

**Fakultet tehničkih nauka
Novi Sad**

Seminarski rad
Predmet: Programski prevodioci

Resource Acquisition is Initialization - RAI

Student:
Elena Akik, IN 6/2019

Februar, 2022

Sadržaj

1	Uvod	2
2	Osnovni koncept funkcionisanja RAI	3
2.1	U kojim programskim jezicima RAI može biti iskorišćen?	3
3	Način kreiranja RAI	4
3.1	Način kreiranja RAI konstruktora	4
3.1.1	Prethodna alokacija(eng. <i>Previous allocation</i>)	4
3.1.2	Alokacija u konstruktoru(eng. <i>Allocation in constructor</i>)	4
3.1.3	„Kasnija“ alokacija(eng. <i>Later allocation</i>)	4
3.2	Način kreiranja RAI destruktora	5
3.2.1	Prethodna alokacija(eng. <i>Previous allocation</i>)	5
3.2.2	Alokacija u konstruktoru(eng. <i>Allocation in constructor</i>)	5
3.2.3	„Kasnija“ alokacija(eng. <i>Later allocation</i>)	5
4	Prednosti RAI	6
5	Primeri korišćenja RAI u programskom kodu	7
6	Zaključak	8
7	Literatura	9

2 Osnovni koncept funkcionisanja RAI

U RAI, alokacija resursa izvršava se prilikom kreiranja objekta i obavlja je konstruktor, dok se dealokacija vrši tokom uništavanja objekta i obavlja je destruktor.

Suština je da se resurs drži tačno onoliko dugo koliko postoji i sam objekat za koji je isti vezan. Sve dok je objekat validan – resurs je upotrebljiv; kada se objekat uništi – resurs se oslobađa.

RAI je moćan koncept jer omogućava da se osigura da resursi „ne cure“ kada dođe do pojave izuzetaka (eng. exceptions), što predstavlja korektnu garanciju povezanu sa validnošću programskog koda.

2.1 U kojim programskim jezicima RAI može biti iskorišćen?

Koncept RAI može biti iskorišćen u:

- Jezici koji imaju korisnički definisane tipove alocirane na steku, koji će biti sklonjeni sa steka prilikom standardnog "čišćenja" steka (eng. stack cleaning up) - primer: C++
- Jezici koji funkcionišu po principu reference-counted garbage collection, to jest, svaki objekat broji koliko referenci poseduje. - primer: VB6

Sa druge strane, RAI nije pogodan za korišćenje kod jezika koji pri "čišćenju" objekata funkcionišu po principu nepredvidive garbage collection, poput Java. Ukoliko jezik garantuje "čišćenje" svih objekata pre no što program prestane sa radom, u tom slučaju RAI se može primeniti u određenim situacijama.

Ukoliko se pak koristi C++ sa garbage collection-om koje pritom nije definisano jezikom, već može biti obezbeđeno prilikom izvršavanja, tada se garbage collection neće odnositi na objekte alocirane na steku, te se RAI može koristiti u tom slučaju.

3 Način kreiranja RAI

3.1 Način kreiranja RAI konstruktora

Pri kreiranju konstruktora, važno je uzeti u obzir dinamičke konture odgovarajućeg resursa, jer od istih zavisi način kreiranja konstruktora: - Prethodna alokacija(eng. previous allocation) – „vlasništvo“ se prenosi u konstruktor

- Alokacija unutar konstruktora(eng. allocation inside constructor)

- „Kasnija“ alokacija(eng. Later allocation) – referenca se briše u konstruktoru

3.1.1 Prethodna alokacija(eng. *Previous allocation*)

U slučaju prethodne alokacije, resurs je prethodno već dodeljen ili postoji pre no što se pozove konstruktor. RAI objekat tada „uspostavlja“ pristup, ali ne alocira dati resurs. Vlasništvo nad resursom daje se RAI objektu koji u zavisnosti od implementacije, može biti odgovoran za uništavanje resursa ili odustajanje od vlasništva nakon uništenja.

Primer 1 - Fiksnim hardverskim uređajima poput pristupa UART-u, može se upravljati prosledjivanjem reference konstruktoru. Tada, resurs postaje vlasništvo i njime upravlja RAI objekat, te se sav pristup resursu vrši preko objekta, a destruktork se odriče vlasništva.

Primer 2 - Poruke iz dinamičkih bafera mogu biti prosledjene korisničkom programu od strane operativnog sistema, povratnog poziva ili drugog spoljnog uređaja. Vlasništvo nad porukom prenosi se u konstruktor RAI objekta, te se pristup poruci odvija preko datog objekta, a destruktork otpušta poruku. Dati pristup ima smisla koristiti kada se upravlja vlasništvom i pravom pristupa, ali ne i samostalnom alokacijom ili pribavljanjem resursa.

3.1.2 Alokacija u konstruktoru(eng. *Allocation in constructor*)

U slučaju alokacije u okviru konstruktora, resurs biva alociran unutar istog. ukoliko alokacija ne bude uspešna, doći će do pojave izuzetka(eng. exception). Ukoliko se resurs pak uspešno alocira, zagarantovano je da će ostati dostupan sve dok ga destruktork ne oslobodi.

Primer 1

Neka je alociran blok memorije, odnosno, određeni bafer. Kada je neophodno alocirati blok memorije s kojim će se naknadno raditi i čija alokacija ne bi smela biti neuspešna, trebalo bi izabrati alokaciju koja se vrši striktno u konstruktoru.

Primer 2

Pri otvaranju određene datoteke takođe je korektno odlučiti se za ovaj princip, odnosno otvoriti fajl u okviru konstruktora. Iako potencijalno postoje uslovi da se fajl korektno otvori, neophodno je to proveriti na osnovu rezultata koji vraća sistemski poziv open. Ukoliko pak fajl nije uspešno otvoren i ne postoji konstruktor u datom okruženju, neophodno je da se desi izuzetak. Kada je reč o zatvaranju pomenutog fajla, isto obavlja destruktork.

Primer 3

Tipični primer za ovaj metod je *lockguard* u okviru *mutex* – a. Konstruktor za lock guard, koji je RAI objekat, uzima već postojeće-instancirani mutex objekat i vrši poziv mutex-lock. Ovaj poziv može i ne mora izvršiti blokiranje, no, sigurno je da neće biti neuspešno izvršen. Destruktork *lockguard* poziva mutex unlock.

Ima smisla koristiti ovaj princip alokacije ukoliko je potrebno zauzeti resurs, uz uslov da alokacije ne sme da prođe neuspešno. Ukoliko proces alokacije prođe neuspešno, neophodno je da dođe do pojave izuzetka.

Takođe, dodatna pažnja treba biti usmerena an to da će destruktork biti pozvan samo za one objekte koji su prethodno "izgrađeni", ali ne i za objekte čiji je kreiranje dovelo do pojave određenog izuzetka.

3.1.3 „Kasnija“ alokacija(eng. *Later allocation*)

Ovaj princip alokacije podrazumeva da je referenca na resurs obrisana i ne postoji u okviru konstruktora, te se alokacija vrši kasnije, što je praksa za resurse koji su dinamički ili prolazni.

Primer 1

Ukoliko se otvara fajl na eksternom memorijskom uređaju, potrebno je obratiti pažnju na sledeće:

- Inicijalno otvaranje može proći neuspešno, pri čemu je svaki sledeći pokušaj takođe neuspešan.
- Ukoliko se fajl pak uspešno otvori, eksterni memorijski uređaj sa druge strane može u bilo kom trenutku vremena postati "nedostupan" za rad, pri čemu se gubi referenca na dati fajl. Ovaj problem rešava se brisanje reference u konstruktoru i "pomeranjem" svih poziva funkcija za otvaranje, zatvaranje i ponovno otvaranje i pristup fajlu u okviru same metode datog objekta. Pre bilo kog pristupa resursu, prvobitno se izvršava provera validnosti resursa. Na kraju, destruktork mora takođe proveriti referencu, pre no što izvrši zatvaranje.

Primer 2

Kada je reč o network konekcijama, takođe ne postoji garancija konstantnog uspešnog otvaranja i pristupa. Čak i da otvaranje uspešno prođe, u bilo kom trenutku, konekcija se može izgubiti iz velikog broja potencijalnih razloga. Ovaj problem rešava se tako što se obriše referenca iz konstruktora i sav kod koji se odnosi na otvaranje i kasnije održavanje koncekcije prenosi se u metode RAI objekta. Pre pristupa konekciji, prvo se vrši provera validnosti iste. Na kraju, destruktork mora proveriti referencu ka konekciji pre izvršenja operacije zatvaranja.

Ovaj pristup pogodan je dinamičke i prolazne resurse. RAI objekti izgrađeni na ovaj način su fleksibilni i potpuno odgovaraju dinamičkoj prirodi svojih resursa.

3.2 Način kreiranja RAI destruktora

Destruktork prati upravo onaj proces koji je prethodno izabran za kreiranje konstruktora. Destruktork je neophodan da bi se na korektan način upravljalo oslobađanjem ili uništavanjem prethodno zauzetog resursa.

Takođe, kao i kod kreiranja konstruktora, postoje 3 načina za kreiranje RAI destruktora.

3.2.1 Prethodna alokacija(eng.*Previous allocation*)

U ovom slučaju, resurs je bio prealociran i prosleđen u konstruktoru objekta, te je sad potrebno naći način na koji će se "vratiti" vlasništvo nad preuzetim resursom.

- Ukoliko je resurs bio prosleđen kao pokazivač ili parametar objekta, tada će zbog parametra, sam destruktork objekta biti automatski pozvan, te nije potrebna dodatna akcija.

U ovom slučaju, zagarantovano je da će referenca na resurs biti validna u destruktorku, jer je prethodno alocirana.

3.2.2 Alokacija u konstruktoru(eng.*Allocation in constructor*)

U slučaju alokacije unutar konstruktora, resurs biva dodeljen u konstruktoru i potrebno ga je osloboditi u destruktorku. Primeri nekoliko mogućnosti kod ovog principa:

- Ukoliko je konstruktor kreirao nit, destruktork uništava nit.
- Ukoliko je konstruktor otvorio datoteku, destruktork treba da zatvori datoteku.
- Ukoliko se konstruktor poveže sa drugim resursom, destruktork je taj koji treba da prekine vezu.

U ovom slučaju, referenca na resurs zagarantovano je važeća u destruktorku. Ukoliko pak konstruktor izazove pojavu izuzetka, destruktork neće biti pozvan.

3.2.3 „Kasnija“ alokacija(eng.*Later allocation*)

U ovom slučaju govorimo o resursima koji su dodeljeni van samog konstruktora. Takođe, obrađuje kako dinamičke, tako i "prolazne" resurse stečene tokom životnog veka objekta.

U slučaju "kasnije" alokacije, konstruktor je obrisao referencu i nije dodelio resurs. Kada se pokrene destruktork, referenca postaje važeća ako je resurs dodeljen objektu. Kasnije, na destruktorku je da da proveriti da li je referenca validna i da oslobodi resurs.

Primer 1

Pretpostavimo da u RAI objektu koju poseduje referencu na fajl, konstruktor obriše tu istu referencu. Ukoliko je fajl bio otvoren tokom rada sa programom i ukoliko je ostao otvoren, referenca će biti postavljena i destruktork će morati zatvoriti fajl.

4 Prednosti RAI

1. Obezbeđuje enkapsulaciju – logika upravljanja resursima definisana je na jednom mestu u klasi, a ne na svakom mestu poziva
2. Obezbeđuje bezbednost izuzetaka (za resurse steka) – resursi steka su resursi koji se oslobađaju u istom opsegu u kome su i zauzeti, tako što je vezan resurs za životni vek promenljive steka tj lokalne promenljive deklarisan u datom opsegu
 - *Ako se pojavi izuzetak i adekvatno rukovanje izuzecima je na tom mestu, jedini kod koji će biti izvršen pri izlasku iz trenutnog opsega su destruktori objekata koji su deklarisan u tom opsegu.*
3. Obezbeđuje lokalizaciju – omogućava da se logika akvizicije i oslobađanja zapiše jedna pored druge, omogućeno time što se definicija konstruktora i destruktora pišu jedna pored druge u definiciji klase.

Dodatne napomene:

- Upravljanje resursima treba da bude vezano za životni vek objekata kako bi se obezbedila automatska alokacija.
- Resursi bivaju pribavljeni prilikom inicijalizacije, kada ne postoji mogućnost da budu iskorišćeni pre nego što postanu dostupni, odnosno, u suprotnom slučaju, oslobađaju se uništavanjem objekata za koje su vezani, što je zagarantovan proces čak i u slučaju pojave greške.
- Upoređujući RAI sa finally blokom u Javi, Stroustrup je rekao da u "realnim sistemima postoji mnogo više zauzimanja resursa nego samih vrsta resursa, te da tehnika koju je nazvao "pribavljanje resursa je inicijalizacija" (eng. *RAI*) dovodi do znatno kraćeg koda nego upotreba "finally" konstrukta"¹.

¹Bjarne Stroustrup: „In realistic systems, there are far more resource acquisitions than kinds of resources, so the 'resource acquisition is initialization' technique leads to less code than use of a 'finally' construct“

5 Primeri korišćenja RAII u programskom kodu

```
#include <fstream>
#include <iostream>
#include <mutex>
#include <stdexcept>
#include <string>

void WriteToFile(const std::string& message) {
    // [mutex] is to protect access to [file] (which is shared across threads).
    static std::mutex mutex;

    // Lock [mutex] before accessing [file].
    std::lock_guard<std::mutex> lock(mutex);

    // Try to open file.
    std::ofstream file("example.txt");
    if (!file.is_open()) {
        throw std::runtime_error("unable to open file");
    }

    // Write [message] to [file].
    file << message << std::endl;

    // [file] will be closed first when leaving scope (regardless of exception)
    // [mutex] will be unlocked second (from Lock destructor) when leaving scope
    // (regardless of exception).
}
```

Slika 2: Usage of RAII, C++11 example

```
std::mutex m;

void bad()
{
    m.lock();           // acquire the mutex
    f();                // if f() throws an exception, the mutex is never released
    if(!everything_ok()) return; // early return, the mutex is never released
    m.unlock();         // if bad() reaches this statement, the mutex is released
}

void good()
{
    std::lock_guard<std::mutex> lk(m); // RAII class: mutex acquisition is initialization
    f();                             // if f() throws an exception, the mutex is released
    if(!everything_ok()) return;      // early return, the mutex is released
}                                     // if good() returns normally, the mutex is released
```

Slika 3: Classes with open()/close(), lock()/unlock(), or init()/copyFrom()/destroy() member functions - typical examples of non-RAII classes

6 Zaključak

U okviru ovog rada, obrađena je tematika bazirana konceptu RAI, koji se najčešće veže za programski jezik C++.

Korišćenje RAI, idioma programskog jezika C++, u velikoj meri pojednostavljuje upravljanje resursima, smanjuje ukupnu količinu koda i dodatno pomaže da se obezbedi ispravnost programa.

Njegova moć upravo leži u garancijama koje pruža: ukoliko se koristi na pravilan način, destruktor za RAI objekat zagantovano će biti pozvan i svi prethodno zauzeti resursi biće oslobođeni, uz dodatnu garanciju da resursi neće nestati prilikom pojave nekog od izuzetaka u programu.

7 Literatura

- (1) Bjarne Stroustrup, The C++ Programming Language (3rd Edition), Addison-Wesley Pub Co, 2000
- (2) LINK: Hackcraft - section: RAII
- (3) LINK: Stroustrup.com - finally concept in Java versus RAII in C++
- (4) LINK: Github repository - section: "Using RAII to prevent leaks"
- (5) LINK: CppReference - section about RAII and its use in STL, mutex