Razvoj softvera

Sa primerima u programskom jeziku C++

Nikola Ajzenhamer

15. oktobar 2023.

Sadržaj

Predgovor

Ideja za formiranje ovog teksta se javila nakon dve godine držanja vežbi iz kursa „Razvoj softvera" na završnoj godini osnovnih akademskih studija smerova „informatika" i „računarsko i informatika" na Matematičkom fakultetu, Univerziteta u Beogradu. Motivacija za pisanje teksta je nastala nakon sagledavanja nekoliko desetina softvera koje su studenti razvijali na kursu, kao deo svojih predispitnih obaveza. Ovaj proces, koji traje približno dva meseca, istovremeno je i najzahtevniji i najbolji način da se studenti susretnu sa pitanjima i problemima koji će im se pojavljivati svakodnevno pri profesionalnom radu nakon svojih studija. Timski rad, pod nadzorom profesora i asistenata, za ideju ima da simulira okruženje u kojem će oni raditi u svojoj budućnosti, a insistiranje na samostalnom suočavanju sa problemima tokom razvoja i traženjem rešenja produbljivanjem znanja iz programskog jezika u kojem se softver razvija, zatim istraživanjem postojećih biblioteka, čitanjem dokumentacije i pretraživanjem rešenja na vebu (uz usmeravanje studenata od strane profesora i asistenata), produbljava se kritičko razmišljanje i usvajanje sekundarnih veština koje kompanije danas cene u juniorskim programerima.

Dosadašnji materijali za vežbe za ovaj kurs su obuhvatali programe koji su za cilj imali da uvedu studente u programski jezik C++ (i, eventualno, u neke biblioteke) i da kroz taj jezik ilustruju neke od koncepata koji se javljaju u razvoju softvera. Međutim, autorov utisak je da većina studenata kodove smatra samo kao sredstvo za učenje jezika C++ i definisanje granica gradiva koje se polaže na praktičnom delu završnog ispita. Dopunjavanjem postojećih materijala u vidu kodova aplikacija, tako da ih prati i tekst koji detaljnije obrazlaže odluke do kojih se došlo prilikom razvoja tih aplikacija, omogućuje se da se date aplikacije iskoriste i kao inspiracija za smišljanje načina za implementaciju zahteva koje studenti definišu pri razvoju timskog projekta. Na primer, studentski tim bi mogao brže da započne razvoj svoje aplikacije ako analizom zahteva utvrdi da će proces razvoja biti pravolinijski i da su zahtevi unapred poznati i

nepromenljivi, pa se odluči da koristi metod vodopada; ili ako utvrdi da je za potrebe softvera neophodno da grafički prikaže elemente domena i da grafički njima upravlja, pa će se fokusirati na produbljivanju znanja iz radnog okvira grafičke scene; ili će utvrditi da je neophodan pažljiv inkrementalni razvoj u kojem bi svako napredovanje u razvoju moglo da izazove greške u postojećoj implementaciji, pa će se odlučiti za razvoj vođen testovima; itd.

Dodatna motivacija za pisanje ovog teksta jeste pokušaj za približavanjem teorijskih koncepata iz razvoja softvera studentima. U gotovo svim obrazovnim sistemima koji obuhvataju programerske discipline, pa i u našem, akcenat je stavljen na praktične primene kako bi studenti što pre savladali neophodne veštine programiranja, sa potencijalno nepravedno postavljenim paralelnim zahtevima za visokim kvalitetom i brzinom rada, a koje samo tržište programiranja diktira. Iako je čin programiranja veština koja je, sasvim razumno, neophodna svakome ko želi da se bavi programiranjem, ipak ne smemo zaboraviti da je teorijska podloga ta koja omogućuje da programiramo softver na dobar način. Na primer, bez detaljne teorijske analize modela konkurentnog izvršavanja i traženjem rešenja problema sinhronizacije, mi ne bismo danas znali kako i kada da paralelizujemo izračunavanja naših programa. Još jedan primer je da razmišljanje o dobrim i lošim stranama različitih obrazaca za arhitekturu softvera nas postavlja na ispravan put kada započinjemo razvoj softvera. Za razliku od toga, ako se ne bavimo arhitekturom softvera, vrlo verovatno je da naš projekat neće biti uspešan ili će u krajnjoj liniji biti izuzetno težak za održavanje. Dakle, predupređivanje praktičnih problema je moguće jedino ako smo upoznati sa teorijskim aspektima okvira u kojima se ti problemi mogu ispoljiti. U nadi da će se budući studenti koji budu slušali ovaj kurs ozbiljnije posvetiti časovima predavanjima i teorijskim konceptima koji se budu obrađivali na tim časovima, svaki zadatak koji je implementiran u ovom tekstu ima za cilj da prikaže neki aspekt teorije razvoja softvera.

Od čitaoca ovog teksta se očekuje da poseduje duboko razumevanje programskog jezika C. Velikom broju koncepata iz programskog jezika C, kao što su dinamički alocirana memorija i korisnički definisani tipovi, biće posvećena pažnja i u ovom tekstu, ali iz ugla programskog jezika C++. Iako za veliki deo programskog jezika C++ postoji odgovarajući uvod u ovom tekstu, on se ipak ne može koristiti kao osnovna literatura za sam jezik. Zbog toga, od čitaoca se očekuje samoinicijativa u istraživanju klasa, funkcija i drugih koncepata koji se pominju u tekstu, kao i u pronalaženju odgovora na pitanja koja će čitalac sam sebi postaviti tokom čitanja teksta. Dodatno, u određenim zadacima se koriste biblioteke koje nisu deo standardne biblioteke jezika C++ u cilju ilustrovanja raznolikosti ekosistema jezika C++ (predma je ilustrovan tek njegov mali deo).

Konkretno, neki zadaci će podrazumevati korišćenje biblioteka Qt6 i Catch2, i u takvim zadacima će upotreba biblioteka biti eksplicitno označena. Važno je napomenuti da se programski jezik C++, kao i pomenute biblioteke, koriste isključivo kao alati kojima se konkretizuje diskusija o aspektima razvoja softvera kroz razvoj konkretnih aplikacija. To znači da čitalac može da sve zadatke implementira u bilo kom drugom popularnijem programskom jeziku (koji podržava odgovarajuću paradigmu) ili korišćenjem alternativnih biblioteka. Dodatno, različiti koncepti koje studenti osnovnih studija na nižim godinama usvajaju biće im od velike koristi za uspešno savladavanje ovog gradiva. Različite programske paradigme, algoritmi, strukture podataka i elementarni koncepti teorije operativnih sistema bi trebalo da budu makar konceptualno poznati čitaocu, a kroz tekst će se ukratko podsetiti na neke od njih, tamo gde je autor smatrao da je takvo podsećanje neophodno.

O stilu pisanja

Ceo tekst je pisan u LaTeXjeziku za obeležavanje. Tekst je većinski napisan u podrazumevanoj familiji fonta. Delovi teksta koji predstavljaju elemente programskog jezika poput identifikatora, ključnih reči i dr. biće napisani familijom fontova jednake širine, na primer, `class`, `int a`, `std::accumulate` i sl. Duži fragmenti koda biće izdvojeni u okruženje u istom stilu kao i linijski elementi programskog jezika, na primer:

```
#include <iostream>

int main() {
    std::cout << "Zdravo, svete!" << std::endl;
    std::cerr << "Zdravo, svete!" << std::endl;

    return 0;
}
```

Svi kodovi su pisani na engleskom jeziku, na insistiranje autora. Razlog za ovo jeste veliki broj stranih kompanija u ekosistemu tržišta programiranja, koji uveliko nadmašuje broj „domaćih” kompanija. Ipak, svi tekstovi pod komentarima i niskama su napisani na srpskom jeziku: oni pod komentarima služe čitaocima da bolje razumeju kod, a oni pod niskama da ipak skrenu pažnju da je tekst koji prati kod napisan na srpskom jeziku.

Tekstovi zadataka koji ilustruju koncepte kroz knjigu imaju svoje okruženje. Svaki zadatak prati jedinstveni broj za brzu identifikaciju, kao i naziv. Nazivi su dati na engleskom jeziku, kako bi pratili jezik u kojem je pisan kod. U nastavku dajemo primer zadatka.

Zadatak 0: Hello world

Napisati aplikaciju koja ispisuje poruku „Zdravo, svete!" na standardni izlaz i standardni izlaz za greške.

Kada je neophodno prikazati izlaz iz programa, biće korišćena okruženja za standardni izlaz i standardni izlaz za greške, koji su dati u nastavku, redom.

Standardni izlaz

```
Zdravo, svete!
```

Standardni izlaz za greške

```
Zdravo, svete!
```

S obzirom da je ovaj tekst namenjen pre svega za upotrebu uz časove vežbe, u tekstu nećemo prikazivati proces pripreme okruženja za programiranje, kao ni procese prevođenja aplikacija i pokretanja. Ipak, skrenućemo pažnju da je za odgovarajuće operativne sisteme neophodno instalirati najažurniju verziju C++ kompilatora koji podržava makar standard iz 2017. godine. Dodatno, neki zadaci zahtevaju Qt6 biblioteku ili Catch2 biblioteku, pa se čitaocu ostavlja da istraži najpogodnije metode za instaliranje tih biblioteka.

Glava 1

Metodologije razvoja softvera

Ukoliko ste napisali makar jedan program do sada, onda ste se sigurno zapitali odakle da počnete kada neko od Vas zahteva da isprogramirate neku aplikaciju. Kada otvorimo editor ili neko razvojno okruženje, suočavamo se sa praznim dokumentom u kojem treba da smestimo kod koji će implementirati to što korisnici, ili, u tokom studija, asistenti i profesori, od Vas zahtevaju. Prazan ekran može biti podjednako stresan kao i prazan papir sa pisca koji započinje novi roman. Da li pisci započinju pisanje romana tako što otvore neki procesor teksta i kucaju reči? Neki pisci zaista tako rade. Međutim, postoji značajan broj pisaca koji nije u stanju da napiše nijednu rečenicu sve dok ne pripremi „teren" za pisanje, na primer, osmišljavajući likove i konflikte među njima, razvojem radnje ili postavljanjem tih likova i radnji u konkretan fiktivni (ili stvarni) svet. Razvoj softvera podseća na pisanje romana možda i više nego što bismo to očekivali. Iako je moguće otvoriti editor i kodirati bez prethodne pripreme „terena", kao i u slučaju pisanja romana, rezultat koji ćemo dobiti će verovatno imati veliki broj grešaka, postojaće čudno implementirani delovi koda ili ćemo dobiti kod koji je teško održiv. Kao što neki pisci prvo pišu nacrt svog romana kako bi njihova priča bila konciznija i bez velikih problema, tako i neki programeri prvo analiziraju problem zadatka koji je postavljen pred njih kako bi kod koji budu programirali bio boljeg kvaliteta.

Tokom razvoja informacionih tehnologija i sam proces razvoja softvera je napredovao, što je prirodno dovelo do definisanja procesa prema kojima se softver

razvija, prateći određene principe. Ovi procesi opisuju ono što nazivamo *životni ciklus razvoja softvera*, odnosno, opisuju korake koje svi učesnici u razvoja softvera treba da prate kako bi se softver uspešno razvio. Ovi koraci se nazivaju *faze razvoja softvera*. Svaka metodologija detaljno opisuje broj faza razvoja softvera, detaljan opis svake od tih faza i redosled kojim se te faze smenjuju. Iako bismo mogli da sagledamo svaku metodologiju sa njenim fazama nezavisno od ostalih metodologija, ipak se uočavaju neke faze koje su zajedničke za sve metodologije i mi ćemo započeti ovo poglavlje izlaganjem ovih faza razvoja.

1.1 Faze razvoja softvera

Kao što smo rekli, veliki broj metodologija razvoja softvera definiše sličan skup faza razvoja. One faze koje možemo primetiti u (gotovo) svim metodologijama razvoja softvera su: *planiranje*, *analiza*, *projektovanje* i *implementacija*. Svake od ovih faza stavljaju fokus na određene *postupke* koje učesnici u razvoja softvera čine. Takođe, u opisu određene faze učestvuju i *tehnike* koje proizvode *artefakte*, odnosno, rezultate postupaka sprovedene tim tehnikama. Prodiskutujmo sada o svakoj fazi razvoja softvera nešto detaljnije:

Planiranje Faza planiranja predstavlja početnu tačku svih metodologija razvoja softvera. Faza planiranja obuhvata dve važne aktivnosti:

1. U fazi planiranja, naručioci softvera identifikuju potrebu za kreiranjem novog ili unapređenjem postojećeg softvera. Na primer, neka je marketinška organizacija $\mathcal{A}$ primetila da će u skorijoj budućnosti dobiti veliki broj novih klijenata. Kako bi uspela da zadovolji potrebe za povećanjem posla, ona mora da zaposli veliki broj novih zaposlenih. U te svrhe, želi da promeni softver za upravljanje informacijama o trenutnim zaposlenim, ali i o kandidatima za buduća zaposlenja u toj organizaciji. Prepoznavanje potreba za izradu softvera rezultuje *zahtevom* za razvoj tog softvera. Organizacija $\mathcal{B}$ koja se bavi razvojem softvera sklapa ugovor o razvoju odgovarajućeg softverskog sistema sa organizacijom $\mathcal{A}$. Na strani organizacije $\mathcal{A}$ se formira strana „kupca", a na strani organizacije $\mathcal{B}$ se formira strana „razvijaoca" softvera. U ovom trenutku, softver koji treba da se razvije započinje svoj životni ciklus.
2. Faza planiranja obuhvata i proces *upravljanja* razvojem softvera. Preciznije, prepoznaju se poslovi koje naručioci softvera moraju da uspu-

ne, odlučuje se o inicijalnom budžetu razvoja, detaljnije se razmatraju ostale faze razvoja, formiraju se timovi koji učestvuju u razvoju, postavljaju se početne procene vezane za utrošene resurse, razmatraju se rizici, itd. Upravljanje proizvodi plan na osnovu kojeg će softver biti razvijen.

Analiza U fazi analize se detaljnije pristupa posmatranju okruženja, odnosno, konteksta u okviru kojeg će se budući softver koristiti. Faza analiza obuhvata naredne aktivnosti:

1. Definišu se korisnici softvera, sagledavaju se poslovni procesi koji već postoje u kontekstu, definišu se eventualni novi procesi koje budući softver mora da implementira, razmatraju se razne vrste ograničenja koje softver mora da zadovolji, itd.
2. Na osnovu prethodnih definisanih elemenata, konstruišu se konceptualni okviri postojećeg sistema, kao i novog sistema koji se razvija. Kako bi se ovi konceptualni okviri definisali, razvijaoci, zajedno sa kupcima, međusobno razmenjuju informacije od značaja. Zajedničkim snagama se formiraju *zahtevi* sistema. Zahtevi mogu biti *funkcionalni* ili *nefunkcionalni*. Funkcionalni zahtevi obuhvataju sve ono što softvera treba da implementira, a nefunkcionalni zahtevi su sve ono *kako* softver treba da funkcioniše. Na primer, funkcionalni zahtevi softvera za organizaciju $\mathcal{A}$ su: upis novog kandidata u sistem, organizovanje testova, prikupljanje i obrada rezultata testova, prosleđivanje obaveštenja o ishodu zaposlenja, upravljanje ugovorima, upravljanje isplatama, sistemi za nagrađivanje, itd. Nefunkcionalni zahtevi su: sistem mora da bude dostupan kao desktop i mobilna aplikacija, sistemu se može pristupiti na daljinu u bilo kom trenutku preko interneta, sistem mora da obezbedi adekvatnu bezbednost informacija, itd.
3. Rezultat faze analize je *predlog softvera*. Razvijaoci softvera predstavljaju konstruisani plan kupcima. Kupci zatim odlučuju da li je dati predlog korektan i da li softver može da započne izradu ili treba da se vrši dorada predloga.

Projektovanje Faza projektovanja opisuje detalje funkcionisanja softvera. Razmatraju se razni elementi informacionih sistema: od odabira hardvera i mrežne arhitekture do softverskih elemenata kao što su: operativni sistemi na kojima će softver raditi, programski jezici u kojima će se softver razvijati, sistemi za upravljanje bazama podataka sa kojima će softver

sarađivati, datoteke koje će softver koristiti, korisnički interfejsi koji će biti dizajnirani, itd. Faza projektovanja ima za cilj da detaljnije opiše način na koji će se softver razvijati, ali i način na koji će se koristiti. Faza projektovanja podrazumeva naredne aktivnosti:

1. Definisanje *arhitekture* softvera, odnosno, opisa svih potrebnih elemenata informacionih sistema, kao i načina na koji oni međusobno sarađuju. Različitim softverskim sistemima odgovaraju različite arhitekture, te je posao softverskih projektanata da prepoznaju arhitekturu koja će „najbolje" odgovarati softveru koji se razvija. Pod ovim mislimo na zadovoljavanje zahteva koji se međusobno takmiče, kao što su: brzina isporučivanja softvera, kvalitet koda, budžet i dr.
2. Pored arhitekture softvera, u ovoj fazi se prepoznaju svi podaci i objekti koji su od interesa i definišu se njihove karakteristike. Kontekst ovde igra važnu ulogu. Na primer, posmatrajmo objekat `Čovek` i neke njegove osobine: ime, prezime, JMBG, prethodna zaposlenja, obrazovanje, boja očiju, visina i težina. U kontekstu softvera za upravljanje zaposlenih organizacije $\mathcal{A}$, boja očiju, visina i težina su karakteristike koje nemaju značaja za poslovne procese vezane za zaposlenje kandidata, dok su obrazovanje i prethodna zaposlenja vrlo korisne informacije. *Specifikacija podataka* koja se dobija kao rezultat ovih aktivnosti uključuje detaljan opis informacija kojima softver upravlja, zajedno sa tehnikama njihovog upravljanja, kao što je specifikacija baza podataka, datoteka i sl.
3. Pored podataka, od važnosti su i *procesi* koji treba da budu implementirani. Projektanti definišu ulaze i izlaze ovih procesa, kao i tehnike njihove međusobne saradnje u cilju obezbeđivanja funkcionalnih zahteva.

Arhitektura, specifikacija podataka i procesi ulaze u sastav *specifikacije sistema* na osnovu kojeg se vrši implementacija.

Implementacija U ovoj fazi se softver konstruiše. Međutim, u ovoj fazi postoje još neke aktivnosti kojih se čitaoci možda ne bi dosetili na prvi pogled. Ova faza podrazumeva nekoliko aktivnosti:

1. *Izgradnja* softvera je osnovna očigledna aktivnost ove faze. Na osnovu specifikacije sistema, programeri pristupaju izradi softvera korišćenjem odgovarajućih tehnika i alata.

2. *Testiranje* softvera predstavlja još jednu očiglednu aktivnost. Što je softver složeniji i obimniji, verovatnoća da se defekti pojave u radu softvera je utoliko veća.
3. *Isporučivanje* softvera podrazumeva pakovanje rezultata izgradnje koda, instaliranje tih paketa, podešavanje SUBP, mrežne arhitekture, itd.
4. *Obučavanje* korisnika je od velikog značaja. Ukoliko korisnici ne umeju da koriste vaš proizvod, onda je on u potpunosti neupotrebljiv, bez obzira na to koliko je dobro implementiran.
5. *Održavanje* softvera podrazumeva pružanje podrške za razne vrste problema koji se mogu javiti tokom korišćenja softvera. Najčešće, održavanje softvera podrazumeva ispravljanje defekata u kodu usled nedosledne implementacije funkcionalnih zahteva, ali može obuhvatati i doimplementiranje novih zahteva, ponovno isporučivanje (na primer, usled promene lokacije organizacije kupaca ili zamene opreme), dodatno obučavanje (na primer, u slučaju novih zaposlenih), itd.

Sada kada smo upoznali faze u životnom veku razvoja softvera, pređimo na opisivanje nekih metodologija razvoja softvera.

1.2 Metodologije razvoja softvera

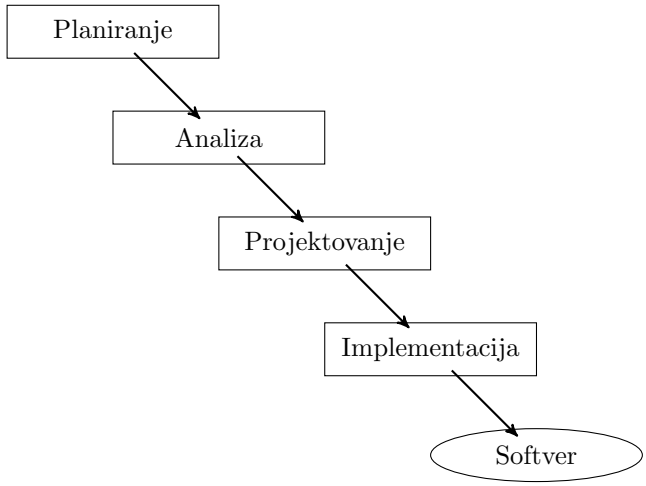
Metodologije razvoja softvera, kao što smo rekli, podrazumevaju formalizovani pristup implementacije životnog veka razvoja softvera kroz opisivanje faza razvoja i rezultata tih faza.

1.2.1 Strukturne metodologije razvoja softvera

Prve metodologije razvoja softvera su tzv. *strukturne metodologije*. One su se pojavile 1980-ih godina i značajne su po tome što su zamenile dotadašnje *ad hok* pristupe razvoju softvera.

Najpoznatija strukturna metodologija je *metodologija vodopada*. Životni ciklus razvoja softvera po metodologiji vodopada karakteriše jednosmeran prolazak kroz faze. Jednom kada se faza razvoja završi, na nju se nikad neće kasnije vraćati. Shematski, ovo se može ilustrovati kao na slici 1.1. S obzirom da se kroz svaku fazu prolazi tačno jednom, to znači da svaka faza podrazumeva iscrpno sprovođenje aktivnosti u okviru te faze, pre nego što se pređe na narednu fazu.

Neke od prednosti metodologije vodopada su: jednostavnost sprovođenja metodologije, rano formiranje svih zahteva i detaljna dokumentacija. Zbog svoje jednostavnosti, ova metodologija je poprilično korisna novim timova sa malim brojem članova koji nemaju iskustva sa drugim metodologijama razvoja. Ukoliko je softver koji se razvija relativno jednostavan i uskog obima, onda je i sagledavanje čitavog konteksta u kojem se razvija dosta jednostavno, te je manja šansa da se neki deo propusti u fazi analize i projektovanja sistema.



Slika 1.1: Metodologija vodopada.

Metodologija vodopada ima veliki broj problema. Za početak, ukoliko je softver koji se razvija složen ili obiman, onda se do faze implementacije dolazi tek nakon nekoliko meseci ili, čak, godina. U međuvremenu se zahtevi mogu promeniti, te se mora ponovo pristupiti fazi analize ili projektovanja. Ovo dovodi do produžavanja vremena razvoja softvera, probijanja rokova i budžeta i drugih sličnih problema. Dodatno, s obzirom da se implementacija vrši tek nakon što je projektovanje kompletirano, u fazi projektovanja ne postoji povratna informacija o tome da li su arhitektura i dizajn dobro osmišljeni. Ako razvojni tim previdi važan zahtev, postoji šansa da se dodatan kod koji treba implementirati neće dobro uklopiti u postojeći dizajn, što dovodi do skupog posleimplementacionog programiranja.

Pogledajmo tok razvoja jednostavne aplikacije po metodologiji vodopada[1].

Zadatak 1: Average

Metodologijom vodopada implementirati aplikaciju koja izračunava aritmetičku sredinu kompleksnih brojeva.

Planiranje Pretpostavimo da je naručilac softvera naš prijatelj koji je profesor matematike u srednjoj školi i koji želi da koristi male aplikacije na svojim časovima kako bi ubrzao računanje zadataka. On već ima nekoliko aplikacija komandne linije napisane u programskom jeziku C++ za sabiranje, oduzimanje, množenje i deljenje kompleksnih brojeva, ali mu nedostaje program za računavanje aritmetičke sredine kompleksnih brojeva.

Analiza U razgovoru sa naručiocem, shvatamo da mu je važna preciznost zapisa kompleksnih brojeva, kao i da ih može imati proizvoljan broj. Nije mu važan format unosa, odnosno, ispisa kompleksnih brojeva, osim da realni i imaginarni delovi moraju biti realni brojevi. Dodatnim razgovorima sa naručiocem, dolazimo do dodatnog zahteva, a to je ispisivanje svih učitanih kompleksnih brojeva pre izračunavanja aritmetičke sredine.

Neki od problema koji se mogu javiti su: nedoslednost formata zapisa kompleksnih brojeva i nedovoljna količina memorije u slučaju unosa ogromnog broja podataka. Prilikom projektovanja i implementacije, moramo voditi računa da se očuva format prilikom unosa, procesiranja i ispisivanja podataka. Takođe, zbog potencijalne velike količine podataka, potrebno je čuvati podatke na dovoljnoj količini memorije (dakle, ne na stek okviru neke funkcije).

Projektovanje Naručilac nam je saopštio da koristi Linux i da je upoznat sa radom komandne linije. Rešenje koje mu predlažemo jeste implementacija aplikacije komandne linije za Linux koja sa standardnog ulaza učitava podatke i ispisuje rezultat na standardni izlaz. Naručilac je zadovoljan datim predlogom. Za implementaciju ćemo koristiti programski jezik C++ i njegovu standardnu biblioteku. Format kompleksnih brojeva koji ćemo koristiti je $(real, imaginary)$, gde je $real$ realni deo, a $imaginary$ imaginarni deo kompleksnog broja. Za čuvanje

[1] S obzirom da ovo poglavlje, pored diskusije o metodologijama razvoja softvera, ima za cilj uvođenje čitalaca u programski jezik C++, zadaci koje ćemo implementirati će biti jednostavni, te će „pripremne" faze razvoja biti uglavnom vrlo kratke, odnosno, akcenat će biti stavljen na implementaciji.

unetih kompleksnih brojeva, možemo koristiti dinamički niz koji predstavlja kolekciju podataka neprekidno alociranih na hip memoriji.

Implementacija S obzirom da je važna visoka preciznost izračunavanja, koristićemo `long double` za zapis realnog i imaginarnog dela kompleksnog broja. U standardnoj biblioteci postoji šablonska klasa `std::complex<T>` sa specijalizacijom `std::complex<long double>` koja nam omogućava unos i ispisivanje kompleksnog broja, kao i ostale neophodne matematičke operacije nad ovim tipom. Ova klasa je definisana u sistemskom zaglavlju `complex`.

```
#include <complex>

int main() {
    std::complex<long double> number;

    return 0;
}
```

Učitavanje vršimo sa standardnog ulaza, koji je u programskom jeziku C++ dostupan kroz objekat `std::cin`, definisan u zaglavlju `iostream`. Ovaj objekat predstavlja primer tzv. ulaznog toka. Ulazni tokovi služe kao apstrakcija sistema iz kojih se mogu čitati podaci određenih tipova. Postoji šablonska funkcija naziva `operator>>`, koja učitava podatke određenog tipa sa nekog ulaznog toka. Argumenti ove funkcije su objekat koji predstavlja ulazni tok i promenljiva u koju će podatak biti smešten. Da bi ova funkcija mogla da se pozove nad ulaznim tokom `is` i promenljivom `x` tipa `T`, mora da postoji tzv. specijalizacija pomenute šablonske funkcije `operator>>` za tip `T` kojim je deklarisana promenljiva `x`. Srećom po nas, u standardnoj biblioteci je definisana takva specijalizacija za tip `std::complex<long double>`, tako da možemo koristiti `operator>>` za čitanje kompleksnog broja sa standardnog ulaza (ili bilo kojeg drugog ulaznog toka). Štaviše, format kojim se kompleksan broj učitava je upravo onaj koji smo definisali u fazi projektovanja softvera. Povratna vrednost ove funkcije je upravo objekat ulaznog toka koji je prosleđen kao prvi argument.

Napomenimo da ćemo u nastavku implementacije prikazivati samo segmente kodova za zahtev koji se trenutno implementira. Od čitaoca se očekuje da je u stanju da rekonstruiše kod na osnovu ovih segmenata tokom čitanja teksta. Kompletna rešenja su data u dodatku A.

```
#include <iostream>
```

```
...

int main() {
    std::complex<long double> number;

    while (std::cin >> number) {

    }

    return 0;
}
```

Primetimo sintaksu kojom se poziva `operator>>`. Svi operatori u programskom jeziku C++ imaju sposobnost da se pozivaju na dati način, umesto da se pozivaju eksplicitno: `operator>>(std::cin, number)`. Zbog toga ćemo češće govoriti „operator `>>`" nego „funkcija `operator>>`".

Kada se pročita karakter za kraj unosa (`EOF`), objekat standardnog ulaza će biti postavljen kao nevalidan, pa nakon što se vrati kao povratna vrednost operatora `>>`, biće izvršena implicitna konverzija u tip `bool` i to u vrednost `false`, pa možemo iskoristiti upravo tu činjenicu da završimo sa učitavanjem.

Za čuvanje unetih kompleksnih brojeva, možemo koristiti šablonsku klasu `std::vector<T>` koja predstavlja kolekciju podataka neprekidno alociranih na hip memoriji. Ova kolekcija ima metod `push_back` koji očekuje da joj prosledimo objekat koji će smestiti u taj vektor. Ova kolekcija je definisana u sistemskom zaglavlju `vector`.

```
#include <vector>
...

int main() {
    std::complex<long double> number;
    std::vector<std::complex<long double>> numbers;

    while (std::cin >> number) {
        numbers.push_back(number);
    }

    return 0;
}
```

Pređimo na ispisivanje učitanih kompleksnih brojeva. U standardnoj biblioteci programskog jezika C++, na raspolaganju nam je objekat `std::cout` koji predstavlja standardni izlaz. Ovo je primer tzv. izlaznih tokova, odnosno, sistema u koje je moguće upisivati podatke. Nad izlaznim tokovima je moguće primetiti šablonski operator `<<`. Semantika ovog operatora u potpunosti odgovara operatoru `>>`, sa razlikom da se primenjuje nad izlaznim tokovima i da služi za upisivanje podataka u taj izlazni tok.

Prolazak kroz vektor je moguće uraditi na više načina. Jedan način jeste korišćenje klasične `for` petlje sa inicijalizatorom, uslovom izlaska i korakom, u kombinaciji sa metodom `size` definisanim u klasi `std::vector<T>` koji vraća broj elemenata u vektoru (u konstantnoj vremenskoj složenosti) i operatoru `[]` kojim se dohvata element na datom indeksu u vektoru.

```
int main() {
    ...

    for (size_t i = 0; i < numbers.size(); ++i) {
        std::complex<long double> number = numbers[i];
        std::cout << number << std::endl;
    }

    return 0;
}
```

Drugi način jeste korišćenje tzv. kolekcijske `for` petlje. Ona ima nešto drugačiju strukturu od klasične `for` petlje. U kolekcijskoj `for` petlji se navodi:

1. tip elementa koji se nalazi u kolekciji;
2. identifikator koji će se koristiti za deklarisanje promenljive koja će u jednoj iteraciji sadržati vrednost jednog elementa iz kolekcije;
3. karakter dvotačke;
4. kolekcija kroz koju se iterira.

Prednost korišćenja kolekcijske `for` petlje jeste u tome što ne moramo da eksplicitno upravljamo indeksima kolekcije, pa smanjujemo verovatnoću za nastanak grešaka u aplikacijama. Sa druge strane, u svojoj osnovnoj varijanti koja

je ovde prikazana, kolekcijska `for` petlja prolazi redom kroz elemente, tako da ne možemo definisati korak, kao što to možemo u klasičnoj `for` petlji[2].

```
int main() {
    ...

    for (std::complex<long double> number : numbers) {
        std::cout << number << std::endl;
    }

    return 0;
}
```

Ono o čemu moramo da vodimo računa jeste efiksnost naših aplikacija. Trenutno, postoji problem sa korišćenjem bilo kojeg od opisanih načina za prolazak kroz vektor kompleksnih brojeva, a to je suvišno kopiranje svakog kompleksnog broja prilikom iteriranja kroz vektor `numbers`. Naime, u oba pristupa, ono što se dešava „u pozadini" jeste da se u svakoj iteraciji kopira jedan-po-jedan kompleksan broj iz vektora (koji se nalazi na hip memoriji) u promenljivu identifikatora `number` koja se nalazi na stek memoriji funkcije `main`. U slučaju da vektor sadrži milion kompleksnih brojeva, to će značiti milion kopiranja tih objekata samo zarad ispisivanja njihove vrednosti.

Jedan način da to predupredimo jeste korišćenje pokazivača.

```
int main() {
    ...

    for (size_t i = 0; i < numbers.size(); ++i) {
        std::complex<long double> *number = &numbers[i];
        std::cout << *number << std::endl;
    }

    return 0;
}
```

Alternativno, ukoliko ne želimo da vodimo računa o indeksima, možemo koristiti iteratore. Iterator je koncept koji predstavljaju apstrakciju pokazivača nad

[2]Naravno, postoje mehanizmi kojima se ovo ipak može postići u korišćenjem kolekcijske `for` petlje, ali o njima neće biti diskusije u ovom tekstu.

elementima neke kolekcije. Skoro svaka kolekcija ima definisane metode `begin` i `end` koji vraćaju iteratore na početak, odnosno, kraj kolekcije, redom. Iteratori se nazivaju tako zato što olakšavaju proces iteracije kroz kolekciju. Ako imamo iterator na jedan element kolekcije, možemo „pomeriti" taj iterator na naredni element u toj kolekciji pozivom operatora `++`. Dohvatanje vrednosti na koju „pokazuje" iterator se vrši njegovim dereferenciranjem, tj. pozivom operatora `*` slično kao kod pokazivača.

```
int main() {
    ...

    for (std::vector<std::complex<long double>>::iterator i =
            numbers.begin(); i < numbers.end(); ++i) {
        std::cout << *i << std::endl;
    }

    return 0;
}
```

Očigledno, iteratori imaju svoj tip, koji zavisi od kolekcije kroz koju se iterira, kao što vidimo u kodu iznad. Eksplicitno navođenje ovih tipova može biti vrlo zamorno. Srećom, postoji ključna reč `auto` koja vrši automatsko prepoznavanje tipa vrednosti koji se dodeljuje promenljivoj.

```
int main() {
    ...

    for (auto i = numbers.begin(); i < numbers.end(); ++i) {
        std::cout << *i << std::endl;
    }

    return 0;
}
```

Ukoliko bismo želeli da insistiramo na korišćenju kolekcijske `for` petlje, upali bismo u nevolju, jer pokazivače i iteratore ne možemo tako lako koristiti u kolekcijskoj `for` petlji. Da bi se izbeglo kopiranje elemenata, kao i korišćenje pokazivača, u programski jezik C++ je uveden koncept referenci. Referenca[3],

[3] C++ je izuzetno složen programski jezik. Na primer, ovako definisan pojam reference

poput pokazivača, predstavlja promenljivu čija je vrednost adresa neke druge adresabilne vrednosti (promenljive, elementa niza, elementa vektora, itd.). Razlika je u tome što ne moramo da koristimo operatore za adresiranje (`&`) i dereferenciranje (`*`) kao kad se koriste pokazivači. Umesto toga, samo je neophodno da prilikom definicije reference navedemo da je u pitanju referenca na neki tip, što se navodi karakterom `&` nakon odgovarajućeg tipa. Na primer, ako postoji deklarisana promenljiva `int x;`, onda možemo kreirati referencu na tu promenljivu navođenjem `int &r_x = x;` i nadalje možemo koristiti `r_x` podjednako kao i `x`.

```
int main() {
    ...

    for (std::complex<long double> &number : numbers) {
        std::cout << number << std::endl;
    }

    return 0;
}
```

Reference se mogu koristiti paralelno sa ključnom rečju `auto`. Tako, na primer, ako postoji deklarisana promenljiva `int x;`, onda možemo kreirati referencu na tu promenljivu navođenjem `auto &r_x = x;`. Ključna reč `auto` će u ovom primeru zaključiti da je vrednost `x` tipa `int`, pa će promenljiva `r_x` biti tipa `int &`, tj. biće tipa „referenca na `int`".

```
int main() {
    ...

    for (auto &number : numbers) {
        std::cout << number << std::endl;
    }
```

predstavlja tzv. levu referencu ili jednostruku referencu. Postoje i tzv. desne reference ili dvostruke reference, o kojima će biti reči u nekim drugim poglavljima koji, uprošćeno rečeno, čuvaju adrese na privremene vrednosti, odnosno, vrednosti koji žive vrlo kratko i na koje se u normalnim okolnostima ne može referisati (na primer, takve su doslovne vrednosti). Stvari se dodatno usložnjavaju uvođenjem parametarskog polimorfizma i koncepta univerzalnih referenci kao referenci šablonskih parametara koje mogu da igraju ulogu i levih i desnih referenci. Visok stepen složenosti programskog jezika C++ je njegova najveća mana, zbog čega je i najviše kritikovan od strane programerske zajednice.

```
    return 0;
}
```

Kao i preko pokazivača, i putem reference je moguće promeniti vrednost objekta čiju adresu ta referenca čuva. Ponovo, ako postoji definicija promenljive `int x = 0;` i referenca na tu promenljivu `auto &r_x = x;`, onda izračunavanjem naredbe `r_x = 1;` se menja vrednost objekta čiju adresu čuva referenca `r_x`, tj. promenljiva `x` dobija vrednost `1`. Isto tako, u našem kodu iznad niko ne garantuje da će vrednosti elemenata u vektoru `numbers` ostati nepromenjene nakon izračunavanja kolekcijske `for` petlje. Ukoliko želimo da forsiramo nepromenljivost podataka u našem kodu, od ključnog je značaja da takve objekte (ili pokazivače ili reference na te objekte) da definišemo kao konstantne, navođenjem ključne reči `const` ispred tipa. Ukoliko programer pokuša da promeni vrednost konstantne promenljive, kompilator će prijaviti grešku i prevođenje će biti neuspešno, što je mnogo bolje ponašanje, nego da se greške pronalaze u fazi izvršavanja debagovanjem.

```
int main() {
    ...

    for (const auto &number : numbers) {
        std::cout << number << std::endl;
    }

    return 0;
}
```

Da bismo izračunali aritmetičku sredinu kompleksnih brojeva koji se nalaze u vektoru, možemo koristiti neke od `for` petlji o kojima smo diskutovali do sada, iterirajući kroz vektor, dohvatajući referencu (ili iterator) na element, pa zatim sabirati jedan-po-jedan kompleksan broj sa tekućom vrednošću sume, počevši od neutrala. Tu sumu je zatim potrebno podeliti veličinom vektora i ispisati na standardni izlaz.

```
int main() {
    ...

    // Podrazumevano se kreira kompleksni broj (0,0).
    const std::complex<long double> neutral;
```

```
    const auto complex_sum = neutral;
    for (const auto &number : numbers) {
        complex_sum += number;
    }
    const auto numbers_size = const_cast<double>(number.size());
    std::cout << complex_sum / numbers_size << std::endl;

    return 0;
}
```

Za sabiranje elemenata vektora možemo iskoristiti šablonsku funkciju standardne biblioteke `std::accumulate`, definisanu u sistemskom zaglavlju `numeric`. Ova funkcija ima naredna četiri argumenta:

1. iterator na početak kolekcije;
2. iterator na kraj kolekcije;
3. vrednost koja predstavlja neutral za sabiranje. Tip povratne vrednosti ove funkcije će biti tip neutrala koji se prosledi kao ovaj argument, tako da je neophodno da vodimo računa na prosledimo vrednost odgovarajućeg tipa.
4. (opcioni argument) vrednost koja će se iskoristiti kao funkcija za sabiranje. Očekuje se da funkcija ima dva argumenta, pri čemu je prvi argument trenutna „akumulirana" vrednost, a drugi argument naredni element iz kolekcije. Podrazumevano se koristi operator `+`.

S obzirom da je operator `+` definisan za kompleksne brojeve, ne moramo da sami pišemo funkciju za sabiranje kompleksnih brojeva. Dodatno, u skladu sa napomenom iznad, moramo da se osiguramo da kao neutral prosledimo kompleksan broj, a ne `0`, `0.0` ili nešto slično, kako bi povratna vrednost funkcije bila upravo kompleksan broj.

```
#include <numeric>
...

int main() {
    ...

    // Podrazumevano se kreira kompleksni broj (0,0).
    const std::complex<long double> neutral;
```

```
    const auto complex_sum =
        std::accumulate(numbers.begin(), numbers.end(), neutral);
    const auto numbers_size = const_cast<double>(number.size());
    std::cout << complex_sum / numbers_size << std::endl;

    return 0;
}
```

Kompletan kod je dat u rešenju 1.

Prevođenjem za Linux sistem i pokretanjem programa iz komandne linije ispitujemo da li program radi korektno. Za sada ćemo testirati programe tek na nekoliko test primera.

Test primer 1

Standardni ulaz

```
(1, 5)
(2, 4)
(7, 1)
EOF
```

Standardni izlaz

```
(3.33333,3.33333)
```

Test primer 2

Standardni ulaz

```
(5, 0)
(-2, 0)
(6, 0)
EOF
```

Standardni izlaz

```
(3,0)
```

Test primer 3

Standardni ulaz

```
EOF
```

Standardni izlaz

```
(0,0)
```

Naručiocu isporučujemo izvršnu datoteku i uputstvo za korišćenje aplikacije sa test primerima. Nakon testiranja programa na naručiočevom računaru, zaključujemo da je isporuka uspešno obavljena i završavamo projekat.

Implementirajmo još jednu jednostavnu aplikaciju metodom vodopada. Iako će i ovog puta akcenat biti na implementaciji, ovde ćemo nešto detaljnije opisati faze planiranja i analize, sa detaljnijim prikazom primera dijaloga koji se može voditi između kupca i razvijaoca.

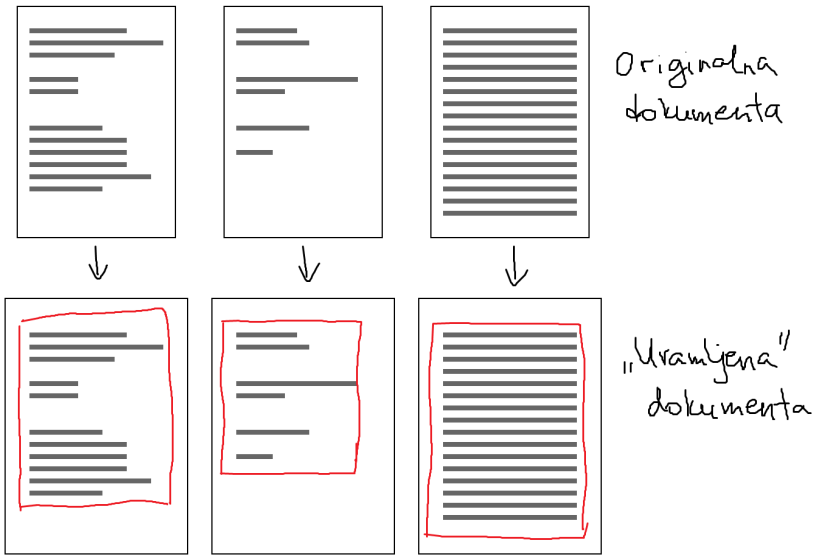
Zadatak 2: Stars

Metodologijom vodopada razviti aplikaciju koja za dati tekst u datoteci prikazuje isti taj tekst uokviren „ramom".

Planiranje Naručilac nam je saopštio da koristi Linux i da je upoznat sa radom komandne linije.

- Rešenje koje mu predlažemo jeste implementacija konzolne aplikacije za Linux koja čita tekst zapisan u tekstualnoj datoteci `input.txt` i ispisuje rezultat u datoteci `output.txt`. Naručilac objašnjava da želi da može da koristi program za proizvoljne datoteke koje već postoje na sistemu, bez potrebe da pravi njihove kopije.

- U tom slučaju predlažemo da se putanje do ulazne i izlazne datoteke navode kao obavezni argumenti komandne linije. Naručiocu se dopada ideja da može da navodi proizvoljne putanje do ulaznih datoteka, ali preferira da ne navodi putanju do izlazne datoteke, već bi želeo da se ona automatski napravi u istom direktorijumu u kojem se nalazi ulazna datoteka.
- Nakon dijaloga sa naručiocem, dolazimo do finalnog predloga rešenja: Naručilac pri pokretanju aplikacije navodi tačno jedan argument komandne linije koji predstavlja putanju do ulazne datoteke, a aplikacija će kreirati izlaznu datoteku sa odgovarajućim imenom koje će biti prepoznatljivo u istom direktorijumu. Naručilac prihvata ovaj predlog i prelazi se na narednu fazu razvoja.



Slika 1.2: Grafički prikaz ulaznih podataka za zadatak 2 i skica rezultata koju je naručilac softvera priložio.

Analiza Naručilac nam je isporučio primere tekstualnih dokumenata i skice rezultata, što je prikazano na slici 1.2. Sadržaj dokumenata je precrtan radi

očuvanja bezbednosti podataka. U nastavku dajemo primer dijaloga iz kojeg dolazimo do novih informacija o potrebama naručioca.

- Razvijači: Iz priloženih primera dokumenata deluje da svaki dokument ima različit broj linija, kao i da su linije potencijalno različitih dužina.
- Naručilac: Tako je. Zapravo, primere koje sam vam prosledio imaju mali broj linija, ali su sama dokumenta reda veličine milion linija.
- Razvijači: Dakle, moramo da vodimo računa o efikasnosti rada programa.
- Naručilac: Međutim, same linije su relativno kratke, na primer, oko 80 karaktera, s obzirom da dokumente šaljemo putem mejla, pa želimo da primaoci tih pisma mogu da pročitaju i na mobilnim uređajima bez velikih poteškoća.
- Razvijači: Da li to znači da možemo da postavimo gornju granicu za dužinu linije?
- Naručilac: Bolje ne. U planu nam je da pravimo kopije dokumenata gde je tekst biti različite širine za različite korisnike. Tako će neki od njih čitati „šira" dokumenta na većim ekranima, pa bi aplikacija koju pravite trebalo da podrži i takva dokumenta.
- Razvijači: U redu. S obzirom da su tekstualne datoteke, da li je u redu da „ram" bude sačinjen od karaktera zvezdica?
- Naručilac: Da, tako smo zamislili. Stavite i jedan „prazan" red sa svih strana unutar rama.

Iz analize sistema vidimo da su dokumenti veliki, pa bi trebalo izbegavati skupe operacije. S obzirom da će program biti primenjen na velike tekstove, potrebno je čuvati podatke u nekoj sekvencijalnoj strukturi podataka na hip memoriji.

Projektovanje Za implementaciju ćemo koristiti programski jezik C++ i njegovu standardnu biblioteku. Za čuvanje unetih linija, možemo koristiti vektor niski. Kao i u zadatku 1, sav kod ćemo pisati u funkciji `main`.

Osmislimo sada način implementacije rama. Iz istraživanja i analize sistema smo zaključili da je širina rama jednaka širini najduže linije u dokumentu plus četiri karaktera. Ovo je zbog toga što, na primer, linija ulazne datoteke

„`Zdravo`" u izlaznoj datoteci postaje „`* Zdravo *`", dakle imamo po dva dodatna karaktera ispred i iza svake linije. Dakle, neophodno je pronaći dužinu najveće linije. S obzirom da program treba da radi efikasno, ne bi trebalo da prvo učitamo linije, pa tek nakon toga da pronađemo dužinu najveće linije (jer bismo u tom slučaju dva puta linearno prolazili kroz ceo vektor). Umesto toga, izračunavanje najduže linije ćemo uraditi u hodu, tokom učitavanja linija.

Već u fazi projektovanja možemo prepoznati neke probleme koji se mogu pojaviti, kao što je neuspešno otvaranje datoteka. Zbog toga, u zavisnosti od vrste problema do kojeg može doći, naša aplikacija definiše naredne povratne kodove:

- Kod 0 označava uspešan završetak programa.
- Kod 1 označava problem sa argumentima komandne linije.
- Kod 2 označava problem sa ulaznom datotekom.
- Kod 3 označava problem sa ulaznom datotekom.

Apstraktan opis implementacije se sastoji od narednih koraka:

- Proveriti argumente komandne linije. U slučaju greške, obavestiti korisnika o grešci i prekinuti izvršavanje sa kodom 1.
- Otvoriti ulaznu datoteku za čitanje ako je moguće. U slučaju greške, obavestiti korisnika o grešci i prekinuti izvršavanje sa kodom 2.
- Pročitati sve linije iz datoteke i smestiti ih u vektor niski. Prilikom učitavanja izračunati dužinu najduže linije. Zatvoriti ulaznu datoteku.
- Konstruisati putanju do izlazne datoteke na osnovu putanje do ulazne datoteke. Otvoriti izlaznu datoteku za pisanje ako je moguće. U slučaju greške, obavestiti korisnika o grešci i prekinuti izvršavanje sa kodom 3.
- Upisati linije iz vektora niski, pri čemu se one ispisuju kao deo rama, uzimajući u obzir diskusiju izloženu iznad o konstrukciji „rama". Zatvoriti izlaznu datoteku.
- Završiti izvršavanje sa kodom 0.

Implementacija Implementaciju započinjemo proveravanjem argumenata komandne linije.

```
#include <iostream>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        std::cerr
            << "Neispravan broj argumenata komandne linije. "
            << "Molimo prosledite samo naziv ulazne datoteke."
            << std::endl;
        return 1;
    }

    return 0;
}
```

Već u ovom trenutku možemo da primetimo korišćenje tzv. magičnih vrednosti. Zbog toga, uvedimo prvo jedan enumerator koji prebrojava povratne kodove naše aplikacije. Zatim, uvedimo konstantu za određivanje očekivanog broja argumenata.

```
#include <iostream>

enum ReturnCodes {
    Success = 0,
    InvalidNumberOfArguments,
    CannotOpenInputFile,
    CannotOpenOutputFile
};

int main(int argc, char *argv[]) {
    const auto expected_number_of_arguments = 2;

    if (argc != expected_number_of_arguments) {
        std::cerr
            << "Neispravan broj argumenata komandne linije. "
            << "Molimo prosledite samo naziv ulazne datoteke."
            << std::endl;
        return ReturnCodes::InvalidNumberOfArguments;
```

```
    }

    return ReturnCodes::Success;
}
```

Nakon provere argumenata, prelazimo na učitavanje linija iz datoteke koja je navedena kao prvi argument. Prvo pokušavamo da otvorimo datoteku za čitanje, konstruisanjem objekta `std::ifstream` koji predstavlja ulazni datotečki tok, definisanom u sistemskom zaglavlju `fstream`. Ako prilikom konstrukcije ovog toka prosledimo putanju do datoteke, on će ujedno pokušati da je otvori. Uspešnost ove operacije možemo proveriti pozivom metoda `is_open`.

```
#include <fstream>

...

std::ifstream input_file(argv[1]);
if (!input_file.is_open()) {
    std::cerr
        << "Neuspesno otvaranje ulazne datoteke: "
        << argv[1]
        << std::endl;
    return ReturnCodes::CannotOpenInputFile;
}
```

Ako je datoteka uspešno otvorena, možemo da učitamo njen sadržaj. Najjednostavniji način da ovo uradimo jeste da učitavamo jednu po jednu liniju funkcijom `std::getline`, definisanu u sistemskom zaglavlju `string`. Ova funkcija ima tri argumenta:

1. ulazni tok;
2. objekat klase `std::string` u koji će biti upisani pročitani karakteri sa ulaznog toka;
3. (opcioni argument) niska koja predstavlja delimiter. Podrazumevano se koristi oznaka za novi red.

Povratna vrednost ove funkcije je ulazni tok iz kojeg se vršilo čitanje, tako da možemo ovu informaciju da iskoristimo kao uslov izlaska iz petlje (kada se pročita kraj datoteke, tj. `EOF`, ulazni tok će biti invalidiran, odnosno, implicitno

konvertibilan u `false`). Nakon što završimo učitavanje, zatvaramo datoteku, s obzirom da nam nije više neophodna.

```
#include <vector>
#include <string>

...

std::vector<std::string> messages;
std::string line;

while (std::getline(input_file, line)) {
    messages.push_back(line);
    line.clear();
}

input_file.close();
```

Ako objekat klase `std::string`, definisane u sistemskom zaglavlju `string`, konstruišemo bez argumenata, podrazumevano će ta niska biti prazna. Ovu činjenicu možemo da iskoristimo tako što ćemo prosleđivati objekat `line` kao drugi argument funkciji `std::getline` koji će upisati jednu liniju iz ulazne datoteke `input_file` u objekat `line`. Zatim, učitanu liniju unosimo na kraj vektora i čistimo njen sadržaj kako bi ona bila prazna za narednu iteraciju.

Prisetimo se da smo rekli da ćemo tokom učitavanja linija iz datoteke usput i da pronalazimo dužinu najveće linije.

```
#include <vector>
#include <string>

...

std::vector<std::string> messages;
std::string line;
size_t maximum_length = 0u;

while (std::getline(input_file, line)) {
    messages.push_back(line);
    maximum_length = std::max(maximum_length, line.size());
    line.clear();
```

```
}

input_file.close();
```

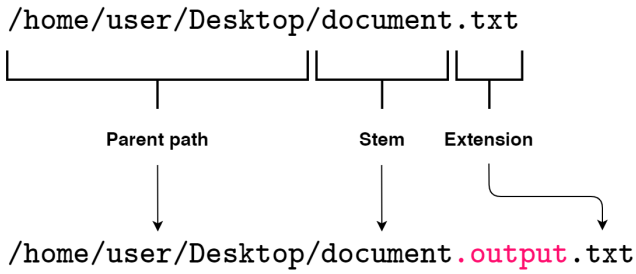
Takođe, ne smemo da zaboravio naručiocev zahtev da je potrebno dodati prazne redove ispred i iza teksta.

```
#include <vector>
#include <string>

...

std::vector<std::string> messages;
std::string line;
size_t maximum_length = 0u;

messages.push_back(std::string());
while (std::getline(input_file, line)) {
    messages.push_back(line);
    maximum_length = std::max(maximum_length, line.size());
    line.clear();
}
messages.push_back(std::string());

input_file.close();
```

Pređimo sada na ispisivanje linija u izlaznu datoteku. Kao i u slučaju učitavanja, potrebno je da otvorimo datoteku za pisanje. Prvo definišemo putanju na kojoj se nalazi datoteka u koju će biti upisan rezultat. Na slici 1.3 opisan je način izračunavanja putanje do izlazne datoteke na osnovu putanje do ulazne datoteke, koja je data kao prvi argument komandne linije.

Vidimo da je neophodno upravljati raznim informacijama iz putanje koja je zadata niskom. U tu svrhu nam može pomoći klasa `std::filesystem::path`, definisana u sistemskom zaglavlju `filesystem`. Ova klasa definiše razne metode za izračunavanje delova putanje i upravljanje njima.

```
#include <filesystem>
namespace fs = std::filesystem;

...
```



Slika 1.3: Grafički prikaz konstrukcije putanje do izlazne datoteke od putanje do ulazne datoteke.

```
const auto input_filename = fs::path(argv[1]);
const auto directory_name = input_filename.parent_path();
const auto input_stem = input_filename.stem();
const auto extension = input_filename.extension();

auto output_filename = directory_name;
output_filename /= input_stem;
output_filename += fs::path(".output");
output_filename += extension;
```

Kao što vidimo, prvi korak je izračunavanje svakog neophodnog dela: naziva direktorijuma u kojem se ulazna datoteka nalazi (`directory_name`), naziv ulazne datoteke bez ekstenzije (`input_stem`), kao i ekstenzije (`extension`). Putanju do izlazne datoteke rekonstruišemo koristeći operator `/=` koji nadovezuje dve putanje korišćenjem separatora putanje[4] i operator `+=` koji nadovezuje dve putanje bez korišćenja separatora putanje.

Zatim otvaramo datoteku sa pisanje konstruisanjem objekta `std::ofstream`, koji ima sličnu semantiku kao `std::ifstream`. Primetimo da putanja do datoteke može da se zada i kao objekat `std::filesystem::path`.

```
std::ofstream output_file(output_filename);
if (!output_file.is_open()) {
    std::cerr
```

[4]U zavisnosti od operativnog sistema na kojem se aplikacija izvršava, biće korišćen odgovarajući separator putanje.

```
        << "Neuspesno otvaranje izlazne datoteke: "
        << output_filename
        << std::endl;
    return ReturnCodes::CannotOpenOutputFile;
}
```

Konstruišemo liniju koja predstavlja prvu (gornju) i poslednju (donju) liniju „rama" od zvezdica. Ispisivanje u izlaznu datoteku započinjemo ispisivanjem gornje linija „rama", pa zatim prelazimo na ispisivanje jedne po jedne linije iz vektora, pri čemu je neophodno da pre svake linije ispišemo karaktere zvezdice i razmaka (levi deo „rama"), a nakon svake linije da ispišemo karaktere razmaka i zvezdice (desni deo „rama"). Primetimo još jednu stvar – s obzirom da su linije različite dužine, desni deo „rama" neće biti poravnat automatski. Neophodno je da nakon ispisivanja svake linije, a pre ispisivanja desnog dela „rama", ispišemo onoliko razmaka koliko tekućoj liniji nedostaje do širine najduže linije. Nakon ispisivanja svih linija, ispisujemo donju liniju „rama" i zatvaramo datoteku za pisanje.

```
const std::string bar(maximum_length + 4, '*');

output_file << bar << std::endl;
for (const auto &line : messages) {
    output_file
        << "* " << line
        << std::string(maximum_length - line.size(), ' ')
        << " *" << std::endl;
}
output_file << bar << std::endl;

output_file.close();
```

Kompletan kod je dat u rešenju 2.

Naručiocu isporučujemo izvršnu datoteku i uputstvo za korišćenje aplikacije sa test primerima. Nakon testiranja programa na naručiočevom računaru, zaključujemo da je isporuka uspešno obavljena i završavamo projekat.

1.2.2 Metodologije rapidnog razvoja softvera

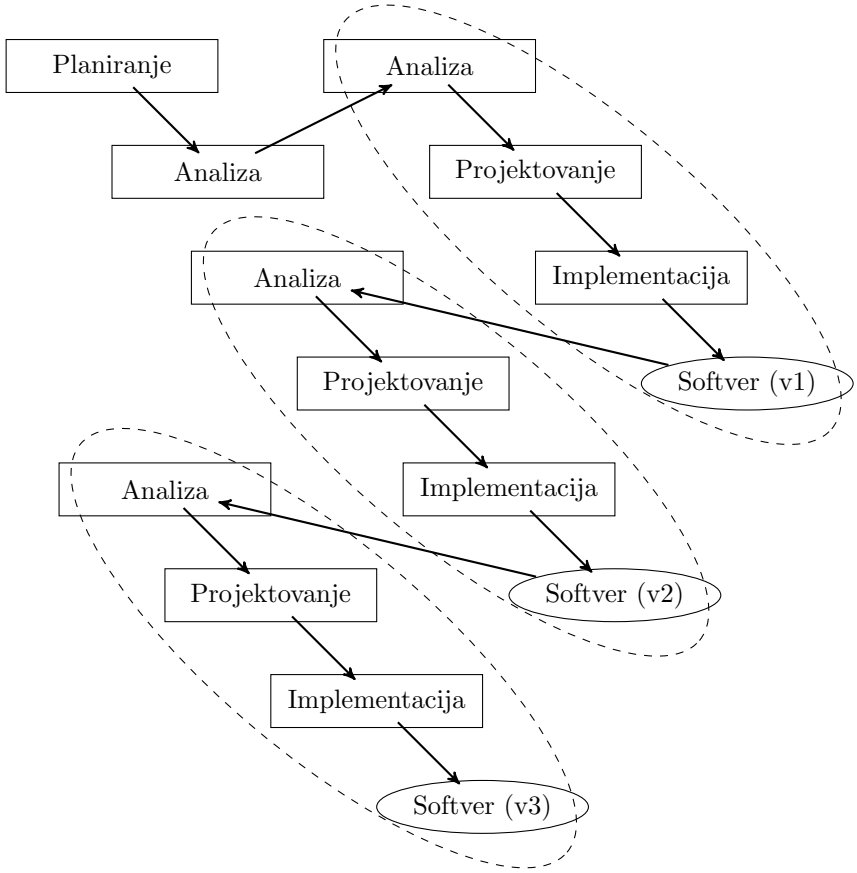
Nakon strukturnih metodologija razvoja softvera, 1990-ih godina su se pojavile *metodologije rapidnog razvoja softvera* (eng. *rapid application development*, skr. *RAD*). One su imale za cilj da reše neke od nedostataka strukturnih metodologija, pre svega da ubrzaju isporuku softvera kupcima. Brže isporučivanje sistema vodi ka tome da se kupci bolje pripreme za korišćenje sistema, kao i da se stvori veće samopouzdanje između kupaca i razvijaoca softvera.

Jedna od metodologija rapidnog razvoja softvera jeste *metodologija faznog razvoja*. Osnovna ideja ove metodologije jeste u razbijanju celokupnog sistema u niz *verzija* koje se razvijaju sekvencijalno. Drugim rečima, u fazi analize se identifikuje celokupni sistem na konceptualnom nivou, a zatim svi učesnici u razvoju kategorišu zahteve sistema u verzije. Grafički prikaz ove metodologije je dat na slici 1.4.

Oni zahtevi koji su najfundamentalniji i najvažniji za rad softvera se kategorišu u verziju 1 softvera. Nakon toga se prelazi na faze detaljnije analize zahteva, zatim projektovanja i implementacije, ali samo za deo sistema koji je identifikovan verzijom 1! Kada se prva verzija implementira, započinje se na radu verzije 2. Dodatna analiza se vrši na osnovu prethodno identifikovanih zahteva i kombinuje se sa novim idejama koje su proizašle iz projektovanja i implementacije verzije 1. Nakon ovoga se projektuje i implementira verzija 2. Ceo proces se ponavlja sve dok se ceo softver ne implementira.

Očigledna prednost ove metodologije je u brzom isporučivanju verzije 1 softvera koji korisnici mogu da koriste. Iako softver nema sve željene funkcionalnosti, on ipak predstavlja upotrebljiv rezultat koji služi svrsi za koju je namenjen, usled pažljivog odabira zahteva koji su implementirani u toj verziji. Dodatno, s obzirom da korisnici imaju mogućnost da rade sa softverom relativno pri početku njegovog razvoja, ukoliko se identifikuju važni dodatni zahtevi koji su bili inicijalno propušteni, oni se jeftino mogu dodati u narednoj verziji, za razliku od strukturnih metodologija, gde su takvi postupci veoma skupi ili, čak, nemogući.

S obzirom da sve prednosti ove metodologije zavise od ispravnog odabira funkcionalnosti, postoji rizik da će učesnici u razvoju odabrati neadekvatan skup funkcionalnosti, pogotovo ako su neiskusni u prepoznavanju i odabiru važnih zahteva. Posledica lošeg izbora podskupa zahteva jeste rano isporučivanje softvera koji je funkcionalno neupotrebljiv (ili nedovoljno upotrebljiv), što dovodi do gubitka potencijala za njegovu iskorišćenost u poslovnim primenama kupca.



Slika 1.4: Metodologija faznog razvoja.

Zadatak 3: Fast typing

Metodologijom faznog razvoja razviti aplikaciju koja omogućava korisniku da igra igru brzog kucanja.

Planiranje U igri brzog kucanja igraču se prikazuju nasumične reči iz neke baze reči koje on mora da otkuca u što bržem vremenu. Svaka reč nosi unapred definisan broj poena (na primer, broj poena odgovara dužini reči). Igrač osvaja poene unošenjem reči koje su mu prikazane, pri čemu vrednost reči opada kako igra teče. S obzirom da su jasni okviri u kojima ćemo implementirati ovu igru, kao i da su nam poznata pravila, nećemo se zadržavati na fazi planiranja. Napomenimo samo da će aplikacija koju razvijamo biti aplikacija komandne linije.

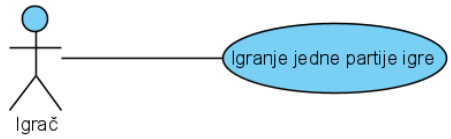
Analiza sistema Faza analize ima veliku važnost u metodologiji faznog razvoja, s obzirom da je neophodno da sagledamo u potpunosti zahteve koji su postavljeni pred nama, kako bismo mogli da odaberemo one koji će biti deo verzije 1. Ono što prvo primećujemo analizom pravila igre jeste da je neophodno da postoje mehanizmi za upravljanje rečima. Pored upravljanja rečima, od korisnika moramo da tražimo unos reči i da vršimo obradu unosa, a takodje, potrebno je i prikazivati trenutno stanje igre. Dakle, potrebno je osmisliti interakciju sa korisnikom. Konačno, aplikacija mora implementirati sistem za upravljanje poenima. Na osnovu ove analize, dolazimo do narednih funkcionalnih zahteva:

1. Aplikacija ima skup reči.
2. Učitavati podatke iz neke baze reči.
3. Odabrati podskup reči koji će se prikazivati u toku jedne igre.
4. Postoji mehanizam inicijalnog bodovanja reči.
5. Omogućiti prikaz reči na ekranu.
6. Reči treba prikazivati na neki interesantan način.
7. Igrač treba da unosi reči.
8. Aplikacija izračunava i pamti broj poena koji je korisnik dobio u slučaju ispravno unesene reči.
9. Reči koje je korisnik ispravno uneo moraju da nestanu sa prikaza.
10. Ako korisnik unese neispravnu reč, potrebno je definisati pravilo za kažnjavanje.
11. Aplikacija prikazuje trenutno osvojen broj poena tokom igre.
12. Potrebno je pratiti vreme kroz igru.

13. Na kraju igre se prikazuje rezultat.

Na osnovu ovih zahteva, potrebno je prepoznati delove logike koji čine jednu kompletnu celinu. Takve logičke celine se nazivaju *slučajevi upotrebe*. Na primer, iz definicije zahteva igre brzog kucanja možemo prepoznati jedan slučaj upotrebe – „Igranje jedne partije igre". Svaki slučaj upotrebe opisuje jedan poslovni proces koji naš softver treba da implementira. Naravno, složeniji softveri implementiraju znatno veći broj ovih logičkih celina.

Grafički, slučajevi upotrebe se predstavljaju UML dijagramima slučajeva upotrebe. U sekciji 1.3 ćemo detaljnije govoriti o načinu izrade dijagrama slučajeva upotrebe, a za sada ćemo samo prikazati dijagram slučaja upotrebe „Igranje jedne partije igre" na slici 1.5.



Slika 1.5: UML dijagram slučaja upotrebe „Igranje jedne partije igre".

Na dijagramu je prikazan i jedan *akter*, „Igrač", koji učestvuje u tom slučaju upotrebe. Slučajevi upotrebe i akteri su nam korisni za prepoznavanje poslovnih procesa koje naš softver treba da implementira, kao i korisničkih uloga u tim procesima.

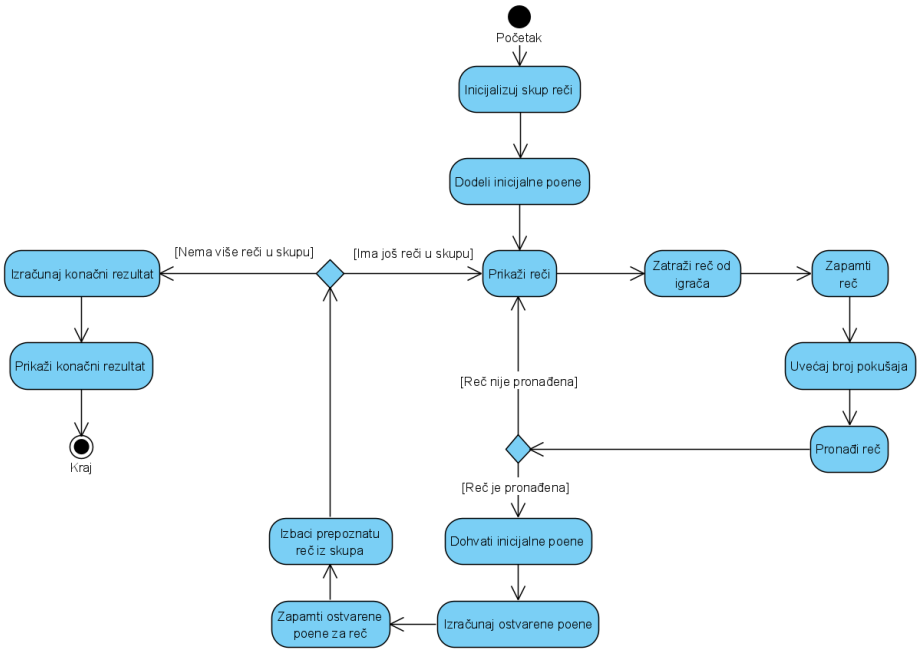
Priprema za verziju 1 Identifikujmo sada podskup zahteva koji će biti implementirani u verziji 1. Od prepoznatih funkcionalnih zahteva, jedan podskup zahteva koji bi se mogao naći kao deo verzije 1 je $\{1, 4, 5, 7, 8, 9, 13\}$. Ovde ćemo skrenuti pažnju da je skup funkcionalnosti koji smo odabrali oko 50% svih pronađenih zahteva. U zavisnosti od složenosti i obima projekta, ovakav skup može biti previše opširan za verziju 1. Naime, što je obim funkcionalnosti koje se implementiraju u nekoj verziji veći, to su faze analize, projektovanja i implementacije u toj verziji složenije i to vodi ka značajnom opterećenju prvih verzija u odnosu na naredne. Drugim rečima, metodologija konvergira ka metodologiji vodopada. Sa druge strane, nekada će 50% ključnih zahteva zapravo obuhvatati značajno manji obim celog sistema, te u takvom slučaju ne dolazi do problema. Dakle, u početnoj fazi analize treba posebno voditi računa o definisanju skupa funkcionalnosti, njihovim prioritetima i kategorisanjem u verzije.

Analiza verzije 1 Pređimo sada na analizu svakog prepoznatog zahteva u verziji 1.

- **Zahtev 1:** Za ovaj zahtev je dovoljno da implementiramo neki unapred zapisan skup reči. S obzirom da nam zahtev za biranjem podskupa reči iz celog skupa nije deo verzije 1, moraćemo da fiksiramo skup od tačno onoliko reči koliko ćemo prikazivati igraču na početku igre. Neka taj broj bude 10. Kako bismo imali neku vrstu raznolikosti ovog skupa reči na početku svake igre, možemo fiksirati nekoliko ovakvih skupova, na primer, tri.
- **Zahtev 4:** Odlučujemo se da svaka reč nosi onoliko poena koliko je njena dužina, pomnožena brojem 3.
- **Zahtev 5:** Jedan korak u igri podrazumeva ispisivanje reči i učitavanje reči od igrača. Za početak ćemo prikazivati reči jedne ispod druge na standardnom izlazu. Kako bismo razlikovali reči iz prethodnog koraka od reči u tekućem koraku igre, ispisivaćemo liniju nakon korisničkog unosa.
- **Zahtev 7:** Od igrača ćemo očekivati da unosi reči sa standardnog ulaza.
- **Zahtev 8:** S obzirom da u ovoj verziji aplikacije ne pamtimo vreme za koje je korisnik uneo reči, potrebno je da smislimo sistem bodovanja koji ne zavisi od vremena. Neka aplikacija dodeli za svaku tačno upisanu reč onoliko koliko ta reč nosi poena inicijalno, podeljeno sa brojem reči koje je do sada korisnik uneo.
- **Zahtev 9:** Ovaj zahtev možemo jednostavno implementirati brisanjem reči iz skupa, kada je korisnik ispravno unese.
- **Zahtev 13:** Na kraju igre sabiramo poene koje je korisnik osvojio i prikazujemo ih na standardni izlaz.

Kao bismo detaljnije razumeli deo sistema koji treba implementirati, možemo pretočiti date zahteve u *dijagram aktivnosti*, koji je dat na slici 1.6. Dijagram aktivnosti služi da grafički prikaže jedan konkretan poslovni proces. O izgradnji dijagrama aktivnosti više detalja biće u sekciji 1.3. Dijagram aktivnosti sa slike 1.6 prikazuje poslovni proces „Igranje jedne partije igre" koji odgovara istoimenom slučaju upotrebe. Primetimo da je opseg ovog procesa saglasan sa kontekstom koji smo definisali obimom odabranih zahteva. Očekivano je da će ovaj proces evoluirati zajedno sa verzijama našeg softvera, s obzirom da koristimo faznu metodologiju razvoja. Da smo koristili metodologiju vodopada za

razvoj ove aplikacije, onda bi dijagram aktivnosti sadržao elemente svih zahteva koje smo identifikovali na početku faze analize. Zanimljivo je analizirati kako iste tehnike u istim fazama razvoja utiču na proces razvoja u različitim metodologijama.



Slika 1.6: UML dijagram aktivnosti za poslovni proces „Igranje jedne partije igre".

Projektovanje verzije 1 S obzirom da još uvek nismo diskutovali o naprednijim tehnikama projektovanja softvera, oslonićemo se na intuiciju i prethodno iskustvo kako bismo se pomerili od imperativnog načina razmišljanja. Uistinu, posmatranjem dijagrama aktivnosti sa slike 1.6, lako bismo se dosetili da za svaku aktivnost u dijagramu implementiramo po jednu funkciju. Međutim, možemo dosta bolje od toga postavljanjem samo jednog pitanja: Ko treba da sprovede određenu aktivnost? Odgovaranjem na ovo pitanje se polako odmičemo od imperativnog ka objektno-orijentisanom načinu razmišljanja. Razmotrimo odgovornost sprovođenja aktivnosti iz prikazanog dijagrama aktivnosti, počevši

od prve aktivnosti, „Inicijazuj skup reči", pa pomeranjem u pravcu strelica ka ostalim aktivnostima:

- Setimo se da smo u slučaju upotrebe „Igranje jedne partije igre" identifikovali jednog aktera „Igrač" koji na neki način učestvuje u sistemu. Međutim, sprovođenje aktivnosti „Inicijazuj skup reči" nikako ne zahteva postojanje igrača, te zbog toga odgovornost sigurno nije na njemu. Dakle, mogli bismo da uvedemo novi objekat u naš sistem – nazovimo taj objekat „Rečnik". Pažljivi čitaoci će se sigurno zapitati zbog čega nismo ovaj objekat postavili kao aktera u slučaju upotrebe. Odgovor na ovo pitanje je zbog toga što u slučaju upotrebe „Rečnik" je deo softvera igre, a ne neki korisnik ili drugi sistem koji koristi ili beneficira od softvera koji mi treba da implementiramo.

- Aktivnost „Dodeli inicijalne poene" je posebno interesantna. Na prvi pogled, s obzirom da će ona implementirati neku vrstu operacije nad rečima, deluje da bi trebalo da bude odgovornost objekta „Rečnik". Međutim, ako malo bolje porazmislimo, objekat „Rečnik" bi u tom slučaju trebalo da bude upoznat sa pravilima igre. Drugim rečima, on bi upravljao rečima, ali bi upravljao i pravilima igre. To bi već sada trebalo da nam deluje problematično[5]. Bolje je da odgovornost o bodovanju dodelimo novom objektu kojeg ćemo uvesti u sistem – „Bodovanje".

- Ponovo, mogli bismo aktivnosti „Prikaži reči" i „Zatraži reč od igrača" da dodelimo objektu „Rečnik", međutim, onda bi taj objekat određivao način na koji se podaci učitavaju, na primer, korišćenjem standardnog ulaza, grafičkog interfejsa ili na neki drugi način. Umesto toga, smislenije je kreirati novi objekat, „Podsistem za interakciju sa korisnikom" (skr. PIK), koji će koordinisati interakciju sa igračem.

- Naredna aktivnost je „Zapamti reč" i ovo je prva aktivnost koju izvršava akter „Igrač" u našem sistemu. Pošto je on nezavisan deo od sistema, i za njega ćemo kreirati novi objekat sa istim nazivom.

- ...

Istom procedurom nastavljamo dalje po dijagramu aktivnosti. Kada dođemo do kraja, trebalo bi da za svaku aktivnost imamo definisan objekat koji je od-

[5]Videćemo detaljnije zašto u poglavlju 3, kada budemo diskutovali o SOLID principima razvoja.

govoran za sprovođenje te aktivnosti. Tako dobijamo naredni spisak objekata i aktivnosti za koje su oni zaduženi:

- Igrač
  - Zapamti reč
  - Zapamti ostvarene poene za reč
  - Uvećaj broj pokušaja
  - Izračunaj konačni rezultat
- Rečnik
  - Inicijalizuj skup reči
  - Pronađi reč
  - Izbaci prepoznatu reč iz skupa
- Bodovanje
  - Dodeli inicijalne poene
  - Dohvati inicijalne poene za reč
  - Izračunaj poene za reč
- PIK
  - Prikaži reči
  - Zatraži reč od igrača
  - Prikaži konačni rezultat

Dobijeni spisak predstavlja osnovu za dizajn softvera – na osnovu ovog dizajna ćemo vršiti implementaciju u narednoj fazi. U poglavlju 3 ćemo se detaljnije osvrnuti na objektno-orijentisane tehnike projektovanja softvera.

Osvrnimo se sada na projektovanje specifikacije podataka:

- Reči je potrebno čuvati i na nivou „Rečnika", ali i na nivou „Bodovanja". Međutim, u slučaju drugog objekta, pored samih reči, mora da postoji informacija o bodovima. Zbog toga, reči u „Rečniku" možemo čuvati u strukturi podataka skup (radi bržeg pretraživanja u odnosu na vektor), a u „Bodovanju" u strukturi podataka mapa (radi bržog pronalaženja bodovanja).
- S obzirom da smo se opredelili da igru razvijamo kao aplikaciju komandne linije, reči ćemo učitavati putem standardnog ulaza, a prikazivaćemo ih putem standardnog izlaza.

Implementacija verzije 1 Za svaki objekat ćemo implementirati po jedan korisnički-definisani tip. U programskom jeziku C++, to je moguće uraditi implementiranjem *klase* ili *strukture*. Razlika je u tome što su sve članice (tj. atributi i metodi) u klasi podrazumevano *privatnog pristupa* (odnosno, dostupne su samo iz drugih metoda te klase), a u strukturi su podrazumevano *javnog pristupa* (odnosno, sve druge funkcije im mogu pristupati). S obzirom da ćemo se objektno-orijentisanom paradigmom baviti detaljnije u poglavlju 3, za potrebe razvoja ove aplikacije koristićemo strukture. Takođe, napomenimo da će svaka struktura biti implementirana u posebnoj datoteci *zaglavlja* (eng. *header*) istog naziva sa ekstenzijom `.hpp`. U glavnoj jedinici prevođenja (`main.cpp`) svaka od ovih datoteka zaglavlja će biti jednom uvezena pomoću pretprocesorske direktive `#include`. S obzirom da će neke datoteke zaglavlja učitavati druge datoteke zaglavlja, neophodno je da svaka datoteka zaglavlja sadrži tzv. *štitnik zaglavlja* (eng. *header guard*) – trojku pretprocesorskih direktiva `#ifndef`, `#define` i `#endif` koja sprečavaju da se jedna datoteka zaglavlja uveze više od jednom u istu jedinicu prevođenja. Time sprečavamo da se identifikatori iz datoteka zaglavlja definišu više puta u aplikaciji.

Započnimo prvo implementacijom igrača. Definišemo strukturu `Player` koja treba da implementira aktivnosti za koje je odgovoran igrač:

- Zapamti reč (`setCurrentWord`)
- Zapamti ostvarene poene za reč (`awardPoints`)
- Uvećaj broj pokušaja (`increaseTries`)
- Izračunaj konačni rezultat (`calculateTotalPoints`)

Za svaku od ovih aktivnosti, kreiraćemo po jedan metod u strukturi `Player` i ti metodi su navedeni u zagradama iznad. Za potrebe metoda `setCurrentWord`, potrebno je da struktura `Player` čuva informacije o tekućoj reči koju je igrač uneo. Dakle, potrebno je da definišemo jedan atribut ove strukture tipa `std::string`. Trenutna implementacija ove strukture izgleda:

```
#ifndef PLAYER_H
#define PLAYER_H

#include <string>

struct Player {
```

```
    std::string m_currentWord;

    void setCurrentWord(const std::string &word) {
        m_currentWord = word;
    }
};

#endif // PLAYER_H
```

S obzirom da će druge klase koristiti informaciju o učitanoj reči, korisno bi bilo da kreiramo metod koji će dohvatati atribut `m_currentWord`. Kako bismo izbegli kopiranje niski, možemo vratiti referencu na reč:

```
#ifndef PLAYER_H
#define PLAYER_H

#include <string>

struct Player {
    std::string m_currentWord;

    void setCurrentWord(const std::string &word) {
        m_currentWord = word;
    }

    const std::string &getCurrentWord() const {
        return m_currentWord;
    }
};

#endif // PLAYER_H
```

Primetimo dve stvari:

- Metod `getCurrentWord` iza potpisa, a pre tela, sadrži ključnu reč `const`. U programskom jeziku C++ je moguće definisani ne samo promenljive, već i metode kao konstantne. Pojednostavljeno rečeno, konstantni metodi su oni koji ne menjaju *stanje* objekta. Metod `getCurrentWord` se može smatrati konstantnim zato što on samo dohvata vrednost atributa `m_currentWord`, pri čemu ta operacija nikako neće promeniti vrednost

atributa `m_currentWord` (trenutno je ovo jedini atribut koji čini stanje objekta strukture `Player`). Informacija o tome da li je neki metod konstantan ili ne pomaže kompilatoru da uhvati neke greške u fazi prevođenja. Na primer, ako smo u kodu označili da je neki objekat konstantan, onda će nam kompilator signalizirati da pokušavamo da ga menjamo ukoliko smo nad takvim objektom pozvali nekonstantan metod. U tom trenutku onda treba razmisliti da li smo zaista želeli da promenimo objekat pozivom tog metoda (i u tom slučaju treba da izmenimo definiciju metoda da ne bude konstantan) ili očekujemo da se taj objekat zaista ne menja (i u tom slučaju treba osmisliti drugi način za postizanje funkcionalnosti koju smo želeli da postignemo pozivom tog metoda). U svakom slučaju, grešku smo dobili u fazi prevođenja, a ne u fazi pokretanja, što nam značajno olakšava razvoj.

- S obzirom da je metod `getCurrentWord` konstantan, kao i želimo da se dohvatanje vrednosti `m_currentWord` vrši po referenci kako bismo izbegli kopiranje, onda ta referenca koja se vraća kao povratna vrednost mora biti konstantna. Ovo ima smisla zbog toga što vraćanje nekonstantne reference ne garantuje da će vrednost na koju ta referenca referiše ostati nepromenjena. U suprotnom, označavanje metoda `getCurrentWord` kao konstantnog bi izgubilo svaki smisao.

Implementacija metoda `awardPoints` koji pamti ostvaren broj poena za tekuću reč koju je korisnik uneo zahteva postojanje neke strukture podataka koja će skladištiti poene. Neka to bude, na primer, vektor. Dakle, potrebno je da u strukturu `Player` dodamo novi atribut – vektor poena. Metod `awardPoints` će samo dodati novu vrednost u taj vektor. Ovaj metod menja stanje objekta nad kojim se poziva (menjajući stanje vektora `m_awardedPoints`), te zbog toga ne može biti definisan kao konstantan[6].

```
#ifndef PLAYER_H
#define PLAYER_H

#include <vector>

...
```

[6]Preciznije, metod `push_back` šablonske klase `std::vector<T>` je deklarisan kao nekonstantan, a konstantni metodi ne smeju pozivati nekonstantne metode.

```
struct Player {
    ...
    std::vector<double> m_awardedPoints;

    void awardPoints(double points) {
        m_awardedPoints.push_back(points);
    }
};

#endif // PLAYER_H
```

Za potrebe implementacije metoda `increaseTries` moramo čuvati broj pokušaja koji je korisnik imao. Dakle, definišemo još jedan atribut `m_totalTries` tipa `unsigned` strukture `Player`. Atributima možemo dodeliti početne vrednosti, kao što je prikazano u kodu ispod[7]. S obzirom da će druge klase čitati ovaj podatak, ima smisla definisati pomoćni konstatni metod za dohvatanje vrednosti ovog atributa, `getTotalTries`.

```
#ifndef PLAYER_H
#define PLAYER_H

...

struct Player {
    ...
    unsigned m_totalTries = 0u;

    void increaseTries() {
        m_totalTries++;
    }

    unsigned getTotalTries() const {
        return m_totalTries;
    }
};

#endif // PLAYER_H
```

[7]Postoje i drugi načini koji su više u duhu objektno-orijentisane paradigme i o njima ćemo govoriti u narednim poglavljima.

Poslednji metod `calculateTotalPoints` koji implementiramo u ovoj klasi služi za računanje ostvarenih poena. Ovo se jednostavno postiže sumiranjem svih vrednosti iz vektora poena. Možemo iskoristiti algoritam `std::accumulate` koji smo već videli.

```
#ifndef PLAYER_H
#define PLAYER_H

...

struct Player {
    ...

    double calculateTotalPoints() const {
        return std::accumulate(m_awardedPoints.cbegin(),
                               m_awardedPoints.cend(), 0.0);
    }
};

#endif // PLAYER_H
```

Cela struktura `Player` sada izgleda kao u nastavku:

```
#ifndef PLAYER_H
#define PLAYER_H

#include <string>
#include <vector>
#include <numeric>

struct Player {
    std::string m_currentWord;
    std::vector<double> m_awardedPoints;
    unsigned m_totalTries = 0u;

    void setCurrentWord(const std::string &word) {
        m_currentWord = word;
    }

    const std::string &getCurrentWord() const {
```

```
        return m_currentWord;
    }

    void awardPoints(double points) {
        m_awardedPoints.push_back(points);
    }

    void increaseTries() {
        m_totalTries++;
    }

    unsigned getTotalTries() const {
        return m_totalTries;
    }

    double calculateTotalPoints() const {
        return std::accumulate(m_awardedPoints.cbegin(),
                               m_awardedPoints.cend(), 0.0);
    }
};

#endif // PLAYER_H
```

Pređimo sada na implementaciju rečnika. Definišemo strukturu `Dictionary` koja ima jedan atribut `m_words` tipa `std::set<std::string>`. Ovaj atribut se koristi u svim metodima koje je potrebno da implementiramo. Kao i u prethodnom slučaju, pored aktivnosti za koje je odgovorna ova struktura, navodimo nazive metoda strukture u zagradama. Dodatno, navodimo kratak opis implementacije svakog metoda.

- Inicijalizuj skup reči (`initWords`): kreiraćemo tri skupa nasumičnih reči koje će predstavljati naše kandidate. Zatim ćemo generisati pseudoslučajan celi broj u intervalu $[0, 2]$ koji će definisati koji skup-kandidat će postati skup koji će predstavljati vrednost atributa `m_words`.
- Pronađi reč (`isWordPresent`): koristićemo metod `find` šablonske klase `std::set<T>` kako bismo pokušali da pronađemo iterator na reč koja se prosleđuje ovom metodu. Ukoliko metod vrati validan iterator (tj. iterator koji ne pokazuje na kraj skupa), onda je reč pronađena. U suprotnom, reč ne postoji u skupu.

- Izbaci prepoznatu reč iz skupa (`removeWord`): koristićemo metod `erase` šablonske klase `std::set<T>` kako bismo izbacili iz skupa reč koja se prosleđuje ovom metodu.

U nastavku dajemo čitavu implementaciju strukture na osnovu datih opisa. Primetimo postojanje dva pomoćna metoda: `hasMoreWords` i `getWords`. Prvi služi za ispitivanje da li je skup reči prazan, a drugi za dohvatanje skupa reči.

```
#ifndef DICTIONARY_HPP
#define DICTIONARY_HPP

#include <set>
#include <string>
#include <random>

struct Dictionary {
    std::set<std::string> m_words;

    void initWords() {
        const std::set<std::string> candidateWords0 =
            {"brother", "female", "dog", "pull", "proper",
             "hall", "drop", "chase", "neck", "drive"};
        const std::set<std::string> candidateWords1 =
            {"name", "chance", "needle", "pupil", "brown",
             "narrow", "children", "really", "cheese", "century"};
        const std::set<std::string> candidateWords2 =
            {"mountain", "half", "protect", "nation", "children",
             "fat", "autumn", "public", "problem", "certain"};

        std::random_device rd;
        std::default_random_engine g(rd());
        const auto randomChoice = g() % 3;
        switch (randomChoice) {
        case 0:
            m_words = candidateWords0;
            break;
        case 1:
            m_words = candidateWords1;
            break;
```

```
        case 2:
            m_words = candidateWords2;
            break;
        }
    }

    bool isWordPresent(const std::string &word) const {
        return m_words.find(word) != m_words.cend();
    }

    void removeWord(const std::string &word) {
        m_words.erase(word);
    }

    bool hasMoreWords() const {
        return !m_words.empty();
    }

    const std::set<std::string> &getWords() const {
        return m_words;
    }
};

#endif // DICTIONARY_HPP
```

Pređimo sada na implementaciju bodovanja kroz strukturu `PointSystem`. Već smo rekli da ova struktura sadrži mapu koja preslikava reči (niske) u njihove bodove (brojeve u pokretnom zarezu). Aktivnosti za koje je ova struktura zadužena, zajedno sa nazivima metoda koji ih implementiraju i kratkim opisima su:

- Dodeli inicijalne poene (`initPoints`): metodu prosleđujemo konstantnu referencu na rečnik kako bismo dohvatili informacije o svim rečima. Zatim, iteriramo kroz skup reči u rečniku i unosimo informacije u mapu, zajedno sa inicijalnim bodovima (za šta smo rekli da koristimo vrednost trostruke dužine reči).

- Dohvati inicijalne poene za reč (`getInitialPoints`): dohvatamo vrednost bodova iz mape koja odgovara reči koja se prosleđuje ovom metodu.

- Izračunaj poene za reč (`calculatePointsForCurrentWord`): metodu prosleđujemo konstantnu referencu na igrača kako bismo dohvatiti trenutnu reč koju je igrač ispravno uneo, zajedno sa tekućim brojem pokušaja. U ovoj verziji implementacije, ovaj metod jednostavno deli ove dve vrednosti i vraća rezultat deljenja.

U nastavku dajemo celokupnu implementaciju ove strukture.

```
#ifndef POINTSYSTEM_HPP
#define POINTSYSTEM_HPP

#include <map>
#include <string>

#include "Dictionary.hpp"
#include "Player.hpp"

struct PointSystem {
    std::map<std::string, double> m_points;

    void initPoints(const Dictionary &dict) {
        for (const auto &word : dict.getWords()) {
            m_points.emplace(word, 3.0 * word.size());
        }
    }

    double getInitialPoints(const std::string& word) const {
        return m_points.at(word);
    }

    double calculatePointsForCurrentWord(const Player& player) {
        return getInitialPoints(player.getCurrentWord()) /
                player.getTotalTries();
    }
};

#endif // POINTSYSTEM_HPP
```

Konačno, poslednja struktura koju implementiramo jeste `SUI`. Ova struktura sadrži nekoliko konstantnih atributa koje ćemo koristiti u metodima za lepše

ispisivanje reči na standardnom izlazu. Pored ovih atributa, struktura implementira naredne aktivnosti:

- Prikaži reči (`displayWords`): prvo „čistimo" standardni izlaz od prethodnog unosa prikazivanjem onoliko novih redova kolika je visina prozora konzole u karakterima (pretpostavljamo da je taj broj `60`). Zatim, prikazujemo reči iz rečnika.
- Zatraži reč od igrača (`readWordFromPlayer`): sa standardnog ulaza čitamo jednu nisku i smeštamo je kao podatak u objektu strukture `Player` koji se prosleđuje ovom metodu po referenci.
- Prikaži konačni rezultat (`displayFinalResult`): na standardnom izlazu ispisujemo poruku, zajedno sa brojem poena koji se prosleđuje ovom metodu.

U nastavku dajemo celokupnu implementaciju ove strukture.

```
#ifndef SUI_HPP
#define SUI_HPP

#include <iostream>
#include <string>
#include <iterator>
#include <random>

#include "Dictionary.hpp"
#include "Player.hpp"

struct SUI {
    const unsigned WINDOW_WIDTH = 80u;
    const unsigned WINDOW_HEIGHT = 60u;
    const std::string BAR = std::string(WINDOW_WIDTH, '=');
    const std::string SCROLL_SPACE = std::string(WINDOW_HEIGHT, '\n');

    void displayWords(const Dictionary &dict) const {
        std::cout << SCROLL_SPACE << std::endl;
        std::cout << BAR << std::endl;
        for (const auto &word : dict.getWords()) {
            std::cout << word << std::endl;
```

```
        }
        std::cout << BAR << std::endl;
    }

    void readWordFromPlayer(Player& player) const {
        std::string word;
        std::cin >> word;
        player.setCurrentWord(word);
    }

    void displayFinalResult(double points) const {
        std::cout << "Osvojeno poena: " << points << std::endl;
    }
};

#endif // SUI_HPP
```

Sada kada su sve aktivnosti implementirane, sve što je preostalo jeste da ih povežemo u skladu sa dijagramom aktivnosti sa slike 1.6. Ovaj proces, koji smo nazvali „Igranje jedne partije igre", implementiraćemo kao funkciju `playOneGame`. Iako nema potrebe to raditi u ovom programu, s obzirom da je ovo jedina funkcija koja se poziva iz funkcije `main`, u slučaju da želimo da dodamo još neku funkcionalnost u budućnosti, biće lakše to uraditi ukoliko smo čitav proces odvojili u posebnu funkciju (ili metod klase). U kodu ispod možemo primetiti kako se instanciraju objekti korisnički-definisanih tipova na stek okviru funkcije.

```
#include "Dictionary.hpp"
#include "PointSystem.hpp"
#include "SUI.hpp"
#include "Player.hpp"

void playOneGame() {
    Dictionary dict;
    dict.initWords();

    PointSystem pointSystem;
    pointSystem.initPoints(dict);

    const SUI sui;
```

```
    Player player;

    while (dict.hasMoreWords()) {
        sui.displayWords(dict);
        sui.readWordFromPlayer(player);
        player.increaseTries();

        if (dict.isWordPresent(player.getCurrentWord())) {
            const auto points =
                pointSystem.calculatePointsForCurrentWord(player);
            player.awardPoints(points);
            dict.removeWord(player.getCurrentWord());
        }
    }

    sui.displayFinalResult(player.calculateTotalPoints());
}

int main() {
    playOneGame();

    return 0;
}
```

Priprema za verziju 2 Pretpostavimo da smo nakon određenog perioda dobili naredne utiske o verziji 1 naše aplikacije:

- Igračima nije jasan sistem bodovanja. Potrebno je prikazati detalje ocenjivanja na kraju rada programa. Ovaj utisak pretvaramo u novi zahtev pod brojem 14.
- Igrači su primetili da se reči uvek prikazuju uređene leksikografski rastuće. Igra bi bila zanimljivija kada bi se poredak reči menjao u svakom koraku. Ovaj utisak pretvaramo u novi zahtev pod brojem 15.

Dodatno, tokom faze implementacije smo primetili da nekoliko metoda iz različitih klasa koristi generator pseudoslučajnih brojeva tako što svaki put inicijalizuju neophodne objekte pri pozivu tih metoda. Možda ne bi bilo loše da logiku koja se bavi pseudoslučajnim brojevima izdvojemo u poseban objekat. Ovaj utisak pretvaramo u novi zahtev pod brojem 16.

Novi spisak zahteva koji postavljamo pred fazu analize za verziju 2 je:

2. Učitavati podatke iz neke baze reči.
3. Odabrati podskup reči koji će se prikazivati u toku jedne igre.
6. Reči treba prikazivati na neki interesantan način.
10. Ako korisnik unese neispravnu reč, potrebno je definisati pravilo za kažnjavanje.
11. Aplikacija prikazuje trenutno osvojen broj poena tokom igre.
12. Potrebno je pratiti vreme kroz igru.
14. Potrebno je prikazati detalje ocenjivanja na kraju rada programa.
15. U svakom koraku prikazati preostale reči u neodređenom redosledu.
16. Izdvojiti generator pseudoslučajnih brojeva u poseban objekat.

Skup funkcionalnosti koje biramo za verziju 2 je $\{3, 10, 12, 14, 15, 16\}$.

Analiza verzije 2

Analiza, projektovanje i implementacija verzije 2.

Analizirajmo svaki prepoznati zahtev u verziji 2.

- **Zahtev 3:**
- **Zahtev 10:**
- **Zahtev 12:**
- **Zahtev 14:**
- **Zahtev 15:**
- **Zahtev 16:**

Projektovanje verzije 2

Implementacija verzije 2

Implementirati zahtev 15 pomoću `std::unordered_map`.

Novi spisak zahteva koji postavljamo pred ulazak u fazu analize za verziju 2 je:

2. Učitavati podatke iz neke baze reči.
6. Reči treba prikazivati na neki interesantan način.
11. Aplikacija prikazuje trenutno osvojen broj poena tokom igre.

Priprema za verziju 3 U verziji 3 ćemo implementirati preostale zahteve.

Analiza verzije 3

Analiza, projektovanje i implementacija verzije 3.

Analizirajmo svaki prepoznati zahtev u verziji 3.

- **Zahtev 2:**
- **Zahtev 6:**
- **Zahtev 11:**

Projektovanje verzije 3

Implementacija verzije 3 Važno je napomenuti da smo u ovoj implementaciji tek zagrebali po površini objektno-orijentisanih tehnika, tako da opisana implementacija nije ni jedina niti najbolja. Ipak, s obzirom da je ideja bila da podstaknemo čitaoce da razmišljaju i o drugim tehnikama koje mogu koristiti prilikom projektovanja i implementacije softvera, za sada smo zadovoljni onime što smo implementirali.

1.3 Tehnike analize

1.3.1 Prepoznavanje zahteva

Mozganje

Analiza rešenja

Analiza uzroka

Analiza očekivanja

1.3.2 Dokumentovanje zahteva

Definicija zahteva

Korisničke priče

Dijagram klase-odgovornosti-kolaboracije (CRC)

1.3.3 Identifikacija poslovnih procesa. UML dijagram slučajeva upotrebe

Elementi dijagrama slučajeva upotrebe

Primer

1.3.4 Modeliranje poslovnih procesa. UML dijagram aktivnosti

Elementi dijagrama aktivnosti

Primer

1.3.5 Modeliranje ponašanja. UML dijagram sekvence

Elementi dijagrama sekvence

Primer

Beleške

Veliki deo literature koji se bavi diskusijom o metodologijama razvoja softvera ili zaobilazi predstavljanje tih metodologija kroz konkretne primere i bavi se isključivo njihovom teorijom ili opsežno demonstrira njihove elemente kroz kompleksne softverske sisteme čije implementacije se prožimaju kroz nekoliko poglavlja. Ideja ovog poglavlja jeste da postepeno uvede čitaoce, koji su do sada razvijali relativno jednostavne programe, u temu metodologija razvoja softvera kroz razvoj jednostavnih aplikacija.

Tokom životnog ciklusa razvoja softvera, primetili smo da se prirodno postavljaju neka pitanja, kao što su:

- Planiranje: Kako se prepoznaju i biraju projekti za razvoj? Kako se formiraju timovi? Kako se upravlja rizicima?
- Analiza: Šta tačno predstavlja korak prepoznavanja i definisanja zahteva? Kako se zahtevi zapisuju? Koje su tehnike za prikupljanje zahteva? Kako se vrši analiza zahteva? Kako se modeluje poslovna logika?
- Projektovanje: Kako se koristi predlog softvera iz faze analize za kreiranje arhitekture sistema? Koje vrste arhitekture sistema postoje? Kako dizajnirati softver? Kako dizajnirati specifikaciju podataka.
- Implementacija: Kako dodeljivati zadatke programerima? Koja je uloga dokumentacije? Koja je uloga testiranja? Kako se softver isporučuje kupcu?

Da bismo dobili odgovore na ova, ali i druga pitanja, potrebno je da razumemo same ciljeve svake faze, ali i da se upoznamo sa velikim brojem tehnika koje se koriste u tim fazama. Ove tehnike imaju za cilj da formalizuju odgovore na data pitanja u vidu konkretnih rezultata koji će se koristiti u drugim tehnikama, najčešće u kasnijim fazama razvoja.

S obzirom da je ovaj tekst namenjen pre svega programerima koji se upuštaju u teoriju razvoja softvera i njene praktične primene, na fazu planiranja nismo stavljali veliki akcenat. Kada je faza analize u pitanju, u sekciji 1.3 smo opisali nekoliko jednostavnih tehnika koje se oslanjaju na analizu zahteva u softveru. U nekim od narednih poglavlja biće nešto detaljnije predstavljeni elementi koji imaju značajnije uloge u fazama projektovanja i implementacija softvera.

Glava 2

Upravljanje dinamičkim resursima

Pod dinamičkim resursima smatramo sve one objekte za koje je programer zadužen da upravlja ispravno. Specijalno, važan fokus ćemo staviti na životni vek, odnosno, govorićemo o važnosti upravljanja konstrukcijom i destrukcijom (uništavanjem) takvih objekata.

Tako, na primer, u programskom jeziku C++, objekti koji su konstruisani na stek memoriji biće automatski uništeni kada izvršavanje programa izađe iz opsega u kojem su ti objekti konstruisani. Sa druge strane, objekti koji su konstruisani na hip memoriji moraju se „ručno" uništiti pozivanjem odgovarajućih naredbi programskog jezika. Drugim rečima, ukoliko takve objekte ne uništavamo kada nam nisu više potrebni, oni će nepotrebno zauzimati prostor u memoriji i zagušavati rad i aplikacije u kojoj su kreirani, ali i drugih aplikacija koje se izvršavaju u isto vreme. U ovakvoj situaciji, programer nije ispravno upravljao objektima na hip memoriji onako kako je trebalo, pa su ti objekti dinamički resursi.

Prilikom završavanja rada programa, većina popularnih operativnih sistema automatski oslobađa dinamičke resurse koje je taj program koristio, kao što su dinamički objekti na hipu ili datoteke. Zbog toga bi neko mogao postaviti pitanje zašto onda da uopšte vodimo računa o upravljanju dinamičkim resursima ako će oni u nekom trenutku svakako biti oslobođeni kada se program završi. Postoji nekoliko problema sa ovakvim zaključkom:

- Većina kompleksnijih softverskih sistema imaju za cilj da rade na duže

staze, što može podrazumevati bilo koji interval od nekoliko sati do nekoliko godina. Na takvom, dugoročnom izvršavanju sistema, svaki problem, ma koliko inicijalno beznačajan bio, brzo će eskalirati u veliki problem.

- Čak i da njihovo izvršavanje ne traje dugo, postoje aplikacije koje generišu veliki broj dinamičkih objekata. Primer ovakve aplikacije su video igre. Ukoliko se veliki broj objekata ne oslobađa kada više nisu potrebni aplikaciji, onda će se radna memorija vrlo brzo prepuniti i operativni sistem će morati da takvu aplikaciju nasilno prekine. Zamislite da se ovako ponašanje desi Vama u sred odlučujuće bitke protiv neprijatelja u video igri i verujemo da će Vam biti jasan problem.

- Čak i da ne generišu veliki broj dinamičkih objekata, postoje aplikacije koje mogu da drže vlasništvo nad dinamičkim objektima tako da im druge aplikacije ne mogu pristupati. Primer ovakve aplikacije jeste aplikacija koja radi sa nekom bazom podataka u konkurentnom okruženju. Ako sistem za upravljanje bazom podataka dodeli (vrlo restriktivan) katanac takvoj aplikaciji nad nekom tabelom, druge aplikacije neće moći da rade sa istom tom tabelom sve dok je ova ne oslobodi. Ako aplikacija drži dobijeni katanac duže nego što joj je zaista neophodno, usporava se rad čitavog sistema.

Zbog toga, ovo poglavlje služi da uvede programera u koncept dinamičkih objekata i predstavlja neke tehnike za njihovo upravljanje.

2.1 Dinamički objekti

Dinamički objekti su objekti struktura ili klasa koji se nalaze na hip memoriji. Konstrukciju i destrukciju takvih dinamičkih resursa mora eksplicitno da zatraži od operativnog sistema, a takođe, u obavezi je da ih eksplicitno uništi nakon što mu više nisu neophodni. U narednom kodu definišemo korisnički-definisan tip `point` koji se sastoji od dve `double` vrednosti i predstavlja tačku u realnoj ravni. U `main` funkciji se vrši konstrukcija dinamičkog objekta pozivom operatora `new`.

```
struct point {
    point(double x, double y) {
        m_x = x;
        m_y = y;
```

```
    }

    double m_x;
    double m_y;
};

int main() {
    point *p = new point(3.5, -1.2345);

    return 0;
}
```

Povratna vrednost operatora `new` je pokazivač na jedan dinamički objekat tipa `point`, tj. `point *` koji čuvamo u promenljivoj `p`. Možemo koristiti i automatsko zaključivanje tipova koristeći ključnu reč `auto`. Primetimo da će kompilator zaključiti da je tip pokazivač, pa nema potrebe za navođenjem `*` nakon `auto`.

```
int main() {
    auto p = new point(3.5, -1.2345);

    return 0;
}
```

U svakom slučaju, ovaj dinamički objekat živi sve dok ga programer ne oslobodi ručno, što se ne dešava nikada u ovom primeru, ili dok program ne završi sa radom. Oslobađanje memorije koju zauzima neki dinamički objekat se izvršava pozivanjem operatora `delete` kojem se prosleđuje pokazivač na taj dinamički objekat.

```
int main() {
    auto p = new point(3.5, -1.2345);
    delete p;

    return 0;
}
```

S obzirom da dinamički objekti žive i nakon opsega u kojem su definisani, moramo da vodimo računa da zabeležimo informaciju o tome da su to dinamički

objekti, kako bismo mogli da ih oslobodimo u nekom trenutku. U narednom fragmentu koda, promenljiva `p` je definisana lokalno za funkciju `create_new_point` i ona biva automatski uništena nakon uništavanja stek okvira za tu funkciju, ali dinamički objekat koji je konstruisan i dalje živi. U ovoj situaciji, izgubili smo informaciju o adresi tog dinamičkog objekta i niko više ne može da ga oslobodi do kraja izvršavanja programa. Ova pojava se naziva curenje memorije.

```
void create_new_point(double x, double y) {
    auto p = new point(x, y);
}

int main() {
    create_new_point(3.5, -1.2345);

    return 0;
}
```

Ispravka ovog fragmenta koda bi se mogla izvršiti vraćanjem pokazivača na dinamički objekat, smeštanjem adrese dinamičkog objekta u neku globalnu promenljivu i sl.

```
point *create_new_point(double x, double y) {
    auto p = new point(x, y);
    return p;
}

int main() {
    auto p = create_new_point(3.5, -1.2345);
    delete p;

    return 0;
}
```

Primetimo način na koji se objekat strukture `point` konstruiše u svim prethodnim fragmentima koda – navođenjem vrednosti koordinata prilikom konstrukcije objekta. Ovo se postiže definisanjem specijalnog metoda u okviru definicije strukture koji se naziva konstruktor.

```
struct point {
    point(double x, double y) {
```

```
        m_x = x;
        m_y = y;
    }

    double m_x;
    double m_y;
};
```

Konstruktor mora da ima isti identifikator kao i naziv strukture. Konstruktor strukture `point` prihvata dva `double` argumenta i te vrednosti dodeljuje atributima tipa `double` koje ta struktura sadrži. Bez definicije konstruktora, konstrukcija objekta strukture i inicijalizacija njenih atributa bi se izvršavala odvojeno. U primeru ispod, kompilator u strukturu `point` umeće tzv. podrazumevani konstruktor koji nema argumenata i ima prazno telo i taj konstruktor se poziva prilikom konstrukcije objekta primenom operatora `new`.

```
struct point {
    double m_x;
    double m_y;
};

int main() {
    auto p = new point();
    p->m_x = 3.5;
    p->m_y = -1.2345;

    return 0;
}
```

Pored konstruktora, programski jezik C++ omogućava korisniku da definiše metod koji će biti pozvan prilikom destrukcije objekta. Takav metod se naziva destruktor i on se poziva bez obzira na način na koji se objekat uništava – automatski ili dinamički.

```
#include <iostream>

struct point {
    point(double x, double y) {
        m_x = x;
        m_y = y;
```

```
        std::cout
            << "Konstruisem tacku ("
            << m_x << "," << m_y << ")"
            << std::endl;
    }

    ~point() {
        std::cout
            << "Destruisem tacku ("
            << m_x << "," << m_y << ")"
            << std::endl;
    }

    double m_x;
    double m_y;
};
```

Destruktor mora da ima isti identifikator kao i naziv strukture, pri čemu mu prethodi karakter `~`. Sa definisanom strukturom `point` kao u fragmentu koda iznad, možemo zaista da potvrdimo da se u narednom fragmentu koda ne oslobađa memorija dinamičkog objekta ispravno.

```
int main() {
    auto p = new point(3.5, -1.2345);

    return 0;
}
```

Standardni izlaz

```
Konstruisem tacku (3.5, -1.2345)
```

Za razliku od toga, naredni fragment koda ilustruje ispravno oslobađanje dinamičkog objekta.

```
int main() {
    auto p = new point(3.5, -1.2345);
    delete p;
```

```
    return 0;
}
```

Standardni izlaz

```
Konstruisem tacku (3.5, -1.2345)
Destruisem tacku (3.5, -1.2345)
```

Naravno, isto važi i za dinamički objekat koji je konstruisan u različitom opsegu od onog iz kojeg se uništava.

```
point *create_new_point(double x, double y) {
    auto p = new point(x, y);
    return p;
}

int main() {
    auto p = create_new_point(3.5, -1.2345);
    delete p;

    return 0;
}
```

Standardni izlaz

```
Konstruisem tacku (3.5, -1.2345)
Destruisem tacku (3.5, -1.2345)
```

Jedna stvar o kojoj nismo diskutovali jeste uspešnost alokacije. Naime, postoje situacije u kojima operativni sistem nije u stanju da dodeli programu zatraženu memoriju. U takvim situacijama, operator `new` će vratiti vrednost `nullptr`, odnosno, nevalidan pokazivač[1]. Dakle, ukoliko je alokacija neuspešna, neophodno je implementirati deo koda koji će i takvu situaciju obraditi. U našim, jednostavnim aplikacijama, dovoljno je da, na primer, prikažemo korisniku poruku na standardni izlaz za greške i prekinemo dalje izvršavanje programa.

[1] Ono što smo smatrali za `NULL` u programskom jeziku C. Međutim, u programskom jeziku C++ ne korišćenje ovog makroa smatra lošom praksom.

```
int main() {
    auto p = new point(3.5, -1.2345);

    // Uslov je moguce napisati i kao: if(!p)
    if (nullptr != p) {
        std::cerr << "Alokacija tacke je neuspesna!" << std::endl;
        return EXIT_FAILURE;
    }

    delete p;

    return EXIT_SUCCESS;
}
```

U slučaju uspeha, dobijamo naredni izlaz.

Standardni izlaz

```
Konstruisem tacku (3.5, -1.2345)
Destruisem tacku (3.5, -1.2345)
```

U slučaju neuspeha, dobijamo naredni izlaz.

Standardni izlaz za greške

```
Alokacija tacke je neuspesna!
```

2.1.1 Osvrt na objekte sa automatskim životnim vekom

Dinamički objekti nisu uvek neophodni. Prvi savet pri radu sa dinamičkim objektima bi bio: Nemojte koristiti dinamičke objekte ako nema potrebe za njima. Drugim rečima, ako objekti mogu da stanu na stek okvire funkcija i njihovo korišćenje je dovoljno samo u okvirima tih stek okvira. Jedan objekat strukture `point` može se smestiti na stek okvir funkcije `main`. Nakon završavanja funkcije `main`, objekat `p` se automatski uništava.

```
int main() {
    point p(3.5, -1.2345);
```

```
    return 0;
}
```

Standardni izlaz

```
Konstruisem tacku (3.5, -1.2345)
Destruisem tacku (3.5, -1.2345)
```

Primetimo da sada promenljiva `p` nije tipa `point *` kao u prethodnim primerima, već je tipa `point`. Dakle, ova promenljiva se nalazi na steku i memorija koju ona zauzima na steku je (makar) veličine dve `double` vrednosti. Za razliku od toga, kada je promenljiva `p` bila tipa `point *`, tada je zauzimala onoliko koliko zauzimaju svi pokazivači za dati sistem za koji se program prevodio.

Inicijalizacija objekata na stek memoriji se može napisati na nekoliko načina[2]. U narednom fragmentu koda, svi objekti su kreirani na steku i za konstrukciju svih njih se poziva konstruktor sa dva argumenata tačno jednom.

```
int main() {
    point p1(3.5, -1.2345);
    point p2{3.5, -1.2345};
    point p3 = point(3.5, -1.2345);
    point p4 = point{3.5, -1.2345};
    auto p5 = point(3.5, -1.2345);
    auto p6 = point{3.5, -1.2345};

    return 0;
}
```

Specijalno, ako struktura ima definisan konstruktor bez argumenata (bez obzira da li smo ga eksplicitno napisali ili je kompilator umetnuo podrazumevani konstruktor), onda se objekat može konstruisati na stek memoriji i na sledeće načine. Očigledno, i za konstrukciju svakog od tih objekata se poziva konstruktor bez argumenata tačno jednom.

```
int main() {
    point p1;
```

[2]Posmatranjem jednostavne strukture kao što je `point`, deluje da nema smisla zašto je ovo dozvoljeno u programskom jeziku C++. Međutim, svaka od tehnika inicijalizacija ima svoju istoriju tokom višedecenijskog razvoja jezika.

```
    point p2();
    point p3{};
    point p4 = point();
    point p5 = point{};
    auto p6 = point();
    auto p7 = point{};

    return 0;
}
```

Naravno, isto kao i dinamičke objekte, i objekte na steku možemo konstruisati u različitim opsezima.

```
point create_new_point(double x, double y) {
    auto p(x, y);
    return p;
}

int main() {
    auto p = create_new_point(3.5, -1.2345);

    return 0;
}
```

Ovde moramo voditi računa o efikasnosti operacija. Naime, sve vrednosti se prilikom prosleđivanja funkcijama kao argumenti prenose po vrednosti, odnosno, pravi se njihova kopija. Na primer, ako se objekat koji je kreiran u jednoj funkciji prosleđuje drugoj funkciji, onda će on biti kopiran.

```
#include <iostream>

struct point {
    point(double x, double y) {
        m_x = x;
        m_y = y;
    }

    ~point() {
        std::cout
            << "Destruisem tacku ("
```

```
            << m_x << "," << m_y << ")"
            << std::endl;
    }

    double m_x;
    double m_y;
};

void print_point(point p_fun) {
    std::cout
        << '(' << p_fun.m_x << ',' << p_fun.m_y << ')'
        << std::endl;
}

int main() {
    point p(3.5, -1.2345);
    print_point(p);

    return 0;
}
```

Standardni izlaz

```
(3.5, -1.2345)
Destruisem tacku (3.5, -1.2345)
Destruisem tacku (3.5, -1.2345)
```

Kao što vidimo, postoje dva uništavanja objekta, pa samim tim i dve konstrukcije objekta[3]. Zaključujemo da se, prilikom prosleđivanja objekta `p` iz funkcije `main` u funkciju `print_point` konstruisala kopija objekta `p` u argument `p_fun`. Zbog toga, prvo ispisivanje iz destruktora koje vidimo dolazi prilikom poziva destruktora nad objektom `p_fun`, u trenutku završavanja funkcije `print_point`. Drugo ispisivanje u destruktoru se događa prilikom poziva destruktora nad objektom `p` u trenutku završavanja funkcije `main`.

Jednostavno rešenje ovog problema jeste upotreba referenci. Sve što je potrebno uraditi jeste promeniti deklaraciju funkcije `print_point` tako da je njen

[3]Ovde namerno preskačemo ispisivanje u konstruktoru, pošto bismo u tom slučaju videli samo jedno ispisivanje prilikom konstrukcije, što bi moglo da zbuni čitaoca. Kako se drugi objekat kreirao ako se nije pozvao konstruktor razjasnićemo u poglavlju 3.

argument referenca na tačku, tj. `point &`. Sada se kopira samo adresa objekta `p` umesto celog objekta. Zbog toga što se ne pravi kopija objekta `p`, očekujemo samo jedno ispisivanje iz destruktora na standardnom izlazu i zaista, upravo to se i dešava.

```
void print_point(point &p_fun) {
    std::cout
        << '(' << p_fun.m_x << ',' << p_fun.m_y << ')'
        << std::endl;
}
```

Standardni izlaz

```
(3.5, -1.2345)
Destruisem tacku (3.5, -1.2345)
```

Ipak, nekada je korišćenje dinamičkih objekata neophodno, tako da ćemo se u nastavku ovog poglavlja baviti dinamičkim objektima i tehnikama njihovog upravljanja.

Zadatak 4: Points

Napisati program komandne linije koji zahteva od korisnika da unese nenegativan ceo broj $n > 1.000.000$ putem standardnog ulaza, a zatim konstruiše n tačaka u pravougaoniku $\Pi = [-1.000, 1.000] \times [-1.000, 1.000] \cap \mathbf{R}$ koristeći generator pseudoslučajnih brojeva. Korisnik zatim unosi nenegativan ceo broj $k < n$. Na standardni izlaz ispisati nasumično generisanu tačku koja je najudaljenija od k-te nasumično generisane tačke.

Očigledno, ovaj program zahteva veliku količinu memorije u kojoj će smestiti sve tačke. Pod pretpostavkom da jedna tačka zauzima 16 bajtova (tj. dve veličine tipa `double`, od kojih je svaki po 8 bajtova), program zahteva oko 1,49 GB.

Hajde da prvo opišemo tok izvršavanja naše aplikacije:

- Aplikacija zahteva od korisnika da unese nenegativan ceo broj $n > 1.000.000$.
- Potrebno je generisati n tačaka u pravougaoniku Π. Već sada možemo da razmišljamo o podeli koda na funkcije koje implementiraju jednu i samo jednu celovitu funkcionalnost. Da bismo generisali n tačaka, potrebno je

za svaku tačku generišemo dve `double` vrednosti. To već odgovara jednoj funkciji. Druga funkcija će koristiti tu funkciju n puta, za svaku generisanu tačku.

- Aplikacija zahteva od korisnika da unese nenegativan ceo broj $k < n$.
- Linearnom pretragom treba da pronađemo tačku koja je najudaljenija od odabrane k-te tačke. Za ovu potrebu možemo napisati funkciju koja izračunava udaljenost između dve tačke i funkciju koja je koristi n puta, za svaku generisanu tačku.
- Na kraju, ispisujemo pronađenu najudaljeniju tačku.

Implementaciju postavljamo definicijom strukture `point` koja čuva koordinate tačke.

```
struct point {
    point(double x, double y) {
        m_x = x;
        m_y = y;
    }

    double m_x;
    double m_y;
};

int main() {


    return 0;
}
```

Učitavanje je jednostavno. Obratimo pažnju na odabir tipa promenljive koja čuva informaciju o odabranom broju n koji je korisnik uneo.

```
#include <iostream>
...

int main() {
    unsigned long long n;
    std::cout << "Unesite broj n veci od 1.000.000: ";
```

```
    std::cin >> n;

    return 0;
}
```

Sledeći korak je generisanje n nasumičnih tačaka. U tu svrhu, definišemo prvo funkciju za generisanje dve `double` vrednosti. U sistemskom zaglavlju `random`, definisane su klase od značaja za generisanje pseudoslučajnih brojeva. Neke od tih klasa definišu tipove raspodela (svi se nalaze u prostoru imena `std`):

- Uniformne raspodele: `uniform_int_distribution` za generisanje celih brojeva i `uniform_real_distribution` za generisanje brojeva u pokretnom zarezu;
- Normalne raspodele: `normal_distribution`, `chi_squared_distribution`, `student_t_distribution` i dr.
- Puasonove raspodele: `poisson_distribution`, `exponential_distribution`, `gamma_distribution` i dr.
- ...

Ove klase se koriste u kombinaciji sa tzv. mašinama za generisanje brojeva, koje implementiraju razne strategije biranja pseudoslučajnih brojeva iz datih raspodela. Najčešće se koriste predefinisani generatori, kao što su:

- `std::default_random_engine`, koji zavisi od implementacije kompilatora;
- `std::mt19937`, koji koristi Mersenne Twister algoritam za generisanje pseudoslučajnih brojeva;
- `minstd_rand`, koji koristi algoritam linearnog kongruenta za generisanje pseudoslučajnih brojeva;
- ...

Zrno za biranje pseudoslučajnih brojeva se postavlja pozivom metoda `seed` nad mašinom za generisanje brojeva i prosleđivanjem celog broja koji predstavlja zrno.

U fragmentu koda ispod koristimo `std::default_random_engine` nad uniformnom raspodelom brojeva u pokretnom zarezu. Za uniformnu raspodelu je

neophodno da navedemo opseg iz kojeg se biraju brojevi. S obzirom da funkcija „vraća" dve generisane `double` vrednosti, odlučili smo da se te vrednosti menjaju kroz argumente funkcije, tako da je neophodno da prosledimo reference kako bismo ih izmenili[4].

```
#include <random>
...

void generate_two_random_doubles(double &x, double &y) {
    static const auto lower = -1000.0;
    static const auto upper = 1000.0;

    static std::uniform_real_distribution<double> unif(lower, upper);
    static std::default_random_engine re;
    // re.seed(0);

    x = unif(re);
    y = unif(re);
}
```

Zatim, možemo napisati funkciju koja generiše n tačaka i vraća ih kao povratnu vrednost. Ono o čemu treba razmišljati jeste kako ćemo čuvati te tačke. Potrebna nam je struktura podataka koja ima efikasan indeksni pristup, kroz koju se može prolaziti sekvencijalno i čija je operacija dodavanja novog elementa efikasna. Struktura `std::vector<T>` koju smo već videli zadovoljava sve ove uslove.

Kako bismo ilustrovali rad sa pokazivačima, ovoga puta neće čuvati objekte u vektoru, već pokazivače. Razlog za ovo jeste u tome što će nam ovo nekada biti korisno, pogotovo kad radimo sa hijerarhijama klasa[5].

Ono o čemu moramo da vodimo računa jeste da vektor ne poziva operator `delete` nad elementima koje čuva. To ne bi ni imalo smisla da radi. Ako vektor čuva objekte, a ne pokazivače, onda će poziv operatora `delete` nad objektima proizvesti kompilatorsku grešku. Umesto toga, kada se vektor uništava, i njegov destruktor se pozove, on samo uklanja elemente koje je čuvao. Prilikom uklanjanja objekata, biće pozvan njihov destruktor, ali prilikom uklanjanja pokazivača, oni će samo biti uklonjeni sa njihove memorijske lokacije gde su bili smešteni,

[4] Naravno, mogli smo koristiti i pokazivače, ali nepisano pravilo je da se preferiraju reference umesto pokazivača kada god je to moguće. Mi ćemo usvojiti to pravilo u našim aplikacijama.

[5] Više o ovome ćemo reći u poglavlju 5.

ali dinamički objekti na koje pokazuju će i dalje postojati u hip memoriji. Zbog toga, ne smemo zaboraviti da oslobodimo memoriju za alocirane tačke kada ih program ne bude koristio.

Jedan način da implementiramo ovu funkciju jeste da vraćamo vektor kao povratnu vrednost.

```
#include <vector>
...

std::vector<const point *> generate_n_points(unsigned long long n) {
    std::vector<const point *> p_vec;

    ...

    return p_vec;
}

int main() {
    ...

    auto p_vec = generate_n_points(n);

    return 0;
}
```

Alternativno, možemo proslediti referencu ka vektoru u kojem ćemo čuvati pokazivače.

```
#include <vector>
...

void generate_n_points(std::vector<const point *> &p_vec,
                       unsigned long long n) {
    ...
}

int main() {
    ...

    std::vector<const point *> p_vec;
```

```
    generate_n_points(p_vec, n);

    return 0;
}
```

Mi ćemo koristiti drugi pristup, iako efektivno nema razlike među njima. Pažljivi čitalac bi se zapitao zašto je ovo ovako, odnosno, zar ne bi trebalo da je prvi slučaj neefikasniji zbog toga što će funkcija vratiti kopiju vektora prilikom povratka u `main`. Ovo je važna primedba i čitalac koji se ovo zapitao je na dobrom putu da postane C++ programer, s obzirom da o ovakvim stvarima treba uvek razmišljati. Razlog zašto se ovo ipak ne dešava je u konceptu koji se zove izbegavanje kopije. Naime, s obzirom da se vektor `p_vec`, koji se vraća kao povratna vrednost funkcije `generate_n_points` više ne koristi nigde, kompilator je u stanju da izvrši optimizaciju kojom će se osigurati da ne dođe do kopiranja u vektor `p_vec` u funkciji `main`. Ovo je primer tehnike optimizacije povratnih vrednosti. Radi ilustracije koncepta, dajemo naredni primer u kojem će se pozvati konstruktor strukture `T` samo jednom za potrebe konstrukcije promenljive `t`, umesto 4 puta, kako bi neko na prvi pogled pomislio, analiziranjem samo sintakse koda.

```
struct T { ... };

T f() { return T(); }

int main() {
    T t = T(T(f()));

    return 0;
}
```

Dopunimo implementaciju funkcije `generate_n_points`. Funkcija prvo priprema dve promenljive tipa `double` u koju će biti upisane nasumično generisane vrednosti. Zatim, na osnovu ovih vrednosti se generiše dinamički objekat tačke i pokazivač se čuva u vektor.

```
void generate_n_points(std::vector<const point *> &p_vec,
                       unsigned long long n) {
    double x, y;

    for (auto i = 0ull; i < n; i++) {
```

```
        generate_two_random_doubles(x, y);

        const auto p = new point(x, y);
        p_vec.push_back(p);
    }
}
```

Razmislimo sada o uspešnosti alokacije. Ukoliko nismo uspeli da alociramo memoriju za neku od n tačaka (na primer, ako je ponestalo memorije), trebalo bi da prekinemo izvršavanje, uz neku poruku korisniku. Ovo možemo uraditi tako što ćemo „vratiti" prazan vektor kroz argument `p_vec`. Naravno, ako je alokacija bila neuspešna za i-tu tačku, ne smemo da zaboravimo da obrišemo tačke $0, 1, \ldots, i-1$. U tu svrhu, implementirajmo prvo metod za brisanje tačaka iz vektora.

```
void delete_points(std::vector<const point *> &p_vec) {
    for (const auto &p : p_vec) {
        delete p;
    }
}
```

Sada dopunjujemo implementaciju funkcije `generate_n_points` tako da proverava alokaciju tačaka i vrši neophodno brisanje.

```
void generate_n_points(std::vector<const point *> &p_vec,
                       unsigned long long n) {
    double x, y;

    for (auto i = 0ull; i < n; i++) {
        generate_two_random_doubles(x, y);

        const auto p = new point(x, y);
        if (!p) {
            delete_points(p_vec);
            break;
        }

        p_vec.push_back(p);
    }
}
```

Naravno, neophodno je i da dopunimo implementaciju `main` funkcije koja proverava da li je alokacija prošla neuspešno. Ako jeste, ispisaćemo korisniku poruku na standardni izlaz za greške i prekinuti dalje izvršavanje programa.

```
int main() {
    ...

    std::vector<const point *> p_vec;
    generate_n_points(p_vec, n);

    if (p_vec.empty()) {
        std::cerr << "Alokacija je neuspesna!" << std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Sledeći korak je učitavanje broja k.

```
int main() {
    ...

    unsigned long long k;
    std::cout << "Unesite broj k manji od n: ";
    std::cin >> k;

    return EXIT_SUCCESS;
}
```

Zatim, potrebno je da pronađemo najudaljeniju tačku od k-te tačke. Prvo definišimo funkciju koja računa rastojanje između dve tačke. Razne matematičke funkcije su dostupne u sistemskom zaglavlju `cmath`, kao što su: funkcije za izračunavanje apsolutne vrednosti (`std::abs`, `std::fabs`, ...), eksponencijalne i logaritamske funkcije (`std::exp`, `std::log`, `std::log10`, `std::log2`, ...), funkcije stepenovanja i korenovanja (`std::pow`, `std::sqrt`, ...), trigonometrijske funkcije (`std::sin`, `std::cos`, ...) i dr.

```
#include <cmath>
...
```

```
double distance_between_points(const point *p1, const point *p2) {
    return std::sqrt(std::pow(p1->m_x - p2->m_x, 2) +
                     std::pow(p1->m_y + p2->m_y, 2));
}
```

Dalje, implementiramo funkciju koja linearnom pretragom pronalazi tačku koja je najudaljenija od k-te tačke korišćenjem funkcije `distance_between_points`.

```
const point *find_farthest_point_from_kth(
        std::vector<const point *> &p_vec,
        unsigned long long n,
        const point *kth_point) {
    auto max_dist = distance_between_points(p_vec[0], kth_point);
    auto idx_max_dist = 0ull;

    for (auto i = 1ull; i < n; ++i) {
        const auto curr_dist = distance_between_points(p_vec[i],
                                                       kth_point);
        if (curr_dist > max_dist) {
            max_dist = curr_dist;
            idx_max_dist = i;
        }
    }

    return p_vec[idx_max_dist];
}

int main() {
    ...

    const auto kth_point = p_vec[k];
    const auto p_found = find_farthest_point_from_kth(p_vec,
                                                      n,
                                                      kth_point);

    return EXIT_SUCCESS;
}
```

k-tu tačku smo izdvojili u `main` funkciji da bismo mogli da je ispišemo na

standardni izlaz zajedno sa pronađenom tačkom.

```
int main() {
    ...

    std::cout
        << "Tacka koja je najudaljenija od tacke ("
        << kth_point->m_x << "," << kth_point->m_y << ") je tacka ("
        << p_found->m_x << "," << p_found->m_y << ")." << std::endl;

    return EXIT_SUCCESS;
}
```

Konačno, ne smemo da zaboravimo da oslobodimo memoriju za svaku tačku iz vektora.

```
int main() {
    ...

    delete_points(p_vec);

    return EXIT_SUCCESS;
}
```

Kompletan kod je dat u rešenju 3.

Jedan potencijalni nedostatak prethodnog rešenja jeste taj što se svaka tačka alocira na nekoj lokaciji u memoriji, nezavisno od drugih tačaka. Ukoliko se generiše veliki broj tačaka (a to se definitivno radi), može doći do fragmentacije memorije. Nekada je bolje alocirati veliki broj objekata jedan za drugim u memoriji. U tu svrhu nam mogu pomoći nizovi. Naravno, s obzirom da i dalje govorimo o velikom broju tačaka, ideju o alociranju niza tačaka na stek okviru odmah otpisujemo. Dakle, potrebno je da koristimo niz dinamičkih objekata.

Nizovi dinamičkih objekata se u programskom jeziku C++ konstruišu i brišu korišćenjem operatora `new[]` i `delete[]`, redom. Operator `new[]` očekuje da mu se prosledi broj dinamičkih objekata i on traži od operativnog sistema da mu alocira neprekidan segment podataka u memoriji.

```
struct T {
    T() {}

    ...
};

int main() {
    int *p1 = new int[10];
    delete[] p1;

    size_t number_of_ts = 1000;
    auto p2 = new T[number_of_ts];
    delete[] p2;

    return 0;
}
```

Primetimo da je za alociranje niza dinamičkih objekata struktura neophodno da ta struktura ima konstruktor bez argumenata, bilo da smo ga eksplicitno naveli ili da je kompilator umetnuo podrazumevani konstruktor.

Napomenimo još i da se provera uspešnosti alokacije radi na isti način kao u slučaju alokacije jednog dinamičkog objekta – proverom jednakosti sa `nullptr` vrednosti.

Zadatak 5: Array of points

Implementirati rešenje zadatka 4 korišćenjem nizova dinamičkih objekata.

S obzirom da se zahtevi zadatka nisu promenili, prikazaćemo samo ključne izmene.

Za početak, neophodno je da struktura `point` ima konstruktor bez argumenata, a s obzirom da nam konstruktor sa dva argumenta nije neophodan, možemo da ukloniti iz koda.

```
struct point {
    double m_x;
    double m_y;
};
```

Funkcija koja generiše n tačaka sada mora da alocira niz dinamičkih objekata. Njen potpis moramo promeniti da vraća alocirani niz tačaka. U slučaju

neuspešne alokacije, funkcija može da vrati `nullptr` kako bismo u `main` funkciji znali da je došlo do greške i izvršili odgovarajuću akciju.

```
point *generate_n_points(unsigned long long n) {
    point *p = new point[n];
    if (!p) {
        return nullptr;
    }

    ...

    return p;
}

int main() {
    ...

    const auto p_arr = generate_n_points(n);
    if (!p_arr) {
        std::cerr << "Alokacija je neuspesna!" << std::endl;
        return EXIT_FAILURE;
    }

    delete[] p_arr;

    return EXIT_SUCCESS;
}
```

S obzirom da nam pristupanje i-tom elementu niza dinamičkih objekata vraća sam taj objekat, umesto da baratamo pokazivačima, sada možemo koristiti reference. Zbog toga, prvo menjamo implementaciju funkcije `distance_between_points` da koristi reference umesto pokazivače.

```
double distance_between_points(const point &p1, const point &p2) {
    return std::sqrt(std::pow(p1.m_x - p2.m_x, 2) +
                     std::pow(p1.m_y + p2.m_y, 2));
}
```

Funkcija `find_farthest_point_from_kth` sada prihvata pokazivač na početak niza dinamičkih objekata. Primetimo da nam je ovde neophodno da prosledimo

i broj elemenata ovog niza, što nije bila neophodna informacija u implementaciji sa vektorom (jer broj elemenata vektora možemo dobiti pozivanjem metoda `size` ili korišćenjem kolekcijske `for` petlje).

Takođe, kako bismo izbegli nepotrebno kopiranje tačke, vraćamo referencu na tačku u nizu. Prikažimo prvo implementaciju, pa prodiskutujmo o ovome.

```
const point &find_farthest_point_from_kth(
        const point *p_arr,
        unsigned long long n,
        const point &kth_point) {
    auto max_dist = distance_between_points(p_arr[0], kth_point);
    auto idx_max_dist = 0ull;

    for (auto i = 1ull; i < n; ++i) {
        const auto curr_dist = distance_between_points(p_arr[i],
                                                       kth_point);
        if (curr_dist > max_dist) {
            max_dist = curr_dist;
            idx_max_dist = i;
        }
    }

    return p_arr[idx_max_dist];
}

int main() {
    ...

    const auto &kth_point = p_arr[k];
    const auto &p_found = find_farthest_point_from_kth(p_arr,
                                                       n,
                                                       kth_point);

    std::cout
        << "Tacka koja je najudaljenija od tacke ("
        << kth_point.m_x << "," << kth_point.m_y << ") je tacka ("
        << p_found.m_x << "," << p_found.m_y << ")." << std::endl;

    return EXIT_SUCCESS;
```

```
}
```

Na ovom mestu bi bilo korisno skrenuti pažnju na koncept vraćanja referenci umesto vraćanja vrednosti direktno. Da je povratna vrednost ove funkcije `const point`, došlo bi do kopiranja tačke `p_arr[idx_max_dist]` u promenljivu `p_found` u koju se smešta povratna vrednost funkcije `find_farthest_point_from_kth`. Očigledno, ovde kompilator ne sme da primeni optimizaciju povratne vrednosti zbog toga što se tačka `p_arr[idx_max_dist]` nalazi u nizu i ne može da „ukrade" njenu implementaciju za konstrukciju promenljive `p_found`. Zbog toga se ovde vrši kopiranje.

Međutim, kopiranje možemo izbeći ako vratimo referencu na objekat `p_arr[idx_max_dist]`, pri čemu promenljiva `p_found` isto mora biti deklarisana kao referenca, što vidimo iz fragmenta koda iznad. Svaka druga kombinacija rezultuje kopiranjem tačke:

- Ako navedemo povratnu vrednost funkcije kao `point` ili `const point`, onda kompilator ne može samostalno da odluči da vrati njenu referencu jer ne zna da li će se ova funkcija pozivati ili za potrebe kopiranja povratne vrednosti ili za dohvatanje reference (ili za oba slučaja upotrebe).
- Ključna reč `auto` nikad ne dedukuje referencu na tip. Mogli bismo da izvučemo sličan zaključak za ovo ponašanje kao iz prethodne tačke.

Zbog toga, ako želimo da vratimo referencu na neki objekat kao povratnu vrednost funkcije, moramo da eksplicitno deklarišemo tip povratne vrednosti funkcije kao referencu odgovarajućeg tipa, kao i da eksplicitno deklarišemo tip promenljive koja čuva povratnu vrednost funkcije kao referencu odgovarajućeg tipa.

Postoji još jedna napomena koju treba imati na umu ovde, a to je kada je moguće vratiti referencu na objekat, a kada je to „zabranjeno". Naime, neki objekti imaju privremeni životni vek i ukoliko pokušamo da dohvatimo referencu na takve objekte, proizvešćemo nedozvoljen pristup, što dovodi do nasilnog prekidanja programa. Jedna takva situacija je opisana narednim fragmentom koda.

```
struct T { int x = 0; };

T &f() {
    T t;
    return t;
```

```
}

int main() {
    T &t = f();
    t.x = 1;

    return 0;
}
```

Naime, ovaj jednostavni program proizvodi nasilno prekidanje u naredbi `t.x = 1;`. Objasnimo zašto se ovo dešava. U funkciji `f` se kreira objekat `t` strukture `T` koji je lokalni objekat za tu funkciju. To znači da on ima automatski životni vek i da će biti uništen nakon što stek okvir funkcije `f` nestane nakon što ta funkcija završi svoje izvršavanje. Međutim, to dalje znači da, ukoliko pokušamo da vratimo referencu na ovaj objekat, mi ćemo zapravo vratiti referencu na objekat koji je uništen. Naredbom `t.x = 1;` mi pokušavamo da pristupimo objektu na adresi `t` koji ne postoji, što je nedozvoljen pristup.

Zašto onda ne dobijamo nedozvoljen pristup kada vraćamo referencu u funkciji `find_farthest_point_from_kth`? Razlog zbog toga je u tome što ova funkcija vraća referencu na objekat u nizu dinamičkih objekata `p_arr` čiji je životni vek duži od izvršavanja funkcije `find_farthest_point_from_kth`. Drugim rečima, dinamički objekti u nizu `p_arr` će sigurno postojati u memoriji i nakon izvršavanja funkcije `find_farthest_point_from_kth`.

Kompletan kod je dat u rešenju 4.

Ovaj zadatak je imao za cilj da ilustruje upotrebu niza dinamičkih objekata, raspoređenih neprekidno u memoriji, i da uporedi pristup u odnosu na korišćenje vektora pokazivača. Zapravo, vektor onako kako smo ga koristili na početku, u pozadini predstavlja niz dinamičkih objekata, sa dodatnim metodama za njegovo jednostavnije upravljanje. U većini slučajeva u praksi trebalo bi preferirati korišćenje struktura podataka iz standardne biblioteke nego „ručno" upravljanje memorijom.

Naravno, sa druge strane, nekada nije moguće alocirati veliku količinu neprekidne memorije, pa je u takvoj situaciji jedino moguće pokušati alociranje jednog po jednog dinamičkog objekta.

2.1.2 Deljenje dinamičkih objekata

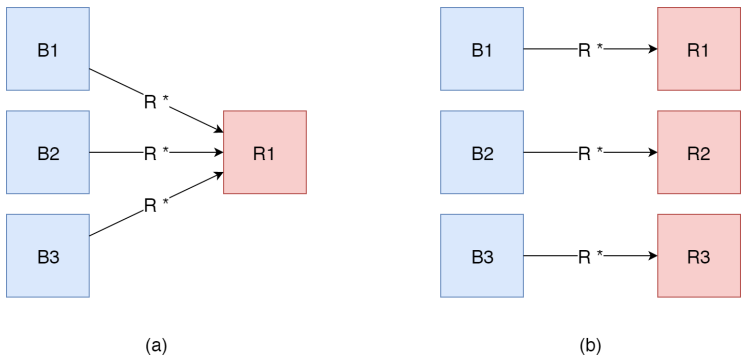
Sada kada smo se upoznali sa osnovnim tehnikama konstrukcije i uništavanja dinamičkih objekata, obratićemo pažnju na koncept deljenja dinamičkih objekata. Naime, često se dešava da nekoliko objekata treba da operiše nad nekim drugim dinamičkim objektima. Na primer, neka imamo objekte `b1`, `b2` i `b3` neke strukture `B` koji treba da upravljaju nad objektom `r1` neke strukture `R`.

Dva moguća modela upravljanja dinamičkog objekta `r1` u opisanoj situaciji su prikazana na slici 2.1. U slučaju da svi objekti strukture `B` treba da operišu nad istim objektom, pogodnije je implementirati situaciju na slici 2.1(a). U tom slučaju, svaki od objekata strukture `B` može čuvati pokazivač na isti objekat `r1` strukture `R`. Međutim, ovde postoji problem određivanja trenutka u kojem se ovi objekti korektno uništavaju. Naime, ako implementiramo da se prilikom uništavanja nekog od objekata `B` uništi i objekat `R` za koji on čuva pokazivač (na primer, neka se uništi objekat `b1`), onda će ostali `B` objekti (u ovom primeru, to su `b2` i `b3`) imati nevalidan pokazivač i može doći do nedozvoljenog pristupa memoriji, pa i do nasilnog prekidanja aplikacije. Sa druge strane, ako uništavanje `B` objekata nikako ne utiče na uništavanje `R` objekata, onda nakon uništavanja svih `B` objekata, izgubićemo sve pokazivače na `R` objekte i time uvodimo problem curenja memorije u aplikaciji.

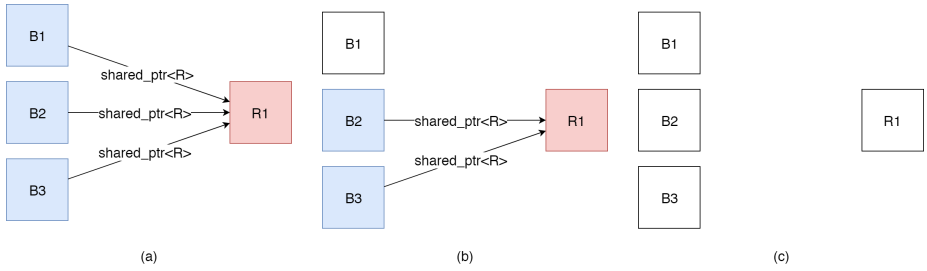
Kako bismo predupredili problem nekorektnom upravljanja dinamičkim objektima, možemo svakom `B` objektu dodeliti njegovu instancu `R` objekta nad kojom će operisati, što je ilustrovano na slici 2.1(b). Sada, ukoliko uništavanjem nekog `B` objekta uništimo i odgovarajući `R` objekat, onda sprečavamo pojavu curenja memorije, a rešavamo i problem nedozvoljenog pristupa jer će, uništavanjem jednog `B` objekta, svi ostali `B` objekti imati svoje kopije `R` objekta. Međutim, i ova situacija ima svoje nedostatke. Prvo, očigledno je da se povećava količina memorije koja je neophodna za skladištenje svih kopija `R` objekata. Preciznije, ukoliko je potrebno čuvati m objekata strukture `R`, pri čemu nad svakim `R` objektom operiše M_i objekata strukture `B`, onda se veličina neophodne memorije povećava sa m objekata na $\sum_{i=1}^{m} M_i > m$ objekata. Dodatno, potrebno je implementirati održavanje integriteta podataka – izmenom jedne kopije `R` objekta moraju se ažurirati vrednosti svih ostalih kopija tog objekta. Dakle, pored povećavanja memorijske zahtevnosti, dobijamo i povećanu količinu potrošnje resursa za procesiranje.

Svi opisani problemi se mogu rešiti korišćenjem specijalne vrste pokazivača koji se nazivaju deljeni pokazivači. Osnovna ideja deljenih pokazivača jeste da se broji koliko objekata (u našem primeru, `B` objekata) trenutno čuva deljeni pokazivač na željeni dinamički objekat (u našem primeru, `R` objekat). Onog

trenutka kada novi objekat čuva deljeni pokazivač na taj dinamički objekat, brojač se uvećava za 1. Onog trenutka kada se postojeći objekat koji čuva deljeni pokazivač na taj dinamički objekat uništi, brojač se smanjuje za 1. Ukoliko se brojač smanji na 0, to znači da ne postoji nijedan objekat koji čuva deljeni pokazivač, pa se i sam dinamički resurs uništava.



Slika 2.1: Situacija kada objekti `b1`, `b2` i `b3` istovremeno operišu nad objektom `r1`. Na slici (a) svi objekti rade nad istom instancom u memoriji, dok na slici (b) svaki objekat ima svoju instancu nad kojom operiše.



Slika 2.2: Situacija kada objekti `b1`, `b2` i `b3` istovremeno operišu nad objektom `r1` putem deljenog pokazivača. Na slici (a) svi `B` objekti čuvaju deljeni pokazivač ka jednoj instanci `R` objekta u memoriji. Na slici (b) je prikazana situacija kada je objekat `b1` uništen, zajedno sa njegovim deljenim pokazivačem. Na slici (c) je prikazana situacija kada su svi `B` objekti uništeni, kao i svi njihovi deljeni pokazivači, zajedno sa `R` objektom.

Slika 2.2 ilustruje upotrebu deljenih pokazivača sa tri `B` objekta i jednim `R` objektom. Svaki `B` objekat sadrži jedan deljeni pokazivač ka `R` objektu. Važno je napomenuti da svaki deljeni pokazivač čuva informaciju o tačno jednom `R` objektu u memoriji. Ovo je ilustrovano na slici 2.2(a). U slučaju da se objekat `b1` uništi, i njegov deljeni pokazivač se uništava. Međutim, `R` objekat neće biti uništen zato što ostali `B` objekti sadrže deljeni pokazivač (tj. brojač je u tom trenutku jednak 2), što je ilustrovano na slici 2.2(b). Tek kada sva tri `B` objekta budu uništena, neće postojati nijedan deljeni pokazivač ka `R` objektu, pa će onda i on biti uništen, kao na slici 2.2(c).

Deljeni pokazivač je u standardnoj biblioteci programskog jezika C++ predstavljen šablonskom klasom `std::shared_ptr<T>`, definisanom u sistemskom zaglavlju `memory`. Konstrukcija objekta ove klase se vrši šablonskom funkcijom `std::make_shared<T>`, koja prihvata proizvoljan broj argumenata koji se koriste za konstrukciju dinamičkog objekta o kojem će se taj deljeni pokazivač starati. Pogledajmo jedan jednostavan primer koji ilustruje korišćenje deljenih pokazivača.

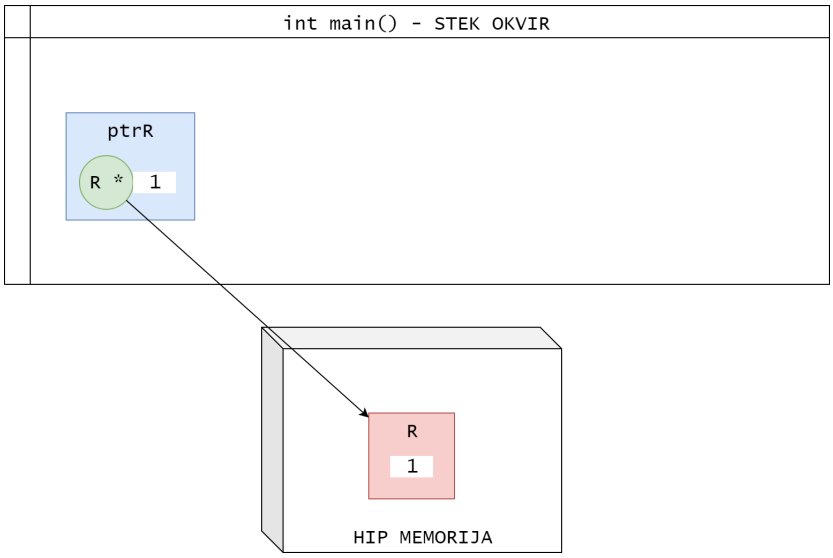
```
#include <memory>
#include <iostream>

struct R {
    int m_x;

    R(int x) {
        m_x = x;
    }

    ~R() {
        std::cout << "R objekat je unisten" << std::endl;
    }
};

struct B {
    std::shared_ptr<R> m_ptrR;
};

int main() {
    // Blok 1
    {
```

```
        auto ptrR = std::make_shared<R>(1); // brojac: 1

        B b1, b2;
        b1.m_ptrR = ptrR; // brojac: 2
        b2.m_ptrR = ptrR; // brojac: 3

        // Blok 2
        {
            B b3;

            b3.m_ptrR = ptrR; // brojac: 4

            std::cout
            << b1.m_ptrR->m_x << " "
            << b2.m_ptrR->m_x << " "
            << b3.m_ptrR->m_x << " " << std::endl;

            b1.m_ptrR->m_x = 5;

            std::cout
                << b1.m_ptrR->m_x << " "
                << b2.m_ptrR->m_x << " "
                << b3.m_ptrR->m_x << " " << std::endl;
        }
        // brojac: 3

        std::cout
            << b1.m_ptrR->m_x << " "
            << b2.m_ptrR->m_x << " " << std::endl;
    }
    // brojac: 0 -> unistava se R objekat

    std::cout << "Kraj programa" << std::endl;

    return 0;
}
```

Analizirajmo kod deo-po-deo. Prvo, definišemo strukturu `R` koja čuva jedan ceo broj za potrebe testiranja. Objekat ove strukture konstruišemo prosleđivanjem

celog broja konstruktoru, koji će zapamtiti tu vrednost. Dodatno, definišemo destruktor koji ispisuje poruku na standardni izlaz kako bismo uočili trenutak u kojem se izvršava uništavanje ovog objekta. Zatim, definišemo strukturu `B` koja čuva jedan deljeni pokazivač na objekat strukture `R`.

U funkciji `main()`, prvo konstruišemo jedan deljeni pokazivač ka objektu strukture `R` koji će biti konstruisan celobrojnom vrednošću 1. Taj deljeni pokazivač smeštamo u promenljivu `ptrR` i brojač deljenih pokazivača se uvećava sa 0 na 1. U pozadini je deljeni pokazivač zatražio memoriju na hip memoriji za smeštanje jednog `R` objekta i pokazivač `R *` na taj objekat je interno sačuvan u tom deljenom pokazivaču. Ovo je ilustrovano na slici 2.3.

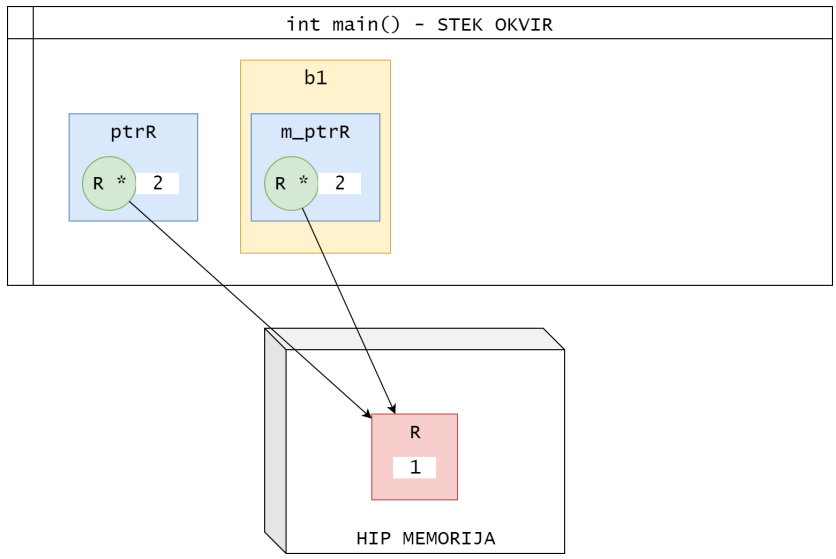


Slika 2.3: Grafički prikaz stek okvira funkcije `main` i stanja hip memorije prilikom konstrukcije deljenog pokazivača.

Zatim konstruišemo tri `B` objekta na stek okviru funkcije `main()`[6]. Nakon

[6]Mada bismo možda preferirali da i `B` objekti budu alocirani na hip memoriji, pogotovo što ih ima više od `R` objekata, ovde je ideja da čitaocu objasnimo kako funkcionišu deljeni pokazivači kroz čuvanje tačno jednog objekta na hipu i analiziranjem njegovog životnog veka u odnosu na životni vek i strukturu deljenih pokazivača koji „vode računa" o tom objektu. Smatramo da su posledice ovog izbora jasniji dijagrami koji se koriste u tekstu i bolje razumevanje gradiva od strane čitaoca. Ipak, u zadacima koji slede, koristićemo isključivo dinamičke

konstrukcije `B` objekata, svakom od njih postavljamo njegov interni deljeni pokazivač na maločas napravljen `ptrR`. Ono što se u ovom trenutku dešava jeste *kopiranje deljenog pokazivača* `ptrR`, *ali je vrlo važno napomenuti da ne dolazi do kopiranja samog dinamičkog objekta* `R` *koji se nalazi sakriven u tom deljenom pokazivaču.* Ovo je ilustrovano na slici 2.4.

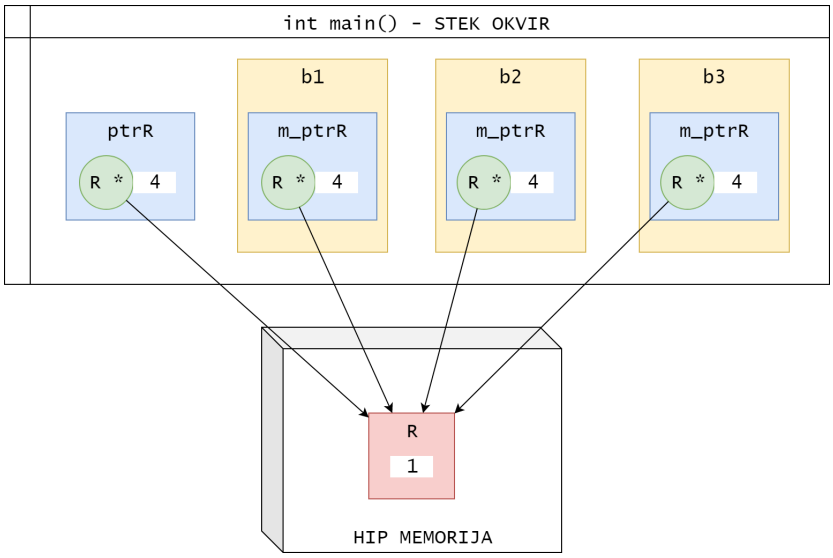


Slika 2.4: Grafički prikaz stek okvira funkcije `main` i stanja hip memorije nakon kopiranja deljenog pokazivača `ptrR` u objekat `b1`.

Kopiranjem deljenog pokazivača `ptrR` u deljenje pokazivače u objektima `b2` i `b3`, dolazimo do stanja programa u kojem sva tri `B` objekta imaju informaciju o istom, deljenom `R` objektu, kao i da je brojač deljenih objekata trenutno jednak 4, što je prikazano na slici 2.5.

Kako bismo ilustrovali da svi `B` objekti imaju informaciju o istom objektu, prvo ispisujemo celi broj koji se sadrži u `R` objektu preko odgovarajućih deljenih pokazivača iz objekata `b1`, `b2` i `b3`. Očekivan ispis se sastoji od tri jedinice, i zaista, dobijamo upravo taj ispis na standardnom izlazu.

objekte, pa ćemo tada proširiti ovu diskusiju.



Slika 2.5: Grafički prikaz stek okvira funkcije `main` i stanja hip memorije nakon konstruisanja i dodeljivanja deljenih pokazivača preostalim `B` objektima.

Standardni izlaz

```
1 1 1
```

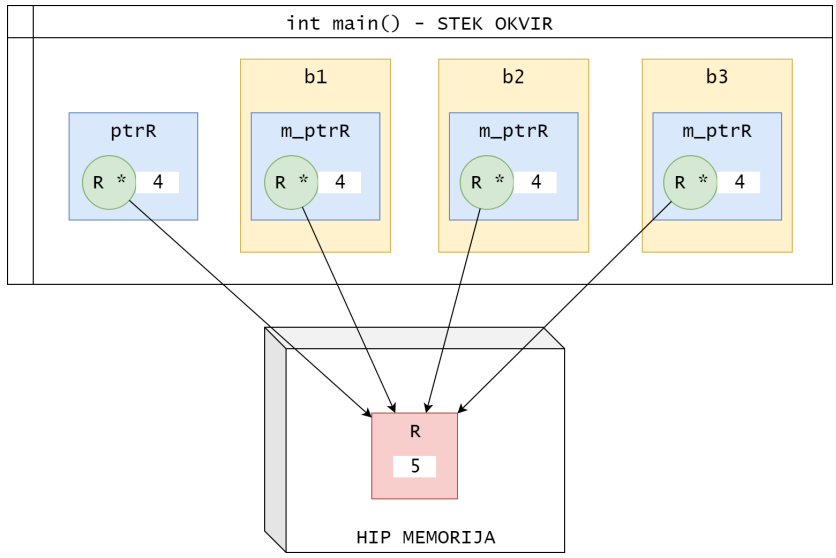
Dalje, pokušavamo sa izmenom celobrojne vrednosti preko deljenog pokazivača iz objekta `b1`. Nova celobrojna vrednost zapamćena u `R` objektu je 5. Nakon ponovnog ispisa, primećujemo da i objekti `b2` i `b3` vide izmenu koja je izvršena preko objekta `b1`. Dakle, `R` objekat je zaista deljen između ovih `B` objekata.

Standardni izlaz

```
5 5 5
```

Ova situacija je ilustrovana na slici 2.6.

Nakon isteka opsega definisanog „blokom 2" u kodu, životni vek objekta `b3` se završava i taj objekat se uništava, zajedno sa svojim članicama, tj. sa njegovim deljenim pokazivačem. U tom trenutku, brojač deljenih pokazivača se smanjuje na 3 u svim ostalim deljenim pokazivačima, kao na slici 2.7.



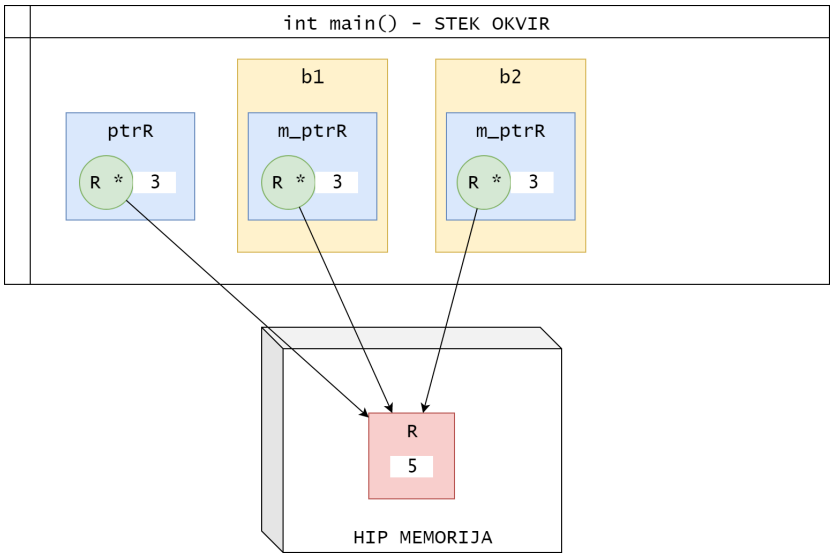
Slika 2.6: Grafički prikaz stek okvira funkcije `main` i stanja hip memorije nakon konstruisanja i dodeljivanja deljenih pokazivača preostalim `B` objektima.

Naravno, ostali `B` objekti i dalje imaju pristup deljenom `R` objektu, s obzirom da on nije uništen. Kako bismo se uverili u ovo, ispisujemo celobrojnu vrednost sačuvanu u deljenom `R` objektu preko deljenih pokazivača iz objekata `b1` i `b2`.

Standardni izlaz

```
5 5
```

Konačno, nakon isteka opsega definisanog „blokom 1" u kodu, sa stek okvira se uništavaju svi `B` objekti, kao i deljeni pokazivač `ptrR`. Svakom destrukcijom ovih objekata se brojač deljenih pokazivača smanjuje za 1. S obzirom da će taj brojač dobiti vrednost 0 nakon uništava poslednjeg od tih objekata, onda se i deljeni `R` objekat briše sa hip memorije, čime dobijamo stanje kao na slici 2.8. Primetimo iz ispisa na standardnom izlazu da se uništavanje izvršava pre završetka programa, što je i očekivano, s obzirom da ne postoji nijedan objekat koji sadrži deljeni pokazivač ka ovom dinamičkom objektu.



Slika 2.7: Grafički prikaz stek okvira funkcije `main` i stanja hip memorije nakon isteka „bloka 2".

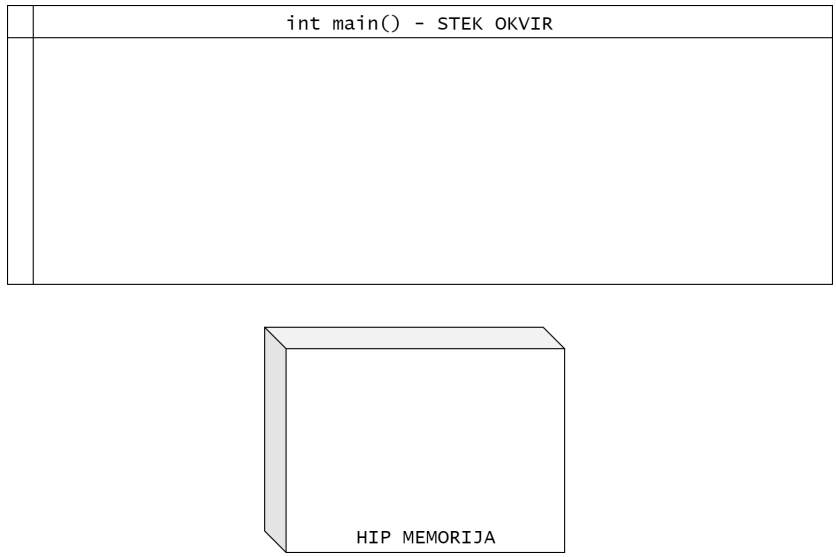
Standardni izlaz

```
R objekat je unisten
Kraj programa
```

Kao što vidimo, svi zahtevi za „deljenjem" objekata koje smo postavili pred sobom su ispunjeni, pri čemu smo izbegli uvođenje problema u radu sa dinamičkim resursima i preveliku složenost programa. Deljeni pokazivači imaju i svoju cenu. Naime, svaki deljeni pokazivač mora da održava informacije o stanju ostalih deljenih pokazivača poput brojača. Ipak, u većini aplikacija, ova cena je neprimetno mala u odnosu na korist koju ovi pokazivači donose.

Zadatak 6: Courses

Napisati program komandne linije koji omogućava korisniku da upisuje studente na kurseve. Kursevi su definisani svojim nazivom i pozitivnim brojem bodova, a studenti svojim indeksom i spiskom upisanih kurseva.



Slika 2.8: Grafički prikaz stek okvira funkcije `main` i stanja hip memorije nakon isteka „bloka 1".

Program prvo traži od korisnika da unese pozitivan broj k, a zatim učitava k kurseva (za svaki kurs se čita prvo naziv, pa zatim broj bodova). Zatim, program traži od korisnika da unese pozitivan broj n, a zatim učitava n indeksa studenata. Nakon učitavanja jednog indeksa studenta, program za svaki kurs postavlja pitanje korisniku da li želi da upiše studenta na kurs. Ako je odgovor pozitivan, zabeležiti informaciju o upisu. Nakon učitavanja svih studenata, program ispisuje za svakog studenta spisak kurseva koje je taj student upisao.

Pozabavimo se prvo definisanjem struktura koje se pominju u tekstu ovog zadatka. Započnimo od strukture `Course` koja predstavlja jedan kurs. Kao što vidimo, kursevi imaju svoj naziv, koji ćemo predstaviti klasom `std::string` i bodove, koje ćemo predstaviti tipom `unsigned`. Kako bismo lakše konstruisali `Course` objekte, implementiraćemo i konstruktor koji prihvata naziv kursa i broj bodova.

```
#include <string>
```

```
struct Course {
    std::string m_title;
    unsigned m_ects;

    Course(std::string title, unsigned ects) {
        m_title = title;
        m_ects = ects;
    }
};
```

Struktura `Student`, pored informacije o indeksu studenta, koju ćemo predstaviti klasom `std::string`, treba da sadrži i spisak kurseva. Primetite iz teksta zadatka da jedan kurs može da se nađe u spisku za više studenata. Drugim rečima, kursevi su deljeni objekti među objektima studenata. Zbog toga, bilo bi zgodno koristiti deljene pokazivače, tj. objekte klase `std::shared_ptr<Course>`. Naravno, pošto jedan student može biti upisan na više kurseva, moramo da čuvamo deljene pokazivače ka kursevima u neku kolekciju – odlučujemo se za vektor.

```
#include <vector>
#include <memory>

...

struct Student {
    std::string m_index;
    std::vector<std::shared_ptr<Course>> m_courses;

    Student(std::string index) {
        m_index = index;
    }
};
```

Primetite da u konstruktoru `Student` objekata prosleđujemo samo indeks studenta na osnovu kojeg se inicijalizuje atribut `m_index`, a da se vektor deljenih pokazivača na kurseve `m_courses` podrazumevano inicijalizuje na prazan vektor.

Sada prelazimo na implementaciju zahteva u funkciji `main()`. Prvo dajmo apstraktan opis toka izvršavanja programa:

- Učitavanje kurseva u neku kolekciju podataka, na primer, vektor.
- Učitavanje studenata u neku kolekciju podataka, na primer, vektor.
  - Za svakog studenta, ispisujemo spisak dostupnih kurseva i upisujemo studenta na odgovarajuće kurseve.
- Izlistavanje podataka o studentima.

Prvo učitavamo broj kurseva k sa standardnog ulaza. Zatim, pripremamo vektor u koji ćemo smeštati deljene pokazivače na kurseve koje učitavamo. Za svaki kurs $1, 2, \ldots, k$ učitavamo od korisnika naziv i broj bodova sa standardnog ulaza i konstruišemo jedan deljeni pokazivač, koji smeštamo u vektor.

```
#include <iostream>

...

std::cout << "Unesite broj kurseva:" << std::endl;
unsigned k;
std::cin >> k;

std::vector<std::shared_ptr<Course>> courses;

for (auto i = 0u; i < k; ++i) {
    std::cout << "Unesite naziv kursa: " << std::endl;
    std::string title;
    std::cin >> title;

    std::cout << "Unesite broj bodova: " << std::endl;
    unsigned ects;
    std::cin >> ects;

    auto course = std::make_shared<Course>(title, ects);
    courses.push_back(course);
}
```

U tekstu zadatka nije eksplicitno specifikovano kako se `Student` objekti skladište. Međutim, pošto znamo da upravljamo dinamičkom memorijom bezbedno korišćenjem deljenih pokazivača, možemo i `Student` objekte čuvati na hip memoriji.

Sa standardnog ulaza prvo učitavamo broj studenata n, a zatim, za svakog studenta $1, 2, \ldots, n$ učitavamo indeks, konstruišemo deljeni pokazivač, upisujemo studenta na kurseve i dodajemo ga u vektor deljenih pokazivača na studente.

```
std::cout << "Unesite broj studenata:" << std::endl;
unsigned n;
std::cin >> n;

std::vector<std::shared_ptr<Student>> students;

for (auto i = 0u; i < n; ++i) {
    std::cout << "Unesite indeks studenta: " << std::endl;
    std::string index;
    std::cin >> index;

    auto student = std::make_shared<Student>(index);

    // Upisivanje studenata na kurseve
    ...

    students.push_back(student);
}
```

U ovom delu koda ostalo je da implementiramo upisivanje studenata na kurseve, tj. popunjavanje spiska upisanih kurseva za svakog studenta. Očigledno, neophodno je da iteriramo kroz prethodno popunjeni vektor `courses`. Kostur ovog dela koda bi mogao da izgleda kao u nastavku:

```
for (const auto course : courses) {
    ...
}
```

Ono što bi trebalo da primetimo je da se u svakoj iteraciji kopira jedan po jedan element iz vektora `courses` u promenljivu `course`. Međutim, ovoga puta ovo nije skupa operacija, zato što se *ne kopira ceo objekat* `Course` na koji taj deljeni pokazivač pokazuje, već se kopira jedan `std::shared_ptr<Course>`, pri čemu se uvećava brojač deljenih pokazivača koji pokazuju na tekući objekat `Course` za jedan. Na kraju tekuće iteracije, kopirani deljeni pokazivač se uništava i brojač deljenih pokazivača se smanjuje za jedan.

Alternativni način, koji je neznatno brži, jeste korišćenje referenci:

```
for (const auto &course : courses) {
    ...
}
```

U ovom slučaju, u svakoj iteraciji promenljiva `course` predstavlja referencu na tekući element iz vektora `courses`. Važno je obratiti pažnju da u ovom slučaju ne dolazi do kopiranja deljenih pokazivača, pa samim tim ni uvećanja brojača, odnosno, smanjenja brojača na kraju iteracije. U ovom slučaju, svugda gde budemo koristili `course`, operišemo direktno nad elementom iz vektora, umesto nad kopiranim deljenim pokazivačem.

Nama u ovom zadatku nije važno koji ćemo pristup koristiti, ali mnogo važnije je bilo objasniti razliku između ova dva pristupa. Na kraju se odlučujemo za rad sa referencama.

U svakoj iteraciji petlje, na standardni izlaz ispisujemo naziv i broj bodova tekućeg kursa iz iteracije i pitamo korisnika da li želi da upiše studenta na tekući kurs. Ukoliko korisnik ne odgovori pozitivno, prelazimo na sledeću iteraciju. Inače, u vektor deljenih pokazivača za datog studenta dodajemo deljeni pokazivač na tekući kurs. Bilo da smo koristili pristup zasnovan na referencama ili ne, dodavanjem deljenog pokazivača u vektor se pravi kopija tog deljenog pokazivača i uvećava se brojač deljenih pokazivača na tekući kurs za jedan.

```
for (const auto &course : courses) {
    std::cout
        << "Da li zelite da upisete tekuceg studenta na kurs "
        << course->m_title
        << " (" << course->m_ects << ")? [da/ne]" << std::endl;

    std::string response;
    std::cin >> response;
    if (response != "da") {
        continue;
    }

    student->m_courses.push_back(course);
}
```

Nakon što završimo učitavanje svih studenata, svaki student će sadržati vektor deljenih pokazivača na kurseve koje se upisao. To znači da nam deljeni pokazivači u vektoru `courses` na steku funkcije `main()` više nisu potrebni, pa možemo očistiti taj vektor. Time se efektivno brišu svi deljeni pokazivači na

kurseve iz tog vektora i smanjuju se brojači za sve kurseve za jedan. Ukoliko je postojao kurs koji smo učitali na početku kojeg nije upisao nijedan student, onda se taj kurs u potpunosti briše, čime oslobađamo svu nepotrebnu memoriju iz našeg programa do tog trenutka.

```
courses.clear();
```

Konačno, prolazimo kroz vektor `students` i ispisujemo indekse svakog studenta, praćene spiskom upisanih kurseva.

```
for (const auto &student : students) {
    std::cout
        << "Student sa indeksom " << student->m_index
        << " ima naredne upisane kurseve:" << std::endl;

    for (const auto& course : student->m_courses) {
        std::cout
            << '\t' << course->m_title
            << " (" << course->m_ects << ")" << std::endl;
    }
}
```

Kompletan kod je dat u rešenju 5.

U nastavku dajemo primer interakcije sa programom.

Standardni ulaz/izlaz

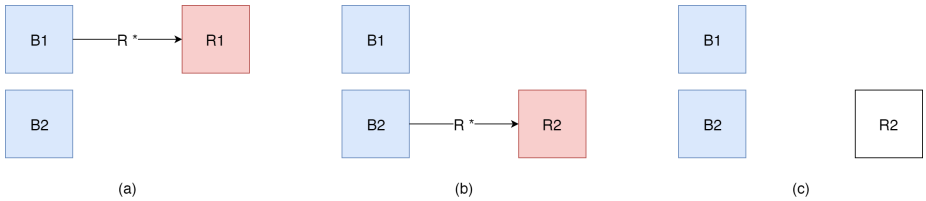
```
Unesite broj kurseva:
3

Unesite naziv kursa:
RazvojSoftvera
Unesite broj bodova:
6

Unesite naziv kursa:
Programiranje1
Unesite broj bodova:
8
```

```
Unesite naziv kursa:
TopologijaA
Unesite broj bodova:
5

Unesite broj studenata:
2

Unesite indeks studenta:
1/2021

Da li zelite da upisete tekuceg studenta na kurs
RazvojSoftvera (6)? [da/ne]
da
Da li zelite da upisete tekuceg studenta na kurs
Programiranje1 (8)? [da/ne]
da
Da li zelite da upisete tekuceg studenta na kurs
TopologijaA (5)? [da/ne]
ne

Unesite indeks studenta:
2/2021

Da li zelite da upisete tekuceg studenta na kurs
RazvojSoftvera (6)? [da/ne]
ne
Da li zelite da upisete tekuceg studenta na kurs
Programiranje1 (8)? [da/ne]
da
Da li zelite da upisete tekuceg studenta na kurs
TopologijaA (5)? [da/ne]
da

Student sa indeksom 1/2021 ima naredne upisane kurseve:
        RazvojSoftvera (6)
```

```
        Programiranje1 (8)
Student sa indeksom 2/2021 ima naredne upisane kurseve:
        Programiranje1 (8)
        TopologijaA (5)
```

2.1.3 Jedinstveno vlasništvo nad dinamičkim objektima

Posmatrajmo sada narednu situaciju. Neka postoji objekat `r1` strukture `R` koji predstavlja dinamički objekat i neka postoje dva objekta `b1` i `b2` strukture `B`, pri čemu ovoga puta najviše jedan od `B` objekata može da operiše nad `R` objektom. Ovu situaciju nazivamo jedinstvenim vlasništvom nad dinamičkim objektom. Na slici 2.9 ilustrovan je opisani koncept. Primetimo da u jednom trenutku tačno jedan od objekata `b1` i `b2` sadrži pokazivač ka dinamičkom objektu `r1`, dok onaj drugi objekat nema pristup tom istom dinamičkom objektu.



Slika 2.9: Na slici (a) objekat `b1` ima jedistveno vlasništvo, dok na slici (b) objekat `b2` ima jedinstveno vlasništvo nad dinamičkim objektom `r1`. Na slici (c) nijedan objekat nema jedinstveno vlasništvo, te se dinamički objekat može uništiti.

Razmotrimo sada implementacione napore da se ovakav zahtev implementira pomoću sirovih pokazivača. Prenos jedinstvenog vlasništva sa objekta `b1` na objekat `b2` nad dinamičkim objektom `r1` opisan je narednom procedurom:

- Sačuvamo vrednost pokazivača ka `r1` u objektu `b2`.
- Postavimo vrednost pokazivača `R *` u objektu `b1` na `nullptr`.

Prirodno se postavlja pitanje uništavanja dinamičkog objekta. U ovakvoj situaciji, dovoljno je proveriti da li pokazivač ka dinamičkom objektu u `B` objektu sadrži neku validnu adresu – u kojem slučaju je potrebno uništiti taj dinamički objekat – ili sadrži `nullptr` – u kojem slučaju nije potrebno izvršiti nikakvu

akciju. Ipak, važno je napomenuti da postoji slučaj u kojem je moguće izazvati curenje memorije ukoliko nismo oprezni, a to je slučaj kada se svi objekti odreknu jedinstvenog vlasništva nad dinamičkim objektom. Takva situacija je prikazana na slici 2.9(c). Dakle, ukoliko ne prebacujemo vlasništvo na neki drugi objekat, već se samo odričemo vlasništva, to znači da nijedan objekat neće čuvati informaciju o dinamičkom objektu. U tom slučaju, procedura za odricanje od vlasništva objekta `b1` nad dinamičkim objektom `r1` izgleda:

- Obrisati dinamički objekat `r1`.
- Postaviti vrednost pokazivača `R *` u objektu `b1` na `nullptr`.

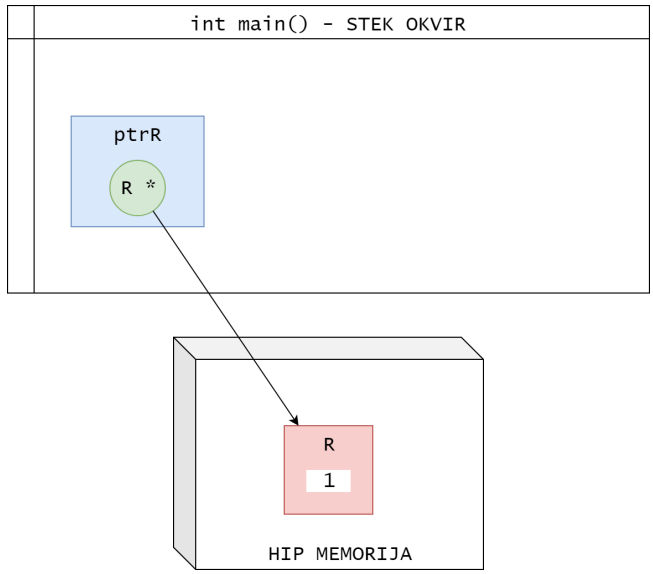
Zaključujemo da je rukovanje dinamičkim objektima u slučaju jedinstvenog vlasništva pogodnije nego u slučaju deljenih dinamičkih objekata. Ograničavanje zahteva, tamo gde je to moguće, često smanjuje prostor za nastajanje grešaka. Ipak, netrivijalne procedure za njihovo upravljanje koje su opisane iznad, iako se sastoje od po svega dva koraka, uvode mogućnost da nepažljivi programeri izazovu probleme u radu aplikacije.

U standardnoj biblioteci programskog jezika C++ definisana je šablonska klasa `std::unique_ptr<T>`, kojom se jednostavno i bezbedno postižu opisani zahtevi za jedinstvenim vlasništvom dinamičkih objekata. Konstrukcija objekata ove klase, koje ćemo nazivati jedinstvenim pokazivačima, vrši se šablonskom funkcijom `std::make_unique<T>`. Date klasa i funkcija su definisani u sistemskom zaglavlju `memory`. Pogledajmo jedan jednostavan primer koji ilustruje korišćenje jedinstvenog pokazivača.

```
#include <memory>
#include <iostream>

struct R {
    int m_x;

    R(int x) {
        m_x = x;
    }

    ~R() {
        std::cout << "R objekat je unisten" << std::endl;
    }
};
```

```
struct B {
    std::unique_ptr<R> m_ptrR;
};

int main() {
    auto ptrR = std::make_unique<R>(1);

    B b1;
    b1.m_ptrR = std::move(ptrR);

    if (b1.m_ptrR) {
        std::cout << "b1: " << b1.m_ptrR->m_x << std::endl;
    }
    else {
        std::cout << "b1 nema jedinstveno vlasnistvo" << std::endl;
    }

    // Blok 1
    {
        B b2;
        b2.m_ptrR = std::move(b1.m_ptrR);

        if (b1.m_ptrR) {
            std::cout << "b1: " << b1.m_ptrR->m_x << std::endl;
        }
        else {
            std::cout << "b1 nema jedinstveno vlasnistvo" << std::endl;
        }

        if (b2.m_ptrR) {
            std::cout << "b2: " << b2.m_ptrR->m_x << std::endl;
        }
        else {
            std::cout << "b2 nema jedinstveno vlasnistvo" << std::endl;
        }
    }

    if (b1.m_ptrR) {
```

```
        std::cout << "b1: " << b1.m_ptrR->m_x << std::endl;
    }
    else {
        std::cout << "b1 nema jedinstveno vlasnistvo" << std::endl;
    }

    return 0;
}
```



Slika 2.10: Grafički prikaz stek okvira funkcije `main` i stanja hip memorije prilikom konstrukcije jedinstvenog pokazivača.

Na samom pocetku, definišemo strukture `R` i `B` kao i u primeru sa deljenih objektima. Zatim, u `main` funkciji definišemo jedinstveni pokazivač `ptrR` koji interno čuva sirovi pokazivač `R *` ka dinamičkom objektu na hip memoriji. Ovo je ilustrovano na slici 2.10.

Zatim kreiramo objekat `b1` strukture `B` sa namerom da taj objekat preuzme jedinstveno vlasništvo nad dinamičkim objektom iz jedinstvenog pokazivača `ptrR`. Naivni pokušaj dodele vrednosti jednog jedinstvenog pokazivača drugom rezultuje kompilatorskom greškom.

```
b1.m_ptrR = ptrR;
```

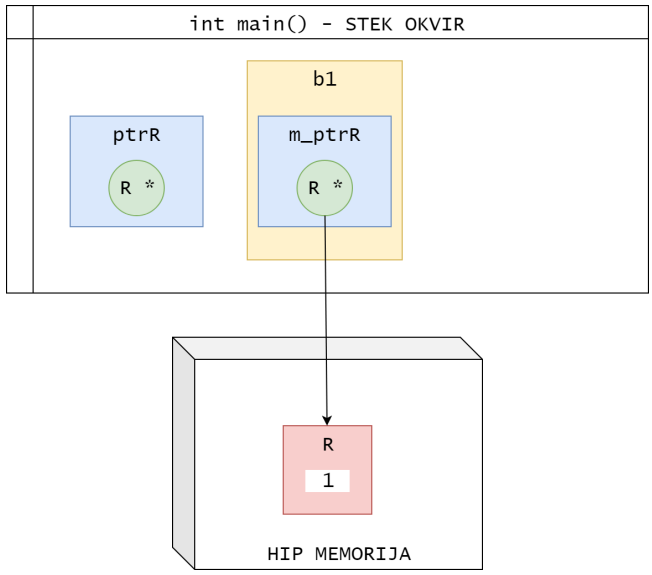
Primer kompilatorske greške je dat u nastavku. Izlaz je skraćen radi čitljivosti.

Standardni izlaz za greške

```
main.cpp(24): error C2280:
  'unique_ptr<R> &unique_ptr<R>::operator=(unique_ptr<R> &)':
      attempting to reference a deleted function
memory(2552): note:
  see declaration of 'unique_ptr<R>::operator='
memory(2552): note:
  'unique_ptr<R> &unique_ptr<R>::operator=(unique_ptr<R> &)':
      function was explicitly deleted
```

Zbog čega se ova kompilatorska greška javlja? Programski jezik C++ razlikuje koncepte *kopiranja* i *pomeranja* vrednosti. Pojednostavljeno rečeno, kopiranje vrednosti podrazumeva da se konstruiše nova vrednost koja predstavlja kopiju postojeće vrednosti bez njene izmene, dok pomeranje vrednosti podrazumeva da se konstruiše vrednost poput postojeće vrednosti, ali ovoga puta, postojeća vrednost postaje „prazna". Još jednostavnije rečeno, pomeranje vrednosti „krade" tu vrednost iz izvornog objekta i postavlja je u novom objektu. Prema pravilu, originalnu vrednost iz koje se „ukralo" ne bi trebalo koristiti nadalje u aplikaciji (jer je ostala „prazna", odnosno, nema vrednost). Programski jezik C++ dozvoljava da programer definiše mehanizme pomoću kojeg će vršiti kopiranje i pomeranje korisnički-definisanih struktura i klasa. Ali, takođe, dozvoljava i mogućnost zabrane kopiranja ili pomeranja. O implementaciji ovih mehanizama će biti detaljnije diskutovano u poglavlju 3.

Dodeljivanje jedne vrednosti drugoj predstavlja operaciju kopiranja. Zbog toga, u fragmentu koda iznad mi pokušavamo da kopiramo jedinstveni pokazivač `ptrR` u jedinstveni pokazivač `b1.m_ptrR`. Međutim, jedinstveni pokazivač je implementiran tako da zabranjuje kopiranje, te zbog toga dobijamo kompilatorsku grešku. Međutim, ako malo razmislimo, zabrana kopiranja jedinstvenih pokazivača ima savršeno smisla. Zašto bismo dozvolili korisniku da kopira podatke o deljenom objektu, ako najviše jedan jedinstveni pokazivač može da čuva te informacije? Upravo zato i koristimo jedinstvene pokazivače. Dakle, nije greška u implementaciji jedinstvenih pokazivača, nego u načinu na koji smo ih koristili.



Slika 2.11: Grafički prikaz stek okvira funkcije `main` i stanja hip memorije prilikom pomeranja jedinstvenog pokazivača u objekat `b1`.

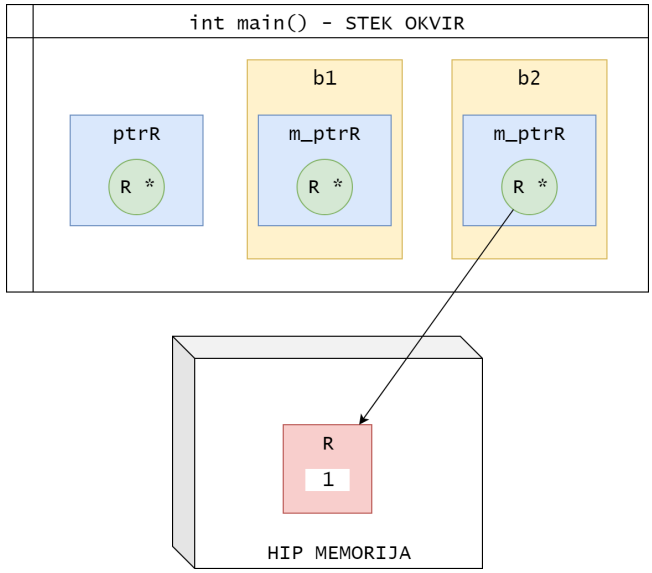
Umesto kopiranja jedinstvenih pokazivača, moramo da kažemo kompilatoru da želimo da se vrši njihovo pomeranje. To je moguće uraditi pozivom šablonske funkcije `std::move`, kao u narednom fragmentu koda.

```
b1.m_ptrR = std::move(ptrR);
```

Važno je napomenuti da ova funkcija ne radi nikakvo pomeranje, već samo naznačuje kompilatoru da ne koristi mehanizam kopiranja prilikom dodeljivanja vrednosti, već da koristi mehanizam pomeranja[7]. Načini, odnosno, mehanizmi

[7]Još malo o složenosti programskog jezika C++. Neke vrednosti će prirodno koristiti mehanizam pomeranja, ako je on implementiran, umesto da se vrši kopiranje. Takve su tzv. *privremene vrednosti*, odnosno, one vrednosti čiji je životni vek vezan za izračunavanje izraza koji ih konstruiše. Dakle, kada se započne izračunavanje izraza koji ih konstruiše i same vrednosti se konstruišu, a nakon što se završi izračunavanje izraza, oni se uništavaju. O privremenim vrednostima ćemo govoriti više u poglavlju 3, ali je vredno napomenuti da one postoje. A kad smo ih pomenuli, onda je vredno i pomenuti da postoje određene optimizacije kompilatora koje mogu da *izgebnu* kopiranje ili pomeranje i time još efikasnije konstruišu nove vrednosti od postojećih. Kao što vidimo, koncepti kopiranja i pomeranja su izuzetno složeni za početnike u programskom jeziku C++, a nekada i za programere sa višegodišnjim iskustvom.

na osnovu kojih će biti izvršeno kopiranje ili pomeranje su ostavljeni programeru da ih implementira. Njihova implementacija se vrši u klasi objekta koji se prosleđuje kao argument funkcije `std::move`. To znači da kompilator očekuje da postoji mehanizam pomeranja definisan u klasi `std::unique_ptr` i, zaista, taj mehanizam i postoji. Pojednostavljeno rečeno, taj mehanizam pomeranja u klasi `std::unique_ptr` implementira upravo onu proceduru za prenos jedinstvenog vlasništva koji smo naveli ranije u tekstu. Grafički, ova operacija proizvodi rezultat koji je dat na slici 2.11.



Slika 2.12: Grafički prikaz stek okvira funkcije `main` i stanja hip memorije prilikom pomeranja jedinstvenog pokazivača u objekat `b2`.

Kako bismo testirali da li je ovo korektno, pokušavamo da ispišemo na standardni izlaz sadržaj iz dinamičkog objekta preko jedinstvenog pokazivača u objektu `b1`. Kao što vidimo, objekat `b1` ima pristup dinamičkom objektu.

Standardni izlaz

```
b1: 1
```

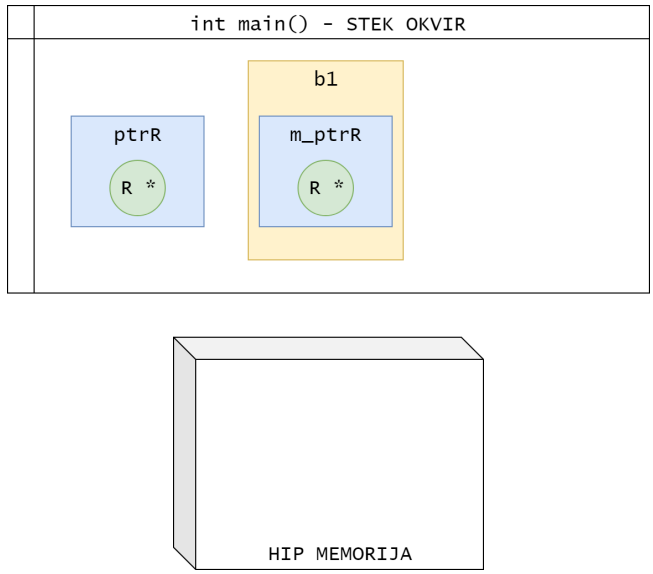
Nadalje, ulazimo u „blok 1” u kojem je definisan objekat `b2`. Kako bismo pre-

bacili jedinstveno vlasništvo nad dinamičkim objektom sa objekta `b1` na objekat `b2`, ponovo je potrebno da pomerimo jedinstveni pokazivač. Ova operacija je ilustrovana na slici 2.12. Ponovo testiramo vlasništvo ispisivanjem odgovarajućih poruka na standardni izlaz. Kao što vidimo, `b1` više nema pristup dinamičkom objektu, ali `b2` ima.

Standardni izlaz

```
b1 nema jedinstveno vlasnistvo
b2: 1
```

Po završetku „bloka 1", automatski biva uništen objekat `b2`, a zajedno sa njim i jedinstveni pokazivač na dinamički objekat. Kako je to bio jedini jedinstveni pokazivač na dinamički objekat, onda i sam dinamički objekat biva uništen. Ova situacijama je ilustrovana na slici 2.13.



Slika 2.13: Grafički prikaz stek okvira funkcije `main` i stanja hip memorije prilikom napuštanja „bloka 1".

Da zaista dolazi do uništavanja, možemo primetiti u ispisu na standardnom izlazu.

Standardni izlaz

```
R objekat je unisten
```

Provere radi, ukoliko pokušamo da pristupimo informaciji o dinamičkom objektu preko jedinstvenog pokazivača iz objekta `b1`, videćemo da to nije moguće uraditi, tako da smo se osigurali od nedozvoljenog pristupa memoriji.

Standardni izlaz

```
b1 nema jedinstveno vlasnistvo
```

Zadatak 7: Study programs

Napisati program komandne linije koji omogućava korisniku da upisuje studente na smerove. Smerovi su definisani svojim nazivom, pozitivnim brojem bodova i spiskom studenata koji su upisani na taj smer, a studenti svojim indeksom i imenom. Program prvo traži od korisnika da unese pozitivan broj s, a zatim učitava s smerova (za svaki smer se čita prvo naziv, pa zatim broj bodova). Zatim, program traži od korisnika da unese pozitivan broj n, a zatim učitava n studenata (za svakog studenta se unosi indeks, pa zatim ime). Nakon učitavanja jednog studenta, program ispisuje informacije o smerovima i njihovim rednim brojevima i zahteva od korisnika da unese redni broj smera na kojem će upisati datog studenta. Nakon unošenja rednog broja smera, zabeležiti informaciju o upisu. Nakon učitavanja svih studenata, program ispisuje za svaki smer spisak studenata koji su upisali taj smer.

Kao i pri rešavanju zadatka 6, započinjemo definisanjem struktura koje ćemo koristiti u aplikaciji. Jedan student je opisan svojim indeksom i imenom, te definišemo strukturu `Student` sa dva atributa tipa `std::string`.

```
#include <string>

struct Student {
    std::string m_index;
    std::string m_name;
```

```
    Student(std::string index, std::string name) {
        m_index = index;
        m_name = name;
    }
};
```

Smerovi, pored elementarnih podataka, moraju da čuvaju informaciju o studentima koji su upisali te smerove. Primetimo iz formulacije zadatka da jedan student može da upiše najviše jedan smer. Drugim rečima, smer drži jedinstveno vlasništvo nad studentima. Ovo je idealna prilika da iskoristimo jedinstvene pokazivače.

Smerove ćemo predstavljati strukturom `StudyProgram` koja ima atribute tipa `std::string` za naziv smera, `unsigned` za broj bodova i vektor jedinstvenih pokazivača na objekte prethodno definisane strukture `Student`.

```
#include <vector>

...

struct StudyProgram {
    std::string m_title;
    unsigned m_ects;
    std::vector<std::unique_ptr<Student>> m_students;

    StudyProgram(std::string title, unsigned ects) {
        m_title = title;
        m_ects = ects;
    }
};
```

Pređimo sada na implementaciju `main` funkcije. Dajmo opis toka izvršavanja programa:

- Učitavanje smerova u neku kolekciju podataka, na primer, vektor.
- Učitavanje jednog po jednog studenta.
  - Za svakog studenta, ispisujemo spisak smerova i upisujemo studenta na odabrani smer.
- Izlistavanje podataka o smerovima.

Prvo učitavamo broj s sa standardnog ulaza. Zatim, pripremamo vektor u koji ćemo smeštati informacije o smerovima. Kako informacije o smerovima nije potrebno deliti kroz aplikaciju, možemo i za njih koristiti jedinstvene pokazivače.

```
#include <iostream>

...

std::cout << "Unesite broj smerova:" << std::endl;
unsigned s;
std::cin >> s;

std::vector<std::unique_ptr<StudyProgram>> studyPrograms;

for (auto i = 0u; i < s; ++i) {
    std::cout << "Unesite naziv smera: " << std::endl;
    std::string title;
    std::cin >> title;

    std::cout << "Unesite broj bodova: " << std::endl;
    unsigned ects;
    std::cin >> ects;

    auto studyProgram = std::make_unique<StudyProgram>(title, ects);
    studyPrograms.push_back(std::move(studyProgram));
}
```

Primetimo da, prilikom dodavanja jedinstvenog pokazivača na kraj vektora pomoću metoda `push_back`, pozivamo funkciju `std::move` kako bismo naglasili da želimo da se pomeri konstruisani pokazivač na `StudyProgram` u vektor. Da ovo nismo uradili, kompilator bi pokušao da kopira jedinstveni pokazivač, za koji smo pokazali da dobijamo kompilatorsku grešku, odnosno, da to nije poželjna operacija. Dakle, koncepti kopiranja i pomeranja vrednosti nisu vezani samo za dodeljivanje vrednosti, već za bilo koju operaciju u kojoj se konstruiše nova vrednost od postojeće. U fragmentu koda iznad, vrednost koja se konstruiše jeste novi poslednji element u vektoru `studyPrograms`, a vrednost na osnovu koje se ona konstruiše je ona u promenljivoj `studyProgram`. S obzirom da se nova vrednost može konstruisati kopiranjem ili pomeranjem postojeće vrednosti `studyProgram`, moramo specifikovati da želimo pomeranje.

Započnimo sada obradu koja se odnosi na studente. Prvo učitavamo broj studenata n, a zatim n puta učitavamo podatke za svakog studenta i kreiramo jedinstveni pokazivač na tekućeg studenta.

```
std::cout << "Unesite broj studenata:" << std::endl;
unsigned n;
std::cin >> n;

for (auto i = 0u; i < n; ++i) {
    std::cout << "Unesite indeks studenta: " << std::endl;
    std::string index;
    std::cin >> index;

    std::cout << "Unesite ime studenta: " << std::endl;
    std::string name;
    std::cin >> name;

    auto student = std::make_unique<Student>(index, name);

    // Upisivanje studenta na smer
    ...
}
```

Sada je neophodno dopuniti petlju operacijom upisivanja studenta na smer. Prvo moramo ispisati informacije o postojećim smerovima, kako bismo mogli od korisnika da zatražimo povratnu informaciju o odabranom smeru. S obzirom da nam je neophodan i redni broj smera, počevši od 1, koristimo klasičnu `for` petlju za iteriranje sa indeksom j i indeksni operator za pristup j-tom smeru iz vektora.

```
for (auto j = 0u; j < studyPrograms.size(); ++j) {
    std::cout
        << j + 1 << ". "
        << studyPrograms[j]->m_title << std::endl;
}
```

Dalje, pitamo korisnika za koji smer se odlučuje. U kodu ispod pretpostavljamo da će korisnik uneti neki od izlistanih rednih brojeva.

```
std::cout
```

```
    << "Unesite identifikator smera za upis studenta: "
    << std::endl;

unsigned chosenSP;
std::cin >> chosenSP;
```

S obzirom da je korisnik uneo indeks koji počinje od 1, moramo pristupiti smeru iz vektora na jednoj poziciji manje od unete, pa u njegovom vektoru studenata pomeriti konstruisani jedinstveni pokazivač na studenta.

```
studyPrograms[chosenSP-1]->m_students.push_back(std::move(student));
```

Kako bi čitalac stekao kompletnu sliku onoga što je urađeno u ovom delu toka programa, dajemo kompletan kod petlje koja obrađuje podatke jednog po jednog studenta.

```
for (auto i = 0u; i < n; ++i) {
    std::cout << "Unesite indeks studenta: " << std::endl;
    std::string index;
    std::cin >> index;

    std::cout << "Unesite ime studenta: " << std::endl;
    std::string name;
    std::cin >> name;

    auto student = std::make_unique<Student>(index, name);

    for (auto j = 0u; j < studyPrograms.size(); ++j) {
        std::cout
            << j + 1 << ". "
            << studyPrograms[j]->m_title << std::endl;
    }
    std::cout
        << "Unesite identifikator smera za upis studenta: "
        << std::endl;

    unsigned chosenSP;
    std::cin >> chosenSP;

    studyPrograms[chosenSP-1]->m_students.push_back(std::move(student));
}
```

Konačno, potrebno je da prođemo kroz vektor smerova i ispišemo informacije. Hajde da iskoristimo kolekcijsku `for` petlju za ovaj zadatak. Pažljivi čitaoci će primetiti da sada nije svejedno hoćemo li koristiti pristup sa referencama ili ne. Zaista, ukoliko pokušamo da prevedemo naredni fragment koda

```
for (const auto studyProgram : studyPrograms) {
    ...
}
```

dobijamo kompilatorsku grešku za kopiranjem jedinstvenih pokazivača. Zato, ovde je neophodno koristiti pristup sa referencama.

```
for (const auto &studyProgram : studyPrograms) {
    std::cout
        << "Upisani studenti na kursu " << studyProgram->m_title
        << " (" << studyProgram->m_ects << "):" << std::endl;

    for (const auto& student : studyProgram->m_students) {
        std::cout
            << '\t' << student->m_name
            << " (" << student->m_index << ")" << std::endl;
    }
}
```

Kompletan kod je dat u rešenju 6.

U nastavku dajemo primer interakcije sa programom.

Standardni ulaz/izlaz

```
Unesite broj smerova:
3

Unesite naziv smera:
Informatika
Unesite broj bodova:
240

Unesite naziv smera:
RacunarstvoIInformatika
```

```
Unesite broj bodova:
240

Unesite naziv smera:
TeorijskaMatematikaIPrimene
Unesite broj bodova:
240

Unesite broj studenata:
5

Unesite indeks studenta:
1/2021
Unesite ime studenta:
MarijaPetrovic
1. Informatika
2. RacunarstvoIInformatika
3. TeorijskaMatematikaIPrimene
Unesite identifikator smera za upis studenta:
1

Unesite indeks studenta:
2/2021
Unesite ime studenta:
IvanJanjic
1. Informatika
2. RacunarstvoIInformatika
3. TeorijskaMatematikaIPrimene
Unesite identifikator smera za upis studenta:
3

Unesite indeks studenta:
3/2021
Unesite ime studenta:
PetarZivojinovic
1. Informatika
2. RacunarstvoIInformatika
```

```
3. TeorijskaMatematikaIPrimene
Unesite identifikator smera za upis studenta:
1

Unesite indeks studenta:
4/2021
Unesite ime studenta:
MilenaIlic
1. Informatika
2. RacunarstvoIInformatika
3. TeorijskaMatematikaIPrimene
Unesite identifikator smera za upis studenta:
2

Unesite indeks studenta:
5/2021
Unesite ime studenta:
JovanaPeric
1. Informatika
2. RacunarstvoIInformatika
3. TeorijskaMatematikaIPrimene
Unesite identifikator smera za upis studenta:
2

Upisani studenti na kursu Informatika (240):
        MarijaPetrovic (1/2021)
        PetarZivojinovic (3/2021)
Upisani studenti na kursu RacunarstvoIInformatika (240):
        MilenaIlic (4/2021)
        JovanaPeric (5/2021)
Upisani studenti na kursu TeorijskaMatematikaIPrimene (240):
        IvanJanjic (2/2021)
```

2.2 Idiom RAII

Kao što smo nekoliko puta do sada istakli, upravljanje dinamičkim resursima je od izuzetnog značaja za ispravan rad svih složenijih aplikacija. Tačka na kojoj smo posebno insistirali jeste da svaki dinamički resurs koji aplikacija „zauzme”

mora korektno da se „oslobodi”. To je obično podrazumevalo neku vrstu inicijalizacije resursa na početku neke operacije, a zatim deinicijalizaciju tog resursa nakon što se ta operacija izračunala. Međutim, programski jezik C++ definiše koncept *izuzetaka* i mehanizama za rad sa njima, što znači da izuzetak može nastati u proizvoljnom izračunavanju. Ovo nam može stvoriti dodatni problem pri upravljanju dinamičkih resursa. Razmotrimo narednu situaciju:

```
#include <iostream>
#include <exception>

struct PositiveNum {
    int m_number;

    PositiveNum() {
        std::cout << "Konstruisan je PositiveNum" << std::endl;
    }

    void setNumber(int number) {
        if (number < 0) {
            throw std::exception("number ne sme biti negativan!");
        }
        m_number = number;
    }

    ~PositiveNum() {
        std::cout << "Unisten je PositiveNum" << std::endl;
    }
};

void f() {
    auto ptr = new PositiveNum();

    int number;
    std::cout << "Unesite celi broj: " << std::endl;
    std::cin >> number;

    ptr->setNumber(number);

    std::cout
```

```
        << "PositiveNum ima vrednost: "
        << ptr->m_number << std::endl;

    delete ptr;
}

int main() {
    try {
        std::cout << "Pozivam funkciju f()" << std::endl;
        f();
        std::cout << "Funkcija f() je uspesno zavrsena" << std::endl;
    } catch (const std::exception &e) {
        std::cout<< "Ispaljen je izuzetak: " << e.what() << std::endl;
    }

    std::cout << "Kraj programa" << std::endl;
    return 0;
}
```

Ovaj primer, iako veštački konstruisan, jasno ilustruje problem do kojeg dolazi u slučaju da korisnik unese nevalidan ulaz. Objekti strukture `PositiveNum` ne smeju sadržati negativne vrednosti, te ukoliko se nedozvoljena vrednost prosledi metodu `setNumber`, biće ispaljen izuzetak. Da bismo razumeli problem do kojeg dolazi, prikažimo prvo slučaj kada ne dolazi izuzetak. Neka, na primer, korisnik unese broj 10. Interakcija sa programom je prikazana u nastavku.

Standardni ulaz/izlaz

```
Pozivam funkciju f()
Konstruisan je PositiveNum

Unesite celi broj:
10

PositiveNum ima vrednost: 10
Unisten je PositiveNum
Funkcija f() je uspesno zavrsena
Kraj programa
```

Kao što vidimo, funkcija `f` se završila uspešno i dinamički resurs je ispravno

oslobođen. Pogledajmo sada interakciju sa programom u slučaju da korisnik unese negativan broj, na primer, -3.

Standardni ulaz/izlaz

```
Pozivam funkciju f()
Konstruisan je PositiveNum

Unesite celi broj:
-3

Ispaljen je izuzetak: number ne sme biti negativan!
Kraj programa
```

Neispravan unos od korisnika je proizveo izuzetak, što povlači da se funkcija `setNumber` nije uspešno završila u funkciji `f`, već se izvršavanje nastavlja u `catch` bloku u funkciji `main`. Međutim, primetimo šta se desilo sa dinamičkim objektom – na standardnom izlazu ne vidimo ispis iz destruktora objekta `PositiveNum` i to sa jasnim objašnjenjem – izvršavanje funkcije `f` se nikada nije dovršilo do kraja, pa samim tim nikad nije pozvan operator `delete`. Dakle, postoji opasnost od curenja memorije i ostalih problema do kojih može doći usled neispravnog oslobađanja memorije.

Programerski idiomi predstavljaju ustaljene i dobro proverene tehnike programiranja koje uklanjaju razne probleme do kojih može doći prilikom izvršavanja programa. Jedan od takvih idioma je poznat pod nazivom „zauzimanje resursa je inicijalizacija" (eng. *resource acquisition is initialization*, skr. *RAII*) i on služi za korektno upravljanje dinamičkim resursima u slučaju pojave izuzetaka. Osnovna ideja idioma RAII jeste da upravljanje dinamičkim objektom omota u neku klasu, tako da se kod koji vrši inicijalizaciju resursa stavi u konstruktor te klase, a da se kod koji vrši oslobađanje resursa stavi u destruktor te klase. Zatim, u funkciji gde se dinamički resurs koristi, kreiramo objekat RAII klase koja upravlja tim dinamičkim resursom, pri čemu taj objekat obavezno mora biti automatskog životnog veka. Pokažimo prvo kako bismo implementirali RAII idiom na prethodnom primeru, pa ćemo objasniti zašto on funkcioniše.

```
struct PositiveNumRAII {
    PositiveNum *m_ptr;

    PositiveNumRAII() {
```

```
        m_ptr = new PositiveNum();
    }

    ~PositiveNumRAII() {
        delete m_ptr;
    }
};

void f() {
    PositiveNumRAII posNum;

    int number;
    std::cout << "Unesite celi broj: " << std::endl;
    std::cin >> number;

    posNum.m_ptr->setNumber(number);

    std::cout
        << "PositiveNum ima vrednost: "
        << posNum.m_ptr->m_number << std::endl;
}
```

Osim dodavanja nove strukture `PositiveNumRAII` i izmene koda samo na onom mestu gde se dinamički resurs koristi, nemamo nigde drugde izmenu u programu – struktura `PositiveNum` čije dinamičke objekte konstruišemo i funkcija `main` su ostale nepromenjene. Slučaj kada korisnik unese validan ulaz ostane identičan kao i malopre, te ga nećemo prikazati. Međutim, pogledajmo šta se dešava ovoga puta kada korisnik unese nevalidan ulaz.

Standardni ulaz/izlaz

```
Pozivam funkciju f()
Konstruisan je PositiveNum

Unesite celi broj:
-3

Unisten je PositiveNum
Ispaljen je izuzetak: number ne sme biti negativan!
```

```
Kraj programa
```

Kao što vidimo, ponovo dolazi do ispaljivanja izuzetka, što je i očekivano s obzirom da nismo menjali kod klase `PositiveNum`, ali ovoga puta vidimo ispis iz destruktora te klase, što znači da se dinamički objekat zaista uništio. RAII idiom, implementiran na ovaj način u programskom jeziku C++, moguć je zbog činjenice da prilikom odmotavanja steka od ispaljivanja izuzetka u nekoj funkciji do nastavka izvršavanja u `catch` bloku koji hvata ispaljeni izuzetak (ako takav blok postoji), *svi objekti sa automatskim životnim vekom se obavezno uništavaju.* Da ovo nije slučaj, onda se RAII idiom ne bi mogao implementirati na ovaj način. Međutim, umotavanje koda koji zauzima, odnosno, oslobađa dinamički resurs u zasebnu klasu, osim što omogućuje ispravno upravljanje tim dinamičkim resursom, takođe doprinosi razdvajanju odgovornosti među klasama, odnosno, funkcijama u kodu, čime se postiže bolji dizajn programa.

2.2.1 RAII i dinamički objekti

Pogledajmo sada kako možemo poboljšati kvalitet rešenja nekih od prethodnih zadataka uvođenjem RAII idioma.

Zadatak 8: RAII points

Implementirati rešenje zadatka 4 korišćenjem RAII idioma.

Ukoliko želimo da uvedemo RAII idiom, potrebno je prvo da razumemo koji su to dinamički resursi koji se konstruišu, kao i koje su procedure za inicijalizaciju, odnosno, oslobađanje tih resursa. Zatim, izdvajamo prepoznate resurse i procedure za njihovo upravljanje u novu, RAII klasu. U zadatku 4, dinamički resurs predstavlja vektor tačaka čija memorija se alocira na hip memoriji. Zbog toga, kreirajmo novu strukturu, `raii_points` koja će upravljati tačkama.

```
struct raii_points {
    std::vector<const point *> points;

    void generate_two_random_doubles(double &x, double &y) {
        static const auto lower = -1000.0;
        static const auto upper = 1000.0;

        static std::uniform_real_distribution<double> unif(lower, upper);
```

```
        static std::default_random_engine re;
        // re.seed(0);

        x = unif(re);
        y = unif(re);
    }

    void generate_n_points(unsigned long long n) {
        double x, y;

        for (auto i = 0ull; i < n; i++) {
            generate_two_random_doubles(x, y);

            const auto p = new point(x, y);
            if (!p) {
                break;
            }

            points.push_back(p);
        }
    }

    raii_points(unsigned long long n) {
        std::cout << "Konstruktor raii_points" << std::endl;
        generate_n_points(n);
    }

    ~raii_points() {
        std::cout << "Destruktor raii_points" << std::endl;
        for (const auto &p : points) {
            delete p;
        }
    }
};
```

Primetite da smo funkcije `generate_two_random_doubles` i `generate_n_points` premestili kao metode ove strukture. Iako to nije od suštinske važnosti za ovaj program, ipak smo postigli da je odgovornost upravljanja tačkama isključivo na ovoj strukturi, čime smo malo popravili i dizajn.

Izmenimo funkciju `distance_between_points` tako da ispali izuzetak ukoliko je rastojanje približno nuli, radi simulacije greške.

```
double distance_between_points(const point *p1, const point *p2) {
    const auto result = std::sqrt(std::pow(p1->m_x - p2->m_x, 2) +
                                  std::pow(p1->m_y + p2->m_y, 2));
    if (result < 0.1) {
        throw std::exception("Tacke su previse blizu jedna drugoj");
    }
    return result;
}
```

Funkcije `find_farthest_point_from_kth` i `main` se menjaju tek toliko da prilagode svoje implementacije novoj strukturi `raii_points`, premda smo ih mogli ostaviti nepromenjene.

```
const point *find_farthest_point_from_kth(
        raii_points &pRAII,
        unsigned long long n,
        const point *kth_point) {
    auto max_dist = distance_between_points(pRAII.points[0], kth_point);
    auto idx_max_dist = 0ull;

    for (auto i = 1ull; i < n; ++i) {
        const auto curr_dist =
            distance_between_points(pRAII.points[i], kth_point);
        if (curr_dist > max_dist) {
            max_dist = curr_dist;
            idx_max_dist = i;
        }
    }

    return pRAII.points[idx_max_dist];
}

int main() {
    unsigned long long n;
    std::cout << "Unesite broj n veci od 1.000.000: ";
    std::cin >> n;
```

```
    try {
        raii_points pRAII(n);

        unsigned long long k;
        std::cout << "Unesite broj k manji od n: ";
        std::cin >> k;

        const auto kth_point = pRAII.points[k];
        const auto p_found =
            find_farthest_point_from_kth(pRAII, n, kth_point);

        std::cout
            << "Tacka koja je najudaljenija od tacke ("
            << kth_point->m_x << "," << kth_point->m_y << ") je tacka ("
            << p_found->m_x << "," << p_found->m_y << ")." << std::endl;
    } catch (const std::exception &e) {
        std::cout << "Ispaljen je izuzetak: " << e.what() << std::endl;
    }

    return EXIT_SUCCESS;
}
```

U slučaju pojave izuzetka, dobijamo naredni ispis na standardnom izlazu, te vidimo da smo uspešno oslobodili svu zauzetu memoriju.

Standardni ulaz/izlaz

```
Unesite broj n veci od 1.000.000: 1000001
Konstruktor raii_points
Unesite broj k manji od n: 100
Destruktor raii_points
Ispaljen je izuzetak: Tacke su previse blizu jedna drugoj
```

Naravno, tačke se oslobađaju i u slučaju uspeha.

Standardni ulaz/izlaz

```
Unesite broj n veci od 1.000.000: 1000001
Konstruktor raii_points
Unesite broj k manji od n: 100
```

```
Tacka koja je najudaljenija od tacke (87.611,-719.712) je
    tacka (-999.628,-998.897).
Destruktor raii_points
```

Kompletan kod je dat u rešenju 7.

Zadatak 9: RAII array of points

Implementirati rešenje zadatka 5 korišćenjem RAII idioma.

Rešenje ovog zadatka je još jednostavnije od prethodnog. Kreirajmo strukturu `raii_points_array` koja čuva pokazivač na dinamički niz tačaka. U konstruktoru pokušavamo sa alokacijom niza i generisanjem tačaka, dok u destruktoru uništavamo alocirani niz. Kao i u prethodnom rešenju, premeštamo funkcije `generate_two_random_doubles` i `generate_n_points` kao metode ove strukture.

```
struct raii_points_array {
    point *points = nullptr;

    void generate_two_random_doubles(double &x, double &y) {
        static const auto lower = -1000.0;
        static const auto upper = 1000.0;

        static std::uniform_real_distribution<double> unif(lower, upper);
        static std::default_random_engine re;
        // re.seed(0);

        x = unif(re);
        y = unif(re);
    }

    void generate_n_points(unsigned long long n) {
        points = new point[n];
        if (!points) {
            return;
```

```
        }

        double x, y;

        for (auto i = 0ull; i < n; i++) {
            generate_two_random_doubles(x, y);

            points[i].m_x = x;
            points[i].m_y = x;
        }
    }

    raii_points_array(unsigned long long n) {
        std::cout << "Konstruktor raii_points_array" << std::endl;
        generate_n_points(n);
    }

    ~raii_points_array() {
        std::cout << "Destruktor raii_points_array" << std::endl;
        delete[] points;
    }
};
```

U ovoj implementaciji, funkciju `find_farthest_point_from_kth` nećemo menjati. Jedino što je potrebno dodatno uraditi jeste izmeniti funkciju `main` tako da se prilagodi obradi izuzetaka i novoj RAII strukturi.

```
int main() {
    unsigned long long n;
    std::cout << "Unesite broj n veci od 1.000.000: ";
    std::cin >> n;

    try {
        const raii_points_array p_arr(n);
        if (!p_arr.points) {
            std::cerr << "Alokacija je neuspesna!" << std::endl;
            return EXIT_FAILURE;
        }
```

```
        unsigned long long k;
        std::cout << "Unesite broj k manji od n: ";
        std::cin >> k;

        const auto &kth_point = p_arr.points[k];
        const auto &p_found =
            find_farthest_point_from_kth(p_arr.points, n, kth_point);

        std::cout
            << "Tacka koja je najudaljenija od tacke ("
            << kth_point.m_x << "," << kth_point.m_y << ") je tacka ("
            << p_found.m_x << "," << p_found.m_y << ")." << std::endl;
    }  catch (const std::exception &e) {
        std::cout << "Ispaljen je izuzetak: " << e.what() << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Kompletan kod je dat u rešenju 8.

2.2.2 RAII i drugi dinamički resursi

Kao što smo videli, RAII idiom nam omogućava da korektno upravljamo dinamičkim objektima na metodološki način. Dodatna prednost RAII idioma jeste ta što se on jednako primenjuje za sve vrste dinamičkih resursa, bez obzira da li su u pitanju dinamički objekti, dinamički nizovi, datoteke, konekcije na baze podataka, soketi i sl. U ovoj sekciji ćemo prikazati kako se RAII idiom može primeniti pri radu sa datotekama i bazama podataka.

Pored unapređenja postojeće implementacije, u zadacima koji slede koristićemo Qt biblioteku, kako bismo se osvrnuli na njene klase za upravljanje dinamičkim resursima.

Zadatak 10: RAII courses

Implementirati rešenje zadatka 6 korišćenjem RAII idioma. Pri pokretanju programa, učitati podatke o kursevima i studentima iz datoteka `courses.txt` i `students.txt`, ukoliko postoje. Na kraju rada programa,

sačuvati izmenjene podatke u ove datoteke.

S obzirom da koristimo Qt biblioteku, valjalo bi zameniti klase iz standardne biblioteke njihovim alternativama u Qt biblioteci. Tako, na primer, umesto klase `std::string` možemo koristiti `QString`, a umesto šablonske klase `std::vector<T>` možemo koristiti `QVector<T>`. Dodatno, s obzirom da se broj klasa povećava, valjalo bi izdvajati implementacije struktura u zasebne datoteke. Zato, kreirajmo datoteku `Course.hpp` u koju ćemo smestiti strukture za upravljanje informacijama o kursevima. Prikažimo prvo već poznatu implementaciju strukture `Course`, neznatno prilagođenu za rad sa Qt bibliotekom.

```
#include <QString>

struct Course {
    QString m_title;
    unsigned m_ects;

    Course(QString title, unsigned ects) {
        m_title = title;
        m_ects = ects;
    }
};
```

U istoj datoteci definišemo strukturu `CourseRAII` za upravljanje informacijama o kursevima u okviru aplikacije. Kao što znamo, kursevi su deljeni resursi, pa ih je potrebno skladištiti u deljenim pokazivačima. Alternativa deljenom pokazivaču `std::shared_ptr<T>` u Qt biblioteci je šablonska klasa `QSharedPointer<T>`. U okviru ove strukture je potrebno da sačuvamo i naziv datoteke `courses.txt` koju ćemo koristiti za čitanje i za pisanje.

```
#include <QSharedPointer>
#include <QVector>

...

struct CourseRAII {
    QVector<QSharedPointer<Course>> m_courses;
    QString m_filename;
};
```

Konstruktor ove strukture treba prvo da otvori datoteku `courses.txt` za čitanje. Ukoliko datoteka ne može da se otvori, potrebno je signalizirati da je došlo do greške, te je potrebno da konstruišemo objekat te greške. U tu svrhu, konstruišimo strukturu `CannotOpenFileException`, u istoimenoj `.hpp` datoteci, čije objekte ćemo koristiti kao izuzetke. Neka njen konstruktor ima jedan argument koji predstavlja putanju do datoteke koju program nije uspeo da otvori.

```
#include <QString>

struct CannotOpenFileException {
    QString m_what;

    CannotOpenFileException(QString filename) {
        m_what = "Cannot open file " + filename;
    }
};
```

Vraćamo se na implementaciju konstruktora strukture `CourseRAII`.

```
#include <QFile>

...

CourseRAII() {
    m_filename = "courses.txt";

    QFile file(m_filename);
    if (!file.open(QFile::ReadOnly | QFile::Text)) {
        throw CannotOpenFileException(m_filename);
    }

    ...
}
```

Datoteku ćemo držati otvorenu za čitanje sve dok ne završimo sa učitavanjem kurseva, tj. sve do kraja konstruktora. Objekat klase `QFile` i sam implementira RAII idiom za upravljanje datotekama, tako da, sve i da se dogodi neki problem tokom izvršavanja konstruktora, datoteka će biti ispravno zatvorena pri isteku opsega važenja promenljive `file`.

Zatim, potrebno je da parsiramo ovu datoteku i konstruišemo objekte strukture `Course`. Dogovorimo se da se informacije o svakom kursu nalaze u jednoj liniji, odvojene karakterom razmaka. Kako bismo mogli da čitamo iz datoteke `file`, potrebno je da konstruišemo tekstualni tok, koji je u Qt biblioteci predstavljen klasom `QTextStream`. Nad objektom ove klase postoji koristan metod `readLine` koji vraća jednu liniju iz datoteke kao `QString`. Čitanje jedne po jedne linije sada možemo implementirati na sledeći način:

```
#include <QTextStream>

...

CourseRAII() {
    ...

    QTextStream in(&file);
    auto line = in.readLine();
    while (!line.isNull()) {
        ...

        line = in.readLine();
    }
}
```

U obradi jedne linije, potrebno je da razdvojimo podatke po karakteru beline, kako bismo dohvatili informacije o jednom kursu. Klasa `QString` definiše jedan koristan pomoćni metod `split` koji radi upravo to što nam je potrebno. Povratna vrednost metoda je lista niski `QStringList` kojima možemo efikasno pristupiti operatorom indeksiranja. Inače bi bilo korisno proveriti da li je broj elemenata ove liste korektan, kao i da svaki od njih sadrži ispravne podatke, ali mi ćemo pretpostaviti da su podaci korektno formatirani u datoteci. Drugim rečima, element na indeksu 0 predstavlja ime kursa, dok element na indeksu 1 predstavlja broj ESPB bodova. Zbog toga, neophodno je konvertovati drugi element u celi broj, za šta, ponovo, postoji koristan metod klase `QString`.

```
CourseRAII() {
    ...

    QTextStream in(&m_file);
    auto line = in.readLine();
```

```
    while (!line.isNull()) {
        const auto stringArr = line.split(' ');
        const auto title = stringArr[0];
        const auto ects = stringArr[1].toUInt();
        ...

        line = in.readLine();
    }
}
```

Preostalo je da kreiramo jedan deljeni pokazivač na kurs čiji podaci su parsirani i da ga sačuvamo u vektoru deljenih pokazivača. Za kreiranje deljenog pokazivača možemo koristiti statički metod `create` klase `QSharedPointer<T>` koji funkcioniše poput funkcije `std::make_shared<T>` iz standardne biblioteke. Kompletan kod konstruktora je dat u nastavku.

```
CourseRAII() {
    m_file.setFileName("courses.txt");
    if (!m_file.open(QFile::ReadWrite | QFile::Text)) {
        throw CannotOpenFileException("courses.txt");
    }

    QTextStream in(&m_file);
    auto line = in.readLine();
    while (!line.isNull()) {
        const auto stringArr = line.split(' ');
        const auto title = stringArr[0];
        const auto ects = stringArr[1].toUInt();
        const auto course =
            QSharedPointer<Course>::create(title, ects);
        m_courses.push_back(course);

        line = in.readLine();
    }
}
```

Destruktor ove klase radi inverznu operaciju, odnosno, otvara datoteku `courses.txt` za čitanje, čime će njen sadržaj biti inicijalno obrisan, a zatim, za svaki kurs iz vektora generiše po jednu liniju podataka i upisuje je u otvorenu datoteku. Naravno, format ispisa mora odgovarati formatu kojim čitamo podatke, tako da

u svakoj liniji ispisujemo naziv kursa, karakter razmaka, broj bodova i karakter za novi red. Za ove potrebe, ilustrovaćemo novu vrstu petlje koja je dostupna isključivo u Qt biblioteci. Petlja `foreach` predstavlja makro za iteriranje kroz elemente, pre svega, Qt kolekcija. Ona se sastoji od tri elementa: naziva promenljive koja sadrži jedan element kolekcije, naziva kolekcije i tela petlje. Primetimo da su element i kolekcija odvojeni karakterom zapete.

```
~CourseRAII() {
    QFile file(m_filename);
    if (!file.open(QFile::WriteOnly | QFile::Text)) {
        return;
    }

    QTextStream out(&file);
    foreach (const auto &course, m_courses) {
        out << course->m_title << ' ' << course->m_ects << '\n';
    }
}
```

Pređimo sada na datoteku `Student.hpp` u kojoj ćemo implementirati strukture za upravljanje podacima o studentima. Čitaocima bi do sada trebalo da bude jasno na koji način treba implementirati strukture `Student` i `StudentRAII`, s obzirom da oni prate istu konceptualnu zamisao kao i strukture koje smo do sada implementirali. Zbog toga, savetujemo da čitaoci prvo pokušaju da ih implementiraju pre nego što pogledaju kompletno rešenje koje dajemo na kraju zadatka.

Funkcija `main` je konceptualno identična kao ona koju smo implementirali u rešenju zadatka 6, te ćemo ovde diskutovati isključivo o razlikama između te implementacije i ovog rešenja. Prvo, važno je napomenuti da Qt biblioteka ne raspolaže globalnim objektima za rad sa standardnim ulazom i izlazima. Zbog toga, neophodno je konstruisati tekstualne tokove koje će naš program koristiti. To je moguće uraditi prosleđivanjem *hendlera* na standardni ulaz i izlaz, `stdin` i `stdout`, konstruktoru klase `QTextStream`, zajedno sa odgovarajućim zastavicama za tip toka.

```
#include <QTextStream>

int main() {
    // Kreiranje ulaznog/izlaznog toka za standardni ulaz/izlaz
    QTextStream cin(stdin, QFile::ReadOnly);
```

```
    QTextStream cout(stdout, QFile::WriteOnly);

    ...
}
```

Zatim, definišemo `try-catch` blok u kojem pokušavamo sa izvršavanjem opisanog toka programa. Tok programa započinjemo konstrukcijom objekata RAII klasa koji upravljaju kursevima i studentima.

```
int main() {
    // Kreiranje ulaznog/izlaznog toka za standardni ulaz/izlaz
    QTextStream cin(stdin, QFile::ReadOnly);
    QTextStream cout(stdout, QFile::WriteOnly);

    try {
        // Ucitavanje kurseva i studenata iz datoteke
        CourseRAII coursesRAII;
        StudentRAII studentsRAII;

        ...
    } catch (const CannotOpenFileException &e) {
        cout << e.m_what << '\n'; cout.flush();
    }

    return 0;
}
```

Ostatak funkcije `main` je (gotovo) identičan poznatoj implementaciji. Primetimo naredne razlike:

- Nakon svakog ispisivanja na standardni izlaz, moramo pozvati metod `flush` kako bi se podaci zaista prikazali na standardnom izlazu. Ovo moramo raditi zbog toga što ne postoji ekvivalent `std::endl` objektu iz standardne biblioteke.

- Podatke o novim kursevima i studentima koje dobijamo od korisnika više ne skladištimo u vektorima u okviru funkcije `main`, već u RAII objektima, s obzirom da želimo da sačuvamo i te nove kurseve, odnosno, studente u datoteke.

- Upisivanje studenata na kurseve radimo tek nakon što učitamo i nove kurseve i nove studente od korisnika. Ovo ima više smisla raditi sada kada, pored studenata koje učitavamo od korisnika, imamo i studente čiji podaci su sačuvani u datoteci `students.txt`.

```
int main() {
    // Kreiranje ulaznog/izlaznog toka za standardni ulaz/izlaz
    QTextStream cin(stdin, QFile::ReadOnly);
    QTextStream cout(stdout, QFile::WriteOnly);

    try {
        // Ucitavanje kurseva i studenata iz datoteke
        CourseRAII coursesRAII;
        StudentRAII studentsRAII;

        // Ucitavanje kurseva od korisnika
        cout << "Unesite broj kurseva:\n"; cout.flush();
        unsigned k;
        cin >> k;

        for (auto i = 0u; i < k; ++i) {
            cout << "Unesite naziv kursa: \n"; cout.flush();
            QString title;
            cin >> title;

            cout << "Unesite broj bodova: \n"; cout.flush();
            unsigned ects;
            cin >> ects;

            const auto course =
                QSharedPointer<Course>::create(title, ects);
            coursesRAII.m_courses.push_back(course);
        }

        // Ucitavanje studenata od korisnika
        cout << "Unesite broj studenata:\n"; cout.flush();
        unsigned n;
        cin >> n;
```

```
for (auto i = 0u; i < n; ++i) {
    cout << "Unesite indeks studenta: \n"; cout.flush();
    QString index;
    cin >> index;

    cout << "Unesite ime studenta: \n"; cout.flush();
    QString name;
    cin >> name;

    const auto student =
        QSharedPointer<Student>::create(index, name);
    studentsRAII.m_students.push_back(student);
}

// Upisivanje studenata na kurseve
foreach (const auto &student, studentsRAII.m_students) {
    foreach (const auto &course, coursesRAII.m_courses) {
        cout
            << "Da li zelite da upisete studenta "
            << student->m_name
            << " (" << student->m_index << ")"
            << " na kurs " << course->m_title
            << " (" << course->m_ects << ")?"
            << " [da/ne]\n"; cout.flush();

        QString response;
        cin >> response;
        if (response != "da") {
            continue;
        }

        student->m_courses.push_back(course);
    }
}

// Izlistavanje informacija o studentima
foreach (const auto &student, studentsRAII.m_students) {
    cout
        << "Student " << student->m_name
```

```
                << " (" << student->m_index << ")"
                << " ima naredne upisane kurseve:\n"; cout.flush();

            foreach (const auto &course, student->m_courses) {
                cout
                    << '\t' << course->m_title
                    << " (" << course->m_ects
                    << ")\n"; cout.flush();
            }
        }
    } catch (const CannotOpenFileException &e) {
        cout << e.m_what << '\n'; cout.flush();
    }

    return 0;
}
```

Kompletan kod je dat u rešenju 9.

Zadatak 11: RAII study programs

Implementirati rešenje zadatka 7 korišćenjem RAII idioma. Pri pokretanju programa, učitati podatke o studijskim programima i studentima iz tabela `STUDYPROGRAMS` i `STUDENTS` u SQLite bazi podataka `FACULTY`, ukoliko postoje. Na kraju rada programa, sačuvati izmenjene podatke u date tabele.

U ovom zadatku ćemo predstaviti osnovnu ideju za upravljanje bazama podataka kao dinamičkim resursima. Većina sistema za upravljanje bazama podataka zahteva da aplikacija ostvari konekciju sa bazom podataka i da tu konekciju održava „živom" sve dok ima potrebe za upravljanjem podacima u okviru te baze podataka. Dodatno, aplikacija bi trebalo da eksplicitno naznači kada više ne želi da radi sa bazom podataka raskidanjem konekcije sa njom. Ovime smo zapravo opisali procedure „zauzeća" i „oslobađanja" baze podataka kao dinamičkog resursa. Implementirajmo ove procedure kao deo RAII idioma.

U datoteci `DatabaseManager.hpp` kreiramo novu strukturu `DatabaseManager` koja će implementirati RAII idiom za upravljanje jednom konekcijom ka nekoj

bazi podataka. Za te potrebe možemo koristiti klasu `QSqlDatabase`, čije se instance mogu dobiti pozivom statičkog metoda `addDatabase` te klase. Argument koji se prosleđuje jeste drajver koji se koristi za povezivanje na odgovarajući SUBP. U ovom primeru ćemo koristiti SQLite SUBP zbog svoje lake dostupnosti[8].

```
struct DatabaseManager {
    QSqlDatabase m_database;

    DatabaseManager() {
        m_database = QSqlDatabase::addDatabase("QSQLITE");
        m_database.setHostName("localhost");
        m_database.setUserName("student");
        m_database.setPassword("abcdef");
        m_database.setDatabaseName("FACULTY");

        if (m_database.open()) {
            qDebug() << "Uspesna konekcija sa BP";
            const auto hasTransactionSupport =
                m_database.driver()->hasFeature(QSqlDriver::Transactions);
            qDebug() << "Konekcija"
                << (hasTransactionSupport ? "podrzava" : "ne podrzava")
                << "transakcije";
        }
        else {
            qDebug() << "Neuspesna konekcija sa BP";
        }
    }

    ~DatabaseManager() {
        if (m_database.isOpen()) {
            m_database.close();
            qDebug() << "Zatvaram konekciju sa BP";
        }
    }
};
```

[8]Qt biblioteka podržava veliki broj SUBP i za neke od njih je potrebno prevesti odgovarajuće *dodatke*. Više informacija je moguće pronaći na adresi `https://doc.qt.io/qt-6/sql-driver.html`.

Metod `open` klase `QSqlDatabase` otvara konekciju ka bazi podataka čije informacije su specifikovane metodima `setHostName`, `setUserName` i sl. Nad `QSqlDatabase` objektom možemo dobiti razne informacije o SUBP preko njegovog drajvera, poput toga da li podržava transakcije, kao što je ilustrovano u fragmentu koda iznad.

Raskidanje konekcije sa bazom podataka se izvršava pozivom metoda `close`, kao što je ilustrovano u destruktoru strukture `DatabaseManager` iznad.

Nastavimo dalje, opisujući strukture podataka iz zadatka. Podaci o studentima su jednostavniji, te ćemo njih predstaviti prvo.

```
#include <QString>

struct Student {
    QString m_index;
    QString m_name;

    Student(QString index, QString name) {
        m_index = index;
        m_name = name;
    }
};
```

Porazmislimo sada šta je sve neophodno za implementaciju RAII idioma u cilju upravljanja objektima prikazane strukture `Student`:

1. Pre nego što bilo šta uradimo, neophodno je da imamo ostvarenu konekciju ka bazi podataka `FACULTY`. Za to ćemo koristiti opisanu strukturu `DatabaseManager`. S obzirom da je ovaj podatak neophodan za dohvatanje podataka o studentima, on predstavlja dobar kandidat za argument konstruktora RAII strukture.

2. Zatim, treba dohvatiti podatke iz tabele `STUDENTS` iz baze podataka. Međutim, šta ako u bazi podataka ne postoji data tabela? Ispostavlja se da, ako datoteka koja odgovara SQLite bazi podataka ne postoji na disku (u radnom direktorijumu aplikacije), onda će ona biti kreirana, ali biće prazna, odnosno, neće sadržati nijednu tabelu. Dakle, trebalo bi prvo proveriti da li odgovarajuća tabela postoji i, ako ne postoji, kreirati je.

3. Tek sada je moguće pročitati sadržaj iz odgovarajuće tabele i kreirati odgovarajuće objekte strukture `Student`.

Tačke 2 i 3 iznad predstavljaju dobre kandidate za metode strukture `StudentRAII` koji će biti pozvani u konstruktoru.

```
#include <QSharedPointer>
#include <QVector>

#include "DatabaseManager.hpp"

// ...

struct StudentRAII {
    QSharedPointer<DatabaseManager> m_db;
    QVector<QSharedPointer<Student>> m_students;

    // ...

    StudentRAII(QSharedPointer<DatabaseManager> db) {
        m_db = db;
        create_table_if_not_exists();
        read_students_from_table();
    }
}
```

Implementirajmo prvo metod `create_table_if_not_exists`. Proveravanje da li tabela postoji definisana u katalogu baze podataka moguće je ispitivanjem sadržaja `QStringListe` koja se dobija kao povratna vrednost metoda `tables` iz klase `QSqlDatabase`. Opcioni parametar tipa `QSql::TableType` predstavlja enumerator koji određuje tip tabela čiji će nazivi biti dohvaćeni u rezultatu. Podrazumevana vrednost je `QSql::Tables`, odnosno, biće dohvaćene sve tabele vidljive za prijavljenog korisnika, ali ne i pogledi niti sistemske tabele.

```
void create_table_if_not_exists() {
    if (!m_db->m_database.tables().contains("STUDENTS")) {
        // ...
    }
}
```

Ako data lista niski ne sadrži tabelu naziva `STUDENTS`, onda ju je neophodno kreirati. Za te potrebe možemo koristiti klasu `QSqlQuery` koja nudi podršku za rad sa *dinamičkim* SQL naredbama. Objekat ove klase se konstruiše

prosleđivanjem reference na objekat `QSqlDatabase`. Sama SQL naredba, koja se zadaje kao niska, izvršava se nad tom bazom podataka prosleđivanjem metodu `exec` nad objektom klase `QSqlQuery`. Ovaj metod vraća `true` ako je naredba uspešno izvršena, odnosno, `false` u suprotnom. Tekstualni oblik SQL naredbe mora sadržati validnu sintaksu za dati SUBP nad kojim se izvršava. Ukoliko je izvršavanje SQL naredbe prošlo neuspešno i želimo da ispitamo do koje greške je došlo, možemo koristiti metod `lastError` čija je povratna vrednost objekat klase `QSqlError`. Ova klasa se može koristiti za dohvatanje raznih informacija o greški koja je nastala, kao što je njen SQL kod i tekstualni opis greške. Ove informacije su nam dovoljne za finaliziranje implementacije trenutnog metoda.

```
#include <QSqlQuery>
#include <QSqlError>

// ...

void create_table_if_not_exists() {
    if (!m_db->m_database.tables().contains("STUDENTS")) {
        QSqlQuery query(m_db->m_database);
        QString sql =
            "CREATE TABLE STUDENTS ( "
            "   STUDINDEX VARCHAR(9) PRIMARY KEY NOT NULL, "
            "   NAME VARCHAR(50) NOT NULL "
            ")";
        if (!query.exec(sql)) {
            throw query.lastError();
        }
    }
}
```

Metod `read_students_from_table` će pratiti istu ideju kao i prethodni metod, sa dodatnom razlikom u tipu SQL naredbe koja se izvršava. Naime, s obzirom da broj redova u tabeli može biti proizvoljan, neophodno je koristiti kursor za prolazak kroz tabelu. Naime, kada se SQL naredba koja je tipa `SELECT` izvrši nad bazom podataka, SUBP konstruiše objekat u bazi podataka koji se naziva *kursor* i koji se koristi za prolazak kroz rezultujuću tabelu `SELECT` naredbe. Kursorom možemo dohvatati rezultate iz tabele po principu red-po-red. Kada se konstruiše kursor nad rezultujućom tabelom u bazi podataka, on se *pozicionira* nad specijalnim, nevalidnim redom koji se nalazi „pre" prvog reda u tabeli. Da

bismo mogli da čitamo podatke nad nekim redom, potrebno je prvo da *pozicioniramo* kursor na taj red, a zatim da čitamo vrednosti iz odgovarajućih kolona pomoću tog kursora.

Klasa `QSqlQuery` je dovoljno moćna da podržava rad sa kursorima. Pozicioniranje na *naredni* red u rezultujućoj tabeli se izvršava pozivanjem metoda `next`. Metod vraća `true` ukoliko je kursor uspešno pozicioniran na naredni red, a `false` inače. S obzirom da se kursor nakon izvršavanja `SELECT` naredbe nalazi na redu *pre* prvog reda, neophodno je pozvati ovaj metod kako bi se kursor pozicionirao na početak tabele, pre prvog čitanja podataka.

```
void read_students_from_table() {
    QSqlQuery query(m_db->m_database);
    QString sql = "SELECT * FROM STUDENTS";
    if (!query.exec(sql)) {
        throw query.lastError();
    }

    while (query.next()) {
        // ...
    }
}
```

Čitanje podataka iz reda na kojem je kursor pozicioniran se vrši pozivom metoda `value` nad objektom `QSqlQuery`. Ovaj metod ima dva preopterećenja:

1. Preopterećenje koje prihvata jedan argument tipa `int` dohvata podatak iz kolone rezultujuće tabele čiji redni broj odgovara tom argumentu, pri čemu indeksiranje počinje od 0. Ovo može biti korisno ukoliko rezultujuća tabela `SELECT` naredbe nema imenovane kolone ili ukoliko razvijač želi da prolazi petljom kroz kolone rezultata. Korišćenje ovog preopterećenja se ne preporučuje za SQL naredbe oblika `SELECT * ...` s obzirom da je redosled kolona nedefinisan u tom slučaju.

2. Preopterećenje koje prihvata jedan argument tipa `const QString &` dohvata podatak iz kolone rezultujuće tabele čiji naziv odgovara tom argumentu. Vredno je napomenuti da je ovo preopterećenje nešto manje efikasnije od prvog.

Oba preopterećenja vraćaju objekat klase `QVariant` koja predstavlja nešto poput unije. Naime, objekti ove klase mogu da čuvaju vrednosti različitih tipova,

ali u datom trenutku, najviše jednog tipa. Da bismo dobili podatak određenog tipa, na raspolaganju su metodi poput `toInt`, `toString` i drugi.

Pogledajmo sada kompletnu definiciju metoda `read_students_from_table` koji prolazi kroz redove tabele `STUDENTS`, dohvatajući informacije o indeksu i nazivu studenta kao niske, a zatim konstruiše deljene pokazivače na objekte strukture `Student` i skladišti ih u vektor `m_students` – atribut RAII klase.

```
void read_students_from_table() {
    QSqlQuery query(m_db->m_database);
    QString sql = "SELECT * FROM STUDENTS";
    if (!query.exec(sql)) {
        throw query.lastError();
    }

    while (query.next()) {
        const auto index = query.value("STUDINDEX").toString();
        const auto name = query.value("NAME").toString();

        auto student =
            QSharedPointer<Student>::create(index, name);
        m_students.push_back(student);
    }
}
```

Pozabavimo se sada implementacijom oslobađanja resursa. Kako bi podaci iz klase biti trajno zapamćeni, potrebno ih je upisati u tabelu `STUDENTS` pre oslobađanja. Ovde je potrebno voditi računa o unosu podataka. Naime, ukoliko bismo samo iterirali kroz svaki objekat i upisivali ga u bazu podataka, naišli bismo na grešku već pri prvom upisu. Razlog za ovo je taj što podaci na početku vektora `m_students` već postoje u datoj tabeli, pa bi se uneli duplikati. Na osnovu jedinstvenog ograničenja postavljenog preko primarnog ključa – indeksa studenta – SUBP bi prijavio grešku.

Mi ćemo ovaj problem rešiti prvo brisanjem kompletne tabele `STUDENTS`, a zatim unošenjem jednog po jednog reda. Alternativno, mogli bismo čuvati neku vrstu indikatora koji objekti već postoje u bazi, a koji ne, pa unositi samo one koji ne postoje u bazi podataka. Još jedna alternativa je čuvanje dva vektora – jednog koji čuva objekte koji su učitani pri inicijalizaciji i drugog koji skladišti objekte novih studenata – pa bi se upisivanje vršilo samo nad objektima iz drugog vektora. Ostavljamo čitaocima da implementiraju ove alternativne mehanizme za vežbu.

Naša implementacija indikuje konstrukciju dodatna dva metoda strukture `StudentsRAII` koja će biti pozvana u destruktoru.

Prvi od njih, `delete_from_table` briše sadržaj tabele `STUDENTS`. Ova operacija se identično implementira kao i kreiranje iste tabele, te ćemo prikazati samo kod. Primetimo da ovoga puta ne ispaljujemo izuzetak u slučaju greške. Naime, ispaljivanje izuzetka iz destruktora klase se smatra vrlo lošom praksom, te ćemo mi to izbegavati. Umesto toga, metodi će vraćati Bulove vrednosti koje će indikovati uspešnost operacije.

```
bool delete_from_table() {
    QSqlQuery query(m_db->m_database);
    QString sql = "DELETE FROM STUDENTS";
    if (!query.exec(sql)) {
        const auto e = query.lastError();
        qDebug()
            << "SQLCODE: " << e.nativeErrorCode()
            << ", SQLERR: " << e.text()
            << ", SQL:" << sql;

        return false;
    }
    return true;
}
```

Za unos podataka u odgovarajuću tabelu implementiraćemo metod `insert_students`. Ovaj metod treba da iterira kroz vektor dinamičkih pokazivača na studente, i da za svakog od njih unese po jedan novi red u tabelu. I ovo je operacija koju bismo sa dosadašnjim znanjem mogli da implementiramo – mogli bismo da formulišemo nisku koja predstavlja tekstualnu reprezentaciju `INSERT` naredbe sa odgovarajućim podacima iz svakog objekta, pa da za svaku tu nisku konstruišemo izvršivi upit u bazi podataka i da ga, konačno, izvršimo. Po konstrukciji prethodne rečenice, čitaocima može zvučati kao da je opisana operacija neefikasna i zaista je i tako. Problem je u tome što, pri radu sa dinamičkim SQL naredbama, SUBP mora da izvrši dve operacije – jedna je *pripremanje* upita za izvršavanje (parsiranje niske koja sadrži SQL naredbu, kreiranje *pripremljene SQL naredbe*, kreiranje plana izvršavanja naredbe, itd.), a druga je *izvršavanje* pripremljene naredbe sa datim ulazima (i, eventualno, pripreme izlaza u slučaju `SELECT` naredbe). Ukoliko se ista SQL naredba izvršava n puta za (potencijalno) različite ulaze, to znači n pripremanja i n izvršavanja te naredbe. Izvršavanja

sigurno ne može biti manje, ali pripremanja može, i to tačno jedno! Sve što je potrebno uraditi u programu jeste odvojiti faze pripremanja i izvršavanja.

Pripremanje SQL naredbe iz njenog tekstualnog oblika se vrši pozivom metoda `prepare` nad objektom klase `QSqlQuery`. Metod zahteva nisku koja, ukoliko SQL naredba ima ulaznih podataka, umesto konkretnih podataka sadrži *parametarske oznake*. Ove oznake predstavljaju „rupe" koje će SUBP zameniti konkretnim vrednostima u fazi izvršavanja upita. Parametarske oznake se navode kao identifikatori koja prethodni karakter dvotačka (`:`). Na primer, konkretna SQL naredba `"INSERT␣INTO␣TABELA␣VALUES␣(1,␣'abc')"` može se zapisati sa parametarskim oznakama kao `"INSERT␣INTO␣TABELA␣VALUES␣(:broj,␣:niska)"`. Pogledajmo sada kako se može pripremiti naredba za unos novog reda u tabelu `STUDENTS`.

```
bool insert_students() {
    QSqlQuery query(m_db->m_database);
    QString sql = "INSERT INTO STUDENTS VALUES (:index, :name)";
    if (!query.prepare(sql)) {
        const auto e = query.lastError();

        qDebug()
            << "SQLCODE: " << e.nativeErrorCode()
            << ", SQLERR: " << e.text()
            << ", SQL:" << sql;
        return false;
    }

    // ...

    return true;
}
```

Pre izvršavanja pripremljene naredbe, potrebno je postaviti vrednosti ovih parametarskih oznaka pozivom metoda `bindValue`, čiji je prvi argument niska sa nazivom parametarske oznake čija se vrednost postavlja, a drugi argument predstavlja samu vrednost. Ovo nam je dovoljno znanja da kompletiramo implementaciju metoda `insert_students`.

```
bool insert_students() {
    QSqlQuery query(m_db->m_database);
    QString sql = "INSERT INTO STUDENTS VALUES (:index, :name)";
```

```
    if (!query.prepare(sql)) {
        const auto e = query.lastError();

        qDebug()
            << "SQLCODE: " << e.nativeErrorCode()
            << ", SQLERR: " << e.text()
            << ", SQL:" << sql;
        return false;
    }
    foreach (const auto &student, m_students) {
        query.bindValue(":index", student->m_index);
        query.bindValue(":name", student->m_name);
        if (!query.exec()) {
            const auto e = query.lastError();

            qDebug()
                << "SQLCODE: " << e.nativeErrorCode()
                << ", SQLERR: " << e.text()
                << ", SQL:" << sql;
            return false;
        }
    }
    return true;
}
```

Destruktor se sada može implementirati na sledeći način.

```
~StudentRAII() {
    const auto isDeleted = delete_from_table();
    if (!isDeleted) {
        return;
    }

    const auto isInserted = insert_students();
    if (!isInserted) {
        return;
    }

    qDebug()
```

```
        << "Uspesno su sacuvane informacije o studentima u BP";
}
```

Postoji jedan važan problem sa trenutnom implementacijom skladištenja podataka. Naime, postavlja se pitanje kako će tabela `STUDENTS` izgledati ukoliko, na primer, operacija brisanja prođe uspešno, ali operacija upisivanja ne. Ili, na primer, ukoliko upisivanje „pukne" na nekom objektu. S obzirom da, podrazumevano, SQLite radi po režim *automatskog potvrđivanja izmena*, sve izmene posle svake SQL naredbe koja se izvrši nad bazom podataka bivaju trajno sačuvane. Drugim rečima, u slučaju da se izvrši brisanje, ali ne i upisivanje, tabela će ostati prazna. U drugom hipotetičkom (ali vrlo realnom) slučaju, tabela će sadržati podatke samo o nekim studentima, ali ne i o svim. Štaviše, nakon završetka aplikacije, podaci o studentima koji nisu sačuvani biće trajno izgubljeni. Ovo je veliki problem u radu aplikacije. Trebalo bi obezbediti da se obe operacije izvrše ili u celosti ili uopšte ne[9].

Na sreću, SQLite SUBP podržava koncept *transakcija* koji nam obezbeđuje upravo opisan scenario. Transakciju nad bazom podataka započinjemo pozivanjem metoda `transaction` nad objektom klase `QSqlDatabase` koji vraća `true` ukoliko je transakcija uspešno započeta, a `false` inače. Započinjanjem transakcije se isključuje režim automatskog potvrđivanja izmena, što znači da izmene neće biti trajno zapamćene. Ipak, od programera se zahteva da eksplicitno završi započetu transakciju pozivom jednog od metoda `commit` ili `rollback`. Prvim metodom signaliziramo da je sve prošlo uspešno i da izmene mogu biti trajno zapamćene. Drugim metodom signaliziramo da nešto nije bilo u redu i da se sve izmene od početka transakcije ponište. Oba metoda će završiti započetu transakciju, čime se ponovo uključuje režim automatskog potvrđivanja izmena. Valjalo bi napomenuti da za neke SUBP ovi metodi neće biti uspešni ukoliko postoji aktivan kursor. Zbog toga, ukoliko želimo da završimo transakciju nakon obrade kursorom, neophodno je pozvati metod `finish` nad objektom klase `QSqlQuery` kako bismo naznačili da je odgovarajući kursor neaktivan (ili, alternativno, uništiti objekat klase `QSqlQuery`).

Pogledajmo sada koje izmene je neophodno uraditi u našoj klasi kako bi implementirala željeni rad sa transakcijama. Prvo, dopunjavamo destruktor koji započinje novu transakciju. Ukoliko transakcija ne može biti započeta, odustajemo od bilo kakve druge operacije. Zatim, na kraju destruktora, samo ako su obe operacije prošle uspešno, potvrđujemo izmene.

[9]Zapravo, ovo je tek prvi korak. Idealno, u slučaju da u bilo kom trenutku dođe do greške, bilo bi bolje da aplikacija pokuša ponovo sa celom operacijom, zatraži dalju akciju od korisnika ili nešto treće. Mi ćemo se ovde zaustaviti na najjednostavnijoj varijanti – „sve ili ništa".

```
~StudentRAII() {
    if (!m_db->m_database.transaction()) {
        return;
    }

    const auto isDeleted = delete_from_table();
    if (!isDeleted) {
        return;
    }

    const auto isInserted = insert_students();
    if (!isInserted) {
        return;
    }

    m_db->m_database.commit();
    qDebug()
        << "Uspesno su sacuvane informacije o studentima u BP";
}
```

Druga izmena jeste poništavanje izmena u metodima `delete_from_table` i `insert_students` na svim mestima gde može doći do problema, odnosno, tamo gde obrađujemo SQL greške.

```
bool delete_from_table() {
    QSqlQuery query(m_db->m_database);
    QString sql = "DELETE FROM STUDENTS";
    if (!query.exec(sql)) {
        const auto e = query.lastError();

        m_db->m_database.rollback();
        qDebug()
            << "SQLCODE: " << e.nativeErrorCode()
            << ", SQLERR: " << e.text()
            << ", SQL:" << sql;

        return false;
    }
    return true;
```

```
}

bool insert_students() {
    QSqlQuery query(m_db->m_database);
    QString sql = "INSERT INTO STUDENTS VALUES (:index, :name)";
    if (!query.prepare(sql)) {
        const auto e = query.lastError();

        m_db->m_database.rollback();

        qDebug()
            << "SQLCODE: " << e.nativeErrorCode()
            << ", SQLERR: " << e.text()
            << ", SQL:" << sql;
        return false;
    }
    foreach (const auto &student, m_students) {
        query.bindValue(":index", student->m_index);
        query.bindValue(":name", student->m_name);
        if (!query.exec()) {
            const auto e = query.lastError();

            m_db->m_database.rollback();

            qDebug()
                << "SQLCODE: " << e.nativeErrorCode()
                << ", SQLERR: " << e.text()
                << ", SQL:" << sql;
            return false;
        }
    }
    return true;
}
```

Ovime smo završili obradu podataka o studentima. Na sličan način se implementira obrada smerova. Taj deo implementacije, zajedno sa funkcijom `main` ostavljamo čitaocima za vežbu. Naravno, ukoliko postoji potreba za time, čitaoci mogu konsultovati kompletno rešenje.

Kompletan kod je dat u rešenju 10.

2.3 Stabla dinamičkih objekata

Još jedna tehnika za upravljanje dinamičkim objektima jesu stabla dinamičkih objekata. Osnovna ideja jeste da svaki objekat održava spisak dinamičkih objekata koji predstavljaju njegovu decu (i, eventualno, da svako dete održava informaciju o svom roditelju[10]). Kreiranjem veza roditelj-dete između dinamičkih objekata moguće je implementirati jednostavno oslobađanje objekata. Naime, prilikom uništavanja nekog objekta, dovoljno je proći kroz spisak njegove dece i uništiti ih. Ukoliko su ta deca imala svoju decu, onda će i ona biti automatski uništena. Ovime se postiže da je jedino potrebno eksplicitno uništiti koren stabla dinamičkih objekata. U slučaju da se koren nalazi na steku neke funkcije (na primer, `main`) kao objekat automatskog životnog veka, onda čak nije neophodno ni njega eksplicitno uništiti.

Qt biblioteka vrlo intenzivno koristi opisanu tehniku pomoću klase `QObject`. Naime, svim `QObject`ima je moguće dodeliti roditelja prosleđivanjem pokazivača na roditeljski objekat `QObject␣*` konstruktoru ili pozivom metoda `setParent`. Pogledajmo naredni primer.

```
#include <QObject>
#include <QDebug>

int main() {
    QObject parent;

    QObject *child1 = new QObject(&parent), *child2 = new QObject();
    child2->setParent(&parent);

    qDebug()
        << "Broj dece u roditeljskom objektu je:"
        << parent.children().size();

    return 0;
```

[10]Drugi smer veze može biti značajan za potrebe razmene dece između roditeljskih objekata u stablu. U daljem tekstu nećemo ulaziti u detalje implementacije ovog mehanizma, već ćemo samo prikazani jednu postojeću implementaciju u biblioteci Qt.

```
}
```

Kreiramo roditeljski `QObject` kao objekat automatskog životnog veka. Zatim, prilikom konstrukcije `child1` dinamičkog objekta, prosleđujemo adresu roditelja konstruktoru, čime povezujemo taj objekat kao dete objekta `parent`. Dodatno, kreiramo dinamički objekat `child2` koji inicijalno nema roditelja. Ipak, u narednom redu postavljamo roditelja pozivom metoda `setParent`. Kao što vidimo u ispisu u nastavku, roditeljski objekat dobija informaciju da ima odgovarajući broj dece, bez obzira na način na koji su mu dodeljeni.

Standardni izlaz

```
Broj dece u roditeljskom objektu je: 2
```

U ovakvoj strategiji je važno voditi računa o tome da neki objekat ne bude dete više roditelja jer bi u tom slučaju dolazilo do višestrukog oslobađanja iste memorije, čime bi se proizvela greška. Na sreću, `QObject`i umeju da vode računa o ovome.

```
#include <QObject>
#include <QDebug>

int main() {
    QObject parent1;
    QObject parent2;

    auto child1 = new QObject(&parent1);
    auto child2 = new QObject(&parent2);

    qDebug()
        << "Broj dece u objektu parent1 je:"
        << parent1.children().size();
    qDebug()
        << "Broj dece u objektu parent2 je:"
        << parent2.children().size();

    qDebug() << "----------------------------------------";
    qDebug() << "Dodajemo child kao dete objektu parent1";
    child2->setParent(&parent1);
```

```
    qDebug() << "----------------------------------------";

    qDebug()
        << "Broj dece u objektu parent1 je:"
        << parent1.children().size();
    qDebug()
        << "Broj dece u objektu parent2 je:"
        << parent2.children().size();

    return 0;
}
```

Standardni izlaz

```
Broj dece u objektu parent1 je: 1
Broj dece u objektu parent2 je: 1
----------------------------------------
Dodajem child2 kao dete objektu parent1
----------------------------------------
Broj dece u objektu parent1 je: 2
Broj dece u objektu parent2 je: 0
```

Stabla dinamičkih objekata su nezaobilazan alat u razvoju aplikacija koje koriste Qt biblioteku, te ih je važno razumeti. Ipak, njihova puna izražajnost će nam značiti tek kad budemo razumeli tehnike objektno-orijentisanog razvoja aplikacija, što će nam biti tema narednog poglavlja.

Beleške

U ovom poglavlju smo opisali neke osnovne tehnike upravljanja dinamičkim resursima i skrenuli pažnju čitaocima na neke od čestih problema koji se javljaju u radu sa dinamičkim resursima. Iako je veliki akcenat bio stavljen na upravljanje dinamičkim resursima u programskom jeziku C++, dinamički resursi se javljaju i u drugim programskim jezicima, bibliotekama i dr. Čak i jezici koji poseduju automatsko oslobađanje memorije sakupljačima otpadaka mogu uzrokovati probleme pri radu sa, na primer, raznim vrstama tokova (datotekama, soketima, i sl.) ili bazama podataka. RAII idiom može pomoći i u takvim

okruženjima, te se čitaocima sugeriše da razmisle o alternativnim načinima na koje mogu implementirati ovaj idiom u svojim aplikacijama.

Glava 3

Objektno-orijentisane tehnike razvoja

3.1 UML dijagram klasa

3.2 Osnovni koncepti objektno-orijentisane analize i dizajna

Zadatak 12: Fraction

...

3.3 Sakrivanje detalja implementacije unutrašnjim klasama

Zadatak 13: Linked list

...

3.4 Specijalizacija i generalizacija

Zadatak 14: Student

...

3.5 Odnosi između klasa

Zadatak 15: Vehicle park

...

3.6 Kreiranje složenih hijerarhija klasa višestrukim nasleđivanjem

Primer koji izdvaja zajedničko ponašanje dve klase koje nemaju veze jedna sa drugom u zasebnu klasu koju obe klase nasleđuju.

3.6.1 Problem dijamanta

Beleške

Glava 4

Aplikacije sa grafičkim korisničkim interfejsom

4.1 Dizajn grafičkog korisničkog interfejsa

O različitim kontrolama i raspoređivanju kontrola (kontejnerima).

- Mini zadatak koji ima za cilj dizajniranje složenijeg korisničkog interfejsa, ali koji nema funkcionalnosti. Obavezno koristiti sve vrste raspoređjivanja (hbox, vbox, form i grid) i neke česte kontrole, poput, `QLabel`, `QProgressBar`, `QPushButton`, `QLineEdit`, `QTextEdit`, `QRadioButton`, `QCheckBox`, `QListWidget`, `QTableWidget`, `QTreeWidget`, i kontejnere, poput `QGroupBox`.

4.2 Petlja događaja

O petlji događaja u Qt-ovim aplikacijama.

- Mini zadatak koji dopunjava funkcionalnosti kontrola iz gornjeg zadatka.

4.3 Implementiranje grafičkih aplikacija

O Qt-ovom radnom okviru grafičke scene (ima dosta toga na prezentaciji).

- Mini zadatak sa `QGraphicsItem` i nekim osnovnim funkcionalnostima: iscrtavanje, klik (selektovanje) i sl.
- Mini zadatak sa `QGraphicsObject` i zašto bi se koristio (za implementaciju elemenata grafičke scene koji imaju podršku za signale i slotove).

4.4 Animacije

4.4.1 Animiranje pomoću tajmera

- Mini primer sa Qt tajmerima i `advance` metodima iz `QGraphicsScene` i `QGraphicsItem`. Skrenuti pažnju da se objekti ne smeju brisati u istom ciklusu petlje događaja (i prikazati implementaciju metoda `QGraphicsScene::advance`). Možda pojednostaviti onaj primer sa miševima.

4.4.2 Radni okvir za animiranje

- Mini primer sa animiranjem svojstava objekata.
- Mini primer sa animiranjem pomoću konačnih automata.

Beleške

Glava 5

Polimorfizam

Ideja ovog poglavlja je da prikaže popularne tehnike programiranja koje koriste polimorfno ponašanje.

5.1 Hijerarhijski polimorfizam

5.1.1 Interfejsi kao sredstvo održavanja ugovora

Primer koji ima dve klase `WorkerA` i `WorkerB` koje komuniciraju preko zajedničkih interfejsa `IJob` i `IResult`. Klasa `WorkerA` instancira klasu `Job` koja nasleđuje `IJob` i prevazilazi odgovarajuće metode na neki način i šalje taj objekat klasi `WorkerB`. Klasa `WorkerB` na osnovu javnog interfejsa iz `IJob` instancira objekat klase `Result` koja nasleđuje `IResult` i vraća taj objekat klasi `WorkerA`. Klasa `WorkerA` koristi taj objekat preko javnog interfejsa. Ni u jednom trenutku `WorkerB` ne zna kako je posao implementiran, dok `WorkerA` ne zna kako je rezultat implementiran i to ni ne treba da znaju.

5.1.2 Entity-Component radni okvir

Nešto o ovom obrascu. Možda ovo ima smisla premestiti u arhitekturu?

5.2 Parametarski polimorfizam

Zadatak 16: Template functions

...

Zadatak 17: Template pair

...

Zadatak 18: Template linked list

...

Beleške

Glava 6

Refaktorisanje i svođenje problema

6.1 Smernice za refaktorisanje

Spisak predloga za prepoznavanje kada se nešto može refaktorisati, sa mini primerima.

Zadatak 19: Double code

Refaktorisati naredni kod.

```
function f1() {
    ...
}

function f2() {
    ...
}

f1();
f2();
```

Zadatak 20: Big function

Refaktorisati naredni kod.

```
function f() {
    ...
}

f();
```

Zadatak 21: Big class

Refaktorisati naredni kod.

```
class C {
    ...
};

C c;
c.f1();
c.f2();
c.f3();
```

6.2 Kolekcije standardne biblioteke

Ideja ove sekcije je da prikaže korišćenje standardne biblioteke za ilustrovanje nekih smernica za refaktorisanje.

Smisliti zadatke koji ilustruju upotrebu različitih kolekcija. Na primer, zašto bi negde imalo smisla koristiti vektor (brz pristup podacima?), jednostruko-povezanu listu (puno brisanja?), dvostruko-povezanu listu (implementacija undo-redo operacija?), mapa (rečnik poslova – videti zadatak 23), skup (?).

Zadatak 22: Chain

...

Umesto da se napiše nekoliko funkcija za upravljanje zahtevima koji pozivaju međusobno jedni druge, upravljanje se pretvara u lanac: https://refactoring.guru/design-patterns/chain-of-responsibility. Umesto direktne implementacije ovog obrasca

za projektovanje, mogu se zahtevi smeštati u vektora, pa se korišćenjem `accumulate` algoritma (ili nekog sličnog) provlači zahtev kroz lanac.

Zadatak 23: Tasks

...

Zadatak ilustruje kako korišćenje mape može zameniti gomilu `if-else` ili `switch` naredbi.

Skica rešenja: Definisati hijerarhiju klasa poslova. Apsktraktna bazna klasa `Task` ima čisto virtualni metod `do`. Klase koje nasleđuju ovu klasu su, na primer, `Cut`, `Copy`, `Paste` i one implementiraju metod `do` na svoj način. Aplikacija čuva mapu tipa `std::map<std::string, Task *>` i kada korisnik unese neku naredbu, aplikacija izvršava:

```
class Task {
    public: virtual void do(std::string ARGUMENT) = 0;
};
class Cut : public Task {
    public: void do(std::string ARGUMENT) override { ... }
};
class Copy : public Task {
    public: void do(std::string ARGUMENT) override { ... }
};
class Paste : public Task {
    public: void do(std::string ARGUMENT) override { ... }
};

std::map<std::string, Task *> tasks;
map["cut"] = new Cut();
map["copy"] = new Copy();
map["paste"] = new Paste();

void do_task(std::string ACTION, std::string ARGUMENT)
{
    tasks.at(ACTION)->do(ARGUMENT);
}
```

umesto da implementacija izgleda nalik sledećem:

```
void cut(std::string) { ... }
```

```
void copy(std::string) { ... }
void paste(std::string) { ... }

void do_task(std::string ACTION, std::string ARGUMENT)
{
    if (ACTION == "cut") {
        cut(ARGUMENT);
    }
    else if (ACTION == "copy") {
        copy(ARGUMENT);
    }
    else if (ACTION == "paste") {
        paste(ARGUMENT);
    }
}
```

6.3 Algoritmi standardne biblioteke

Ideja ove sekcije je da prikaže kako se problemi često mogu svesti na druge probleme za koje su rešenja već poznata.

Beleške

Glava 7

Razvoj vođen testovima

Za ovo poglavlje će verovatno biti dovoljno da se prekopira ono iz prezentacije koju sam napravio ove godine.

7.1 Jedinični testovi

O pisanju jediničnih testova. Biblioteka Catch2.

7.2 Principi razvoja vođenog testovima

O TDD-u sa primerom razvoja klase `BigInteger`.

Beleške

Glava 8

Konkurentno programiranje

8.1 Blokiranje izvršavanja

Na primer, skupe operacije blokiraju GKI.

8.2 Podela posla prema zadacima

Paralelizacija posla tako da svaka nit radi različit posao.

8.3 Podela posla prema podacima

Paralelizacija posla tako da svaka nit radi istu stvar nad podskupom podataka.

8.4 Sinhronizacija niti

Muteksi i mutexlocker-i, za sada. Možda posle dodati semafore, atomic i dr.

8.5 Odloženo izvršavanje

QFuture i sl. Asinhrono izvršavanje.

Beleške

Glava 9

Serijalizacija i deserijalizacija podataka

9.1 Pojam i uloga (de)serijalizacije podataka

Odnos sa višeprocesonom komunikacijom. Formati podataka (XML, JSON, binarni zapis, korisnički-definisan zapis (tekstualni, PDF), ...).

9.2 Organizacija aplikacije za (de)serijalizaciju

Ovde diskutovati o različitim dizajnima za implementiranje. Makar govoriti o narednim dizajnima:

- Svaka klasa implementira po 2 metoda: jedan za serijalizaciju u konkretan format, a drugi za deserijalizaciju iz konkretnog formata.
- Korišćenje međuformata i hijerarhije klasa koja u osnovi ima `ISerializer` i `ISerializable` interfejse. Prvi interfejs sadrži čisto virtualne metode `load` i `save` za serijalizaciju u datoteku iz međuformata, tj. deserijalizaciju iz datoteke u međuformat. Drugi interfejs sadrži čisto virtualne metode `load` i `save` za serijalizaciju u međuformat iz tog objekta, tj. deserijalizaciju iz tog objekta u međuformat. Za međuformat se može koristiti `QVariant`.

Govoriti o prednostima drugog dizajna u odnosu na prvi dizajn.

Beleške

Glava 10

Arhitektura softvera

10.1 Klijent-server arhitektura

Na primer, klijent i server chat pomocu Qt-ovih klasa za rad sa TCP klijentima i serverima.

10.2 Višeslojna arhitektura

Uvodna priča o različitim višeslojnim arhitekturama koji se često koriste (3-layer, MVC, MVVM, PAC, i sl.).

10.2.1 Model-pogled arhitektura

Ovde bi išao onaj primer koji deserijalizuje podatke o studentima u jedan `TreeModel`, a zatim prikazuje taj jedan model u tri različita pogleda – `ListView` koji prikazuje indekse, `TableView` koji prikazuje informacije o studentima i `TreeView` koji prikazuje informacije i o upisanim kursevima.

Taj primer se može dopuniti da se omogući editovanje podataka kako bi se pokazalo da izmena u jednom pogledu biva oslikava u svim ostalim pogledima (pošto se koristi jedan isti model za sve poglede).

10.3 Arhitektura zasnovana na događajima

Ilustrovati na slozenoj Qt aplikaciji koja koristi signale i slotove za komunikaciju izmedju razlicitih komponenti. Komponente bi trebalo da budu relativno jednostavne i da se bave jednom stvari, a da ih ima makar tri ili četiri.

Beleške

Dodatak A

Rešenja zadataka

Rešenje 1

```
#include <complex>
#include <iostream>
#include <vector>
#include <numeric>

int main() {
    std::complex<long double> number;
    std::vector<std::complex<long double>> numbers;

    while (std::cin >> number) {
        numbers.push_back(number);
    }

    for (const auto &number : numbers) {
        std::cout << number << std::endl;
    }

    // Podrazumevano se kreira kompleksni broj (0,0).
    const std::complex<long double> neutral;
    const auto complex_sum =
        std::accumulate(numbers.begin(), numbers.end(), neutral);
```

```
    const auto numbers_size = const_cast<double>(number.size());
    std::cout << complex_sum / numbers_size << std::endl;

    return 0;
}
```

Rešenje 2

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <filesystem>
namespace fs = std::filesystem;

enum ReturnCodes {
    Success = 0,
    InvalidNumberOfArguments,
    CannotOpenInputFile,
    CannotOpenOutputFile
};

int main(int argc, char *argv[]) {
    const auto expected_number_of_arguments = 2;

    if (argc != expected_number_of_arguments) {
        std::cerr
            << "Neispravan broj argumenata komandne linije. "
            << "Molimo prosledite samo naziv ulazne datoteke."
            << std::endl;
        return ReturnCodes::InvalidNumberOfArguments;
    }

    std::ifstream input_file(argv[1]);
    if (!input_file.is_open()) {
        std::cerr
            << "Neuspesno otvaranje ulazne datoteke: "
            << argv[1]
```

```
        << std::endl;
    return ReturnCodes::CannotOpenInputFile;
}

std::vector<std::string> messages;
std::string line;
size_t maximum_length = 0u;

messages.push_back(std::string());
while (std::getline(input_file, line)) {
    messages.push_back(line);
    maximum_length = std::max(maximum_length, line.size());
    line.clear();
}
messages.push_back(std::string());

input_file.close();

const auto input_filename = fs::path(argv[1]);
const auto directory_name = input_filename.parent_path();
const auto input_stem = input_filename.stem();
const auto extension = input_filename.extension();

auto output_filename = directory_name;
output_filename /= input_stem;
output_filename += fs::path(".output");
output_filename += extension;

std::ofstream output_file(output_filename);
if (!output_file.is_open()) {
    std::cerr
        << "Neuspesno otvaranje izlazne datoteke: "
        << output_filename
        << std::endl;
    return ReturnCodes::CannotOpenOutputFile;
}
```

```
    const std::string bar(maximum_length + 4, '*');

    output_file << bar << std::endl;
    for (const auto &line : messages) {
        output_file
            << "* " << line
            << std::string(maximum_length - line.size(), ' ')
            << " *" << std::endl;
    }
    output_file << bar << std::endl;

    output_file.close();

    return ReturnCodes::Success;
}
```

Rešenje 3

```
#include <iostream>
#include <random>
#include <vector>
#include <cmath>

struct point {
    point(double x, double y) {
        m_x = x;
        m_y = y;
    }

    double m_x;
    double m_y;
};

void generate_two_random_doubles(double &x, double &y) {
    static const auto lower = -1000.0;
    static const auto upper = 1000.0;

    static std::uniform_real_distribution<double> unif(lower, upper);
```

```
    static std::default_random_engine re;
    // re.seed(0);

    x = unif(re);
    y = unif(re);
}

void delete_points(std::vector<const point *> &p_vec) {
    for (const auto &p : p_vec) {
        delete p;
    }
}

void generate_n_points(std::vector<const point *> &p_vec,
                       unsigned long long n) {
    double x, y;

    for (auto i = 0ull; i < n; i++) {
        generate_two_random_doubles(x, y);

        const auto p = new point(x, y);
        if (!p) {
            delete_points(p_vec);
            break;
        }

        p_vec.push_back(p);
    }
}

double distance_between_points(const point *p1, const point *p2) {
    return std::sqrt(std::pow(p1->m_x - p2->m_x, 2) +
                     std::pow(p1->m_y + p2->m_y, 2));
}

const point *find_farthest_point_from_kth(
        std::vector<const point *> &p_vec,
```

```
        unsigned long long n,
        const point *kth_point) {
    auto max_dist = distance_between_points(p_vec[0], kth_point);
    auto idx_max_dist = 0ull;

    for (auto i = 1ull; i < n; ++i) {
        const auto curr_dist = distance_between_points(p_vec[i],
                                                       kth_point);
        if (curr_dist > max_dist) {
            max_dist = curr_dist;
            idx_max_dist = i;
        }
    }

    return p_vec[idx_max_dist];
}

int main() {
    unsigned long long n;
    std::cout << "Unesite broj n veci od 1.000.000: ";
    std::cin >> n;

    std::vector<const point *> p_vec;
    generate_n_points(p_vec, n);

    if (p_vec.empty()) {
        std::cerr << "Alokacija je neuspesna!" << std::endl;
        return EXIT_FAILURE;
    }

    unsigned long long k;
    std::cout << "Unesite broj k manji od n: ";
    std::cin >> k;

    const auto kth_point = p_vec[k];
    const auto p_found = find_farthest_point_from_kth(p_vec,
                                                      n,
```

```
                                                    kth_point);

    std::cout
        << "Tacka koja je najudaljenija od tacke ("
        << kth_point->m_x << "," << kth_point->m_y << ") je tacka ("
        << p_found->m_x << "," << p_found->m_y << ")." << std::endl;

    delete_points(p_vec);

    return EXIT_SUCCESS;
}
```

Rešenje 4

```
#include <iostream>
#include <random>
#include <cmath>

struct point {
    double m_x;
    double m_y;
};

void generate_two_random_doubles(double &x, double &y) {
    static const auto lower = -1000.0;
    static const auto upper = 1000.0;

    static std::uniform_real_distribution<double> unif(lower, upper);
    static std::default_random_engine re;
    // re.seed(0);

    x = unif(re);
    y = unif(re);
}

point *generate_n_points(unsigned long long n) {
    point *p = new point[n];
    if (!p) {
```

```
        return nullptr;
    }

    double x, y;

    for (auto i = 0ull; i < n; i++) {
        generate_two_random_doubles(x, y);

        p[i].m_x = x;
        p[i].m_y = x;
    }

    return p;
}

double distance_between_points(const point &p1, const point &p2) {
    return std::sqrt(std::pow(p1.m_x - p2.m_x, 2) +
                     std::pow(p1.m_y + p2.m_y, 2));
}

const point &find_farthest_point_from_kth(
        const point *p_arr,
        unsigned long long n,
        const point &kth_point) {
    auto max_dist = distance_between_points(p_arr[0], kth_point);
    auto idx_max_dist = 0ull;

    for (auto i = 1ull; i < n; ++i) {
        const auto curr_dist = distance_between_points(p_arr[i],
                                                       kth_point);
        if (curr_dist > max_dist) {
            max_dist = curr_dist;
            idx_max_dist = i;
        }
    }

    return p_arr[idx_max_dist];
}
```

```
int main() {
    unsigned long long n;
    std::cout << "Unesite broj n veci od 1.000.000: ";
    std::cin >> n;

    const auto p_arr = generate_n_points(n);
    if (!p_arr) {
        std::cerr << "Alokacija je neuspesna!" << std::endl;
        return EXIT_FAILURE;
    }

    unsigned long long k;
    std::cout << "Unesite broj k manji od n: ";
    std::cin >> k;

    const auto &kth_point = p_arr[k];
    const auto &p_found = find_farthest_point_from_kth(p_arr,
                                                       n,
                                                       kth_point);

    std::cout
        << "Tacka koja je najudaljenija od tacke ("
        << kth_point.m_x << "," << kth_point.m_y << ") je tacka ("
        << p_found.m_x << "," << p_found.m_y << ")." << std::endl;

    delete[] p_arr;

    return EXIT_SUCCESS;
}
```

Rešenje 5

```
#include <string>
#include <vector>
#include <memory>
#include <iostream>
```

```
struct Course {
    std::string m_title;
    unsigned m_ects;

    Course(std::string title, unsigned ects) {
        m_title = title;
        m_ects = ects;
    }
};

struct Student {
    std::string m_index;
    std::vector<std::shared_ptr<Course>> m_courses;

    Student(std::string index) {
        m_index = index;
    }
};

int main() {
    // Ucitavanje kurseva
    std::cout << "Unesite broj kurseva:" << std::endl;
    unsigned k;
    std::cin >> k;

    std::vector<std::shared_ptr<Course>> courses;

    for (auto i = 0u; i < k; ++i) {
        std::cout << "Unesite naziv kursa: " << std::endl;
        std::string title;
        std::cin >> title;

        std::cout << "Unesite broj bodova: " << std::endl;
        unsigned ects;
        std::cin >> ects;

        auto course = std::make_shared<Course>(title, ects);
```

```
    courses.push_back(course);
}

// Ucitavanje studenata
std::cout << "Unesite broj studenata:" << std::endl;
unsigned n;
std::cin >> n;

std::vector<std::shared_ptr<Student>> students;

for (auto i = 0u; i < n; ++i) {
    std::cout << "Unesite indeks studenta: " << std::endl;
    std::string index;
    std::cin >> index;

    auto student = std::make_shared<Student>(index);

    // Upisivanje studenata na kurseve
    for (const auto &course : courses) {
        std::cout
            << "Da li zelite da upisete tekuceg studenta na kurs "
            << course->m_title
            << " (" << course->m_ects << ")? [da/ne]" << std::endl;

        std::string response;
        std::cin >> response;
        if (response != "da") {
            continue;
        }

        student->m_courses.push_back(course);
    }

    students.push_back(student);
}

// Posto svaki student sada cuva svoj spisak kurseva,
// mozemo ocistiti vektor kurseva,
```

```
    // cime se unistavaju deljeni pokazivaci na te kurseve.
    courses.clear();

    // Izlistavanje informacija o studentima
    for (const auto &student : students) {
        std::cout
            << "Student sa indeksom " << student->m_index
            << " ima naredne upisane kurseve:" << std::endl;

        for (const auto& course : student->m_courses) {
            std::cout
                << '\t' << course->m_title
                << " (" << course->m_ects << ")" << std::endl;
        }
    }

    return 0;
}
```

Rešenje 6

```
#include <string>
#include <vector>
#include <memory>
#include <iostream>

struct Student {
    std::string m_index;
    std::string m_name;

    Student(std::string index, std::string name) {
        m_index = index;
        m_name = name;
    }
};

struct StudyProgram {
    std::string m_title;
```

```
    unsigned m_ects;
    std::vector<std::unique_ptr<Student>> m_students;

    StudyProgram(std::string title, unsigned ects) {
        m_title = title;
        m_ects = ects;
    }
};

int main() {
    std::cout << "Unesite broj smerova:" << std::endl;
    unsigned s;
    std::cin >> s;

    std::vector<std::unique_ptr<StudyProgram>> studyPrograms;

    for (auto i = 0u; i < s; ++i) {
        std::cout << "Unesite naziv smera: " << std::endl;
        std::string title;
        std::cin >> title;

        std::cout << "Unesite broj bodova: " << std::endl;
        unsigned ects;
        std::cin >> ects;

        auto studyProgram = std::make_unique<StudyProgram>(title, ects);
        studyPrograms.push_back(std::move(studyProgram));
    }

    std::cout << "Unesite broj studenata:" << std::endl;
    unsigned n;
    std::cin >> n;

    for (auto i = 0u; i < n; ++i) {
        std::cout << "Unesite indeks studenta: " << std::endl;
        std::string index;
        std::cin >> index;
```

```
        std::cout << "Unesite ime studenta: " << std::endl;
        std::string name;
        std::cin >> name;

        auto student = std::make_unique<Student>(index, name);

        for (auto j = 0u; j < studyPrograms.size(); ++j) {
            std::cout
                << j + 1 << ". "
                << studyPrograms[j]->m_title << std::endl;
        }
        std::cout
            << "Unesite identifikator smera za upis studenta: "
            << std::endl;

        unsigned chosenSP;
        std::cin >> chosenSP;

        studyPrograms[chosenSP-1]->m_students.push_back(std::move(student));
    }

    for (const auto &studyProgram : studyPrograms) {
        std::cout
            << "Upisani studenti na kursu " << studyProgram->m_title
            << " (" << studyProgram->m_ects << "):" << std::endl;

        for (const auto& student : studyProgram->m_students) {
            std::cout
                << '\t' << student->m_name
                << " (" << student->m_index << ")" << std::endl;
        }
    }

    return 0;
}
```

Rešenje 7

```
#include <iostream>
#include <random>
#include <vector>
#include <cmath>

struct point {
    point(double x, double y) {
        m_x = x;
        m_y = y;
    }

    double m_x;
    double m_y;
};

struct raii_points {
    std::vector<const point *> points;

    void generate_two_random_doubles(double &x, double &y) {
        static const auto lower = -1000.0;
        static const auto upper = 1000.0;

        static std::uniform_real_distribution<double> unif(lower, upper);
        static std::default_random_engine re;
        // re.seed(0);

        x = unif(re);
        y = unif(re);
    }

    void generate_n_points(unsigned long long n) {
        double x, y;

        for (auto i = 0ull; i < n; i++) {
            generate_two_random_doubles(x, y);

            const auto p = new point(x, y);
```

```
            if (!p) {
                break;
            }

            points.push_back(p);
        }
    }

    raii_points(unsigned long long n) {
        std::cout << "Konstruktor raii_points" << std::endl;
        generate_n_points(n);
    }

    ~raii_points() {
        std::cout << "Destruktor raii_points" << std::endl;
        for (const auto &p : points) {
            delete p;
        }
    }
};

double distance_between_points(const point *p1, const point *p2) {
    const auto result = std::sqrt(std::pow(p1->m_x - p2->m_x, 2) +
                                  std::pow(p1->m_y + p2->m_y, 2));
    if (result < 0.1) {
        throw std::exception("Tacke su previse blizu jedna drugoj");
    }
    return result;
}

const point *find_farthest_point_from_kth(
        raii_points &pRAII,
        unsigned long long n,
        const point *kth_point) {
    auto max_dist = distance_between_points(pRAII.points[0], kth_point);
    auto idx_max_dist = 0ull;

    for (auto i = 1ull; i < n; ++i) {
```

```
        const auto curr_dist =
            distance_between_points(pRAII.points[i], kth_point);
        if (curr_dist > max_dist) {
            max_dist = curr_dist;
            idx_max_dist = i;
        }
    }

    return pRAII.points[idx_max_dist];
}

int main() {
    unsigned long long n;
    std::cout << "Unesite broj n veci od 1.000.000: ";
    std::cin >> n;

    try {
        raii_points pRAII(n);

        unsigned long long k;
        std::cout << "Unesite broj k manji od n: ";
        std::cin >> k;

        const auto kth_point = pRAII.points[k];
        const auto p_found =
            find_farthest_point_from_kth(pRAII, n, kth_point);

        std::cout
            << "Tacka koja je najudaljenija od tacke ("
            << kth_point->m_x << "," << kth_point->m_y << ") je tacka ("
            << p_found->m_x << "," << p_found->m_y << ")." << std::endl;
    } catch (const std::exception &e) {
        std::cout << "Ispaljen je izuzetak: " << e.what() << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Rešenje 8

```
#include <iostream>
#include <random>
#include <cmath>

struct point {
    double m_x;
    double m_y;
};

struct raii_points_array {
    point *points = nullptr;

    void generate_two_random_doubles(double &x, double &y) {
        static const auto lower = -1000.0;
        static const auto upper = 1000.0;

        static std::uniform_real_distribution<double> unif(lower, upper);
        static std::default_random_engine re;
        // re.seed(0);

        x = unif(re);
        y = unif(re);
    }

    void generate_n_points(unsigned long long n) {
        points = new point[n];
        if (!points) {
            return;
        }

        double x, y;

        for (auto i = 0ull; i < n; i++) {
            generate_two_random_doubles(x, y);

            points[i].m_x = x;
```

```
            points[i].m_y = x;
        }
    }

    raii_points_array(unsigned long long n) {
        std::cout << "Konstruktor raii_points_array" << std::endl;
        generate_n_points(n);
    }

    ~raii_points_array() {
        std::cout << "Destruktor raii_points_array" << std::endl;
        delete[] points;
    }
};

double distance_between_points(const point &p1, const point &p2) {
    const auto result = std::sqrt(std::pow(p1.m_x - p2.m_x, 2) +
                                  std::pow(p1.m_y + p2.m_y, 2));
    if (result < 0.1) {
        throw std::exception("Tacke su previse blizu jedna drugoj");
    }
    return result;
}

const point &find_farthest_point_from_kth(
        const point *p_arr,
        unsigned long long n,
        const point &kth_point) {
    auto max_dist = distance_between_points(p_arr[0], kth_point);
    auto idx_max_dist = 0ull;

    for (auto i = 1ull; i < n; ++i) {
        const auto curr_dist =
            distance_between_points(p_arr[i], kth_point);
        if (curr_dist > max_dist) {
            max_dist = curr_dist;
            idx_max_dist = i;
        }
```

```
    }

    return p_arr[idx_max_dist];
}

int main() {
    unsigned long long n;
    std::cout << "Unesite broj n veci od 1.000.000: ";
    std::cin >> n;

    try {
        const raii_points_array p_arr(n);
        if (!p_arr.points) {
            std::cerr << "Alokacija je neuspesna!" << std::endl;
            return EXIT_FAILURE;
        }

        unsigned long long k;
        std::cout << "Unesite broj k manji od n: ";
        std::cin >> k;

        const auto &kth_point = p_arr.points[k];
        const auto &p_found =
            find_farthest_point_from_kth(p_arr.points, n, kth_point);

        std::cout
            << "Tacka koja je najudaljenija od tacke ("
            << kth_point.m_x << "," << kth_point.m_y << ") je tacka ("
            << p_found.m_x << "," << p_found.m_y << ")." << std::endl;
    }  catch (const std::exception &e) {
        std::cout << "Ispaljen je izuzetak: " << e.what() << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Rešenje 9

```
// ------------------------------------------------------
// Datoteka: CannotOpenFileException.hpp

#ifndef CANNOTOPENFILEEXCEPTION_HPP
#define CANNOTOPENFILEEXCEPTION_HPP

#include <QString>

struct CannotOpenFileException {
    QString m_what;

    CannotOpenFileException(QString filename) {
        m_what = "Cannot open file " + filename;
    }
};

#endif // CANNOTOPENFILEEXCEPTION_HPP

// ------------------------------------------------------
// Datoteka: Course.hpp

#ifndef COURSE_HPP
#define COURSE_HPP

#include <QString>
#include <QVector>
#include <QSharedPointer>
#include <QFile>
#include <QTextStream>

#include "CannotOpenFileException.hpp"

struct Course {
    QString m_title;
    unsigned m_ects;

    Course(QString title, unsigned ects) {
        m_title = title;
```

```
        m_ects = ects;
    }
};

struct CourseRAII {
    QVector<QSharedPointer<Course>> m_courses;
    QString m_filename;

    CourseRAII() {
        m_filename = "courses.txt";

        QFile file(m_filename);
        if (!file.open(QFile::ReadOnly | QFile::Text)) {
            throw CannotOpenFileException(m_filename);
        }

        QTextStream in(&file);
        auto line = in.readLine();
        while (!line.isNull()) {
            const auto stringArr = line.split(' ');
            const auto title = stringArr[0];
            const auto ects = stringArr[1].toUInt();
            const auto course =
                QSharedPointer<Course>::create(title, ects);
            m_courses.push_back(course);

            line = in.readLine();
        }
    }

    ~CourseRAII() {
        QFile file(m_filename);
        if (!file.open(QFile::WriteOnly | QFile::Text)) {
            return;
        }

        QTextStream out(&file);
        foreach (const auto &course, m_courses) {
```

```
            out << course->m_title << ' ' << course->m_ects << '\n';
        }
    }
};

#endif // COURSE_HPP

// ----------------------------------------------------------
// Datoteka: Student.hpp

#ifndef STUDENT_H
#define STUDENT_H

#include <QString>
#include <QVector>
#include <QSharedPointer>
#include <QFile>

#include "Course.hpp"

struct Student {
    QString m_index;
    QString m_name;
    QVector<QSharedPointer<Course>> m_courses;

    Student(QString index, QString name) {
        m_index = index;
        m_name = name;
    }
};

struct StudentRAII {
    QVector<QSharedPointer<Student>> m_students;
    QString m_filename;

    StudentRAII() {
        m_filename = "students.txt";
```

```
        QFile file(m_filename);
        if (!file.open(QFile::ReadOnly | QFile::Text)) {
            throw CannotOpenFileException(m_filename);
        }

        QTextStream in(&file);
        QString line = in.readLine();
        while (!line.isNull()) {
            const auto stringArr = line.split(' ');
            const auto index = stringArr[0];
            const auto name = stringArr[1];
            const auto student =
                QSharedPointer<Student>::create(index, name);
            m_students.push_back(student);

            line = in.readLine();
        }
    }

    ~StudentRAII() {
        QFile file(m_filename);
        if (!file.open(QFile::WriteOnly | QFile::Text)) {
            return;
        }

        QTextStream out(&file);
        foreach (const auto &student, m_students) {
            out << student->m_index << ' ' << student->m_name << '\n';
        }
    }
};

#endif // STUDENT_H

// ------------------------------------------------------
// Datoteka: main.cpp
#include <QTextStream>
```

```
#include "Course.hpp"
#include "Student.hpp"
#include "CannotOpenFileException.hpp"

int main() {
    // Kreiranje ulaznog/izlaznog toka za standardni ulaz/izlaz
    QTextStream cin(stdin, QFile::ReadOnly);
    QTextStream cout(stdout, QFile::WriteOnly);

    try {
        // Ucitavanje kurseva i studenata iz datoteke
        CourseRAII coursesRAII;
        StudentRAII studentsRAII;

        // Ucitavanje kurseva od korisnika
        cout << "Unesite broj kurseva:\n"; cout.flush();
        unsigned k;
        cin >> k;

        for (auto i = 0u; i < k; ++i) {
            cout << "Unesite naziv kursa: \n"; cout.flush();
            QString title;
            cin >> title;

            cout << "Unesite broj bodova: \n"; cout.flush();
            unsigned ects;
            cin >> ects;

            const auto course =
                QSharedPointer<Course>::create(title, ects);
            coursesRAII.m_courses.push_back(course);
        }

        // Ucitavanje studenata od korisnika
        cout << "Unesite broj studenata:\n"; cout.flush();
        unsigned n;
        cin >> n;
```

```
for (auto i = 0u; i < n; ++i) {
    cout << "Unesite indeks studenta: \n"; cout.flush();
    QString index;
    cin >> index;

    cout << "Unesite ime studenta: \n"; cout.flush();
    QString name;
    cin >> name;

    const auto student =
        QSharedPointer<Student>::create(index, name);
    studentsRAII.m_students.push_back(student);
}

// Upisivanje studenata na kurseve
foreach (const auto &student, studentsRAII.m_students) {
    foreach (const auto &course, coursesRAII.m_courses) {
        cout
            << "Da li zelite da upisete studenta "
            << student->m_name
            << " (" << student->m_index << ")"
            << " na kurs " << course->m_title
            << " (" << course->m_ects << ")?"
            << " [da/ne]\n"; cout.flush();

        QString response;
        cin >> response;
        if (response != "da") {
            continue;
        }

        student->m_courses.push_back(course);
    }
}

// Izlistavanje informacija o studentima
foreach (const auto &student, studentsRAII.m_students) {
    cout
```

```
                << "Student " << student->m_name
                << " (" << student->m_index << ")"
                << " ima naredne upisane kurseve:\n"; cout.flush();

            foreach (const auto &course, student->m_courses) {
                cout
                    << '\t' << course->m_title
                    << " (" << course->m_ects
                    << ")\n"; cout.flush();
            }
        }
    } catch (const CannotOpenFileException &e) {
        cout << e.m_what << '\n'; cout.flush();
    }

    return 0;
}
```

Rešenje 10

```
// ----------------------------------------------------------
// Datoteka: DatabaseManager.hpp

#ifndef DATABASEMANAGER_HPP
#define DATABASEMANAGER_HPP

#include <QSqlDatabase>
#include <QSqlDriver>
#include <QDebug>

struct DatabaseManager {
    QSqlDatabase m_database;

    DatabaseManager() {
        m_database = QSqlDatabase::addDatabase("QSQLITE");
        m_database.setHostName("localhost");
        m_database.setUserName("rs");
        m_database.setPassword("1234");
```

```
        m_database.setDatabaseName("FACULTY");

        if (m_database.open()) {
            qDebug() << "Uspesna konekcija sa BP";
            const auto hasTransactionSupport =
                m_database.driver()->hasFeature(QSqlDriver::Transactions);
            qDebug() << "Konekcija"
                << (hasTransactionSupport ? "podrzava" : "ne podrzava")
                << "transakcije";
        }
        else {
            qDebug() << "Neuspesna konekcija sa BP";
        }
    }

    ~DatabaseManager() {
        if (m_database.isOpen()) {
            m_database.close();
            qDebug() << "Zatvaram konekciju sa BP";
        }
    }
};

#endif // DATABASEMANAGER_HPP

// ------------------------------------------------------
// Datoteka: Student.hpp

#ifndef STUDENT_HPP
#define STUDENT_HPP

#include <QString>
#include <QSharedPointer>
#include <QVector>
#include <QSqlQuery>
#include <QSqlError>
```

```
#include "DatabaseManager.hpp"

struct Student {
    QString m_index;
    QString m_name;

    Student(QString index, QString name) {
        m_index = index;
        m_name = name;
    }
};

struct StudentRAII {
    QSharedPointer<DatabaseManager> m_db;
    QVector<QSharedPointer<Student>> m_students;

    void create_table_if_not_exists() {
        if (!m_db->m_database.tables().contains("STUDENTS")) {
            QSqlQuery query(m_db->m_database);
            QString sql =
                "CREATE TABLE STUDENTS ( "
                "   STUDINDEX VARCHAR(9) PRIMARY KEY NOT NULL, "
                "   NAME VARCHAR(50) NOT NULL "
                ")";
            if (!query.exec(sql)) {
                throw query.lastError();
            }
        }
    }

    void read_students_from_table() {
        QSqlQuery query(m_db->m_database);
        QString sql = "SELECT * FROM STUDENTS";
        if (!query.exec(sql)) {
            throw query.lastError();
        }
```

```
    while (query.next()) {
        const auto index = query.value("STUDINDEX").toString();
        const auto name = query.value("NAME").toString();

        auto student =
            QSharedPointer<Student>::create(index, name);
        m_students.push_back(student);
    }
}

StudentRAII(QSharedPointer<DatabaseManager> db) {
    m_db = db;
    create_table_if_not_exists();
    read_students_from_table();
}

bool delete_from_table() {
    QSqlQuery query(m_db->m_database);
    QString sql = "DELETE FROM STUDENTS";
    if (!query.exec(sql)) {
        const auto e = query.lastError();

        m_db->m_database.rollback();
        qDebug()
            << "SQLCODE: " << e.nativeErrorCode()
            << ", SQLERR: " << e.text()
            << ", SQL:" << sql;

        return false;
    }
    return true;
}

bool insert_students() {
    QSqlQuery query(m_db->m_database);
    QString sql = "INSERT INTO STUDENTS VALUES (:index, :name)";
    if (!query.prepare(sql)) {
```

```
        const auto e = query.lastError();

        m_db->m_database.rollback();

        qDebug()
            << "SQLCODE: " << e.nativeErrorCode()
            << ", SQLERR: " << e.text()
            << ", SQL:" << sql;
        return false;
    }
    foreach (const auto &student, m_students) {
        query.bindValue(":index", student->m_index);
        query.bindValue(":name", student->m_name);
        if (!query.exec()) {
            const auto e = query.lastError();

            m_db->m_database.rollback();

            qDebug()
                << "SQLCODE: " << e.nativeErrorCode()
                << ", SQLERR: " << e.text()
                << ", SQL:" << sql;
            return false;
        }
    }
    return true;
}

~StudentRAII() {
    if (!m_db->m_database.transaction()) {
        return;
    }

    const auto isDeleted = delete_from_table();
    if (!isDeleted) {
        return;
    }
```

```
        const auto isInserted = insert_students();
        if (!isInserted) {
            return;
        }

        m_db->m_database.commit();
        qDebug()
            << "Uspesno su sacuvane informacije o studentima u BP";
    }
};

#endif // STUDENT_HPP

// ------------------------------------------------------------
// Datoteka: StudyProgram.hpp

#ifndef STUDYPROGRAM_HPP
#define STUDYPROGRAM_HPP

#include <QString>
#include <QVector>
#include <QSharedPointer>
#include <QSqlQuery>
#include <QSqlError>

#include "Student.hpp"
#include "DatabaseManager.hpp"

struct StudyProgram {
    QString m_title;
    unsigned m_ects;
    QVector<QSharedPointer<Student>> m_students;

    StudyProgram(QString title, unsigned ects) {
        m_title = title;
        m_ects = ects;
    }
};
```

```
struct StudyProgramRAII {
    QSharedPointer<DatabaseManager> m_db;
    QVector<QSharedPointer<StudyProgram>> m_studyPrograms{};

    void create_table_if_not_exists() {
        if (!m_db->m_database.tables().contains("STUDYPROGRAMS")) {
            QSqlQuery query(m_db->m_database);
            QString sql =
                "CREATE TABLE STUDYPROGRAMS ( "
                "   TITLE VARCHAR(50) PRIMARY KEY NOT NULL, "
                "   ECTS SMALLINT NOT NULL "
                ")";
            if (!query.exec(sql)) {
                throw query.lastError();
            }
        }
    }

    void read_studyprograms_from_table() {
        QSqlQuery query(m_db->m_database);
        QString sql = "SELECT * FROM STUDYPROGRAMS";
        if (!query.exec(sql)) {
            throw query.lastError();
        }

        while (query.next()) {
            const auto title = query.value("TITLE").toString();
            const auto ects = query.value("ECTS").toUInt();

            auto studyProgram =
                QSharedPointer<StudyProgram>::create(title, ects);
            m_studyPrograms.push_back(studyProgram);
        }
    }

    StudyProgramRAII(QSharedPointer<DatabaseManager> db) {
        m_db = db;
```

```
    create_table_if_not_exists();
    read_studyprograms_from_table();
}

bool delete_from_table() {
    QSqlQuery query(m_db->m_database);
    QString sql = "DELETE FROM STUDYPROGRAMS";
    if (!query.exec(sql)) {
        const auto e = query.lastError();

        m_db->m_database.rollback();

        qDebug()
            << "SQLCODE: " << e.nativeErrorCode()
            << ", SQLERR: " << e.text()
            << ", SQL:" << sql;
        return false;
    }
    return true;
}

bool insert_studyprograms() {
    QSqlQuery query(m_db->m_database);
    QString sql = "INSERT INTO STUDYPROGRAMS VALUES (:title, :ects)";
    if (!query.prepare(sql)) {
        const auto e = query.lastError();

        m_db->m_database.rollback();

        qDebug()
            << "SQLCODE: " << e.nativeErrorCode()
            << ", SQLERR: " << e.text()
            << ", SQL:" << sql;
        return false;
    }
    foreach (const auto &studyProgram, m_studyPrograms) {
        query.bindValue(":title", studyProgram->m_title);
```

```
            query.bindValue(":ects", studyProgram->m_ects);
            if (!query.exec()) {
                const auto e = query.lastError();

                m_db->m_database.rollback();

                qDebug()
                    << "SQLCODE: " << e.nativeErrorCode()
                    << ", SQLERR: " << e.text()
                    << ", SQL:" << sql;
                return false;
            }
        }
        return true;
    }

    ~StudyProgramRAII() {
        if (!m_db->m_database.transaction()) {
            return;
        }

        const auto isDeleted = delete_from_table();
        if (!isDeleted) {
            return;
        }

        const auto isInserted = insert_studyprograms();
        if (!isInserted) {
            return;
        }

        m_db->m_database.commit();
        qDebug()
            << "Uspesno su sacuvane informacije o smerovima u BP";
    }
};

#endif // STUDYPROGRAM_HPP
```

```
// ------------------------------------------------------
// Datoteka: main.cpp
#include <QSharedPointer>
#include <QSqlError>
#include <QDebug>
#include <QTextStream>

#include "DatabaseManager.hpp"
#include "Student.hpp"
#include "StudyProgram.hpp"

int main() {
    // Kreiranje ulaznog/izlaznog toka za standardni ulaz/izlaz
    QTextStream cin(stdin, QIODevice::ReadOnly);
    QTextStream cout(stdout, QIODevice::WriteOnly);

    // Konekcija ka bazi podataka
    auto db = QSharedPointer<DatabaseManager>::create();

    try {
        // Ucitavanje studenata i smerova iz baze podataka
        StudentRAII studentsRAII(db);
        StudyProgramRAII studyProgramsRAII(db);

        // Ucitavanje smerova od korisnika
        cout << "Unesite broj smerova:" << '\n'; cout.flush();
        unsigned s;
        cin >> s;

        for (auto i = 0u; i < s; ++i) {
            cout << "Unesite naziv smera: " << '\n'; cout.flush();
            QString title;
            cin >> title;

            cout << "Unesite broj bodova: " << '\n'; cout.flush();
            unsigned ects;
            cin >> ects;
```

```
    const auto studyProgram =
        QSharedPointer<StudyProgram>::create(title, ects);
    studyProgramsRAII.m_studyPrograms.push_back(studyProgram);
}

// Ucitavanje studenata od korisnika
cout << "Unesite broj studenata:" << '\n'; cout.flush();
unsigned n;
cin >> n;

for (auto i = 0u; i < n; ++i) {
    cout << "Unesite indeks studenta: " << '\n'; cout.flush();
    QString index;
    cin >> index;

    cout << "Unesite ime studenta: " << '\n'; cout.flush();
    QString name;
    cin >> name;

    const auto student =
        QSharedPointer<Student>::create(index, name);
    studentsRAII.m_students.push_back(student);
}

// Upisivanje studenata na smerove
foreach(const auto &student, studentsRAII.m_students) {
    int j = 0;
    foreach(const auto &studyProgram,
            studyProgramsRAII.m_studyPrograms) {
        cout
            << ++j << ". "
            << studyProgram->m_title << '\n'; cout.flush();
    }

    cout
        << "Unesite identifikator smera za upis studenta: "
        << '\n'; cout.flush();
```

```
            unsigned chosenSP;
            cin >> chosenSP;

            studyProgramsRAII.m_studyPrograms[chosenSP-1]
                ->m_students.push_back(student);
        }

        // Izlistavanje informacije o smerovima
        foreach(const auto &studyProgram,
                studyProgramsRAII.m_studyPrograms) {
            cout
                << "Upisani studenti na kursu "
                << studyProgram->m_title
                << " (" << studyProgram->m_ects << "):"
                << '\n'; cout.flush();

            foreach (const auto &student,
                     studyProgram->m_students) {
                cout
                    << '\t' << student->m_name
                    << " (" << student->m_index << ")"
                    << '\n'; cout.flush();
            }
        }
    } catch (const QSqlError &e) {
        qDebug()
            << "SQLCODE: " << e.nativeErrorCode()
            << ", SQLERR: " << e.text();
    }

    return 0;
}
```