoc` używa jej w ciekawy sposób: Można jej użyć do przetestowania twoich przykładów, aby z czasem nie stały się przestarzałe. Jeśli masz jakiś kod C, ale `rustdoc` myśli, że to Rust, to dlatego, że zapomniałeś adnotacji, `rustdoc` narzekał podczas próby wygenerowania dokumentacji.

Dokumentacja jako dowód

Omówmy naszą przykładową dokumentację:

`` 'rust

///

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

fn foo () {}

Zauważysz, że nie potrzebujesz `fn main()` ani czegoś innego. `rustdoc` automatycznie doda main() wokół Twojego kodu we właściwym miejscu. Na przykład:

`` 'rust

///

/// use std :: rc :: Rc; /// /// let five = Rc :: new (5); /// `` `

fn foo () {}

Stanie się testem:

`` 'rust

fn main () {

use std :: rc :: Rc;

let five = Rc :: new (5);

}

Oto kompletny algorytm, którego rustdoc używa do post-processingu przykładów:

1. Wszelkie resztki #! Atrybuty [Foo] pozostają nienaruszone jako atrybut skrzyni.

2. Wstawione są niektóre typowe atrybuty, w tym unused_variables , unused_assignments , unused_mut , unused_attributes i dead_code . Małe przykłady czasami powodują te kłaczki.

3. Jeśli przykład nie zawiera extern crate , to extern crate <micrate>; jest wstawiony.

4. Wreszcie, jeśli przykład nie zawiera fn main , tekst jest zawinięty w fn main () {twój_kod} Czasami to nie wystarcza. Na przykład wszystkie te przykłady kodu z ///, o których mówiliśmy? Zwykły tekst:

/// Some documentation.

fn foo () {}

Wygląda inaczej w dniu wyjazdu:

/// Some documentation.

fn foo () {}

Tak, to prawda: możesz dodać wiersze zaczynające się od # , które zostaną usunięte z danych wyjściowych, ale zostaną użyte w kompilacji twojego kodu. Możesz to wykorzystać na swoją korzyść. W tym przypadku komentarze do dokumentacji muszą odnosić się do jakiejś funkcji, więc jeśli chcę pokazać tylko jeden komentarz do dokumentacji, muszę dodać małą definicję funkcji poniżej. Jednocześnie jest tam tylko po to, aby zadowolić kompilator, więc ukrycie go sprawia, że przykład jest

czystszy. Możesz użyć tej techniki do szczegółowego wyjaśnienia dłuższych przykładów, zachowując jednocześnie możliwość testowania dokumentacji. Na przykład ten kod:

```
let x = 5 ;

let y = 6 ;

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

Oto wyjaśnienie, renderowane:

Najpierw przypisujemy x wartość pięć:

```
let x = 5 ;

# let y = 6 ;

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

Następnie przypisujemy sześć do y:

```
# let x = 5 ;

let y = 6 ;

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

Na koniec wypisujemy sumę x i y:

```
# let x = 5 ;

# let y = 6 ;

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

Oto to samo wyjaśnienie, w postaci zwykłego tekstu:

Najpierw przypisujemy x wartość pięć:

```
let x = 5;

# let y = 6;

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

Następnie przypisujemy sześć do y:

```
# let x = 5;

let y = 6;

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

Na koniec wypisujemy sumę x i y:

```
# let x = 5;

# let y = 6;

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

Powtarzając wszystkie części przykładu, możesz upewnić się, że Twój xample nadal się kompiluje, pokazując tylko części istotne dla twojego wyjaśnienia.

Dokumentowanie makr

Oto przykład dokumentacji do makra:

/// Panic with a provided message unless the expression is evaluated to true.

///

/// # Examples

///

///

/// # # [macro_use] extern crate foo; /// # fn main () {/// panic_unless! (1 + 1

== 2, "The mathematics is broken."); /// #} /// /// /// should_panic /// # #

[macro_use] extern crate foo; /// # fn main () {/// panic_unless! (true == false,

"I am broken."); /// #} /// ``

[macro_export]

macro_rules! panic_unless {($ condition: expr, $ ($ rest: expr), +) => ({if! $ condition {panic! ($ ($ rest), +);}}); }

fn main () {}

Zauważysz trzy rzeczy: musimy dodać własną linię `extern crate`, abyśmy mogli dodać atrybut` # [macro_use] `. Po drugie, będziemy musieli dodać własne `main()`. Na koniec rozsądne użycie znaku `#` w celu skomentowania tych dwóch rzeczy, tak aby zostały one pokazane w danych wyjściowych.

Uruchamianie testów dokumentacji

Aby uruchomić testy, możesz:

bash

$ rustdoc --test path / to / my / crate / root.rs

or

$ test charge

Prawidłowy, test ładowania testuje również wbudowaną dokumentację. Jednak ładuję test , nie przetestuje skrzynek binarnych, tylko biblioteki. Dzieje się tak ze względu na sposób działania rustdoc: łączy się on z biblioteką do przetestowania, ale w przypadku pliku binarnego nie ma do czego odsyłać. Jest jeszcze kilka adnotacji, które są przydatne, aby pomóc rustdoc zrobić właściwe rzeczy podczas testowania kodu:

/// `` `ignore

/// fn foo () {

///

fn foo () {}

Dyrektywa `ignore` mówi Rustowi, aby zignorował kod. Jest to forma, której prawie nigdy nie będziesz chciał, ponieważ jest najbardziej ogólna. Zamiast tego rozważ ocenianie za pomocą `text`, jeśli nie jest to kod, lub użyj `# `s, aby uzyskać działający przykład, który pokazuje tylko część, która Cię interesuje.

`` 'rust

/// `` `should_panic

/// assert! (false);

///

fn foo () {}

`should_panic` informuje` rustdoc`, że kod musi się poprawnie skompilować, ale bez konieczności pomyślnego przejścia testu.

`` 'rust

/// `` `no_run

/// loop {

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

///}

///

fn foo () {}

Atrybut `no_run` skompiluje twój kod, ale go nie wykona. Jest to ważne w przypadku przykładów takich jak „Oto jak uruchomić usługę sieciową”, które należy upewnić się, że się skompilują, ale może to spowodować nieskończoną pętlę!

Dokumentowanie modułów

Rust ma inny typ komentarza do dokumentacji, `//!`. Ten komentarz nie dokumentuje następnego elementu, komentuje element, który go zawiera. Innymi słowy:

`` 'rust

mod foo {

//! This is documentation for the `foo` module.

//!

//! # Examples

// ...

}

Tutaj zobaczysz //! najczęściej używany: do dokumentacji modułu. Jeśli masz moduł w foo.rs , często po otwarciu jego kodu zobaczysz to:

//! Moduł do używania `foo`s.

//!

//! Moduł `foo` zawiera wiele funkcjonalności bla bla blaInna dokumentacja

Wszystkie te zachowania działają również na plikach innych niż Rust. Ponieważ komentarze są zapisywane w języku Markdown, często są to pliki .md . Kiedy piszesz dokumentację do plików Markdown, nie potrzebujesz prefiksu

dokumentację wraz z komentarzami. Na przykład:

/// # Przykłady

///

///

/// use std :: rc :: Rc; /// /// let five = Rc :: new (5); /// ``` `

fn foo () {}

to poprostu

markdown

Examples

use std :: rc :: Rc;

let five = Rc :: new (5);

gdy znajduje się w pliku Markdown. Jest tylko jeden szczegół, pliki przeceny muszą mieć taki tytuł:

markdown

% Title

This is the sample documentation

Ten wiersz % musi znajdować się w pierwszym wierszu pliku.

atrybuty doc

Na głębszym poziomie komentarze do dokumentacji to inny sposób zapisywania atrybutów dokumentacji:

/// Este

fn foo () {}

[doc = "this"]

fn bar () {}

they are the same as these:

//! Este

#! [doc = "/// this"]

Ten atrybut nie jest często używany do pisania dokumentacji, ale może być przydatny podczas zmiany niektórych opcji lub pisania makra.

Reeksport

rustdoc pokaże dokumentację publicznego reeksportu w obu miejscach:

extern crate foo;

pub use foo :: bar;

Powyższe spowoduje utworzenie dokumentacji dla paska w ramach dokumentacji dla skrzyni foo , a także dokumentacji dla Twojej skrzynki. Będzie to ta sama dokumentacja w obu miejscach. To zachowanie można stłumić za pomocą no_inline :

extern crate foo;

[doc (no_inline)]

pub use foo :: bar;

Sterowanie HTMLem

Możesz kontrolować niektóre aspekty kodu HTML generowanego przez rustdoc poprzez plik

#! [Doc] version of the attribute:

#! [doc (html_logo_url = "http://www.rust-lang.org/logos/rust-logo-

128x128-blk-v2.png",

html_favicon_url = "http://www.rust-lang.org/favicon.ico",

html_root_url = "http://doc.rust-lang.org/")]

Spowoduje to skonfigurowanie kilku opcji z logo, ikoną ulubionych i głównym adresem URL.

Opcje generacji

rustdoc zawiera również kilka opcji w wierszu poleceń dostosowania:

--html-in-header FILE : includes the FILE content at the end of the <head> ... </head>

section .

--html-before-content FILE : includes the FILE content after <body> , before the

rendered content (including the search bar).

--html-after-content FILE - Includes the FILE content after all rendered content.

Uwaga dotycząca bezpieczeństwa

Markdown w komentarzach do dokumentacji jest ustawiony na surowo na ostatniej stronie. Uważaj na dosłowny HTML:

/// <script> alert (document.cookie) </script>

fn foo () {}

Iteratory

Porozmawiajmy o cyklach. Pamiętasz cykl Rust? Oto przykład:

```
for x in 0..10 {

println! ("{}", x);

}
```

Teraz, gdy wiesz więcej o Ruście, możemy szczegółowo omówić działanie tego kodu. Zakresy (0..10) są iteratorami. Iterator to coś, co możemy wielokrotnie wywoływać metodą .next(), a iterator dostarcza nam sekwencji elementów.

Na przykład:

```
let mut range = 0..10;

loop {

match rank.next () {

Some (x) => {

println! ("{}", x);

},

None => {break}

}

}
```

Tworzymy łącze (zmienną) do zakresu, naszego iteratora. Następnie przechodzimy przez cykl pętli z wewnętrznym dopasowaniem. To dopasowanie wykorzystuje wynik funkcji rank.next () , który daje nam odniesienie do następnej wartości w iteratorze. next zwraca opcję <i32> , w tym przypadku, która będzie Some (i32), gdy mamy wartość i None, gdy zabraknie nam wartości. Jeśli otrzymamy Some (i32) , wypisujemy to, a jeśli otrzymamy None , przerywamy pętlę, pozostawiając ją przez break . Ten przykładowy kod jest zasadniczo taki sam, jak nasza wersja pętli for for . Pętla for jest po prostu praktycznym sposobem napisania konstrukcji pętli/dopasowania/przerwania. Jednak pętle for nie są jedyną rzeczą, która wykorzystuje iteratory. Napisanie własnego iteratora wymaga zaimplementowania Iteratora Trait . Chociaż robienie tego wykracza poza zakres tego przewodnika, Rust zapewnia szereg przydatnych iteratorów do wykonywania różnych zadań. Zanim o tym porozmawiamy, powinniśmy porozmawiać o antypatronie. Wspomniany antywzorzec polega na użyciu zakresów w sposób opisany powyżej. Tak, właśnie rozmawialiśmy o tym, jak fajne są zakresy. Ale są też bardzo prymitywne. Na przykład, jeśli musimy przejrzeć zawartość wektora, możemy pokusić się o napisanie czegoś takiego:

```
let nums = vec! [1, 2, 3];

for i in 0..nums.len () {

println! ("{}", nums [i]);
```

}

Nie jest to ściśle gorsze niż użycie iteratora. Możesz bezpośrednio iterować do wektorów, napisz to:

```
let nums = vec! [1, 2, 3];

for num in & nums {

println! ("{}", num);

}
```

Istnieją dwa powody, dla których należy to zrobić w ten sposób. Po pierwsze, wyraź to, czego chcemy bardziej bezpośrednio. Iterujemy przez cały wektor, zamiast iterować po indeksach, a następnie indeksować wektor. Po drugie, ta wersja jest bardziej wydajna: w pierwszej wersji będziemy sprawdzać dodatkowe limity, ponieważ używa indeksowania, nums [i] . W drugim przykładzie i ponieważ w iteratorze podajemy odniesienie do każdego elementu w tym samym czasie, nie ma sprawdzania limitu. Jest to bardzo powszechne w iteratorach: możemy zignorować niepotrzebne kontrole limitów, jednocześnie wiedząc, że jesteśmy bezpieczni. Jest jeszcze jeden szczegół, który nie jest w 100% jasny z powodu operacji println! . num jest typu & i32 . Odniesienie do i32 , a nie i32 . drukuj! obsługuje dereferencje za nas, dlatego tego nie widzimy. Ten kod jest również poprawny:

```
let nums = vec! [1, 2, 3];

for num in & nums {

println! ("{}", * num);

}
```

Teraz odwołujemy się jawnie do num. Dlaczego & nums daje nam referencje? Po pierwsze dlatego, że wyraźnie o to prosimy za pomocą & . Po drugie, gdyby sam dał nam dane, musielibyśmy je posiadać, co wiązałoby się z utworzeniem kopii danych, a następnie przekazaniem nam tej kopii. Z referencjami pożyczamy tylko referencję do danych, a więc przekazujemy tylko referencję, bez konieczności przenoszenia członkostwa. Więc teraz, gdy ustaliliśmy, że zakresy czasami nie są tym, czego chcemy, porozmawiajmy o tym, czego chcemy. Istnieją trzy szerokie klasy rzeczy, które są istotne: iteratory, iterator , adaptery i konsumenci. Oto kilka definicji:

Iteratory zapewniają sekwencję wartości.

Adaptery iteratorów działają na jednym iteratorze, tworząc nowy iterator z inną sekwencją danych wyjściowych.

Konsumenci działają na iteratorze, tworząc ostateczny zestaw wartości.

Porozmawiajmy najpierw o konsumentach, ponieważ widzieliśmy już iterator, zakresy.

Konsumenci

Konsument działa na iteratorze, zwracając jakąś wartość lub wartości. Najczęstszym konsumentem jest collect() . Ten kod się nie kompiluje, ale pokazuje intencję:

```
let uno_hasta_cien = (1..101) .collect ();
```

Jak widać, na iteratorze wywołujemy funkcję collect(). collect() przyjmuje tyle wartości, ile zapewnia iterator, zwracając zbiór wyników. Dlaczego więc ten kod się nie kompiluje? Rust nie może określić,

jakie przedmioty chcesz kolekcjonować, dlatego musisz mu o tym powiedzieć. To jest wersja, która się kompiluje:

```
let uno_hasta_cien = (1..101) .collect :: <Vec <i32>> ();
```

Jeśli pamiętasz, składnia :: <> pozwala dać wskazówkę co do typu, w naszym przypadku mówimy, że chcemy wektora liczb całkowitych. Nie zawsze jest konieczne użycie pełnego typu. _ pozwoli ci podać częściową wskazówkę dotyczącą typu:

```
let one_up to one hundred = (1..101) .collect :: <Vec <_>> ();
```

To mówi „Zbierz w Vec <T> , proszę, ale wywnioskuj, że to T dla mnie.”. _ jest z tego powodu czasami nazywany „symbolem zastępczym typu”.

collect() jest najczęstszym konsumentem, ale są też inne. find() jest jednym z nich:

```
let mayor_a_cuarenta_y_dos = (0..100)

.find (| x | * x> 42);

match mayor_a_cuarenta_y_dos {

Some (_) => println! ("We have some numbers!"),

None => println! ("No numbers were found :("),

}
```

find otrzymuje zamknięcie i działa na odwołaniu do każdego elementu iteratora. Wspomniane zamknięcie zwraca wartość true, jeśli element jest tym, którego szukamy, a false w przeciwnym razie. Ponieważ możemy nie znaleźć elementu spełniającego nasze kryteria, find zwraca opcję zamiast elementu. Innym ważnym konsumentem jest fold . To wygląda tak:

```
let sum = (1..4) .fold (0, | sum, x | sum + x);
```

fold (base, | accumulator, element | ...) . Przyjmuje dwa argumenty: pierwszy to element o nazwie base . Drugi to domknięcie, które z kolei przyjmuje dwa argumenty: pierwszy nazywa się akumulatorem, a drugi elementem. W każdej iteracji wywoływane jest zamknięcie, a wynik jest używany jako wartość akumulatora w następnej iteracji. W pierwszej iteracji podstawą jest wartość akumulatora. Cóż, to trochę mylące. Przyjrzyjmy się wartościom wszystkich rzeczy w tym iteratorze:

baseaccumulatorelementclosure result

0 0 one one

0 one two 3

0 3 3 6

Wywołaliśmy fold() z tymi argumentami:

```
# (1..4)

.fold (0, | sum, x | sum + x);
```

Więc 0 to nasza podstawa, suma to nasz akumulator, a x to nasz element. W pierwszej iteracji przypisujemy sumę do 0, a x jest pierwszym elementem naszego zakresu, 1 . Następnie dodajemy sumę i x , co daje nam 0 + 1 = 1 . W drugiej iteracji ta wartość staje się wartością naszego akumulatora, sum

, a elementem jest drugi element w zakresie, 2 . 1 + 2 = 3 i podobnie staje się wartością akumulatora dla ostatniej iteracji. W tej iteracji x jest ostatnim elementem, 3 , a 3 + 3 = 6 , końcowym wynikiem naszej sumy . 1 + 2 + 3 = 6 , to jest wynik, który otrzymujemy.

Uff. Fold może na pierwszy rzut oka wydawać się trochę dziwny, ale kiedy kliknie, możesz go używać wszędzie. Zawsze, gdy masz listę rzeczy i potrzebujesz pojedynczego wyniku, odpowiednie jest spasowanie. Konsumenci są ważni ze względu na dodatkową właściwość iteratorów, o której jeszcze nie mówiliśmy: lenistwo. Porozmawiajmy więcej o iteratorach, a przekonasz się, dlaczego konsumenci są ważni.

Iteratory

Jak powiedzieliśmy wcześniej, iterator to coś, co możemy wielokrotnie wywoływać metodą .next() i zwraca sekwencję elementów. Ponieważ musimy wywołać metodę, iteratory mogą być leniwe i nie generować wszystkich wartości z góry. Na przykład ten kod nie generuje liczb 1-99 . Zamiast tego tworzy wartość reprezentującą sekwencję:

```
let nums = 1..100;
```

Ponieważ nie zrobiliśmy nic z zakresem, sekwencja nie została wygenerowana. Dodajmy konsumenta:

```
let nums = (1..100) .collect :: <Vec <i32>> ();
```

Teraz funkcja collect() będzie wymagać podania pewnych liczb z zakresu, a zatem będzie musiała wykonać pracę polegającą na wygenerowaniu sekwencji. Zakresy to jedna z dwóch podstawowych form iteratorów, z którymi się spotkasz. Drugi to iter() . iter () może przekształcić wektor w prosty iterator, który zapewnia jeden element na raz:

```
let nums = vec! [1, 2, 3];

for num in nums.iter () {

println! ("{}", num);

}
```

Te dwa podstawowe iteratory powinny być przydatne. Istnieją bardziej zaawansowane iteratory, w tym te, które są nieskończone. Dość o iteratorach, adaptery iteratorowe to ostatnia koncepcja związana z iteratorami, o której powinniśmy wspomnieć. Zróbmy to!

Adaptery iteratorów

Iteratory adapterów pobierają iterator i modyfikują go w jakiś sposób, tworząc nowy. Najprostszy nazywa się mapa:

```
(1..100) .map (| x | x + 1);
```

map jest wywoływana w innym iteratorze, map tworzy nowy iterator, w którym każde odwołanie do elementu ma domknięcie, które zostało podane jako argument. Poprzedni kod da nam liczby 2-100, no, prawie! Jeśli skompilujesz przykład, otrzymasz ostrzeżenie:

```
warning: unused result which must be used: iterator adapters are lazy

and do nothing unless consumed, # [warn (unused_must_use)] on by

default
```

(1..100) .map (| x | x + 1);

Lenistwo znowu atakuje! To zamknięcie nigdy nie zostanie uruchomione. Ten przykład nie drukuje żadnej liczby:

(1..100) .map (| x | println! ("{}", X));

Jeśli próbujesz uruchomić zamknięcie w iteratorze, aby uzyskać jego skutki uboczne, użyj for .

for i in (1 ..). take (5) {

println! ("{}", i);

}

Spowoduje to wyświetlenie:

jeden

dwa

345

filter() to adapter, który jako argument przyjmuje domknięcie. To zamknięcie zwraca wartość true lub false . Nowy iterator, który filter() tworzy tylko elementy, dla których zamknięcie zwraca true :

for i in (1..100) .filter (| & x | x% 2 == 0) {

println! ("{}", i);

}

Spowoduje to wydrukowanie wszystkich liczb parzystych od jednego do stu. (Zauważ, że ponieważ filtr nie zużywa iterowanych elementów, przekazywane jest do niego odwołanie do każdego elementu, dlatego predykat używa wzorca & x do wyodrębnienia liczby całkowitej).

(one..)

.filter (| & x | x% 2 == 0)

.filter (| & x | x% 3 == 0)

.take (5)

.collect :: <Vec <i32>> ();

Powyższe daje wektor zawierający 6 , 12 , 18 , 24 i 30 . To jest mała próbka rzeczy, które iteratory, adaptery iteratorów i konsumenci mogą ci pomóc. Istnieje wiele naprawdę przydatnych iteratorów, a także fakt, że możesz napisać własny. Iteratory zapewniają bezpieczny i wydajny sposób manipulowania wszystkimi rodzajami list. Na pierwszy rzut oka są nieco niezwykłe, ale jeśli trochę się nimi pobawisz, uzależnisz się.

Zbieżność

Współbieżność i równoległość to dwa niezwykle ważne tematy w informatyce, są one również gorącym tematem w dzisiejszej branży. Komputery coraz częściej mają coraz więcej rdzeni, mimo to wciąż niektórzy programiści nie są gotowi, aby w pełni z nas korzystać. Bezpieczeństwo zarządzania pamięcią Rust dotyczy również historii współbieżności. Nawet współbieżne programy muszą być bezpieczne dla

pamięci, bez warunków wyścigu. System typów w Rust sprosta wyzwaniu i daje potężne możliwości wnioskowania o współbieżnym kodzie w czasie kompilacji. Zanim zaczniemy mówić o funkcjach współbieżności, które są dostępne w Rust, ważne jest, aby coś zrozumieć: Rust jest na niskim poziomie, na tyle niskim, że wszystkie te funkcje są zaimplementowane w standardowej bibliotece. Oznacza to, że jeśli nie podoba ci się jakiś aspekt sposobu, w jaki Rust obsługuje współbieżność, możesz zaimplementować alternatywny sposób robienia tego. Mój jest żywym przykładem działania tej zasady.

Wątki

Standardowa biblioteka Rust zapewnia bibliotekę do obsługi wątków, która pozwala na równoległe uruchamianie kodu rdzy. Oto podstawowy przykład użycia std ::

thread :

```
use std :: thread;

fn main () {

thread :: spawn (|| {

println! ( "Hello from a thread!" );

});

}
```

Metoda thread :: spawn () przyjmuje jako argument zamknięcie, które jest wykonywane w nowym wątku. thread :: spawn () zwraca uchwyt do nowego wątku, którego można użyć do oczekiwania na zakończenie wątku, a następnie wyodrębnić jego wynik:

```
use std :: thread;

fn main () {

let handle = thread :: spawn (|| {

"Hello from a thread!"

});

println! ( "{}" , handle.join (). unwrap ());

}
```

Wiele języków ma możliwość uruchamiania wątków, ale jest to bardzo niepewne. Istnieją całe książki o tym, jak zapobiegać błędom, które występują w wyniku współdzielenia stanu zmiennego. Rust pomaga w swoim systemie stawek, zapobiegając warunkom wyścigu w czasie kompilacji. Porozmawiajmy o tym, jak skutecznie udostępniać rzeczy między wątkami.

Współdzielony sejf o zmiennym stanie

Ze względu na system typów Rusta mamy koncepcję, która brzmi jak kłamstwo: „bezpieczny współdzielony stan zmienny”. Wielu programistów zgadza się, że współdzielony stan mutable jest bardzo, bardzo zły. Ktoś kiedyś powiedział: Wspólny, zmienny stan jest źródłem wszelkiego zła. Większość języków próbuje poradzić sobie z tym problemem poprzez część „zmienną”, ale Rust konfrontuje się z tym, rozwiązując część „wspólną”. To samo członkostwo, które zapobiega

niewłaściwemu użyciu wskaźników, pomaga również wyeliminować warunki wyścigu, jeden z najgorszych błędów związanych ze współbieżnością. Jako przykład, program Rust, który miałby warunek kariery w wielu językach. W Rust nie skompilowałbym:

```
use std :: thread;

fn main () {

let mut data = vec! [1u32, 2, 3];

for i in 0..3 {

thread :: spawn (move || {

data [i] + = 1;

});

}

thread :: sleep_ms (50);

}
```

Powyższe powoduje błąd:

```
8:17 error: capture of moved value: `data`

data [i] + = 1;

^ ~~~
```

W tym przypadku wiemy, że nasz kod powinien być bezpieczny, ale Rust nie jest tego pewien. Nie jest to do końca bezpieczne, gdybyśmy mieli odniesienie do danych w każdym wątku, a wątek należy do odniesienia, mielibyśmy trzech właścicieli! To jest złe. Możemy to naprawić, używając typu Arc <T> , wskaźnika z atomową liczbą odwołań. Część „atomowa” oznacza, że udostępnianie między wątkami jest bezpieczne. Arc <T> zakłada jeszcze jedną właściwość dotyczącą swojej zawartości, aby upewnić się, że można ją bezpiecznie udostępniać między wątkami: zakłada, że jej zawartość to Sync . Ale w naszym przypadku chcemy zmutować wartość. Potrzebujemy faceta, który upewni się, że tylko jedna osoba naraz może zmutować to, co jest w środku. W tym celu możemy użyć typu Mutex <T> . Oto druga wersja naszego kodu. Nadal nie działa, ale z innego powodu:

```
use std :: thread;

use std :: sync :: Mutex;

fn main () {

let mut data = Mutex :: new (vec! [1u32, 2, 3]);

for i in 0..3 {

let data = data.lock (). unwrap ();

thread :: spawn (move || {

data [i] + = 1;

});
```

```
}

thread :: sleep_ms (50);

}
```

Oto błąd:

```
: 9: 9: 9:22 error: the trait `core :: marker :: Send` is not implemented for the type` std :: sync :: mutex :: MutexGuard <'_, collections :: vec ::

Vec> `[E0277]

: 11 thread :: spawn (move || {

^ ~~~~~~~~~~~~

: 9: 9: 9:22 note: `std :: sync :: mutex :: MutexGuard <'_, collections ::

vec :: Vec>` cannot be sent between threads safely

: 11 thread :: spawn (move || {

^ ~~~~~~~~~~~~
```

Widzisz, Mutex ma metodę blokady, która ma tę sygnaturę:

```
fn lock (& self) -> LockResult <mutexguard>

</ mutexguard
```

Ponieważ wysyłanie nie jest zaimplementowane dla MutexGuard <T> , nie możemy przenieść MutexGuard <T> między wątkami, co powoduje błąd. Możemy użyć Arc <T>, aby poprawić błąd. Oto funkcjonalna wersja:

```
use std :: sync :: {Arc, Mutex};

use std :: thread;

fn main () {

let data = Arc :: new (Mutex :: new ( vec! [ 1u32 , 2 , 3 ]));

for i in 0 .. 3 {

let data = data.clone ();

thread :: spawn (move || {

let mut data = data.lock (). unwrap ();

data [i] + = 1 ;

});

}

thread :: sleep_ms ( 50 );

}
```

Teraz wywołujemy clone() na naszym Arc , co zwiększa wewnętrzny licznik. Ten uchwyt jest następnie przenoszony do nowego wątku. Przyjrzyjmy się bliżej treści wątku:

```
# use std :: sync :: {Arc, Mutex};

# use std :: thread;

# fn main () {

# let data = Arc :: new (Mutex :: new ( vec! [ 1u32 , 2 , 3 ]));

# for i in 0 .. 3 {

# let data = data.clone ();

thread :: spawn (move || {

let mut data = data.lock (). unwrap ();

data [i] + = 1 ;

});

#}

# thread :: sleep_ms ( 50 );

#}
```

Najpierw wywołujemy lock() , która pobiera blokadę wzajemnego wykluczenia. Ponieważ ta operacja może się nie powieść, ta metoda zwraca Result <T, E> , a ponieważ jest to tylko przykład, wykonujemy na nim unwrap (), aby uzyskać odwołanie do danych. Prawdziwy kod miałby tutaj bardziej niezawodną obsługę błędów. Możemy go mutować, ponieważ mamy blokadę. Wreszcie, gdy wątki działają, czekamy na kulminację krótkiego timera. To nie jest idealne: mogliśmy wybrać rozsądny czas oczekiwania, ale najprawdopodobniej będziemy czekać dłużej niż to konieczne lub za krótko, w zależności od tego, ile czasu zajmie zakończenie obliczeń przez wątki podczas działania programu. Bardziej precyzyjną alternatywą dla licznika czasu byłoby użycie jednego z mechanizmów zapewnianych przez standardową bibliotekę Rust do synchronizacji między wątkami. Porozmawiajmy o nich: kanały.

Kanały

Oto wersja naszego kodu, która używa kanałów do synchronizacji, zamiast czekać na określony czas:

```
use std :: sync :: {Arc, Mutex};

use std :: thread;

use std :: sync :: mpsc;

fn main () {

let data = Arc :: new (Mutex :: new ( 0u32 ));

let (tx, rx) = mpsc :: channel ();

for _ in 0 .. 10 {

let (data, tx) = (data.clone (), tx.clone ());
```

```
thread :: spawn (move || {

let mut data = data.lock (). unwrap ();

* data + = 1 ;

tx.send (());

});

}

for _ in 0 .. 10 {

rx.recv ();

}

}
```

Używamy metody mpsc :: channel () do zbudowania nowego kanału. Wysyłamy (przez send) kanałem pojedynczą (), a następnie czekamy na powrót dziesięciu z nich. Chociaż ten kanał wysyła tylko ogólny sygnał, możemy wysłać dowolne dane, które są wysyłane przez kanał!

```
use std :: thread;

use std :: sync :: mpsc;

fn main () {

let (tx, rx) = mpsc :: channel ();

for _ in 0 .. 10 {

let tx = tx.clone ();

thread :: spawn (move || {

let answer = 42u32 ;

tx.send (response);

});

}

rx.recv (). ok (). expect ( "The response could not be received" );

}
```

U32 jest wysłane, ponieważ możemy zrobić jego kopię. Tworzymy więc wątek i prosimy go o obliczenie odpowiedzi, a następnie wysyła odpowiedź z powrotem do nas (za pomocą send()) przez kanał.

Paniki

Panika! spowoduje nagłe zakończenie bieżącego wątku. Możesz użyć wątków Rust jako prostego mechanizmu izolacji:

```
use std :: thread;
```

```
let result = thread :: spawn (move || {

panic! ( "ups!" );

}). join ();

assert! (result.is_err ());
```

Nasz Thread zwraca Result , co pozwala nam sprawdzić, czy wątek jest spanikowany, czy nie.

Obsługa błędów

Plany najlepiej opracowane przez myszy i ludzi często się nie udają. „Tae a Moose”, Robert Burns Czasami coś idzie nie tak. Ważne jest, aby mieć plan na wypadek nieuniknionego zdarzenia. Rust ma bogate wsparcie dla obsługi błędów, które mogą (bądźmy szczerzy: wystąpią) w twoich programach. Istnieją dwa rodzaje błędów, które mogą wystąpić w twoich programach: awaria i panika. Porozmawiamy o różnicach między nimi, a następnie omówimy, jak sobie z nimi poradzić. Następnie omówimy, jak promować wady paniki.

Porażka kontra panika

Rust używa dwóch terminów, aby rozróżnić dwie formy błędu: niepowodzenie i panika. Awaria to błąd, z którego możemy jakoś się podnieść. Panika to nieodwracalny błąd. Co rozumiemy przez „odzyskać”? Cóż, w większości przypadków spodziewana jest możliwość wystąpienia błędu. Rozważmy na przykład funkcję parsowania:

```
"5" .parse ();
```

Ta metoda konwertuje ciąg znaków na inny typ. Ale ponieważ jest to ciąg znaków, nie możesz być pewien, że konwersja rzeczywiście się powiedzie. Na przykład, na co należy to przekonwertować?:

```
"hello5world" .parse ();
```

Powyższe nie działa. Wiemy, że metoda parse() będzie skuteczna tylko dla niektórych wpisów. Jest to oczekiwane zachowanie. Dlatego nazywamy ten błąd niepowodzeniem. Z drugiej strony czasami zdarzają się błędy, które są nieoczekiwane lub których nie możemy naprawić. Klasycznym przykładem jest asercja! :

```
# let x = 5 ;

assert! (x == 5 );
```

Używamy asercji! oświadczyć, że coś jest prawdą (prawdą). Jeśli stwierdzenie nie jest prawdziwe, to coś jest bardzo nie tak. Na tyle źle, że nie możesz kontynuować biegania w obecnym stanie. Innym przykładem jest użycie nieosiągalnego! () Makro:

```
use Event :: NewLaunch;

enum Event {

New Launch,

}

fn probability (_: & Event) -> f64 {

// the actual implementation would be more complex, of course
```

```
0.95
}
fn descriptive_probability (event: Event) -> & 'static str {
match probability (& event) {
1.00 => "true",
0.00 => "impossible",
0.00 ... 0.25 => "very unlikely",
0.25 ... 0.50 => "unlikely",
0.50 ... 0.75 => "probable",
0.75 ... 1.00 => "very likely",
}
}
fn main () {
println! ("{}", descriptive_probability (NewLaunch));
}
```

Powyższe spowoduje błąd:

error: non-exhaustive patterns: `_` not covered [E0004]

Chociaż wiemy, że omówiliśmy każdy możliwy przypadek, Rust może nie wiedzieć. Nie wiesz, jakie jest prawdopodobieństwo między 0,0 a 1,0. Dlatego dodajemy kolejny przypadek:

```
use Event :: NewLaunch;
enum Event {
New Launch,
}
fn probability (_: & Event) -> f64 {
// the actual implementation would be more complex, of course
0.95
}
fn probabilidad_descriptiva (event: Event) -> & 'static str {
match probability (& event) {
1.00 => "true" ,
0.00 => "impossible" ,
```

```
0.00 ... 0.25 => "very unlikely" ,

0.25 ... 0.50 => "unlikely" ,

0.50 ... 0.75 => "probable" ,

0.75 ... 1.00 => "very likely" ,

_ => unreachable! ()

}

}

fn main () {

println! ( "{}" , descriptive_probability (NewLaunch));

}
```

Nigdy nie powinniśmy dojść do przypadku _ , dlatego używamy makra, aby to wskazać. unreacheable!() generuje inny typ błędu niż Result . Rust nazywa takie błędy paniką.

Obsługa błędów za pomocą opcji i wyniku

Najprostszym sposobem wskazania, że funkcja może zawieść, jest użycie typu Option <T> . Na przykład metoda find na łańcuchach próbuje zlokalizować wzorzec w ciągu, zwraca Option :

```
let s = "foo" ;

assert_eq! (s.find ( 'f' ), Some ( 0 ));

assert_eq! (s.find ( 'z' ), None );
```

Jest to odpowiednie dla prostych przypadków, ale nie daje nam zbyt wielu informacji w przypadku awarii. Co by było, gdybyśmy chcieli wiedzieć, dlaczego funkcja się nie powiodła? W tym celu możemy użyć typu Result <T, E> . Jak to wygląda:

```
enum Result <T, E> {

Ok (T),

Err (E)

}
```

To wyliczenie jest dostarczane przez Rust, dlatego nie musisz go definiować, jeśli chcesz go użyć. Wariant Ok (T) reprezentuje sukces, a wariant Err (E) reprezentuje porażkę. Zwrócenie wyniku zamiast opcji jest zalecane w większości nietrywialnych przypadków:

Oto przykład użycia Result :

```
# [derive (Debug)]

enum Version {Version1, Version2}

# [derive (Debug)]

enum ErrorParseo {InvalidHeadLength, Invalid Version}
```

```
fn parsear_version (header: & [ u8 ]) -> Result <Version, ErrorParseo> {
if header.len () < 1 {
return Err (ErrorParseo :: InvalidHeadLength);
}
header match [ 0 ] {
1 => Ok (Version :: Version1),
2 => Ok (Version :: Version2),
_ => Err (ErrorParseo :: InvalidHeadLength)
}
}
fn main () {
let version = parsear_version (& [ 1 , 2 , 3 , 4 ]);
match version {
Ok (v) => {
println! ( "working with version: {:?}" , v);
}
Err (e) => {
println! ( "head parsing error: {:?}" , e);
}
}
}
```

Ta funkcja korzysta z enum, ErrorParseo, aby wyświetlić listę błędów, które mogą wystąpić. Cecha Debug to ta, która pozwala nam wydrukować wartość wyliczeniową przy użyciu operacji formatowania {:?} .

Nieodwracalne błędy z paniką!

W przypadku nieoczekiwanego błędu, którego nie można naprawić, makro panic! Służy do wywołania paniki. Taka panika nagle zakończy bieżący wątek wykonania, wyświetlając komunikat o błędzie:

```
panic! ("boom");
```

Powoduje to

wątek „” spanikował na „boom”, hello.rs:2

kiedy go uruchomisz.

Ponieważ takie sytuacje są stosunkowo rzadkie, używaj paniki oszczędnie.

Promowanie niepowodzeń w panice

W pewnych okolicznościach, nawet wiedząc, że dana funkcja może zawieść, możemy chcieć potraktować tę awarię jako panikę. Na przykład io :: stdin (). Read_line (& bufor mut) zwraca Result <usize> , gdy wystąpi błąd odczytu linii. To pozwala nam zarządzać i ewentualnie odzyskać w przypadku błędu. Jeśli nie chcemy obsłużyć błędu i zamiast tego po prostu przerwać działanie programu, możemy użyć metody unwrap() :

io :: stdin (). read_line (& mut buffer) .unwrap ();

unwrap() wpadnie w panikę (panikę!), jeśli Result to Err. To w zasadzie mówi: „Daj mi wartość, a jeśli coś pójdzie nie tak, po prostu przerwij wykonanie". Jest to mniej niezawodne niż dopasowanie błędu i próba odzyskania, ale jednocześnie jest znacznie krótsze. Czasami nagłe zakończenie jest właściwe. Istnieje sposób na wykonanie powyższego, który jest trochę lepszy niż unwrap ():

let mut buffer = String :: new ();

let read_bytes = io :: stdin (). read_line (& mut buffer)

.okay()

.expect ("Failed to read line");

ok() konwertuje Result na Option , a oczekiwanie () robi to samo co unwrap () , ale otrzymuje komunikat jako argument. Ta wiadomość jest przekazywana do paniki! podstawowego, zapewniając lepszy komunikat o błędzie.

Używaj try!

Kiedy piszemy kod, który wywołuje wiele funkcji zwracających typ Result , obsługa błędów może być uciążliwa. Makro próbne! Ukrywa część powtarzającego się kodu do propagacji błędów w stosie wywołań.

try! replaces:

use std :: fs :: File;

use std :: io;

use std :: io :: prelude :: *;

struct info {

name: String ,

age: i32 ,

grade: i32 ,

}

fn escribir_info (info: & Info) -> io :: Result <()> {

let mut file = File :: create ("my_best_friends.txt") .unwrap ();

if let Err (e) = writeln! (& mut file, "name: {}" , info.name) {

```
return Err (e)
}
if let Err (e) = writeln! (& mut file, "age: {}" , info.edad) {
return Err (e)
}
if let Err (e) = writeln! (& mut file, "grade: {}" , info.rgrado) {
return Err (e)
}
return Ok (());
}
```

Z:

```
use std :: fs :: File;
use std :: io;
use std :: io :: prelude :: *;
struct info {
name: String ,
age: i32 ,
grade: i32 ,
}
fn escribir_info (info: & Info) -> io :: Result <()> {
let mut file = File :: create ( "my_best_friends.txt" ) .unwrap ();
try! ( writeln! (& mut file, "name: {}" , info.name));
try! ( writeln! (& mut file, "age: {}" , info.edad));
try! ( writeln! (& mut file, "grade: {}" , info.grad));
return Ok (());
}
```

Otocz wyrażenie try! spowoduje pomyślne rozpakowanie wartości (Ok), chyba że wynikiem jest Err , w którym to przypadku Err jest zwracane wcześniej przez funkcję, która zawija próbę. Ważne jest, aby wspomnieć, że możesz użyć tylko try! z funkcji, która zwraca Result , co oznacza, że nie możesz użyć try! wewnątrz main() , ponieważ main() nic nie zwraca.

Interfejs funkcji obcych / zewnętrznych

Wstęp

W tym przewodniku wykorzystamy bibliotekę kompresji/dekompresji snappy jako wprowadzenie do pisania powiązań z zewnętrznym kodem. Rust obecnie nie może bezpośrednio wywoływać kodu w bibliotece C++, ale snappy zawiera interfejs C (udokumentowany w snappy-ch). Poniżej znajduje się przydatny przykład wywołania obcej funkcji, która skompiluje się, zakładając, że Snappy jest zainstalowany:

```
# #! [feature (libc)]

extern crate libc;

use libc :: size_t;

# [link (name = "snappy")]

external

fn snappy_max_compressed_length (source_length: size_t) -> size_t;

}

fn main () {

let x = unsafe {snappy_max_compressed_length (100)};

println! ("maximum length of a 100 bytes compressed buffer: {}", x);

}
```

Blok extern to lista sygnatur funkcji w obcej bibliotece, w tym przypadku z binarnym interfejsem aplikacji (ABI) platformy C. Atrybut # [link (...)] jest używany do instruowania konsolidatora, aby łączył się z biblioteką snappy, aby można było rozpoznać symbole. Zakłada się, że interfejsy do obcych funkcji są niepewne, dlatego ich wywołania muszą znajdować się wewnątrz niebezpiecznego bloku {} jako obietnica dla kompilatora, że wszystko, co jest w nim zawarte, jest naprawdę bezpieczne. Biblioteki w C czasami ujawniają interfejsy, które nie są bezpieczne wątkowo, a prawie każda funkcja, która przyjmuje wskaźnik jako argument, nie jest poprawna dla wszystkich możliwych danych wejściowych, ponieważ wskaźnik może być wiszącym wskaźnikiem, a płaskie wskaźniki są pozostawiane w tyle. poza bezpiecznym modelem pamięci Rusta. Deklarując typy argumentów funkcji obcej, kompilator Rust nie może sprawdzić, czy deklaracja jest poprawna, więc jej poprawne określenie jest część utrzymywania poprawnego wiązania w czasie wykonywania. Blok extern można rozszerzyć, aby obejmował pełny interfejs API snappy:

```
# #! [feature (libc)]

extern crate libc;

use libc :: {c_int, size_t};

# [link (name = "snappy")]

external

fn snappy_compress (input: * const u8,

input_length: size_t,

compressed: * mut u8,
```

```
compressed_length: * mut size_t) -> c_int;

fn snappy_uncompress (compressed: * const u8,

compressed_length: size_t,

uncompressed: * mut u8,

uncompressed_length: * mut size_t) -> c_int;

fn snappy_max_compressed_length (source_length: size_t) -> size_t;

fn snappy_uncompressed_length (compressed: * const u8,

compressed_length: size_t,

result: * mut size_t) -> c_int;

fn snappy_validate_compressed_buffer (compressed: * const u8,

compressed_length: size_t) -> c_int;

}

# fn main () {}
```

Tworzenie bezpiecznego interfejsu

Płaski interfejs C musi być opakowany, aby zapewnić bezpieczeństwo zarządzania pamięcią, a także wykorzystanie koncepcji wysokiego poziomu, takich jak wektory. Biblioteka może wybierać między ujawnieniem bezpiecznego interfejsu wysokiego poziomu a ukryciem niezabezpieczonych szczegółów wewnętrznych. Zawijanie funkcji oczekujących na bufory wymaga użycia modułu slice :: raw do manipulowania wektorami Rusta jako wskaźnikami pamięci. Gwarantuje się, że wektory rdzy będą ciągłym blokiem pamięci. Długość to liczba aktualnie zawartych elementów, a pojemność to całkowity rozmiar przydzielonej pamięci. Długość jest mniejsza lub równa pojemności.

```
# #! [feature (libc)]

# extern crate libc;

# use libc :: {c_int, size_t};

# unsafe fn snappy_validate_compressed_buffer (_: * const u8 , _: size_t)

-> c_int { 0 }

# fn main () {}

pub fn validate_buffer_compressed (src: & [ u8 ]) -> bool {

unsafe {

snappy_validate_compressed_buffer (src.as_ptr (), src.len () as

size_t) == 0

}

}
```

Funkcja opakowująca kompresję bufora_bufora korzysta z niebezpiecznego bloku, ale upewnia się, że wywołanie go jest bezpieczne dla wszystkich danych wejściowych, wykluczając niebezpieczne z sygnatury funkcji. Funkcje snappy_compress i snappy_uncompress są bardziej złożone, ponieważ do zachowania danych wyjściowych należy przypisać bufor. Funkcji snappy_max_compressed_length można użyć do przypisania wektorowi maksymalnej pojemności wymaganej do przechowywania skompresowanych danych wyjściowych. Wektor można następnie przekazać do funkcji snappy_compress jako parametr wyjściowy. Parametr wyjściowy jest również przekazywany, aby uzyskać rzeczywistą długość po kompresji w celu przypisania długości.

```
# #! [feature (libc)]

# extern crate libc;

# use libc :: {size_t, c_int};

# unsafe fn snappy_compress (a: * const u8 , b: size_t, c: * mut u8 ,

# d: * mut size_t) -> c_int { 0 }

# unsafe fn snappy_max_compressed_length (a: size_t) -> size_t {a}

# fn main () {}

pub fn compress (orig: & [ u8 ]) -> Vec < u8 > {

unsafe {

let long_orig = orig.len () as size_t;

let porig = orig.as_ptr ();

let mut long_dest = snappy_max_compressed_length (long_orig);

let mut dest = Vec :: with_capacity (long_dest as usize);

let pdest = dest.as_mut_ptr ();

snappy_compress (porig, long_orig, pdest, & mut long_dest);

dest.set_len (long_dest as usize);

dest

}

}
```

Dekompresja jest podobna, ponieważ snappy przechowuje zdekompresowaną długość jako część formatu kompresji, a snappy_uncompressed_length uzyska dokładny rozmiar wymaganego bufora.

```
# #! [feature (libc)]

# extern crate libc;

# use libc :: {size_t, c_int};

# unsafe fn snappy_uncompress (compressed: * const u8 ,

# compressed_length: size_t,
```

```
# uncompressed: * mut u8 ,
# uncompressed_length: * mut size_t) -> c_int { 0 }
# unsafe fn snappy_uncompressed_length (compressed: * const u8 ,
# compressed_length: size_t,
# result: * mut size_t) -> c_int { 0 }
# fn main () {}
pub fn unzip (orig: & [ u8 ]) -> Option < Vec < u8 >> {
unsafe {
let long_orig = orig.len () as size_t;
let porig = orig.as_ptr ();
let mut long_dest: size_t = 0 ;
snappy_uncompressed_length (porig, long_orig, & mut
long_dest);
let mut dest = Vec :: with_capacity (long_dest as usize);
let pdest = dest.as_mut_ptr ();
if snappy_uncompress (porig, long_orig, pdest, & mut long_dest)
== 0 {
dest.set_len (long_dest as usize);
Some (dest)
} else {
None // SNAPPY_INVALID_INPUT
}
}
}
```

Niszczyciele

Biblioteki zewnętrzne zwykle przenoszą członkostwo zasobów do kodu wywołującego. Kiedy to nastąpi, musimy użyć niszczycieli rdzy, aby zapewnić bezpieczeństwo i gwarancję uwolnienia tych zasobów (zwłaszcza w przypadku paniki).

Wywołania zwrotne z kodu C do funkcji Rust

Niektóre biblioteki zewnętrzne wymagają użycia wywołań zwrotnych do zgłaszania wywołującemu ich stanu lub częściowych danych. Funkcje zdefiniowane w Rust można przekazać do zewnętrznej biblioteki. Warunkiem jest to, aby funkcja wywołania zwrotnego była oznaczona jako zewnętrzna i z poprawną konwencją wywoływania, aby umożliwić wywołanie z kodu C. Funkcję wywołania zwrotnego

można następnie wysłać poprzez wywołanie rejestru do biblioteki w C i jej późniejsze wywołanie stamtąd. Podstawowym przykładem jest kod Rusta:

```
extern fn callback (a: i32) {

println! ("I have been called from C with the value {0}", a);

}

# [link (name = "extlib")]

external

fn register_callback (cb: extern fn (i32)) -> i32;

fn shoot_callback ();

}

fn main () {

unsafe {

register_callback (callback);

shoot_callback (); // Trigger the callback

}

}
```

Code C:

```
typedef void (* callback_rust) (int32_t) ;

callback_rust cb;

int32_t register_callback (callback_rust callback) {

cb = callback;

return 1 ;

}

void shoot_callback () {

cb ( 7 ); // Call callback (7) in Rust

}
```

W tym przykładzie main() Rusta wywoła trigger_callback() w C, co z kolei wywoła callbackback() w Rust.

Kierowanie wywołań zwrotnych na obiekty Rust

Powyższy przykład pokazał, jak funkcja globalna może zostać wywołana z kodu C. Jeśli czasami chwytam, chcę, aby wywołanie zwrotne wskazywało na specjalny obiekt Rusta. Może to być obiekt reprezentujący opakowanie dla odpowiedniego obiektu w C. Wszystko to można osiągnąć, przekazując płaski wskaźnik do biblioteki C. Biblioteka C może następnie dołączyć wskaźnik do obiektu Rust w

powiadomieniu. Umożliwi to wywołaniu zwrotnemu niepewny dostęp do przywoływanego obiektu Rust.

Kod Rust:

```
# [repr (C)]
struct RustObject {
a: i32,
// other members
}
extern "C" fn callback (target: mut ObjectRust, to: i32) {
println! ("I have been called from C with the value {0}", a);
unsafe {
// Update the value in RustObject with the value received from the
callback
(* target) .a = a;
}
}
# [link (name = "extlib")]
external
fn register_callback (target: mut ObjectRust,
cb: extern fn (* mut ObjectRust, i32)) -> i32;
fn shoot_callback ();
}
fn main () {
// Creating the object that will be referenced in the callback
let mut object_rust = Box :: new (ObjectRust {a: 5});
unsafe {
register_callback (& mut * object_rust, callback);
shoot_callback ();
}
}
```

Code C:

```
typedef void (* callback_rust) ( void *, int32_t) ;

void * target_cb;

callback_rust cb;

int32_t register_callback ( void * callback_target, callback_rust

callback) {

cb_target = callback_target;

cb = callback;

return 1 ;

}

void shoot_callback () {

cb (cb_target, 7 ); // I will call callback (& RustObject, 7) in Rust

}
```

Asynchroniczne wywołania zwrotne

W powyższych przykładach wywołania zwrotne są wywoływane jako bezpośrednia reakcja na wywołanie funkcji do biblioteki zewnętrznej w C. Kontrola nad bieżącym wątkiem jest zmieniana z Rust na C w celu wykonania wywołania zwrotnego, ale ostatecznie wywołanie zwrotne jest wykonywane w tym samym wątku który wywołał funkcję, która wywołała wywołanie zwrotne. Sprawy komplikują się, gdy zewnętrzna biblioteka tworzy własne wątki i wywołuje z nich wywołania zwrotne. W takich przypadkach dostęp do struktur danych Rust w ramach wywołań zwrotnych jest szczególnie niepewny i należy zastosować odpowiednie mechanizmy synchronizacji. Oprócz klasycznych mechanizmów synchronizacji, takich jak muteksy, jedną z możliwości w Rust jest użycie kanałów (w std :: sync :: mpsc) do przesyłania danych z wątku C, który wywołuje wywołanie zwrotne do wątku Rusta. Jeśli asynchroniczne wywołanie zwrotne dotyczy specjalnego obiektu w przestrzeni adresowej Rust, jest również absolutnie konieczne, aby żadne inne wywołania zwrotne nie były wykonywane przez bibliotekę C po zniszczeniu odpowiedniego obiektu Rust. Można to zrobić poprzez wyrejestrowanie wywołania zwrotnego w destruktorze obiektu i zaprojektowanie biblioteki w sposób gwarantujący, że żadne wywołanie zwrotne nie zostanie wykonane po wyrejestrowaniu.

Link

Atrybut link w blokach extern zapewnia podstawowy blok budulcowy, który instruuje rustc, jak łączyć się z bibliotekami natywnymi. Obecnie istnieją dwie formy atrybutu link:

```
# [link (name = "foo")]

# [link (name = "foo", kind = "bar")]
```

W obu przypadkach foo to nazwa biblioteki natywnej, z którą łączymy się, aw drugim przypadku bar to typ biblioteki natywnej, z którą łączy się kompilator. Obecnie istnieją trzy rodzaje znanych bibliotek natywnych:

Dynamics - # [link (name = "readline")]

Statics - # [link (name = "my_build_dependency", kind = "static")]

Frameworks - # [link (name = "CoreFoundation", kind = "framework")]

Pamiętaj, że frameworki są dostępne tylko w celach OSX. Różne wartości rodzaju mają na celu rozróżnienie, w jaki sposób biblioteka natywna uczestniczy w łączu. Z punktu widzenia wiązania kompilator Rust tworzy dwa rodzaje artefaktów: częściowe (rlib / staticlib) i końcowe (dylib / binary). Zależności od natywnych bibliotek i platform są propagowane do ostatecznej granicy artefaktu, podczas gdy zależności statyczne nie są w ogóle propagowane, ponieważ biblioteki statyczne są integrowane bezpośrednio z kolejnym artefaktem. Oto kilka przykładów wykorzystania tego modelu:

* Natywna zależność kompilacji: Czasami podczas pisania określonego typu kodu Rust wymagany jest klej C / C ++, ale dystrybucja kodu w formacie biblioteki jest tylko obciążeniem. W takim przypadku kod zostanie zarchiwizowany w libfoo.a, a następnie rdza skrzyni zadeklaruje zależność poprzez # [link (name = "foo", kind = "static")] .

Niezależnie od typu danych wyjściowych dla skrzynki, statyczna biblioteka natywna zostanie uwzględniona w danych wyjściowych, co oznacza, że dystrybucja natywnej biblioteki statycznej nie jest konieczna. Normalna zależność dynamiczna: wspólne biblioteki systemowe (takie jak readline) są dostępne w wielu systemach i często statyczna kopia tych bibliotek może nie istnieć. Kiedy ta zależność jest zawarta w paczce Rusta, częściowe cele (takie jak rlibs) nie będą łączyć się z biblioteką, ale kiedy rlib jest zawarte w ostatecznym celu (jak plik binarny), biblioteka zostanie połączona.

W systemie OSX frameworki zachowują się z taką samą semantyką jak biblioteki dynamiczne.

Niebezpieczne bloki

Niektóre operacje, takie jak odwoływanie się do płaskich wskaźników lub wywołania funkcji, które zostały oznaczone jako niebezpieczne, są dozwolone tylko w niebezpiecznych blokach. Niebezpieczne bloki izolują niepewność i są obietnicą dla kompilatora, że niepewność nie wycieknie z bloku. Z drugiej strony niebezpieczne funkcje reklamują niepewność światu zewnętrznemu. Niepewna funkcja jest zapisana w następujący sposób:

unsafe fn kaboom (ptr: * const i32) -> i32 {*} ptr

Ta funkcja może być wywołana tylko z niebezpiecznego bloku lub innej niebezpiecznej funkcji.

Dostęp do zewnętrznych zmiennych globalnych

Zagraniczne interfejsy API czasami eksportują zmienną globalną, która może na przykład śledzić pewien stan globalny. Aby uzyskać dostęp do tych zmiennych, musisz zadeklarować je w blokach zewnętrznych za pomocą słowa kluczowego static :

#! [feature (libc)]

extern crate libc;

[link (name = "readline")]

external

static rl_readline_version: libc :: c_int;

}

fn main () {

println! ("You have version {} of readline.",

```
rl_readline_version as i32);

}
```

Alternatywnie może być konieczna zmiana stanu globalnego zapewnianego przez obcy interfejs. Aby to zrobić, statics można zadeklarować za pomocą mut, aby je zmutować.

```
# #! [feature (libc)]

extern crate libc;

use std :: ffi :: CString;

use std :: ptr;

# [link (name = "readline")]

external

static mut rl_prompt: * const libc :: c_char;

}

fn main () {

let prompt = CString :: new ("[my-awesome-shell] $"). unwrap ();

unsafe {

rl_prompt = prompt.as_ptr ();

println! ("{:?}", rl_prompt);

rl_prompt = ptr :: null ();

}

}
```

Zauważ, że każda interakcja ze statycznym mutem jest niepewna, zarówno podczas odczytu, jak i zapisu. Radzenie sobie ze zmiennym stanem globalnym wymaga wielkiej ostrożności.

Konwencje połączeń zewnętrznych

Większość obcego kodu ujawnia ABI C, a Rust domyślnie stosuje konwencję wywoływania platformy podczas wywoływania funkcji zewnętrznych. Niektóre funkcje obce, w szczególności interfejs API systemu Windows, używają innej konwencji wywoływania. Rust zapewnia sposób informowania kompilatora o tym, której konwencji należy użyć:

```
# #! [feature (libc)]

extern crate libc;

# [cfg (all (target_os = "win32", target_arch = "x86"))]

# [link (name = "kernel32")]

# [allow (non_snake_case)]

extern "stdcall" {
```

fn SetEnvironmentVariableA (n: * const u8 , v: * const u8) -> libc ::

c_int;

}

fn main () {}

Dotyczy to całego bloku extern. Lista obsługiwanych ograniczeń ABI to:

stdcall

aapcs

cdecl

fastcall

Rust

rust-intrinsic

system

C

win64

Większość otchłani jest oczywista, ale system abi może wydawać się trochę dziwny. To ograniczenie wybiera odpowiedni ABI do współpracy z bibliotekami docelowymi. Na przykład w win32 z architekturą x86 oznacza to, że używanym abi będzie stdcall . Jednak w x86_64 system Windows używa konwencji o nazwie C , a następnie używa C . To przekłada się na to, że w naszym poprzednim przykładzie mogliśmy użyć extern "system" {...} do zdefiniowania bloku dla wszystkich systemów Windows, nie tylko x86.

Współpraca z kodem zewnętrznym

Rust zapewnia, że dystrybucja struktury jest kompatybilna z reprezentacją platformy w C tylko wtedy, gdy zastosowany jest atrybut # [repr (C)]. # [repr (C, spakowane)] może służyć do dystrybucji członków bez dopełnienia. # [repr (C)] można również zastosować do wyliczenia. Pudełka rdzy (Box <T>) używają wskaźników, które nie dopuszczają wartości null, jako uchwytów wskazujących na zawarty obiekt. Nie należy ich jednak tworzyć ręcznie, ponieważ są obsługiwane przez wewnętrznych dyspozytorów. Odwołania można bezpiecznie założyć jako bezpośrednie, niepuste wskaźniki do typu. Jednak złamanie sprawdzania pożyczania lub reguł zmienności nie gwarantuje bezpieczeństwa, dlatego w razie potrzeby preferowane jest użycie płaskich wskaźników (*), ponieważ kompilator nie może na ich podstawie założyć wielu rzeczy. Wektory i ciągi znaków mają ten sam układ w pamięci, a narzędzia do interakcji z interfejsami API języka C są dostępne w modułach vec i str. Jednak ciągi znaków nie są zakończone w \ 0 . Jeśli potrzebujesz ciągu znaków kończącego się na NUL do interakcji z C, musisz użyć typu CString w module std::ffi. Standardowa biblioteka zawiera aliasy typów i definicje funkcji dla standardowej biblioteki C w module libc, a Rust domyślnie używa libc i libm.

„Optymalizacja wskaźnika zerowego”

Niektóre typy są zdefiniowane jako nie null . Obejmuje to odwołania (& T , & mut T), pola (Box <T>) i wskaźniki funkcji (extern "abi" fn ()). Kiedy łączysz się z C, czasami używane są wskaźniki, które mogą być puste. W szczególnym przypadku ogólne wyliczenie zawierające dokładnie dwa warianty, z których

jeden nie zawiera danych, a drugi zawiera jedno pole, kwalifikuje się do „optymalizacji wskaźnika zerowego”. Gdy wspomniane wyliczenie jest tworzone za pomocą jednego z typów niedopuszczających wartości null, jest reprezentowane jako pojedynczy wskaźnik, a wariant bez danych jest reprezentowany jako wskaźnik zerowy. Tak więc opcja <extern "C" fn (c_int) -> c_int> to sposób, w jaki reprezentuje się wskaźnik funkcji zerowej za pomocą ABI z C.

Wywoływanie kodowania Rusta z C

Możesz chcieć skompilować kod Rusta, aby można go było wywoływać z języka C. Jest to łatwe, ale wymaga pewnych rzeczy:

```
# [no_mangle]

pub extern fn hello_rust () -> * const u8 {

"Hello world! \ 0" .as_ptr ()

}

# fn main () {}
```

Extern sprawia, że ta funkcja jest zgodna z konwencją wywoływania C, jak omówiono w „ Konwencjach wywoływania obcego ”. Atrybut no_mangle wyłącza zniekształcanie nazw Rusta, ułatwiając wiązanie.

FFI i panika

Ważne jest, aby być świadomym paniki! os podczas pracy z FFI. Panika! na granicy FFI jest niezdefiniowane zachowanie. Jeśli piszesz kod, który może wywołać panikę, musisz uruchomić go w innym wątku, aby panika nie rozprzestrzeniła się na C:

```
use std :: thread;

# [no_mangle]

pub extern fn oh_no () -> i32 {

let h = thread :: spawn (| | {

panic! ( "Oops!" );

});

match h.join () {

Ok (_) => 1 ,

Err (_) => 0 ,

}

}

# fn main () {}
```

Borrow i AsRef

Utracone cechy Borrow i AsRef są bardzo podobne, ale różne. Oto krótki przegląd znaczenia tych cech:

Borrow

Cecha Borrow jest używana, gdy piszemy strukturę danych i z jakiegoś powodu chcemy użyć typu posiadanego lub pożyczonego jako synonimu. Na przykład HashMap ma metodę get, która używa Borrow :

fn get (& self, k: & Q) -> Option <& V>

where K: Borrow,

Q: Hash + Eq

Ten podpis jest skomplikowany. Teraz interesuje nas parametr K. Odnosi się do parametru samej mapy HashMap:

struct HashMap {

Parametr K to typ klucza używanego przez HashMap. Patrząc ponownie na sygnaturę get() , możemy użyć get() tylko wtedy, gdy klucz implementuje Borrow <Q> . W ten sposób możemy stworzyć HashMap, która używa kluczy String, ale podczas wyszukiwania use & str s:

use std :: collections :: HashMap;

let mut map = HashMap :: new ();

map.insert ("Foo" .to_string (), 42);

assert_eq! (map.get ("Foo"), Some (& 42));

Dzieje się tak, ponieważ standardowa biblioteka ma Borrow <str> dla String impl . W przypadku większości typów, gdy chcesz wybrać typ posiadany lub pożyczony, wystarczy & T. Jednak jednym z obszarów, w których pożyczanie jest skuteczne, jest sytuacja, w której występuje więcej niż jeden rodzaj pożyczonych papierów wartościowych. Jest to szczególnie prawdziwe w przypadku referencji i wycinków: możesz mieć zarówno & T one, jak i mut T . Jeśli chcemy zaakceptować oba rodzaje, pożyczka jest tym, czego potrzebujemy:

use std :: borrow :: Borrow;

use std :: fmt :: Display;

fn foo <T: Borrow < i32 > + Display> (a: T) {

println! ("a is a borrowed: {}" , a);

}

let mut i = 5 ;

foo (& i);

foo (& mut i);

Spowoduje to wydrukowanie dwukrotnie pożyczonej wartości: 5.

AsRef

Cecha AsRef jest cechą konwersji. Służy do konwersji pewnej wartości na odwołanie w kodzie ogólnym. Lubię to:

let s = "Hello" .to_string ();

```
fn foo <T: AsRef < str >> (s: T) {

let slice = s.as_ref ();

}
```

Którego powinieneś użyć?

Widzimy, że oba są do siebie podobne: oba mają do czynienia z pewnego rodzaju wersjami posiadanymi i pożyczonymi. Jednak są różne. Wybierz opcję Wypożycz, jeśli chcesz abstrahować różne typy wypożyczeń lub gdy budujesz strukturę danych, która traktuje posiadane i pożyczone wartości w równoważny sposób, na przykład mieszając i porównując. Użyj AsRef, gdy chcesz przekonwertować coś bezpośrednio na referencję i piszesz ogólny kod.

Kanały dystrybucji

Projekt Rust wykorzystuje koncepcję zwaną „kanałami dystrybucji" do zarządzania wydaniami. Ważne jest, aby zrozumieć ten proces, abyś mógł zdecydować, której wersji Rust powinien używać twój projekt.

Przegląd

Istnieją trzy kanały dla wydań Rust:

Nocturnal (nightly)

Beta

Stable

Wydania nocne są tworzone raz dziennie. Co sześć tygodni najnowsza nocna wersja jest promowana do wersji „Beta". W tym momencie będziesz otrzymywać tylko łatki, które naprawiają poważne błędy. Sześć tygodni później beta jest promowana do „Stabilnej" i staje się nową wersją 1.x. Proces ten przebiega równolegle. Co sześć tygodni, tego samego dnia, w którym nokturn osiąga poziom beta, beta zostaje awansowana na stabilną. Kiedy 1.x zostaje wydany, w tym samym czasie zostaje wydany 1. (x + 1) -beta, a noc staje się pierwszą wersją 1. (x + 2) -nightly .

Wybór wersji

Ogólnie rzecz biorąc, o ile nie masz konkretnego powodu, powinieneś korzystać ze stabilnego kanału dystrybucji. Te wydania są przeznaczone dla szerokiego grona odbiorców. Jednak w zależności od twojego zainteresowania Rust, możesz wybrać noc. Bilans jest następujący: na kanale nocnym można skorzystać z nowych funkcji Rusta. Jednak niestabilne funkcje mogą ulec zmianie, dlatego każdy klub nocny ma możliwość złamania kodu. Jeśli korzystasz ze stabilnej wersji, nie masz dostępu do funkcji eksperymentalnych, ale następna wersja Rust nie będzie sprawiać większych problemów ze względu na zmiany.

Pomaganie ekosystemowi poprzez CI

A co z betą? Polecamy wszystkim użytkownikom Rusta korzystającym ze stajni kanału dystrybucji testuje również swoje systemy ciągłej integracji kanału beta. Pomoże to ostrzec zespół w przypadku przypadkowej regresji. Ponadto testowanie w trybie nocnym może wykryć regresje znacznie wcześniej, dlatego jeśli nie masz nic przeciwko posiadaniu trzeciej kompilacji, doceniamy możliwość przetestowania we wszystkich kanałach.

Składnia i semantyka

Ta sekcja dzieli Rusta na małe części, po jednej dla każdej koncepcji. Jeśli chcesz nauczyć się Rusta od podstaw, przeczytanie tej sekcji po kolei jest bardzo dobrym sposobem, aby to zrobić. Te sekcje stanowią odniesienie do każdej koncepcji, jeśli czytasz inny samouczek i znajdziesz coś niejasnego, możesz znaleźć wyjaśnienie gdzieś w tej sekcji.

Linki zmienne

Praktycznie każdy program inny niż „Hello World" używa linków zmiennych. Te linki do zmiennych wyglądają tak:

```
fn main () {

let x = 5 ;

}
```

Umieszczanie fn main () { w każdym przykładzie jest trochę uciążliwe, więc w przyszłości będziemy je pomijać. Jeśli postępujesz krok po kroku, pamiętaj o edycji funkcji main() zamiast jej pomijania. W przeciwnym razie pojawi się błąd. W wielu językach nazywa się to zmienną , ale powiązania zmiennych w Rust mają kilka asów w rękawie. Na przykład lewa strona wyrażenia let to „wzorzec", a nie prosta nazwa zmiennej. Oznacza to, że możemy robić takie rzeczy jak:

```
let (x, y) = ( 1 , 2 );
```

Po obliczeniu tego wyrażenia x będzie równe jeden, a y będzie równe dwa. Wzory są naprawdę potężne i mają własną sekcję w książce. Na razie nie potrzebujemy tych udogodnień, po prostu pamiętajmy o tym na bieżąco. Rust jest językiem o typach statycznych, co oznacza, że z góry określamy nasze typy i są one sprawdzane w czasie kompilacji. Dlaczego więc nasz pierwszy przykład się kompiluje? Cóż, Rust ma coś takiego jak „wnioskowanie o typie". Jeśli możesz określić typ czegoś, Rust nie wymaga wpisywania typu. Możemy dodać typ, jeśli chcemy. Typy występują po dwukropku (:) :

```
let x: i32 = 5 ;
```

Gdybym poprosił resztę klasy o przeczytanie tego na głos, powiedzielibyście: „x jest łączem do typu i32 i wartością pięć". W tym przypadku zdecydowaliśmy się przedstawić x jako 32-bitową liczbę całkowitą ze znakiem. Rust ma wiele różnych typów pierwotnych liczb całkowitych. Zaczynają się od i dla liczb całkowitych ze znakiem i u dla liczb całkowitych bez znaku. Możliwe rozmiary liczb całkowitych to 8, 16, 32 i 64 bity. W przyszłych przykładach moglibyśmy opisać typ w komentarzu. Przykład wyglądałby tak:

```
fn main () {

let x = 5 ; // x: i32

}
```

Zwróć uwagę na podobieństwa między adnotacją a składnią, której używasz z let. Włączenie tego rodzaju komentarza nie jest idiomatyczne w Ruście, ale będziemy go od czasu do czasu umieszczać, aby pomóc Ci zrozumieć, jakie typy wnioskuje Rust. Domyślnie łącza są niezmienne. Ten kod nie skompiluje się:

```
let x = 5;
```

x = 10;

Spowoduje to następujący błąd:

error: re-assignment of immutable variable `x`

x = 10;

^ ~~~~~~

Jeśli chcesz, aby łącze zmienne było zmienne, możesz użyć mut :

letniech mut x = 5 ; // mut x: i32

x = 10;

Nie ma jednego powodu, dla którego powiązania zmiennych są domyślnie niezmienne, ale możemy o tym myśleć przez pryzmat jednego z głównych celów Rusta: bezpieczeństwa. Jeśli zapomnisz powiedzieć mut . kompilator to zauważy i poinformuje cię, że zmutowałeś coś, czego nie zamierzałeś mutować. Gdyby łącza do zmiennych były domyślnie zmienne, kompilator nie byłby w stanie ci tego powiedzieć. Jeśli miałeś zamiar coś zmutować, rozwiązanie jest bardzo proste: dodaj mut . Istnieją inne dobre powody, aby w miarę możliwości unikać stanu mutable, ale wykraczają one poza zakres tego przewodnika. Ogólnie rzecz biorąc, często można uniknąć jawnej mutacji, co jest preferowane w Rust. To powiedziawszy, czasami mutacja jest właśnie tym, czego potrzebujesz, dlatego nie jest zabroniona. Wróćmy do linków do zmiennych. Dowiązania zmienne w Rust mają jeszcze jeden aspekt, który różni się od innych języków: zanim będzie można ich użyć, wymagana jest inicjalizacja do wartości. Przetestujmy to. Zmień swój plik src / main.rs, aby wyglądał tak:

fn main () {

let x: i32 ;

println! ("Hello, world!");

}

Możesz użyć kompilacji ładunku w wierszu poleceń, aby go skompilować. Otrzymasz ostrzeżenie, ale nadal wyświetli się „Witaj, świecie!”:

Compiling hello_world v0.0.1 (file: /// home / you / projects /

hello_world)

src / main.rs: 2: 9: 2:10 warning: unused variable: `x`, # [warn

(unused_variable)]

on by default

src / main.rs: 2 let x: i32;

^

Rust ostrzega nas, że nigdy nie używamy łącza zmiennego, ale ponieważ nigdy go nie używamy, nie ma niebezpieczeństwa ani winy. Jednak sytuacja się zmienia, jeśli faktycznie spróbujemy użyć x . Zróbmy to. Zmień swój program, aby wyglądał tak:

fn main () {

```
let x: i32;

println! ("The value of x is: {}", x);

}
```

Spróbuj to skompilować. Otrzymasz błąd:

```
$ build charge

Compiling hello_world v0. 0.1 (file: /// home / you / projects /

hello_world)

src / main.rs: 4 : 39 : 4 : 40 error: use of possibly uninitialized variable:

`x`

src / main.rs: 4 println! ( "The value of x is: {}" , x);

^

note: in expansion of format_args!

<std macros>: 2 : 23 : 2 : 77 note: expansion site

<std macros>: 1 : 1 : 3 : 2 note: in expansion of println!

src / main.rs: 4 : 5 : 4 : 42 note: expansion site

error: aborting due to previous error

Could not compile `hello_world`.
```

Rust nie pozwoli ci użyć wartości, która nie została wcześniej zainicjowana. Porozmawiajmy o tym, co dodaliśmy do println poniżej! . Jeśli dołączysz parę kluczy ({} , niektórzy nazywają je wąsami / wąsami ...) w ciągu do wydrukowania, Rust zinterpretuje to jako żądanie interpolacji jakiejś wartości. Interpolacja ciągów to termin w informatyce, który oznacza „umieść to w ciągu". Dodajemy przecinek, a następnie x , aby wskazać, że chcemy, aby była to wartość interpolowana. Przecinek służy do oddzielenia argumentów, które przekazujemy do funkcji i makr, w przypadku przekazywania więcej niż jeden. Kiedy użyjesz nawiasów klamrowych, Rust spróbuje wyświetlić wartość w sensowny sposób po sprawdzeniu jej typu. Jeśli chcemy określić bardziej szczegółowy format, dostępnych jest wiele opcji . Na razie będziemy trzymać się domyślnego zachowania: liczby całkowite nie są bardzo trudne do wydrukowania.

Cechy

Każdy program w Rust ma co najmniej jedną funkcję, główną funkcję:

```
fn main() {

}
```

Jest to najprostsza możliwa deklaracja funkcji. Jak wspomnieliśmy wcześniej, fn mówi „to jest funkcja", po której następuje nazwa, nawiasy, ponieważ ta funkcja nie otrzymuje żadnych argumentów, a następnie nawiasy klamrowe, aby wskazać treść funkcji. Oto funkcja o nazwie foo :

```
fn foo () {
```

}

A co z przyjmowaniem argumentów? Oto funkcja, która drukuje liczbę:

```
fn print_number (x: i32 ) {

drukuj! ("x to: {}" , x);

}
```

Oto kompletny program, który korzysta z print_number :

```
fn print_number (x: i32 ) {

println! ( "x is: {}" , x);

}
```

Jak zobaczysz, argumenty funkcji działają bardzo podobnie do instrukcji let : do nazwy argumentu dodajesz typ po dwukropku. Oto kompletny program, który dodaje dwie liczby, a następnie je wyświetla:

```
fn main () {

print_sum ( 5 , 6 );

}

fn print_sum (x: i32 , y: i32 ) {

println! ( "the sum is: {}" , x + y);

}
```

Argumenty są oddzielone przecinkiem, zarówno w przypadku wywołania funkcji, jak i jej zadeklarowania. W przeciwieństwie do let , musisz zadeklarować typy argumentów. Następujące elementy nie zostaną skompilowane:

```
fn print_sum (x, y) {

println! ("sum is: {}", x + y);

}
```

Otrzymasz następujący błąd:

```
expected one of `!`, `:`, or `@`, found `)`

fn print_sum (x, y) {
```

To przemyślana decyzja projektowa. Chociaż możliwe jest pełne wnioskowanie programu, języki, które go posiadają, takie jak Haskell, często sugerują, że jawne dokumentowanie typów jest dobrą praktyką. Zgadzamy się, że zmuszanie funkcji do deklarowania typów przy jednoczesnym umożliwieniu wnioskowania w ramach funkcji jest wspaniałym kompromisem między całkowitym wnioskowaniem a brakiem wnioskowania. A co ze zwróceniem wartości? Oto funkcja, która dodaje jedynkę do liczby całkowitej:

```
fn sum_one (x: i32 ) -> i32 {
```

```
x + 1

}
```

Funkcje w Rust zwracają dokładnie jedną wartość, a typ jest deklarowany po „strzałce”, która jest łącznikiem (-), po którym następuje znak większości (>). Ostatni wiersz funkcji określa, co zwraca. Zauważysz tutaj brak średnika. Dodaj to:

```
fn sum_one (x: i32) -> i32 {

x + 1;

}
```

Otrzymalibyśmy błąd:

```
error: not all control paths return a value

fn sum_one (x: i32) -> i32 {

x + 1;

}

help: consider removing this semicolon:

x + 1;

^
```

Powyższe ujawnia dwie interesujące rzeczy na temat Rust: jest to język oparty na wyrażeniach, a średniki różnią się od innych języków opartych na „nawiasach klamrowych i średnikach”. Te dwie rzeczy są ze sobą powiązane.

Wyrażenia a zdania

Rdza w języku opartym głównie na wyrażeniach. Istnieją tylko dwa rodzaje instrukcji, wszystko inne jest wyrażeniem. Jaka jest różnica? Wyrażenia zwracają wartość, a instrukcje nie. Dlatego kończymy z błędem „nie wszystkie ścieżki kontrolne zwracają wartość” tutaj: instrukcja x + 1; nie zwraca wartości. Istnieją dwa rodzaje instrukcji w Rust: „instrukcje instrukcji” i „instrukcje wyrażeń”. Wszystko inne jest wyrazem. Porozmawiajmy najpierw o deklaracjach. W niektórych językach łącza zmienne mogą być zapisywane jako wyrażenia, a nie tylko instrukcje. Jak w rubinie:

```
x = y = 5
```

Jednak w Ruście użycie let w celu wprowadzenia łącza do zmiennej nie jest wyrażeniem. Poniższe spowoduje błąd w czasie kompilacji:

```
let x = (let y = 5); // oczekiwany identyfikator, znalezione słowo kluczowe `let`
```

Kompilator mówi nam, że czekał na początek wyrażenia, a let może być tylko instrukcją, a nie wyrażeniem. Zauważ, że przypisanie zmiennej, która została wcześniej przypisana (np. y = 5) jest wyrażeniem, ale jej wartość nie jest szczególnie przydatna. W przeciwieństwie do innych języków, w których przypisanie jest oceniane na przypisaną wartość (np. 5 w poprzednim przykładzie), w Rust wartość przypisania jest pustą krotką (), ponieważ przypisana wartość może mieć jednego właściciela, a dowolna wartość zwrócona przez Another być zaskakującym:

```
let mut y = 5 ;

let x = (y = 6 ); // x has the value `()`, not `6`
```

Drugim typem instrukcji w Rust jest instrukcja wyrażenia. Jego celem jest przekształcenie dowolnego wyrażenia w zdanie. W praktyce gramatyka Rusta oczekuje, że po zdaniu będzie następować więcej zdań. Oznacza to, że możesz używać średników do oddzielania jednego wyrażenia od drugiego. To powoduje, że Rust wygląda bardzo podobnie do tych języków, w których średnik jest wymagany na końcu każdej linii, w rzeczywistości zobaczysz średniki na końcu prawie każdej linii Rusta, którą zobaczysz. Jaki więc wyjątek sprawia, że mówimy „prawie”? Widzieliście to już w powyższym kodzie:

```
fn sum_one (x: i32 ) -> i32 {

x + 1

}
```

Nasza funkcja twierdzi, że zwraca i32 , ale ze średnikiem zwróciłaby () . Rust stwierdza, że prawdopodobnie nie jest to to, czego chcemy, i sugeruje usunięcie średnika z błędu, który widzieliśmy wcześniej.

Wczesne powroty

Ale co z wcześniejszymi powrotami? Rust zapewnia na to zarezerwowane słowo, return :

```
return :

fn foo (x: i32 ) -> i32 {

return x;

// we will never reach this code!

x + 1

}
```

Używanie zwrotu jako ostatniego wiersza funkcji jest poprawne, ale jest uważane za zły styl:

```
fn foo (x: i32 ) -> i32 {

return x + 1 ;

}
```

Poprzednia definicja bez zwrotu może wyglądać nieco dziwnie, jeśli nigdy nie pracowałeś z językiem opartym na wyrażeniach, ale ta z czasem staje się intuicyjna.

Funkcje rozbieżne

Rust ma specjalną składnię dla „funkcji rozbieżnych”, które są funkcjami niepowracającymi:

```
fn diverges () ->! {

panic! ( "This function never returns!" );

}
```

panika! jest to makro podobne do println! (), które już widzieliśmy. W przeciwieństwie do println! () , Panika! () Powoduje nagłe zakończenie bieżącego wątku i wyświetlenie podanego komunikatu. Ponieważ ta funkcja spowoduje nagłe wyjście, nigdy nie wróci, dlatego ma typ ' ! ', co brzmi:' rozbieżne'. Funkcji rozbieżnej można użyć jako dowolnego typu:

```
# fn diverges () ->! {

# panic! ("This function never returns!");

#}

let x: i32 = diverge ();

let x: String = diverge ();
```

Wskaźniki funkcji

Możemy również tworzyć łącza do zmiennych wskazujących na funkcje:

```
let f: fn (i32) -> i32;
```

f jest powiązaniem ze zmienną wskazującą na funkcję, która przyjmuje i32 jako argument i zwraca i32. Na przykład:

```
fn mas_uno (i: i32 ) -> i32 {

i + 1

}

// no type inference

let f: fn ( i32 ) -> i32 = mas_uno;

// type inference

let f = more_one;
```

We can then use f to call the function:

```
# fn mas_uno (i: i32 ) -> i32 {i + 1 }

# let f = more_one;

let six = f ( 5 );
```

Typy pierwotne

Rust ma zestaw typów, które są uważane za „prymitywne”. Oznacza to, że są one zintegrowane z językiem. Rust jest skonstruowany w taki sposób, że standardowa biblioteka zapewnia również szereg użytecznych typów opartych na prymitywach, ale te są najbardziej prymitywne.

Boolean

Rust ma wbudowany typ logiczny o nazwie bool . Ma dwie możliwe wartości true i false:

```
let x = true ;

let y: bool = fase ;
```

Typowym zastosowaniem logicznych jest if warunkowe . Więcej dokumentacji bool eanos można znaleźć w dokumentacji standardowej biblioteki (angielski).

char

Typ char reprezentuje pojedynczą wartość skalarną Unicode. Możesz tworzyć char s z pojedynczymi cudzysłowami: (')

let x = 'x';

let dos_corazones = ‘**’;

W przeciwieństwie do innych języków oznacza to, że char w Rust nie jest pojedynczym bajtem, ale cztery. Więcej dokumentacji dotyczącej znaków można znaleźć w dokumentacji biblioteki standardowej.

Typy liczbowe

Rust ma różne typy liczb w kilku kategoriach: ze znakiem i bez znaku, stałe i zmienne, zmiennoprzecinkowe i liczby całkowite. Te typy składają się z dwóch części: kategorii i rozmiaru. Na przykład u16

jest typem bez znaku o rozmiarze szesnastu bitów. Więcej bitów pozwala na przechowywanie większych liczb. Jeśli literał liczbowy nie ma niczego, co spowodowałoby wywnioskowanie jego typu, używane są typy domyślne:

let x = 42 ; // x has type i32

let y = 1.0 ; // and it has type f64

Oto lista różnych typów numerycznych wraz z linkami do ich dokumentacji w bibliotece standardowej (w języku angielskim):

i8

i16

i32

i64

u8

u16

u32

u64

isize

usize

f32

f64

Przyjrzyjmy się każdej z różnych kategorii.

Ze znakiem i bez znaku

Typy całkowite występują w dwóch odmianach: ze znakiem i bez znaku. Aby zrozumieć różnicę, rozważ liczbę o rozmiarze czterech bitów. Czterobitowa liczba ze znakiem pozwoliłaby przechowywać liczby od -8 do +7 . Liczby ze znakiem używają „reprezentacji uzupełnienia do dwóch". Liczba czterech bitów bez znaku, ponieważ nie trzeba zapisywać wartości ujemnych, może przechowywać wartości od 0 do +15 . Typy bez znaku używają u dla swojej kategorii, a typy ze znakiem używają i . I oznacza liczbę całkowitą (liczbę całkowitą). Zatem u8 to ośmiobitowa liczba bez znaku, a i8 to ośmiobitowa liczba ze znakiem.

Stałe typy rozmiarów

Typy o stałym rozmiarze mają określoną liczbę bitów w swojej reprezentacji. Prawidłowe rozmiary to 8 , 16 , 32 i 64 . Zatem u32 jest 32-bitową liczbą całkowitą bez znaku, a i64 jest 64-bitową liczbą całkowitą ze znakiem.

Typy o zmiennym rozmiarze

Rust zapewnia również typy, których rozmiar zależy od rozmiaru wskaźnika na maszynie bazowej. Te typy mają kategorię „rozmiar" i występują zarówno w wersji ze znakiem, jak i bez znaku. Powoduje to dwa typy isize i use .

Typy zmiennoprzecinkowe

Rust ma również dwa rodzaje zmiennoprzecinkowych: f32 i f64 . Odpowiadają one numerom pojedynczej i podwójnej precyzji IEEE-754.

Ustalenia

Podobnie jak wiele języków programowania, Rust ma typy list reprezentujące sekwencję rzeczy. Najbardziej podstawowym jest układ , lista elementów tego samego typu i o ustalonej wielkości. Domyślnie ustalenia są niezmienne.

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

let mut m = [ 1 , 2 , 3 ]; // m: [i32; 3]
```

Tablice mają typ [T; N] . Porozmawiamy o notacji T w części ogólnej. N jest stałą w czasie kompilacji dla długości aranżacji. Istnieje skrót do inicjalizacji każdego z elementów tablicy do tej samej wartości. W tym przykładzie każdy element a zostanie zainicjowany na 0 :

```
let a = [ 0 ; 20]; // a: [i32; 20]
```

Możesz uzyskać liczbę elementów tablicy a za pomocą a.len ()

```
let a = [ 1 , 2 , 3 ];

printl! ("a ma {} elementów" , a.len());
```

Możesz uzyskać dostęp do określonego elementu tablicy za pomocą notacji indeksu dolnego:

```
let a = [ 1 , 2 , 3 ];

println! ( "a has {} elements" , a.len ());
```

Podindeksy zaczynają się od zera, jak w większości języków programowania, więc pierwsze imię to nazwy [0], a drugie imię to nazwy [1] . Powyższy przykład drukuje: Drugie imię to: Brian . Jeśli

spróbujesz użyć indeksu dolnego, którego nie ma w tablicy, pojawi się błąd: sprawdzanie ograniczeń dostępu do tablicy odbywa się w czasie wykonywania. Taki błądzący dostęp jest źródłem wielu błędów w systemowych językach programowania. Więcej dokumentacji dotyczącej tablic s można znaleźć w dokumentacji biblioteki standardowej.

Plastry

Wycinek jest odniesieniem do (lub „widokiem" w obrębie) innej struktury danych. Plasterki są przydatne do umożliwienia bezpiecznego i wydajnego dostępu do części tablicy bez konieczności kopiowania. Na przykład możesz chcieć odwołać się do pojedynczej linii w pliku, który został wcześniej odczytany z pamięci. Z natury wycinki nie są tworzone bezpośrednio, są tworzone dla zmiennej, która już istnieje. Plasterki mają długość, mogą być zmienne lub niezmienne i pod wieloma względami zachowują się jak tablice:

niech za = [0 , 1 , 2 , 3 , 4];

niech środek = & a [1 .. 4]; // Kawałek a: tylko elementy 1, 2 i 3

niech zakończy się = & a [..]; // Plasterek zawierający wszystkie elementy a.

Plasterki mają typ & [T] . Porozmawiamy o T, gdy omówimy leki generyczne. Więcej dokumentacji plasterków można znaleźć w dokumentacji standardowej biblioteki (angielski)

str

Strumień Rusta jest najbardziej prymitywnym typem łańcucha. Jako typ bez rozmiaru i sam w sobie nie jest zbyt użyteczny, ale staje się bardzo przydatny, gdy zostanie umieszczony za referencją, na przykład & str . Dlatego zostawimy to tutaj. Więcej dokumentacji dla str można znaleźć w dokumentacji standardowej biblioteki (angielski)

Krotki

Krotka to uporządkowana lista o stałym rozmiarze. Lubię to:

let x = (1 , "hello");

Nawiasy i przecinki tworzą tę krotkę o długości dwa. Oto ten sam kod, ale z adnotacjami typu:

let x: (i32 , & str) = (1 , "hello");

Jak widać, typ krotki jest taki sam jak krotka, ale każda pozycja ma typ zamiast wartości. Uważni czytelnicy zauważą również, że krotki są heterogeniczne: mamy w tej krotce i32 ya & str. W systemowych językach programowania ciągi znaków są nieco bardziej złożone niż w innych językach. Na razie po prostu przeczytaj & str jako wycinek ciągu znaków. Wkrótce dowiemy się więcej. Możesz przypisać jedną krotkę do drugiej, jeśli mają te same zawarte typy i tę samą arity . Krotki mają tę samą arność, gdy są tego samego rozmiaru.

let mut x = (1 , 2); // x: (i32, i32)

let y = (2 , 3); // y: (i32, i32)

x = y;

Możesz uzyskać dostęp do pól krotki za pomocą nieustrukturyzowanego let . Oto przykład:

let (x, y, z) = (1 , 2 , 3);

```
println! ( "x is {}" , x);
```

Pamiętasz, jak wcześniej powiedziałem, że lewa strona instrukcji let jest potężniejsza niż zwykłe przypisanie powiązania? Oto jesteśmy. Możemy umieścić wzór po lewej stronie let , a jeśli pasuje do prawej strony, możemy przypisać wiele powiązań jednocześnie do zmiennej. W tym przypadku let „dekonstruuje” lub „podziela” krotkę i przypisuje części do trzech powiązań do zmiennej. Ten wzorzec jest bardzo potężny i będziemy go często powtarzać w przyszłości. Możesz usunąć niejednoznaczność między jednoelementową krotką a wartością ujętą w nawiasy za pomocą przecinka:

```
( 0 ,); // single element tuple

( 0 ); // zero enclosed in parentheses
```

Indeksowane w krotkach

Możesz także uzyskać dostęp do pól krotki za pomocą składni indeksowania:

```
let tuple = ( 1 , 2 , 3 );

let x = tuple. 0 ;

let y = tuple. 1 ;

let z = tuple. 2 ;

println! ( "x is {}" , x);
```

Podobnie jak indeksowanie tablicowe, zaczyna się od zera, ale w przeciwieństwie do indeksowania tablicowego są one używane. , zamiast [] s. Więcej dokumentacji dotyczącej krotek można znaleźć w dokumentacji biblioteki standardowej (w języku angielskim)

Cechy

Funkcje też mają typ! Wyglądają tak:

```
fn foo (x: i32 ) -> i32 {x}

let x: fn ( i32 ) -> i32 = foo;
```

W tym przypadku x jest „wskaźnikiem” do funkcji, która otrzymuje i32 i zwraca i32 .

Komentarze

Teraz, gdy widzieliśmy już niektóre funkcje, warto zapoznać się z komentarzami. Komentarze to notatki, które zostawiasz innym programistom w celu wyjaśnienia niektórych aspektów twojego kodu. Kompilator przeważnie je ignoruje. Rust ma dwa rodzaje komentarzy, które powinny Cię zainteresować: komentarze do linii i komentarze do dokumentacji

```
// Komentarze wiersza to wszystko po znaku „//” i rozciągają się do końca wiersza

let  x = 5; // to jest również komentarz do wiersza

// Jeśli masz długie wyjaśnienie czegoś, możesz dodać komentarz

// wstaw kilka razem. Umieść spację między // a swoim komentarzem za pomocą

// aby były bardziej czytelne.
```

Innym typem komentarza jest komentarz do dokumentacji (lub komentarz do dokumentu). Komentarze do dokumentu używają /// zamiast // i obsługują notację Markdown wewnątrz:

/// Dodaj jeden do podanej liczby

///

/// # Przykłady

///

///

/// let five = 5; /// /// assert_eq! (6, sum_one (5)); /// # fn sum_one (x: i32) -

> i32 {/// # x + 1 /// #} /// `` `fn sum_one (x: i32) -> i32 {x + 1}

Istnieje inny styl komentarzy, `//!`, służący do komentowania elementów kontenera (np. skrzynek, modułów lub funkcji), zamiast elementów, które po nich następują. Są powszechnie używane w katalogu głównym skrzynki (lib.rb) lub w katalogu głównym modułu (mod.rs):

//! # Standardowa biblioteka Rusta //! //! Biblioteka standardowa Rust zapewnia funkcjonalność //! środowisko uruchomieniowe niezbędne do tworzenia oprogramowania //! Rdzawa przenośność. `` ''

Przy pisaniu komentarzy do dokumentów podanie kilku przykładów użycia jest bardzo, bardzo przydatne. Zauważysz, że wykorzystaliśmy makro: assert_eq! . Porównuje dwie wartości i panikuje! lub jeśli to nie to samo. Jest to bardzo przydatne w dokumentacji. Jest też inne makro, potwierdzaj! , co panikuje! lub jeśli podana wartość to false . Możesz użyć narzędzia rustdoc do wygenerowania dokumentacji HTML z tych komentarzy, a także uruchomić kod z przykładów jako testy!

If

Podejście Rusta do if nie jest szczególnie skomplikowane, ale jest bardziej podobne do if, które można znaleźć w językach z dynamicznym typowaniem niż w tradycyjnych językach systemowych. Porozmawiajmy o tym trochę, aby upewnić się, że rozumiesz wszystkie niuanse. if jest specyficzną formą bardziej ogólnego pojęcia, „gałąź”. Nazwa pochodzi od gałęzi w drzewie: jest to punkt decyzyjny, w którym w zależności od opcji można wybrać kilka ścieżek. W przypadku if istnieje jeden wybór, który prowadzi do dwóch ścieżek:

let x = 5;

if x == 5 {

println! ("x is five!");

}

Gdyby wartość x zmieniła się na inną, ten wiersz nie zostałby wydrukowany. Mówiąc dokładniej, jeśli wyrażenie po if zostanie ocenione na true , wówczas wykonywany jest blok kodu. Jeśli false , ten blok nie jest wywoływany. Jeśli chcesz, aby coś się stało w przypadku false , musisz użyć else :

let x = 5;

if x == 5 {

println! ("x is five!");

```
} else {
println! ("x is not five :(");
}
```

Jeśli jest więcej niż jeden przypadek, użyj innego, jeśli:

```
let x = 5;
if x == 5 {
println! ("x is five!");
} else if x == 6 {
println! ("x is six!");
} else {
println! ("x is neither five nor six :(");
}
```

To wszystko jest bardzo standardowe. Możesz jednak również to zrobić:

```
let x = 5;
let y = if x == 5 {
10
} else {
fifteen
}; // y: i32
```

Które możemy (i prawdopodobnie powinniśmy) zapisać tak:

```
let x = 5;
let y = if x == 5 {10} else {15}; // y: i32
```

To działa, ponieważ if jest wyrażeniem. Wartość wyrażenia jest wartością ostatniego wyrażenia w wybranym bloku. If bez else zawsze daje w wyniku () jako wartość.

Cykle

Rust obecnie zapewnia trzy podejścia do wykonywania czynności iteracyjnych. pętla, while i for. Każde z tych podejść ma swoje zastosowania.

loop

Cykl pętli nieskończonej jest najprostszym cyklem dostępnym w Rust. Używając słowa kluczowego loop , Rust umożliwia iterację w nieskończoność, aż do osiągnięcia jakiejś instrukcji zakończenia. Nieskończona pętla Rust wygląda tak:

```
loop {
```

```
println! ("Itera forever!");
}
```

while

Rust ma również pętlę while. To wygląda tak:

```
let mut x = 5 ; // mut x: i32
let mut completed = false ; // mut completed: bool
while ! completed {
x + = x - 3 ;
println! ( "{}" , x);
if x% 5 == 0 {
completed = true ;
}
}
```

Pętle while są właściwym wyborem, gdy nie jesteś pewien, ile razy musisz wykonać iterację. Jeśli potrzebujesz nieskończonej pętli, możesz pokusić się o napisanie czegoś takiego:

```
while true {
```

Jednak pętla jest zdecydowanie najlepsza w tym przypadku:

```
loop {
```

Analiza przepływu sterowania w Rust traktuje tę konstrukcję inaczej niż a while true , ponieważ wiemy, że będziemy iterować w nieskończoność. Ogólnie rzecz biorąc, im więcej informacji dostarczamy kompilatorowi, tym kompilator może działać lepiej w odniesieniu do bezpieczeństwa i generowania kodu, dlatego zawsze powinieneś preferować pętlę, gdy planujesz iterację w nieskończoność

For

Pętla for służy do iteracji określoną liczbę razy. Cykle dla Rusta działają jednak inaczej niż systemy innych języków programowania. Dla Rusta nie wygląda to tak jak ten cykl dla „stylu C”

```
for (x = 0 ; x < 10 ; x ++) {
printf("% d\n" , x);
}
```

Zamiast tego pętla for Rusta wygląda tak:

```
for  x in 0 .. 10 {
println! ( "{}" , x); // x: i32
}
```

W nieco bardziej abstrakcyjnych kategoriach:

```
for var in expression {

code

}
```

Wyrażenie jest iteratorem. Iterator zwraca serię elementów. Każdy element jest iteracją cyklu. Ta wartość jest z kolei przypisana do nazwy var , która obowiązuje w ciele cyklu. Po zakończeniu cyklu następna wartość jest pobierana z iteratora i jest powtarzana jeszcze raz. Gdy nie ma więcej wartości, pętla for kończy się. W naszym przykładzie 0..10 jest wyrażeniem, które przyjmuje pozycję początkową i końcową i zwraca iterator powyżej tych wartości. Górna granica jest wyłączna, więc nasza pętla wypisze od 0 do 9 , a nie 10 . Nawiasem mówiąc, Ruth nie posiada pętli for w stylu C. Ręczne sterowanie każdym elementem cyklu jest skomplikowane i podatne na błędy, nawet dla doświadczonych programistów C.

Wymień się

Gdy chcesz śledzić, ile razy wykonałeś iterację, możesz użyć funkcji .enumerate() .

W zakresach:

```
for (i, j) in ( 5 .. 10 ) .enumerate () {

println! ( "i = {} and j = {}" , i, j);

}
```

Wyjazd:

```
i = 0 i j = 5

i = 1 i j = 6

i = 2 i j = 7

i = 3 i j = 8

i = 4 i j = 9
```

Nie zapomnij umieścić nawiasów wokół zakresu.

W iteratorach:

```
# let lines = "hello \ nworld" .lines ();

for (line_number, line) in lineas.enumerate () {

println! ( "{}: {}" , line_number, line);

}
```

Wyjścia:

```
0: Zawartość pierwszego wiersza

1: Zawartość wiersza drugiego

2: Treść wiersza trzeciego

3: Zawartość wiersza czwartego
```

Wcześniejsze zakończenie iteracji

Rzućmy okiem na pętlę while, którą widzieliśmy wcześniej:

```
let mut x = 5 ;

let mut completed = false ;

while ! completed {

x + = x - 3 ;

println! ( "{}" , x);

if x% 5 == 0 {

completed = true ;

}

}
```

Musimy utrzymywać wiążącą zmienną mut , complete , aby wiedzieć, kiedy powinniśmy wyjść z cyklu. Rust ma dwa słowa kluczowe, które pomagają nam modyfikować proces iteracji: przerwać i kontynuować. W takim przypadku możemy napisać cykl w lepszy sposób za pomocą break :

```
let mut x = 5 ;

loop {

x + = x - 3 ;

println! ( "{}" , x);

if x% 5 == 0 { break ; }

}
```

Teraz wykonujemy pętlę w nieskończoność i używamy break, aby przerwać pętlę wcześniej. Korzystanie z jawnego zwrotu działa również dobrze w przypadku wcześniejszego zakończenia cyklu. Kontynuacja jest podobna, ale zamiast kończyć pętlę, powoduje przejście do następnej iteracji. Poniższe spowoduje wydrukowanie liczb nieparzystych:

```
for x in 0 .. 10 {

if x% 2 == 0 { continue ; }

println! ( "{}" , x);

}
```

Zapętlaj tagi

Możesz także napotkać sytuacje, w których masz zagnieżdżone pętle i musisz określić, do której pętli należą instrukcje break lub continue. Jak w większości języków, domyślnie przerwanie lub kontynuacja dotyczy najbardziej wewnętrznej pętli. W przypadku, gdy chcesz zastosować przerwanie lub kontynuację do któregokolwiek z cykli zewnętrznych, możesz użyć etykiet, aby określić, do którego cyklu ma zastosowanie instrukcja break lub continue. Poniższe zostaną wydrukowane tylko wtedy, gdy zarówno x, jak i y są nieparzyste:

```
'exterior : for x in 0 .. 10 {

'interior : for and in 0 .. 10 {

if x% 2 == 0 { continue 'exterior ; } // continue the loop above x

if and% 2 == 0 { continue 'inside ; } // continue the loop above and

println! ( "x: {}, y: {}" , x, y);

}

}
```

Należący

Ten przewodnik jest jednym z trzech wprowadzających system członkostwa w Rust. Jest to jedna z najbardziej unikalnych i nieodpartych cech Rusta, z którą programiści Rusta powinni być zaznajomieni. To dzięki członkostwu Rust osiąga swój najważniejszy cel, bezpieczeństwo. Istnieje kilka różnych koncepcji, z których każda ma swój własny rozdział:

* członkostwo, to, które aktualnie czytasz

* pożyczka i związane z nią charakterystyczne „referencje”

* czas życia, zaawansowana koncepcja pożyczki

Te trzy rozdziały są ze sobą powiązane, w kolejności. Będziesz potrzebować wszystkich trzech, aby w pełni zrozumieć system członkostwa w Rust.

Bramka

Zanim przejdziemy do szczegółów, dwie ważne uwagi dotyczące systemu członkostwa. Rust koncentruje się na bezpieczeństwie i szybkości. Rust osiąga te cele poprzez wiele „abstrakcji o zerowych kosztach”, co oznacza, że w Rust abstrakcje kosztują tak mało, jak to możliwe, aby działały. System członkostwa jest doskonałym przykładem abstrakcji o zerowych kosztach. Wszystkie analizy, o których będziemy mówić w tym przewodniku, są przeprowadzane w czasie kompilacji. Nie ponosisz żadnych kosztów w czasie działania żadnej z tych funkcji. Jednak ten system ma pewien koszt: krzywą uczenia się. Wielu nowych użytkowników Rusta doświadcza czegoś, co nazywamy „walką z sprawdzaniem wypożyczeń”, sytuacji, w której kompilator Rusta odmawia skompilowania programu, który autor uważa za prawidłowy. Dzieje się tak często, ponieważ mentalny model działania członkostwa programisty nie jest zgodny z obecnymi regułami zaimplementowanymi w Rust. Prawdopodobnie doświadczasz podobnych rzeczy na początku. Jest jednak dobra wiadomość: inni doświadczeni programiści Rust donoszą, że po pewnym czasie pracy z zasadami systemu członkostwa mają coraz mniej problemów z sprawdzaniem pożyczek. Mając to na uwadze, nauczmy się o przynależności.

Należący

Zmienna Bindings posiadająca właściwość Rust: te „mają członkostwo”, co jest powiązane. Oznacza to, że gdy powiązanie ze zmienną wykracza poza zakres, Rust zwalnia powiązane z nim zasoby. Na przykład:

```
fn foo () {

let v = vec! [ 1 , 2 , 3 ];
```

}

Kiedy v wchodzi w zakres, tworzony jest nowy Vec <T>. W tym przypadku wektor przydziela również trochę pamięci z kopca dla trzech elementów. Kiedy v wyjdzie poza zakres na końcu foo() , Rust wyczyści wszystko związane z wektorem, w tym pamięć przydzieloną z kopca. Dzieje się to deterministycznie, na końcu zakresu.

Semantyka ruchu

Jest tu coś bardziej subtelnego, Rust upewnia się, że istnieje tylko jedno powiązanie z danym zasobem. Na przykład, jeśli mamy wektor, możemy przypisać go do innego powiązania ze zmienną:

```
let v = vec! [ 1 , 2 , 3 ];

let v2 = v;
```

Ale jeśli później spróbujemy użyć v, otrzymamy błąd:

```
let v = vec! [1, 2, 3];

let v2 = v;

println! ("v [0] is: {}", v [0]);
```

The error looks like this:

```
error: use of moved value: `v` (use of moved value:` v`)

println! ("v [0] is: {}", v [0]);

^
```

Coś podobnego dzieje się, gdy zdefiniujemy funkcję, która przyjmuje członkostwo i spróbujemy użyć czegoś po przekazaniu tego jako argumentu:

```
fn take (v: Vec) {

// what happens here is not relevant

}

let v = vec! [1, 2, 3];

take (v);

println! ("v [0] is: {}", v [0]);
```

Ten sam błąd: „użycie przeniesionej wartości”. Kiedy przenosimy członkostwo na coś, mówimy, że „przenieśliśmy” rzecz, do której się odnosimy. Nie potrzebujesz do tego żadnej specjalnej adnotacji, co domyślnie robi Rust.

Szczegóły

Powód, dla którego nie możemy użyć powiązania ze zmienną po jej przeniesieniu, jest subtelny, ale ważny. Kiedy piszemy taki kod:

```
let v = vec! [ 1 , 2 , 3 ];

let v2 = v;
```

Pierwsza linia przydziela pamięć dla obiektu wektora v i danych, które zawiera. Obiekt wektorowy jest następnie przechowywany w stosie stosu i zawiera wskaźnik do zawartości ([1, 2, 3]) przechowywanej w kopcu. Kiedy przenosimy v do v2 , tworzona jest kopia tego wskaźnika dla v2 . Wszystko to oznacza, że w kopcu byłyby dwa wskaźniki zawartości wektorów. Narusza to gwarancje bezpieczeństwa Rust, wprowadzając warunek wyścigu. Dlatego Rust zabrania używania v po wykonaniu ruchu. Co ważne, niektóre optymalizacje mogą usuwać kopie bajtów ze stosu, w zależności od pewnych okoliczności. Może więc nie być tak nieefektywny, jak początkowo wygląda.

Kopiowanie typów

Ustaliliśmy, że kiedy przenosimy członkostwo do innego powiązania ze zmienną, nie możemy użyć oryginalnego powiązania. Istnieje jednak cecha, która zmienia to zachowanie, nazywa się Kopiuj. Nie omawialiśmy jeszcze cech, ale na razie możecie je postrzegać jako adnotacje dotyczące konkretnego typu, które dodają dodatkowe zachowanie. Na przykład:

```
let v = 1 ;

let v2 = v;

println! ( "v is: {}" , v);
```

W tym przypadku v to i32 , który implementuje cechę Kopiuj . Oznacza to, że podobnie jak w ruchu, kiedy przypisujemy v do v2 , tworzona jest kopia danych. Ale w przeciwieństwie do tego, co dzieje się w ruchu, możemy użyć v później. Dzieje się tak, ponieważ i32 nie ma wskaźników danych nigdzie indziej, kopiowanie go oznacza pełne kopiowanie. Wszystkie typy pierwotne implementują cechę Kopiuj, a ich członkostwo nie jest przenoszone, jak można by przypuszczać, zgodnie z regułami członkostwa. Na przykład poniższe dwa fragmenty kodu kompilują się tylko dlatego, że typy i32 i bool implementują cechę Copy .

```
fn main () {

let a = 5 ;

let _y = fold (a);

println! ( "{}" , a);

}

fn double (x: i32 ) -> i32 {

x * 2

}

fn main () {

let a = true ;

let _y = change_truth (a);

println! ( "{}" , a);

}

fn change_truth (x: bool ) -> bool {
```

! x

}

Gdybyśmy mieli typy, które nie implementowały cechy Copy , otrzymalibyśmy błąd kompilacji przy próbie użycia przeniesionej wartości.

error: use of moved value: `a`

println! ("{}", a);

^

Omówimy, jak tworzyć własne kopie typów w sekcji cech.

Więcej niż przynależność

Oczywiście, jeśli musimy zwrócić przynależność z każdą zapisywaną funkcją:

fn foo (v: Vec < i32 >) -> Vec < i32 > {

// zrób coś z v

// zwracanie członkostwa

v

}

Sprawy stałyby się dość nużące. Jest jeszcze gorzej, ponieważ mamy więcej rzeczy, do których chcemy należeć:

fn foo (v1: Vec < i32 >, v2: Vec < i32 >) -> (Vec < i32 >, Vec < i32 >, i32)

{

// zrób coś z v1 i v2

// zwracanie członkostwa, a także wynik naszej funkcji

(v1, v2, 42)

}

let v1 = vec! [1 , 2 , 3];

let v2 = vec! [1 , 2 , 3];

let (v1, v2, answer) = foo (v1, v2);

Fuj! Typ zwracany, linia powrotu i wywołanie funkcji stają się znacznie bardziej skomplikowane. Na szczęście Rust oferuje ułatwienie, pożyczkę, ułatwienie, które pomaga nam rozwiązać ten problem. To temat kolejnej części!

Referencje i pożyczka

Ten przewodnik jest jednym z trzech wprowadzających system członkostwa w Rust. Jest to jedna z najbardziej unikalnych i atrakcyjnych cech Rusta, z którą programiści Rusta powinni być dobrze zaznajomieni. Członkostwo to sposób, w jaki Rust osiąga swój najwyższy cel, bezpieczeństwo zarządzania pamięcią. Istnieje kilka różnych koncepcji, z których każda ma swój własny rozdział:

członkostwo, główna koncepcja

kredyt, ten, który teraz czytasz

razy życia, zaawansowana koncepcja pożyczki

Te trzy rozdziały są ze sobą powiązane i uporządkowane. Będziesz musiał przeczytać wszystkie trzy, aby w pełni zrozumieć system członkostwa.

Cele

Zanim przejdziemy do szczegółów, dwie ważne uwagi dotyczące systemu członkostwa. Rust koncentruje się na bezpieczeństwie i szybkości. Rust osiąga te cele poprzez wiele „abstrakcji o zerowych kosztach", co oznacza, że w Rust abstrakcje kosztują tak mało, jak to możliwe, aby działały. System członkostwa jest doskonałym przykładem abstrakcji o zerowych kosztach. Wszystkie analizy, o których będziemy mówić w tym przewodniku, są przeprowadzane w czasie kompilacji. Nie ponosisz żadnych kosztów w czasie działania żadnej z tych funkcji. Jednak ten system ma pewien koszt: krzywą uczenia się. Wielu nowych użytkowników Rusta doświadcza czegoś, co nazywamy „walką z sprawdzaniem wypożyczeń", sytuacji, w której kompilator Rusta odmawia skompilowania programu, który autor uważa za prawidłowy. Dzieje się tak często, ponieważ mentalny model działania członkostwa programisty nie jest zgodny z obecnymi regułami zaimplementowanymi w Rust. Prawdopodobnie doświadczasz podobnych rzeczy na początku. Jest jednak dobra wiadomość: inni doświadczeni programiści Rust donoszą, że po pewnym czasie pracy z zasadami systemu członkostwa mają coraz mniej problemów z sprawdzaniem pożyczek. Mając to na uwadze, poznajmy pożyczkę.

Pożyczka

Na końcu sekcji członkostwa mieliśmy brzydką funkcję, która wyglądała tak:

```
fn foo (v1: Vec < i32 >, v2: Vec < i32 >) -> ( Vec < i32 >, Vec < i32 >, i32 )

{

// zrób coś z v1 i v2

// zwracanie członkostwa, a także wynik naszej funkcji

(v1, v2, 42 )

}

niech v1 = vec! [ 1 , 2 , 3 ];

niech v2 = vec! [ 1 , 2 , 3 ];

niech (v1, v2, odpowiedź) = foo (v1, v2);
```

Powyższe nie jest jednak idiomatyczne Rust, ponieważ nie korzysta z zalet pożyczki. Oto pierwszy krok:

```
fn foo (v1: & Vec < i32 >, v2: & Vec < i32 >) -> i32 {

// zrób coś z v1 i v2

// zwrócenie odpowiedzi

42

}
```

```
let v1 = vec! [ 1 , 2 , 3 ];

let v2 = vec! [ 1 , 2 , 3 ];

let (v1, v2, answer) = foo (v1, v2);

// możemy tutaj użyć v1 i v2
```

Zamiast przyjmować Vec <i32> s jako argumenty, bierzemy referencję: & Vec <i32> . I zamiast bezpośrednio przekazywać v1 i v2, przekazujemy & v1 i & v2 . Nazywamy typ & T „referencją” i zamiast przejąć własność zasobu, pożycza go. Zmienne łącze, które coś pożycza, nie zwalnia zasobu, gdy wykracza poza zakres. Oznacza to, że po wywołaniu foo() możemy ponownie skorzystać z oryginalnych powiązań zmiennych. Odwołania są niezmienne, podobnie jak łącza zmienne. Przekłada się to na to, że w foo () wektory nie mogą być zmieniane:

```
fn foo (v: & Vec) {

v.push (5);

}

let v = vec! [];

foo (& v);
```

fails with:

```
error: cannot borrow immutable borrowed content `* v` as mutable

v.push (5);

^
```

Wstawienie wartości powoduje mutację w wektorze i nie wolno nam tego robić.

referencje i mut

Istnieje drugi rodzaj referencji&mut T . „Zmienne odniesienie”, które umożliwia mutację pożyczanego zasobu. Na przykład:

```
let mut x = 5 ;

{

let y = & mut x;

* and + = 1 ;

}

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

Powyższe spowoduje wydrukowanie 6 . Uczynimy y zmiennym odniesieniem do x , a następnie dodamy jeden do tego, na co wskazuje y. Zauważysz, że x również musiało być oznaczone jako mut , w przeciwnym razie nie bylibyśmy w stanie wziąć zmiennej pożyczki o niezmiennej wartości. Zauważysz również, że dodaliśmy gwiazdkę (*) przed y, czyniąc ją *, a to dlatego, że y jest odwołaniem do & mut. Będziesz także musiał ich użyć, aby uzyskać dostęp do treści odniesienia. W przeciwnym razie referencje & mut są jak referencje. Istnieje duża różnica między nimi i ich interakcjami. Zauważyłeś, że

w poprzednim przykładzie jest coś, co nie pachnie zbyt dobrze, ponieważ potrzebujemy tego dodatkowego zakresu z { y } . Jeśli je usuniemy, otrzymamy błąd:

błąd: nie można pożyczyć `x` jako niezmiennego, ponieważ jest również pożyczone jako

error: cannot borrow `x` as immutable because it is also borrowed as

mutable

println! ("{}", x);

^

uwaga: tutaj występuje poprzednie wypożyczenie `x`; zmienna pożyczka zapobiega kolejnym ruchom, pożyczkom lub modyfikacjom `x` aż do zakończenia pożyczki

let y = & mut x;

^

uwaga: poprzednie wypożyczenie kończy się tutaj

fn main () {

}

^

Podobno są zasady.

Zasady

Oto zasady dotyczące pożyczki w Rust:

Po pierwsze, każda pożyczka musi znajdować się w królestwie nie większym niż królestwo właściciela. Po drugie, możesz mieć jeden lub drugi z tych rodzajów pożyczek, ale nie oba jednocześnie:

* jedno lub więcej odniesień (& T) do zasobu,

* dokładnie zmienną referencję (& mut T).

Możesz zauważyć, że jest to bardzo podobne, ale nie to samo, do definicji wyścigu:

Istnieje „stan wyścigu”, gdy dwa lub więcej wskaźników uzyskuje dostęp do tej samej lokalizacji w pamięci w tym samym czasie, gdzie co najmniej jeden zapisuje, a operacje nie są zsynchronizowane. Z referencjami możesz mieć tyle, ile chcesz, ponieważ żadne z nich nie pisze. Jeśli piszesz i potrzebujesz dwóch lub więcej wskaźników do tej samej pamięci, możesz mieć tylko jeden & mut naraz. W ten sposób Rust zapobiega wyścigom w czasie kompilacji: dostaniemy błędy, jeśli złamiemy zasady. Mając to na uwadze, rozważmy ponownie nasz przykład.

Myślę o zakresach

Oto kod:

let mut x = 5;

let y = & mut x;

* and + = 1;

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

Powyższy kod generuje następujący błąd:

```
error: cannot borrow `x` as immutable because it is also borrowed as

mutable

println! ("{}", x);

^
```

Dzieje się tak, ponieważ naruszyliśmy zasady: mamy & mut T wskazujące na x , w związku z czym nie wolno nam tworzyć żadnych & T . Jedna rzecz lub inna. Notatka wskazuje, jak myśleć o tym problemie:

```
uwaga: poprzednie wypożyczenie kończy się tutaj

fn main () {

}

^
```

Innymi słowy, zmienna pożyczka jest utrzymywana przez resztę naszego przykładu. Chcemy, aby nasza zmienna pożyczka zakończyła się, zanim spróbujemy wywołać println! i zróbmy niezmienną pożyczkę. W Rust pożyczka jest powiązana z obszarem, na którym obowiązuje pożyczka. Nasze tereny wyglądają tak:

```
let mut x = 5;

let y = & mut x; // - + x loan & mut starts here

// |

* and + = 1; // |

// |

println! ("{}", x); // - + - attempt to borrow x here

// - + x loan & mut ends here
```

Konflikt zakresów: nie możemy utworzyć & x, gdy y jest w zakresie. Kiedy więc dodamy nawiasy klamrowe:

```
let mut x = 5 ;

{

let y = & mut x; // - + x loan & mut starts here

* and + = 1 ; // |

} // - + ... and ends here

println! ( "{}" , x); // <- attempt to borrow x here
```

Bez problemu. Nasza zmienna pożyczka wykracza poza zakres, zanim stworzymy niezmienną pożyczkę. Zakres jest kluczem do sprawdzenia, jak długo trwa pożyczka.

Problemy, którym zapobiega pożyczka

Dlaczego mamy takie restrykcyjne przepisy? Cóż, jak zauważyliśmy, te zasady zapobiegają warunkom wyścigu. Jakiego rodzaju problemy powodują warunki wyścigu? Tutaj kilka.

Unieważnienie iteratora

Przykładem jest „przesłonięcie iteratora”, które występuje, gdy próbujesz zmutować kolekcję podczas iteracji nad nią. Kontroler pożyczek Rusta zapobiega temu:

```
let mut v = vec! [ 1 , 2 , 3 ];

for i in & v {

println! ( "{}" , i);

}
```

Powyższe wydruki od jednego do trzech. Podczas iteracji wektorów otrzymujemy tylko odniesienia do ich elementów. v sam w sobie jest niezmiennie pożyczany, co oznacza, że nie możemy go zmienić podczas iteracji:

```
let mut v = vec! [1, 2, 3];

for i in & v {

println! ("{}", i);

v.push (34);

}
```

Oto błąd:

```
error: cannot borrow `v` as mutable because it is also borrowed as immutable

v.push (34);

^

uwaga: tutaj występuje poprzednie wypożyczenie `v`; niezmienna pożyczka zapobiega

kolejne ruchy lub zmienne pożyczki `v` aż do końca pożyczki

for i in & v {

^

uwaga: poprzednie wypożyczenie kończy się tutaj

for i in & v {

println! (“{}”, i);

v.push (34);

}

^
```

Nie możemy zmodyfikować v, ponieważ jest ono pożyczone przez cykl.

używać po zwolnieniu

Referencje nie powinny żyć dłużej niż zasób, na który wskazują Rust sprawdzi zakres twoich referencji, aby upewnić się, że to prawda. Gdyby Rust nie zweryfikował tej właściwości, moglibyśmy przypadkowo użyć nieprawidłowej referencji. Na przykład:

```
let y: & i32;

{

let x = 5;

y = & x;

}

println! ("{}", and);
```

Otrzymujemy następujący błąd:

```
error: `x` does not live long enough

y = & x;

^

uwaga: odwołanie musi być ważne dla sufiksu bloku następującego po instrukcji 0 at 2:16 ...

let y: & i32;

{

let x = 5;

y = & x;

}

uwaga: ... ale pożyczona wartość jest ważna tylko dla następującego po niej sufiksu bloku wyciąg 0 o 4:18

let x = 5;

y = & x;

}
```

Innymi słowy, y jest ważne tylko dla obszaru, w którym istnieje x. Gdy tylko x opuści, odwołanie jest nieważne. Dlatego błąd mówi, że pożyczka „nie żyje wystarczająco długo” („nie żyje wystarczająco długo”), ponieważ nie jest ważna przez odpowiednią ilość czasu. Ten sam problem pojawia się, gdy referencja jest deklarowana przed zmienną, do której się odnosi. Dzieje się tak, ponieważ zasoby w tym samym zakresie są zwolnione w kolejności odwrotnej do kolejności, w jakiej zostały zgłoszone:

```
let y: & i32;

let x = 5;

y = & x;
```

```
println! ("{}", and);
```

Otrzymujemy ten błąd:

```
error: `x` does not live long enough
y = & x;
^
uwaga: odwołanie musi być ważne dla sufiksu bloku następującego po instrukcji 0 at 2:16 ...
let y: & i32;
let x = 5;
y = & x;
println! ("{}", and);
}
uwaga: ... ale pożyczona wartość jest ważna tylko dla następującego po niej sufiksu bloku stwierdzenie 1 o 3:14
let x = 5;
y = & x;
println! ("{}", and);
}
```

W powyższym przykładzie y jest zadeklarowane przed x , co oznacza, że y żyje dłużej niż x , co jest niedozwolone.

Czasy życia

Ten przewodnik jest jednym z trzech wprowadzających system członkostwa w Rust. Jest to jedna z najbardziej unikalnych i atrakcyjnych cech Rusta, z którą programiści Rusta powinni być dobrze zaznajomieni. Członkostwo to sposób, w jaki Rust osiąga swój najwyższy cel, bezpieczeństwo zarządzania pamięcią. Istnieje kilka różnych koncepcji, z których każda ma swój własny rozdział:

* członkostwo, główna koncepcja

*[pożyczka] [pożyczka] i związane z nią charakterystyczne „referencje”

*Czasy życia, te, które teraz czytasz

Te trzy rozdziały są ze sobą powiązane i uporządkowane. Będziesz musiał przeczytać wszystkie trzy, aby w pełni zrozumieć system członkostwa.

Cele

Zanim przejdziemy do szczegółów, dwie ważne uwagi dotyczące systemu członkostwa. Rust koncentruje się na bezpieczeństwie i szybkości. Rust osiąga te cele poprzez wiele „abstrakcji o zerowych kosztach”, co oznacza, że w Rust abstrakcje kosztują tak mało, jak to możliwe, aby działały. System członkostwa jest doskonałym przykładem abstrakcji o zerowych kosztach. Wszystkie analizy, o których będziemy mówić w tym przewodniku, są przeprowadzane w czasie kompilacji. Nie ponosisz

żadnych kosztów w czasie działania żadnej z tych funkcji. Jednak ten system ma pewien koszt: krzywą uczenia się. Wielu nowych użytkowników Rusta doświadcza czegoś, co nazywamy „walką z pożyczaczem”, sytuacją, w której kompilator Rusta odmawia skompilowania programu, który autor uważa za poprawny. Dzieje się tak często, ponieważ mentalny model działania członkostwa programisty nie jest zgodny z obecnym

zasady zaimplementowane w Rust. Prawdopodobnie doświadczasz podobnych rzeczy na początku. Jest jednak dobra wiadomość: inni doświadczeni programiści Rust donoszą, że po pewnym czasie pracy z zasadami systemu członkostwa mają coraz mniej problemów z sprawdzaniem pożyczek. Mając to na uwadze, poznajmy czasy życia.

Czasy życia

Użyczenie odniesienia do innego zasobu, którego właścicielem jest ktoś inny, może być trudne. Na przykład wyobraź sobie ten zestaw operacji:

Dostaję uchwyt do jakiegoś zasobu.

Pożyczam ci odnośnik do źródła.

Decyduję, że skończyłem z zasobem i zwalniam go, dopóki nadal masz do niego odniesienie.

Decydujesz się użyć zasobu.

O nie! Twoje odwołanie wskazuje na nieprawidłowy zasób. Nazywa się to wiszącym wskaźnikiem lub użyciem po zwolnieniu, gdy zasobem jest pamięć. Aby to naprawić, musimy upewnić się, że krok czwarty nigdy nie nastąpi po kroku trzecim. System członkostwa w Rust robi to poprzez koncepcję zwaną czasem życia, która opisuje zakres, w jakim referencja jest ważna. Kiedy mamy funkcję, która przyjmuje referencję jako argument, możemy określić czas życia referencji w sposób dorozumiany lub jawny:

```
// implicit

fn foo (x: & i32 ) {

}

// explicit

fn bar < 'a > (x: & ' a i32 ) {

}
```

Litera „a” oznacza „czas życia a”. Technicznie rzecz biorąc, każda referencja ma powiązany czas życia, ale kompilator pozwala na pominięcie ich w typowych przypadkach. Zanim do tego przejdziemy, spójrzmy na jawny fragment kodu:

```
fn bar <'a> (...)
```

Wcześniej rozmawialiśmy trochę o składni funkcji, ale nie omawialiśmy <> s po nazwie funkcji. Funkcja może mieć „parametry ogólne” pomiędzy <> s, a czasy życia są rodzajem parametru ogólnego. W dalszej części książki omówimy inne rodzaje leków generycznych, ale na razie skupmy się tylko na aspekcie czasu życia. Używamy <>, aby zadeklarować nasze czasy życia. To mówi, że pasek ma czas życia, „a . Gdyby miał odniesienia jako parametry, wyglądałby tak:

```
fn bar <'a,' b = ""> (...)
```

Dlatego na naszej liście parametrów używamy czasów życia, które nazwaliśmy:

... (x: & 'a i32)

Gdybyśmy chcieli skierowania i mut , moglibyśmy wykonać następujące czynności:

... (x: & 'mut i32)

Jeśli porównasz & mut i32 z & 'do mut i32', są one takie same, to po prostu ten czas w życiu', aby znaleźć się pomiędzy & a mut i32 . Czytamy & mutujemy i32 jako „zmienne odniesienie do i32” i & „mutujemy i32 jako” zmienne odniesienie do i32 z czasem życia „a”.

W struct s

Będziesz także potrzebować wyraźnych czasów życia podczas pracy ze struct s:

struct Foo < 'a > {

x: & 'to i32 ,

}

fn main () {

let y = & 5 ; // this is the same as `let _y = 5; let y = & _y; `

let f = Foo {x: y};

println! ("{}" , fx);

}

Jak widać, struktury mogą mieć również żywotność. Podobnie jak funkcje,

struct Foo < 'a > {

x: & 'a i32 ,

#}

deklaruje czas życia i

struct Foo < 'a > {

x: & 'to i32 ,

#}

korzysta z niego. Więc po co nam czas życia tutaj? Musimy się upewnić, że żadne odniesienie do Foo nie może żyć dłużej niż odniesienie do i32, które zawiera.

bloki impl

Zaimplementujmy metodę w Foo:

struct Foo < 'a > {

x: & 'to i32 ,

}

```
impl < 'a > Foo < ' a > {

fn x (& self ) -> & 'a i32 { self .x}

}

fn main () {

let y = & 5 ; // this is the same as `let _y = 5; let y = & _y; `

let f = Foo {x: y};

println! ( "x is: {}" , fx ());

}
```

Jak widać, musimy zadeklarować czas życia dla Foo w wierszu impl . Powtarzamy 'a dwa razy, tak jak w funkcjach: impl <' a> definiuje czas życia 'a , a Foo <' a> z niego korzysta.

Wielokrotna żywotność

Jeśli masz wiele referencji, możesz wielokrotnie użyć tego samego czasu życia:

```
fn x_o_y < 'a > (x: & ' a str , y: & ' a str ) -> & ' a str {

# X

#}
```

Powyższe mówi, że zarówno x, jak i y żyją w tym samym zakresie i że wartość zwracana jest również żywa dla tego zakresu. Gdybyś chciał, aby x i y miały różne czasy życia, mógłbyś skorzystać z wielu parametrów czasu życia:

```
fn x_o_y < 'a , ' b > (x: & 'a str , y: & ' b str ) -> & 'a str {

# X

#}
```

W tym przykładzie x i y mają różne prawidłowe zakresy, ale zwracana wartość ma taki sam czas życia jak x .

Myślę o zakresach

Jednym ze sposobów myślenia o okresach życia jest wizualizacja środowiska, w którym odniesienie jest ważne. Na przykład:

```
fn main () {

let y = & 5 ; // - + and enter scope

// |

// stuff // |

// |

} // - + and leaves scope
```

Dodanie naszego Foo:

```
struct Foo < 'a > {
x: & 'to i32 ,
}
fn main () {
let y = & 5 ; // - + and enter scope
let f = Foo {x: y}; // - + f enters scope
// stuff // |
// |
} // - + fyy go out of scope
```

Nasze f mieści się w zakresie y , dlatego wszystko działa. Co by się stało inaczej? Poniższy kod nie zadziała:

```
struct Foo <'a> {
x: & 'to i32,
}
fn main () {
let x; // - + x enters scope
// |
{// |
let y = & 5; // --- + and enter scope
let f = Foo {x: y}; // --- + f enters scope
x = & f.x; // | | mistake here
} // --- + fyy go out of scope
// |
println! ("{}", x); // |
} // - + x goes out of scope
```

Uff! Jak widać tutaj, zakresy f i y są mniejsze niż zakres x . Ale kiedy robimy x = & f.x , czynimy x odniesieniem do czegoś, co jest poza zakresem. Nazwane cykle życia to sposób na nadanie tym ustawieniom nazwy. Nadanie czemuś nazwy jest pierwszym krokiem do tego, by móc o tym mówić.

static

Czas życia zwany „statycznym” to szczególny czas życia. Wskazuje to, że coś ma czas życia całego programu. Większość programistów Rusta zna „static” w przypadku ciągów znaków:

```
let x: & 'static str = "Hello, world." ;
```

Literały ciągów znaków mają typ & 'static str, ponieważ referencja jest zawsze żywa: są one umieszczane w segmencie danych końcowego pliku binarnego. Innym przykładem są globalne:

static FOO: i32 = 5 ;

let x: & 'static i32 = & FOO;

Powyższe dodaje i32 do segmentu danych pliku binarnego, a x jest do niego odniesieniem.

Elizja życia

Rust obsługuje potężne wnioskowanie o typach w ciałach funkcji, ale w sygnaturach elementów zabronione jest zezwalanie na wnioskowanie oparte wyłącznie na sygnaturze. Jednak ze względów ergonomicznych w firmach funkcyjnych ma zastosowanie bardzo ograniczone wtórne wnioskowanie zwane „elizją czasu życia". „Elizja czasu życia" wnioskuje tylko na podstawie składników sygnatury, bez polegania na treści funkcji, wnioskuje tylko o parametrach czasu życia i robi to za pomocą tylko trzech łatwych do zapamiętania i jednoznacznych reguł. Wszystko to sprawia, że pisanie podpisu jest skrótem czasowym, bez potrzeby ukrywania zaangażowanych typów, ponieważ zostanie do nich zastosowane pełne wnioskowanie lokalne. Mówiąc o elizji czasu życia, używamy terminów wejściowy czas życia i wyjściowy czas życia. Czas życia wejścia to czas życia związany z parametrem funkcji, a czas życia wyjścia to czas życia związany z wartością zwracaną funkcji. Na przykład następująca funkcja ma czas życia danych wejściowych:

fn foo <'a> (bar: &' a str)

It has an output lifetime:

fn foo <'a> () -> &' a str

The following has a life time in both positions:

fn foo <'a> (bar: &' a str) -> & 'a str

Oto trzy zasady:

Każdy czas życia pominięty w argumentach funkcji staje się innym parametrem czasu życia.

Jeśli istnieje dokładnie jeden czas życia danych wejściowych, wyeliminowany lub nie, ten czas życia jest przypisywany do czasu życia wszystkich elips w wartościach zwracanych przez tę funkcję.

Jeśli istnieje wiele czasów życia wejścia, ale jednym z nich jest & self lub & mut self , własny czas życia jest przypisywany do wszystkich wykluczonych czasów życia wyjścia. W przeciwnym razie błędem jest unikanie czasu życia danych wyjściowych.

Przykłady

Oto kilka przykładów funkcji z wykluczonymi okresami życia. Połączyliśmy każdy przykład pominiętego czasu życia z jego rozszerzoną formą.

fn print (s: & str); // elided

fn print <'a> (s: &' a str); // expanded

fn debug (lvl: u32, s: & str); // elided

fn debug <'a> (lvl: u32, s: &' a str); // expanded

// In the example above, `lvl` does not need a lifetime because it is not a

reference (` & `). Only things related to references (like a `struct` that

contains a reference) need lifetime.

fn substr (s: & str, until: u32) -> & str; // elided

fn substr <'a> (s: &' a str, until: u32) -> & 'a str; // expanded

fn get_str () -> & str; // ILLEGAL, no inputs

fn frob (s: & str, t: & str) -> & str; // ILLEGAL, two inputs

fn frob <'a,' b = ""> (s: & 'a str, t: &' b str) -> & str; // Expanded:

Output lifetime is ambiguous

fn get_mut (& mut self) -> & mut T; // elided

fn get_mut <'a> (&' a mut self) -> & 'a mut T; // expanded

fn args (& mut self, args: & [T]) -> & mut Command // elided

fn args <'a,' b, = "" T: ToCStr = ""> (& 'a mut self, args: &' b [T]) -> &

'a mut Command // expanded

fn new (buf: & mut [u8]) -> BufWriter; // elided

fn new <'a> (buf: &' a mut [u8]) -> BufWriter <'a> // expanded

Zmienność

Zmienność, czyli zdolność do zmiany rzeczy, działa trochę inaczej w Rust niż w innych językach. Pierwszym aspektem zmienności jest to, że nie jest ona domyślnie włączona:

let x = 5;

x = 6; // error!

Możemy wprowadzić zmienność za pomocą słowa kluczowego mut :

let mut x = 5 ;

x = 6 ; // no problem!

To jest link do zmiennej zmiennej. Gdy link do zmiennej jest zmienny, oznacza to, że możesz zmienić to, na co wskazuje link. Tak więc w powyższym przykładzie nie zmieniasz wartości w x , zamiast tego link zmienił się z jednego i32 na inny. Jeśli chcesz zmienić to, co link wskazuje na zmienną, będziesz potrzebować zmiennego odniesienia:

let mut x = 5 ;

let y = & mut x;

y jest niezmiennym linkiem zmiennej do zmiennego odniesienia, co oznacza, że nie możesz powiązać y z czymś innym (y = & mut z), ale możesz zmutować wszystko, z czym y jest powiązane (* y = 5). Bardzo subtelna różnica. Oczywiście, jeśli potrzebujesz obu:

```
let mut x = 5 ;
```

```
let mut y = & mut x;
```

Teraz i może być powiązany z inną wartością, a wartość, do której się odnosi, można zmienić. Ważne jest, aby pamiętać, że mut jest częścią wzorca , więc możesz robić takie rzeczy jak:

```
let ( mut x, y) = ( 5 , 6 );
```

```
fn foo ( mut x: i32 ) {
#}
```

Zmienność wewnętrzna a zmienność zewnętrzna

Jednak kiedy mówimy, że coś jest „niezmienne” w Rust, nie oznacza to, że nie można tego zmienić: mówimy, że coś ma „zewnętrzną zmienność”. Rozważmy na przykład Arc <T> :

```
use std :: sync :: Arc;
let x = Arc :: new ( 5 );
let y = x.clone ();
```

Kiedy wywołujemy clone() , Arc <T> musi zaktualizować licznik odwołań. Chociaż nie użyliśmy tutaj żadnego muta, x jest niezmiennym łączem, nie bierzemy również & mut 5 ani więcej. Więc co się dzieje? Aby to zrozumieć, musimy wrócić do sedna filozofii, która kieruje Rust, bezpieczeństwa w zarządzaniu pamięcią i mechanizmu, za pomocą którego Rust to gwarantuje, systemu członkostwa, a dokładniej pożyczki: Możesz mieć jedno lub drugie z te dwa rodzaje pożyczek, ale nie oba jednocześnie:

* jedno lub więcej odniesień (& T) do zasobu,

* dokładnie zmienną referencję (& mut T).

To jest prawdziwa definicja „niezmienności”: czy bezpiecznie jest mieć dwa wskaźniki? W przypadku Arc <T> 's, jeśli: mutacja jest całkowicie zawarta w samej strukturze. Nie jest dostępny dla użytkownika. Z tego powodu clone() & T s zwraca . Gdybym dostarczył & mut T s, byłby to problem. Inne typy, takie jak std :: cell module , mają przeciwieństwo: wewnętrzną zmienność. Na przykład:

```
use std :: cell :: RefCell;
let x = RefCell :: new ( 42 );
let y = x.borrow_mut ();
```

RefCel zapewnia & mut odniesienia do tego, co zawierają za pomocą metody lend_mut() . Czy to nie jest niebezpieczne? A co jeśli zrobimy:

```
use std :: cell :: RefCell;
let x = RefCell :: new (42);
let y = x.borrow_mut ();
let z = x.borrow_mut ();
# (and Z);
```

To w efekcie spowoduje panikę w czasie wykonywania. Oto, co robi RefCell: wymusza reguły pożyczania Rusta w czasie wykonywania i panikuje! jeśli zasady te zostaną naruszone. To pozwala nam podejść do innego aspektu reguł zmienności Rusta. Najpierw porozmawiajmy o tym.

Zmienność na poziomie pola

Zmienność jest właściwością pożyczki (&mut) lub zmiennego połączenia (let mut). Przekłada się to na przykład na to, że nie możesz mieć struktury z niektórymi zmiennymi i niektórymi niezmiennymi polami:

```
struct Point {

x: i32,

mut y: i32, // nope

}
```

Zmienność struktury polega na jej połączeniu ze zmienną:

```
struct Point {

x: i32,

y: i32,

}

let mut a = Point {x: 5, y: 6};

ax = 10;

let b = Point {x: 5, y: 6};

bx = 10; // error: cannot assign to immutable field `bx`
```

Jednak używając Cell <T> , możesz emulować zmienność na poziomie pola:

```
use std :: cell :: Cell;

struct Point {

x: i32 ,

y: Cell < i32 >,

}

let dot = Dot {x: 5 , y: Cell :: new ( 6 )};

dot.y.set ( 7 );

println! ( "y: {:?}" , punto.y);
```

Spowoduje to wydrukowanie y: Cell {value: 7} . Zaktualizowaliśmy i zadowalająco.

Struktury

Struktury (struct s) to sposób na tworzenie bardziej złożonych typów danych. Na przykład, gdybyśmy wykonywali obliczenia obejmujące współrzędne w przestrzeni 2D, potrzebowalibyśmy zarówno wartości x, jak i wartości y:

```
let origin_x = 0 ;

let origin_y = 0 ;
```

Struktura pozwala nam połączyć oba w jeden zunifikowany typ danych:

```
struct Point {

x: i32 ,

y: i32 ,

}

fn main () {

let origin = Point {x: 0 , y: 0 }; // origin: Point

println! ( "The origin is in ({}, {})" , origin.x, origin.y);

}
```

Dużo się tu dzieje, więc skupmy się na tym. Deklarujemy strukturę za pomocą słowa kluczowego struct , po którym następuje nazwa. Zgodnie z konwencją, struktury zaczynają się wielką literą i mają wielbłądzią wielkość liter: DotInTheSpace , a nie Dot_In_The_Space . Możemy jak zwykle utworzyć instancję naszej struktury za pomocą let , ale używamy składni klucz: wartość, aby przypisać każde pole. Kolejność nie musi być taka sama jak w pierwotnym oświadczeniu. Wreszcie, ponieważ mamy nazwy pól, możemy uzyskać do nich dostęp za pomocą notacji kropkowej: origin.x . Wartości w struct s są domyślnie niezmienne, podobnie jak inne łącza do zmiennych w Rust. Użyj mut, aby uczynić je zmiennymi:

```
struct Point {

x: i32 ,

y: i32 ,

}

fn main () {

let mut dot = Dot {x: 0 , y: 0 };

dot.x = 5 ;

println! ( "The origin is in ({}, {})" , dot.x, dot.y);

}
```

Spowoduje to wydrukowanie Początek jest w (5, 0) .

Rust nie obsługuje zmienności pól na poziomie języka, dlatego nie możesz napisać czegoś takiego:

```
struct Point {
```

```
mut x: i32,

y: i32,

}
```

Zmienność jest właściwością zmiennego łącza, a nie samej struktury. Jeśli jesteś przyzwyczajony do zmienności na poziomie pola, może to początkowo wydawać się nieco dziwne, ale znacznie upraszcza sprawę. Pozwala nawet na modyfikowanie rzeczy tylko przez krótki okres czasu:

```
struct Point {

x: i32,

y: i32,

}

fn main () {

let mut dot = Dot {x: 0, y: 0};

dot.x = 5;

let dot = dot; // now, this new link to variable cannot be changed

dot.y = 6; // this causes an error
```

Zaktualizuj składnię

Struktura może zawierać .., aby wskazać, że chcesz użyć kopii innej struktury dla niektórych wartości. Na przykład:

```
struct Point3d {

x: i32 ,

y: i32 ,

z: i32 ,

}

let mut dot = Dot3d {x: 0 , y: 0 , z: 0 };

point = Point3d {y: 1 , .. point};
```

To przypisuje nowy punkt y , ale zachowuje stare wartości x i z . Nie musi to być ta sama struktura, możesz skorzystać z tej składni podczas tworzenia nowych, a ona skopiuje wartości, których nie określisz:

```
# struct Punto3d {

# x: i32 ,

# y: i32 ,

# z: i32 ,

#}
```

let origin = Point3d {x: 0 , y: 0 , z: 0 };

let dot = Dot3d {z: 1 , x: 2 , .. origin};

Struktury krotek (struktury krotek)

Rust ma inny typ danych, który jest jak hybryda między krotką a strukturą, zwany strukturą tulla (tuple struct). Struktury krotki mają nazwę, ale ich pola nie:

struct Color (i32 , i32 , i32);

struct Point (i32 , i32 , i32);

Poprzednie dwa nie będą takie same, nawet jeśli mają te same wartości:

struct Color (i32 , i32 , i32);

struct Point (i32 , i32 , i32);

let black = Color (0 , 0 , 0);

let origin = Point (0 , 0 , 0);

Prawie zawsze lepiej jest użyć struktury niż krotki struktury. Zamiast tego moglibyśmy napisać Color i Punto w następujący sposób:

struct Color {

red: i32 ,

blue: i32 ,

green: i32 ,

}

struct Point {

x: i32 ,

y: i32 ,

z: i32 ,

}

Teraz zamiast stanowisk mamy nazwiska. Dobre nazwy są ważne, a dzięki struct mamy nazwy. Istnieje przypadek, w którym krotka struktury jest bardzo użyteczna i jest to krotka struktury z pojedynczym elementem. Nazywane wzorcem nuevotipo (newtype), ponieważ tworzą nowy typ, różny od zawartej w nim wartości, wyrażający samą semantykę:

struct Inches (i32);

let length = Inches (10);

let Inches (integer_length) = length;

println! ("length is {} inches" , integer_length);

Jak zauważysz, możesz wyodrębnić całą zawartość za pomocą destruct let , tak jak w zwykłych krotkach. W tym przypadku Cale (długość_liczby_całkowitej) pozwala przypisać 10 do liczby_długości_liczby_całkowitej .

Struktury podobne do jednostek

Możesz zdefiniować strukturę bez żadnego członka:

elektron strukturalny;

Ta struktura jest nazywana type-unit (unit-like) ze względu na swoje podobieństwo do pustej krotki () , czasami nazywanej unit (unit). Jako struktura krotek definiuje nowy typ. Powyższe rzadko jest przydatne samo w sobie (chociaż czasami może służyć jako typ znacznika), ale w połączeniu z innymi cechami może stać się przydatne. Na przykład biblioteka może wymagać utworzenia struktury, która implementuje określoną cechę do obsługi zdarzeń. Jeśli nie masz żadnych danych do zapisania w strukturze, możesz po prostu utworzyć strukturę typu unit .

Wyliczenia

Enum (enum) w Rust to typ reprezentujący dane, które mogą być jednym z zestawu możliwych wariantów:

enum Message {

Leave,

ChangeColor (i32 , i32 , i32)

Move {x: i32 , y: i32 },

Write (String),

}

Każdy wariant może opcjonalnie mieć powiązane dane. Składnia definiowania wariantów jest podobna do składni używanej do definiowania struktur (struct s): możesz mieć warianty bez danych (takie jak struktury jednostek), warianty z nazwanymi danymi i warianty z danymi nienazwanymi (takie jak struktury krotek) .). Jednak w przeciwieństwie do definicji struktur, enum jest pojedynczym typem. Wartość wyliczeniowa może pasować do dowolnego wariantu. Z tego powodu wyliczenie jest czasami nazywane „typem sumy”: zbiór możliwych wartości wyliczenia jest sumą zestawów

możliwe wartości dla każdego wariantu. Używamy :: składni, aby wykorzystać każdy wariant: warianty mieszczą się w zakresie enum . Co sprawia, że następujące stwierdzenie jest ważne:

enum Message {

Move {x: i32 , y: i32 },

#}

let x: Message = Message :: Move {x: 3 , y: 4 };

enum Turn Game Table {

Move {cells: i32 },

Pass,

}

let y: GameTurnTurn = TableTurnTurn :: Move {cells: 1 };

Oba warianty mają nazwę Move , ale ponieważ ich zakres mieści się w nazwie wyliczenia, można ich używać bez konfliktów. Wartość typu wyliczeniowego zawiera informacje o tym, który to wariant, oprócz wszelkich danych powiązanych z tym wariantem. Nazywa się to czasem „związkiem tagowanym”, ponieważ dane zawierają „znacznik” wskazujący, jaki to jest typ. Kompilator wykorzystuje te informacje, aby upewnić się, że uzyskujemy dostęp do danych w wyliczeniu w bezpieczny sposób. Na przykład nie można po prostu próbować zmienić struktury wartości tak, jakby była jednym z możliwych wariantów:

fn process_color_change (msg: Message) {

let Message :: ChangeColor (r, v, a) = msg; // compile-time error

}

Brak obsługi tego typu operacji może wydawać się nieco ograniczający, ale jest to ograniczenie, które możemy pokonać. Istnieją dwa sposoby, implementacja równości na własną rękę lub poprzez dopasowywanie wzorców za pomocą wyrażeń dopasowujących, których nauczysz się w następnej sekcji. Nadal nie wiemy wystarczająco dużo o Rust, aby samodzielnie wdrożyć równość, ale zobaczymy to w sekcji cech.

Konstruktory jako funkcje

Konstruktor wyliczenia może być również użyty jako funkcja. Na przykład:

enum Message {

Write (String),

#}

let m = Message :: Write ("Hello, world" .to_string ());

Is the same as

enum Message {

Write (String),

#}

fn foo (x: String) -> Message {

Message :: Write (x)

}

let x = foo ("Hello, world" .to_string ());

Nie jest to dla nas od razu przydatne, ale kiedy przejdziemy do domknięć, porozmawiamy o przekazywaniu funkcji jako argumentów do innych funkcji. Na przykład za pomocą iteratorów możemy przekonwertować wektor z String s na wektor z Message :: Write

S:

```
# enum Message {

# Write ( String ),

#}

let v = vec! [ "Hello" .to_string (), "World" .to_string ()];

let v1: Vec <Message> = v.into_iter (). map (Message :: Write) .collect ();
```

Dopasowanie

Często proste if / else nie wystarcza, ponieważ masz więcej niż dwie możliwe opcje. Ponadto warunki mogą być złożone. Rust ma zarezerwowane słowo match , które pozwala zastąpić skomplikowane konstrukcje if / else czymś potężniejszym. Wymeldować się:

```
let x = 5 ;

match x {

1 => println! ( "one" ),

2 => println! ( "two" ),

3 => println! ( "trees" ),

4 => println! ( "four" ),

5 => println! ( "five" ),

_ => println! ( "something else" ),

}
```

match przyjmuje wyrażenie, a następnie rozwidla się na podstawie jego wartości. Każde ramię gałęzi ma postać wartość => wyrażenie . Gdy wartość jest zgodna, oceniane jest wyrażenie ramienia. Nazywa się to dopasowanie terminem „dopasowywanie wzorców”, którego implementacją jest dopasowanie. Jest cała sekcja o wzorach, która obejmuje wszystkie możliwe wzory. Jaka jest więc duża zaleta? Cóż, jest ich kilka. Po pierwsze dopasowanie narzuca wyczerpanie czeków („sprawdzanie kompletności”). Czy widzisz ostatnie ramię, to z podkreśleniem (_)? Jeśli usuniemy to ramię, Rust zwróci nam błąd:

```
error: non-exhaustive patterns: `_` not covered
```

Innymi słowy, Rust próbuje nam powiedzieć, że zapomnieliśmy o jakiejś wartości. Ponieważ x jest liczbą całkowitą, Rust wie, że x może mieć wiele różnych wartości - na przykład 6 . Jednak bez _ nie ma pasującego ramienia, w związku z czym Rust odmawia kompilacji kodu. _ działa jak „ramię, które wszystko łapie”. Jeśli żadne z pozostałych ramion nie pasuje, ramię z _ będzie, a ponieważ powiedzieliśmy „przechwyć wszystko ramię”, mamy teraz ramię dla każdej możliwej wartości x , w wyniku czego nasz program pomyślnie się skompiluje. match jest również wyrażeniem, co oznacza, że możemy go użyć po prawej stronie let lub bezpośrednio tam, gdzie używane jest wyrażenie:

```
let x = 5 ;

let number = match x {

1 => "one" ,

2 => "two" ,
```

```
3 => "three" ,

4 => "four" ,

5 => "five" ,

_ => "something else" ,

};
```

Czasami jest to dobry sposób na przekonwertowanie czegoś z jednego typu na inny.

Dopasowywanie w wyliczeniach

Innym ważnym zastosowaniem słowa kluczowego match jest przetwarzanie możliwych wariantów wyliczenia:

```
enum Message {

Leave,

ChangeColor ( i32 , i32 , i32 )

Move {x: i32 , y: i32 },

Write ( String ),

}

fn exit () { / * ... * / }

fn change_color (r: i32 , g: i32 , b: i32 ) { / * ... * / }

fn move_cursor (x: i32 , y: i32 ) { / * ... * / }

fn process_message (msg: Message) {

match msg {

Message :: Exit => exit (),

Message :: ChangeColor (r, g, b) => change_color (r, g, b),

Message :: Move {x: x, y: y} => move_cursor (x, y),

Message :: Write (s) => println! ( "{}" , s),

};

}
```

Ponownie kompilator Rust sprawdza wyczerpanie, żądając posiadania ramienia dla każdego wariantu wyliczenia. Jeśli pominiesz jeden, Rust wygeneruje błąd w czasie kompilacji, chyba że użyjesz _ . W przeciwieństwie do poprzednich zastosowań match , nie można w tym celu użyć instrukcji if . Możesz skorzystać z instrukcji if let, która może być postrzegana jako krótka forma dopasowania.

Wektor

„Wektor” to dynamiczna tablica, zaimplementowana jako typ standardowej biblioteki Vec<T>. T oznacza, że możemy mieć wektory dowolnego typu (zajrzyj do rozdziału [rodzaje] [rodzaje], aby

uzyskać więcej informacji). Wektory zawsze przechowują swoje dane na kopcu. Możesz tworzyć wektory za pomocą makra vec! :

```
let v = vec! [ 1 , 2 , 3 , 4 , 5 ]; // v: Vec <i32>
```

(Zauważ, że w przeciwieństwie do makra println!, którego używaliśmy w przeszłości, używamy nawiasów kwadratowych [] z makrem vec! . Rust pozwala na użycie dowolnego w każdej sytuacji, tym razem jest to czysto umowne) Istnieje alternatywna forma z vec! powtórzyć wartość początkową:

```
let v = vec! [ 0 ; 10]; // dziesięć zer
```

Dostęp do elementów

Aby uzyskać wartość w określonym indeksie wektora, używamy [] s:

```
let v = vec! [ 1 , 2 , 3 , 4 , 5 ];

println! ("Trzecim elementem v jest {}" , v [ 2 ]);
```

Indeksy zaczynają się od 0 , więc trzecim elementem jest v [2] .

Iteracja

Gdy masz już wektor, możesz przeglądać jego elementy za pomocą for . Istnieją trzy wersje:

```
let mut v = vec! [ 1 , 2 , 3 , 4 , 5 ];

for i in & v {

println! ( "A reference to {}" , i);

}

for i in & mut v {

println! ( "A mutable reference to {}" , i);

}

for i in v {

println! ( "Taking membership of the vector and its element {}" , i);

}
```

Ciągi znaków

Ciągi znaków są ważną koncepcją do opanowania przez każdego programistę. System obsługi ciągów znaków w Rust różni się nieco od tego w innych językach, ze względu na skupienie się na programowaniu systemowym. Tak długo, jak masz strukturę danych o zmiennym rozmiarze, sprawy mogą być trochę trudne, a ciągi znaków to struktura danych, która może różnić się rozmiarem. To powiedziawszy, ciągi znaków w Rust również działają inaczej niż w niektórych innych językach programowania systemowego, takich jak C. Przejdźmy do szczegółów. „Ciąg znaków” („ciąg”) to sekwencja wartości skalarnych Unicode zakodowanych jako strumień bajtów UTF-8. Gwarantuje się, że wszystkie ciągi znaków są prawidłowym kodowaniem sekwencji UTF-8. Ponadto, w przeciwieństwie do innych języków systemowych, ciągi znaków nie są zakończone znakiem null i mogą zawierać bajty zerowe. Rust ma dwa główne typy ciągów znaków: & str i String . Porozmawiajmy najpierw o & str . Są to tak zwane „plastry sznurkowe”. Literały łańcuchowe są typu & „static str”:

let greeting = "Hello." ; // greeting: & 'static str

Ten ciąg znaków jest przypisywany statycznie, co oznacza, że jest przechowywany w naszym skompilowanym programie i istnieje przez cały czas jego wykonywania. Link powitalny jest odniesieniem do estetycznie przypisanego ciągu znaków. Fragmenty ciągów znaków mają stały rozmiar i nie można ich modyfikować. Z drugiej strony String to ciąg znaków przypisany z kopca. Ten łańcuch może rosnąć i gwarantuje się, że będzie to UTF-8. Ciągi są zwykle tworzone przez konwersję fragmentu ciągu znaków przy użyciu metody to_string .

let mut s = "Hello" .to_string (); // mut s: String println! ("{}" , s);

s.push_str (", world.");

println! ("{}" , s);

The String s haran coercion to a & str with & :

fn get_scrap (bit: & str) {

println! ("I received: {}" , piece);

}

fn main () {

let s = "Hello" .to_string ();

receive_piece (& s);

}

Ten przymus nie występuje w przypadku funkcji, które akceptują jedną z cech & str zamiast & str . Na przykład TcpStream :: connect ma parametr typu ToSocketAddrs . A & str jest w porządku, ale ciąg musi zostać jawnie przekonwertowany za pomocą & * .

use std :: net :: TcpStream;

TcpStream :: connect ("192.168.0.1:3000"); // parameter & str

let address_string = "192.168.0.1:3000" .to_string ();

TcpStream :: connect (& * address_string); // converting address_string

to & str

Wyświetlanie łańcucha jako & str jest tanie, ale konwersja & str na ciąg wymaga przydziału pamięci. Nie ma powodu, aby to robić, chyba że jest to konieczne!

Indeksowane

Ponieważ ciągi znaków są poprawne w formacie UTF-8, nie obsługują indeksowania:

let s = "hello";

println! ("The first letter of s is {}", s [0]); // ERROR!!!

Dostęp do wektora za pomocą [] jest zwykle bardzo szybki. Ale ponieważ każdy znak zakodowany w łańcuchu UTF-8 może mieć wiele bajtów, musisz przeszukać cały łańcuch, aby znaleźć n^{th} literę ciągu. Jest to znacznie droższa operacja i nie chcemy wprowadzać zamieszania. Nawet „litera” nie jest

dokładnie czymś zdefiniowanym w Unicode. Możemy wybrać wyświetlanie ciągu znaków jako pojedynczych bajtów lub punktów kodowych:

```
let hachiko = " 忠　犬　ハ　チ　公" ;

for b in hachiko.as_bytes () {

print! ( "{}," , b);

}

println! ( "" );

for c in hachiko.chars () {

print! ( "{}," , c);

}

println! ( "" );
```

Powyższe wydruki:

229, 191, 160, 231, 138, 172, 227, 131, 143, 227, 131, 129, 229, 133, 172,

忠, 犬, ハ, チ, 公,

Jak widać, bajtów jest więcej niż znaków (char s).

Możesz uzyskać coś podobnego do takiego indeksu:

```
# let hachiko = " 忠　犬　ハ　チ　公" ;

let dog = hachiko.chars (). nth ( 1 ); // something like hachiko [1]
```

To podkreśla, że musimy przejść od początku listy znaków.

Krajanie na plastry

Możesz uzyskać fragment ciągu znaków za pomocą składni cut:

```
let dog = "hachiko" ;

let hachi = & dog [ 0 .. 5 ];
```

Należy jednak pamiętać, że są to przesunięcia bajtów, a nie przesunięcia znaków. Tak więc następujące elementy zakończą się niepowodzeniem w czasie wykonywania:

```
let dog = " 忠　犬　ハ　チ　公";

let hachi = & dog [0..2];
```

z tym błędem:

thread '' panicked at 'index 0 and / or 2 in ` 忠　犬　ハ　チ　公` do not lie on character boundary

Powiązanie

Jeśli masz String , możesz połączyć & str na końcu:

```
let hello = "Hello" .to_string ();
```

let mundo = "world!" ;

let hello_world = hello + world;

Ale jeśli masz dwa String s, potrzebujesz & :

let hello = "Hello" .to_string ();

let mundo = "world!" .to_string ();

let hello_world = hello + & world;

Dzieje się tak, ponieważ & String może automatycznie wymuszać & str . Ta funkcja nosi nazwę „Koercje Deref”.

Ogólny

Czasami podczas pisania funkcji lub struktury danych możemy chcieć, aby działała z wieloma typami argumentów. W Rust możemy to osiągnąć za pomocą generyków. Typy generyczne nazywane są „polimorfizmem parametrycznym” w teorii typów, co oznacza, że są to typy lub funkcje, które mają wiele kształtów („poli” oznacza wielokrotność, „przekształcenie” oznacza kształt) dla określonego parametru („parametryczny”). Tak czy inaczej, dość o teorii typów, spójrzmy na ogólny kod. Standardowa biblioteka Rust zapewnia typ, Option <T> , który jest ogólny:

enum Option <T> {

Some (T),

None ,

}

Część <T> , którą widzieliśmy już kilka razy, wskazuje, że jest to ogólny typ danych. W deklaracji naszego wyliczenia, gdziekolwiek widzimy literę T, zastępujemy ten typ tym samym typem, który został użyty w typie ogólnym. Oto przykład użycia Option <T> z kilkoma dodatkowymi adnotacjami:

let x: Option < i32 > = Some (5);

W deklaracji typu mówimy Option <i32> . Zwróć uwagę, jak to wygląda podobnie do Option <T> . Więc w tej opcji T ma wartość i32 . Po prawej stronie wiązania wykonujemy Some (T) , gdzie T wynosi 5 . Ponieważ 5 to i32 , obie strony pasują, a Rust jest szczęśliwy. Gdyby się nie zgadzały, otrzymalibyśmy błąd:

let x: Option = Some (5);

// error: mismatched types: expected `core :: option :: Option`,

// found `core :: option :: Option <_>` (expected f64 but found integral variable)

To nie znaczy, że nie możemy tworzyć Option <T> s, które zawierają f64 . Po prostu muszą pasować:

let x: Option < i32 > = Some (5);

let y: Option < f64 > = Some (5.0f64);

Bardzo dobry. Jedna definicja, wiele zastosowań.

Rodzaje niekoniecznie muszą być ogólne dla jednego typu. Rozważ inny podobny typ w standardowej bibliotece Rusta, Result <T, E> :

```
enum Result <T, E> {

Ok (T),

Err (E),

}
```

Ten typ jest ogólny dla dwóch typów: T i E . Nawiasem mówiąc, wielkie litery mogą być dowolne. Moglibyśmy zdefiniować Wynik <T, E> jako:

```
enum Result <A, Z> {

Ok (A),

Err (Z),

}
```

chcieć. Konwencja mówi, że pierwszym parametrem ogólnym musi być T , typu „, i że używamy E dla „błąd”. Rust jednak to nie obchodzi. Typ Result <T, E> służy do zwracania wyniku obliczenia, z możliwością zwrócenia błędu w przypadku niepowodzenia takiego obliczenia.

Funkcje ogólne

Możemy pisać funkcje, które przyjmują typy ogólne o podobnej składni:

```
fn receives_any_thing <T> (x: T) {

// do something with x

}
```

Składnia składa się z dwóch części: <T> mówi „ta funkcja jest ogólna dla typu T”, a część x: T mówi „x ma typ T”. Wiele argumentów może mieć ten sam typ:

```
fn receives_two_things_from_the same_type <T> (x: T, y: T) {

// ...

}
```

Moglibyśmy napisać wersję, która otrzymuje wiele typów:

```
fn receives_two_things_from_different_types <T, U> (x: T, y: U) {

// ...

}
```

Struktury ogólne

Możesz także przechowywać typ ogólny w strukturze:

```
struct Point <T> {

x: T,
```

```
y: T,
}

let integer_source = Point {x: 0 , y: 0 };
let float_origin = Point {x: 0.0 , y: 0.0 };
```

Podobnie jak w przypadku funkcji, w sekcji <T> deklarujemy parametry generyczne, po czym używamy również x:T w deklaracji typu. Gdy chcemy dodać implementację dla ogólnej struktury, wystarczy zadeklarować parametr type po impl :

```
# struct Point <T> {
# x: T,
# y: T,
#}
#
impl <T> Point <T> {
fn exchange (& mut self ) {
std :: mem :: swap (& mut self .x, & mut self .y);
}
```

Do tej pory widziałeś tylko generyczne, które akceptują absolutnie każdy typ. Są przydatne w wielu przypadkach, widziałeś już Option <T> , a później spotkasz uniwersalne kontenery, takie jak Vec <T> . Z drugiej strony, czasami będziesz chciał zamienić tę elastyczność na większą moc ekspresji. Przeczytaj o ograniczeniach cech, aby zobaczyć, dlaczego i jak.

Cechy

Cecha to narzędzie językowe, które informuje kompilator Rusta o funkcjonalności, jaką musi zapewniać typ. Pamiętacie impla? Słowo kluczowe używane do wywołania funkcji o składni metody ?

```
struct Circle {
x: f64 ,
y: f64 ,
radius: f64 ,
}
impl Circle {
fn area (& self ) -> f64 {
std :: f64 :: consts :: PI * ( self .radio * self .radio)
}
```

}

Cechy są podobne, z wyjątkiem tego, że definiujemy cechę tylko z sygnaturą metody, a następnie implementujemy cechę dla struktury. Więc:

struct Circle {

x: f64 ,

y: f64 ,

radius: f64 ,

}

trait HasArea {

fn area (& self) -> f64 ;

}

impl HasArea for Circle {

fn area (& self) -> f64 {

std :: f64 :: consts :: PI * (self .radio * self .radio)

}

}

Jak widać, blok cech wygląda bardzo podobnie do bloku impl, ale nie definiujemy bloku, tylko sygnaturę typu. Kiedy implementujemy cechę, używamy Impl Trait for Item , zamiast tylko Impl Item .

Granice cech dla funkcji generycznych

Cechy są przydatne, ponieważ pozwalają facetowi składać pewne obietnice dotyczące jego zachowania. Funkcje generyczne mogą to wykorzystać do ograniczenia typów, które akceptują. Rozważ tę funkcję, która się nie kompiluje:

fn imrimir_area (figure: T) {

println! ("This figure has an area of {}", figura.area ());

}

Rust narzeka:

error: no method named `area` found for type` T` in the current scope

Ponieważ T może być dowolnego typu, nie możemy być pewni, że implementuje metodę area. Ale możemy dodać „ograniczenie cechy” do naszego ogólnego T , upewniając się, że je implementuje:

trait HasArea {

fn area (& self) -> f64 ;

#}

fn print_area <T: HasArea> (shape: T) {

```
println! ( "This figure has an area of {}" , figura.area ());
}
```

Składnia <T: hasArea> przekłada się na „dowolny typ, który implementuje cechę hasArea”. Ponieważ cechy definiują sygnatury typów funkcji, możemy być pewni, że każdy typ implementujący hasArea będzie miał metodę .area() . Oto rozszerzony przykład tego, jak to działa:

```
trait HasArea {
fn area (& self ) -> f64 ;
}
struct Circle {
x: f64 ,
y: f64 ,
radius: f64 ,
}
impl HasArea for Circle {
fn area (& self ) -> f64 {
std :: f64 :: consts :: PI * ( self .radio * self .radio)
}
}
Square struct {
x: f64 ,
y: f64 ,
side: f64 ,
}
Impl HasArea for Square {
fn area (& self ) -> f64 {
self. side * self. side
}
}
fn imrimir_area <T: tieneArea> (figure: T) {
println! ( "This figure has an area of {}" , figura.area ());
}
fn main () {
```

```
let c = Circle {
x: 0.0f64 ,
y: 0.0f64 ,
radius: 1.0f64 ,
};
let s = Square {
x: 0.0f64 ,
y: 0.0f64 ,
side: 1.0f64 ,
};
print_area (c);
print_area (s);
}
```

Ten program generuje dane wyjściowe:

Ta figura ma pole 3,141593

Ta figura ma pole 1

Jak widać, print_area jest teraz ogólny, ale zapewnia również, że udostępniliśmy prawidłowe typy. Jeśli przekażemy nieprawidłowy typ:

```
print_area (5);
```

Otrzymujemy błąd w czasie kompilacji:

błąd: cecha `HasArea` nie jest zaimplementowana dla typu` _` [E0277]

Granice cech dla struktur generycznych

Twoje ogólne struktury mogą również skorzystać z ograniczeń cech. Wszystko, co musisz zrobić, to dodać ograniczenie podczas deklarowania parametrów typu. Oto nowy typ Rectangle <T> i jego działanie is_square :

```
struct Rectangle <T> {
x: T,
y: T,
width: T,
height: T,
}
impl <T: PartialEq > Rectangle <T> {
```

```
fn is_square (& self ) -> bool {
self. width == self. height
}
}
fn main () {
let mut r = Rectangle {
x: 0 ,
y: 0 ,
width: 47 ,
height: 47 ,
};
assert! (r.es_cuadrado ());
r.height = 42 ;
assert! (! r.es_cuadrado ());
}
```

is_square () musi sprawdzić, czy boki są równe, a do tego typy muszą być typu, który implementuje rdzeń cechy :: cmp :: PartialEq :

Impl Rectangle {...}

Teraz prostokąt można zdefiniować na podstawie dowolnego typu, który można porównać przez równość. Zdefiniowaliśmy nową strukturę Rectangle, która akceptuje liczby o dowolnej precyzji, obiekty dowolnego typu, o ile można je porównać przez równość. Czy moglibyśmy zrobić to samo dla naszych struktur HasArea , Square i Circle ? Tak, ale wymagają mnożenia, a żeby z tym pracować, musimy wiedzieć więcej o cechach operatorów.

Zasady wdrażania cech

Do tej pory dodaliśmy tylko implementacje cech do struktur, ale możesz zaimplementować dowolną cechę dla dowolnego typu. Technicznie rzecz biorąc, moglibyśmy zaimplementować TieneArea dla i32:

```
trait HasArea {
fn area (& self ) -> f64 ;
}
impl TieneArea for i32 {
fn area (& self ) -> f64 {
println! ( "this is silly" );
* self as f64
```

}

}

5 .area ();

Implementowanie metod na tych prymitywnych typach jest uważane za kiepski styl, nawet jeśli jest to możliwe. Może to wyglądać jak na starym zachodzie, ale istnieją dwa ograniczenia dotyczące wdrażania cech, które zapobiegają wymknięciu się sytuacji spod kontroli. Po pierwsze, jeśli cecha nie jest zdefiniowana w twoim zakresie, nie ma zastosowania. Oto przykład: biblioteka standardowa zapewnia cechę Write, która dodaje dodatkową funkcjonalność do File s, umożliwiając operacje wejścia/wyjścia pliku. Domyślnie plik nie miałby swoich metod:

let mut f = std :: fs :: File :: open ("foo.txt"). ok (). expect ("Failed to

open foo.txt");

let buf = b "anything"; // byte string literal. buf: & [u8; 8]

let result = f.write (buf);

result.unwrap (); // ignore the error

Here is the error:

error: type `std :: fs :: File` does not implement any method in scope

named` write`

let result = f.write (buf);

^ ~~~~~~~~~

Najpierw musimy użyć cechy Write :

use std :: io :: Write;

let mut f = std :: fs :: File :: open ("foo.txt"). ok (). expect ("Failed to

open foo.txt");

let buf = b "anything";

let result = f.write (buf);

result.unwrap (); // ignore the error

Powyższe skompiluje się bez błędów. Oznacza to, że nawet jeśli ktoś zrobi coś złego, na przykład doda metody do i32 , nie wpłynie to na ciebie, chyba że użyjesz tej cechy. Jest jeszcze jedno ograniczenie dotyczące implementacji cech: jedno z dwóch, albo cecha, albo typ, dla którego piszesz impl , musi być zdefiniowane przez Ciebie. Moglibyśmy więc zaimplementować cechę hasArea dla typu i32 , ponieważ HasArea jest w naszym kodzie. Ale gdybyśmy spróbowali zaimplementować ToString , cechę dostarczoną przez Rust dla i32 , nie moglibyśmy, ponieważ ani cecha, ani typ nie występują w naszym kodzie. Ostatnia rzecz dotycząca cech: funkcje generyczne z limitem cechy używają „monomorfizacji” („monomorfizacji”) (mono: jeden, morfy: kształt) i dlatego są wysyłane statycznie. Co to znaczy? Więcej informacji znajdziesz w rozdziale poświęconym obiektom cech.

Wiele limitów cech

Widziałeś, że możesz ograniczyć parametr typu ogólnego za pomocą cechy:

```
fn foo <T: Clone > (x: T) {

x.clone ();

}
```

Jeśli potrzebujesz więcej niż jednego limitu, możesz użyć + :

```
use std :: fmt :: Debug;

fn foo <T: Clone + Debug> (x: T) {

x.clone ();

println! ( "{:?}" , x);

}
```

T musi teraz być zarówno Clone, jak i Debug .

Klauzula wheere

Pisanie funkcji z zaledwie kilkoma typami ogólnymi i niewielką liczbą ograniczeń cech nie jest takie brzydkie, ale wraz ze wzrostem liczby składnia staje się trochę dziwna:

```
use std :: fmt :: Debug;

fn foo <T: Clone , K: Clone + Debug> (x: T, y: K) {

x.clone ();

y.clone ();

println! ( "{:?}" , and);

}
```

Nazwa funkcji jest daleko po lewej stronie, a lista parametrów jest daleko po prawej stronie. Granice cech przeszkadzają. Rust ma rozwiązanie i nazywa się „klauzula Where”:

```
use std :: fmt :: Debug;

fn foo <T: Clone , K: Clone + Debug> (x: T, y: K) {

x.clone ();

y.clone ();

println! ( "{:?}" , and);

}

fn bar <T, K> (x: T, y: K) where T: Clone , K: Clone + Debug {

x.clone ();

y.clone ();

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

```
}

fn main () {

foo ( "Hello" , "world" );

bar ( "Hello" , "world" );

}
```

foo () używa zademonstrowanej wcześniej składni, a bar () używa klauzuli where . Wszystko, co musisz zrobić, to pozostawić ograniczenia podczas definiowania parametrów typu, a następnie dodać gdzie po liście parametrów. W przypadku dłuższych list można dodać spacje:

```
use std :: fmt :: Debug;

fn bar <T, K> (x: T, y: K)

where T: Clone ,

K: Clone + Debug {

x.clone ();

y.clone ();

println! ( "{:?}" , and);

}
```

Taka elastyczność może zwiększyć przejrzystość w złożonych sytuacjach. Klauzula where jest również potężniejsza niż najprostsza składnia. Na przykład:

```
Trait become <Output> {

fn convert (& self ) -> Output;

}

impl become < i64 > for i32 {

fn convert (& self ) -> i64 {* self as i64 }

}

// can be called with T == i32

fn normal <T: ConvertA < i64 >> (x: & T) -> i64 {

x.convert ()

}

// can be called with T == i64

inverse fn <T> () -> T

// pesto is user ConvertA as if it were "ConvertA <i64>"

where i32 : ConvertA <T> {
```

```
42 .convert ()

}
```

Powyższe pokazuje dodatkową cechę where : dopuszcza ograniczenia, gdzie lewa strona jest dowolnym typem (w tym przypadku i32), a nie tylko prostym parametrem typu (jak T).

Metody domyślne

Jeśli już wiesz, jak typowy implementator zdefiniuje metodę, możesz pozwolić, aby Twoja cecha zapewniała metodę domyślną:

```
trait Foo {

fn is_valid (& self ) -> bool ;

fn is_invalid (& self ) -> bool {! self .es_valido ()}

}
```

Implementatorzy cech Foo muszą zaimplementować es_valido() , ale nie muszą implementować es_valido() . Otrzymają go domyślnie. Mogą również zastąpić domyślną implementację, jeśli chcą:

```
# trait Foo {

# fn is_valid (& self ) -> bool ;

#

# fn is_invalid (& self ) -> bool {! self .es_valido ()}

#}

struct UsaDefault;

impl Foo for UsaDefault {

fn is_valid (& self ) -> bool {

println! ( "UsaDefault.es_valid called." );

true

}

}

struct OverwriteDefault;

impl Foo for OverwriteDefault {

fn is_valid (& self ) -> bool {

println! ( "Override default._valid call." );

true

}

fn is_invalid (& self ) -> bool {
```

```
println! ( "OverwriteDefault.es_valid call!" );
true // this implementation is a self-contradiction!
}
}
```

Dziedzictwo

Czasami wdrożenie jednej cechy wymaga wdrożenia innej:

```
trait Foo {
fn foo (& self );
}
trait FooBar : Foo {
fn foobar (& self );
}
```

Implementatorzy FooBar muszą również zaimplementować Foo , tak jak poniżej:

```
# trait Foo {
# fn foo (& self );
#}
# trait FooBar : Foo {
# fn foobar (& self );
#}
struct Baz;
impl Foo for Baz {
fn foo (& self ) { println! ( "foo" ); }
}
impl FooBar for Baz {
fn foobar (& self ) { println! ( "foobar" ); }
}
```

łąd: cecha `main :: Foo` nie jest zaimplementowana dla typu` main ::

Baz` [E0277]

Upuszczać

Teraz, gdy omówiliśmy cechy, porozmawiajmy o konkretnej cesze zapewnianej przez standardową bibliotekę Rust, Drop . Cecha Drop umożliwia wykonanie kodu, gdy wartość wykracza poza zakres. Na przykład:

```
struct HasDrop;

impl Drop for HasDrop {

fn drop (& mut self ) {

println! ( "Dropping!" );

}

}

fn main () {

let x = HasDrop;

// let's do something

} // x leaves scope here
```

Kiedy x wychodzi poza zakres na końcu main() , wykonywany jest kod Drop. Drop ma metodę, zwaną także drop() . Ta metoda przyjmuje zmienne odwołanie do self . Otóż to! Mechanika gry Drop jest bardzo prosta, jednak jest kilka szczegółów. Na przykład wartości są usuwane w odwrotnej kolejności niż zostały zadeklarowane. Oto inny przykład:

```
Explosive struct {

power: i32 ,

}

impl Drop for Explosive {

fn drop (& mut self ) {

println! ( "BOOM multiplied by {} !!!" , self .power);

}

}

fn main () {

let firecracker = Explosive {power: 1 };

let tnt = Explosive {power: 100 };

}
```

Powyższe wydrukuje:

```
BOOM pomnożony przez 100 !!!

BOOM pomnożony przez 1 !!!
```

TNT jest pierwsze niż petarda, ponieważ powstał później. Ostatni do wejścia, pierwszy do wyjścia. Więc do czego służy Drop? Ogólnie rzecz biorąc, służy do czyszczenia wszelkich zasobów powiązanych ze strukturą. Na przykład typ Arc <T> to facet z liczeniem referencji. Po wywołaniu Drop zmniejszy liczbę odwołań, a jeśli całkowita liczba odwołań wynosi zero, usunie wartość bazową.

Upuszczać

Teraz, gdy omówiliśmy cechy, porozmawiajmy o konkretnej cesze zapewnianej przez standardową bibliotekę Rust, Drop . Cecha Drop umożliwia wykonanie kodu, gdy wartość wykracza poza zakres. Na przykład:

```
struct HasDrop;

impl Drop for HasDrop {

fn drop (& mut self ) {

println! ( "Dropping!" );

}

}

fn main () {

let x = HasDrop;

// let's do something

} // x leaves scope here
```

Kiedy x wychodzi poza zakres na końcu main() , wykonywany jest kod Drop. Drop ma metodę, zwaną także drop() . Ta metoda przyjmuje zmienne odwołanie do self . Otóż to! Mechanika gry Drop jest bardzo prosta, jednak jest kilka szczegółów. Na przykład wartości są usuwane w odwrotnej kolejności niż zostały zadeklarowane. Oto inny przykład:

```
Explosive struct {

power: i32 ,

}

impl Drop for Explosive {

fn drop (& mut self ) {

println! ( "BOOM multiplied by {} !!!" , self .power);

}

}

fn main () {

let firecracker = Explosive {power: 1 };

let tnt = Explosive {power: 100 };

}
```

Powyższe wydrukuje:

```
BOOM pomnożony przez 100 !!!

BOOM pomnożony przez 1 !!!
```

TNT jest pierwsze niż petarda, ponieważ powstał później. Ostatni do wejścia, pierwszy do wyjścia. Więc do czego służy Drop? Ogólnie rzecz biorąc, służy do czyszczenia wszelkich zasobów powiązanych ze strukturą. Na przykład typ Arc <T> to facet z liczeniem referencji. Po wywołaniu Drop zmniejszy liczbę odwołań, a jeśli całkowita liczba odwołań wynosi zero, usunie wartość bazową.

if let

if let pozwala na łączenie if i let w celu zmniejszenia kosztów niektórych typów dopasowywania wzorców. Załóżmy na przykład, że mamy jakąś opcję Option <T> . Chcemy wywołać na nim funkcję, jeśli jest to Some <T> , ale nic nie robimy, jeśli jest to None . To byłoby coś takiego:

```
# let option = Some ( 5 );

# fn foo (x: i32 ) {}

match option {

Some (x) => {foo (x)},

None => {},

}
```

Nie musimy tutaj używać dopasowania, na przykład moglibyśmy użyć if :

```
# let option = Some ( 5 );

# fn foo (x: i32 ) {}

if option.is_some () {

let x = option.unwrap ();

foo (x);

}
```

Żadna z tych dwóch opcji nie jest szczególnie atrakcyjna. Możemy użyć if let, aby zrobić to samo, ale w lepszy sposób:

```
# let option = Some ( 5 );

# fn foo (x: i32 ) {}

if let Some (x) = option {

foo (x);

}
```

Jeśli wzorzec dobrze pasuje, wiąże odpowiednią część wartości z identyfikatorami we wzorcu, a następnie ocenia wyrażenie. Jeśli wzór nie pasuje, nic się nie dzieje. Jeśli chcesz coś zrobić na wypadek, gdyby wzór nie pasował, możesz użyć w przeciwnym razie :

else :

```
# let option = Some ( 5 );

# fn foo (x: i32 ) {}
```

```
# fn bar () {}

if let Some (x) = option {

foo (x);

} else {

Pub();

}
```

while let

Podobnie, podczas gdy let można użyć, gdy chcesz warunkowo iterować, o ile wartość pasuje do określonego wzorca. Konwertuj kod w ten sposób:

```
# let option: Option < i32 > = None ;

loop {

match option {

Some (x) => println! ( "{}" , x),

_ => break ,

}

}
```

W takim kodzie:

```
# let option: Option < i32 > = None ;

while let Some (x) = option {

println! ( "{}" , x);

}
```

Obiekty cech

Gdy kod zawiera polimorfizm, potrzebny jest mechanizm określający, która konkretna wersja powinna zostać uruchomiona. Ten mechanizm nazywa się biurem. Istnieją dwie główne formy wysyłki: wysyłka statyczna i wysyłka dynamiczna. Chociaż prawdą jest, że Rust preferuje wysyłanie statyczne, obsługuje również wysyłanie dynamiczne poprzez mechanizm zwany „obiektami cech”.

Bazy

Do końca tego rozdziału będziemy potrzebować cechy i kilku implementacji. Stwórzmy prosty, Foo . Foo ma jedną metodę, która zwraca String .

```
# trait Foo { fn method (& self ) -> String ; }

impl Foo for u8 {

fn method (& self ) -> String { format! ( "u8: {}" , * self )}

}
```

```
impl Foo for String {
fn method (& self ) -> String { format! ( "string: {}" , * self )}
}
impl Foo for String {
fn method (& self ) -> String { format! ( "string: {}" , * self )}
}
```

Wysyłka statyczna

Możemy użyć cechy do wykonania statycznej wysyłki, używając limitów cech:

```
# trait Foo { fn method (& self ) -> String ; }
# impl Foo for u8 { fn method (& self ) -> String { format! ( "u8: {}" , * self )}}
# impl Foo for String { fn method (& self ) -> String { format! ( "string: {}" , * self )}}
fn do_something <T: Foo> (x: T) {
x.method ();
}
fn main () {
let x = 5u8 ;
let y = "Hello" .to_string ();
do_something (x);
do_something (y);
}
```

Rust używa „monomorfizacji” do statycznej wysyłki w tym kodzie. Co oznacza, że utworzysz specjalną wersję do_something() zarówno dla u8, jak i String , a następnie zastąpisz miejsca wywołań wywołaniami tych wyspecjalizowanych funkcji. Innymi słowy, Rust generuje coś takiego:

```
# trait Foo { fn method (& self ) -> String ; }
# impl Foo for u8 { fn method (& self ) -> String { format! ( "u8: {}" , *
self )}}
# impl Foo for String { fn method (& self ) -> String { format! ( "string:
{}" , * self )}}
fn do_something_u8 (x: u8 ) {
x.method ();
}
fn do_something_string (x: String ) {
```

```
x.method ();

}

fn main () {

let x = 5u8 ;

let y = "Hello" .to_string ();

do_something_u8 (x);

do_something_string (y);

}
```

Powyższe ma wielką zaletę: statyczna wysyłka umożliwia wstawianie wywołań funkcji online, ponieważ odbiorca jest znany w czasie kompilacji, a wstawianie online jest kluczem do dobrej optymalizacji. Wysyłanie statyczne jest szybkie, ale ma wadę: „rozdęcie kodu” w wyniku wielokrotnego wstawiania kopii tej samej funkcji do pliku binarnego, po jednej dla każdego typu. Ponadto kompilatory nie są doskonałe i mogą „optymalizować” kod, spowalniając go. Na przykład funkcje wstawiane online w niespokojny sposób zawyżają pamięć podręczną instrukcji (a pamięć podręczna rządzi wszystkim wokół nas). Dlatego # [inline] i # [inline (always)] należy używać ostrożnie i jest to jeden z powodów, dla których użycie dynamicznej wysyłki jest czasami bardziej wydajne. Jednak częstym przypadkiem jest to, że wysyłka statyczna jest bardziej wydajna. Można mieć funkcję cienkiego opakowania wysyłaną statycznie, wykonując dynamiczną wysyłkę, ale nie odwrotnie, tj.; wywołania statyczne są bardziej elastyczne. Dlatego biblioteka standardowa stara się być wysyłana dynamicznie, kiedy tylko jest to możliwe.

Dynamiczna wysyłka

Rust zapewnia dynamiczną wysyłkę poprzez funkcję zwaną „obiektami cech”. Obiekty cech, takie jak & Foo lub Box <Foo> , są normalnymi wartościami przechowującymi wartość dowolnego typu, który implementuje daną cechę, przy czym dokładny typ można określić tylko w czasie wykonywania. Obiekt cechy można uzyskać ze wskaźnika do określonego typu, który implementuje cechę, konwertując ją (np. & x na & Foo) lub stosując koercję (np. używając & x jako argumentu funkcji, która otrzymuje & Foo). Te wymuszenia i konwersje działają również dla wskaźników takich jak & mut T a & mut Foo i Box <T> a Box <Foo> , ale to na razie wszystko. Przymusy i nawrócenia są identyczne. Operację tę można postrzegać jako „usunięcie” wiedzy kompilatora o konkretnym typie wskaźnika, dlatego obiekty cech są czasami określane jako usuwanie typu. Wracając do poprzedniego przykładu, możemy użyć tej samej cechy do dynamicznej wysyłki z konwersją obiektów cech:

```
# trait Foo { fn method (& self ) -> String ; }

# impl Foo for u8 { fn method (& self ) -> String { format! ( "u8: {}" , *

self )}}

# impl Foo for String { fn method (& self ) -> String { format! ( "string:

{}" , * self )}}

fn do_something (x: & Foo) {

x.method ();
```

```
}
fn main () {
let x = 5u8 ;
do_something (& x as & Foo);
}
```

lub pod przymusem:

```
# trait Foo { fn method (& self ) -> String ; }
# impl Foo for u8 { fn method (& self ) -> String { format! ( "u8: {}" , *
self )}}
# impl Foo for String { fn method (& self ) -> String { format! ( "string:
{}" , * self )}}
fn do_something (x: & Foo) {
x.method ();
}
fn main () {
let x = "Hello" .to_string ();
do_something (& x);
}
```

Funkcja, która otrzymuje obiekt cechy, nie jest wyspecjalizowana dla każdego z typów implementowanych przez Foo: generowana jest tylko jedna kopia, co czasami (ale nie zawsze) skutkuje mniejszą inflacją kodu. Jednak dynamiczna wysyłka odbywa się kosztem wymagania najwolniejszych wywołań funkcji wirtualnych, co skutecznie blokuje wszelkie możliwości wstawiania online i **zzwiązanych z tym optymalizacji.**

Dlaczego wskaźniki?

Rust, w przeciwieństwie do wielu zarządzanych języków, nie umieszcza rzeczy za domyślnymi wskaźnikami, co powoduje, że typy mają różne rozmiary. Znajomość rozmiaru wartości w czasie kompilacji jest ważna dla takich rzeczy, jak przekazywanie jej jako argumentu do funkcji, przenoszenie jej na stos i przydzielanie (i cofanie alokacji) dla niej miejsca w kopcu do przechowywania. Dla Foo musielibyśmy mieć wartość, która mogłaby być mniejsza niż String (24 bajty) lub u8 (1 bajt), jak również każdy inny typ, który Foo może zaimplementować w zależnych skrzynkach (dowolna liczba bajtów). Nie ma sposobu, aby zagwarantować, że ten drugi przypadek będzie działał, jeśli wartości nie są przechowywane we wskaźniku, ponieważ te inne typy mogą mieć dowolny rozmiar. Umieszczenie wartości za wskaźnikiem oznacza, że rozmiar wartości nie ma znaczenia, gdy rzucamy obiektem cechy, tylko rozmiar samego wskaźnika.

Reprezentacja

Metody cechy można wywoływać na obiekcie cechy za pośrednictwem rejestru wskaźnika funkcji, tradycyjnie zwanego „vtable” (utworzonego i zarządzanego przez kompilator). Obiekty cech są jednocześnie proste i złożone: ich reprezentacja i dystrybucja są dość proste, ale istnieje kilka dość rzadkich komunikatów o błędach i kilka zaskakujących zachowań do odkrycia. Zacznijmy od najprostszego, reprezentacji obiektu cechy w czasie wykonywania. Moduł std :: raw zawiera struktury z dystrybucjami, które są tak samo skomplikowane, jak typy wbudowane, w tym obiekty cech :

```
# mod foo {

pub struct TraitObject {

pub data: * mut (),

pub vtable: * mut (),

}

#}
```

Oznacza to, że obiekt cechy, taki jak & Foo, składa się ze wskaźnika „data” i wskaźnika „vtable”. Wskaźnik danych wskazuje na dane (nieznanego typu T) przechowywane przez obiekt cechy, a wskaźnik vtable wskazuje na vtable (tabelę metod wirtualnych) („tabelę metod wirtualnych”) odpowiadającą implementacji Foo dla T . Vtable jest zasadniczo strukturą wskaźników funkcji, wskazującą na konkretny segment kodu maszynowego dla każdej implementacji metody. Wywołanie metody takiej jak object_trait.method () zwróci prawidłowy wskaźnik z tabeli vtable, a następnie wykona dynamiczne wywołanie tego wskaźnika. Na przykład: struct FooVtable {

```
destroyer: fn (* mut ()),

size: usize,

alignment: usize,

method: fn (* const ()) -> String,

}

// u8:

fn call_method_in_u8 (x: * const ()) -> String {

// the compiler guarantees that this function is only called

// with `x` pointing to a u8

let byte: & u8 = unsafe {& * (x as * const u8)};

byte.metodo ()

}

static Foo_vtable_para_u8: FooVtable = FooVtable {

destroyer: / * compiler magic * /,

size: 1,

alignment: 1,
```

```
// conversion to a function pointer
method: call_method_in_u8 as fn (* const ()) -> String,
};
// String:
fn call_method_in_String (x: * const ()) -> String {
// the compiler guarantees that this function is only called
// with `x` pointing to a String
let string: & String = unsafe {& * (x as * const String)};
string.metodo ()
}
static Foo_vtable_para_String: FooVtable = FooVtable {
destroyer: / * compiler magic * /,
// values for a 64-bit computer, divide them in half for a 32-bit
size: 24,
alignment: 8,
method: call_method_in_String as fn (* const ()) -> String,
};
```

Pole destruktora w każdym vtable wskazuje na funkcję, która wyczyści wszystkie zasoby typu vtable: dla u8 jest to trywialne, ale dla String zwolni pamięć. Jest to konieczne, aby przejąć obiekty cech, takie jak Box <Foo> , które muszą wyczyścić zarówno mapowanie Box, jak i typ wewnętrzny, gdy wyjdą poza zakres. Pola rozmiaru i wyrównania przechowują rozmiar usuniętego typu i jego wymagania dotyczące wyrównania; są one zasadniczo nieużywane w tej chwili, ponieważ informacje są osadzone w niszczarce, ale będą używane w przyszłości, ponieważ obiekty cech są stopniowo uelastyczniane. Załóżmy, że mamy pewne wartości, które implementują Foo . Wyraźny sposób konstruowania i używania obiektów z cechą Foo może wyglądać trochę jak (ignorując niespójności między typami: i tak są to wskaźniki):

```
let a: String = "foo" .to_string ();
let x: u8 = 1;
// let b: & Foo = & a;
let b = TraitObject {
// store the data
data: & a,
// store the methods
vtable: & Foo_vtable_para_String
```

```
};
// let y: & Foo = x;
let y = TraitObject {
// store the data
data: & x,
// store the methods
vtable: & Foo_vtable_para_u8
};
// b.metodo ();
(b.vtable.metodo) (b.data);
// and.method ();
(y.vtable.metodo) (y.data);
```

Bezpieczeństwo obiektu

Nie każdej cechy można użyć do stworzenia obiektu cechy. Na przykład wektory implementują Clone , ale jeśli spróbujemy stworzyć obiekt cechy:

```
let v = vec! [1, 2, 3];
let o = & v as & Clone;
```

Otrzymujemy błąd:

```
błąd: nie można przekonwertować na obiekt cechy, ponieważ cecha `core :: clone ::
Klon` nie jest bezpieczny obiektowo [E0038]
niech o = & v jako & Clone;
^ ~
uwaga: cecha nie może wymagać tego `Self: Sized`
niech o = & v jako & Clone;
^ ~
```

Błąd mówi, że Clone nie jest „bezpieczny obiektowo”. Do tworzenia obiektów cech można używać tylko bezpiecznych cech lub obiektów. Cecha jest bezpieczna dla obiektów, jeśli oba warunki są spełnione:

cecha nie wymaga Self: Size

wszystkie jego metody są bezpieczne dla obiektów

Co zatem sprawia, że metoda jest bezpieczna dla obiektów? Każda metoda powinna wymagać, aby Self: Size lub wszystkie z poniższych:

nie może mieć żadnych parametrów typu

nie powinieneś używać Self

Uff! Jak widzimy, prawie wszystkie te reguły mówią o Self . Dobrą intuicją byłoby „z wyjątkiem szczególnych okoliczności, jeśli twoja metoda cech używa Self , nie jest bezpieczna dla obiektów”.

Domknięcia

Czasami warto zawinąć funkcję i jej wolne zmienne, aby uzyskać lepszą przejrzystość i możliwość ponownego użycia. Wolne zmienne, których można użyć, pochodzą z zakresu zewnętrznego i są „zamknięte”, gdy są używane w funkcji. Stąd nazwa „zamknięcie”. Rust zapewnia bardzo dobrą implementację, jak zobaczymy poniżej.

Składnia

Zamknięcia wyglądają tak:

```
let sum_one = | x: i32 | x + 1 ;

assert_eq! ( 2 , sum_one ( 1 ));
```

Tworzymy link do zmiennej sum_one i przypisujemy ją do domknięcia. Argumenty zamknięcia przechodzą między kreskami (|), a ciało jest wyrażeniem, w tym przypadku x + 1 . Pamiętaj, że {} jest wyrażeniem, więc możemy również mieć domknięcia wielowierszowe:

```
let sum_two = | x | {

let mut result: i32 = x;

result + = 1 ;

result + = 1 ;

Outcome

};

assert_eq! ( 4 , sum_two ( 2 ));
```

Zauważysz kilka rzeczy dotyczących domknięć, które różnią się nieco od zwykłych funkcji zdefiniowanych za pomocą fn . Po pierwsze, nie musimy zapisywać typów argumentów i zwracanych wartości. Możemy:

```
let sum_one = | x: i32 | -> i32 {x + 1 };

assert_eq! ( 2 , sum_one ( 1 ));
```

Ale nie musimy. Dlaczego? Zasadniczo zostało to wdrożone w ten sposób ze względów ergonomicznych. Podczas gdy określanie pełnego typu dla nazwanych funkcji jest przydatne do takich rzeczy, jak dokumentacja i wnioskowanie o typie, pełny podpis w zamknięciach jest rzadko dokumentowany, ponieważ prawie zawsze są one anonimowe i nie powodują problemów typu błąd. - odległość, jaką może spowodować wnioskowanie w nazwanych funkcjach. Druga składnia jest podobna, ale nieco inna. Dodałem tutaj spacje dla łatwego porównania:

```
fn sum_one_v1 (x: i32 ) -> i32 {x + 1 }

let sum_one_v2 = | x: i32 | -> i32 {x + 1 };

let sum_one_v3 = | x: i32 | x + 1 ;
```

Małe różnice, ale są podobne.

Zamknięcia i jego okolice

Środowisko dla zamknięcia może zawierać łącza do zmiennej zakresu, która je opakowuje oprócz zmiennych lokalnych i parametrów. W ten sposób:

let num = 5 ;

let sum_num = | x: i32 | x + num;

assert_eq! (10 , sum_num (5));

Zamknięcie suma_num odnosi się do łącza let w swoim zakresie: num . Dokładniej, pożycz link. Jeśli zrobimy coś, co jest w konflikcie z tym linkiem, otrzymamy taki błąd:

let mut num = 5;

let sum_num = | x: i32 | x + num;

let y = & mut num;

Co zawodzi z:

błąd: nie można pożyczyć „num” jako zmiennego, ponieważ jest również pożyczone jako

niezmienny

niech y = & mut liczba;

^ ~~

uwaga: poprzednie zapożyczenie `num` występuje tutaj z powodu użycia w zamknięciu; zmienna pożyczka zapobiega kolejnym przesunięciom lub zmiennym pożyczkom `num` aż do zakończenia pożyczki

niech suma_num = | x | x + liczba;

^ ~~~~~~~~~~

uwaga: poprzednie wypożyczenie kończy się tutaj

fn main () {

let mut num = 5;

let sum_num = | x | x + num;

let y = & mut num;

}

^

Nieco rozwlekły błąd, ale równie przydatny! Jak mówi, nie możemy pożyczyć zmiennej pożyczki w num, ponieważ zamknięcie już ją pożycza. Jeśli pozostawimy zamknięcie poza zakresem, możliwe jest:

let mut num = 5 ;

{

```
let sum_num = | x: i32 | x + num;

} // sum_num goes out of scope, the loan ends here

let y = & mut num;
```

Jeśli jednak wymaga tego twoje zamknięcie, Rust zajmie przynależność i przeniesie środowisko. Poniższe nie działa:

```
let nums = vec! [1, 2, 3];

let toma_nums = || nums;

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

We get this error:

```
note: `nums` moved into closure environment here because it has type

`[closure (()) -> collections :: vec :: Vec]`, which is non-copyable

let toma_nums = || nums;

^ ~~~~~~
```

Vec <T> ma przynależność do swojej treści i dlatego odwołując się do niej w ramach naszego domknięcia, musimy przyjąć przynależność nums . To tak samo, jak gdybyśmy podali liczby jako argument funkcji, która przyjęłaby jej członkostwo.

Zamknięcia przesuwają się

Możemy wymusić, aby nasze zamknięcie przynależy do jego środowiska za pomocą słowa kluczowego move :

```
let num = 5 ;

let take_name_num = move | x: i32 | x + num;
```

Teraz, nawet jeśli obecne jest słowo kluczowe move, zmienne mają normalną semantykę. W tym przypadku 5 implementuje Copy , aw konsekwencji num_take przejmuje członkostwo w kopii num . Jaka jest różnica?

```
let mut num = 5 ;

{

let mut sum_num = | x: i32 | num + = x;

sum_num ( 5 );

}

assert_eq! ( 10 , num);
```

W tym przypadku nasze domknięcie przyjęło zmienne odwołanie do num , a kiedy wywołamy sum_num , wyciszy podstawową wartość, tak jak się spodziewaliśmy. Musimy również zadeklarować sum_num jako mut , ponieważ mutujemy jego środowisko. Jeśli zmienimy to na ruch zamknięcia , jest inaczej:

```
let mut num = 5 ;

{

let mut sum_num = move | x: i32 | num + = x;

sum_num ( 5 );

}

assert_eq! ( 5 , num);

aser_eq! ( 5 , liczba);
```

Dostajemy tylko 5. Zamiast brać zmienną pożyczkę na nasz numer, bierzemy nieruchomość na kopię. Innym sposobem myślenia o ruchu zamknięć jest: dostarczają one zamknięciu własnego rekordu aktywacji. Bez move zamknięcie można powiązać z rekordem aktywacji, który je utworzył, podczas gdy ruch zamknięcia jest samowystarczalny. Co oznacza na przykład, że generalnie nie można zwrócić z funkcji domknięcia nieruchomego. Ale zanim zaczniemy mówić o otrzymywaniu domknięć jako parametrów i używaniu ich jako wartości zwracanych, powinniśmy porozmawiać trochę więcej o ich implementacji. Podobnie jak systemowy język programowania Rust, daje ci mnóstwo kontroli nad tym, co robi twój kod, a domknięcia nie różnią się niczym.

Implementacja zamknięć

Implementacja domknięć w Rust jest trochę inna niż w innych językach. W Rust domknięcia są faktycznie alternatywną składnią cech. Zanim przejdziesz dalej, musisz przeczytać rozdział o cechach, a także rozdział o przedmiotach z cechami . Czy już je przeczytałeś? Doskonały. Klucz do zrozumienia, jak działają domknięcia, jest trochę dziwny: użycie () do wywołania funkcji, na przykład foo () , jest operatorem przeciążenia. Wychodząc z tego założenia, wszystko inne pasuje. W Rust wykorzystujemy system cech do przeciążania operatorów. Funkcje wywoływania nie różnią się. Istnieją trzy cechy, które możemy przeciążyć:

```
# mod foo {

pub trait Fn <Args>: FnMut <Args> {

extern "rust-call" fn call (& self, args: Args) -> Self :: Output;

}

pub trait FnMut <Args>: FnOnce <Args> {

extern "rust-call" fn call_mut (& mut self, args: Args) -> Self ::

Output;

}

pub trait FnOnce <Args> {

type Output;

extern "rust-call" fn call_once (self, args: Args) -> Self :: Output;

}

#}
```

Zauważysz kilka różnic między tymi cechami, ale dużą jest jaźń:

Fn otrzymuje & self , FnMut bierze & mut self , a FnOnce otrzymuje self . Powyższe obejmuje wszystkie trzy typy ja poprzez zwykłą składnię wywołania metody. Ale zostały podzielone na trzy cechy, a nie tylko jedną. Daje nam to dużą kontrolę nad rodzajem zamknięć, które możemy otrzymać. Składnia || {} to alternatywna składnia dla tych trzech cech. Rust wygeneruje strukturę dla otoczenia, zaimplementuje odpowiednią cechę, a następnie zrobi z niej użytek.

Odbieranie domknięć jako argumentów

Teraz, gdy wiemy, że domknięcia są cechami, wiemy, jak je akceptować i zwracać: tak jak każdą inną cechę! Powyższe oznacza również, że mamy do wyboru wysyłkę statyczną lub dynamiczną. Najpierw stwórzmy funkcję, która odbiera coś, co można wywołać, wykonuje na nim wywołanie, a następnie zwraca wynik:

```
fn call_with_one <F> (some_closure: F) -> i32

where F: Fn ( i32 ) -> i32 {

some_closure ( 1 )

}

let answer = call_with_one (| x | x + 2 );

assert_eq! ( 3 , answer);
```

Mijamy nasze zamknięcie, | x | x + 2 , do call_with_one . call_with_one robi to, co sugeruje: wywołuje zamknięcie, podając jako argument 1.

Przyjrzyjmy się bardziej szczegółowo podpisowi call_with_one:

```
fn call_with_one <F> (some_closure: F) -> i32

# where F: Fn ( i32 ) -> i32 {

# some_closure ( 1 )}
```

Ponieważ Fn jest cechą, możemy ją ograniczyć do naszego rodzaju. W tym przypadku nasze zamknięcie otrzymuje i32 i zwraca i32 , dlatego ogólny limit, którego używamy, to Fn (i32) -> i32 . Jest jeszcze jedna kluczowa kwestia: ponieważ ograniczamy generyczny za pomocą cechy, wywołanie zostanie zmonomorfizowane, a co za tym idzie, będziemy wykonywać statyczną wysyłkę w zamknięciu. To super fajne. W wielu językach domknięcia są z natury przypisane z kopca i prawie zawsze będą wymagały dynamicznej wysyłki. W Rust możemy przypisać środowisko naszych zamknięć ze stosu, a także statycznie wysłać wywołanie. Dzieje się tak dość często w przypadku iteratorów i ich adapterów, które otrzymują domknięcia jako argumenty. Oczywiście, jeśli zależy nam na dynamicznej wysyłce, też możemy ją mieć. Obiekt cechy, jak zwykle, obsługuje ten przypadek:

```
fn call_with_one (some_closure: & Fn ( i32 ) -> i32 ) -> i32 {

some_closure ( 1 )

}

let answer = call_with_one (& | x | x + 2 );

assert_eq! ( 3 , answer);
```

Otrzymujemy teraz obiekt cechy, an & Fn . I musimy odnieść się do naszego zamknięcia, kiedy przekazujemy je do call_with_one , dlatego używamy & || .

Wskaźniki i domknięcia funkcji

Wskaźnik funkcji jest rodzajem domknięcia, które nie ma środowiska. W konsekwencji możemy przekazać wskaźnik funkcji do dowolnej funkcji, która otrzyma domknięcie jako argument:

```
fn call_with_one (some_closure: & Fn ( i32 ) -> i32 ) -> i32 {

some_closure ( 1 )

}

fn sum_one (i: i32 ) -> i32 {

i + 1

}

let f = sum_one;

let answer = call_with_one (& f);

assert_eq! ( 2 , answer);
```

W poprzednim przykładzie nie potrzebujemy ściśle zmiennej pośredniej f, działa również nazwa funkcji:

```
let answer = call_with_one (& sum_one);
```

Powracające zamknięcia

Kod w stylu funkcjonalnym bardzo często zwraca domknięcia w różnych sytuacjach. Jeśli spróbujesz zwrócić zamknięcie, możesz popełnić błąd. Na początku może się to wydawać dziwne, ale później zrozumiemy. Prawdopodobnie próbowałbyś zwrócić zamknięcie z takiej funkcji:

```
fn factory () -> (Fn (i32) -> i32) {

let num = 5;

| x | x + num

}

let f = factory ();

let answer = f (1);

assert_eq! (6, answer);
```

Powyższe generuje te długie, ale powiązane błędy:

```
error: the trait `core :: marker :: Sized` is not implemented for the type

`core :: ops :: Fn (i32) -> i32` [E0277]

fn factory () -> (Fn (i32) -> i32) {
```

^ ~~~~~~~~~~~~~~~

note: `core :: ops :: Fn (i32) -> i32` does not have a constant size known

at compile-time

fn factory () -> (Fn (i32) -> i32) {

^ ~~~~~~~~~~~~~~~

error: the trait `core :: marker :: Sized` is not implemented for the type`

core :: ops :: Fn (i32) -> i32` [E0277]

let f = factory ();

^

note: `core :: ops :: Fn (i32) -> i32` does not have a constant size known at compile-time

let f = factory ();

^

Aby zwrócić coś z funkcji, Rust musi znać rozmiar zwracanego typu. Ale ponieważ Fn jest cechą, może to być różne rzeczy o różnych rozmiarach: wiele typów może implementować Fn . Łatwym sposobem na określenie rozmiaru czegoś jest odwołanie się do tego, ponieważ odniesienia mają znany rozmiar. Zamiast powyższego możemy napisać:

fn factory () -> & (Fn (i32) -> i32) {

let num = 5;

| x | x + num

}

let f = factory ();

let answer = f (1);

assert_eq! (6, answer);

But we get another error:

error: missing lifetime specifier [E0106]

fn factory () -> & (Fn (i32) -> i32) {

^ ~~~~~~~~~~~~~~~

Dobrze. Ponieważ mamy referencję, musimy zapewnić dożywotnią. ale nasza funkcja fabryka () nie otrzymuje żadnych argumentów i dlatego uniknięcie czasów życia nie jest w tym przypadku możliwe. Jakie mamy opcje? Spróbujmy „statycznie:

fn factory () -> & 'static (Fn (i32) -> i32) {

let num = 5;

| x | x + num

```
}

let f = factory ();

let answer = f (1);

assert_eq! (6, answer);
```

Ale dostajemy kolejny błąd:

```
error: mismatched types:

expected `& 'static core :: ops :: Fn (i32) -> i32`,

found `[closure @: 7: 9: 7:20]`

(expected & -ptr,

found closure) [E0308]

| x | x + num

^ ~~~~~~~~~~
```

Błąd mówi nam, że nie mamy & 'static Fn (i32) -> i32 , ale [closure @ <anon>: 7: 9: 7:20] . Chwileczkę, co? Ponieważ nasze zamknięcie generuje własną strukturę dla środowiska, a także implementację dla Fn , FnMut i FnOnce , te typy są anonimowe. Istnieją tylko dla tego zamknięcia. To dlatego Rust wyświetla je jako zamknięcie @ <anon> zamiast jakiejś automatycznie wygenerowanej nazwy. Błąd mówi również o typie zwracanym, który ma być referencją, ale to, co próbujemy zwrócić, nie jest. Co więcej, nie możemy bezpośrednio przypisać obiektowi „statycznego” czasu życia. Dlatego zastosujemy inne podejście i zwrócimy „obiekt cechy”, zawijając Fn w pudełko. Poniższe prawie działa::

```
fn factory () -> Box i32> {

let num = 5;

Box :: new (| x | x + num)

}

# fn main () {

let f = factory ();

let answer = f (1);

assert_eq! (6, answer);

#}
```

Jest jeszcze jeden ostatni problem:

```
error: closure may outlive the current function, but it borrows `num`,

which is owned by the current function [E0373]

Box :: new (| x | x + num)

^ ~~~~~~~~~~
```

Cóż, jak omówiliśmy wcześniej, domknięcia pożyczają swoje środowisko. I w tym przypadku nasze środowisko opiera się na 5 przypisanym ze stosu, zmiennej num . Dzięki temu pożyczka posiada dożywotni rekord aktywacji. Jeśli to zamknięcie zostanie zwrócone, wywołanie funkcji może się zakończyć, rekord aktywacji zniknie, a nasze zamknięcie będzie przechwytywaniem śmieciowego środowiska pamięci! Dzięki ostatniej poprawce możemy sprawić, by to działało:

```
fn factory () -> Box < Fn ( i32 ) -> i32 > {

let num = 5 ;

Box :: new (move | x | x + num)

}

# fn main () {

let f = factory ();

let answer = f ( 1 );

assert_eq! ( 6 , answer);

#}
```

Dokonując wewnętrznego zamknięcia ruchem Fn, stworzyliśmy nowy rekord aktywacji dla naszego zamknięcia. Zapakowując go w Pudełko , zapewniliśmy mu znany rozmiar, co pozwala uniknąć naszego rekordu aktywacji.