

O'REILLY®

Programowanie w języku Rust

WYDAJNOŚĆ I BEZPIECZEŃSTWO



Jim Blandy
Jason Orendorff

Helion 

Tytuł oryginału: Programming Rust: Fast, Safe Systems Development

Tłumaczenie: Adam Bochenek (wstęp, rozdz. 1 – 20), Krzysztof Sawka (rozdz. 21)

ISBN: 978-83-283-5740-2

© 2019 Helion S.A.

Authorized Polish translation of the English edition of Programming Rust
ISBN 9781491927281© 2018 Jim Blandy and 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 jakiegokolwiek 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 Helion SA 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 Helion SA nie ponoszą również
żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/prrust>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/prrust.zip>

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Wstęp	15
1. Dlaczego Rust?	19
Bezpieczeństwo typów	21
2. Pierwsze spotkanie z Rustem	25
Pobranie i instalacja Rusta	25
Prosta funkcja	28
Pisanie i uruchamianie testów	29
Obsługa argumentów wiersza poleceń	30
Prosty serwer web	34
Programowanie współbieżne	41
Czym jest zbiór Mandelbrota?	42
Parsowanie argumentów wiersza poleceń	46
Odwzorowanie pikseli na liczby zespolone	48
Rysowanie zbioru	50
Zapis obrazu do pliku	51
Program Mandelbrota działający współbieżnie	53
Uruchomienie programu	57
Przezroczyste bezpieczeństwo	57
3. Typy proste	59
Typy maszynowe	62
Typy całkowite	62
Typy zmiennoprzecinkowe	65
Typ logiczny	67
Typ znakowy	67

Krotki	69
Typy wskaźnikowe	71
Referencje	71
Pudełka	72
Wskaźniki niechronione	72
Tablice, wektory i podzbiory	72
Tablice	73
Wektory	74
Dodawanie pojedynczych elementów do wektora	77
Podzbiory	77
Typ String	79
Literały łańcuchowe	79
Łańcuchy bajtów	80
Łańcuchy znaków w pamięci	80
Typ String	82
Podstawowe cechy typu String	83
Inne typy znakowe	83
Co dalej?	84
4. Reguła własności	85
Reguła własności	87
Przeniesienie własności	91
Więcej operacji związanych z przeniesieniem własności	96
Przeniesienie własności a przepływ sterowania	97
Przeniesienie własności a struktury indeksowane	98
Typy kopiowalne	100
Rc i Arc: własność współdzielona	103
5. Referencje	107
Referencje jako wartości	111
Referencje Rusta kontra referencje C++	111
Referencje a operacja przypisania	112
Referencje do referencji	112
Porównywanie referencji	113
Referencje nigdy nie są puste	114
Referencje do wyrażeń	114
Referencje do podzbiorów i zestawów metod	115
Bezpieczeństwo referencji	115
Referencja do zmiennej lokalnej	115
Parametry w postaci referencji	118
Referencje jako argumenty	120

Referencja jako wartość zwracana	121
Struktura zawierająca referencje	122
Odrębny cykl życia	124
Pomijanie parametrów cyklu życia	125
Referencje współdzielone kontra mutowalne	127
Walka ze sztormem na morzu obiektów	134
6. Wyrażenia	137
Język wyrażeń	137
Bloki kodu i średniki	138
Deklaracje	140
if i match	141
if let	143
Pętle	144
Wyrażenie return	146
Analiza przepływu sterowania	146
Wywołania funkcji i metod	148
Pola i elementy	149
Operatory referencji	150
Operatory arytmetyczne, bitowe, porównania i logiczne	150
Przypisanie	151
Rzutowanie typów	152
Domknięcia	153
Priorytety operatorów	153
Co dalej?	156
7. Obsługa błędów	157
Błąd panic	157
Odwiniecie stosu	158
Przerywanie procesu	159
Typ Result	160
Przechwytywanie błędów	160
Alias typu Result	161
Wyświetlanie informacji o błędach	162
Propagacja błędów	163
Jednoczesna obsługa błędów różnych typów	164
Błędy, które nie powinny się zdarzyć	166
Ignorowanie błędów	167
Obsługa błędów w funkcji main()	167
Definiowanie własnego typu błędu	168
Co daje nam typ Result?	169

8. Paczki i moduły	171
Paczki	171
Profile budowania	174
Moduły	174
Umieszczanie modułów w oddzielnych plikach	175
Ścieżki i importy	177
Standardowe preludium	179
Podstawowe składniki modułów	179
Zmiana programu w bibliotekę	181
Katalog src/bin	183
Atrybuty	184
Testy i dokumentacja	186
Testy integracyjne	188
Dokumentacja	189
Doc-testy	191
Definiowanie zależności	193
Wersje	194
Cargo.lock	195
Publikowanie paczek na stronie crates.io	196
Obszary robocze	198
Więcej fajnych rzeczy	199
9. Struktury	201
Struktury z polami nazywanymi	201
Struktury z polami numerowanymi	204
Struktury puste	204
Reprezentacja struktur w pamięci	205
Definiowanie metod w bloku impl	206
Struktury generyczne	209
Struktury z parametrem cyklu życia	210
Dziedziczenie standardowych zestawów metod	211
Zmienność wewnętrzna	212
10. Typy wyliczeniowe i wzorce	217
Typy wyliczeniowe	218
Typy wyliczeniowe zawierające dane	220
Typ wyliczeniowy w pamięci	221
Większe struktury danych stosujące typy wyliczeniowe	222
Generyczne typy wyliczeniowe	223

Wzorce	226
Literały, zmienne i symbole wieloznaczne	228
Wzorce w postaci krotki lub struktury	230
Wzorce z referencjami	231
Dopasowanie do wielu wartości	233
Warunki dodatkowe	234
Wzorce @	235
Gdzie używamy wzorców	235
Wypełnianie drzewa binarnego	236
Podsumowanie	238
11. Zestawy metod (interfejsy) i typy generyczne	239
Stosowanie zestawów metod	241
Obiekt implementujący zestaw metod	242
Struktura obiektu implementującego	243
Funkcje generyczne	244
Na co się zdecydować?	247
Definiowanie i implementacja zestawów metod	249
Metody domyślne	250
Implementacja zestawów metod dla istniejących już typów	251
Zestaw metod a słowo kluczowe Self	252
Rozszerzanie zestawu metod (dziedziczenie)	254
Metody statyczne	254
W pełni kwalifikowana nazwa metody	255
Zestawy metod definiujące relacje między typami	257
Typy powiązane	257
Generyczny zestaw metod (czyli jak działa przeciążanie operatorów)	260
Zaprzyjaźnione zestawy metod (czyli jak działa generator liczb pseudolosowych)	261
Inżynieria wsteczna ograniczeń	263
Wnioski	266
12. Przeciążanie operatorów	267
Operatory arytmetyczne i bitowe	268
Operatory jednoargumentowe	270
Operatory dwuargumentowe	271
Operatory przypisania złożonego	272
Test równości	273
Porównania szeregujące	276
Interfejsy Index i IndexMut	278
Inne operatory	281

13. Interfejsy narzędziowe	283
Drop	284
Sized	287
Clone	289
Copy	290
Deref i DerefMut	291
Default	294
AsRef i AsMut	296
Borrow i BorrowMut	297
From i Into	299
ToOwned	301
Borrow i ToOwned w działaniu	302
14. Domknięcia	305
Przechwytywanie zmiennych	306
Domknięcia, które pożyczają wartość	307
Domknięcia, które przejmują własność	308
Typy funkcji i domknięć	309
Domknięcia a wydajność	311
Domknięcia a bezpieczeństwo	313
Domknięcia, które zabijają	313
FnOnce	314
FnMut	315
Funkcje zwrotne	317
Skuteczne korzystanie z domknięć	320
15. Iteratory	323
Iterator i IntoIterator	324
Tworzenie iteratorów	326
Metody iter i iter_mut	326
Implementacje interfejsu IntoIterator	326
Metoda drain	328
Inne źródła iteratorów	329
Adaptery	330
map i filter	330
filter_map i flat_map	333
scan	335
take i take_while	335
skip i skip_while	336
peekable	337
fuse	338

Iteratory obustronne i rev	339
inspect	340
chain	341
enumerate	341
zip	342
by_ref	342
cloned	344
cycle	344
Konsumowanie iteratorów	345
Proste agregaty: count, sum i product	345
max i min	345
max_by i min_by	346
max_by_key i min_by_key	346
Porównywanie sekwencji elementów	347
any i all	348
position, rposition oraz ExactSizeIterator	348
fold	349
nth	349
last	350
find	350
Tworzenie kolekcji: collect i FromIterator	350
Zestaw metod Extend	352
partition	353
Implementacja własnych iteratorów	354
16. Kolekcje	359
Przegląd kolekcji	360
Vec<T>	361
Dostęp do elementów	362
Iteracja	363
Zwiększanie i zmniejszanie wielkości wektora	364
Łączenie	367
Podział	367
Zamiana	369
Sortowanie i wyszukiwanie	370
Porównywanie podzbiorów	371
Elementy losowe	372
Reguły zapobiegające konfliktom w czasie iteracji	372
VecDeque<T>	373
LinkedList<T>	375

BinaryHeap<T>	376
HashMap<K, V> i BTreeMap<K, V>	377
Entry	380
Iterowanie map	381
HashSet<T> i BTreeSet<T>	382
Iteracja zbioru	383
Kiedy równe wartości są różne	383
Operacje dotyczące całego zbioru	383
Haszowanie	385
Niestandardowe algorytmy haszujące	386
Kolekcje standardowe i co dalej?	387
17. Tekst i łańcuchy znaków	389
Podstawy Unicode	389
ASCII, Latin-1 i Unicode	390
UTF-8	390
Kierunek tekstu	392
Znaki (typ char)	392
Klasyfikacja znaków	392
Obsługa cyfr	393
Zmiana wielkości liter	394
Konwersja znaku do i z liczby całkowitej	394
Typy String i str	395
Tworzenie łańcuchów znaków	396
Prosta inspekcja	396
Dołączanie i wstawianie tekstu	397
Usuwanie tekstu	398
Konwencje nazewnicze dotyczące wyszukiwania i iterowania	399
Wyszukiwanie tekstu i wzorce	399
Wyszukiwanie i zamiana	400
Iterowanie tekstu	401
Obcinanie	403
Zmiana wielkości liter w łańcuchach	403
Konwersja tekstu do wartości innego typu	404
Konwersja wartości innego typu do tekstu	404
Tworzenie referencji innego typu	405
Tekst jako UTF-8	406
Tworzenie tekstu na podstawie danych UTF-8	406
Alokacja warunkowa	407
Łańcuchy znaków jako kolekcje generyczne	409

Formatowanie wartości	410
Formatowanie tekstu	411
Formatowanie liczb	412
Formatowanie innych typów	414
Formatowanie wartości z myślą o debugowaniu	415
Formatowanie i debugowanie wskaźników	416
Wiązanie argumentów za pomocą indeksu i nazwy	416
Dynamiczne definiowanie długości i precyzji	417
Formatowanie własnych typów	417
Użycie języka formatowania we własnym kodzie	419
Wyrażenia regularne	421
Podstawowe użycie wyrażeń regularnych	421
Wyrażenia regularne w trybie leniwym	422
Normalizacja	423
Rodzaje normalizacji	424
Biblioteka unicode-normalization	425
18. Operacje wejścia/wyjścia	427
Obiekty typu reader i writer	428
Obiekty typu reader	429
Buforowany obiekt typu reader	431
Przeglądanie tekstu	432
Pobieranie tekstu	434
Obiekty typu writer	435
Pliki	436
Wyszukiwanie	437
Inne rodzaje obiektów reader i writer	437
Dane binarne, kompresja i serializacja	439
Pliki i katalogi	440
OsStr i Path	440
Metody typów Path i PathBuf	442
Funkcje dostępu do systemu plików	443
Odczyt zawartości katalogu	444
Funkcje bezpośrednio związane z platformą	446
Obsługa sieci	447
19. Programowanie współbieżne	451
Podział i łączenie wątków	453
spawn i join	454
Obsługa błędów w różnych wątkach	456
Współdzielenie niemutowalnych danych przez różne wątki	457

Rayon	459
Zbiór Mandelbrota raz jeszcze	461
Kanały	463
Wysyłanie wartości	464
Odczyt wartości	467
Uruchomienie potoku	468
Cechy kanałów i ich wydajność	470
Bezpieczeństwo wątków: Send i Sync	472
Współpraca iteratora i kanału	474
Potoki i co dalej?	475
Stan współdzielony mutowalny	476
Czym jest muteks?	476
Mutex<T>	478
mut i Mutex	480
Dlaczego Mutex to nie zawsze dobry pomysł?	480
Zakleszczenie (deadlock)	481
Zatruty muteks	482
Kanały z wieloma nadawcami stosujące muteksy	482
Blokady odczytu/zapisu (RwLock<T>)	483
Zmienne warunkowe (Condvar)	485
Typy atomowe	485
Zmienne globalne	487
Rust i pisanie programów wielowątkowych	489
20. Makra	491
Podstawy	492
Rozwijanie makra	493
Niezamierzone skutki	495
Powtórzenia	496
Makra wbudowane	498
Debugowanie makr	499
Makro json!	500
Typy składników	501
Makra a rekurencja	504
Makra i zestawy metod	505
Zakres i higiena	507
Import i eksport makr	509
Unikanie błędów składniowych w procesie dopasowywania	511
macro_rules! i co dalej?	512

21. Kod niebezpieczny	513
Dlaczego niebezpieczny?	514
Bloki unsafe	515
Przykład: skuteczny typ łańcucha znaków ASCII	516
Funkcje unsafe	518
Kod niebezpieczny czy funkcja niebezpieczna?	520
Niezdefiniowane zachowanie	521
Zestawy metod unsafe	523
Wskaźniki niechronione	525
Bezpieczne tworzenie dereferencji wskaźników niechronionych	528
Przykład: RefWithFlag	529
Wskaźniki dopuszczające wartość pustą	531
Rozmiary i rozmieszczanie typów	531
Operacje arytmetyczne na wskaźnikach	532
Wchodzenie do pamięci i wychodzenie z pamięci	534
Przykład: GapBuffer	537
Bezpieczeństwo błędów paniki w kodzie niebezpiecznym	543
Funkcje obce: wywoływanie kodu C i C++ w środowisku Rusta	544
Wyszukiwanie wspólnych reprezentacji danych	544
Deklarowanie obcych funkcji i zmiennych	547
Korzystanie z funkcji i bibliotek	549
Interfejs niechroniony dla biblioteki libgit2	552
Interfejs bezpieczny dla biblioteki libgit2	558
Podsumowanie	568
Skorowidz	569

Dlaczego Rust?

W pewnych warunkach — ale z myślą o takich właśnie warunkach powstał Rust — bycie dziesięć razy, a nawet dwa razy szybszym od konkurencji jest kwestią być albo nie być. Kwestią decydującą o losie systemu na rynku oprogramowania. Na rynku sprzętu komputerowego zresztą też.

— Graydon Hoare (<http://graydon.livejournal.com/236436.html>)

*Wszystkie komputery przetwarzają obecnie równoległe...
Programować współbieżnie to znaczy programować.*

— Michael McCool i współautorzy, *Structured Parallel Programming*

Błąd parsera TrueType pozwolił na włamanie do systemu i inwigilację; żyjemy w czasach, w których kwestie bezpieczeństwa dotyczą absolutnie całego oprogramowania.

— Andy Wingo (<https://twitter.com/andywingo/status/610765099498872832>)

Języki programowania systemowego mają za sobą długą drogę. Od 50 lat używamy języków wysokiego poziomu do pisania systemów operacyjnych. I przez cały ten czas mamy dwa problemy, których nie umiemy do końca rozwiązać:

- Trudno jest stworzyć w pełni bezpieczny kod. Szczególnie trudne jest prawidłowe zarządzanie pamięcią w C i C++. Użytkownicy od dziesięcioleci cierpią z powodu konsekwencji tych błędów. A spektakularne luki w bezpieczeństwie systemów znane są co najmniej od roku 1988, kiedy pojawił się robak Morrisa (ang. *Morris worm*).
- Tworzenie wielowątkowego kodu jest bardzo skomplikowane. A jedynie kod wielowątkowy jest w stanie w pełni wykorzystać możliwości współczesnego sprzętu. Nawet bardzo doświadczeni programiści podchodzą do tego zagadnienia z respektem. Współbieżność generuje nowe typy potencjalnych problemów i błędów, które na dodatek znacznie trudniej jest zlokalizować i naprawić.

Dlatego zachęcamy do poznania Rusta: bezpiecznego, współbieżnego języka programowania o wydajności porównywalnej z C i C++.

Rust to nowy język programowania systemowego opracowany przez Mozillę i związaną z nią społeczność. Podobnie jak C i C++, Rust zapewnia programistom ścisłą kontrolę nad wykorzystaniem pamięci i utrzymuje bliski związek między typami używanymi w programie a ich reprezentacją po stronie maszyny, na której program działa. Pomaga to programistom ocenić koszty

działania programu. Rust spełnia postulaty zawarte w artykule Bjarna Stroustrupa zatytułowanym *Abstraction and the C++ Machine Model*:

Generalnie implementacje C++ stosują się do zasady zerowego narzutu: jeśli czegoś nie używamy, to za to nie płacimy. A idąc dalej: tego, co zawarte jest w kodzie, nie da się zrealizować lepiej.

Do dwóch powyższych obietnic Rust dodaje jeszcze dwie własne: bezpieczne zarządzanie pamięcią i poprawnie działająca współbieżność.

Kluczem do spełnienia wszystkich tych obietnic jest nowatorska reguła własności, zasada przeniesienia własności oraz obsługa referencji. Poprawne użycie tych reguł sprawdzane jest już na etapie kompilacji kodu i stanowi trudne do przecenienia uzupełnienie dla elastycznego systemu typów języka Rust. Reguła własności zapewnia przejrzysty i łatwy do kontrolowania czas życia każdej wartości, dzięki czemu zbędne staje się wprowadzanie na poziomie języka mechanizmu odświeżania pamięci. Elastyczne i bezpieczne staje się zarządzanie innymi rodzajami zasobów, takimi jak pliki czy gniazda sieciowe. Reguła przeniesienia własności jasno określa, w jaki sposób własność obiektu transferowana jest z jednej zmiennej do innej, referencje zaś pozwalają na dostęp do wartości obiektów bez posiadania prawa własności. Ponieważ wielu programistów mogło nie spotkać się ze wspomnianymi przed chwilą pojęciami nigdy wcześniej, zagadnieniom tym poświęcimy w całości rozdziały 4. i 5.

Te same reguły własności stanowią również podstawę niezawodnego modelu programowania współbieżnego w języku Rust. W większości języków nie istnieje związek pomiędzy obiektami służącymi do synchronizacji wątków (muteks) a danymi, do których dostęp jest za pomocą tych obiektów chroniony. Jest to realizowane wyłącznie na podstawie umowy i komentarzy w kodzie. Rust już na poziomie kompilacji sprawdza, czy muteks faktycznie chroni przed nieprawidłowym dostępem do danych. W przypadku większości języków zaleca się, żeby nie korzystać w programie głównym ze struktury danych, którą zaczęliśmy przetwarzać w osobnym wątku. Rust po prostu na to nie pozwoli. Rust już na poziomie kompilacji zapobiega sytuacjom, w których może dojść do wyścigu o dane.

Rust nie jest językiem w pełni obiektywnym, choć wiele cech języka obiektowego posiada. Nie jest też językiem funkcyjnym, choć wpływ języków funkcyjnych na pewne zastosowane w nim rozwiązania są widoczne, szczególnie w obszarze sposobu zwracania wyników. Rust bazuje na C i C++, ale ma tyle nowatorskich rozwiązań, że kod napisany w tym języku nie przypomina kodu w C czy C++. Wydaje się, że najlepiej wstrzymać się z porównaniami i ostateczną ocenę wystawić dopiero w momencie, gdy opanuje się ten język w stopniu pozwalającym na swobodne w nim programowanie.

Najlepszym sprawdzianem dla nowych pomysłów jest zastosowanie ich w świecie rzeczywistym. Dlatego też najnowszy silnik przeglądarki internetowej, o nazwie Servo, Mozilla zdecydowała się zaimplementować właśnie w języku Rust. Okazało się bowiem, że cele postawione przed Rustem i potrzeby Servo bardzo dobrze do siebie pasują. Przeglądarka musi działać szybko i niezawodnie oraz bezpiecznie obsługiwać niezaufane dane. Servo wykorzystuje bezpieczną współbieżność Rusta, w oddzielnych wątkach realizuje nawet to, co w C++ zrównoleglic byłoby naprawdę trudno. Tak naprawdę Rust i Servo dorastały razem. Servo korzystało z nowych rozwiązań dodawanych do języka, a Rust ewoluował w oparciu o opinie wyrażane przez programistów Servo.

Bezpieczeństwo typów

Rust jest językiem bezpiecznie typowanym. Co to znaczy? Samo pojęcie bezpieczeństwa brzmi dobrze, ale na czym ono ma w tym przypadku polegać?

Oto definicja *działania nieokreślonego* zaczerpnięta z normy języka C w specyfikacji 1999, znanej jako C99:

Działanie nieokreślone

Działanie w przypadku użycia nieprzenośnej lub błędnej konstrukcji programu lub błędnych danych, dla których niniejszy standard nie nakłada żadnych wymagań

Przeanalizujmy działanie następującego programu napisanego w języku C:

```
int main(int argc, char **argv) {
    unsigned long a[1];
    a[3] = 0x7ffff7b36cebUL;
    return 0;
}
```

Zgodnie ze specyfikacją C99, ponieważ program sięga do elementu tablicy, który znajduje się poza jej zakresem (indeks 3 elementu wykracza poza dopuszczalny zakres tablicy `a`), zachowanie programu jest nieokreślone. Oznacza to, że może wydarzyć się cokolwiek. Kiedy uruchomiliśmy program na laptopie Jima, efekt był następujący:

```
undef: Error: .netrc file is readable by others.
undef: Remove password or make file unreadable by others.
```

Program wyświetlił komunikat i zakończył się z błędem. Co ciekawe, na laptopie Jima nie było pliku o nazwie `.netrc`. Jeśli wykonasz podobny eksperyment u siebie, wydarzy się najprawdopodobniej zupełnie coś innego.

Kod maszynowy funkcji `main` wygenerowany przez kompilator C umieszcza na stosie tablicę `a`, po niej znajduje się adres powrotny funkcji. Tak więc wpisanie wartości `0x7ffff7b36cebUL` do elementu `a[3]`, który nie istnieje, bo jest już poza tablicą, zmienia wartość adresu powrotnego. Adres ten wskazuje teraz na kod ze środka biblioteki standardowej C. Trafiliśmy akurat na fragment, który przegląda plik `.netrc` w poszukiwaniu hasła. Zakładany powrót z funkcji `main` zamienia się na wywołanie kodu maszynowego odpowiadającego następującym wierszom biblioteki standardowej:

```
warnx_("Error: .netrc file is readable by others.");
warnx_("Remove password or make file unreadable by others.");
goto bad;
```

Dopuszczenie do sytuacji, w której odwołanie się do elementu tablicy ma wpływ na zmianę adresu powrotu funkcji, jest, jak widać, w pełni zgodne ze standardami języka C. Operacja o nieokreślonym działaniu po prostu niesie ze sobą nieokreślony wynik. Program może zrobić *dosłownie cokolwiek*.

Standard C99 daje kompilatorowi (i programiście) *carte blanche*. Celem jest umożliwienie dowolnej optymalizacji zmierzającej do uzyskania szybszego kodu. Zamiast kompilatora odpowiedzialnego za wykrywanie i obsługę niepożądanych zachowań (takich jak na przykład wychodzenie poza zakres tablicy), standard zakłada, że programista jest odpowiedzialny za zapewnienie, aby takie nieprawidłowe sytuacje nigdy nie miały miejsca.

Niestety doświadczenie pokazuje, że w pisaniu kodu o odpowiedniej jakości nie jesteśmy dobrzy. Będąc studentem na Uniwersytecie w Utah, Peng Li zmodyfikował używane tam kompilatory C i C++ w taki sposób, by raportowały wykryte w czasie kompilacji potencjalne źródła nieokreślonego zachowania kompilowanych przez siebie programów. Odkrył, że tego typu niedoskonałości występują bardzo często i że dotyczy to również projektów, które mają status bardzo ważnych i w związku z tym spełniają ponoć wysokie standardy kodowania. Działanie nieokreślone często prowadzi do wykorzystywania luk i wprowadzania niebezpiecznego kodu. Robak Morrisa rozprzestrzenił się z jednej maszyny na drugą właśnie przy użyciu rozwinięcia pokazanej przez nas techniki. Także dziś wiele wirusów działa właśnie w ten sposób.

Przy okazji tego przykładu zdefiniujemy pewne pojęcia. Jeśli program został napisany w ten sposób, że żadne jego użycie nie jest w stanie doprowadzić do sytuacji, którą moglibyśmy nazwać działaniem nieokreślonym, możemy powiedzieć, że program jest dobrze zdefiniowany. Jeśli mechanizmy kontroli samego języka programowania są w stanie zapewnić, że każdy program jest dobrze zdefiniowany, język taki uznać możemy za bezpieczny.

Starannie napisany program C lub C++ może być bezpieczny, ale sam język tego nie gwarantuje. C i C++ nie mają systemu typów, który zapewnia poprawność działającego programu. Przykład pokazany przez nas kompiluje się bez błędów, choć później działa nieprzewidywalnie. Bezpieczny pod tym względem jest natomiast Python. Program napisany w tym języku poświęca czas procesora na wykrywanie i obsługę indeksów, które wykraczają poza dopuszczalny zakres. Dzieje się to w sposób znacznie bardziej przyjazny niż w C:

```
>>> a = [0]
>>> a[3] = 0x7ffff7b36ceb
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

Program napisany w Pythonie w analogicznej sytuacji zgłosił wyjątek. A to nie jest już działanie nieokreślone. Specyfikacja języka Python jasno precyzuje, że przypisanie do `a[3]` zakończy się zwróceniem wyjątku typu `IndexError`. Oczywiście istnieją sposoby, by i za pomocą Pythona doprowadzić do nietypowych sytuacji, ale trzeba w tym celu skorzystać z dodatkowych modułów, takich jak na przykład *ctypes*. Sam język jest natomiast pod tym względem bezpieczny. Podobnie jak Java, JavaScript, Ruby czy Haskell.

Zwróć uwagę, że bezpieczeństwo na poziomie systemu typów nie zależy od tego, czy sprawdzenie dokonywane jest na poziomie kompilacji czy w czasie działania programu (czyli od tego, czy język jest typowany statycznie czy dynamicznie). C sprawdza typy podczas kompilacji i bezpieczny nie jest. Python robi to w czasie działania i bezpieczeństwo zapewnia.

Ironią jest, że dominujące języki programowania systemowego, czyli C i C++, nie zapewniają bezpieczeństwa na poziomie typów, podczas gdy większość języków aplikacyjnych cechą taką posiada. Biorąc pod uwagę, że C i C++ używane są do implementacji fundamentów systemu, którym powierzono odpowiedzialność za bezpieczeństwo systemu i kontakt z zewnętrznymi danymi, którym nie można ufać, omawiana przez nas cecha bezpieczeństwa na poziomie samego języka powinna być priorytetem.

Jest to bolączka, która trapi nas od kilku dekad i Rust próbuje być na to lekarstwem. Jest to wydajny język programowania systemowego zapewniający równocześnie bezpieczeństwo na poziomie systemu typów. Rust jest przeznaczony do implementacji podstawowych warstw systemowych, które wymagają najwyższej wydajności i drobiazgowej kontroli nad zasobami, ale nadal gwarantuje poziom przewidywalności zapewniany przez bezpieczeństwo typów. W niniejszej książce napiszemy, jak Rust radzi sobie z pogodzeniem tych sprzecznych na pierwszy rzut oka postulatów.

Ten szczególny rodzaj bezpieczeństwa, który oferuje nam Rust, ma zaskakujące konsekwencje dla programowania wielowątkowego. Programowanie współbieżne zawsze było trudne w C i C++. Dlatego programiści sięgali do tych technik niechętnie, zwykle wtedy, gdy kod jednowątkowy okazywał się niezdolny do osiągnięcia wymaganej wydajności. Rust gwarantuje już na poziomie języka, że współbieżny kod jest wolny od wyścigu do danych, wychwytyjąc nieprawidłowe użycie muteksów lub innych obiektów synchronizacji na etapie kompilacji. W języku Rust możesz używać współbieżności, nie martwiąc się o to, że popełnisz błąd lub że Twój kod będzie niezrozumiały dla innych.

A co w sytuacji, gdy nie mamy wyjścia i musimy bezwzględnie skorzystać z bezpośredniego, „surowego” wskaźnika, biorąc całą odpowiedzialność za działanie tego fragmentu programu na siebie? Istnieje taka furтка, możemy w Rustie pisać tzw. *niebezpieczny kod*, o czym więcej piszemy w rozdziale 21. Na szczęście większość programów nie będzie z tych mechanizmów korzystać.

Elastyczny system typów języka Rust nie służy tylko i wyłącznie zapewnieniu bezpieczeństwa i uchronieniu nas przed wystąpieniem niepożądanych sytuacji. Doświadczony programista używa odpowiednich typów w sposób pozwalający dobrze i zwięźle oddać jego intencje. Sprzyjają temu szczególnie zestawy metod (ang. *traits*) oraz typy generyczne, którym poświęcamy rozdział 11. To za ich pomocą możemy zwięźle i elastycznie wyrazić zestaw wspólnych cech grupy typów i później efektywnie te cechy wykorzystywać.

Celem naszej książki jest przekazanie Ci wiedzy i doświadczeń potrzebnych do pisania programów w języku Rust w sposób poprawny, to znaczy taki, aby były wydajne, bezpieczne oraz przejrzyste. Z naszego doświadczenia wynika, że Rust jest ważnym krokiem naprzód w programowaniu systemowym, i chcemy Cię za pomocą niniejszej książki do tego przekonać.

A

adapter, 330
 chain, 341
 cloned, 344
 cycle, 344
 enumerate, 341
 filter, 331
 filter_map, 333
 flat_map., 333
 fuse, 338
 inspect, 340
 map, 330
 peekable, 337
 rev, 340
 scan, 335
 skip, 336
 skip_while, 336
 take, 335
 take_while, 335
 zip, 342
algorytmy haszujące, 386
alias typu Result, 161
aliasy typów, 180
all, 348
alokacja warunkowa, 407
any, 348
architektura Flux, 321
ASCII, 80, 390, 516
AsMut, 296
AsRef, 296
atomowe operacje, 452
atomy, 184

B

bezpieczeństwo, 57, 313
 błędów paniki, 543
 referencji, 115

 typów, 21
 wątków, 472
biblioteka, 181
 libgit2, 552, 558
 Rayon, 459
 unicode-normalization, 425
BinaryHeap<T>, 361, 376
blokady
 odczytu/zapisu, 483
 typu muteks, 477
bloki
 extern, 181
 kodu, 138
 unsafe, 515
błąd panic, 157
błędy, 157
 ignorowanie, 167
 jednoczesna obsługa, 164
 paniki, 543
 propagacja, 163
 przechwytywanie, 160
 różnych typów, 164
 składniowe, 511
 typ Result, 160, 169
 w funkcji main, 167
 własny typ, 168
 wyświetlanie informacji, 162
Borrow, 297, 302
BorrowMut, 297
BTreeMap<K, V>, 361, 377
BTreeSet<T>, 361, 382
bufor odstępów, 537
by_ref, 342

C

Cargo.lock, 195
chain, 341
Clone, 289

cloned, 344
collect, 350
Condvar, 485
Copy, 290
count, 345
cycle, 344
cyfry, 393

D

dane binarne, 439
deadlock, 481
debugowanie, 415
 makr, 499
 wskaźników, 416
Default, 294
definiowanie
 metod, 206
 zależności, 193
 zestawów metod, 249
deklaracje, 140
 obcych funkcji, 547
 obcych zmiennych, 547
 składników lokalnych, 140
Deref, 291
dereferencja wskaźników
 niechronionych, 528
DerefMut, 291
doc-testy, 191
dokumentacja, 186, 189
dołączanie tekstu, 397
domknięcia, 153, 305
 bezpieczeństwo, 313
 Fn, 316
 FnMut, 316
 FnOnce, 316
 pożyczające wartość, 307
 przejmujące własność, 308
 typy, 309
 wydajność, 311

dopasowanie do wielu wartości, 233
drain, 328
Drop, 284
drzewa binarne, 236
dwukropek, 177
dynamiczne definiowanie
 długości, 417
 precyzji, 417
dziedziczenie, 254
 metod, 211

E

elementy, 149
Entry, 380
enumerate, 341
ExactSizeIterator, 348
Extend, 352

F

filter, 330
filter_map, 333
find, 350
flat_map, 333
Flux, 321
FnMut, 315
FnOnce, 314
fold, 349
formatowanie
 liczb, 412
 tekstu, 411
 wartości, 410, 415
 własnych typów, 417
 wskaźników, 416
From, 299
funkcja, 28, 148, 179
 copy_to, 446
 main(), 167
 spawn, 454
funkcje
 dostępu do systemu plików, 443
 generyczne, 60, 244
 niebezpieczne, 520
 obce, 544
 powiązane, 206
 unsafe, 518
 zwrotne, 317
fuse, 338

G

GapBuffer, 537
generator liczb pseudolosowych,
 261
generyczne
 funkcje, 244
 typy wyliczeniowe, 223
generyczny zestaw metod, 260
gruby wskaźnik, 77

H

HashMap<K, V>, 361, 377
HashSet<T>, 361, 382
haszowanie, 385

I

ignorowanie błędów, 167
implementacja
 interfejsu Drop, 284
 interfejsu IntoIterator, 326
 własnych iteratorów, 354
 zestawów metod, 249, 251
importy, 177, 181
indeks odwrócony, 464, 474
informacje
 o błędach, 162
 o łańcuchu znaków, 396
inspect, 340
instalacja, 25
interfejs, 239
 AsMut, 296
 AsRef<T>, 296
 Copy, 291
 Default, 294
 Drop, 284
 Extend, 352
 FnOnce, 314
 FromIterator, 351
 Index, 278
 IndexMut, 278
 IntoIterator, 326, 324
interfejsy
 bezpieczne, 558
 funkcji obcych, 544
 narzędziowe, 283
 niechronione, 552

Into, 299
IntoIterator, 324, 326
inżynieria wsteczna ograniczeń, 263
iter, 326
iter_mut, 326
iteracja, 372
 zbioru, 383
iterator, 323, 399
 ExactSizeIterator, 348
Iterator, 324
iteratory
 konsumowanie, 345
 obustronne, 339
 tworzenie, 326
 własne, 354
 źródła, 329
iterowanie
 map, 381
 tekstu, 401

J

język wyrażeń, 137
join, 454

K

kacze typowanie, 60
kanał synchroniczny, 471
kanały, 463
 cechy, 470
 muteksy, 482
 wydajność, 470
katalog src/bin, 183
katalogi, 440
kierunek tekstu, 392
klasyfikacja znaków, 392
klonowanie wartości, 290
kod niebezpieczny, 513, 520
kolejka dwustronna, 360, 373
kolekcja, 359
 BinaryHeap<T>, 361, 376
 BTreeMap<K, V>, 361, 377
 BTreeSet<T>, 361, 382
 HashMap<K, V>, 361, 377
 HashSet<T>, 361, 382
 LinkedList<T>, 360, 375
 Vec<T>, 360, 361
 VecDeque<T>, 360, 373

- kolekcje
 - niestandardowe, 387
 - tworzenie, 350
- kompresja, 439
- kontrakt, 514
- konwersja
 - tekstu, 404
 - wartości, 404
 - znaku, 394
- kopiec binarny, 361, 376
- krotki, 69, 230

L

- last, 350
- Latin-1, 390
- len, 348
- liczby zespolone, 48
- LinkedList<T>, 360, 375
- lista wiązana, 360, 375
- literały, 228
 - łańcuchowe, 79

Ł

- łańcuch
 - bajtów, 80
 - znaków, 80, 389, 396, 516
 - znaków jako kolekcje generyczne, 409
- łączenie wątków, 453

M

- makra, 491
 - a rekurencja, 504
 - debugowanie, 499
 - eksport, 509
 - higieniczne, 507
 - import, 509
 - niezamierzone skutki, 495
 - powtórzenia, 496
 - proceduralne, 512
 - rozwijanie, 493
 - typy składników, 503
 - wbudowane, 498
 - zakres, 507
 - zestawy metod, 505

- makro
 - format(), 82
 - format_args!, 419
 - json!, 500
- map, 330
- mapa
 - iterowanie, 381
 - typ Entry, 380
 - uporządkowana, 361, 377
 - z haszowaniem, 361, 377

- max, 345
- max_by, 346
- max_by_key, 346
- metoda, 148, 180
 - .join(), 456
 - all, 348
 - any, 348
 - by_ref, 343
 - clone, 289
 - collect, 350
 - concat(), 82
 - count, 345
 - default, 294
 - drain, 328
 - find, 350
 - fold, 349
 - iter, 326
 - iter_mut, 326
 - join(separator), 82
 - last, 350
 - len, 348
 - max, 345
 - max_by_key, 346
 - min, 345
 - min_by, 346
 - min_by_key, 346
 - nth, 349
 - partition, 353
 - position, 348
 - product, 345
 - rposition, 348
 - sum, 345
 - to_string(), 82
- metody, *Patrz także:* zestaw metod
 - kwalifikowana nazwa, 255
 - domyślne, 250
 - interfejsu PartialOrd, 277
 - statyczne, 208, 254
 - w bloku impl, 206

- min, 345
- min_by, 346
- min_by_key, 346
- moduły, 174, 181
 - składniki, 179
- morze obiektów, 134
 - zsynchronizowanych, 452
- mut, 480
- muteks, 476
 - zatruty, 482
- Mutex, 480
- Mutex<T>, 478

N

- typ [T, 72
- nazwa metody, 255
- niezdefiniowane zachowanie, 521
- normalizacja, 423
- nth, 349

O

- obcinanie tekstu, 403
- obiekt
 - buforowany typu reader, 431
 - implementujący zestaw metod, 242, 243
- obiekty
 - typu reader, 428, 429
 - typu writer, 428, 435
- obsługa
 - błędów, 157, 456
 - błędów w funkcji main(), 167
 - cyfr, 393
 - jednoczesna błędów, 164
 - sieci, 447
- obszary robocze, 198
- odczyt
 - wartości, 467
 - zawartości katalogu, 444
- odstęp, 537
- odwinięcie stosu, 158
- operacja przypisania, 112
- operacje
 - arytmetyczne na wskaźnikach, 532
 - wejścia/wyjścia, 427
- operator ::, 177

operatory, 281
 arytmetyczne, 150, 268
 bitowe, 150, 268
 dwuargumentowe, 271
 jednoargumentowe, 270
 logiczne, 150
 porównania, 150, 277
 priorytety, 153
 przeciążanie, 260, 267
 przypisania złożonego, 272
 referencji, 150
OsStr, 440

P

paczki, 171
 publikowanie, 196
pamięć, 534
parametr cyklu życia, 119, 210
parametry, 118
 cyklu życia, 125
parsowanie argumentów, 46
partition, 353
Path, 440, 442
PathBuf, 442
peekable, 337
pętla, 144
 for, 144
 loop, 144
 while, 144
 while let, 144
plik README.md, 199
pliki, 436, 440
 zapisywanie danych, 51
 z modułami, 175
pobieranie tekstu, 434
podzbiory, 77
podział wątków, 453
pola, 149
 nazywane, 201
 numerowane, 204
polimorfizm, 239
porównania szeregujące, 276
porównywanie sekwencji
 elementów, 347
position, 348
potoki, 468, 452
 adapterów iteratora, 474
 wątków, 474

priorytety operatorów, 153
product, 345
programowanie współbieżne, 41,
 451
propagacja błędów, 163
przechwytywanie
 błędów, 160
 zmiennych, 306
przeciążanie operatorów, 260, 267
przeglądanie tekstu, 432
przeniesienie własności, 91
 operacje, 96
 przepływ sterowania, 97
 struktury indeksowane, 98
przepływ sterowania, 97, 146
przerywanie procesu, 159
przymus dereferencji, 292
przypisanie, 151
publikowanie paczek, 196
pudełka, 72
pule wątków, 452

R

Rayon, 459
referencja &Self, 292
referencje, 71, 107, 150, 287
 bezpieczeństwo, 115
 C++, 111
 do podzbiorów, 115
 do referencji, 112
 do wyrażeń, 114
 do zestawów metod, 115
 do zmiennej lokalnej, 115
 innego typu, 405
 jako argumenty, 120
 jako wartości, 111, 121
 mutowalne, 109, 127
 odrębny cykl życia, 124
 operacja przypisania, 112
 parametry, 118
 porównywanie, 113
 puste, 114
 rusta, 111
 w strukturze, 122
 współdzielone, 109, 127, 133
RefWithFlag, 529
reguła własności, 85, 87

rekurencja, 504
relacja
 częściowej równoważności, 275
 równoważności, 275
relacje między typami, 257
rev, 340
rodzaje normalizacji, 424
rozmiary typów, 531
równoległość danych, 452
rposition, 348
Rust, 19
RwLock<T>, 483
rysowanie zbioru, 50
rzutowanie typów, 152

S

scan, 335
serializacja, 439
serwer web, 34
sieć, 447
sized type, 287
skip, 336
skip_while, 336
składniki modułów, 179
skrypt kompilacji, 551
słowo kluczowe Self, 252
spawn, 454
stałe, 180
stan współdzielony mutowalny, 476
str, 395
String, 395
struktura, 201, 230
 literału
 zmiennoprzecinkowego, 65
 zawierająca referencje, 122
struktury
 generyczne, 209
 indeksowane, 98
 puste, 204
 reprezentacja w pamięci, 205
 stosujące typy wyliczeniowe,
 222
 z parametrem cyklu życia, 210
 z polami nazywanymi, 201
 z polami numerowanymi, 204
sum, 345
surowy łańcuch znaków, 79

symbol
\$x, 497
?Sized, 288
<T>, 209
symbole wieloznaczne, 233

Ś

ścieżki, 177
średniki, 138

T

tablice, 72, 73
take, 335
take_while, 335
tekst, 389
 UTF-8, 406
testy, 29, 186
 integracyjne, 188
 równości, 273
ToOwned, 301
tworzenie
 iteratorów, 326
 kolekcji, 350
 łańcuchów znaków, 396
 referencji, 405
 tekstu, 406
typ, 179
 &[T], 72
 &mut [T], 72
 char, 392
 logiczny, 67
 łańcuchowy OsStr, 441
 o stałym rozmiarze, 287
 o zmiennym rozmiarze, 287
 Path, 442
 PathBuf, 442
 Result, 160, 169
 str, 395
 String, 79, 82, 83, 395
 Vec<T>, 72
 wyliczeniowy w pamięci, 221
 znakowy, 67
typy
 atomowe, 485
 całkowite, 62
 domknięć, 309
 funkcji, 309

generyczne, 239
iterowalne, 325
języka C, 545
kopiowalne, 100
maszynowe, 62
powiązane, 257
proste, 59
rozmiary, 531
rozmieszczanie, 531
wskaźnikowe, 71
wyliczeniowe, 218
 generyczne, 223
 zawierające dane, 220
zmiennoprzecinkowe, 65
znakowe, 83

U

Unicode, 80, 389, 390
unsafe, 515, 518
unsized type, 287
uruchamianie
 potoku, 468
 programu, 57
 testu, 29
usuwanie tekstu, 398
UTF-8, 80, 390
użycie
 funkcji, 549
 wyrażeń regularnych, 421

V

Vec<T>, 360, 361
VecDeque<T>, 360, 373

W

wątki
 bezpieczeństwo, 472
 działające w tle, 452
 łączenie, 453
 obsługa błędów, 456
 podział, 453
 współdzielenie
 niemutowalnych danych, 457
wektor, 360, 361
 dostęp do elementów, 362
 elementy losowe, 372

iteracja, 363, 372
łączenie, 367
podział, 367
porównywanie podzbiorów,
 371
sortowanie, 370
wyszukiwanie, 370
zamiana elementów, 369
zmniejszanie wielkości, 364
zwiększanie wielkości, 364
wektory, 74
wersje, 194
wiązanie argumentów, 416
wiersz poleceń, 30
własność współdzielona, 103
wskaźniki
 const w C++, 133
 dopuszczające wartość pustą,
 531
 niechronione, 72, 525
 operacje arytmetyczne, 532
 słabe, 105
współbieżność, 41, 451
wstawianie tekstu, 397
wydajność, 311, 470
wyłączny dostęp, 480
wyrażenia, 137
 regularne, 421
 regularne w trybie leniwym, 422
wyrażenie
 if, 141
 if let, 143
 match, 142
 return, 146
wysyłanie wartości, 464
wyszukiwanie, 437
 tekstu, 399
wywołania funkcji i metod, 148
wzorce, 226, 228, 399
 @, 235
 krotki, 230
 nieodrzucałne, 236
 odrzucałne, 236
 struktury, 230
 z referencjami, 231
wzorec Model-View-Controller,
 320

Z

- zakleszczenie, 481
- zależności, 193
- zapis do pliku, 51
- zaprzyjaźnione zestawy metod, 261
- zarządzanie pamięcią, 534
- zbiory
 - iteracja, 383
 - operacje, 383
 - równe wartości, 383
- zbiór
 - Mandelbrota, 42, 53, 461
 - uporządkowany, 382
 - z haszowaniem, 361, 382
- zestaw metod, 239
 - Borrow, 297
 - BorrowMut, 297
 - definiowanie, 249

- DerefMut, 291
- DoubleEndedIterator, 339
- Extend, 352
- From, 299
- generyczny, 260
- implementacja, 249, 251
- Into, 299
- Iterator, 324
- obiekt implementujący, 242
- przeciążanie operatorów, 268
- relacje między typami, 257
- rozszerzanie, 254
- Send, 472
- słowo kluczowe Self, 252
- std::marker::Sync, 472
- stosowanie, 241
- struktura obiektu
 - implementującego, 243

- ToOwned, 301
- unsafe, 523
- zaprzyjaźniony, 261
- zip, 342
- zmiana wielkości liter, 394, 403
- zmiennie, 228
 - globalne, 487
 - lokalne, 115
 - warunkowe, 485
 - zainicjalizowane, 536
- zmienność wewnętrzna, 212
- znacznik, 221
- znaki, 392

Ź

- źródła iteratorów, 329

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>

GRUPA
Helion 

Rust. Programowanie systemowe. Najlepiej zacząć od podstaw!

Programowanie systemowe zwykle nie interesuje twórców aplikacji. Niemniej warunkiem jej poprawnego działania jest właśnie kod systemowy. Programowanie systemowe zapewnia między innymi działanie systemu operacyjnego, sterowników, systemu plików, kodeków, a także zarządzanie pamięcią czy obsługę sieci. Jako że dotyczy wykorzystania zasobów, każdy szczegół, każdy bajt pamięci operacyjnej i każdy cykl procesora ma znaczenie. Rust – wyjątkowe narzędzie, cenione za szybkość, współbieżność i bezpieczeństwo – sprawi, że tworzenie kodu systemowego będzie łatwiejsze. Jednak tym, którzy dotychczas używali C#, Javy czy Pythona, język ten może się wydawać dość trudny do zrozumienia.

Ta książka jest znakomitym wprowadzeniem do języka Rust, pozwala też rozeznac się w zasadach programowania systemowego. Pokazuje, w jaki sposób zapewnić w kodzie bezpieczeństwo pamięci i wątków oraz sprawić, aby program był wykonywany szybko i bez błędów. Poszczególne zagadnienia zostały przedstawione jasno i przystępnie, a prezentowane koncepcje – zilustrowane licznymi przykładami kodu. Nie zabrakło również wskazówek ułatwiających bezproblemowe tworzenie wydajnego i bezpiecznego kodu. Książka jest przeznaczona przede wszystkim dla programistów systemowych, jednak przyda się także twórcom aplikacji, którym pozwoli zrozumieć zasady rządzące językiem Rust, a w efekcie stworzyć lepszy i łatwiejszy w utrzymaniu kod.

W książce między innymi:

- solidne wprowadzenie do języka Rust
- podstawowe typy danych, własności i referencje
- obsługa błędów w języku Rust
- obsługa wejścia-wyjścia, makra i współbieżność
- obsługa niebezpiecznego kodu

Jim Blandy – programuje od niemal czterdziestu lat. W 1990 roku zaangażował się w tworzenie wolnego oprogramowania. Zajmował się projektami GNU Emacs, GNU Guile, a także GDB (debugger GNU). Obecnie rozwija narzędzia deweloperskie dostępne w przeglądarce Firefox. Interesuje się biologią, astronomią i gotowaniem.

Jason Orendorff – napisał jeden z modułów silnika JavaScript przeglądarki Firefox. Jest aktywnym członkiem społeczności deweloperów w Nashville. Interesuje się gramatyką, pieczeniem i podrózami w czasie. Chętnie pomaga ludziom zrozumieć trudne zagadnienia.

	<p>Sprawdź nasze szkolenia!</p>  <p>AKADEMIA IT & BUSINESS</p> <p>WWW.SZKOLENIA.HELION.PL</p>	<p>KOD KORZYŚCI Sięgnij po więcej! ▶</p>  <p>ISBN 978-83-283-5740-2</p>  <p>9 788328 357402</p>
 <p>helion.pl</p>		
 <p>HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl</p>		
<p>INFORMATYKA W NAJLEPSZYM WYDANIU</p>		<p>Cena: 99,00 zł</p>