Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/20BE29Jakovljevic

KONCEPT VLASNIŠTVA PRI RUKOVANJU MEMORIJOM I NJEGOVA PRIMENA U RUST PROGRAMSKOM JEZIKU

OWNERSHIP APPROACH IN MEMORY MANAGEMENT AND ITS USE IN RUST PROGRAMMING LANGUAGE

Uroš Jakovljević, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *U ovom radu su opisani glavni principi i ideje koncepta vlasništva i kako su oni implementirani u programskom jeziku Rust, radi rešavanja problema sa memorijom. Analizirano je kako garancije koje pruža u pogledu bezbednosti memorije utiču na njegovu upotrebljivost, mogućnosti i performanse. Analiziran je i nebezbedni deo programskog jezika i koje su njegove mogućnosti, kao i kako ih programeri koriste.*

Ključne reči: *Rust, upravljanje memorijom*

Abstract – *This paper describes main principles and ideas of ownership approach in memory management and how they are implemented in Rust programming language, in order to solve memory problems. It has been analyzed how memory guarantees it provides affect its usability and capabilities. The unsafe part of programming language and its possibilities were also analyzed, as well as how programmers use them.*

Keywords: *Rust, memory management*

1. UVOD

Proces kontrolisanja i koordiniranja glavne memorije računara je jedan od glavnih izazova pri dizajnu programskog jezika. Ovaj proces je poznat pod nazivom upravljanje memorijom (*eng. memory management*). Upravljanje memorijom je jedna od stvari koje su najbitnije za pouzdan dizajn programskog jezika. Postoje dva široko rasprostranjena pristupa upravljanju memorijom, a to su ručni i automatski. Automatski je poznat i kao „sakupljač otpada“ (*eng. Garbage Collector*). Poslednjih godina, veliku popularnost je dostigao i treći pristup upravljanju memorijom, a on je zasnovan na principu vlasništva (*eng. ownership*) [1].

U ovom radu će biti opisani glavni principi i ideje koncepta vlasništva i kako su oni implementirani u programskom jeziku Rust, radi rešavanja problema sa memorijom kao što su na primer: curenje memorije (*eng. memory leak*), viseći pokazivači (*eng. dangling pointers*) kao i korišćenje memorije nakon njenog oslobađanja. Biće pomenuti neki najčešći problemi sa memorijom koji se mogu javiti i biće analizirano da li Rust programski jezik rešava ove probleme i ukoliko rešava, na koji način se to postiže. Nakon toga će biti reči o tome kako garancije koje Rust pruža u pogledu bezbednosti memorije utiču na njegovu upotrebljivost i mogućnosti. Govoriće se o nebezbednom delu Rust programskog jezika i šta njegova upotreba donosi kao prednosti kao i koji su rizici pri njegovom korišćenju. Na kraju će biti prikazano poređenje Rust-a sa drugim trenutno aktuelnim programskim jezicima, gde će biti reči o performansama samog jezika, to jest da li memorijske garancije koje on pruža utiču na same performanse tokom izvršavanja programa.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bila doc. dr Dunja Vrbaški.

2. OSNOVNE KARAKTERISTIKE RUST PROGRAMSKOG JEZIKA I UPRAVLJANJE MEMORIJOM PO PRINCIPU VLASNIŠTVA

Ozbiljni problemi sa memorijom su veoma česti u programskim jezicima sa ručnim upravljanjem memorijom. Sa druge strane, programski jezici sa automatskim upravljanjem memorijom rešavaju većinu ovih problema preko svojih sakupljača otpada, ali imaju značajan *overhead* i uticaj na performanse tokom izvršavanja programa. Koncept vlasništva u upravljanju memorijom je koncipiran tako da deterministički rešava probleme koji mogu nastati sa memorijom, bez toga da dolazi do uticaja na performanse tokom izvršavanja programa.

2.1. Koncept vlasništva u programskom jeziku Rust

Rust je programski jezik koji za upravljanje memorijom koristi koncept vlasništva. Kompajler u Rust programskom jeziku (*rustc*) eliminiše većinu problema sa memorijom bez korišćenja sakupljača otpada. Ovaj kompajler ima striktna pravila i provere u vezi sa upravljanjem memorijom po principu vlasništva [1].

Rust kompajler sprovodi, odnosno osigurava, sledeća pravila vlasništva [2]:

1. Svaka vrednost mora da poseduje svog vlasnika (*eng. owner*).
2. U datom trenutku može postojati samo jedan vlasnik.
3. Vrednost se odbacuje onda kada njen vlasnik izađe iz opsega (*eng. scope*).

Rust koristi statičke opsege (*eng. static scoping*) koji definišu opseg u programskom kodu u kojem su entiteti, kao što su to promenljive, validni. Promenljiva je validna od momenta u kome je deklarisana, pa do kraja opsega u kome je deklarisana.

Odnos opsega i validnosti promenljivih sličan je u Rust-u kao i u drugim programskim jezicima.

U svrhu demonstracije pravila vlasništva u Rust-u i načina na koji Rust zauzima i oslobađa memoriju, biće korišćen složeni tip *String*. Razlog za ovo je taj što je *String* tip, čija se tačna veličina ne zna u trenutku kompajliranja, se skladišti u hip memoriji (*eng. heap memory*). Za razliku od njega, prosti tipovi, fiksne veličine, koji se skladište na steku (*eng. stack*) mogu brzo i trivijalno da se kopiraju u nezavisne instance i uklanjaju sa steka kad izađu iz opsega [2].

Zahtevanje memorije se obavlja onda kada programer to uradi, dok se oslobađanje memorije obavlja automatski, onda kada promenljiva u čijem je vlasništvu ta memorija, izađe iz opsega. Kada promenljiva izađe iz opsega, Rust poziva specijalnu funkciju pod nazivom *drop* u koju autor složenog tipa, kao što je *String*, smešta kod koji je potreban da se oslobodi memorija za datu *String* promenljivu. Ova funkcija se poziva automatski na kraju opsega.

Postoje tri načina pomoću kojih promenljive i podaci vrše međusobnu interakciju [2]:

1. prenos vlasništva,
2. kloniranje,
3. kopiranje – samo za promenljive na steku.

Kada imamo dve promenljive fiksne veličine koje se skladište na steku i kada jednoj dodelimo vrednost druge, ta vrednost se kopira i svaka je nezavisno smeštena na stek. Sa druge strane, ako nešto slično uradimo sa promenljivom složenog tipa koja je smeštena u hip memoriji efekat je drugačiji. Promenljiva *String* tipa se sastoji iz tri dela: pokazivač na deo memorije u hip-u u kojem se nalazi sadržaj *String*-a, dužinu *String*-a i kapacitet. Ta tri podatka su fiksne veličine i oni se čuvaju na steku. Sam sadržaj *String*-a koji može biti promenljive veličine se nalazi na hip-u [2].

Kada dodelimo vrednost jedne promenljive drugoj kopiraju se podaci koji se nalaze na steku, odnosno pokazivač, dužina i kapacitet *String*-a. Podaci, odnosno sam sadržaj *String*-a koji se nalazi u hip memoriji se ne kopiraju. Nakon dodeljivanja vrednosti, menja se vlasnik te memorije i promenljiva iz koje se dodeljuje vrednost prestaje da bude validna, dok promenljiva kojoj se dodeljuje vrednost postaje validna i može se koristiti sve dok ne izađe iz opsega. Ovakav vid dodele vrednosti se naziva prenosom vlasništva.

Ukoliko postoji potreba da se uradi potpuno kopiranje promenljive složenog tipa to je moguće uraditi pomoću metode *clone* [2].

Mehanizam prosleđivanja vrednosti funkcijama je sličan kao i mehanizam dodele vrednosti promenljivama. Prosleđivanje promenljive funkciji će dovesti ili do prenosa vlasništva ili do kopiranja, kao što je to slučaj i sa dodelom. Što se tiče povratnih vrednosti funkcija, preko njih takođe može da dođe do prenosa vlasništva. Kada se prosleđuje neka promenljiva funkciji, prosleđuje se i njeno vlasništvo i samim tim gubi se mogućnost korišćenja te promenljive u funkciji iz koje je prosleđena. Ovo može da se reši pomoću povratne vrednosti funkcije, odnosno vraćanjem vlasništva nazad u pozivajuću funkciju. Međutim, to može biti poprilično nepraktično, pogotovo ukoliko postoji veći broj promenljivih i funkcija kojima se one prosleđuju. Da bi se ovaj problem rešio, odnosno da bi funkcijama mogle da se proslede vrednosti promenljivih, ali bez prosleđivanja vlasništva, u Rust programskom jeziku postoje reference (*eng. references*) i koncept pozajmljivanja (*eng. borrowing*) [2].

2.2. Reference i pozajmljivanje

Reference se označavaju pomoću posebnog karaktera ampersenda (&). Ono što je odlika reference jeste to da pomoću nje prosleđujemo vrednost, ali ne i vlasništvo nad promenljivom. To znači da kada referenca izađe iz opsega, ne oslobađa se memorija na koju ona pokazuje pozivom funkcije *drop*, već ostaje na onoj promenljivoj koja je vlasnik. Ovakvo prosleđivanje vrednosti se naziva pozajmljivanje jer vlasništvo ostaje isto, dok onaj koji pozajmljuje vrednost i dalje može tu vrednost da koristi. Ono što je karakteristika ovakvih referenci je to da one mogu samo da čitaju vrednost na koju pokazuju, ali ne i da je menjaju (*eng. immutable references*). Ovakvih referenci može biti veći broj u okviru istog dela koda, odnosno istog opsega [2].

Pored ovoga, referenci se može dozvoliti i da menja vrednost na koju pokazuje (*eng. mutable reference*). Ovakve reference se u kodu dobijaju pomoću ključne reči *mut*. One imaju veoma bitno ograničenje, a to je da ako postoji referenca koja može da menja vrednost na koju pokazuje, u istom opsegu ne sme da postoji još neka referenca na tu vrednost. To se odnosi i na reference koje samo čitaju vrednost, ukoliko postoji referenca koja je menja [2].

3. BEZBEDNOST MEMORIJE

The Microsoft Security Response Center procenjuje da 70% bagova koji imaju oznaku uobičajenih ranjivosti i izloženosti (eng. *CVE - common vulnerabilities and exposure*) su izazvani nedostacima u memoriji, odnosno upravljanju memorijom [3]. Ovo znači da samim osiguravanjem načina na koji se rukuje memorijom dosta uobičajenih ranjivosti se može izbeći. U ovom poglavlju će biti reči o nekim najčešćim ranjivostima vezanim za upravljanje memorijom i kako su one izbegnute u Rust programskom jeziku.

3.1. Najčešće ranjivosti vezane za upravljanje memorijom

Neke od ranjivosti koje se najčešće sreću, a proizilaze iz lošeg upravljanja memorijom, su [3]:

- Korišćenje memorije nakon njenog oslobađanja (*eng. use after free*).
- Dvostruko oslobađanje memorije (*eng. double free*).
- Dereferenciranje pokazivača koji ne pokazuje na validnu adresu (*eng. dereferencing a null pointer*).
- Curenje bafera (*eng. buffer overflow*) i čitanje van bafera (*eng. buffer overread*).
- Trke do podataka (*eng. data races*).

U Rust programskom jeziku, zbog same semantike vlasništva i pravila vlasništva koja su objašnjena u prethodnom poglavlju, problem korišćenja nakon oslobađanja se eliminiše već tokom kompajliranja

programa, odnosno sam kompajler će prijaviti grešku ukoliko se tako nešto pokuša. Kada se vrednost oslobodi, oslobađa se i vlasništvo i samim tim svaki pokušaj korišćenja te vrednosti ponovo nije validan.

Dvostruko oslobađanje memorije se takođe sprečava zbog same semantike vlasništva i činjenice da neka memorija može imati samo jednog vlasnika u datom trenutku, što osigurava da ne dođe dva puta do oslobađanja iste memorije. Memorija se oslobodi samo jednom i to onda kada vlasnik izađe iz opsega i ne koristi se više.

Dereferenciranje pokazivača koji ne pokazuje na validnu adresu se javlja pri pokušaju dereferenciranja *null* pokazivača koji najčešće pokazuje na adresu 0. Rust ovaj problem rešava tako što uopšte nema *null* vrednosti, već koristi *Option* tip za definisanje potencijalno neinicijalizovane vrednosti. Svaki *Option* je ili *Some* i sadrži neku vrednost, ili *None* i ne sadrži vrednost [4].

Curenje bafera je jedna od najpoznatijih ranjivosti vezanih za upravljanje memorijom. Na primer, programski jezik C i njegova standardna biblioteka imaju dosta ranjivosti koje su to omogućile. Sprečavanje curenja bafera se jednostavno radi proveravanjem granica samog bafera (*eng. bound checking*) tokom izvršavanja programa. Rust ovo radi automatski. Ista stvar važi i za čitanje izvan bafera.

Trke do podataka nastaju kada postoji više strana sa kojih se isti podaci pišu i čitaju. Rust programski jezik koristi princip vlasništva i pravila pozajmljivanja koji za posledicu imaju sprečavanje trka do podataka.

To se moglo videti iz prethodnog poglavlja, gde po tim pravilima, istovremeno može postojati više strana ili niti koje samo čitaju podatke, ili samo jedna strana koja može da piše podatke. To proizilazi iz pravila opisanih u poglavlju 2.2 gde na podatak može postojati samo jedna referenca koja piše u taj podatak. Ukoliko postoji takva referenca, drugih referenci na iste podatke ne može biti. Veći broj referenci je dozvoljen ukoliko one samo vrše čitanje podataka.

4. NEBEZBEDNI DEO RUST PROGRAMSKOG JEZIKA

Svi koncepti o kojima je do sada pisano u ovom radu zasnivaju se na pravilima i principima kojima sam Rust kompajler za vreme kompajliranja osigurava bezbednost memorije. Međutim, Rust programski jezik se može koristiti i na način na koji se ne primenjuju sva ova pravila. Ovakav Rust se naziva *unsafe Rust*, odnosno nebezbedni Rust. Nebezbedni Rust funkcioniše kao i klasičan Rust, s tim da ima dodatne mogućnosti koje su u određenim situacijama neophodne i same po sebi ne mogu da garantuju bezbednost memorije.

4.1. Funkcionalnosti nebezbednog Rust-a

Da bi se koristile funkcionalnosti nebezbednog Rust-a upotrebljava se ključna reč *unsafe* nakon koje se započinje novi blok koji sadrži nebezbedan kod. Velika prednost korišćenja ovakvog pristupa je i to što se samo unutar označenog bloka mogu koristiti funkcionalnosti nebezbednog Rust-a.

Postoji pet funkcionalnosti koje su dostupne u nebezbednom Rust-u, a nisu u običnom i to su [5]:

- Dereferenciranje „ogoljenog“ pokazivača (*eng. dereferencing a raw pointer*).
- Pozivanje nebezbedne funkcije ili metode.
- Pristup ili modifikovanje izmenljive statičke promenljive (*eng. mutable static variable*).
- Implementacija nebezbedne osobine (*eng. unsafe trait*).
- Pristupanje poljima unije (*eng. unions*).

4.1. Upotreba nebezbednog Rust-a

U ovom odeljku će biti reči o tome kako se Rust programski jezik koristi među programerima, odnosno koliki procenat projekata kao i samog koda sadrži *unsafe* blokove i šta se najčešće radi u tim blokovima. Ovo za cilj ima to da se vidi da li garancije koje pruža Rust kompajler ograničavaju programere u smislu da bez korišćenja nebezbednih delova ne mogu da obave zadatke koji su im potrebni.

U studiji iz 2020. godine, analizirano je 31867 paketa, koji se sastoje od sanduka, sa zvaničnog, centralnog Rust repozitorijuma, *crates.io* [6,7].

Na slici 1. može se videti koji procenat prethodno pomenutih sanduka eksplicitno sadrži nebezbedan kod kao i to kakvu vrstu nebezbednog koda sadrži.

Unsafe Feature	#crates	%
None	24,360	76.4
Some	7,507	23.6
Blocks	6,414	20.1
Function Declarations	4,287	13.5
Trait Implementations	1,591	5.0
Trait Declarations	280	0.9

Slika 1. *Prikaz procenta i broja Rust sanduka koji sadrže nebezbedan kod* [6]

Što se tiče veličine nebezbednog koda, za kvantifikovanje veličine koda u nebezbednim blokovima se koristila CFG reprezentacija, odnosno broj MIR instrukcija koje se generišu na osnovu napisanog koda. Veličina većine analiziranih nebezbednih blokova je poprilično mala i 75% njih su imali najviše 21 MIR instrukciju, što se skoro poklapa sa srednjom vrednošću broja MIR instrukcija za sve nebezbedne blokove – 22.0 [6].

Ono što se pokazalo u praksi je da su pozivi nebezbednih funkcija daleko najčešći razlog za korišćenje nebezbednog koda (89.76%), a nakon njih sledi dereferenciranje *raw* pokazivača (10.3%) i korišćenje statičke promenljive koja ima pravo da menja podatke koje sadrži (5.76%). U 99.4% svih slučajeva korišćenja nebezbednog koda, razlozi za njegovo korišćenje su jedna od ove 3 stvari ili njihove kombinacije [6].

5. POREĐENJE RUST PROGRAMSKOG JEZIKA SA DRUGIM PROGRAMSKIM JEZICIMA

Rust programski jezik je osmišljen sa idejom da pruži kontrolu i performanse jezika sa ručnim upravljanjem memorijom, ali da pored toga, pruži sigurnost i eliminiše greške koje dovode do nepravilnog rukovanja memorijom i ugrožavanja bezbednosti memorije. Ovo se postiže bez sakupljača otpada koji bi uticao na zauzeće resursa prilikom izvršavanja programa i samim tim postaje upotrebljiv tamo gde jezici sa automatskim rukovanjem memorijom nisu.

5.1. Poređenje performansi Rust programskog jezika sa drugim jezicima

Rađena je analiza vremena koje je potrebno da se alocira i dealocira promenljiva na hip memoriji: Rust, C, C++ i Java programskim jezicima. Da bi se utvrdilo vreme potrebno za veliki broj alokacija i dealokacija binarno stablo je korišćeno kao struktura podataka nad kojom se radi. Alokacija je merena pri dodavanju čvorova u binarno stablo, dok je dealokacija merena pri operaciji pronalaženja i zamene proizvoljnog čvora u stablu. Rezultati analize brzine alokacije memorije se mogu videti u tabeli 1, dok se brzine dealokacije vide u tabeli 2. Redovi koji ne sadrže broj, označavaju da izvršavanje nije moglo da se obavi do kraja zbog grešaka pri rukovanju memorijom [1].

Tabela 1. *Prosečno vreme izvršavanja programa za alokaciju memorije* [1]

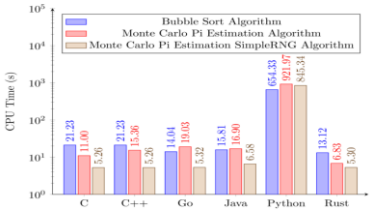
Programski jezik	10 miliona (s)	50 miliona (m)	100 miliona (m)
Rust	25.13	2.73	6.36
Java	8.59	1.36	/
C	19.93	16.84	/
C++	20.54	67.83	/

Tabela 2. *Prosečno vreme izvršavanja programa za dealokaciju memorije* [1]

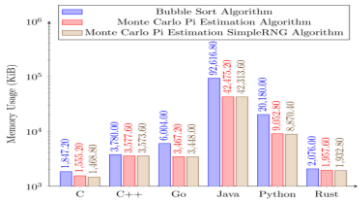
Programski jezik	10 miliona (s)	50 miliona (m)	100 miliona (m)
Rust	48.92	5.39	12.11
Java	14.79	3.17	/
C	36.97	50.12	/
C++	45.14	/	/

Iz ovih rezultata se može videti da koncepti vlasništva i pozajmljivanja koje koristi Rust programski jezik pri rukovanju memorijom pružaju stabilnost programima pisanim u ovom jeziku, odnosno dovode do mnogo manje grešaka sa memorijom tokom samog izvršavanja programa. Što se tiče samih performansi pri zauzimanju i oslobađanju memorije, za manji broj alokacija i dealokacija, Rust je lošiji od C programskog jezika, ali to nije drastična razlika, dok pri većem broju alokacija i dealokacija, Rust ima i bolje performanse i stabilnost u izvršavanju, odnosno, nije dolazilo do prekida rada programa zbog grešaka sa memorijom

Na slici 2 se može videti vreme potrebno za izvršavanje nekih popularnih algoritama u programima pisanim na različitim programskim jezicima, dok se na slici 3 vidi zauzeće memorije tih istih programa tokom izvršavanja. Algoritmi koji su analizirani su *bubble sort* algoritam za sortiranje i Monte Karlo algoritam za procenu vrednosti π [3]. Što se tiče same brzine izvršavanja programa, Rust, C i C++ su približno jednaki, s tim da Rust ima bolje performanse pri izvršavanju algoritma za sortiranje u odnosu na sve, a malo lošije performanse za Monte Karlo estimaciju u odnosu na C koji je bio najbrži. Što se tiče zauzeća memorije, Rust i C imaju ubedljivo najmanje zauzeće memorije od svih programskih jezika za ove algoritme, s tim da C ima blagu prednost u odnosu na Rust.



Slika 2. *Prosečno vreme izvršavanja algoritama* [3]



Slika 3. *Prosečno zauzeće memorije* [3]

6. ZAKLJUČAK

Može se videti da Rust donosi promene u dizajnu programskog jezika koje nisu tu samo da bi se reklo da je nešto drugačije, već one rešavaju stvarne probleme koji postoje u drugim programskim jezicima. Većina korisnika pri izboru programskog jezika pravi kompromis između toga da li da koristi brz programski jezik sa mogućnošću rada na nižem nivou i upravljanjem memorijom, ili programski jezik koji nema takve performanse i mogućnosti, ali pruža sigurnost u vidu bezbednosti memorije. Iz analiza urađenih u ovom radu se može zaključiti da Rust uspeva u tome da omogući slobodu korisniku pri rukovanju memorijom, uz pružanje veoma dobrih garancija da se to rukovanje neće obavljati na neispravan način i što je najbitnije, neće nepovoljno uticati na performanse izvršavanja samih programa.

7. LITERATURA

[1] E. Alhazmi, A. Aljubairy, F. Alhazmi, „Memory Management via Ownership Concept Rust and Swift: Experimental Study“, *IJCA*, vol. 183(22), pp. 1-10, April 2022.

[2] Rust documentation, Ownership, https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (pristupljeno 10.09.2022.)

[3] W. Bugden, A. Alahmar, „Rust: The Programming Language for Safety and Performance“, *IGSCONG'22*, Turska, Jun 2022.

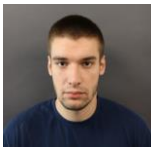
[4] Rust documentation, Option type, https://doc.rust-lang.org/std/option/ (pristupljeno 10.09.2022.)

[5] Rust documentation, Unsafe Rust, https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html (pristupljeno 10.09.2022.)

[6] V. Astrauskas, C. Matheja, F. Poli, P. Müller, A. Summers, „How do programmers use unsafe rust?“, *PACMPL*, vol. 4, pp. 1-27, Jun 2020.

[7] The Rust community's crate registry, https://crates.io/ (pristupljeno 10.09.2022.)

Kratka biografija:



Uroš Jakovljević rođen je u Subotici 1997. god. Master rad na Fakultetu tehničkih nauka iz oblasti Elektrotehnike i računarstva – Računarstvo i automatika odbranio je 2022. god.
kontakt: uros.jakovljevic97@gmail.com