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

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.

Spis treści

ROZDZIAŁ 1.
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 równolegle… 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 dotykają 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 odśmiecania 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 obiektowym, 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++ zrównoleglić 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 rozprzestrzeniał 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 zdefiniujmy pewne pojęcia. Jeśli program został napisany w ten sposób, że żadne jego użycie nie jest w stanie doprowadzić do sytuacji, którą moglibyśmy nazwać działanie 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 cechę 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, wychwytując 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 furtka, możemy w Ruście 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.

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Ź

O'REILLY®

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 rozeznać 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 tworzyć 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 podróżami w czasie. Chętnie pomaga ludziom zrozumieć trudne zagadnienia.

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

ISBN 978-83-283-5740-2
9 788328 357402

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 99,00 zł