O'REILLY

Wydanie II

Programowanie w języku Rust

Wydajność i bezpieczeństwo

Jim Blandy
Jason Orendorff
Leonora Tindall

Tytuł oryginału: Programming Rust: Fast, Safe Systems Development, 2nd Edition

Tłumaczenie: Piotr Rajca na podstawie "Programowanie w języku Rust. Wydajność I bezpieczeństwo" w tłumaczeniu Adama Bochenka i Krzysztofa Sawki

ISBN: 978-83-283-9525-1

© 2022 Helion S.A.

Authorized Polish translation of the English *Programming Rust 2E* ISBN 9781492052593
© 2021 Jim Blandy, Leonora F.S. Tindall, Jason Orendorff.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/prrus2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Wstęp 15

1. Programiści systemowi mogą mieć fajne zabawki 19

Rust zdejmuje ciężar z naszych barków 20
Programowanie współbieżne zostaje ujarzmione 21
A na dodatek Rust wciąż jest szybki 22
Rust ułatwia współpracę 22

2. Pierwsze spotkanie z Rustem 24

rustup i Cargo 25
Funkcje w języku Rust 27
Pisanie i uruchamianie testów 29
Obsługa argumentów wiersza poleceń 30
Prosty serwer WWW 34
Programowanie współbieżne 40
- Czym jest zbiór Mandelbrota? 41
- Parsowanie argumentów wiersza poleceń 45
- Odwzorowanie pikseli na liczby zespolone 47
- Rysowanie zbioru 49
- Zapis obrazu do pliku 50
- Program Mandelbrota działający współbieżnie 51
- Uruchomienie programu 57
- Przezroczyste bezpieczeństwo 58

Narzędzia systemów plików i wiersza poleceń 58
- Interfejs wiersza poleceń 59
- Odczyt i zapis plików 61
- Znajdowanie i zastępowanie 62

Kup ksi k Pole ksi k

3. **Typy proste** 65
Typy numeryczne o ustalonej długości 68
Typy całkowite 68
Sprawdzane, przenoszące, nasycające i przepełniające operacje arytmetyczne 72
Typy zmiennoprzecinkowe 73
Typ logiczny 76
Typ znakowy 76
Krotki 78
Typy wskaźnikowe 80
Referencje 80
Pudełka 81
Wskaźniki niechronione 81
Tablice, wektory i podzbiory 82
Tablice 82
Wektory 83
Podzbiory 86
Typ String 88
Literały łańcuchowe 88
Łańcuchy bajtów 89
Łańcuchy znaków w pamięci 90
Typ String 91
Podstawowe cechy typu String 92
Inne typy znakowe 93
Nazwy zastępcze typów 93
Co dalej? 93

4. **Reguła własności i przenoszenie własności** 94
Reguła własności 96
Przenoszenie własności 100
Więcej operacji związanych z przeniesieniem własności 105
Przeniesienie własności a przepływ sterowania 107
Przeniesienie własności a struktury indeksowane 107
Typy kopiowalne — wyjątki od reguł przenoszenia własności 110
Rc i Arc: własność współdzielona 113

5. **Referencje** 116
Referencje do wartości 117
Stosowanie referencji 120
Referencje Rusta kontra referencje C++ 120
Referencje a operacja przypisania 121
Referencje do referencji 122
Porównywanie referencji 123

Kup ksi k Pole ksi k

Referencje nigdy nie są puste 123
Referencje do wyrażeń 124
Referencje do podzbiorów i zestawów metod 124
Bezpieczeństwo referencji 125
Referencja do zmiennej lokalnej 125
Przekazywanie referencji jako argumentów funkcji 128
Przekazywanie referencji do funkcji 130
Zwracanie referencji 131
Struktura zawierająca referencje 132
Odrębny cykl życia 134
Pomijanie parametrów cyklu życia 136
Referencje współdzielone kontra mutowalne 137
Walka ze sztormem na morzu obiektów 144

6. Wyrażenia 146
Język wyrażeń 146
Priorytety i łączność operatorów 147
Bloki kodu i średniki 150
Deklaracje 151
if i match 153
if let 154
Pętle 155
Sterowanie przepływem w pętlach 157
Wyrażenie return 158
Analiza przepływu sterowania 159
Wywołania funkcji i metod 160
Pola i elementy 161
Operatory referencji 163
Operatory arytmetyczne, bitowe, porównania i logiczne 163
Przypisanie 164
Rzutowanie typów 165
Domknięcia 166
Co dalej? 166

7. Obsługa błędów 167
Błąd panic 167
Odwinięcie stosu 168
Przerywanie procesu 169
Typ Result 170
Przechwytywanie błędów 170
Nazwy zastępcze typu Result 172
Wyświetlanie informacji o błędach 172

Kup ksi k Pole ksi k

Propagacja błędów 174
Jednoczesna obsługa błędów różnych typów 175
Błędy, które nie powinny się zdarzyć 177
Ignorowanie błędów 178
Obsługa błędów w funkcji main() 178
Definiowanie własnego typu błędu 180
Co daje nam typ Result? 181

8. Paczki i moduły 182
Paczki 182
Edycje 185
Profile budowania 186
Moduły 187
Moduły zagnieżdżone 188
Umieszczanie modułów w oddzielnych plikach 190
Ścieżki i importy 192
Standardowe preludium 195
Publiczne deklaracje use 195
Publiczne pola struktur 195
Stałe i zmienne statyczne 196
Zmiana programu w bibliotekę 197
Katalog src/bin 198
Atrybuty 200
Testy i dokumentacja 202
Testy integracyjne 204
Dokumentacja 205
Doc-testy 208
Definiowanie zależności 210
Wersje 211
Cargo.lock 212
Publikowanie paczek na stronie crates.io 213
Obszary robocze 215
Więcej fajnych rzeczy 216

9. Struktury 217
Struktury z polami nazywanymi 217
Struktury z polami numerowanymi 220
Struktury puste 221
Reprezentacja struktur w pamięci 221
Definiowanie metod w bloku impl 222
Przekazywanie self z użyciem typów Box, Rc lub Arc 225
Funkcje powiązane z typami 227

Kup ksi k
Pole ksi k

Powiązane stałe 228
Struktury generyczne 228
Struktury z parametrem cyklu życia 230
Dziedziczenie standardowych zestawów metod 231
Zmienność wewnętrzna 232

10. Typy wyliczeniowe i wzorce ... 236
Typy wyliczeniowe 237
Typy wyliczeniowe zawierające dane 239
Typ wyliczeniowy w pamięci 240
Większe struktury danych stosujące typy wyliczeniowe 241
Generyczne typy wyliczeniowe 243
Wzorce 245
Literały, zmienne i symbole wieloznaczne 247
Wzorce w postaci krotki lub struktury 249
Wzorce typu tablica i fragment 250
Wzorce z referencjami 251
Strażniki wzorców 253
Dopasowanie do wielu wartości 254
Wzorce @ 254
Gdzie używamy wzorców 255
Wypełnianie drzewa binarnego 256
Podsumowanie 258

11. Zestawy metod i typy generyczne ... 259
Stosowanie zestawów metod 261
Obiekt implementujący zestaw metod 262
Funkcje generyczne i parametry typów 264
Na co się zdecydować? 267
Definiowanie i implementacja zestawów metod 269
Metody domyślne 270
Implementacja zestawów metod dla istniejących już typów 272
Zestaw metod a słowo kluczowe Self 273
Rozszerzanie zestawu metod (dziedziczenie) 275
Funkcje powiązane z typami 276
W pełni kwalifikowana nazwa metody 277
Zestawy metod definiujące relacje między typami 278
Typy powiązane 279
Generyczny zestaw metod (czyli jak działa przeciążanie operatorów) 282
Impl Zestaw jako typ wyniku 283
Stałe powiązane 285
Inżynieria wsteczna ograniczeń 286
Zestawy metod u podstaw 289

Kup ksi k Pole ksi k

12. Przeciążanie operatorów ... 290
Operatory arytmetyczne i bitowe 291
Operatory jednoargumentowe 293
Operatory dwuargumentowe 294
Operatory przypisania złożonego 295
Test równości 297
Porównania szeregujące 300
Interfejsy Index i IndexMut 302
Inne operatory 304

13. Interfejsy narzędziowe ... 306
Drop 307
Sized 310
Clone 313
Copy 314
Deref i DerefMut 315
Default 318
AsRef i AsMut 320
Borrow i BorrowMut 321
From i Into 323
TryFrom i TryInto 326
ToOwned 327
Borrow i ToOwned w działaniu 328

14. Domknięcia ... 330
Przechwytywanie zmiennych 331
Domknięcia, które pożyczają wartość 332
Domknięcia, które przejmują własność 333
Typy funkcji i domknięć 334
Domknięcia a wydajność 336
Domknięcia a bezpieczeństwo 338
Domknięcia, które zabijają 338
FnOnce 338
FnMut 340
Copy i Clone dla domknięć 342
Funkcje zwrotne 343
Skuteczne korzystanie z domknięć 347

Kup ksi k Pole ksi k

15. Iteratory ... 349
Iterator i IntoIterator 350
Tworzenie iteratorów 352
Metody iter i iter_mut 352
Implementacje interfejsu IntoIterator 353
Funkcje from_fn i successors 355
Metody drain 356
Inne źródła iteratorów 357
Adaptery iteratorów 358
map i filter 358
filter_map i flat_map 361
flatten 363
take i take_while 364
skip i skip_while 365
peekable 366
fuse 367
Iteratory obustronne i rev 368
inspect 369
chain 370
enumerate 370
zip 371
by_ref 372
cloned i copied 373
cycle 373
Konsumowanie iteratorów 374
Proste agregaty: count, sum i product 374
max i min 375
max_by i min_by 375
max_by_key i min_by_key 376
Porównywanie sekwencji elementów 376
any i all 377
position, rposition oraz ExactSizeIterator 377
fold i rfold 378
try_fold i try_rfold 379
nth i nth_back 380
last 381
find, rfind i find_map 381
Tworzenie kolekcji: collect i FromIterator 382
Zestaw metod Extend 384
partition 384
for_each i try_for_each 385
Implementacja własnych iteratorów 386

Kup ksi k

Pole ksi k

16. Kolekcje 391
Przegląd kolekcji 392
Vec<T> 393
Dostęp do elementów 394
Iteracja 395
Zwiększanie i zmniejszanie wielkości wektora 396
Łączenie 400
Podział 400
Zamiana 403
Sortowanie i wyszukiwanie 403
Porównywanie podzbiorów 405
Elementy losowe 405
Reguły zapobiegające konfliktom w czasie iteracji 406
VecDeque<T> 407
BinaryHeap<T> 409
HashMap<K, V> i BTreeMap<K, V> 410
Entry 414
Iterowanie map 416
HashSet<T> i BTreeSet<T> 417
Iteracja zbioru 418
Kiedy równe wartości są różne 418
Operacje dotyczące całego zbioru 419
Haszowanie 420
Niestandardowe algorytmy haszujące 421
Kolekcje standardowe i co dalej? 423

17. Tekst i łańcuchy znaków 424
Podstawy Unicode 424
ASCII, Latin-1 i Unicode 425
UTF-8 425
Kierunek tekstu 427
Znaki (typ char) 427
Klasyfikacja znaków 427
Obsługa cyfr 429
Zmiana wielkości liter 430
Konwersja znaku do i z liczby całkowitej 430
Typy String i str 431
Tworzenie łańcuchów znaków 432
Prosta inspekcja 432
Dołączanie i wstawianie tekstu 433
Usuwanie i zastępowanie tekstu 435
Konwencje nazewnicze dotyczące wyszukiwania i iterowania 436

Kup ksi k Pole ksi k

Wyszukiwanie tekstu i wzorce 436
Wyszukiwanie i zamiana 437
Iterowanie tekstu 438
Obcinanie 440
Zmiana wielkości liter w łańcuchach 441
Konwersja tekstu do wartości innego typu 441
Konwersja wartości innego typu do tekstu 442
Tworzenie referencji innego typu 443
Tekst jako UTF-8 443
Tworzenie tekstu na podstawie danych UTF-8 444
Alokacja warunkowa 445
Łańcuchy znaków jako kolekcje generyczne 447
Formatowanie wartości 447
Formatowanie tekstu 449
Formatowanie liczb 450
Formatowanie innych typów 452
Formatowanie wartości z myślą o debugowaniu 453
Formatowanie i debugowanie wskaźników 454
Wiązanie argumentów za pomocą indeksu i nazwy 455
Dynamiczne definiowanie długości i precyzji 455
Formatowanie własnych typów 456
Użycie języka formatowania we własnym kodzie 458
Wyrażenia regularne 459
Podstawowe użycie wyrażeń regularnych 460
Wyrażenia regularne w trybie leniwym 461
Normalizacja 462
Rodzaje normalizacji 463
Biblioteka unicode-normalization 464

18. Operacje wejścia – wyjścia 465
Obiekty typu reader i writer 466
Obiekty typu reader 467
Buforowany obiekt typu reader 469
Przeglądanie tekstu 470
Pobieranie tekstu 473
Obiekty typu writer 473
Pliki 475
Wyszukiwanie 476
Inne rodzaje obiektów reader i writer 476
Dane binarne, kompresja i serializacja 478
Pliki i katalogi 480
OsStr i Path 480
Metody typów Path i PathBuf 481

Kup książkę Poleć książkę

Funkcje dostępu do systemu plików 483
Odczyt zawartości katalogu 484
Funkcje bezpośrednio związane z platformą 486
Obsługa sieci 487

19. Programowanie współbieżne 490
Podział i łączenie wątków 492
spawn i join 493
Obsługa błędów w różnych wątkach 495
Współdzielenie niemutowalnych danych przez różne wątki 496
Rayon 498
Zbiór Mandelbrota raz jeszcze 501
Kanały 502
Wysyłanie wartości 504
Odczyt wartości 507
Uruchomienie potoku 508
Cechy kanałów i ich wydajność 509
Bezpieczeństwo wątków: Send i Sync 511
Współpraca iteratora i kanału 514
Potoki i co dalej? 515
Stan współdzielony mutowalny 516
Czym jest muteks? 516
Mutex<T> 518
mut i Mutex 520
Dlaczego Mutex to nie zawsze dobry pomysł? 520
Zakleszczenie (deadlock) 521
Zatruty muteks 522
Kanały z wieloma nadawcami stosujące muteksy 522
Blokady odczytu/zapisu (RwLock<T>) 523
Zmienne warunkowe (Condvar) 524
Typy atomowe 525
Zmienne globalne 527
Rust i pisanie programów wielowątkowych 530

20. Programowanie asynchroniczne 531
Od kodu synchronicznego do asynchronicznego 533
Interfejs Future 534
Funkcje asynchroniczne i wyrażenia await 537
Wywoływanie funkcji asynchronicznych z kodu synchronicznego: block_on 539
Uruchamianie asynchronicznych zadań 542
Asynchroniczne instrukcje blokowe 546
Tworzenie funkcji asynchronicznych
z wykorzystaniem asynchronicznych instrukcji blokowych 549

Kup książkę Poleć książkę

Uruchamianie zadań asynchronicznych w pulach wątków 550
Czy operacje asynchroniczne implementują Send? 551
Wykonywanie długotrwałych obliczeń: yield_now i spawn_local 554
Porównanie różnych rozwiązań asynchronicznych 555
Rzeczywisty asynchroniczny klient HTTP 556
Asynchroniczny klient i serwer 557
Typy błędów i wyników 559
Protokół 559
Pobieranie danych od użytkownika: strumienie asynchroniczne 561
Wysyłanie pakietów 563
Pobieranie pakietów: więcej strumieni asynchronicznych 564
Funkcja main klienta 566
Funkcja main serwera 567
Obsługa połączeń z klientami: asynchroniczne muteksy 568
Tablica grup: muteksy synchroniczne 571
Grupy: kanały rozgłoszeniowe paczki tokio 572
Podstawowe wartości future i wykonawcy: kiedy warto ponownie sprawdzać wartość future? 575
Wywoływanie funkcji aktywujących: spawn_blocking 577
Implementacja funkcji block_on 579
Typ Pin i jego stosowanie 581
Dwa etapy życia danych typ future 581
Przypięte wskaźniki 585
Zestaw metod Unpin 587
Kiedy warto stosować kod asynchroniczny? 588

21. Makra ... 591
Podstawy 592
Rozwijanie makra 593
Niezamierzone skutki 595
Powtórzenia 596
Makra wbudowane 598
Debugowanie makr 600
Pisanie makra json! 601
Typy składników 602
Makra a rekurencja 605
Makra i zestawy metod 606
Zakres i higiena 608
Import i eksport makr 610
Unikanie błędów składniowych w procesie dopasowywania 612
macro_rules! i co dalej? 613

Kup książkę Poleć książkę

22. Kod niebezpieczny 615
Dlaczego niebezpieczny? 616
Bloki unsafe 617
Przykład: wydajny typ łańcucha znaków ASCII 619
Funkcje unsafe 621
Kod niebezpieczny czy funkcja niebezpieczna? 623
Działanie niezdefiniowane 623
Zestawy metod unsafe 626
Wskaźniki niechronione 628
Bezpieczne tworzenie dereferencji wskaźników niechronionych 631
Przykład: RefWithFlag 632
Wskaźniki dopuszczające wartość pustą 634
Rozmiary i rozmieszczanie typów 634
Operacje arytmetyczne na wskaźnikach 635
Wchodzenie do pamięci i wychodzenie z pamięci 636
Przykład: GapBuffer 640
Bezpieczeństwo błędów paniki w kodzie niebezpiecznym 646
Ponowna interpretacja pamięci z użyciem unii 647
Dopasowywanie unii 650
Pożyczanie unii 650

23. Funkcje obce 651
Wyszukiwanie wspólnych reprezentacji danych 652
Deklarowanie obcych funkcji i zmiennych 655
Korzystanie z funkcji i bibliotek 657
Interfejs niechroniony dla biblioteki libgit2 660
Interfejs bezpieczny dla biblioteki libgit2 666
Podsumowanie 676

Skorowidz 677

Kup ksi k Pole ksi k

ROZDZIAŁ 22.

Kod niebezpieczny

O nie myślcie o mnie,
Żem słaba jest niewiasta, tchórzliwa ogromnie,
Że się do gnuśnych liczę i że spokój lubię:
Dla dobrych jam ku szczęściu, dla podłych ku zgubie:
Człek tylko mnie podobien wart największej chwały[1].

— Eurypides, *Medea*

Tajemna radość programowania systemów wiąże się z faktem, że pod fasadą każdego bezpiecznego języka i ostrożnie zaprojektowanej abstrakcji kryje się szalejący huragan niezwykle niebezpiecznego języka maszynowego i kombinowania z bitami. Nie inaczej jest z językiem Rust.

Opisywany przez nas język gwarantuje, że programy są całkowicie automatycznie pozbawiane błędów pamięci i wyścigów do danych za pomocą typów, parametrów cyklu życia, sprawdzania wartości itd. Jednak taka forma automatycznego rozumowania ma swoje ograniczenia; istnieje wiele cennych technik, których język Rust nie uznaje za bezpieczne.

Kod niebezpieczny (ang. *unsafe code*) pozwala poinformować język Rust: „Teraz mam zamiar skorzystać z możliwości, których bezpieczeństwa nie jesteś w stanie zagwarantować”. Poprzez oznaczenie bloku lub funkcji jako niebezpieczne uzyskujesz, między innymi, możliwość wywoływania funkcji `unsafe` w bibliotece standardowej, korzystania ze wskaźników niechronionych i wywoływania funkcji napisanych w innych językach, np. C i C++. Stosowane są wszystkie typowe mechanizmy kontroli języka Rust: sprawdzanie typu, sprawdzanie cyklu życia i sprawdzanie wartości indeksów są realizowane normalnie. Kod niebezpieczny jedynie udostępnia mały zestaw dodatkowych funkcji.

Taka możliwość wykraczania poza bezpieczne granice pozwala implementować wiele podstawowych możliwości języka Rust w sposób przypominający to, jak języki C i C++ implementują własne biblioteki standardowe. To właśnie kod niebezpieczny umożliwia typowi `Vec` skuteczne zarządzanie swoim buforem, modułowi `std::io` komunikowanie się z systemem operacyjnym, a modułom `std::thread` i `std::sync` dostarczanie elementów współbieżności.

[1] Tłum. Jan Kasprowicz.

Kup książkę Poleć książkę

W tym rozdziale znajdziesz omówienie podstaw pracy z niebezpiecznymi funkcjami:

- Bloki `unsafe` wyznaczają granicę pomiędzy tradycyjnym, bezpiecznym kodem napisanym w języku Rust a kodem wykorzystującym rozwiązania niebezpieczne.
- Możesz zaznaczać funkcje jako niebezpieczne (`unsafe`), co stanowi ostrzeżenie dla elementów wywołujących o obecności dodatkowych kontraktów, które muszą być przestrzegane w celu uniknięcia działania niezdefiniowanego.
- Wskaźniki niechronione i ich metody umożliwiają nieograniczony dostęp do pamięci, a także budowanie struktur danych, które w innych okolicznościach są zabronione przez system typów Rusta. W odróżnieniu od referencji, które w Ruście są bezpieczne, lecz ograniczone, wskaźniki niechronione, co może potwierdzić każdy programista C lub C++, są narzędziem potężnym, lecz niebezpiecznym.
- Zrozumienie pojęcia działania niezdefiniowanego pomaga dostrzec, że jego konsekwencje mogą być znacznie poważniejsze niż otrzymywanie nieprawidłowych wyników.
- Niebezpieczne zestawy metod, analogicznie do funkcji `unsafe`, narzucają kontrakt, który musi być przestrzegany przez każdą implementację (a nie przez element wywołujący).

Dlaczego niebezpieczny?

Na początku książki pokazaliśmy napisany w C program, który przestaje działać w nieoczekiwany sposób, ponieważ nie jest w stanie przestrzegać jednej z reguł stanowiących standard tego języka. Możemy dokonać tego samego w języku Rust:

```
$ cat crash.rs
fn main() {
    let mut a: usize = 0;
    let ptr = &mut a as *mut usize;
    unsafe {
        *ptr.offset(3) = 0x7ffff72f484c;
    }
}
$ cargo build
    Compiling unsafe-samples v0.1.0
     Finished debug [unoptimized + debuginfo] target(s) in 0.44 secs
$ ../../target/debug/crash
crash: Error: .netrc file is readable by others.
crash: Remove password or make file unreadable by others.
Segmentation fault (core dumped)
$
```

Program ten „pożycza” mutowalne odniesienie do zmiennej lokalnej `a`, rzutuje je na wskaźnik niechroniony typu `*mut usize`, a następnie wykorzystuje metodę `offset` do utworzenia wskaźnika trzy słowa dalej w pamięci. Akurat w tamtym miejscu przechowywany jest adres zwrotny funkcji `main`. Program nadpisuje go i umieszcza na jego miejscu stałą, przez co powrót z pętli `main` skutkuje nieoczekiwanym zachowaniem. Błąd ten wynika z nieprawidłowego wykorzystania niebezpiecznych właściwości; w tym przypadku jest możliwość dereferencji wskaźników niechronionych.

Kup ksi k Pole ksi k

Właściwość niebezpieczna narzuca *kontrakt* — reguły, których Rust nie jest w stanie automatycznie wymuszać, ale które muszą być mimo to przestrzegane w celu uniknięcia *działania niezdefiniowanego*.

Kontrakt wykracza poza standardową kontrolę typów i parametrów cyklu życia, gdyż narzuca dodatkowe reguły specyficzne dla danej właściwości niebezpiecznej. Zazwyczaj Rust sam w sobie nie zawiera żadnych informacji o kontrakcie — jest on opisany w dokumentacji danej właściwości. Na przykład wskaźnik niechroniony typu zawiera kontrakt uniemożliwiający realizację dereferencji wskaźnika, który przekroczył punkt końcowy pierwotnej referencji. W naszym przykładzie wyrażenie `*ptr.offset(3) = …` stoi w sprzeczności z kontraktem. Jednak jak widać powyżej, program zostaje bez problemu skompilowany: mechanizmy kontrolne nie wykrywają tego naruszenia. Podczas korzystania z właściwości niebezpiecznych to na Tobie, jako programiście, ciąży odpowiedzialność za zapewnianie zgodności kodu z kontraktami.

Wiele właściwości zawiera reguły, których należy przestrzegać podczas prawidłowego korzystania, nie są to jednak kontrakty w przytaczanym znaczeniu, chyba że wśród konsekwencji ich łamania występuje działanie niezdefiniowane. Działanie niezdefiniowane to takie działanie, na które (zgodnie z założeniami języka Rust) kod nie powinien być nigdy wystawiony. Na przykład Rust „zakłada", że nie będziesz nadpisywać adresu zwrotnego wywołania funkcji za pomocą innej wartości. Kod zaliczający testy kontrolne i zgodny z kontraktami wykorzystywanych właściwości niebezpiecznych nie jest w stanie zrealizować takiej operacji. Nasz program narusza kontrakt wskaźnika niechronionego, mamy do czynienia z działaniem niezdefiniowanym i aplikacja zaczyna dziwnie działać.

Jeżeli Twój kod zaczyna działać w niezdefiniowany sposób, oznacza to, że złamałeś swoją część umowy z Rustem i nie będzie on przewidywał konsekwencji. Wydobywanie nieistotnych komunikatów o błędach z głębin bibliotek systemowych i zawieszanie się aplikacji to jedne z możliwych skutków; inną konsekwencją może być przekazanie intruzowi kontroli nad Twoim komputerem. Objawy mogą, bez żadnego ostrzeżenia, różnić się w zależności od wersji języka Rust. Czasami jednak działanie niezdefiniowane nie ma widocznych symptomów. Na przykład jeżeli funkcja `main` nie zostanie nigdy zwrócona (być może wywołuje ona `std::process::exit` w celu przedwczesnego przerwania programu), to uszkodzony adres zwrotny prawdopodobnie nie będzie miał żadnego znaczenia.

Możesz używać niebezpiecznych właściwości jedynie w bloku `unsafe` lub funkcji `unsafe` — obydwa elementy zostaną już niebawem omówione. W ten sposób zostaje utrudnione nieświadome korzystanie z tych właściwości: poprzez wymuszanie tworzenia bloku/funkcji `unsafe` Rust gwarantuje, że masz świadomość dodatkowych reguł, które muszą być przestrzegane przez kod.

Bloki unsafe

Blok `unsafe` przypomina tradycyjny blok Rusta poprzedzony słowem kluczowym `unsafe`, a jedyną różnicę stanowi fakt, że możesz w nim stosować właściwości niebezpieczne:

```
unsafe {
    String::from_utf8_unchecked(ascii)
}
```


Kup ksi k   Pole ksi k

Bez umieszczonego na początku bloku słowa kluczowego `unsafe` nie moglibyśmy korzystać z wyrażenia `from_utf8_unchecked`, które jest funkcją niebezpieczną. Poprzez umieszczenie jej w bloku `unsafe` możesz korzystać z tego kodu w dowolnym miejscu.

Podobnie jak w przypadku tradycyjnego bloku Rusta wartość bloku `unsafe` zależy od jego ostatniego wyrażenia lub wynosi `()`, jeżeli nie występuje wyrażenie. Zaprezentowane wcześniej wywołanie `String::from_utf8_unchecked` dostarcza wartość bloku.

Blok `unsafe` otwiera pięć dodatkowych możliwości:

- Możesz wywoływać funkcje `unsafe`. Każda z nich musi wyznaczać własny kontrakt w zależności od jej przeznaczenia.
- Możesz tworzyć dereferencje wskaźników niechronionych. Bezpieczny kod umożliwia przekazywanie wskaźników niechronionych oraz ich tworzenie poprzez konwersję referencji (a nawet liczb całkowitych), ale tylko kod niebezpieczny pozwala uzyskiwać dostęp do pamięci za ich pomocą. Wskaźniki niechronione (i ich bezpieczne stosowanie) zostały dokładnie omówione w podrozdziale „Wskaźniki niechronione".
- Można odwoływać się do pól typów `union`, co do których kompilator nie ma pewności, że zawierają prawidłowe wzorce bitów odpowiadające zadeklarowanym typom.
- Możesz uzyskiwać dostęp do mutowalnych zmiennych `static`. W punkcie „Zmienne globalne" w rozdziale 19. wyjaśniliśmy, że Rust nie ma pewności, kiedy wątki wykorzystują mutowalne zmienne `static`, dlatego ich kontrakt wymaga zagwarantowania prawidłowej synchronizacji dostępu.
- Możesz uzyskiwać dostęp do funkcji i zmiennych deklarowanych poprzez interfejs funkcji obcych. Są one uznawane za niebezpieczne nawet w przypadku niemutowalności, ponieważ widzi je kod napisany w innych językach, które nie muszą przestrzegać reguł bezpieczeństwa Rusta.

Ograniczenie właściwości niebezpiecznych do bloków `unsafe` nie oznacza, że masz związane ręce. Nic nie stoi na przeszkodzie, aby wstawić blok `unsafe` do kodu i przejść do następnego zadania. Główną zaletą tej techniki jest fakt, że programista koncentruje uwagę na kodzie, którego bezpieczeństwa Rust nie może zagwarantować:

- Nie wprowadzisz przypadkiem właściwości niebezpiecznych, aby odkryć później, że odpowiadasz za kontrakty, o których istnieniu nawet nie wiedziałeś.
- Blok `unsafe` wymaga szczególnej uwagi podczas oceny i weryfikacji kodu. Niektóre projekty zawierają nawet zautomatyzowane mechanizmy, które zaznaczają zmiany kodu wpływające na bloki `unsafe`.
- Planując rozpisywanie bloku `unsafe`, masz czas zastanowić się, czy realizowane zadanie rzeczywiście wymaga podjęcia takich środków. Jeżeli problem stanowi wydajność, to czy masz do dyspozycji mierniki dowodzące, że to naprawdę ona stanowi źródło zatoru? Być może istnieje skuteczny sposób rozwiązania problemu w bezpiecznym środowisku Rusta.

Kup ksi k Pole ksi k

Przykład: wydajny typ łańcucha znaków ASCII

Poniżej przedstawiamy definicję `Ascii`, typu łańcucha znaków sprawiającego, że jego zawartość jest zawsze prawidłowo kodowana w formacie ASCII. Typ ten wykorzystuje właściwość niebezpieczną w celu zapewnienia darmowej konwersji do typu `String`:

```
mod my_ascii {
    use std::ascii::AsciiExt; // Dla u8::is_ascii

    /// Łańcuch znaków zakodowany w formacie ASCII
    #[derive(Debug, Eq, PartialEq)]
    pub struct Ascii(
        // Tutaj musi być przechowywany wyłącznie poprawny tekst ASCII:
        // bajty od `0` do `0x7f`
        Vec<u8>
    );

    impl Ascii {
        /// Tworzy `Ascii` z tekstu ASCII umieszczonego w `bytes`. Zwraca
        /// błąd `NotAsciiError`, jeżeli w `bytes` występuje jakikolwiek znak
        /// niekodowany w formacie ASCII
        pub fn from_bytes(bytes: Vec<u8>) -> Result<Ascii, NotAsciiError> {
            if bytes.iter().any(|&byte| !byte.is_ascii()) {
                return Err(NotAsciiError(bytes));
            }
            Ok(Ascii(bytes))
        }
    }

    // W przypadku nieudanej konwersji zostaje zwrócony wektor, którego nie można było
    // przekonwertować. Należałoby zaimplementować `std::error::Error`; pominięto dla
    // zachowania zwięzłości
    #[derive(Debug, Eq, PartialEq)]
    pub struct NotAsciiError(pub Vec<u8>);

    // Bezpieczna, skuteczna konwersja zaimplementowana za pomocą kodu niebezpiecznego
    impl From<Ascii> for String {
        fn from(ascii: Ascii) -> String {
            // Jeżeli moduł ten jest pozbawiony błędów, okazuje się bezpieczny, gdyż
            // poprawny tekst ASCII stanowi jednocześnie poprawny kod UTF-8
            unsafe { String::from_utf8_unchecked(ascii.0) }
        }
    }
    ...
}
```

Kluczem do tego modułu jest definicja typu `Ascii`. Typ sam w sobie jest oznaczony jako `pub`, dzięki czemu staje się widoczny poza modułem `my_ascii`. Jednak element `Vec<u8>` typu *nie* jest publiczny, zatem jedynie moduł `my_ascii` może konstruować wartość `Ascii` lub odnosić się do jej elementu. W ten sposób mamy całkowitą kontrolę nad tym, co może się tu pojawiać. Dopóki konstruktory i metody publiczne gwarantują ciągłą poprawność nowo utworzonych wartości `Ascii`, pozostała część programu nie może naruszyć tej reguły. Rzeczywiście, konstruktor publiczny `Ascii::from_bytes` sprawdza ostrożnie podany wektor przed stworzeniem z niego wartości `Ascii`.

Kup ksi k Pole ksi k

Dla zachowania zwięzłości nie pokazujemy żadnych metod, można jednak wyobrazić sobie zestaw metod obsługujących tekst, które gwarantują, że wartości `Ascii` zawsze zawierają prawidłowo sformatowany kod ASCII, podobnie jak metody typu `String` zapewniają poprawność treści w kodowaniu UTF-8.

Układ ten umożliwia bardzo skuteczne implementowanie `From<Ascii>` dla typu `String`. Funkcja niebezpieczna `String::from_utf8_unchecked` przyjmuje wektor bajtowy i tworzy z niego typ `String` bez sprawdzania, czy jego zawartość ma poprawny format UTF-8 — kontrakt zrzuca odpowiedzialność za to na kod wywołujący. Na szczęście reguły wymuszane przez typ `Ascii` są dokładnie takie, jakich potrzebujemy do spełnienia kontraktu `from_utf8_unchecked`. W rozdziale 17., w punkcie „UTF-8", każdy blok tekstu ASCII jest również poprawnie sformatowanym tekstem UTF-8, zatem element `Vec<u8>` służy od razu jako bufor typu `String`.

Mając do dyspozycji te definicje, możemy zapisać:

```
use my_ascii::Ascii;

let bytes: Vec<u8> = b"ASCII and ye shall receive".to_vec();

// Wywołanie to nie wymaga alokacji ani kopii tekstu, lecz jedynie przeglądania
let ascii: Ascii = Ascii::from_bytes(bytes)
    .unwrap(); // Wiemy, że te wybrane bajty są prawidłowe

// Wywołanie to jest darmowe: brak alokacji, kopii czy przeglądania
let string = String::from(ascii);

assert_eq!(string, "ASCII, a zostanie wam dane");
```

Nie jest wymagany żaden blok `unsafe` podczas do korzystania z funkcji `Ascii`. Zaimplementowaliśmy bezpieczny interfejs za pomocą niebezpiecznych operacji i sprawiliśmy, że warunki ich kontraktów są spełniane wyłącznie w zależności od kodu modułu, a nie od zachowania jego użytkowników.

Funkcja `Ascii` stanowi jedynie opakowanie wektora `Vec<u8>` ukryte w module wymuszającym dodatkowe reguły wobec umieszczonej zawartości. Mamy tu do czynienia z tzw. *nowym typem* (ang. *newtype*), popularnym wzorcem w języku Rust. Dokładnie w taki sam sposób jest definiowany własny typ `String` języka Rust — jedyną różnicę stanowi fakt, że kodowanie treści ogranicza się do formatu UTF-8, a nie ASCII. W istocie definicja typu `String` w bibliotece standardowej wygląda następująco:

```
pub struct String {
    vec: Vec<u8>,
}
```

Na poziomie zbliżonym do kodu maszynowego, gdzie typy języka Rust nie mają znaczenia, nowy typ i jego element są tak samo reprezentowane w pamięci, zatem konstruowanie nowego typu nie wymaga żadnych instrukcji kodu maszynowego. W konstruktorze publicznym `Ascii::from_bytes` wyrażenie `Ascii(bytes)` uznaje, że reprezentacja wektora `Vec<u8>` będzie teraz przechowywać wartość `Ascii`. Na drodze analogii `String::from_utf8_unchecked` prawdopodobnie po wstawieniu nie wymaga instrukcji kodu maszynowego: wektor `Vec<u8>` jest teraz uznawany za łańcuch znaków.

Kup książkę Poleć książkę

Funkcje unsafe

Definicja funkcji `unsafe` przypomina definicję standardowej funkcji poprzedzonej słowem kluczowym `unsafe`. Ciało funkcji `unsafe` jest automatycznie traktowane jako blok `unsafe`.

Funkcje `unsafe` mogą być wywoływane jedynie wewnątrz bloków `unsafe`. Oznacza to, że oznakowanie funkcji jako niebezpiecznej ostrzega wywołujący ją kod o konieczności spełnienia warunków określonego kontraktu w celu uniknięcia działania niezdefiniowanego.

Na przykład poniżej widzimy nowy konstruktor dla wcześniej zaprezentowanego typu `Ascii`; tworzy on wartość `Ascii` z wektora bajtowego bez sprawdzania poprawności kodowania ASCII:

```
// Fragment ten należy umieścić w module `my_ascii`
impl Ascii {
    /// Konstruuje wartość `Ascii` z elementu `bytes` bez sprawdzania poprawności
    /// zamieszczonego w nim tekstu ASCII
    ///
    /// Konstruktor ten jest nieomylny i bezpośrednio zwraca wartość `Ascii`,
    /// a nie poprzez `Result<Ascii, NotAsciiError>`, jak to ma miejsce
    /// w przypadku konstruktora `from_bytes`
    ///
    /// # Bezpieczeństwo
    ///
    /// Kod wywołujący musi zagwarantować, że `bytes` będzie zawierać wyłącznie znaki ASCII:
    /// bajty o wartości nie większej niż 0x7f. W przeciwnym wypadku skutki stają się
    /// niezdefiniowane
    pub unsafe fn from_bytes_unchecked(bytes: Vec<u8>) -> Ascii {
        Ascii(bytes)
    }
}
```

Przypuszczalnie kod wywołujący `Ascii::from_bytes_unchecked` wie już w jakiś sposób, że dostępny wektor zawiera wyłącznie znaki ASCII, zatem proces sprawdzania narzucany przez `Ascii::from_bytes` okazałby się stratą czasu, a kod wywołujący musiałby zawierać mechanizm obsługi wyników `Err`, które nigdy by nie występowały. `Ascii::from_bytes_unchecked` pozwala odstąpić od procedury sprawdzania i obsługi błędów.

Jednak wcześniej podkreślaliśmy, że duże znaczenie ma, by konstruktory i funkcje typu Ascii gwarantowały, że wartości tego typu będą poprawie sformatowane. Czy funkcja `from_bytes_unchecked` spełnia ten wymóg?

Niezupełnie: `from_bytes_unchecked` realizuje swoją część zobowiązania, przekazując je kodowi wywołującemu poprzez kontrakt. To właśnie obecność kontraktu sprawia, że oznakowanie funkcji jako niebezpieczna jest prawidłową czynnością: pomimo faktu, że funkcja ta sama w sobie nie realizuje niebezpiecznych operacji, wywołujący ją kod musi przestrzegać reguł, których Rust nie może samodzielnie wymuszać w celu uniknięcia działania niezdefiniowanego.

Czy rzeczywiście można spowodować działanie niezdefiniowane poprzez złamanie warunków kontraktu z `Ascii::from_bytes_unchecked`? Owszem. Możesz stworzyć wartość `String` przechowującą nieprawidłowo sformatowane kodowanie UTF-8:

Kup ksi k Pole ksi k

```
// Przyjmijmy, że poniższy wektor stanowi wynik jakiegoś skomplikowanego procesu,
// który miał wygenerować tekst w formacie ASCII. Coś poszło nie tak!
let bytes = vec![0xf7, 0xbf, 0xbf, 0xbf];

let ascii = unsafe {
    // Ten kontrakt funkcji niebezpiecznej został naruszony z powodu
    // przechowywania bajtów niekodowanych w formacie ASCII przez `bytes`
    Ascii::from_bytes_unchecked(bytes)
};

let bogus: String = ascii.into();

// `bogus` przechowuje teraz nieprawidłowe kodowanie UTF-8. Analiza składniowa
// pierwszego znaku generuje `char` zawierający niewłaściwe kodowanie Unicode
// To przykład działania niezdefiniowanego, więc język nie określa,
// w jaki sposób powinna zadziałać poniższa asercja
assert_eq!(bogus.chars().next().unwrap() as u32, 0x1fffff);
```

W niektórych wersjach Rusta na pewnych platformach systemowych ta asercja generowała błąd z następującym komunikatem:

```
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `2097151`,
 right: `2097151`', src/main.rs:42:5
```

Te dwie liczby wydają się równe, jednak nie jest to spowodowane błędem Rusta, lecz wcześniejszego bloku `unsafe`. To właśnie takie sytuacje mamy na myśli, gdy mówimy, że działanie niezdefiniowane prowadzi do nieprzewidywalnych rezultatów.

Uwidaczniamy tu dwa krytyczne fakty dotyczące błędów i niebezpiecznego kodu:

- *Błędy występujące przed blokiem* `unsafe` *mogą naruszać warunki kontraktu*. To, czy blok `unsafe` będzie powodował działanie niezdefiniowane, zależy nie tylko od kodu zawartego w tym bloku, lecz również od kodu dostarczającego wartości wykorzystywane przez ten blok. Wszystkie elementy, od których zależy działanie kodu, mają związek ze spełnianiem warunków kontraktu i są kluczowe dla bezpieczeństwa. Konwersja z `Ascii` do `String` bazująca na konstruktorze `String::from_utf8_unchecked` jest prawidłowa jedynie wtedy, gdy pozostała część modułu prawidłowo utrzymuje niezmienniki `Ascii`.
- *Skutki naruszenia warunków kontraktu mogą wystąpić po opuszczeniu bloku* `unsafe`. Działanie niezdefiniowane wynikające z nieprzestrzegania wymogów kontraktu właściwości niebezpiecznej często nie objawia się w samym bloku `unsafe`. Jak widać powyżej, skonstruowanie fałszywej wartości `String` (`bogus`) może nie powodować problemów aż do znacznie późniejszego momentu wykonywania programu.

Zasadniczo mechanizmy kontroli typów, zarządzania własnością i inne statyczne mechanizmy kontrolne analizują Twój program i starają się uzyskać dowód, że nie będzie wykazywał działania niezdefiniowanego. Skuteczna kompilacja aplikacji oznacza, że przeszła ona pomyślnie te testy. Blok `unsafe` stanowi dziurę w tym mechanizmie: za jego pomocą informujesz środowisko Rusta, że „ten kod jest prawidłowy, zaufaj mi". Prawdziwość tego stwierdzenia zależy od każdego fragmentu programu wpływającego na operacje wykonywane wewnątrz bloku `unsafe`, natomiast konsekwencje pomyłki mogą objawiać się w dowolnym miejscu objętym działaniem niebezpiecznego bloku.

Kup ksi k Pole ksi k

Wprowadzenie słowa kluczowego `unsafe` jest równoznaczne z przypomnieniem, że nie będziesz mógł korzystać ze wszystkich zalet testów bezpieczeństwa języka.

W obliczu wyboru należy zawsze preferować tworzenie bezpiecznych interfejsów niewykorzystujących kontraktów. O wiele łatwiej z nimi pracować, ponieważ użytkownicy mogą liczyć na mechanizmy kontrolne języka Rust gwarantujące unikanie działania niezdefiniowanego w kodzie. Nawet jeżeli w Twojej implementacji występują właściwości niebezpieczne, najlepiej wykorzystywać system typów, parametrów cyklu życia i modułów tak, aby realizować warunki kontraktów jedynie w zakresie bezpieczeństwa, który sam możesz zagwarantować zamiast zrzucać odpowiedzialność na kod wywołujący.

Niestety dość powszechnie spotykamy się z wykorzystywaniem funkcji niebezpiecznych, których dokumentacja nie zawiera opisu kontraktów. Ich twórcy oczekują, że samodzielnie odkryjesz obowiązujące je reguły na podstawie własnego doświadczenia i wiedzy na temat działania kodu. Jeżeli kiedykolwiek miałeś wątpliwości związane z poprawnością stosowania API języka C lub C++, to tutaj też to poczujesz.

Kod niebezpieczny czy funkcja niebezpieczna?

Być może zastanawiasz się, czy lepiej korzystać z bloku `unsafe`, czy po prostu oznaczyć całą funkcję jako niebezpieczną. Zalecamy podjąć najpierw decyzję na temat funkcji:

- Jeżeli istnieje możliwość nieprawidłowego używania funkcji tak, że zostaje ona właściwie skompilowana, ale mimo to powoduje działanie niezdefiniowane, to musisz oznaczyć ją jako niebezpieczną. Kontrakty są regułami prawidłowego korzystania z danej funkcji; obecność kontraktu sprawia, że funkcja staje się niebezpieczna.
- W przeciwnym wypadku funkcja jest bezpieczna: jej prawidłowe wywołanie nie ma prawa powodować działania niezdefiniowanego. Nie należy poprzedzać jej słowem kluczowym `unsafe`.

To, czy w ciele funkcji wykorzystywane są właściwości niebezpieczne, jest nieistotne — znaczenie ma obecność kontraktu. Pokazaliśmy wcześniej funkcję niebezpieczną niewykorzystującą właściwości niebezpiecznych, a także funkcję bezpieczną używającą właściwości niebezpiecznych.

Nie oznaczaj funkcji bezpiecznej słowem kluczowym `unsafe` wyłącznie dlatego, że wykorzystujesz właściwości niebezpieczne w jej ciele. Utrudniasz sobie w ten sposób jej stosowanie i wprowadzasz zamęt wśród użytkowników, którzy będą oczekiwać (prawidłowo) opisu kontraktu w dokumentacji. Zamiast tego wprowadź blok `unsafe`, nawet jeżeli okazuje się nim ciało funkcji.

Działanie niezdefiniowane

Na początku rozdziału stwierdziliśmy, że *działanie niezdefiniowane* to takie „działanie, na które (zgodnie z założeniami języka Rust) kod nie powinien być nigdy wystawiony”. Jest to dziwne sformułowanie, zwłaszcza gdy wiemy na podstawie własnych doświadczeń z innymi językami, że takie zachowania *występują* przypadkowo z różną częstością. Dlaczego pojęcie to okazuje się przydatne podczas wyznaczania zobowiązań kodu niebezpiecznego?

Kup ksi k Pole ksi k

Kompilator tłumaczy jeden język programowania na inny. Kompilator Rusta przekłada program napisany w języku Rust na równoważny program napisany w kodzie maszynowym. Co jednak oznacza stwierdzenie, że dwa programy napisane w zupełnie odmiennych językach są sobie równoważne?

Na szczęście pytanie to jest łatwiejsze dla programistów niż dla językoznawców. Mówimy, że dwa programy są równoważne, jeżeli po uruchomieniu będą zawsze cechować się takim samym widocznym zachowaniem: będą realizować takie same wywołania systemowe, oddziaływać w analogiczny sposób z obcymi bibliotekami itd. Przypomina to trochę test Turinga dla programów: jeżeli nie potrafisz odróżnić zachowania oryginalnego programu od działania jego przekładu na kod maszynowy, to są one sobie równoważne.

Spójrz na ten oto kod:

```
let i = 10;
very_trustworthy(&i);
println!("{}", i * 100);
```

Nawet nie znając definicji `very_trustworthy`, widzimy, że przekazywana jest wyłącznie referencja współdzielona do `i`, co oznacza, że wywołanie funkcji nie zmieni wartości `i`. Skoro wartość przekazywana do `printnln!` będzie zawsze wynosiła `1000`, powyższy kod byłby tłumaczony na język maszynowy tak samo jak w przypadku, gdybyśmy użyli następującego zapisu:

```
very_trustworthy(&10);
println!("{}", 1000);
```

Taka przekształcona wersja działa tak samo jak pierwszy kod, a prawdopodobnie jest nawet odrobinę szybsza. Należy jednak rozpatrywać wydajność tej wersji wyłącznie wtedy, gdy zgodzimy się, że obydwie wersje mają to samo znaczenie. A gdybyśmy zdefiniowali `very_trustworthy` następująco:

```
fn very_trustworthy(shared: &i32) {
    unsafe {
        // Przekształca referencję współdzieloną we wskaźnik mutowalny
        // Jest to działanie niezdefiniowane
        let mutable = shared as *const i32 as *mut i32;
        *mutable = 20;
    }
}
```

Kod ten narusza reguły referencji współdzielonych: zmienia wartość `i` na `20`, pomimo tego, że powinna być zamrożona, ponieważ została użyczona do współdzielenia. W konsekwencji przekształcenie wprowadzone do kodu wywołującego jest teraz wyraźnie widoczne: jeżeli kod zostanie przełożony na język maszynowy, będziemy otrzymywać w wyniku wartość `1000`, a jeżeli kod zostanie pozostawiony w dotychczasowym stanie i wykorzysta nową wartość `i`, otrzymamy wynik `2000`. Nieprzestrzeganie reguł referencji współdzielonych w `very_trustworthy` oznacza, że referencje współdzielone podczas wywoływania będą działały w nieprzewidywalny sposób.

Tego typu problem pojawia się niemal w każdym rodzaju przekształcenia realizowanym w języku Rust. Nawet rozwijanie funkcji w kodzie wywołującym przyjmuje między innymi, że gdy funkcja ta zakończy działanie, to przepływ sterowania powróci do kodu wywołującego. Rozpoczęliśmy jednak ten rozdział od ukazania przykładowego, nieprawidłowo napisanego kodu, który narusza nawet to założenie.

Kup ksi k Pole ksi k

Zasadniczo jest niemożliwe, aby Rust (albo jakikolwiek inny język) był w stanie ocenić, czy przekształcenie programu zachowa jego pierwotne znaczenie, chyba że zaufa fundamentalnym właściwościom języka określającym jego prawidłowe działanie. A to, czy on działa prawidłowo, czy nie, zależy nie tylko od używanego kodu, ale także od innych, może nawet odległych elementów programu. Aby Rust mógł cokolwiek zrobić z Twoim kodem, musi założyć, że pozostała część programu jest prawidłowa.

Oto reguły prawidłowych programów w języku Rust:

- Program nie może czytać niezainicjalizowanej pamięci.
- Program nie może tworzyć nieprawidłowych wartości prostych:
  - referencje, wartości typu `box` oraz wskaźniki do funkcji o wartości `null`,
  - wartości `bool` inne od `0` i `1`,
  - wartości `enum` o nieprawidłowych wartościach wyróżnika,
  - wartości `char` w nieprawidłowym, niezastępczym formacie Unicode,
  - wartości `str` o nieprawidłowym formatowaniu UTF-8,
  - grube wskaźniki o z nieprawidłową długością tablicy (`vtable`) lub podzbioru,
  - dowolne wartości typu `!`.
- Muszą być przestrzegane opisane w rozdziale 5. reguły referencji. Żadna referencja nie może istnieć dłużej od referencji pierwotnej; dostęp współdzielony służy wyłącznie do odczytu, natomiast dostęp mutowalny jest dostępem wyłącznym.
- Program nie może tworzyć dereferencji do pustych, niedopasowanych wskaźników lub korzystać z nieaktualnych wskaźników.
- Program nie może wykorzystywać wskaźnika do uzyskiwania dostępu do pamięci znajdującej się poza alokacją, z którą ten wskaźnik jest powiązany. Wyjaśnimy dokładnie tę regułę w punkcie „Bezpieczne tworzenie dereferencji wskaźników niechronionych”.
- Program musi być pozbawiony wyścigów do danych. Wyścig do danych ma miejsce wtedy, gdy dwa niezsynchronizowane wątki uzyskują dostęp do tego samego obszaru pamięci, a przynajmniej jeden z nich realizuje operację zapisu.
- Program nie może odwijać stosu zgodnie z wywołaniem wygenerowanym przez inny język poprzez interfejs języka obcego, co zostało wyjaśnione w punkcie „Odwinięcie stosu” rozdziału 7.
- Program musi być zgodny z kontraktami funkcji biblioteki standardowej.

Ponieważ nie dysponujemy jeszcze szczegółowym modelem semantyki bloków `unsafe` w Ruście, można przypuszczać, że z upływem czasu powyższa lista będzie ulegać zmianom, aczkolwiek powyższe reguły pozostaną w mocy.

Każde naruszenie powyższych reguł jest działaniem niezdefiniowanym i sprawia, że nie będzie można w pełni zaufać w poprawność podejmowanych przez Rust prób optymalizacji kodu i przekształcenia go do języka maszynowego. Jeśli złamiemy ostatnią z powyższych reguł i przekażemy

Kup ksi k Pole ksi k

do funkcji `String::from_utf8_unchecked` nieprawidłowo sformatowany tekst UTF-8, może się okazać, że faktycznie 2097151 nie będzie równe 2097151.

Kod w Ruście niewykorzystujący właściwości niebezpiecznych zapewnia przestrzeganie powyższych reguł po skompilowaniu programu (zakładając, że sam kompilator będzie działał bezbłędnie: dążymy do tego, jednak krzywa nigdy nie dociera do asymptoty). Zostajesz obarczony odpowiedzialnością za te reguły jedynie w przypadku stosowania właściwości niebezpiecznych.

W językach C i C++ fakt, że program jest kompilowany bez błędów czy ostrzeżeń, ma o wiele mniejsze znaczenie. Jak wiesz z wprowadzenia do tej książki, nawet najlepsze programy napisane w C czy C++ tworzone przez doświadczonych programistów troszczących się o wysoki standard swojego kodu w praktyce wykazują działanie niezdefiniowane.

Zestawy metod unsafe

Niebezpieczny zestaw metod (ang. *unsafe trait*) to zestaw metod zawierający kontrakt, którego Rust nie może sprawdzić lub którego przestrzegania nie może wymusić, a którego warunki programista musi spełnić w celu uniknięcia działania niezdefiniowanego. To Ty musisz zrozumieć kontrakt danego zestawu metod i zagwarantować jego przestrzeganie przez Twój typ.

Funkcja wiążąca zmienne typu z niebezpiecznym zestawem metod zazwyczaj sama w sobie wykorzystuje właściwości niebezpieczne i spełnia warunki ich kontraktu wyłącznie w zależności od kontraktu interfejsu niebezpiecznego. Nieprawidłowa implementacja zestawu metod może wywoływać działanie niezdefiniowane tej funkcji.

Klasycznymi przykładami niebezpiecznych zestawów metod są `std::marker::Send` i `std::mar↪ker::Sync`. Nie definiują one żadnych metod, zatem z łatwością można je implementować dla dowolnego typu. Zawierają one jednak kontrakty: interfejs `Send` wymaga od programisty, aby umożliwił bezpieczne przechodzenie do innego wątku, natomiast `Sync` wymaga bezpiecznego współdzielenia pomiędzy wątkami za pomocą referencji współdzielonych. Na przykład implementacja zestawu metod `Send` dla niewłaściwego typu sprawiłaby, że `std::sync::Mutex` stałby się podatny na wyścigi do danych.

Biblioteka języka Rust zawiera na przykład interfejs niebezpieczny `core::nonzero::Zeroable` dla typów, które można bezpiecznie inicjalizować poprzez wyznaczanie ich wszystkim bajtom wartości `0`. Oczywiście zerowanie typu `usize` jest całkowicie bezpieczne, ale wyzerowanie `&T` daje pustą referencję, która w przypadku braku dereferencji spowoduje zawieszenie aplikacji. Możliwe jest przeprowadzenie pewnych metod optymalizacyjnych dla typów zerowalnych: możesz szybko inicjalizować ich tablicę za pomocą `std::mem::write_bytes` (która w języku Rust jest odpowiednikiem funkcji `memset`) lub wykorzystywać wywołania systemu operacyjnego przydzielające wyzerowane strony. (Interfejs `Zeroable` był niestabilny i w wersji 1.26 języka Rust został przeznaczony wyłącznie do zastosowań wewnętrznych i przeniesiony do paczki `num`, niemniej jest doskonałym, prostym przykładem, zaczerpniętym z rzeczywistości).

Kup ksi k

Pole ksi k

`Zeroable` jest typowym interfejsem znacznikowym pozbawionym metod lub powiązanych typów:

```
pub unsafe trait Zeroable {}
```

Implementacje dla właściwych typów nie są skomplikowane:

```
unsafe impl Zeroable for u8 {}
unsafe impl Zeroable for i32 {}
unsafe impl Zeroable for usize {}
// I tak dalej dla wszystkich typów int
```

Korzystając z tych definicji, możemy napisać funkcję szybko wyznaczającą wektor danej długości zawierający typ `Zeroable`:

```
use core::nonzero::Zeroable;

extern crate core;
use core::nonzero::Zeroable;

fn zeroed_vector<T>(len: usize) -> Vec<T>
    where T: Zeroable
{

    let mut vec = Vec::with_capacity(len);
    unsafe {
        std::ptr::write_bytes(vec.as_mut_ptr(), 0, len);
        vec.set_len(len);
    }
    vec
}
```

Funkcja ta zaczyna się od stworzenia pustego wektora `Vec` o wymaganej pojemności, po czym następuje wywołanie `write_bytes` służące do wypełnienia wolnego bufora zerami. (Funkcja `write_bytes` traktuje `len` jako liczbę elementów `T`, a nie jako liczbę bajtów, zatem jej wywołanie zapełnia cały bufor). Metoda `set_len` wektora zmienia jego długość, nie wpływając na sam bufor — nie jest to bezpieczne, ponieważ musisz zagwarantować, że nowa przestrzeń bufora będzie rzeczywiście zawierała prawidłowo zainicjalizowane wartości typu `T`. Takie właśnie jednak jest zadanie `T: Zeroable`: blok zawierający bajty o wartości 0 reprezentuje prawidłową wartość `T`. Korzystamy z funkcji `set_len` w bezpieczny sposób.

Zróbmy z tego użytek:

```
let v: Vec<usize> = zeroed_vector(100_000);
assert!(v.iter().all(|&u| u == 0));
```

Najwyraźniej `Zeroable` musi być interfejsem niebezpiecznym, ponieważ implementacja naruszająca jego kontrakt może prowadzić do działania niezdefiniowanego:

```
struct HoldsRef<'a>(&'a mut i32);

unsafe impl<'a> Zeroable for HoldsRef<'a> { }

let mut v: Vec<HoldsRef> = zeroed_vector(1);
*v[0].0 = 1; // Aplikacja przestaje działać: dereferencja wskaźnika pustego
```


Kup ksi k Pole ksi k

Rust nie wie, co oznacza interfejs `Zeroable`, zatem nie może wiedzieć, że ten zestaw metod zostaje zaimplementowany dla niewłaściwego typu. Podobnie jak w przypadku wszystkich pozostałych właściwości niebezpiecznych, to Ty musisz pojmować i realizować warunki kontraktu interfejsu niebezpiecznego.

Zwróć uwagę, że kod niebezpieczny nie może zależeć od zwykłych, bezpiecznych i prawidłowo zaimplementowanych interfejsów. Załóżmy, na przykład, że mamy do czynienia z implementacją interfejsu `std::hash::Hasher` zwracającą losową wartość hasza niezwiązaną z wartościami haszowanymi. Zestaw metod wymaga, aby dwukrotne haszowanie tych samych bitów dawało tę samą wartość hasza, implementacja ta nie spełnia jednak owego wymogu — jest po prostu nieprawidłowa. Skoro jednak `Hasher` nie jest interfejsem niebezpiecznym, kod niebezpieczny nie może wykazywać działania niezdefiniowanego podczas korzystania z tego interfejsu. Typ `std::collections::HashMap` został ostrożnie napisany w odniesieniu do kontraktów stanowiących część wykorzystywanych przez niego właściwości niebezpiecznych bez względu na zachowanie interfejsu `Hasher`. Z pewnością tablica nie będzie prawidłowo działać: operacje przeszukiwania będą kończyć się niepowodzeniem, a wpisy będą losowo pojawiać się i znikać. Sama tabela jednak nie będzie wykazywać działania niezdefiniowanego.

Wskaźniki niechronione

Wskaźnik niechroniony w języku Rust nie jest w żaden sposób ograniczany. Za jego pomocą możesz tworzyć wszelkie struktury, których nie da się budować przy użyciu sprawdzanych typów wskaźników, na przykład podwójnie łączone listy czy własne grafy obiektów. Jednak właśnie z powodu jego wielozadaniowości Rust nie wie, czy jest wykorzystywany w bezpieczny sposób, zatem można tworzyć jego dereferencje jedynie w bloku `unsafe`.

Wskaźniki niechronione zasadniczo są równoważne wskaźnikom języków C i C++, są więc pomocne w oddziaływaniach z kodem napisanym w tych językach.

Istnieją dwa rodzaje wskaźników niechronionych:

- `*mut T` jest wskaźnikiem niechronionym dla `T` umożliwiającym modyfikowanie referencji pierwotnej.
- `*const T` jest wskaźnikiem niechronionym dla `T` umożliwiającym jedynie odczytywanie referencji pierwotnej.

(Nie istnieje samodzielny typ `*T`; musisz zawsze wyznaczać `mut` lub `const`.)

Możesz stworzyć wskaźnik niechroniony, dokonując konwersji z referencji, a następnie tworząc jej dereferencję za pomocą operatora *:

```
let mut x = 10;
let ptr_x = &mut x as *mut i32;

let y = Box::new(20);
let ptr_y = &*y as *const i32;

unsafe {
    *ptr_x += *ptr_y;
}
assert_eq!(x, 30);
```


Kup ksi k Pole ksi k

W przeciwieństwie do typu box i referencji wskaźniki niechronione mogą być puste, podobnie jak NULL w C czy nullptr w C++:

```
fn option_to_raw<T>(opt: Option<&T>) -> *const T {
    match opt {
        None => std::ptr::null(),
        Some(r) => r as *const T
    }
}

assert!(!option_to_raw(Some(&("pea", "pod"))).is_null());
assert_eq!(option_to_raw::<i32>(None), std::ptr::null());
```

Przykład ten nie zawiera bloków unsafe: operacje tworzenia, przekazywania i porównywania wskaźników niechronionych są całkowicie niegroźne. Jedynie tworzenie jego dereferencji jest niebezpieczne.

Wskaźnik niechroniony dla typu o zmiennym rozmiarze jest, podobnie jak byłoby w przypadku odpowiadających mu referencji lub wartości typu Box, wskaźnikiem grubym. Wskaźnik *const [u8] zawiera oprócz adresu również długość, natomiast obiekt zestawu metod, taki jak wskaźnik *mut dyn std::io::Write, przechowuje tablicę (vtable).

Rust niejawnie tworzy w różnych sytuacjach dereferencje bezpiecznych typów wskaźników, jednak referencja wsteczna wskaźnika niechronionego musi być wykonana jawnie:

- Operator . nie będzie niejawnie tworzył dereferencji dla wskaźnika niechronionego; musisz napisać (*raw).field lub (*raw).method(...).
- Wskaźniki niechronione nie implementują interfejsu Deref, zatem nie można stosować tu wymuszenia konwersji typu.
- Operatory takie jak == czy < porównują wskaźniki niechronione jako adresy: dwa wskaźniki niechronione są sobie równe, jeżeli wskazują tę samą pozycję w pamięci. Na drodze analogii haszowanie wskaźnika niechronionego dotyczy wskazywanego przezeń adresu, a nie wartości jego referencji pierwotnej.
- Interfejsy formatujące takie jak std::fmt::Display automatycznie podążają za referencjami, ale zupełnie ignorują wskaźniki niechronione. Wyjątek stanowią std::fmt::Debug i std::fmt::Pointer, ukazujące wskaźniki niechronione w postaci adresów heksadecymalnych bez tworzenia dereferencji.

W przeciwieństwie do języków C i C++ operator + w języku Rust nie obsługuje wskaźników niechronionych, ale możesz przeprowadzać operacje na nich za pomocą metod offset i wrapping_offset bądź przy użyciu wygodniejszych metod add, sub, wrapping_add oraz wrapping_sub. Z kolei metoda offset_from zwraca przesunięcie pomiędzy dwoma wskaźnikami wyrażone w bajtach, choć sami musimy się upewnić, czy oba wskaźniki należą do tego samego regionu pamięci (na przykład do tego samego wektora Vec):

```
let trucks = vec!["garbage truck", "dump truck", "moonstruck"];
let first: *const &str = &trucks[0];
let last: *const &str = &trucks[2];
assert_eq!(unsafe { last.offset_from(first) }, 2);
assert_eq!(unsafe { first.offset_from(last) }, -2);
```


Kup książkę Poleć książkę

Wskaźników `first` i `last` nie trzeba poddawać żadnej jawnej konwersji — w zupełności wystarczy określenie ich typów. Rust niejawnie wymusza konwersję referencji na wskaźnikach niechronionych (oczywiście nie działa to w drugą stronę).

Operator `as` umożliwia niemal każdą sensowną konwersję od referencji do wskaźników niechronionych lub pomiędzy dwoma typami wskaźników niechronionych. Być może jednak konieczne będzie rozbicie złożonej konwersji na kilka prostszych etapów:

```
&vec![42_u8] as *const String // Błąd: niewłaściwa konwersja
&vec![42_u8] as *const Vec<u8> as *const String; // Dozwolone
```

Należy zauważyć, że `as` nie przekształca wskaźników niechronionych w referencje. Taka forma konwersji nie byłaby bezpieczna, natomiast `as` powinna pozostawać bezpieczną operacją. Zamiast tego należy stworzyć dereferencję wskaźnika niechronionego (w bloku `unsafe`), a następnie użyczyć otrzymaną wartość.

Zachowaj maksymalną ostrożność podczas wykonywania tej operacji, ponieważ tak stworzona referencja ma nieograniczony czas życia. Może ona istnieć dowolnie długo, ponieważ wskaźnik niechroniony nie wyznacza żadnych podstaw dotyczących czasu życia. W punkcie „Bezpieczny interfejs dla implementacji libgit2” zaprezentujemy kilka przykładów właściwego ograniczania parametru czasu życia.

Wiele typów zawiera metody `as_ptr` i `as_mut_ptr` zwracające wskaźnik niechroniony. Na przykład podzbiory tablicowe i łańcuchy znaków zwracają wskaźniki do pierwszych ich elementów, a niektóre iteratory przygotowują wskaźnik do następnego wygenerowanego przez nie elementu. Typy wskaźników z prawami własności do wskazywanej wartości, np. `Box`, `Rc` czy `Arc`, zawierają funkcje `into_raw` i `from_raw`, przeznaczone, odpowiednio, do konwersji do wskaźnika niechronionego i ze wskaźnika niechronionego. Kontrakty niektórych z tych metod narzucają zaskakujące wymagania, dlatego przed użyciem należy przejrzeć ich dokumentację.

Możesz także konstruować wskaźniki niechronione poprzez konwersję liczb całkowitych, chociaż jedyne wartości tego typu, którym możesz zaufać, najczęściej otrzymywane są właśnie ze wskaźnika. Sposób wykorzystywania wskaźników w ten sposób został zaprezentowany w punkcie „Przykład: RefWithFlag”.

W przeciwieństwie do referencji, wskaźniki niechronione nie są interfejsami `Send` ani `Sync`. Wskutek tego każdy typ zawierający wskaźniki niechronione nie implementuje domyślnie tych zestawów metod. Zasadniczo nie ma nic groźnego w przesyłaniu lub współdzieleniu wskaźników niechronionych pomiędzy wątkami — gdziekolwiek trafiają, do stworzenia ich dereferencji wymagany jest blok `unsafe`. Zważywszy jednak na typowe zadania pełnione przez te wskaźniki, twórcy języka uznali, że domyślne zachowanie będzie bardziej przydatne. W podrozdziale „Zestawy metod unsafe” wyjaśniliśmy mechanizm samodzielnej implementacji interfejsów `Send` i `Sync`.

Kup ksi k Pole ksi k

Bezpieczne tworzenie dereferencji wskaźników niechronionych

Oto rozsądne wskazówki dotyczące bezpiecznego wykorzystywania wskaźników niechronionych:

- Dereferencja wskaźników pustych lub nieaktualnych stanowi działanie niezdefiniowane, ponieważ odnosi się do niezainicjalizowanej pamięci lub wartości wykraczających poza dopuszczalny zakres.
- Dereferencja niewłaściwie rozmieszczonych wskaźników wobec ich referencji pierwotnej stanowi działanie niezdefiniowane.
- Wartości z dereferencji wskaźnika niechronionego mogą być użyczane jedynie wtedy, gdy operacja ta jest zgodna z regułami bezpieczeństwa referencji opisanymi w rozdziale 5.: żadna referencja nie może istnieć dłużej od referencji pierwotnej, dostęp współdzielony pozwala wyłącznie na odczyt, natomiast dostęp mutowalny jest dostępem wyłącznym (łatwo przypadkowo naruszyć tę regułę, ponieważ wskaźniki niechronione często wykorzystywane są do tworzenia struktur danych cechujących się niestandardowymi własnościami współdzielenia lub praw własności).
- Możesz korzystać z referencji pierwotnej wskaźnika niechronionego wyłącznie wtedy, gdy ma prawidłową wartość danego typu. Musisz na przykład zagwarantować, że dereferencja `*const char` zawiera prawidłowy, niezastępczy format Unicode.
- Możesz stosować metody `offset` i `wrapping_offset` wobec wskaźników niechronionych jedynie w celu wskazywania bajtów mieszczących się w zmiennej lub w alokowanym bloku pamięci, do którego odnosi się pierwotny wskaźnik, ewentualnie do pierwszego bajta poza takim rejonem.

  Jeżeli przeprowadzasz operacje arytmetyczne na wskaźniku poprzez jego konwersję do liczby całkowitej, wykonanie na niej potrzebnych obliczeń, a następnie przywrócenie do postaci wskaźnika, w rezultacie możesz otrzymać wskaźnik spełniający reguły metody `offset`.
- Jeżeli wykonujesz operację przyporządkowania do referencji pierwotnej wskaźnika niechronionego, nie możesz naruszać niezmienników żadnego typu stanowiącego część tej referencji. Na przykład jeśli `*mut u8` wskazuje bajt typu `String`, możesz przechowywać wyłącznie wartości w `u8` w postaci łańcucha znaków prawidłowo sformatowanego w kodowaniu UTF-8.

Pomijając regułę użyczania, obowiązują tu takie same zasady korzystania ze wskaźników jak w językach C i C++.

Powód nienaruszalności niezmienników typów powinien być oczywisty. Implementacje wielu standardowych typów w języku Rust zawierają kod niebezpieczny, ale zapewniają jednocześnie bezpieczne interfejsy przy założeniu przestrzegania testów bezpieczeństwa, systemu modułów i reguł widoczności. Wykorzystywanie wskaźników niechronionych do pomijania tych zabezpieczeń może prowadzić do działania niezdefiniowanego.

Nie jest łatwo ustalić pełny i dokładny kontrakt wskaźników niechronionych, a ponadto może on ulegać zmianom wraz z rozwojem języka. Opisane tu zasady powinny jednak pomóc Ci pozostać w bezpiecznej strefie.

Kup ksi k Pole ksi k

Przykład: RefWithFlag

Zaprezentujemy teraz klasyczny przykład sztuczki na poziomie bitowym możliwej do realizacji dzięki wskaźnikom niechronionym, opakowanym w postaci przypominającej całkowicie bezpieczny typ Rusta. Moduł ten definiuje typ `RefWithFlag<'a, T>`, przechowujący zarówno wartości `&'a T`, jak i `bool` w postaci krotki `(&'a T, bool)`, a mimo to zajmuje jedno słowo maszynowe zamiast dwóch. Tego typu technika stosowana jest powszechnie w programach czyszczących pamięć i w maszynach wirtualnych, gdzie pewne typy (załóżmy, na przykład, że typ reprezentujący obiekt) są tak liczne, że dodanie nawet jednego słowa do każdej wartości drastycznie zwiększyłoby wykorzystanie pamięci:

```
mod ref_with_flag {
    use std::marker::PhantomData;
    use std::mem::align_of;

    /// Typy `&T` i `bool` umieszczone w jednym słowie
    /// Typ `T` musi wymagać przynajmniej przydzielenia 2-bajtowego
    ///
    /// Jeżeli jesteś programistą, który nigdy dotąd nie natrafił na wskaźnik
    /// 2⁰-bitowy, którego nie chciałeś „ukraść”, to teraz możesz bezpiecznie tego
    /// dokonać! (Ale w ten sposób to wcale nie jest ekscytujące...)
    pub struct RefWithFlag<'a, T> {
        ptr_and_bit: usize,
        behaves_like: PhantomData<&'a T> // Nie zajmuje miejsca
    }

    impl<'a, T: 'a> RefWithFlag<'a, T> {
        pub fn new(ptr: &'a T, flag: bool) -> RefWithFlag<T> {
            assert!(align_of::<T>() % 2 == 0);
            RefWithFlag {
                ptr_and_bit: ptr as *const T as usize | flag as usize,
                behaves_like: PhantomData
            }
        }

        pub fn get_ref(&self) -> &'a T {
            unsafe {
                let ptr = (self.ptr_and_bit & !1) as *const T;
                &*ptr
            }
        }

        pub fn get_flag(&self) -> bool {
            self.ptr_and_bit & 1 != 0
        }
    }
}
```

W kodzie tym wykorzystujemy fakt, że wiele typów musi być umieszczanych w parzystych adresach pamięci: najmniej istotnym bitem parzystego adresu jest zawsze 0, dlatego możemy przechowywać tam coś innego, a następnie rzetelnie zrekonstruować pierwotny adres poprzez maskowanie dolnego bitu. Nie wszystkie bity nadają się do tego rozwiązania — na przykład typy `u8` i `(bool, [i8; 2])` można umieszczać pod dowolnym adresem. Możemy jednak sprawdzić przydzielenie danego typu do konstrukcji i odrzucać typy, które nie będą działać.

Kup ksi k Pole ksi k

Moduł RefWithFlag możemy wykorzystać w następujący sposób:

```
use ref_with_flag::RefWithFlag;

let vec = vec![10, 20, 30];
let flagged = RefWithFlag::new(&vec, true);
assert_eq!(flagged.get_ref()[1], 20);
assert_eq!(flagged.get_flag(), true);
```

Konstruktor RefWithFlag::new przyjmuje referencję i wartość logiczną, stwierdza, że typ referencji jest odpowiedni, po czym przekształca ją w znacznik niechroniony, a następnie w typ usize. Zostaje zdefiniowany typ usize o rozmiarze wystarczającym do przechowywania wskaźnika w dowolnym procesorze, dla którego tworzony jest kod wynikowy, zatem przekształcenie wskaźnika niechronionego do typu usize i z powrotem jest prawidłową operacją. Wiemy, że otrzymany typ usize musi być parzysty, możemy więc użyć bitowego operatora *OR* | do połączenia go z wartością bool, przekształconą do postaci liczby całkowitej 0 lub 1.

Metoda get_flag wydobywa składową logiczną modułu RefWithFlag. To proste: wystarczy maskować dolny bit i sprawdzić, czy jest niezerowy.

Metoda get_ref wydobywa referencję z modułu RefWithFlag. Maskuje dolny bit typu usize i przekształca go do postaci wskaźnika niechronionego. Operator as nie będzie konwertował wskaźników niechronionych do postaci referencji, możemy jednak stworzyć dereferencję tego wskaźnika niechronionego (oczywiście w bloku unsafe) i ją pożyczyć. Pożyczenie referencji pierwotnej wskaźnika niechronionego sprawia, że otrzymujemy referencję o nieograniczonym czasie życia: Rust pozwoli na użycie referencji niezależnie od cyklu życia zdefiniowanego wokół niej, jeśli taki został określony. Zazwyczaj jednak występuje jakiś określony, dokładniejszy parametr czasu życia, co sprawia, że jest wychwytywanych więcej błędów. W tym przypadku zwracanym typem metody get_ref jest &'a T, zatem Rust widzi, że oczekiwany czas życia referencji musi odpowiadać parametrowi czasu życia 'a typu RefWithFlag, czyli dokładnie tak, jak tego chcemy. Jest to czas życia naszej początkowej referencji.

W pamięci RefWithFlag wygląda tak samo jak typ usize: PhantomData jest typem o rozmiarze zerowym, dlatego pole behaves_like nie zabiera miejsca w strukturze. Jednak typ PhantomData jest niezbędny do określenia sposobu traktowania parametrów czasu życia w kodzie wykorzystującym RefWithFlag. Wyobraź sobie, jak typ ten wyglądałby bez pola behaves_like:

```
// Poniższy kod nie zostanie skompilowany
pub struct RefWithFlag<'a, T: 'a> {
    ptr_and_bit: usize
}
```

W rozdziale 5. stwierdziliśmy, że żadna struktura zawierająca referencje nie może istnieć dłużej od pożyczanych przez nie wartości, gdyż w przeciwnym razie referencje stawałyby się nieaktualnymi wskaźnikami. Struktura musi przestrzegać ograniczeń nakładanych na jej pola. Ma to miejsce również w RefWithFlag: w omawianym kodzie flagged nie może istnieć dłużej od vec, ponieważ flagged.get_ref() zwraca referencję do wektora. Jednak nasz zredukowany typ RefWithFlag nie zawiera żadnych referencji i nigdy nie wykorzystuje parametru cyklu życia 'a. Ma wyłącznie postać typu usize. Skąd Rust ma wiedzieć, czy i jakie ograniczenia dotyczą czasu życia pab? Dołączenie pola PhantomData<&'a T> mówi Rustowi, że RefWithFlag<'a, T> ma być traktowany *tak, jakby* zawierał &'a T bez rzeczywistego wpływania na reprezentację struktury.

Kup ksi k Pole ksi k

Mimo że Rust nie wie, co się tu dzieje (z tego powodu moduł `RefWithFlag` jest uznawany za niebezpieczny), będzie starał się pomagać nam w jak największym stopniu. Jeżeli pominiesz pole `behaves_like`, będzie wyświetlane ostrzeżenie o nieużywaniu parametrów `'a` i `T`, a także sugestia wykorzystania typu `PhantomData`.

`RefWithFlag` wykorzystuje takie same taktyki do unikania działania niezdefiniowanego w bloku `unsafe` jak omówiony wcześniej typ `Ascii`. Typ sam w sobie jest `pub`, ale nie jego pola, co oznacza, że jedynie kod w module `ref_with_flag` może tworzyć wartość `RefWithFlag` lub zaglądać do niej. Nie trzeba dokładnie analizować kodu, aby mieć pewność, że pole `ptr_and_bit` jest prawidłowo skonstruowane.

Wskaźniki dopuszczające wartość pustą

W języku Rust wskaźnik niechroniony o wartości pustej jest bezadresowy, podobnie jak w językach C i C++. Dla dowolnego typu `T` funkcja `std::ptr::null<T>` zwraca wskaźnik pusty `*const T`, natomiast funkcja `std::ptr::null_mut<T>` zwraca wskaźnik pusty `*mut T`.

Istnieje kilka sposobów sprawdzenia, czy dany wskaźnik niechroniony jest pusty. Najprościej jest wykorzystać metodę `is_null`, ale metoda `as_ref` może być wygodniejsza: przyjmuje wskaźnik `*const T` i zwraca `Option<&'a T>`, przekształcając wskaźnik pusty do typu `None`. Na drodze analogii metoda `as_mut` konwertuje wskaźniki `*mut T` do wartości `Option<&'a mut T>`.

Rozmiary i rozmieszczanie typów

Wartość dowolnego typu `Sized` zajmuje w pamięci stałą liczbę bajtów i musi zostać umieszczona w adresie stanowiącym wielokrotność jakiejś wartości *rozmieszczenia* (ang. *alignment*), wyznaczanej przez architekturę urządzenia. Na przykład krotka `(i32, i32)` zajmuje 8 bajtów, a większość procesorów preferuje jej umieszczenie w adresie stanowiącym wielokrotność liczby 4.

Wywołanie funkcji `std::mem::size_of::<T>()` zwraca rozmiar wartości dla typu `T` (w bajtach), natomiast `std::mem::align_of::<T>()` zwraca wymagane rozmieszczenie. Na przykład:

```
assert_eq!(std::mem::size_of::<i64>(), 8);
assert_eq!(std::mem::align_of::<(i32, i32)>(), 4);
```

Rozmieszczenie każdego typu zawsze stanowi potęgę liczby 2.

Rozmiar typu zawsze jest zaokrąglany w górę do wielokrotności jego rozmieszczenia, nawet jeżeli z perspektywy technicznej mógłby zajmować mniej miejsca. Na przykład, mimo że krotka `(f32, u8)` wymaga zaledwie 5 bajtów, `size_of::<(f32, u8)>()` wynosi 8, ponieważ `align_of::<(f32, u8)>()` daje wartość `4`. W ten sposób w przypadku tablicy rozmiar typu elementu zawsze odzwierciedla odstępy pomiędzy dwoma elementami.

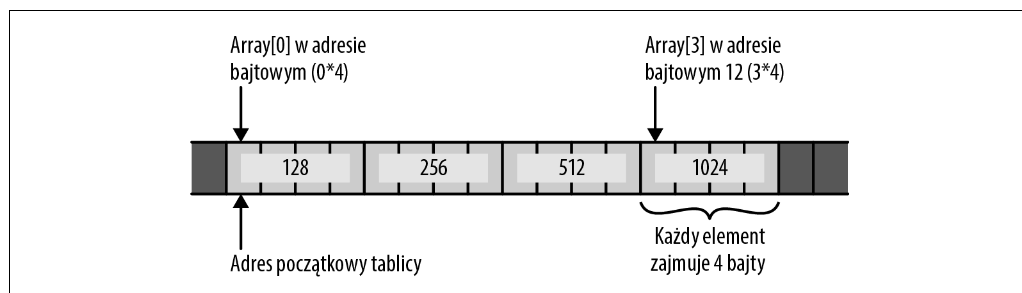
W przypadku typów o nieznanym rozmiarze rozmiar i rozmieszczenie zależą od dostępnej wartości. Mając daną referencję do wartości o nieznanym rozmiarze, funkcje `std::mem::size_of_val` i `std::mem::align_of_val` zwracają, odpowiednio, jej rozmiar i rozmieszczenie. Funkcje te działają wobec referencji zarówno dla typów `Sized`, jak i o nieznanym rozmiarze.

Kup ksi k Pole ksi k

```
// Wskaźniki grube do podzbiorów zawierają długość ich referencji pierwotnej
let slice: &[i32] = &[1, 3, 9, 27, 81];
assert_eq!(std::mem::size_of_val(slice), 20);

let text: &str = "aligator";
assert_eq!(std::mem::size_of_val(text), 9);

use std::fmt::Display;
let unremarkable: &dyn Display = &193_u8;
let remarkable: &dyn Display = &0.0072973525664;

// Poniższe procedury zwracają rozmiar/rozmieszczenie wartości, do której
// odnosi się obiekt zestawu metod, a nie samego obiektu zestawu metod.
// Informacje te są zawarte w tablicy vtable, do której odnosi się obiekt itself.
// zestawu metod
assert_eq!(std::mem::size_of_val(unremarkable), 1);
assert_eq!(std::mem::align_of_val(remarkable), 8);
```

Operacje arytmetyczne na wskaźnikach

Język Rust umieszcza elementy tablicy, podzbioru lub wektora jako pojedynczy, sąsiadujący blok pamięci, co pokazaliśmy na rysunku 22.1. Elementy te są rozmieszczone w regularnych odstępach, tak że jeśli każdy element zajmuje `rozmiar` bajtów, to *i*-ty element ma początek w `i * rozmiar` bajcie.



Rysunek 22.1. Tablica umieszczona w pamięci

Jedną z przydatnych konsekwencji takiego stanu rzeczy jest fakt, że jeśli korzystasz z dwóch wskaźników niechronionych wskazujących elementy tablicy, to porównywanie wskaźników będzie dawało takie same rezultaty jak porównywanie indeksów elementów: jeżeli `i > j`, to wskaźnik niechroniony do *i*-tego elementu jest mniejszy od wskaźnika niechronionego do *j*-tego elementu. Dzięki temu wskaźniki niechronione okazują się przydatne jako granice przeglądania całej tablicy. W istocie dostępny w bibliotece standardowej prosty iterator podzbioru początkowo był zdefiniowany następująco:

```
struct Iter<'a, T> {
    ptr: *const T,
    end: *const T,
    ...
}
```


Kup książkę Poleć książkę

Pole `ptr` wskazuje następny element, jaki powinien zostać uzyskany w wyniku iteracji, natomiast pole `end` pełni rolę granicy: gdy `ptr == end`, następuje zakończenie iteracji.

Inny przydatny skutek takiej struktury tablicy jest następujący: jeżeli `element_ptr` jest wskaźnikiem niechronionym `*const T` lub `*mut T` wskazującym *i*-ty element jakiejś tablicy, to `element_ptr.offset(o)` stanowi wskaźnik niechroniony do *(i + o)*-tego elementu. Definicja ta jest równoważna listingowi:

```
fn offset<T>(ptr: *const T, count: isize) -> *const T
    where T: Sized
{
    let bytes_per_element = std::mem::size_of::<T>() as isize;
    let byte_offset = count * bytes_per_element;
    (ptr as isize).checked_add(byte_offset).unwrap() as *const T
}
```

Funkcja `std::mem::size_of::<T>` zwraca rozmiar typu `T` w bajtach. Z definicji `isize` jest wystarczająco duży, aby przechowywać adres, możesz przekształcić znacznik bazowy do tego typu, przeprowadzić operacje arytmetyczne na tej wartości, a następnie dokonać konwersji wyniku z powrotem do postaci znacznika.

Nic nie stoi na przeszkodzie, aby wygenerować wskaźnik w pierwszym bajcie za tablicą. Nie możesz tworzyć dereferencji takiego wskaźnika, bywa jednak przydatny do symbolizowania granicy pętli czy do sprawdzania wartości.

Jednak wykorzystanie metody `offset` do utworzenia wskaźnika poza tym punktem lub przed początkiem tablicy stanowi działanie niezdefiniowane, nawet jeżeli nie tworzysz dereferencji tego wskaźnika. W celu optymalizacji Rust zakłada, że `ptr.offset(i) > ptr`, gdy wartość `i` jest dodatnia, a `ptr.offset(i) < ptr` dla wartości ujemnej `i`. Założenie to wydaje się bezpieczne, ale może okazać się błędne, jeżeli nastąpi przeciążenie wartości `isize` przez metodę `offset`. Jeśli wartość `i` zostanie ograniczona do tej samej tablicy co `ptr`, zażegnamy ryzyko przeciążenia — mimo wszystko sama w sobie tablica nie jest w stanie przeciążyć granic przestrzeni adresowej (aby zapewnić bezpieczeństwo umieszczania wskaźników w pierwszym bajcie za tablicą, Rust nigdy nie umieszcza wartości w górnym końcu przestrzeni adresowej).

Jeżeli musisz przesunąć wskaźniki poza granice powiązanej z nimi tablicy, masz do dyspozycji metodę `wrapping_offset`. Jest ona równoważna metodzie `offset`, z tym że Rust nie przyjmuje założeń na temat względnej kolejności wskaźników `ptr.wrapping_offset(i)` i `ptr`. Rzecz jasna nadal nie możesz tworzyć dereferencji takich wskaźników, dopóki nie będą znajdować się w tablicy.

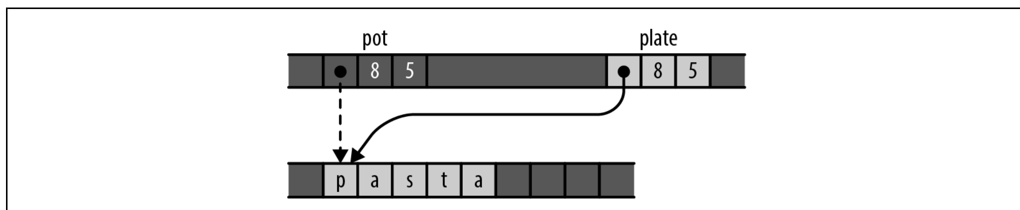
Wchodzenie do pamięci i wychodzenie z pamięci

Jeżeli implementujesz typ samodzielnie zarządzający swoją pamięcią, musisz śledzić obszary pamięci przechowujące używane wartości, a także obszary niezainicjalizowane, podobnie jak Rust czyni w przypadku zmiennych lokalnych. Weźmy pod uwagę następujący kod:

```
let pot = "pasta".to_string();
let plate;
```

Po jego uruchomieniu będziemy mieli do czynienia z sytuacją zaprezentowaną na rysunku 22.2.

Kup ksi k Pole ksi k



Rysunek 22.2. Przenoszenie łańcucha znaków od jednej zmiennej lokalnej do drugiej

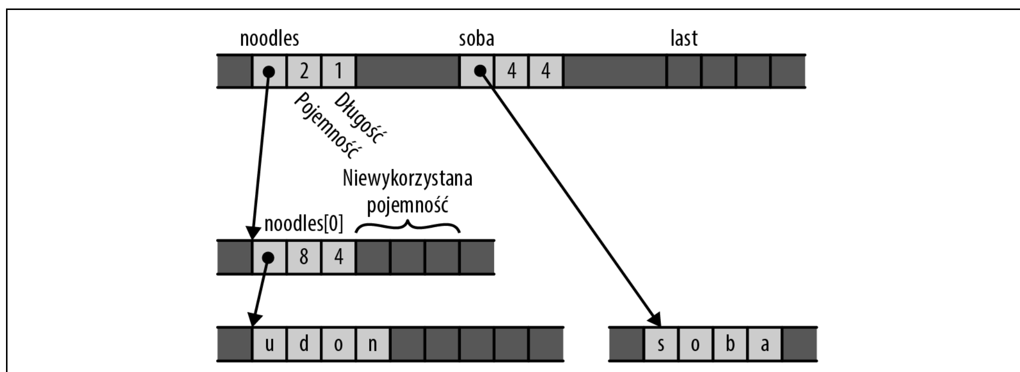
Po wykonaniu operacji przydzielania wartość `pot` pozostaje niezainicjalizowana, natomiast prawa własności do łańcucha znaków należą do `plate`.

Na poziomie kodu maszynowego nie jest ustalone, jak dany ruch wpływa na źródło, w praktyce jednak najczęściej nie ma żadnego wpływu. Po operacji przydzielania wartość `pot` prawdopodobnie ciągle przechowuje wskaźnik, pojemność i długość łańcucha znaków. Oczywiście traktowanie jej jako aktywnej wartości byłoby katastrofalne, dlatego Rust sprawia, że tak się nie dzieje.

Te same kwestie dotyczą struktur danych samodzielnie zarządzających swoją pamięcią. Załóżmy, że uruchamiamy następujący kod:

```
let mut noodles = vec!["udon".to_string()];
let soba = "soba".to_string();
let last;
```

Otrzymany stan pamięci jest widoczny na rysunku 22.3.



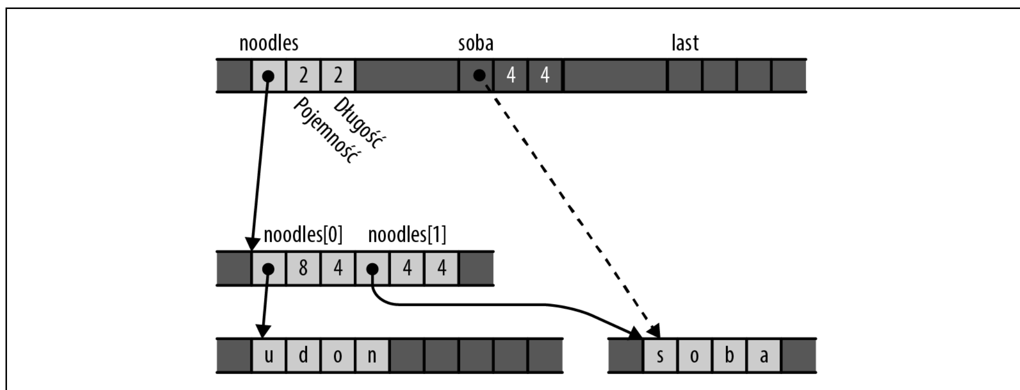
Rysunek 22.3. Wektor z niezainicjalizowaną, dodatkową pojemnością

Wektor ten zawiera dodatkową pojemność umożliwiającą przechowywanie jeszcze jednego elementu, jest ona jednak wypełniona śmieciami, prawdopodobnie pozostałościami poprzedniego użycia pamięci. Powiedzmy, że skorzystamy z następującego kodu:

```
noodles.push(soba);
```

Umieszczenie łańcucha znaków w wektorze przekształci uprzednio niewykorzystaną pamięć w nowy element (rysunek 22.4).

Kup ksi k   Pole ksi k



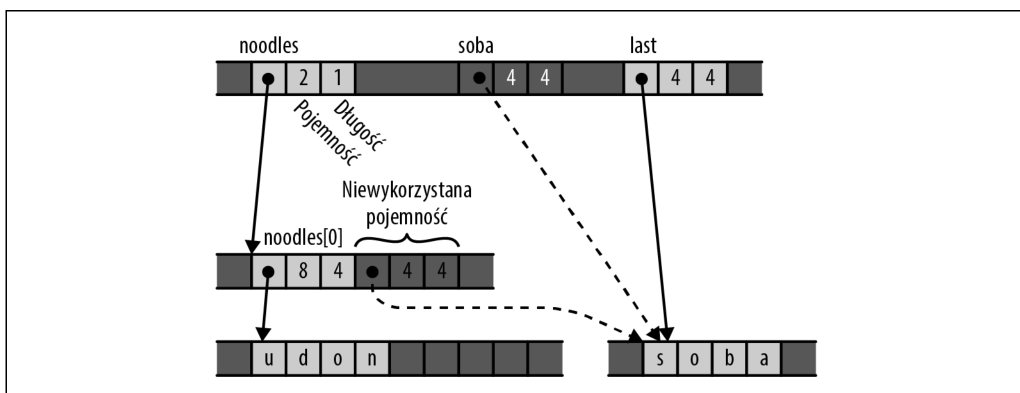
Rysunek 22.4. Sytuacja po umieszczeniu wartości soba w wektorze

Wektor zainicjalizował pustą przestrzeń, w której teraz przechowywany jest łańcuch znaków, a także zwiększył swoją długość tak, aby było wiadomo, że występuje w nim nowy element. Teraz wektor ma prawa własności do łańcucha znaków; możemy odnosić się do jego drugiego elementu, a usunięcie wektora zwolniłoby obydwa łańcuchy znaków. Z kolei wartość `soba` jest niezainicjalizowana.

Zastanówmy się jeszcze, co by się stało, gdybyśmy wyciągnęli wartość z wektora:

```
last = noodles.pop().unwrap();
```

Pamięć wyglądałaby teraz tak, jak na rysunku 22.5.



Rysunek 22.5. Sytuacja po przeniesieniu elementu z wektora do zmiennej last

Zmienna `last` przejęła prawa własności do łańcucha znaków. Długość wektora uległa zmniejszeniu, co wskazuje, że przestrzeń przeznaczona uprzednio do przechowywania łańcucha znaków jest teraz niezainicjalizowana.

Podobnie jak wcześniej w przypadku zmiennych `pot` i `pasta`, zmienne `soba` i `last` oraz niezajęta pamięć wektora prawdopodobnie cechują się identycznymi wzorcami bitowymi. Jedynie zmienna `last` jest uznawana za mającą prawa własności do wartości. Traktowanie dwóch pozostałych rejonów jako aktywnych byłoby błędem.

Kup ksi k
Pole ksi k

Zgodnie z właściwą definicją zmienna zainicjalizowana jest *traktowana jako aktywna*. Zapis w bajtach wartości jest zazwyczaj niezbędnym etapem inicjalizacji, ale wyłącznie dlatego, że czynność ta przygotowuje wartość do traktowania jej jako aktywną. Z punktu widzenia pamięci przeniesienie i skopiowanie dają takie same efekty — różnica pomiędzy nimi polega na tym, że po przeniesieniu źródło nie jest już traktowane jako aktywne, natomiast po skopiowaniu aktywny będzie zarówno obszar źródłowy, jak i obszar docelowy.

W czasie kompilacji Rust śledzi, czy zmienne lokalne są aktywne, i uniemożliwia korzystanie ze zmiennych, których wartości zostały gdzieś przeniesione. Takie typy jak `Vec`, `HashMap`, `Box` itd. dynamicznie śledzą swoje bufory. Jeżeli implementujesz typ samodzielnie zarządzający własną pamięcią, to także musisz się o to zatroszczyć.

Rust zawiera dwie zasadnicze operacje służące do implementacji takich typów:

`std::ptr::read(src)`

Przenosi wartość z lokalizacji wskazywanej przez `src` i przenosi prawa własności na kod wywołujący. Argument `src` powinien mieć postać wskaźnika niechronionego `*const T`, gdzie `T` stanowi typ o określonym rozmiarze. Po wywołaniu funkcji `read` zawartość *src pozostanie niezmieniona, jednak z wyjątkiem przypadków, gdy `T` implementuje `Copy`, należy zapewnić, że program będzie traktować zawartość `*src` jako niezainicjalizowaną.

Właśnie ta operacje kryje się za `Vec::pop`. Wywoływana jest tu funkcja `read` wyciągająca wartość z bufora, a następnie następuje zmniejszenie jego długości, dzięki czemu wiadomo, że przestrzeń ta ma niezainicjalizowaną pojemność.

`std::ptr::write(dest, value)`

Przenosi `value` do lokalizacji wskazywanej przez `dest`, która przed wywołaniem musi być niezainicjalizowaną pamięcią. Od tej chwili prawa własności do wartości ma referencja pierwotna. W tym przypadku `dest` musi być wskaźnikiem niechronionym `*mut T`, a `value` wartością `T`, gdzie `T` oznacza typ o określonym rozmiarze.

Operacja ta stanowi podstawę funkcji `Vec::push`. Zostaje tu wywołana funkcja `write` przenosząca wartość do kolejnej dostępnej przestrzeni, a potem zwiększeniu ulega długość tej przestrzeni, co wskazuje na występowanie w niej prawidłowego elementu.

Obydwie funkcje są statyczne, a nie metody powiązane z typami wskaźników niechronionych.

Należy zauważyć, że nie można realizować tych operacji na dowolnych bezpiecznych typach wskaźników. Wymagają one nieprzerwanego zainicjalizowania ich referencji pierwotnych, zatem przekształcenie niezainicjalizowanej pamięci w wartość lub odwrotnie wykracza poza ich możliwości. W tym kontekście wskaźniki niechronione spełniają nasze oczekiwania.

Biblioteka standardowa zawiera również funkcje umożliwiające przenoszenie tablic z wartościami pomiędzy blokami pamięci:

`std::ptr::copy(src, dst, count)`

przenosi tablicę `count` w pamięci od `src` do `dst`. Przypomina to pętlę wywołań `read` i `write` przenoszących kolejno te wartości. Pamięć docelowa musi być niezainicjalizowana przed wywołaniem, natomiast po wywołaniu pamięć źródłową należy pozostawić niezainicjalizowaną.

Argumenty `src` i `dest` muszą być wskaźnikami niechronionymi `*const T` i `*mut T`, natomiast `count` musi być typu `usize`.

`ptr``.copy_to(`*`dst`*`, `*`count`*`)`

Wygodniejsza wersja funkcji `copy`, która przenosi tablicę o długości `count` elementów, przy czym miejsce, w którym rozpoczyna się kopiowany obszar, jest określone przez `ptr`, a nie przez argument.

`std::ptr::copy_nonoverlapping(src, dst, count)`

Przypomina wywołanie operacji `copy`, jednak w tym przypadku kontrakt wymaga, aby bloki źródłowy i docelowy pamięci nie nakładały się na siebie. Operacja ta może być nieco szybsza od wywołania `copy`.

`ptr``.copy_to_nonoverlapping(dst, count)`

To wygodniejsza wersja funkcji `copy_nonoverlapping`, podobna do `copy_to`.

Istnieją jeszcze dwie rodziny funkcji `read` i `write`, również umieszczone w module `std::ptr`:

`read_unaligned` i `write_unaligned`

przypominają standardowe funkcje `read` i `write`, ale tutaj wskaźnik nie musi być rozmieszczony zgodnie z wymogami typu referencji pierwotnej. Funkcje te mogą być nieco wolniejsze od ich tradycyjnych odpowiedników.

`read_volatile` i `write_volatile`

Sstanowią odpowiedniki ulotnych operacji odczytu i zapisu w językach C i C++.

Przykład: GapBuffer

Poniżej prezentujemy przykład wykorzystania omówionych funkcji wskaźników niechronionych.

Załóżmy, że piszesz edytor tekstu i szukasz typu reprezentującego tekst. Mógłbyś wybrać typ `String` i korzystać z metod `insert` i `remove` do wstawiania lub usuwania znaków podczas wpisywania tekstu przez użytkownika. Jeżeli jednak tekst jest edytowany na początku dużego pliku, metody te okazują się kosztowne: wstawienie nowego znaku oznacza przesunięcie w prawo całej reszty łańcucha znaków w pamięci, natomiast usunięcie znaku wiąże się z przesunięciem łańcucha znaków w lewo. Tego typu operacje powinny być tańsze.

Edytor tekstu Emacs wykorzystuje prostą strukturę danych zwaną *buforem odstępu* (ang. *gap buffer*), umożliwiającą bezustanne wstawianie i usuwanie znaków. Typ `String` przechowuje całą zapasową pojemność na końcu tekstu, przez co operacje `push` i `pop` nie są kosztowne, natomiast w przypadku bufora odstępu pojemność taka mieści się wewnątrz tekstu, w obszarze przeprowadzania edycji. Taka dodatkowa pojemność nosi nazwę *odstępu*. Wstawianie znaków i ich usuwanie w obszarze odstępu jest tanie — w razie potrzeby wystarczy go powiększać lub zmniejszać. Możesz przenosić odstęp do dowolnej lokalizacji poprzez przesuwanie tekstu z jednego końca odstępu do drugiego. Gdy odstęp jest pusty, następuje przeniesienie do większego bufora.

Kup ksi k Pole ksi k

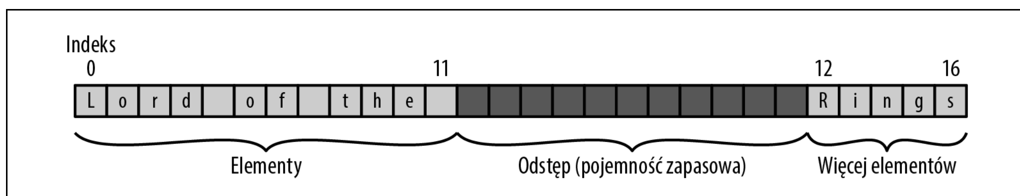
Operacje wstawiania znaków do bufora odstępu i ich usuwania są szybkie, zmiana ich położenia wymaga przeniesienia odstępu na nową pozycję. Przesuwanie elementów zajmuje czas proporcjonalny do pokonanej odległości. Na szczęście typowe czynności edycyjne wiążą się z wprowadzaniem kilku zmian w sąsiedztwie bufora przed przejściem do odległego fragmentu tekstu.

W niniejszym punkcie zaimplementujemy bufor odstępu w środowisku Rusta. Aby nie zawracać sobie głowy kodowaniem UTF-8, sprawimy, że bufor będzie bezpośrednio przechowywał wartości `char`, jednak zasady przeprowadzania tej operacji są takie same w przypadku przechowywania tekstu w jakimkolwiek innym formacie.

Najpierw zaprezentujemy działanie bufora odstępu. Poniższy kod tworzy `GapBuffer`, wstawia do niego jakiś tekst, a następnie przenosi punkt wstawiania tuż przed ostatni wyraz:

```
let mut buf = GapBuffer::new();
buf.insert_iter("Lord of the Rings".chars());
buf.set_position(12);
```

Po uruchomieniu powyższego kodu bufor będzie wyglądał tak, jak na rysunku 22.6.

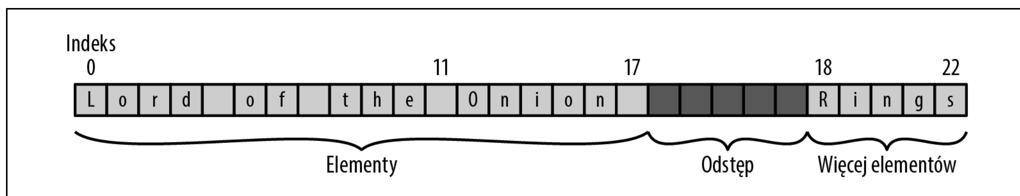


Rysunek 22.6. Bufor odstępu zawierający jakiś tekst

Wstawienie to kwestia wypełnienia odstępu nowym tekstem. Za pomocą poniższego kodu dodajemy słowo i rujnujemy cały film[2]:

```
buf.insert_iter("Onion ".chars());
```

Otrzymujemy w ten sposób stan ukazany na rysunku 22.7.



Rysunek 22.7. Bufor odstępu zawierający jeszcze więcej tekstu

Tak wygląda typ `GapBuffer`:

```
use std;
use std::ops::Range;

pub struct GapBuffer<T> {
```

[2] Czyli z „Władcy Pierścieni" otrzymujemy „Władcę krążków cebulowych" — *przyp. tłum.*

Kup ksi k Pole ksi k

```
    // Przechowalnia elementów. Ma ona wymaganą przez nas pojemność, ale jej
    // długość zawsze jest równa 0. GapBuffer umieszcza swoje elementy i odstęp
    // w tej „nieużywanej” pojemności `Vec`
    storage: Vec<T>,

    // Zakres niezainicjalizowanych elementów w środku `przechowalni`.
    // Elementy znajdujące się przed tym zakresem i po nim są zawsze inicjalizowane
    gap: Range<usize>
}
```

`GapBuffer` wykorzystuje pole `storage` w dziwny sposób[3]. W rzeczywistości nigdy nie przechowuje elementów w wektorze (a właściwie przechowuje, ale niezupełnie). Po prostu wywołuje `Vec::with_capacity(n)`, aby uzyskać wystarczająco duży blok pamięci, by pomieścić `n` wartości, otrzymuje wskaźniki niechronione do tej pamięci poprzez metody `as_ptr` i `as_mut_ptr` wektora, a następnie wykorzystuje bufor do własnych celów. Długość wektora jest zawsze zerowa. Gdy wektor `Vec` przestaje być użyteczny, nie stara się zwalniać swoich elementów, gdyż nie wie, że jakieś zawiera, ale za to zwalnia blok pamięci. Tego właśnie chce `GapBuffer`; zawiera własną implementację operacji `Drop` mającą dostęp do informacji o rozmieszczeniu używanych elementów, dzięki czemu może je prawidłowo porzucać.

Najprostsze metody modułu `GapBuffer` nie stanowią zaskoczenia:

```
impl<T> GapBuffer<T> {
    pub fn new() -> GapBuffer<T> {
        GapBuffer { storage: Vec::new(), gap: 0..0 }
    }

    /// Zwraca liczbę elementów, które ten moduł GapBuffer może przechowywać
    /// bez potrzeby realokacji
    pub fn capacity(&self) -> usize {
        self.storage.capacity()
    }

    /// Zwraca liczbę elementów przechowywanych w danej chwili
    pub fn len(&self) -> usize {
        self.capacity() - self.gap.len()
    }

    /// Zwraca bieżącą pozycję wstawiania
    pub fn position(&self) -> usize {
        self.gap.start
    }

    ...
}
```

W ten sposób wiele kolejnych funkcji zawiera metody zwracające wskaźnik niechroniony do elementu bufora w danym indeksie. W Ruście potrzebna jest jedna taka metoda dla wskaźników `mut` i jedna dla wskaźników `const`. W przeciwieństwie do wcześniej zaprezentowanych metod te nie są publiczne. Dalsza część bloku `impl` wygląda następująco:

[3] Istnieją lepsze rozwiązania wykorzystujące typ `RawVec` z paczki `alloc`, jest ona jednak ciągle niestabilna.

Kup ksi k Pole ksi k

```
/// Zwraca wskaźnik do `index`-owego elementu przechowalni
/// bez względu na odstęp
///
/// Bezpieczeństwo: `index` musi być prawidłowym indeksem `self.storage`
unsafe fn space(&self, index: usize) -> *const T {
    self.storage.as_ptr().offset(index as isize)
}
/// Zwraca wskaźnik mutowalny do `index`-owego elementu przechowalni
/// bez względu na odstęp
///
/// Bezpieczeństwo: `index` musi być prawidłowym indeksem `self.storage`
unsafe fn space_mut(&mut self, index: usize) -> *mut T {
    self.storage.as_mut_ptr().offset(index as isize)
}
```

Aby znaleźć element o określonym indeksie, należy określić, czy indeks ten znajduje się przed odstępem czy za nim, i wybrać odpowiednie rozwiązanie:

```
/// Zwraca przesunięcie w buforze `index`-owego elementu, biorąc pod uwagę
/// odstęp. Funkcja ta nie sprawdza, czy indeks znajduje się w danym zakresie, ale
/// nigdy nie zwraca indeksu znajdującego się wewnątrz odstępu
fn index_to_raw(&self, index: usize) -> usize {
    if index < self.gap.start {
        index
    } else {
        index + self.gap.len()
    }
}

/// Zwraca referencję do `index`'-owego elementu
/// albo `None`, jeżeli `index` znajduje się poza granicami
pub fn get(&self, index: usize) -> Option<&T> {
    let raw = self.index_to_raw(index);
    if raw < self.capacity() {
        unsafe {
            // Porównaliśmy `raw` z self.capacity()
            // i index_to_raw pomija odstęp, więc to rozwiązanie jest bezpieczne
            Some(&*self.space(raw))
        }
    } else {
        None
    }
}
```

Gdy zaczynamy przeprowadzać operacje wstawiania i usuwania w różnych obszarach bufora, musimy przenieść odstęp do nowej lokalizacji. Przeniesienie odstępu w prawo oznacza przesunięcie elementów w lewo i odwrotnie, zupełnie jak pęcherzyk powietrza w poziomicy przesuwa się w jednym kierunku, podczas gdy płyn przemieszcza się w przeciwną stronę:

```
/// Wyznacza `pos` jako bieżącą pozycję wstawiania
/// Jeżeli `pos` znajduje się poza granicami, zostaje zgłoszony błąd paniki
pub fn set_position(&mut self, pos: usize) {
    if pos > self.len() {
        panic!("indeks {} znajduje się poza zakresem dla GapBuffer", pos);
    }

    unsafe {
```


Kup ksi k   Pole ksi k

```
        let gap = self.gap.clone();
        if pos > gap.start {
            // `pos` znajduje się za odstępem. Przenosi odstęp w prawo poprzez
            // przesunięcie elementów znajdujących się po odstępie przed niego
            let distance = pos - gap.start;
            std::ptr::copy(self.space(gap.end),
                           self.space_mut(gap.start),
                           distance);
        } else if pos < gap.start {
            // `pos` znajduje się przed odstępem. Przenosi odstęp w lewo poprzez
            // przesunięcie elementów znajdujących się przed odstępem za niego
            let distance = gap.start - pos;
            std::ptr::copy(self.space(pos),
                           self.space_mut(gap.end - distance),
                           distance);
        }

        self.gap = pos .. pos + gap.len();
    }
}
```

Funkcja ta wykorzystuje metodę `std::ptr::copy` do przesuwania elementów. Metoda `copy` wymaga, aby obszar docelowy nie był zainicjalizowany i pozostawia obszar źródłowy również niezainicjalizowany. Zakresy adresów źródłowego i docelowego mogą się pokrywać, ale `copy` prawidłowo zajmuje się takim przypadkiem. Ponieważ odstęp stanowi przed wywołaniem niezainicjalizowaną pamięć, a funkcja dostosowuje jego położenie do miejsca zajmowanego przez kopię, kontrakt funkcji `copy` zostaje spełniony.

Operacje wstawiania i usuwania elementu są względnie proste. Wstawianie zajmuje jeden fragment odstępu przeznaczony dla elementu, z kolei usuwanie pozbywa się jednej wartości i powiększa odstęp o jeszcze do niedawna zajętą przestrzeń:

```
/// Umieszcza `elt` w bieżącej pozycji wstawiania i
/// pozostawia za nią pozycję wstawiania
pub fn insert(&mut self, elt: T) {
    if self.gap.len() == 0 {
        self.enlarge_gap();
    }

    unsafe {
        let index = self.gap.start;
        std::ptr::write(self.space_mut(index), elt);
    }
    self.gap.start += 1;
}

/// Umieszcza elementy wygenerowane przez `iter` w bieżącej pozycji wstawiania i
/// pozostawia za nimi pozycję wstawiania
pub fn insert_iter<I>(&mut self, iterable: I)
    where I: IntoIterator<Item=T>
{
    for item in iterable {
        self.insert(item)
    }
}
```


Kup książkę Poleć książkę

```
/// Usuwa element tuż za pozycją wstawiania i zwraca ją lub
/// zwraca `None`, jeżeli pozycja wstawiania znajduje się na końcu GapBuffer
pub fn remove(&mut self) -> Option<T> {
    if self.gap.end == self.capacity() {
        return None;
    }

    let element = unsafe {
        std::ptr::read(self.space(self.gap.end))
    };
    self.gap.end += 1;
    Some(element)
}
```

Podobnie do sytuacji, w której `Vec` wykorzystuje metodę `std::ptr::write` do operacji `push` i metodę `std::ptr::read` do operacji `pop`, `GapBuffer` stosuje metodę `write` do operacji `insert` i metodę `read` do operacji `remove`. Podobnie również jak w przypadku konieczności dostosowania długości wektora `Vec` do utrzymywania granicy pomiędzy zainicjalizowanymi elementami a dodatkową pojemnością, `GapBuffer` dopasowuje odstęp.

Po zapełnieniu odstępu metoda `insert` musi powiększyć bufor, aby zachować wolne miejsce. Odpowiedzialna jest za to metoda `enlarge_gap` (ostatnia w bloku `impl`):

```
/// Podwaja pojemność `self.storage`
fn enlarge_gap(&mut self) {
    let mut new_capacity = self.capacity() * 2;
    if new_capacity == 0 {
        // Istniejący wektor jest pusty
        // Wybiera rozsądną pojemność początkową
        new_capacity = 4;
    }

    // Nie mamy pojęcia, w jaki sposób Vec zmienia rozmiar "nieużywanej" pojemności,
    // dlatego zostaje utworzony nowy wektor, do którego zostają
    // przeniesione elementy
    let mut new = Vec::with_capacity(new_capacity);
    let after_gap = self.capacity() - self.gap.end;
    let new_gap = self.gap.start .. new.capacity() - after_gap;

    unsafe {
        // Przenosi elementy znajdujące się przed odstępem
        std::ptr::copy_nonoverlapping(self.space(0),
                                      new.as_mut_ptr(),
                                      self.gap.start);

        // Przenosi elementy znajdujące się za odstępem
        let new_gap_end = new.as_mut_ptr().offset(new_gap.end as isize);
        std::ptr::copy_nonoverlapping(self.space(self.gap.end),
                                      new_gap_end,
                                      after_gap);
    }

    // Zostaje tu zwolniony stary Vec, ale żaden element nie zostaje porzucony,
    // ponieważ długość wektora wynosi 0
    self.storage = new;
    self.gap = new_gap;
}
```


Kup ksi k Pole ksi k

Funkcja set_position musi wykorzystywać metodę copy do przenoszenia elementów w tę i z powrotem wewnątrz odstępu, natomiast funkcja enlarge_gap może stosować metodę copy_nonoverlapping, gdyż przenosi elementy do zupełnie nowego bufora.

Przeniesienie nowego wektora do self.storage powoduje porzucenie starego wektora. Długość starego wektora wynosi 0, dlatego jest traktowany tak, jakby nie zawierał elementów do porzucenia, dzięki czemu bufor zostaje bezproblemowo zwolniony. Co więcej, metoda copy_nonoverlapping pozostawia niezainicjalizowane źródło, zatem takie traktowanie starego bufora jest prawidłowe: prawa własności do elementów należą teraz do nowego bufora.

Musimy w końcu zagwarantować, że porzucenie GapBuffer zwolni wszystkie jego elementy:

```
impl<T> Drop for GapBuffer<T> {
    fn drop(&mut self) {
        unsafe {
            for i in 0 .. self.gap.start {
                std::ptr::drop_in_place(self.space_mut(i));
            }
            for i in self.gap.end .. self.capacity() {
                std::ptr::drop_in_place(self.space_mut(i));
            }
        }
    }
}
```

Elementy znajdują się przed i za odstępem, zatem sprawdzamy każdy rejon i wykorzystujemy funkcję std::ptr::drop_in_place do porzucania każdego elementu. Funkcja drop_in_place przypomina działaniem funkcję drop(std::ptr::read(ptr)), nie zajmuje się jednak przenoszeniem wartości do kodu wywołującego (a zatem działa również z typami o nieokreślonym rozmiarze). Podobnie również jak w przypadku funkcji enlarge_gap, do momentu zakończenia porzucania wektora self.storage jego bufor rzeczywiście staje się niezainicjalizowany.

Tak samo jak w przypadku pozostałych typów opisywanych w tym rozdziale, GapBuffer gwarantuje, że jego własne niezmienniki wystarczą do przestrzegania kontraktu każdej wykorzystywanej przez niego właściwości niebezpiecznej, dlatego nie trzeba oznaczać żadnej z metod publicznych jako niebezpiecznej. GapBuffer implementuje bezpieczny interfejs dla właściwości, której nie można napisać skutecznie w bezpiecznym kodzie.

Bezpieczeństwo błędów paniki w kodzie niebezpiecznym

W języku Rust błędy paniki zazwyczaj nie są w stanie wywoływać działania niezdefiniowanego; makro panic! nie jest właściwością niebezpieczną. Jeżeli jednak postanowisz pracować z kodem niebezpiecznym, bezpieczeństwo błędów paniki stanie się częścią Twoich obowiązków.

Przyjrzyjmy się opisanej w poprzednim punkcie metodzie GapBuffer::remove:

```
pub fn remove(&mut self) -> Option<T> {
    if self.gap.end == self.capacity() {
        return None;
    }

    let element = unsafe {
```


Kup książkę Poleć książkę

```
            std::ptr::read(self.space(self.gap.end))
        };
        self.gap.end += 1;
        Some(element)
    }
```

Wywołanie metody `read` natychmiastowo przenosi element znajdujący się za odstępem poza bufor i pozostawia niezainicjalizowaną przestrzeń. W tym momencie bufor `GapBuffer` znajduje się w stanie niezainicjalizowanym: naruszyliśmy niezmiennik określający, że wszystkie elementy bufora poza odstępem muszą być zainicjalizowane. Na szczęście następna instrukcja powiększa odstęp, tak by objął niezainicjalizowany obszar, dzięki czemu przed zakończeniem funkcji warunek zainicjalizowania bufora ponownie będzie spełniony.

Co by się jednak stało, gdyby po wywołaniu metody `read`, ale przed dostosowaniem `self.gap.end` kod próbował wykorzystać właściwość mogącą wywołać błąd paniki, na przykład indeksowanie podzbioru? Nieoczekiwane przerwanie tej metody pomiędzy tymi dwiema czynnościami sprawiłoby, że `GapBuffer` pozostałby z niezainicjalizowanym elementem wewnątrz odstępu. Kolejne wywołanie funkcji `remove` oznaczałoby ponowną próbę użycia metody `read` — nawet zwykłe porzucenie `GapBuffer` oznaczałoby próbę porzucenia tego elementu. Obydwie te możliwości okazują się działaniem niezdefiniowanym, ponieważ próbują uzyskać dostęp do niezainicjalizowanej pamięci.

Dla metod typu niemal nieunikniona jest konieczność chwilowego rozluźnienia niezmienników typu podczas ich działania, a następnie przywrócenia wszystkiego do uprzedniego ładu, zanim powrócą te niezmienniki. Błąd paniki w trakcie przetwarzania metody mógłby brutalnie skrócić ten etap porządkowania, przez co typ znalazłby się w niestabilnym stanie.

Jeżeli typ korzysta wyłącznie z bezpiecznego kodu, to taka niestabilność może spowodować nieprawidłowe działanie, ale nie wywoła działania niezdefiniowanego. Jednak w przypadku kodu używającego funkcji niebezpiecznych przeważnie niezmienniki muszą spełniać wymogi kontraktów tych właściwości. Wadliwe niezmienniki prowadzą do łamania kontraktów, a zatem do działania niezdefiniowanego.

Podczas pracy z właściwościami niebezpiecznymi musisz poświęcić szczególną uwagę na wykrywanie takich wrażliwych obszarów, w których niezmienniki chwilowo przestają być spełnione, i zadbać, by nie wywoływały one błędów paniki.

Ponowna interpretacja pamięci z użyciem unii

Rust udostępnia wiele użytecznych abstrakcji, jednak oprogramowanie, które piszemy, w efekcie i tak sprowadza się do operowania i przenoszenia bajtów w pamięci. Unie są jednym z najpotężniejszych mechanizmów Rusta służących do manipulowania bajtami oraz wybierania sposobu ich interpretowania. Na przykład dowolną kolekcję 32 bitów, czyli 4 bajty, można interpretować jako liczbę całkowitą lub jako liczbę zmiennoprzecinkową.

Każda z tych interpretacji jest prawidłowa, choć zastosowanie jednej z nich tam, gdzie należy użyć drugiej, doprowadzi zapewne do bezsensownych wyników.

Kup ksi k Pole ksi k

Unię, reprezentującą kolekcję bajtów, które można potraktować jako wartość całkowitą lub zmiennoprzecinkową, można zapisać w następujący sposób:

```
union FloatOrInt {
    f: f32,
    i: i32
}
```

Powyższa unia ma dwa pola: `f` oraz `i`. Ich wartości można przypisywać w taki sam sposób jak w strukturach, jednak w przeciwieństwie do struktur podczas tworzenia unii musimy wybrać tylko jedno z pól. W przypadku struktur ich poszczególne pola odwołują się do różnych miejsc w pamięci, natomiast w przypadku unii poszczególne pola reprezentują inne interpretacje tej samej sekwencji bitów. Przypisanie wartości innemu polu unii oznacza jedynie nadpisanie wszystkich lub tylko niektórych spośród tych bitów, zgodnie z typem użytego pola. W poniższym przykładzie `one` odwołuje się do pojedynczego obszaru pamięci o wielkości 32 bitów, który najpierw zawiera wartość 1 zakodowaną jako zwyczajna liczba całkowita, a następnie zawiera wartość `1.0` zakodowaną jako liczba zmiennoprzecinkowa w formacie IEEE 754. Przypisanie wartości polu `f` oznacza natychmiastowe nadpisanie wartości zapisanej wcześniej w zmiennej `one` typu `FloatOrInt`:

```
let mut one = FloatOrInt { i: 1 };
assert_eq!(unsafe { one.i }, 0x00_00_00_01);
one.f = 1.0;
assert_eq!(unsafe { one.i }, 0x3F_80_00_00);
```

Z tego samego powodu wielkością unii jest rozmiar jej największego pola. Na przykład poniższa unia ma 64 bity wielkości, choć pole `SmallOrLarge::s` jest typu `bool`:

```
union SmallOrLarge {
    s: bool,
    l: u64
}
```

O ile tworzenie unii oraz przypisywanie wartości jej polom zawsze są operacjami bezpiecznymi, to odczyt wartości pól unii zawsze jest operacją niebezpieczną:

```
let u = SmallOrLarge { l: 1337 };
println!("{}", unsafe {u.l}); // Wyświetla 1337
```

Wynika to z faktu, że w odróżnieniu od typów wyliczeniowych unie nie mają znacznika. Kompilator nie dodaje do nich żadnych dodatkowych bitów, które umożliwiłyby rozróżnianie wariantów. Nie ma możliwości określenia w trakcie wykonywania kodu, czy konkretna wartość typu `SmallOrLarge` powinna być interpretowana jako `u64` czy jako `bool`, chyba że określi to jakiś dodatkowy kontekst programu.

Nie ma żadnej gwarancji, że wzorzec bitów danego pola będzie prawidłowy. Na przykład zapisanie wartości w polu `l` zmiennej `SmallOrLarge` nadpisze wartość pola `s` i utworzy wzorzec bitów, który na pewno nie będzie oznaczał nic użytecznego, a najprawdopodobniej nawet nie będzie prawidłową wartością typu `bool`. Dlatego, choć zapisywanie pól unii jest bezpieczne, ich odczyt zawsze wymaga użycia `unsafe`. Odczyt wartości pola `u.s` jest dozwolony wyłącznie w przypadku, gdy jego bity tworzą prawidłową wartość typu `bool`, w przeciwnym razie będzie to działanie niezdefiniowane.

Kup ksi k Pole ksi k

Pomimo tych wszystkich ograniczeń unie mogą być użytecznym sposobem tymczasowej zmiany interpretacji danych, zwłaszcza w przypadku wykonywania obliczeń nie na samych wartościach, lecz na ich reprezentacjach. Na przykład przedstawionego wcześniej typu `FloatOrInt` z powodzeniem można użyć do wyświetlania bitów wartości zmiennoprzecinkowej, choć sam typ `f32` nie udostępnia formatowania binarnego:

```
let float = FloatOrInt { f: 31337.0 };
// Wyświetla 1000110111101001101001000000000
println!("{:b}", unsafe { float.i });
```

Te proste przykłady na pewno będą działać zgodnie z oczekiwaniami na wszystkich wersjach kompilatora, jednak nie ma żadnych gwarancji, że pola unii będą się rozpoczynać w określonym miejscu — aby je uzyskać, należy dodać do definicji unii specjalny atrybut, który poinstruuje kompilator, jak rozmieścić pola unii w pamięci. Dodanie atrybutu `#[repr(C)]` zapewni, że wszystkie pola unii będą się zaczynać w miejscu o przesunięciu 0, a nie tam, gdzie umieści je kompilator. Dysponując tymi gwarancjami, można już użyć nadpisywania wartości do pobierania poszczególnych bitów, na przykład bitu znaku liczby całkowitej:

```
#[repr(C)]
union SignExtractor {
    value: i64,
    bytes: [u8; 8]
}

fn sign(int: i64) -> bool {
    let se = SignExtractor { value: int};
    println!( "{:b} ({:?})", unsafe { se.value }, unsafe { se.bytes });
    unsafe { se.bytes[7] >= 0b10000000 }
}

assert_eq!(sign(-1), true);
assert_eq!(sign(1), false);
assert_eq!(sign(i64::MAX), false);
assert_eq!(sign(i64::MIN), true);
```

W tym przypadku bitem znaku jest najbardziej znaczący bit najbardziej znaczącego bajta. Procesory o architekturze x86 stosują format *little-endian* — najmniej znaczący bajt jest zapisywany jako pierwszy — zatem kolejność bitów jest odwrócona: najbardziej znaczącym bajtem nie jest `bytes[0]`, lecz `bytes[7]`. Nie jest to coś, czym zazwyczaj trzeba się przejmować w kodzie Rusta, jednak ten przykład bezpośrednio korzysta z reprezentacji wartości `i64` w pamięci, dlatego te szczegóły niskiego poziomu są istotne.

Unie nie wiedzą, w jaki sposób usuwać swoją zawartość, dlatego wszystkie pola unii muszą być typów implementujących zestaw metod `Copy`. Niemniej jeśli koniecznie musimy zapisać w unii wartość typu `String`, to istnieje sposób, by to zrobić: wystarczy poszukać w dokumentacji biblioteki standardowej informacji o strukturze `std::mem::ManuallyDrop`.

Kup ksi k Pole ksi k

Dopasowywanie unii

Dopasowywanie unii przypomina dopasowywanie struktur, z tym że każdy wzorzec musi określać dokładnie jedno pole:

```
unsafe {
    match u {
        SmallOrLarge { s: true } => { println!("logiczna prawda"); }
        SmallOrLarge { l: 2 } => { println!("wartość całkowita 2"); }
        _ => { println!("coś innego"); }
    }
}
```

Warianty wyrażenia `match`, które można dopasować do wariantu unii bez określonej wartości, zawsze zostaną dopasowane. Poniższy przykład zawsze doprowadzi do działania niezdefiniowanego, jeśli ostatnim zapisanym polem unii `u` było `u.i`:

```
// Działanie niezdefiniowane!
unsafe {
    match u {
        FloatOrInt { f } => { println!("float {}", f) },
        // Ostrzeżenie: wzorzec nieosiągalny
        FloatOrInt { i } => { println!("int {}", i) }
     }
}
```

Pożyczanie unii

Pożyczenie jednego pola unii powoduje pożyczenie jej całej. Oznacza to, że zgodnie ze zwyczajnymi regułami pożyczania utworzenie mutowalnej referencji do pola unii uniemożliwia tworzenie innych referencji zarówno do tego, jak i do wszystkich innych pól unii; oprócz tego utworzenie niemutowalnej referencji do pola uniemożliwia tworzenie referencji mutowalnych.

Jak się przekonamy w następnym rozdziale, Rust ułatwia tworzenie bezpiecznych interfejsów nie tylko do własnego niebezpiecznego kodu, lecz także do kodu napisanego w innych językach programowania. Taki kod, jak sama nazwa wskazuje, może stanowić potencjalne zagrożenie, jednak używany ostrożnie i rozważnie, umożliwia tworzenie kodu o bardzo wysokiej wydajności, a jednocześnie dającego wszystkie gwarancje kodu Rusta.

Kup ksi k Pole ksi k

Skorowidz

A

adapter chain, 370
adapter cloned, 373
adapter cycle, 373
adapter enumerate, 370
adapter filter, 359
adapter filter_map, 361
adapter flat_map., 361
adapter fuse, 367
adapter inspect, 369
adapter map, 358
adapter peekable, 366
adapter rev, 369
adapter skip, 365
adapter skip_while, 365
adapter take, 364
adapter take_while, 364
adapter zip, 371
adaptery, 358
algorytmy haszujące, 421
alias typu Result, 172
all, 377
alokacja warunkowa, 445
any, 377
architektura Flux, 348
ASCII, 90, 425, 619
AsMut, 320
AsRef, 320
atomowe operacje, 491
atrybuty, 200

B

bezpieczeństwo, 58, 338
bezpieczeństwo błędów paniki, 646
bezpieczeństwo referencji, 125
bezpieczeństwo wątków, 511
biblioteka, 197
biblioteka libgit2, 660, 666
biblioteka Rayon, 499
biblioteka unicode-normalization, 464
BinaryHeap<T>, 392, 409
blokady odczytu/zapisu, 523
blokady typu muteks, 517
bloki kodu, 150
bloki unsafe, 617
błąd panic, 167
błędy, 167
  ignorowanie, 178
  jednoczesna obsługa, 175, 553, 559
  propagacja, 174
  przechwytywanie, 170
  typ Result, 170, 181
  w funkcji main, 178
  własny typ, 180
  wyświetlanie informacji, 172
błędy paniki, 646
błędy różnych typów, 175
błędy składniowe, 612
Borrow, 321, 328
BorrowMut, 321
BTreeMap<K, V>, 393, 410
BTreeSet<T>, 393, 417
bufor odstępu, 640
by_ref, 372

C

Cargo.lock, 212
chain, 370
Clone, 313
cloned, 373
collect, 382
Condvar, 524
Copy, 314

Kup ksi k   Pole ksi k

count, 374
cycle, 373
cyfry, 429

D

dane binarne, 478
deadlock, 521
debugowanie, 453
debugowanie makr, 600
debugowanie wskaźników, 454
Default, 318
definiowanie metod, 222
definiowanie zależności, 210
definiowanie zestawów metod, 269
deklaracje, 151
deklaracje składników lokalnych, 152
deklarowanie obcych funkcji, 655
deklarowanie obcych zmiennych, 655
Deref, 315
dereferencja wskaźników niechronionych, 631
DerefMut, 315
doc-testy, 208
dokumentacja, 202, 205
dołączanie tekstu, 433
domknięcia, 166, 330
  bezpieczeństwo, 338
  Fn, 341
  FnMut, 341
  FnOnce, 341
  pożyczające wartość, 332
  przejmujące własność, 333
  typy, 334
  wydajność, 336
dopasowanie do wielu wartości, 254
drain, 356
Drop, 307
drzewa binarne, 256, 389
dynamiczne definiowanie długości, 455
dynamiczne definiowanie precyzji, 455
dziedziczenie, 275
dziedziczenie metod, 231

E

elementy, 161
Entry, 414
enumerate, 370
ExactSizeIterator, 377
Extend, 384

F

filter, 358
filter_map, 361
find, 381
flat_map, 361
Flux, 348
FnMut, 340
FnOnce, 338
fold, 378
formatowanie liczb, 450
formatowanie tekstu, 449
formatowanie wartości, 447, 453
formatowanie własnych typów, 456
formatowanie wskaźników, 454
From, 323
funkcja, 27
  spawn, 493
funkcja copy_to, 486
funkcja main(), 178
funkcje, 160
funkcje dostępu do systemu plików, 483
funkcje generyczne, 66, 264
funkcje niebezpieczne, 623
funkcje powiązane, 223
funkcje unsafe, 621
funkcje zwrotne, 343
fuse, 367

G

GapBuffer, 640
generyczne funkcje, 264
generyczne typy wyliczeniowe, 243
generyczny zestaw metod, 282
gruby wskaźnik, 86

H

HashMap<K, V>, 393, 410
HashSet<T>, 393, 417
haszowanie, 420

I

ignorowanie błędów, 178
implementacja interfejsu Drop, 308
implementacja własnych iteratorów, 386
implementacja zestawów metod, 269, 272, 282, 563

Kup ksi k   Pole ksi k

implementacje interfejsu IntoIterator, 353
importy, 192
indeks odwrócony, 504, 514
informacje o błędach, 172
informacje o łańcuchu znaków, 432
inspect, 369
interfejs
  FromIterator, 382
  IntoIterator, 351
interfejs AsMut, 320
interfejs AsRef<T>, 320
interfejs bezpieczny, 529, 666
interfejs Copy, 314
interfejs Default, 318
interfejs Drop, 308
interfejs Extend, 384
interfejs FnOnce, 339
interfejs funkcji obcych, 651
interfejs Index, 302
interfejs IndexMut, 302
interfejs IntoIterator, 353
interfejs niechroniony, 660
interfejsy narzędziowe, 306
Into, 323
IntoIterator, 350, 353
inżynieria wsteczna ograniczeń, 286
iter, 352
iter_mut, 352
iteracja, 406
iteracja zbioru, 418
iterator
  ExactSizeIterator, 378
Iterator, 350
iteratory, 349, 436
  konsumowanie, 374
  tworzenie, 352
  własne, 386
  źródła, 357
iteratory obustronne, 368
iterowanie map, 416
iterowanie tekstu, 438

J

język wyrażeń, 146
join, 493

K

kacze typowanie, 66
kanał synchroniczny, 511
kanały, 502
  cechy, 509
  muteksy, 522
  wydajność, 509
katalog src/bin, 198
katalogi, 480
kierunek tekstu, 427
klasyfikacja znaków, 427
klonowanie wartości, 313
kod niebezpieczny, 615, 623, 651
kolejka dwustronna, 392, 407
kolekcja
  BinaryHeap<T>, 392, 409
  BTreeMap<K, V>, 393, 410
  BTreeSet<T>, 393, 417
  HashMap<K, V>, 393, 410
  HashSet<T>, 393, 417
  Vec<T>, 392, 393
  VecDeque<T>, 392, 407
kolekcje, 391
  tworzenie, 355, 382
kolekcje niestandardowe, 423
kompresja, 478
kontrakt, 617
konwersja tekstu, 441
konwersja wartości, 442
konwersja znaku, 430
kopiec binarny, 392, 409
krotki, 78, 249

L

last, 381
Latin-1, 425
len, 378
liczby zespolone, 47
literały, 247
literały łańcuchowe, 88

Ł

łańcuch znaków, 432, 619
łańcuchy bajtów, 89
łańcuchy znaków, 90, 424
Łańcuchy znaków jako kolekcje generyczne, 447
łączenie wątków, 492

Kup ksi k

Pole ksi k

M

makra, 591
    a rekurencja, 605
    debugowanie, 600
    eksport, 610
    higieniczne, 609
    import, 610
    niezamierzone skutki, 595
    powtórzenia, 596
    proceduralne, 613
    rozwijanie, 593
    typy składników, 604
    wbudowane, 598
    zakres, 608
    zestawy metod, 606
makro format!(), 92
makro format_args!, 458
makro json!, 601
map, 358
mapa uporządkowana, 393, 410
mapa z haszowaniem, 393, 410
mapy
    iterowanie, 416
    typ Entry, 414
max, 375
max_by, 375
max_by_key, 376
metoda
    all, 377
    any, 377
    collect, 382
    find, 381
    fold, 378
    last, 381
    len, 378
    nth, 380
    partition, 384
    position, 377
    rposition, 377
metoda min_by_key, 376
metoda .join(), 495
metoda by_ref, 372
metoda clone, 313
metoda concat(), 92
metoda count, 374
metoda default, 318
metoda drain, 356
metoda iter, 352
metoda iter_mut, 352
metoda join(separator), 92
metoda max, 375
metoda max_by_key, 376
metoda min, 375
metoda min_by, 375
metoda product, 374
metoda sum, 374
metoda to_string(), 91
metody, 160, Patrz także zestaw metod
    kwalifikowana nazwa, 277
metody domyślne, 270
metody interfejsu PartialOrd, 300
metody max_by, 375
metody statyczne, 276
metody w bloku impl, 222
min, 375
min_by, 375
min_by_key, 376
moduły, 187
morze obiektów, 144
morze zsynchronizowanych obiektów, 491
mut, 520
muteks, 516
muteks zatruty, 522
Mutex, 520
Mutex<T>, 518

N

typ [T, 82
nazwa metody, 277
niezdefiniowane zachowanie, 623
normalizacja, 462
nth, 380

O

obcinanie tekstu, 440
obiekt buforowany typu reader, 469
obiekt implementujący zestaw metod, 262, 263
obiekty typu reader, 466, 467
obiekty typu writer, 466, 473
obsługa błędów, 167, 495
obsługa błędów w funkcji main(), 178
obsługa cyfr, 429
obsługa jednoczesna błędów, 175, 553, 559
obsługa sieci, 487
obszary robocze, 215

Kup ksi k   Pole ksi k

odczyt wartości, 507
odczyt zawartości katalogu, 484
odstęp, 640
odwinięcie stosu, 168
operacja przypisania, 121
operacje arytmetyczne na wskaźnikach, 635
operacje wejścia/wyjścia, 465
operator ::, 192
operatory, 304
    arytmetyczne, 163
    bitowe, 163
    logiczne, 163
    porównania, 163
    przeciążanie, 290
    przeciążanie, 282
operatory arytmetyczne, 291
operatory bitowe, 291
operatory dwuargumentowe, 294
operatory jednoargumentowe, 293
operatory porównania, 300
operatory przypisania złożonego, 295
operatory referencji, 163
OsStr, 480

P

paczki, 182
    publikowanie, 213
pamięć, 636
parametr cyklu życia, 129, 230
parametry cyklu życia, 136
parsowanie argumentów, 45
partition, 384
Path, 480, 481
PathBuf, 481
peekable, 366
pętla for, 156
pętla loop, 156
pętla while, 155
pętla while let, 155
pętle, 155
plik README.md, 216
pliki, 475, 480
    zapisywanie danych, 50
pliki z modułami, 190
pobieranie tekstu, 473
podzbiory, 86
podział wątków, 492
pola, 161
pola nazywane, 217
pola numerowane, 220
polimorfizm, 259
porównania szeregujące, 300
porównywanie sekwencji elementów, 376
position, 377
potok, 508
potoki, 491
potoki adapterów iteratora, 514
potoki wątków, 514
product, 374
program Mandelbrota, 51
programowanie współbieżne, 40, 490
propagacja błędów, 174
przechwytywanie błędów, 170
przechwytywanie zmiennych, 331
przeciążanie operatorów, 282, 290
przeglądanie tekstu, 470
przeniesienie własności, 100
    operacje, 105
    przepływ sterowania, 107
    struktury indeksowane, 107
przepływ sterowania, 107, 159
przerywanie procesu, 169
przymus dereferencji, 315
przypisanie, 164
publikowanie paczek, 213
pudełka, 81
pule wątków, 491

R

Rayon, 498
referencja &Self, 315
referencja mutowalna, 118
referencja współdzielona, 118
referencje, 80, 116, 163, 311
    Bezpieczeństwo, 125
    C++, 120
    do podzbiorów, 124
    do referencji, 122
    do wyrażeń, 124
    do zestawów metod, 124
    do zmiennej lokalnej, 125
    jako argumenty, 130
    jako wartość zwracana, 131
    mutowalne, 137
    Odrębny cykl życia, 134
    operacja przypisania, 121

Kup ksi k   Pole ksi k

referencje
    Porównywanie, 123
    puste, 123
    Rusta, 120
    w strukturze, 132
    współdzielone, 137
    Współdzielone, 144
referencje innego typu, 443
referencje jako wartości, 120
RefWithFlag, 632
reguła własności, 94, 96
rekurencja, 605
relacja częściowej równoważności, 299
relacja równoważności, 298
relacje między typami, 278
rev, 369
rodzaje normalizacji, 463
rozmiary typów, 634
równoległość danych, 491
rposition, 377
RwLock<T>, 523
rysowanie zbioru, 49
rzutowanie typów, 165

S

serializacja, 478
serwer web, 34
sieć, 487
sized type, 310
skip, 365
skip_while, 365
skrypt kompilacji, 659
słowo kluczowe Self, 273
spawn, 493
stan współdzielony mutowalny, 516
str, 431
String, 431
struktura literału zmiennoprzecinkowego, 74
struktura zawierająca referencje, 132
struktury, 217, 249
    reprezentacja w pamięci, 221
    stosujące typy wyliczeniowe, 241
struktury generyczne, 228
struktury indeksowane, 107
struktury puste, 221
struktury z parametrem cyklu życia, 230
struktury z polami nazywanymi, 217
struktury z polami numerowanymi, 220
sum, 374
surowy łańcuch znaków, 89
symbol $x, 597
symbol ?Sized, 312
symbol <T>, 229
symbole wieloznaczne, 254

Ś

ścieżki, 192
średniki, 150

T

tablice, 82
take, 364
take_while, 364
tekst, 424
tekst jako UTF-8, 443
test, 29
test równości, 297
testy, 202
testy integracyjne, 204
ToOwned, 327
tworzenie iteratorów, 352
tworzenie kolekcji, 355, 382
tworzenie łańcuchów znaków, 432
tworzenie referencji, 443
tworzenie tekstu, 444
typ &[T], 82
typ char, 427
typ logiczny, 76
typ łańcuchowy OsStr, 480
typ o stałym rozmiarze, 310
typ o zmiennym rozmiarze, 310
typ Path, 481
typ PathBuf, 481
typ Result, 170, 181
typ str, 431
typ String, 88, 91, 92, 431
typ Vec<T>, 82
typ wyliczeniowy w pamięci, 240
typ znakowy, 76
typy &mut [T], 82
typy atomowe, 525
typy całkowite, 68
typy domknięć, 334
typy funkcji, 334
typy generyczne, 259

Kup ksi k   Pole ksi k

typy iterowalne, 351
typy języka C, 652
typy kopiowalne, 110
typy maszynowe, 68
typy powiązane, 279
typy proste, 65
typy rozmiary, 634
typy rozmieszczanie, 634
typy wskaźnikowe, 80
typy wyliczeniowe, 237
typy wyliczeniowe generyczne, 243
typy wyliczeniowe zawierające dane, 239
typy zmiennoprzecinkowe, 73
typy znakowe, 93

U

Unicode, 90, 424, 425
unsafe, 617, 621
unsized type, 310
uruchamianie testu, 29
uruchomienie potoku, 508
uruchomienie programu, 57
usuwanie tekstu, 435
UTF-8, 90, 425
użycie wyrażeń regularnych, 460
używanie funkcji, 657

V

Vec<T>, 392, 393
VecDeque<T>, 392, 407

W

wątek działający w tle, 491
wątki
  bezpieczeństwo, 511
  łączenie, 492
  obsługa błędów, 495
  podział, 492
  współdzielenie niemutowalnych danych, 496
wektor, 392, 393
  Dostęp do elementów, 394
  Elementy losowe, 405
  iteracja, 406
  Iteracja, 395
  Łączenie, 400
  Podział, 400
  Porównywanie podzbiorów, 405
  Sortowanie, 403
  wyszukiwanie, 403
  zamiana elementów, 403
  zmniejszanie wielkości, 396
  Zwiększanie wielkości, 396
wektory, 83
wersje, 211
wiązanie argumentów, 455
wiersz poleceń, 30
własność współdzielona, 113
wskaźniki
  operacje arytmetyczne, 635
wskaźniki const w C++, 144
wskaźniki dopuszczające wartość pustą, 634
wskaźniki niechronione, 81, 628
wskaźnki słabe, 115
współbieżność, 40, 490
wstawianie tekstu, 433
wydajność, 336, 509
wyłączny dostęp, 520
wyrażenia, 146
wyrażenia regularne, 459
wyrażenia regularne w trybie leniwym, 461
wyrażenie
  if let, 154
wyrażenie if, 153
wyrażenie match, 153
wyrażenie return, 158
wysyłanie wartości, 504
wyszukiwanie, 476
wyszukiwanie tekstu, 436
wywołania funkcji i metod, 160
wzorce, 245, 248, 436
  krotki, 249
  struktury, 249
  z referencjami, 251
wzorce @, 254
wzorce nieodrzucalne, 255
wzorce odrzucalne, 256
wzorzec Model-View-Controller, 347

Z

zakleszczenie, 521
zależności, 210
zapis do pliku, 50
zarządzanie pamięcią, 636

zbiory
    iteracja, 418
    operacje, 419
    równe wartości, 418
zbiór Mandelbrota, 41, 501
zbiór uporządkowany, 417
zbiór z haszowaniem, 393, 417
zestaw metod, 259
    definiowanie, 269
    DerefMut, 315
    DoubleEndedIterator, 368
    From, 323
    generyczny, 282
    implementacja, 269, 272, 282, 563
    Into, 323
    Iterator, 350
    obiekt implementujący, 262
    przeciążanie operatorów, 291
    relacje między typami, 278
    rozszerzanie, 275
    słowo kluczowe Self, 273
    std
        marker
            Sync, 511
    stosowanie, 261
    struktura obiektu implementującego, 263
    ToOwned, 327
zestaw metod Borrow, 321
zestaw metod BorrowMut, 321
zestaw metod Extend, 384
zestawch metod
    Send, 511
zestawy metod unsafe, 626
zip, 371
zmiana wielkości liter, 430, 441
zmienna zainicjalizowana, 639
zmienne, 247
zmienne globalne, 527
zmienne lokalne, 125
zmienne warunkowe, 524
zmienność wewnętrzna, 232
znacznik, 240
znaki, 427

Ź

źródła iteratorów, 357

Kup ksi k   Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

Rust: zadbasz o najwyższą jakość oprogramowania systemowego!

Twórcy aplikacji często zapominają o kodzie systemowym, a to dzięki niemu funkcjonują system operacyjny, sterowniki, system plików czy zarządzanie pamięcią. Żadna aplikacja nie będzie działać bez poprawnego kodu systemowego. Język Rust jest dla programistów systemowych wyjątkowym narzędziem, rozwiązującym wiele znanych od dziesięcioleci problemów. Pozwala uniknąć mnóstwa powszechnie popełnianych błędów i tworzyć należytej jakości kod systemowy.

Dzięki tej książce zaczniesz kodować w języku Rust. Zrozumiesz też istotę programowania systemowego. Dowiesz się, w jaki sposób zapewnić bezpieczeństwo pamięci i wątków, a także jak sprawić, aby program był wykonywany szybko i bez błędów. Nauczysz się bezpiecznego stosowania operacji współbieżnych i poznasz zasady obsługi błędów. Przekonasz się, w jaki sposób Rust umożliwia kontrolę nad zużyciem pamięci i procesora, dodatkowo otrzymasz mnóstwo wskazówek ułatwiających tworzenie wydajnego i bezpiecznego kodu. Przewodnik jest przeznaczony głównie dla programistów systemowych, na lekturze jednak skorzystają również twórcy aplikacji, którzy dowiedzą się, jak mogą pisać lepszy, efektywniejszy i łatwiejszy w utrzymaniu kod.

W książce między innymi:

- solidne wprowadzenie do programowania w języku Rust
- podstawowe typy danych, a także pojęcia związane z własnością i pożyczaniem
- obsługa błędów, paczki i moduły
- zestawy metod i typy generyczne
- domknięcia, iteratory i programowanie asynchroniczne
- zaawansowane mechanizmy języka Rust

Jim Blandy jest programistą od 40 lat. Pracował nad takimi projektami jak GNU Emacs, GNU Guile, GDB. Obecnie pracuje dla Mozilli, zajmuje się zagadnieniami prezentacji graficznej w Firefoksie.

Jason Orendorff zajmuje się nieujawnionymi projektami związanymi z językiem Rust. Wcześniej pracował w Fundacji Mozilla. Potrafi w prosty sposób wyjaśniać trudne zagadnienia.

Leonora Tindall jest inżynierem oprogramowania. Tworzy oprogramowanie systemowe w języku Rust. Pracuje nad wieloma projektami open source. W wolnym czasie konstruuje elektronikę do syntezy dźwięku.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-9525-1

9 788328 395251

Cena: 139,00 zł