Saša Malkov

Objektno orijentisano programiranje

C++ kroz primere

Matematički fakultet

Saša Malkov

Objektno orijentisano programiranje
C++ KROZ PRIMERE

AUTORSKO E-IZDANJE
BEOGRAD, 2019.

Saša Malkov

Objektno orijentisano programiranje
C++ kroz primere

Izdavač: Autor, Beograd

Recenzenti prvog izdanja:

dr Miodrag Živković
mr Aleksandar Samardžić

Obrada teksta: Autor

Korice: Autor

Ovo elektronsko izdanje je po sadržaju identično prvom štampanom izdanju:

CIP – Каталогизација у публикацији
Народна библиотека Србије, Београд

004.42.045
004.432.2C++

МАЛКОВ, Саша
Objektno orijentisano programiranje : C++ kroz primere / Saša Malkov. – 1. izd. – Beograd : Matematički fakultet, 2007 (Beograd : Sa & Mi Grafika). – XIV, 420 ctr. : graf. Prikazi ; 24 cm

Tiraž 300. – Bibliografija: str. 419-420. – Registar.

ISBN 978-86-7589-061-4

а) Објектно оријентисано програмирање б) Програмски језик „C++“

COBIS.SR-ID 138003724

PREDGOVOR

Dovoljno je čak i letimično pregledati naslove izdanja iz oblasti računarstva, da bi se videlo da su objektno orijentisano programiranje i programski jezik C++ neke od tema koje su i brojem i kvalitetom knjiga sasvim solidno pokrivene. Na srpski jezik su prevedene, bez preterivanja, neke od najboljih knjiga posvećenih ovim temama. Čemu, onda, još jedna knjiga?

Iskustvo, stečeno tokom nekoliko godina rada sa studentima na Matematičkom fakultetu Univerziteta u Beogradu, pokazuje da pri savladavanju objektno orijentisanog programiranja i programskog jezika C++ postoje određene teškoće. Sam programski jezik je prilično obiman, a kada se tome pridruže i neophodne tehnike objektno orijentisanog programiranja, širina teme i veliki broj značajnih detalja, ovladavanje materijom postaje prilično ozbiljan problem. Najbolje knjige o programskom jeziku C++ svojim obimom i detaljnošću prilično obeshrabruju čitaoce, koji u njima ne uspevaju da prepoznaju jasne i čvrste oslonce za svoje teško rastuće znanje. S druge strane, izbor neke od knjiga koje su manjeg obima često se pokazuje kao pogrešan korak, jer su neke relativno važne stvari nedovoljno dobro objašnjene, ili čak nisu ni pomenute. Ipak, stiče se utisak da najveći problem predstavlja nedostatak većeg broja složenih primera, ili njihovo izlaganje bez dovoljno detaljnih pratećih objašnjenja, ili čak njihovo potpuno izostajanje. Prava svrha metoda objektno orijentisanog programiranja i mogućnosti programskog jezika C++ dolaze do izražaja tek pri rešavanju složenih problema. Zbog toga se oni teško mogu naučiti kroz jednostavne ili nedovoljno prodiskutovane primere.

Ova knjiga bi trebalo da predstavi dobre primere. Zamišljena je kao praktični priručnik koji bi trebalo da pomogne pri savladavanju osnovnih tehnika objektno orijentisanog programiranja na programskom jeziku C++. Za razliku od referentnih priručnika za programski jezik ili određene metode razvoja, ovde su u centru pažnje problemi. Paralelno sa razmatranjem mogućih načina rešavanja navedenih problema, posvećuje se pažnja elementima programskog jezika i osnovnim tehnikama objektno orijentisanog razvoja softvera. Delovi knjige su pisani i raspoređeni tako da se mogu čitati redom. Čitaoci koji bi

želeli da preskoče neke teme mogu se osloniti na indeks. Uz tekst svakog zadatka naveden je sažet pregled tema koje se obrađuju pri rešavanju.

Većina predstavljenih primera je obrađivana sa više generacija studenata, što je pomoglo da se jasnije izdvoje aspekti problema na koje je neophodno obratiti više pažnje. Pri objašnjavanju postupka rešavanja problema, najviše prostora je posvećeno upravo teže savladivim temama i detaljima. Pri donošenju odluka razmatrani su mogući pristupi problemu i objašnjavani kriterijumi na osnovu kojih se odlučuje.

Primeri u prvom delu knjige su izabrani tako da se kroz njihovo rešavanje razmatraju osnovne tehnike objektno orijentisanog programiranja u programskom jeziku C++. Najviše pažnje se posvećuje definisanju klasa, radu sa dinamičkim strukturama podataka, šablonima i hijerarhijama klasa. Postepeno se uvode elementi standardne biblioteke. Akcenat je stavljen na predstavljanje ispravnih načina primene različitih elemenata jezika i tehnika programiranja, kao i na uobičajene propuste.

U drugom delu se rešavaju ozbiljniji problemi. Kroz svaki od primera se detaljnije upoznaju neki složeniji elementi programskog jezika i njegove standardne biblioteke ili neke tehnike objektno orijentisanog programiranja. Predstavljeno je građenje hijerarhija klasa u različitim okolnostima i na različitim osnovama. Obuhvaćeni su neki od najvažnijih koncepata objektno orijentisanog programiranja i upotrebljene neke od tehnika. Ovaj deo knjige namenjen je čitaocima koji su uspešno savladali osnovne teme i u stanju su da samostalno rešavaju elementarne probleme.

Treći deo predstavljaju dodaci. U prvom dodatku su izloženi neki od najvažnijih elemenata standardne biblioteke programskog jezika C++. Drugi dodatak sadrži nekoliko zadataka za vežbu.

Kako bi trebalo čitati knjigu i učiti objektno orijentisano programiranje na programskom jeziku C++?

Isključivo uz praktičan rad.

Radi temeljnog upoznavanja različitih aspekata programiranja neophodno je posegnuti za literaturom iz odgovarajućih oblasti. Međutim, razumeti kako i zašto neko drugi preporučuje rešavanje nekih problema na način koji demonstrira kao najjednostavniji i najbolji, predstavlja samo prvi korak. Dva osnovna aspekta programiranja su analiza problema i sinteza rešenja. Upoznavanjem osnovnih principa rešavanja problema i razumevanjem principa rešavanja i ponuđenih rešenja se savladavaju neke faze analize problema i logički principi sinteze rešenja, dok se pažljivim praćenjem postupka rešavanja stiče detaljniji uvid u postupak analize i upoznaju konstruktivni aspekti sinteze rešenja. Ipak, da bi se pročitani sadržaji zaista usvojili, najčešće nije dovoljno razumeti tuđe mišljenje, već je neophodno napisati sopstveno rešenje.

Lična dimenzija sinteze programa, koja se tiče kreativnosti programera i odluka koje on tokom rešavanja problema mora da donosi, ne može se steći samim čitanjem ni ove ni bilo koje druge knjige o programiranju. Umeće programiranja predstavlja spoj stručnih znanja i iskustva koje omogućava kvalitetnije i lakše odlučivanje u trenucima kada je potrebno

primeniti stečena znanja i izabrati jedan od brojnih puteva koji mogu voditi do rešenja. Programiranje obuhvata određene kreativne procese, pa se kao i u slučaju drugih oblika stvaranja i odgovarajuće dimenzije programerskog umeća najbolje učvršćuju i proširuju vežbom. Vežbanje omogućava da se stečena znanja potvrde u praksi i temeljnije usvoje, kao i da se otklone neke nedoumice do kojih može doći usled nedovoljno konsultovanja literature ili upoznavanja sa različitim stavovima. Istovremeno, vežbanje pruža priliku da se provere lične sklonosti nekim tehnikama i neke sopstvene ideje, koje možda i nisu u saglasnosti sa stavovima koji su upoznati iz literature. Bilo da se radi o potencijalnim izvorima grešaka ili vrednim originalnim rešenjima, najbolje je da se to razjasni na vreme, kroz vežbu.

Zbog toga je najbolje da se posle čitanja teksta zadatka ne pređe odmah na čitanje izloženog postupka rešavanja i samog rešenja, već da se najpre pokuša samostalno rešavanje. Ponuđeno rešenje bi trebalo iskoristiti kao podsticaj za razmišljanje, a predstavljen način dolaženja do rešenja kao putokaz koji ukazuje ne uvek na *najbolje*, već pre na *neke* moguće puteve.

Postupak izlaganja rešenja u koracima ima za posledicu da su u nekim koracima namerno pravljeni propusti, koji se kasnije otklanjaju. Zato bi svaki od koraka trebalo čitati veoma pažljivo, a na svakom mestu koje se učini sumnjivim pokušati sagledavanje mogućih posledica i eventualnih boljih rešenja.

Knjiga ne predstavlja referentni priručnik ni za programski jezik C++ ni za objektno orijentisano programiranje. Kao jedan od najpotpunijih uvoda u sve aspekte objektno orijentisanog razvoja softvera, sama se preporučuje knjiga „Objektno orijentisano konstruisanje softvera“ [Meyer 1997]. Nezaobilazne knjige o objektno orijentisanom programiranju su i „Uzorci za projektovanje“ [Gamma 1995] i „Refaktorisanje“ [Fowler 1999]. Jedna od najboljih knjiga o programskom jeziku C++ je knjiga „C++ Izvornik“ [Lippman 2005], u kojoj su detaljno objašnjeni praktično svi aspekti jezika, uključujući i značajan deo standardne biblioteke. Rame uz rame sa njom stoje „Programski jezik C++“ [Stroustrup 2000] i „Misliti na programskom jeziku C++“ [Eckel 2000]. U pregledu literature, na kraju knjige, navedene su još neke knjige koje mogu poslužiti za dalje učenje, kao i dve zbirke jednostavnijih zadataka.

Programski jezik C++ i metodi objektno orijentisanog programiranja jesu složeni i obimni, ali to ne bi trebalo prihvatiti kao ograničavajući već kao oslobađajući faktor pri njihovoj primeni. Mogućnosti su neograničene. Zato ovo obraćanje zaključujem željom da lepo i dobro programirate.

Koristim priliku da se zahvalim studentima, koji su svojim brojnim pitanjima podsticali pisanje ove knjige i posredno uticali na njen sadržaj i način izlaganja. Veliku zahvalnost dugujem svojim saradnicama Jeleni Tomašević i Mileni Vujošević-Janičić, čija su pitanja i sugestije bili od velike pomoći pri pisanju nekih poglavlja. Posebno se zahvaljujem recenzentima dr Miodragu Živkoviću i Aleksandru Samardžiću, koji su svojim sugestijama

doprineli preciznosti i kvalitetu knjige. Na kraju, a zapravo na samom početku, jedno veliko „Hvala!“ dr Nenadu Mitiću na nesebičnoj podršci.

Beograd, 2007. Autor

Sadržaj

Detaljan sadržaj

Spisak tablica

Spisak slika

Deo I

Uvodni primeri

Prvi deo knjige sadrži pet primera, u kojima se demonstrira primena osnovnih elemenata programskog jezika C++ u objektno orijentisanom razvoju.

Pretpostavlja se da čitalac poseduje osnovna znanja o sinataksi programskog jezika C++, pojmu klase, tipovima podataka i upotrebi pokazivača, kao i minimalna predznanja o standardnoj biblioteci programskog jezika C++.

Primer 1 - **Razlomak**

Primer 2 - **Lista**

Primer 3 - **Dinamički nizovi**

Primer 4 - **Perionica automobila**

Primer 5 - **Igra „Život“**

1 - Razlomak

1.1 Zadatak

Napisati klasu `Razlomak` i program koji demonstrira njenu upotrebu. U klasi `Razlomak` obezbediti:

- uobičajene aritmetičke operatore;
- operatore poređenja;
- metode za čitanje i pisanje.

Cilj zadatka

Ovaj primer će nam poslužiti da vidimo kako se koncept klase primenjuje u praksi. Kroz postupno razvijanje klase `Razlomak` ukazaćemo na jedan broj principa čije je poštovanje neophodno da bi rezultat rada bio u skladu sa konceptima objektno orijentisanog programiranja na programskom jeziku C++. Ukazaćemo na sledeće elemente programskog jezika C++:

- skrivanje podataka;
- konstruktore;
- listu inicijalizacija u konstruktorima;
- podrazumevane vrednosti argumenata;
- operatore za čitanje i pisanje;
- operatore kao metode klase;
- operatore konverzije tipova.

Pretpostavljena znanja

Za uspešno praćenje rešavanja ovog zadatka pretpostavlja se poznavanje:

- koncepata objektno orijentisanog programiranja na programskom jeziku C++, uključujući:
  - pojam i način definisanja klase;
  - sintaksu definisanja i upotrebe metoda klase;
  - semantiku pokazivača `this`;
- konstantnih tipova;
- osnovnih principa upotrebe pokazivača i referenci.

1.2 Rešavanje zadatka

Nizom koraka ćemo napraviti i unapređivati klasu `Razlomak`, da bismo na kraju dobili prilično upotrebljivu klasu:

Korak 1 - Opisivanje podataka

Svaki razlomak se sastoji od imenioca i brojioca. Za opisivanje složenih tipova podataka u objektno orijentisanim programskim jezicima, pa i u programskom jeziku C++, upotrebljavaju se klase. Zato je prvi korak pri razvoju našeg tipa `Razlomak` upravo definisanje klase `Razlomak` kao složenog tipa koji se sastoji od celobrojnog imenioca i celobrojnog brojioca.

```
class Razlomak
{
```

```
public:
    //-------------------------------------------------------------
    // Članovi podaci
    //-------------------------------------------------------------
    int Imenilac;
    int Brojilac;
};
```

Upotrebu „klase“ `Razlomak` predstavlja jednostavna funkcija `main`:

```
main()
{
    Razlomak x;
    x.Brojilac = 3;
    x.Imenilac = 2;
    cout << x.Brojilac << '/' << x.Imenilac << endl;

    return 0;
}
```

Korak 2 - Enkapsulacija

Jedan od najvažnijih principa OOP je da *osnovni kriterijum pri pravljenju i oblikovanju klasa predstavlja njihovo ponašanje, a ne njihov sadržaj ili unutrašnja struktura*. Primetimo da je prvi korak u opisivanju razlomka načinjen upravo u smeru definisanja njegove interne strukture. U konkretnom slučaju to ne predstavlja problem, ali, kao što ćemo videti u slučaju nešto složenijih primera, takav početak često može biti neugodna smetnja za dalji rad.

Pri opisivanju klasa moguće je neke delove sakriti od korisnika klase. Izlaganje detalja interne strukture klase korisnicima često ima neugodne posledice. Najneugodnije su:

- otežavanje upoznavanja ponašanja klase i njene upotrebe, jer se povećava izložena količina informacija o klasi;
- otežavanje kasnijih izmena interne strukture klase, jer svaka izmena mora biti praćena ispravljanjem i onih delova programa u kojima je pretpostavljana originalna interna struktura.

Prikrivanje interne strukture se obično naziva učaurivanje ili enkapsulacija. Enkapsulacija podrazumeva da se korisniku predstavljaju samo oni metodi i podaci klase čija je upotreba neophodna za njeno funkcionisanje, dok se svi ostali članovi (metodi i podaci) sakrivaju. Enkapsulacija se u programskom jeziku C++ ostvaruje navođenjem deklaracije raspoloživosti (vidljivosti) članova klase u obliku:

```
<raspoloživost>:
```

gde `<raspoloživost>` može biti:

- `private` - *privatni* članovi su na raspolaganju samo unutar definicija metoda klase;
- `public` - *javni* članovi su raspoloživi svim korisnicima klase;

- `protected` - *zaštićeni* članovi su raspoloživi samo unutar definicija metoda klase i njenih naslednika (o nasleđivanju i klasama naslednicama će više reči biti nešto kasnije, u primeru *4 - Perionica automobila*, na strani 123).

Deklarisana raspoloživost se odnosi na sve članove klase koji su definisani nakon te deklaracije, a pre naredne deklaracije raspoloživosti. Podrazumevana raspoloživost za članove klase u programskom jeziku C++ je privatna, a za članove strukture je javna. Zbog čitljivosti definicija klasa preporučuje se da na početku stoje javni članovi, koji su svima potrebni, a tek za njima zaštićeni i privatni članovi koji nisu na raspolaganju korisnicima klase već samo njenim autorima i autorima klasa naslednica. Uobičajeno je da se privatnim članovima imena određuju tako da njihova priroda bude lako uočljiva. U daljem tekstu ćemo za privatne podatke birati imena koja počinju simbolom '_'.

Kako razlikovati ponašanje od strukture? To može biti sasvim jednostavno, ali i prilično teško. U slučaju razlomaka jasno je da *svaki razlomak ima brojilac i imenilac*, ali nije sasvim očigledno kakve su posledice takve konstatacije. Pretpostavimo da je u opisu zahteva za razvoj klase `Razlomak` navedeno da „korisnik razlomka često ima potrebu da zna vrednost imenioca i brojioca“. Ovakva specifikacija je preciznija jer se odnosi samo na ponašanje – kakva god da je interna struktura razlomka, on mora biti u stanju da korisniku pruži informacije o nečemu što nazivamo *brojilac* i *imenilac*. Iako nije lako zamisliti internu reprezentaciju razlomka koja se ne sastoji upravo od brojica i imenioca, važno je da na ovom mestu apstrahujemo internu strukturu i razumemo da zahtevi koji se odnose na ponašanje ne moraju uvek da se neposredno odražavaju i na internu strukturu. Pretpostavimo da u ovom trenutku nije neophodno omogućiti neposrednu promenu samo imenioca ili samo brojioca. Na to ćemo se vratiti kasnije.

U slučaju razlomaka enkapsulaciju izvodimo tako što podatke `Imenilac` i `Brojilac` proglašavamo za privatne. Kako je čitanje ovih podataka neophodno za upotrebu razlomaka, napisaćemo javne metode koji izračunavaju imenilac i brojilac. Takav pristup (privatni podaci i javni metodi za čitanje) se primenjuje kada god neke podatke korisnici moraju imati na raspolaganju, ali njihovo menjanje ili nije potrebno (što je za sada naš slučaj) ili može zahtevati neke složenije operacije.

```
//-------------------------------------------------------------
//  Klasa Razlomak
//-------------------------------------------------------------
class Razlomak
{
public:
    //---------------------------------------------------------
    // Metodi za pristupanje elementima razlomka.
    //---------------------------------------------------------
    int Imenilac() const
        { return _Imenilac; }
    int Brojilac() const
        { return _Brojilac; }
```

```
private:
    //-------------------------------------------------------------
    // Članovi podaci
    //-------------------------------------------------------------
    int _Imenilac;
    int _Brojilac;
};
```

Kao što se ključnom reči `const` u tipu argumenta funkcije ili metoda naglašava da se u telu funkcije ne menja vrednost tog argumenta, tako se njenim navođenjem na kraju deklaracije metoda naglašava da se izvršavanjem metoda ne menja objekat koji izvršava metod (a koji predstavlja implicitni argument metoda). Metodi kojima je na kraju deklaracije navedena ključna reč `const` nazivaju se *konstantni metodi*. Njihov značaj je u tome da konstantni objekti mogu izvršavati samo konstantne metode.

Kako metodi `Imenilac` i `Brojilac` ne menjaju objekat na kome se izračunavaju, oni su označeni kao konstantni metodi.

Izbor imena

Pri davanju imena klasama, metodima, podacima i drugim celinama, obično se definišu neka pravila. Poštovanje pravila doprinosi čitljivosti programa, čime se olakšava njegovo održavanje. U većini primera u knjizi ćemo se držati nekoliko jednostavnih pravila:

- imena klasa, metoda i članova podataka počinju velikim slovom;
- imena funkcija počinju malim slovom;
- imena lokalnih promenljivih počinju malim slovom;
- ako se ime sastoji od više reči, ne umeću se znaci za razdvajanje reči, već svaka reč započinje velikim slovom;
- privatni članovi podaci imaju prefiks '_'.

Ovaj skup pravila nije ni jedini ni najbolji. Izabran je zbog jednostavnosti i čitljivosti u štampanom obliku. Da bismo ukazali i na neke druge moguće načine imenovanja, u primeru *8 - Kodiranje*, na strani 261., se upotrebljava nešto drugačiji način davanja imena.

U praksi je važno da se pri pisanju većih celina (biblioteka ili programa) autori pridržavaju nekih pravila, dok nije toliko bitno kako će ona glasiti. Zbog toga što pravila nisu pretpostavljena, već svaki razvojni tim obično pravi neki svoj izbor, ona se često nazivaju *dogovorenim pravilima imenovanja* ili *konvencijom imenovanja*.

Korak 3 - Konstruktor

Verujemo da je sasvim očigledno da je klasa koju smo dobili u prethodnom koraku potpuno neupotrebljiva. Naime, ne postoji način za postavljanje vrednosti razlomka. Možemo samo čitati vrednosti brojioca i imenioca. Za sada ćemo se držati iznesene pretpostavke da nije potrebno omogućiti promenu vrednosti razlomka, ali zbog toga nam je neophodno bar da omogućimo određivanje neke inicijalne vrednosti.

Važan princip OOP je da *konstrukcija predstavlja inicijalizaciju*. U svakoj klasi je moguće (i veoma poželjno) definisati jedan ili više metoda koji inicijalizuju nove objekte. Takvi metodi se nazivaju *konstruktori* zato što se izvršavaju u fazi pravljenja novih objekata, neposredno nakon alokacije memorije. Dakle, za inicijalizaciju novih objekata zaduženi su konstruktori.

U programskom jeziku C++ konstruktori se definišu kao metodi čije je ime identično imenu klase. Pri deklarisanju konstruktora se ne navodi tip rezultata.

U slučaju razlomaka ima smisla napraviti bar tri konstruktora: konstruktor razlomka sa datim brojiocem i imeniocem (dva celobrojna argumenta), konstruktor razlomka koji predstavlja ceo broj (jedan celobrojni argument) i konstruktor bez inicijalne vrednosti (bez argumenata):

```
class Razlomak
{
public:
    Razlomak( int b, int i )
        {
        _Brojilac = b;
        _Imenilac = i;
        }

    Razlomak( int b )
        {
        _Brojilac = b;
        _Imenilac = 1;
        }

    Razlomak()
        {
        _Brojilac = 0;
        _Imenilac = 1;
        }
...
};
```

Primer upotrebe ovako definisanih konstruktora mogao bi biti:

```
//---------------------------------------------------------------
// Glavna funkcija programa demonstrira upotrebu klase Razlomak.
//---------------------------------------------------------------
main()
{
    Razlomak a(1,2), b(2), c;
    cout << a.Brojilac() << '/' << a.Imenilac() << endl;
    cout << b.Brojilac() << '/' << b.Imenilac() << endl;
    cout << c.Brojilac() << '/' << c.Imenilac() << endl;

    return 0;
}
```

Podrazumevani konstruktor

Ako se pri definisanju neke klase izostavi eksplicitna definicija konstruktora, prevodilac će automatski generisati tzv. *podrazumevani konstruktor*, kao konstruktor bez argumenata sa

praznim telom metoda. Upotreba podrazumevanog konstruktora ima za rezultat inicijalizaciju svih članova podataka primenom odgovarajućih konstruktora bez argumenata.

Dovoljno je da se u okviru klase definiše neki konstruktor, bilo kakvog tipa, pa da se podrazumevani konstruktor više ne generiše. U tom slučaju, ako je ipak potreban konstruktor bez argumenata, neophodno ga je eksplicitno definisati.

Podrazumevane vrednosti argumenata

U programskom jeziku C++ pisanje više sličnih metoda sa različitim brojem argumenata možemo izbeći primenom *podrazumevanih vrednosti argumenata*. Ako u deklaraciji metoda (ili funkcije) iza imena nekog argumenta (ili iza tipa, ako je ime izostavljeno) navedemo

= *<vrednost>*

tada će se, u slučaju pozivanja metoda bez eksplicitnog navođenja vrednosti tog argumenta, podrazumevati da je njegova vrednost upravo `<vrednost>`. Pri tome je važno poštovati ograničenje da se u metodu sa `k` argumenata nekom argumentu a_i može pridružiti podrazumevana vrednost samo ako se i svim narednim argumentima a_{i+1}, a_{i+2},... a_k takođe pridruže neke podrazumevane vrednosti.

Zbog toga je, umesto tri ranije navedena konstruktora klase `Razlomak`, dovoljno napisati jedan konstruktor za čije su argumente definisane podrazumevane vrednosti:

- ako se navedu oba argumenta, onda su to, redom, vrednosti brojioca i imenioca;
- ako se navede samo jedan argument, onda je to vrednost brojioca, a za imenilac pretpostavljamo da je 1;
- ako se ne navede nijedan argument, pretpostavljamo da je brojilac 0 a imenilac 1:

```
class Razlomak
{
public:
    //-------------------------------------------------------------
    // Konstruktor.
    //-------------------------------------------------------------
    Razlomak( int b=0, int i=1 )
        {
        _Brojilac = b;
        _Imenilac = i;
        }
...
};
```

Prethodni primer upotrebe nije potrebno menjati. Određivanjem podrazumevanih vrednosti argumenata samo smo na jednostavniji način definisali različite načine konstrukcije razlomka, dok upotreba metoda ostaje potpuno ista kao i ranije.

Podrazumevani argumenti se mogu navoditi u svim funkcijama i metodima klasa, osim u operatorima. Pri definisanju i upotrebi operatora mora biti ispoštovan definisan broj argumenata. Ukoliko se funkcija ili metod najpre deklarišu, pa tek zatim i definišu, podrazumevane vrednosti argumenata navedene u deklaraciji se ne smeju ponavljati u

definiciji. Štaviše, mogu se dodati nove, polazeći od poslednjeg argumenta za koji u deklaraciji nije navedena podrazumevana vrednost.

Lista inicijalizacija

U telu konstruktora klase `Razlomak` izvodi se samo jednostavna inicijalizacija podataka koji čine novi razlomak. To je prirodna aktivnost za konstruktore, pa zato postoji i posebna, nešto pojednostavljena, notacija za inicijalizaciju podataka. Inicijalizacija članova podataka se opisuje listom pojedinačnih inicijalizacija u formi:

```
<ime člana>(<parametri>)
```

Lista se navodi ispred tela konstruktora, a od deklaracije se razdvaja dvotačkom. Elementi liste se međusobno razdvajaju zapetama. Neke od najvažnijih osobine liste inicijalizacija su:

- u listi se smeju navesti isključivo imena baznih klasa (o tome nešto kasnije) i imena članova podataka koji su eksplicitno deklarisani u klasi u čijem se konstruktoru lista nalazi;
- ako se element liste odnosi na podatak prostog tipa, dopušten je najviše jedan parametar istog tipa koji predstavlja inicijalnu vrednost podatka;
- ako se element liste odnosi na podatak klasnog tipa, parametri moraju brojem i tipom odgovarati argumentima tačno jednog konstruktora definisanog za tu klasu;
- upotreba liste inicijalizacija je efikasnija nego dodeljivanje vrednosti u telu konstruktora – lista navodi kako se podaci članovi konstruišu, dok se dodeljivanjem naknadno menja vrednost već konstruisanim objektima, što je bespotrebno ponavljanje posla;
- redosled izvođenja inicijalizacija ne zavisi od redosleda navođenja u listi, već samo od redosleda navođenja baznih klasa i članova podataka u okviru deklaracije klase o čijem se konstruktoru radi – najpre se inicijalizuju bazne klase (u redosledu u kome su navedene u deklaraciji klase), a zatim članovi podaci (u redosledu u kome su navedeni u deklaraciji klase).

Upotrebom liste inicijalizacija dalje popravljamo konstruktor klase `Razlomak`:

```
class Razlomak
{
public:
    //-------------------------------------------------------------
    // Konstruktor.
    //-------------------------------------------------------------
    Razlomak( int b=0, int i=1 )
        : _Brojilac(b), _Imenilac(i)
        {}
...
};
```

Konceptu metoda konstruktora, koji obezbeđuju inicijalizaciju objekata pri njihovom pravljenju, odgovara koncept metoda *destruktora*, koji obezbeđuju deinicijalizaciju objekata pri

njihovom uklanjanju. Kako naši razlomci predstavljaju relativno jednostavan primer, ne postoji stvarna potreba za deinicijalizacijom pri njihovom uklanjanju. Zbog toga će koncept destruktora i drugih pratećih metoda biti opisan u primeru *2 - Lista*, na strani 44.

```
//-------------------------------------------------------------
//  Klasa Razlomak
//-------------------------------------------------------------
class Razlomak
{
public:
    //---------------------------------------------------------
    // Konstruktor. Destruktor nije potreban.
    //---------------------------------------------------------
    Razlomak( int b = 0, int i = 1 )
        : _Brojilac(b), _Imenilac(i)
        {}

    //---------------------------------------------------------
    // Metodi za pristupanje elementima razlomka.
    //---------------------------------------------------------
    int Imenilac() const
        { return _Imenilac; }
    int Brojilac() const
        { return _Brojilac; }

private:
    //---------------------------------------------------------
    // Članovi podaci
    //---------------------------------------------------------
    int _Imenilac;
    int _Brojilac;
};
```

Kako funkcionalnost ni sada nije izmenjena, kao primer upotrebe možemo i dalje koristiti istu funkciju `main`.

Korak 4 - Čitanje i pisanje

Za čitanje objekata iz toka i zapisivanje objekata u tok uobičajeno se primenjuju operatori << i >>. Predefinisanje operatora u programskom jeziku C++ je relativno jednostavno. Izvodi se na skoro isti način kao i definisanje funkcija i metoda, uz svega nekoliko specifičnosti:

- kao ime funkcije (metoda) se navodi ključna reč `operator` iza koje sledi zapis operatora;
- asocijativnost operatora i broj operanada (tj. argumenata odgovarajuće funkcije) se ne mogu menjati;
- ako se operator definiše kao metod klase, tada se podrazumeva da je prvi operand upravo objekat klase pa se u deklaraciji navodi jedan argument manje.

Operatore je preporučljivo definisati unutar klase, kada god je to moguće.

Kada su u pitanju operatori ulaza i izlaza, potrebno je da se pri njihovom definisanju za nove tipove podataka ispoštuje nekoliko važnih pravila, da bi se novi operatori mogli koristiti

na potpuno isti način kao i oni koji su definisani u standardnoj biblioteci (videti *10.10 Biblioteka tokova*, na strani 383):

- operatore je potrebno definisati za najopštiji tip ulaznog toka (`istream`) i najopštiji tip izlaznog toka (`ostream`) – ovime se obezbeđuje da napisani operatori funkcionišu na svim vrstama tokova, uključujući konzolu, datoteke, memorijske tokove i druge;
- prvi argument ovih operatora je tok, a drugi je objekat koji se zapisuje ili čita – tako se upotrebljavaju već definisani operatori za druge tipove podataka;
- operatori kao rezultat vraćaju tok – to je neophodno da bi se njihova upotreba mogla nadovezivati;
- operator pisanja mora prihvatati konstantne objekte – pisanje ne sme menjati objekat i nema razloga da ne funkcioniše za konstantne objekte;
- tokove je neophodno, a objekte uobičajeno prenositi po referenci – pri čitanju se objekat mora prenositi po referenci (ili primenom pokazivača, ali to nepotrebno komplikuje sintaksu pa ovde neće biti detaljnije obrađivano) da bi uopšte mogao biti menjan, a pri pisanju se tako obezbeđuje efikasniji rad;
- čitanje i pisanje moraju biti međusobno usklađeni, tj. ako neki objekat A zapišemo u tok, a zatim iz tog zapisa pročitamo neki objekat B, tada A i B moraju biti identični – ovo je verovatno najznačajnije pravilo, jer se njegovim narušavanjem dovodi u pitanje smisao operacija čitanja i pisanja.

Navedena pravila vrlo precizno određuju tipove ovih operatora za našu klasu `Razlomak`:

```
ostream& operator << ( ostream& str, const Razlomak& r )
istream& operator >> ( istream& str, Razlomak& r )
```

Dok bismo operator pisanja mogli relativno lako implementirati, sa čitanjem može biti određenih problema. Naime, pri čitanju je potrebno menjati imenilac i brojilac razlomka, a van same klase to nije moguće jer su u pitanju privatni podaci. Ovakav problem se ispoljava veoma često. Zbog toga je dobro da se čitanje i pisanje implementiraju kao metodi klase. Primetimo, međutim, da se operatori << i >> ne mogu implementirati kao metodi klase `Razlomak`, jer im prvi argument nije objekat klase `Razlomak`. Da je kojim slučajem naša klasa `Razlomak` deo standardne biblioteke, tada bi možda imalo smisla ove operatore implementirati, redom, u klasama `ostream` i `istream`. Ovako, preostaje nam da ih implementiramo kao funkcije koje pozivaju odgovarajuće metode klase `Razlomak`:

```
class Razlomak
{
public:
...
    //-------------------------------------------------------------
    // Pisanje i čitanje.
    //-------------------------------------------------------------
    ostream& Pisi( ostream& str ) const
        { return str << _Brojilac << '/' << _Imenilac; }
```

```
        istream& Citaj( istream& str )
            {
            // Za sada ne proveravamo da li su brojilac i imenilac
            // razdvojeni upravo simbolom /
            char c;
            return str >> _Brojilac >> c >> _Imenilac;
            }
...
};

//------------------------------------------------------------
//  Operatori za pisanje i čitanje razlomaka.
//------------------------------------------------------------
ostream& operator<< ( ostream& str, const Razlomak& r )
    { return r.Pisi( str ); }
istream& operator>> ( istream& str, Razlomak& r )
    { return r.Citaj( str ); }
```

Preostaje nam da izmenimo funkciju `main` kako bismo isprobali nove mogućnosti klase `Razlomak`:

```
main()
{
    Razlomak a(1,2), b(-1,-3), c(2), d;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    cout << d << endl;

    cout << "Upisite razlomak oblika a/b: ";
    cin >> a;
    cout << a << endl;

    return 0;
}
```

Primetimo da su operacije pisanja i čitanja implementirane definisanjem po jednog metoda i operatora. U ovom slučaju, to je urađeno zbog toga što se u oba slučaja pristupa privatnim podacima klase, što nije moguće učiniti neposredno iz tela operatora koji ne pripadaju klasi. Zapravo, pisanje se u slučaju klase `Razlomak` može implementirati i korišćenjem javnih metoda, ali to nije uvek slučaj. Da bi se izbeglo nepotrebno pisanje većeg broja celina u sličnim situacijama, programski jezik C++ raspolaže konceptom *prijateljskih* klasa i funkcija.

Klase i funkcije koje su proglašene za *prijatelje* neke klase imaju pun pristup svim privatnim i zaštićenim elementima te klase. U našem primeru, ako bismo operatore proglasili za prijatelje klase `Razlomak`, iz njihovog tela bi se moglo neposredno pristupati imeniocu i brojiocu. Klasa ili funkcija se proglašava za prijatelja navođenjem odgovarajuće deklaraciju u okviru klase, u obliku:

```
friend class <ime klase>;
friend <deklaracija funkcije>;
```

U slučaju klase `Razlomak`, implementacija operacija čitanja i pisanja korišćenjem prijateljskih operatora bi izgledala ovako:

```
class Razlomak
{
public:
...
    //-------------------------------------------------------------
    // Pisanje i čitanje.
    //-------------------------------------------------------------
    friend ostream& operator<< ( ostream& str, const Razlomak& r );
    friend istream& operator>> ( istream& str, Razlomak& r );
...
};

//-----------------------------------------------------------------
//  Operatori za pisanje i čitanje razlomaka.
//-----------------------------------------------------------------
ostream& operator<< ( ostream& str, const Razlomak& r )
    { return str << r._Brojilac << '/' << r._Imenilac; }
istream& operator>> ( istream& str, Razlomak& r )
    {
    char c;
    return str >> r._Brojilac >> c >> r._Imenilac;
    }
```

Postoje bar dva dobra argumenta protiv korišćenja prijateljskih funkcija za čitanje i pisanje. Prvi argument počiva na činjenici da upotreba prijateljskih funkcija i klasa narušava enkapsulaciju. Oko njega se autori često spore, zbog toga što takvo narušavanje enkapsulacije, u konkretnom slučaju, nije značajno niti ugrožava pouzdanost klase, jer operatori čitanja i pisanja, iako se fizički definišu van klase, njoj logički pripadaju. Sa druge strane, svaka iole intenzivnija primena prijateljskih klasa i funkcija sigurno nije dobra i može se tumačiti kao upozorenje da projekat klasa, pa i čitavog programa, nije dobar.

Drugi argument je daleko praktičnije prirode i odnosi se na potrebu da (bar u fazi učenja) jednu istu funcionalnost implementiramo na jedan način. Raznovrsnost u rešavanju ponovljenih problema možda može imati svoju draž za programera, koji bi tu mogao pokazati i znanje i kreativnost, ali otežava kasnije čitanje, razumevanje i održavanje programa. U narednim primerima ćemo videti da se u slučaju hijerarhija klasa operacije pisanja moraju bar jednim delom implementirati kao metodi klasa.

Imajući sve to u vidu, nadalje ćemo se pridržavati prvog predstavljenog načina implementiranja operacija čitanja i pisanja – bez deklaracija prijatelja klase.

Korak 5 - Aritmetičke operacije

Kao što je već naglašeno, pri definisanju klasa se rukovodimo željenim ponašanjem. Verovatno najznačajniji aspekt ponašanja razlomaka jeste izračunavanje aritmetičkih operacija. Da bismo mogli definisati aritmetičke operacije najpre je potrebno da vidimo na koje je sve načine to moguće učiniti u programskom jeziku C++. Poći ćemo od načina koji najviše podseća na pisanje funkcija u programskom jeziku C, da bismo postepeno uvodili specifičnosti programskog jezika C++ i dobili bolje rešenje.

Funkcija koja sabira razlomke može se napisati tako da najpre pravimo novi objekat, inicijalizovan odgovarajućim vrednostima brojioca i imenioca, a zatim taj objekat vratimo kao rezultat funkcije:

```
Razlomak Saberi( Razlomak x, Razlomak y )
{
    Razlomak r(
        x.Brojilac() * y.Imenilac() + x.Imenilac() * y.Brojilac(),
        x.Imenilac() * y.Imenilac()
        );
    return r;
}
```

Ovo je sasvim ispravno rešenje, koje se za proizvoljne razlomke `x`, `y` i `z` može upotrebljavati u obliku:

```
z = Saberi( x, y );
```

Iako je ispravno, ovo rešenje ima i neke loše strane:

- svaki put pri pozivanju funkcije `Saberi` kopiraju se čitavi objekti `x` i `y`;
- svaki put pri vraćanju rezultata kopira se objekat `r`;
- sabiranje razlomaka je deo njihovog ponašanja, pa mu je mesto u klasi `Razlomak`;
- sintaksa funkcije je nepraktična u složenijim izrazima.

Prve dve mane se odnose na sve funkcije u kojima su argumenti i/ili rezultat nekog klasnog tipa. Kako objekti klasa mogu biti prilično glomazni, jasno je da njihovo kopiranje troši dragocene resurse. Dok bismo se u slučaju programskog jezika C verovatno odlučili za prenošenje argumenata posredstvom pokazivača, u slučaju programskog jezika C++ ćemo radije upotrebiti reference. Dakle, prva mana se jednostavno odstranjuje prenošenjem argumenata po referenci:

```
Razlomak Saberi( Razlomak& x, Razlomak& y )
{
    Razlomak r(
        x.Brojilac() * y.Imenilac() + x.Imenilac() * y.Brojilac(),
        x.Imenilac() * y.Imenilac()
        );
    return r;
}
```

Međutim, sada smo napravili novi problem – sabiranje neće raditi za konstantne objekte jer deklaracija funkcije nagoveštava da ona menja argumente. Zbog toga je neophodno da izmenimo tip argumenata tako da bude jasno da se oni neće menjati:

```
Razlomak Saberi( const Razlomak& x, const Razlomak& y )
{
    Razlomak r(
        x.Brojilac() * y.Imenilac() + x.Imenilac() * y.Brojilac(),
        x.Imenilac() * y.Imenilac()
        );
```

```
    return r;
}
```

Upotrebom ključne reči `const` naglašavamo da se argumenti ne menjaju u našoj funkciji, pa se ona zbog toga može primenjivati i na konstantne objekte. Ukoliko, ipak, pokušamo da u funkciji promenimo podatak koji je deklarisan kao konstantan, to će se već u fazi prevođenja programa prepoznati kao greška.

Da bismo izbegli suvišno kopiranje rezultata sabiranja, oslonićemo se na važnu osobinu programskog jezika C++: *konstruktori se mogu upotrebljavati kao funkcije čiji je rezultat novi objekat.* Znajući to, možemo napisati prethodnu funkciju bez pomoćnog objekta `r`:

```
Razlomak Saberi( const Razlomak& x, const Razlomak& y )
{
    return Razlomak(
        x.Brojilac() * y.Imenilac() + x.Imenilac() * y.Brojilac(),
        x.Imenilac() * y.Imenilac()
        );
}
```

Ovo je najviše što možemo postići pisanjem funkcije `Saberi`. Zbog toga što rezultat sabiranja predstavlja novi objekat, on se nikako ne sme vratiti po referenci. Ako bismo pokušali da rezultat vratimo po referenci to bi imalo fatalne posledice po izvršavanje programa, iako neki prevodioci ne bi prijavili ni grešku ni upozorenje. Naime, po referenci bi bio vraćen privremeni objekat, koji nakon završetka rada funkcije više ne bi postojao. Tako bi vraćena referenca bila potpuno neispravna jer bi ukazivala na objekat koji više ne postoji. Dakle, *iz funkcija i metoda se nikako ne sme vraćati referenca ili pokazivač na privremeni (lokalni) objekat*.

Kao treću primedbu naveli smo da je sabiranju mesto u klasi `Razlomak`. Zaista, *sve što se odnosi na ponašanje objekata neke klase trebalo bi da se nalazi upravo u okviru definicije klase, ako je to moguće*. Već smo na primeru operatora čitanja i pisanja videli da nije uvek moguće ispoštovati ovaj princip, ali većinu situacija, pa i ovu sa sabiranjem, je moguće razrešiti na odgovarajući način. Opisivanje sabiranja u okviru klase `Razlomak` implementiramo pisanjem metoda `SaberiSa`. Novi metod ima isti tip rezultata, ali jedan argument manje. Podsetimo se da se pri upotrebi metoda uvek podrazumeva jedan implicitan argument – objekat koji izvršava metod. Zbog toga se metod `SaberiSa` definiše ovako:

```
class Razlomak
{
...
    Razlomak SaberiSa( const Razlomak& r )
        {
        return Razlomak (
            Brojilac() * r.Imenilac() + Imenilac() * r.Brojilac(),
            Imenilac() * r.Imenilac()
            );
        }
...
};
```

i upotrebljava se u obliku:

```
z = x.SaberiSa( y );
```

Primetimo da nam se negde usput izgubila naznaka da je prvi operand konstantan, tj. da neće biti menjan. Kako je sada prvi operand upravo objekat koji izvršava metod, njegovu konstantnost (tj. činjenicu da ga izvršavanje metoda ne menja) ističemo na isti način kao i u slučaju metoda `Brojilac` i `Imenilac` – navođenjem ključne reči `const` na kraju deklaracije metoda:

```
Razlomak SaberiSa( const Razlomak& r ) const
```

Operatori

Da bismo obezbedili prihvatljiviju sintaksu sabiranja razlomaka, preostaje nam još da umesto funkcije definišemo operator. Kao što smo već videli u slučaju operatora čitanja i pisanja, to se postiže sasvim jednostavno, izborom imena funkcije koje počinje ključnom reči `operator` iza koje sledi zapis operatora. Pri izboru operatora valja se rukovoditi pravilom: *operatore bi trebalo definisati i njihova imena birati samo u skladu sa uobičajenim načinom razmišljanja o objektima koje opisujemo*. Na primer, nikako ne bi bilo dobro za operaciju sabiranja upotrebiti operator `*`.

Konačno, dolazimo do završnog oblika operacije sabiranja:

```
class Razlomak
{
...
    Razlomak operator + ( const Razlomak& r ) const
        {
        return Razlomak (
            Brojilac() * r.Imenilac() + Imenilac() * r.Brojilac(),
            Imenilac() * r.Imenilac()
            );
        }
...
};
```

Sada se razlomci mogu sabirati na sasvim jednostavan način, onako kako je to već uobičajeno za celobrojne i realne podatke:

```
z = x + y;
```

Na sličan način implementiramo i oduzimanje, množenje i deljenje:

```
class Razlomak
{
public:
...
    //-------------------------------------------------------------
    // Binarne aritmetičke operacije.
    //-------------------------------------------------------------
    Razlomak operator + ( const Razlomak& r ) const
        {
```

```
        return Razlomak (
            Brojilac() * r.Imenilac() + Imenilac() * r.Brojilac(),
            Imenilac() * r.Imenilac()
            );
        }

    Razlomak operator - ( const Razlomak& r ) const
        {
        return Razlomak(
            Brojilac() * r.Imenilac() - Imenilac() * r.Brojilac(),
            Imenilac() * r.Imenilac()
            );
        }

    Razlomak operator * ( const Razlomak& r ) const
        {
        return Razlomak(
            Brojilac() * r.Brojilac(),
            Imenilac() * r.Imenilac()
            );
        }

    Razlomak operator / ( const Razlomak& r ) const
        {
        return Razlomak(
            Brojilac() * r.Imenilac(),
            Imenilac() * r.Brojilac()
            );
        }
...
};
```

Definisaćemo još i dve unarne operacije koje imaju smisla za razlomke: promenu znaka i recipročnu vrednost. Za promenu znaka upotrebićemo unarni operator -, a za recipročnu vrednost operator ~:

```
class Razlomak
{
public:
...
    //-------------------------------------------------------
    // Unarne aritmetičke operacije.
    //-------------------------------------------------------
    Razlomak operator - () const
        { return Razlomak( -Brojilac(), Imenilac() ); }

    Razlomak operator ~ () const
        { return Razlomak( Imenilac(), Brojilac() ); }
...
};
```

Navedimo primere upotrebe ovih operacija u okviru funkcije `main`:

```
main()
{
    Razlomak a(1,2), b(2,3);
    cout << a << " + " << b << " = " << (a+b) << endl;
    cout << a << " - " << b << " = " << (a-b) << endl;
```

```
    cout << a << " * " << b << " = " << (a*b) << endl;
    cout << a << " / " << b << " = " << (a/b) << endl;
    cout << "- " << b << " = " << (-b) << endl;
    cout << "~ " << b << " = " << (~b) << endl;

    return 0;
}
```

Korak 6 - Operacije poređenja

Razlomci predstavljaju objekte koje ima smisla porediti. Definisaćemo sve uobičajene operatore poređenja. Oni se ne razlikuju značajno od već implementiranih operatora, pa se zbog toga na njima nećemo detaljnije zadržavati. Rezultati svih ovih operacija su logičkog tipa. Primetimo da su implementirane uz primenu množenja umesto deljenja, kako bi se očuvala preciznost. Iako preciznije i efikasnije, to može imati posledice po opseg imenilaca i brojilaca za koje se dobijaju ispravni rezultati poređenja:

```
class Razlomak
{
public:
...
    //-------------------------------------------------------------
    // Operacije poređenja.
    //-------------------------------------------------------------
    bool operator == ( const Razlomak& r ) const
        {
        return Brojilac() * r.Imenilac()
            == Imenilac() * r.Brojilac();
        }

    bool operator != ( const Razlomak& r ) const
        { return !( *this == r ); }

    bool operator > ( const Razlomak& r ) const
        {
        return Brojilac() * r.Imenilac()
             > Imenilac() * r.Brojilac();
        }

    bool operator >= ( const Razlomak& r ) const
        {
        return Brojilac() * r.Imenilac()
            >= Imenilac() * r.Brojilac();
        }

    bool operator < ( const Razlomak& r ) const
        {
        return Brojilac() * r.Imenilac()
             < Imenilac() * r.Brojilac();
        }

    bool operator <= ( const Razlomak& r ) const
        {
        return Brojilac() * r.Imenilac()
            <= Imenilac() * r.Brojilac();
        }
...
};
```

Ispravnost možemo proveriti ovako:

```
main()
{
...
    cout << a << " == " << a << " = " << (a==a) << endl;
    cout << a << " == " << b << " = " << (a==b) << endl;
    cout << a << " < " << b << " = " << (a<b) << endl;
    cout << a << " > " << b << " = " << (a>b) << endl;
...
}
```

Korak 7 - Operatori inkrementiranja i dekrementiranja

Videli smo kako se implementiraju aritmetički operatori i operatori poređenja. Sada ćemo implementirati i operatore ++ i --. Definisanje ovih operatora ima neke specifičnosti, kao uostalom i njihova upotreba:

- svaki od ovih operatora ima po dve forme: prefiksnu i postfiksnu – svaka od njih se posebno definiše;
- ako se definiše samo prefiksni operator, on može da se upotrebljava i u postfiksnoj formi;
- ako se definiše samo postfiksni operator, on se ne može upotrebljavati u prefiksnoj formi;
- ako se definišu i prefiksni i postfiksni operator, nije dobro da oni imaju različitu funkciju;
- pri implementiranju je potrebno voditi računa o prefiksnoj/postfiksnoj semantici.

Najpre sledi primer implementacije ovih operatora, a zatim i neophodna objašnjenja:

```
class Razlomak
{
...
    //-------------------------------------------------------------
    // Operatori inkrementiranja i dekrementiranja
    //-------------------------------------------------------------
    // prefiksno ++
    Razlomak& operator ++ ()
        {
        _Brojilac += _Imenilac;
        return *this;
        }
    // postfiksno ++
    Razlomak operator ++ (int)
        {
        Razlomak r = *this;
        _Brojilac += _Imenilac;
        return r;
        }
```

```
    // prefiksno --
    Razlomak& operator -- ()
        {
        _Brojilac -= _Imenilac;
        return *this;
        }

    // postfiksno –
    Razlomak operator -- (int)
        {
        Razlomak r = *this;
        _Brojilac -= _Imenilac;
        return r;
        }
...
};
```

Primetimo da se ovi operatori razlikuju po broju argumenata. Postfiksni operatori imaju jedan *lažni* argument samo da bi se mogli razlikovati od prefiksnih. Tip rezultata, u načelu, nije značajan i u oba slučaja može biti sa ili bez reference. Primetimo i da ovi operatori nisu konstantni, jer menjaju objekte koji ih izvršavaju.

U slučaju prefiksnih operatora najpre je izmenjena vrednost brojioca, pa je zatim vraćen izmenjen objekat po referenci. Ovde se rezultat može vratiti po referenci jer se ne radi o privremenom podatku već o objektu koji nastavlja da živi i nakon pozivanja ovog metoda. U slučaju postfiksnih operatora, najpre se sačuva originalna vrednost objekta u pomoćnoj promenljivoj, pa se zatim menja vrednost brojioca, da bi se na kraju sačuvana originalna vrednost vratila kao rezultat. Zbog toga što je rezultat zapisan u privremenom objektu `r`, ovde se rezultat ne sme vratiti po referenci. Predstavljenom implementacijom je ispoštovana semantika ovih operatora, da se u prefiksnom slučaju kao rezultat dobija izmenjena a u postfiksom slučaju originalna vrednost objekta.

Korak 8 - Konverzija

Definisanje klase `Razlomak` završićemo definisanjem metoda za konverziju u tip `double` i diskusijom metoda za konverziju.

Najpre primetimo da možemo sasvim jednostavno napisati metod `Double` koji vraća količnik brojioca i imenioca. Pri tome je neophodno da bar jedan od operanada deljenja eksplicitno konvertujemo u tip `double`, jer bi se inače primenilo celobrojno a ne realno deljenje:

```
class Razlomak
{
...
    //-------------------------------------------------------
    //  Konverzija u tip double
    //-------------------------------------------------------
    double Double() const
        { return _Brojilac / (double)_Imenilac; }
...
};
```

Operatori konverzije

Mada je ovaj metod sasvim ispravan, može da smeta što se upotrebljava drugačije nego uobičajene eksplicitne konverzije tipova. Programski jezik C++ omogućava definisanje posebnih operatora za konverziju čija je upotreba ista kao u slučaju uobičajenih konverzija. Potrebno je samo da se definiše metod koji:

- za ime ima ključnu reč `operator` iza koje stoji ime tipa u koji se vrši konverzija;
- nema argumente;
- nema deklarisan tip rezultata;
- najčešće je konstantan.

Tako umesto prethodnog metoda možemo napisati odgovarajući operator za konverziju:

```
class Razlomak
{
...
    //-------------------------------------------------------------
    //  Konverzija u tip double
    //-------------------------------------------------------------
    operator double () const
        { return _Brojilac / (double)_Imenilac; }
...
};
```

Ovaj operator će se automatski upotrebljavati na svim mestima na kojima se navede neki od uobičajenih izraza za konverziju u programskim jezicima C i C++:

```
double(a)
(double)a
static_cast<double>(a)
```

Na ovaj način smo postigli da se razlomak može konvertovati u tip `double`. Međutim, sasvim je verovatno da je čak i pažljivim čitaocima promakla činjenica da je u priči o razlomcima obezbeđen još jedan konvertor, i to implicitan. Radi ilustracije, za trenutak obrišimo (ili još bolje, iskomentarišimo) operator `double()` i primetimo da će naredni izraz biti potpuno ispravan i dati očekivan rezultat:

```
cout << ( Razlomak(1,5) + 2 ) << endl;
```

Konstruktor kao operator konverzije

Kako to radi kada nismo definisali operator sabiranja koji radi za razlomke i cele brojeve? Zanimljiva osobina programskog jezika C++ je da se svaki konstruktor sa jednim argumentom (ili sa više argumenata za koje postoje neke podrazumevane vrednosti) može upotrebiti kao operator konverzije iz tipa argumenta u tip klase čiji je konstruktor u pitanju. Štaviše, i tako napisani konstruktori i napisani operatori za konverziju ne primenjuju se samo eksplicitno (kao u prethodnom primeru upotrebe našeg operatora konverzije `double`), nego i implicitno – kada god prevodilac nije u stanju da pronađe neki metod ili operator odgovarajućeg tipa, pokušava se sa primenom raspoloživih konverzija kako bi se izraz uspešno preveo. U našem

primeru prevodilac nailazi na problematičan operator +, koji nije definisan za prvi operand tipa `Razlomak` i drugi operand tipa `int`. Zbog toga pokušava da pronađe konverziju koja će omogućiti primenu nekog od postojećih operatora. Konkretan primer će se prevesti doslovno kao da je napisano:

```
cout << ( Razlomak(1,5) + Razlomak(2) ) << endl;
```

A zašto smo morali da obrišemo definiciju operatora konverzije `double` da bismo prepoznali takvo ponašanje? Zato što prevodilac preduzima implicitno konvertovanje samo ako postoji *tačno jedno* moguće tumačenje izraza. Ako ima više potencijalnih tumačenja prevodilac prijavljuje grešku dvosmislenosti izraza. U konkretnom slučaju, u prisustvu operatora konverzije `double` bila bi prijavljena greška jer bi postojala dva moguća tumačenja izraza:

```
cout << ( Razlomak(1,5) + Razlomak(2) ) << endl;
cout << ( double(Razlomak(1,5)) + double(2) ) << endl;
```

Na kraju priče o konverzijama primetimo da često nije dobro da prevodilac samostalno primenjuje konverzije u nekim spornim situacijama. Sa operatorima konverzije je jednostavno izaći na kraj – ako ne želimo da se konverzije primenjuju implicitno, možemo umesto operatora konverzije definisati odgovarajući metod. Sa konstruktorima je nešto složenija situacija, jer od njih ne možemo tek tako odustati. Zbog toga je standardom programskog jezika C++ predviđena ključna reč `explicit`, čijim se navođenjem ispred problematične deklaracije konstruktora naglašava da nije dozvoljena njegova implicitna upotreba od strane prevodioca u cilju konvertovanja podataka, već da se sme upotrebljavati samo eksplicitnim navođenjem od strane programera. Ako bismo upotrebili ključnu reč `explicit` ispred konstruktora klase `Razlomak`:

```
class Razlomak
{
...
    explicit Razlomak( int b = 0, int i = 1 )
        : _Brojilac(b), _Imenilac(i)
        {}
...
};
```

tada bi sporni izraz bio protumačen na jedini mogući način:

```
cout << ( double(Razlomak(1,5)) + double(2) ) << endl;
```

Korak 9 - Robusnost

Uvek pri pisanju programa, a posebno pri pisanju klasa koje se mogu upotrebljavati u više programa, potrebno je voditi računa o mogućim problemima do kojih može doći zbog neispravnih podataka. Sposobnost programa ili dela programa da *preživi* neispravnu upotrebu i/ili neispravne podatke naziva se *robusnost programa*.

Iako u osnovne principe programiranja u programskim jezicima C i C++ spada i pretpostavka da se zbog efikasnosti staranje o opsegu obično prepušta programeru (što u

našem slučaju znači „korisniku klase `Razlomak`"), ipak je neophodno eliminisati što više izvora potencijalnih problema. Ako sprečavanje neke vrste problema neminovno dovodi do značajnih negativnih posledica po efikasnost, tada je potrebno doneti odluku šta je u konkretnom slučaju važnije. Pri tome valja imati na umu da je program koji daje neispravan rezultat najčešće potpuno bezvredan, bez obzira na to koliko se brzo taj neispravan rezultat dobija. Doduše, nisu retke ni situacije u kojima je program koji daje tačan rezultat takođe potpuno bezvredan, ukoliko taj rezultat daje uz neprihvatljiv utrošak računarskih resursa.

Analizom napisane klase `Razlomak` možemo uočiti sledeće slabosti koje mogu dovesti do neispravnog rezultata:

- operatori <, <=, > i >= ne daju tačne rezultate ako je imenilac nekog operanda negativan;
- operatori ++ i -- ne daju tačne rezultate ako je imenilac negativan;
- aritmetičke operacije i operacije poređenja mogu dati neispravne rezultate ukoliko neki od međurezultata ispadne iz opsega celobrojnog tipa koji odgovara imeniocu i brojiocu;
- ne postoji provera ispravnosti zapisa pri čitanju razlomka iz toka;
- postoji mogućnost definisanja razlomka sa imeniocem 0.

Prva dva problema se mogu rešiti uvođenjem pravila da imenioci moraju biti pozitivni. Kako se vrednost imenioca postavlja samo u konstruktoru i metodu `Citaj`, potrebno je u njima obezbediti odgovarajuću proveru i korekciju podataka. Takva dopuna nije posebno skupa a veoma je značajna, pa ćemo je ugraditi u našu klasu:

```
class Razlomak
{
public:
...
    explicit Razlomak( int b = 0, int i = 1 )
        : _Brojilac(b), _Imenilac(i)
        {
        if( _Imenilac < 0 ){
            _Brojilac = - _Brojilac;
            _Imenilac = - _Imenilac;
            }
        }
...
    istream& Citaj( istream& str )
        {
        char c;
        str >> _Brojilac >> c >> _Imenilac;
        if( _Imenilac < 0 ){
            _Brojilac = - _Brojilac;
            _Imenilac = - _Imenilac;
            }
        return str;
        }
...
};
```

Problemi sa prekoračenjem opsega se mogu u priličnoj meri umanjiti uvođenjem dodatnog zahteva, da razlomak uvek bude uprošćen koliko je to moguće (tj. da imenilac i brojilac budu uzajamno prosti). Za razliku od prethodnog, ovaj zahtev je relativno skup, jer je svaki put pri konstrukciji razlomka potrebno proveravati da li su imenilac i brojilac uzajamno prosti i uprošćavati ih ako nisu. U ovom primeru ćemo se ipak odlučiti i za tu izmenu. Radi toga ćemo napisati funkciju `nzd`, koja izračunava najveći zajednički delilac dva cela broja, i metod `PopraviPodatke` koji preduzima sve potrebne korekcije imenioca i brojioca, uključujući i prethodno opisanu korekciju u slučaju negativnog imenioca. Metod `PopraviPodatke` je privatan i koristi ga samo klasa `Razlomak` u konstruktoru i metodu `Citaj`:

```
//-----------------------------------------------------------------
//  Funkcija nzd računa najveći zajednički delilac
//-----------------------------------------------------------------
int nzd( int a, int b )
{
    int t;
    //  postaramo se da važi a>=b
    if( a < b ){
        t = b;
        b = a;
        a = t;
        }
    //  stalno održavajući a>=b približavamo se rešenju
    while( b > 0 ){
        t = b;
        b = a % b;
        a = t;
        }
    //  ako je b = 0, to je zato što smo delili sa a bez ostatka
    return a;
}
...
class Razlomak
{
public:
...
    explicit Razlomak( int b = 0, int i = 1 )
        : _Brojilac(b), _Imenilac(i)
        { PopraviPodatke(); }
...
    istream& Citaj( istream& str )
        {
        char c;
        str >> _Brojilac >> c >> _Imenilac;
        PopraviPodatke();
        return str;
        }
...
```

```
private:
    //-------------------------------------------------------------
    // Korigovanje podataka
    //-------------------------------------------------------------
    void PopraviPodatke()
        {
        // sprečavanje negativnog imenioca
        if( _Imenilac < 0 ){
            _Brojilac = - _Brojilac;
            _Imenilac = - _Imenilac;
            }
        // skraćivanje imenioca i brojioca
        int n = nzd( abs(_Brojilac), _Imenilac );
        if( n>1 ){
            _Imenilac /= n;
            _Brojilac /= n;
            }
        }
...
};
```

Sada je poređenje operatorima == i != nezavisno od opsega:

```
class Razlomak
{
...
    bool operator == ( const Razlomak& r ) const
        {
        return Brojilac() == r.Brojilac()
            && Imenilac() == r.Imenilac();
        }
...
};
```

Pri izvršavanju drugih operacija i dalje može doći do prekoračenja opsega celih brojeva, ali će se to dešavati u manjem broju slučajeva, jer su razlomci uprošćeni.

Ispravnost čitanja i pisanja se obično proverava tako što se neispravnosti signaliziraju označavanjem da je operacija prevela tok u neispravno stanje. Pri pisanju obično nisu potrebne nikakve posebne provere jer je u slučaju greške tok već označen kao neispravan. U slučaju čitanja može doći do više problema:

- tok je u stanju EOF (kraj toka, tj. nema dovoljno podataka u toku);
- tok je u neispravnom stanju;
- tok je u ispravnom stanju, ali njegov sadržaj ne odgovara očekivanjima.

Prva dva problema se uglavnom mogu ignorisati, jer činjenica da tok nakon okončavanja naše operacije čitanja ima neispravno stanje (ili stanje EOF) neposredno ukazuje i na to da naše čitanje nije uspešno okončano. Samo treći slučaj bi trebalo da eksplicitno obradimo – ako prepoznamo da sadržaj toka ne odgovara podacima koje čitamo, najčešće je dovoljno da eksplicitno označimo tok kao neispravan (videti *10.10 Biblioteka tokova*, na strani 383):

```
class Razlomak
{
...
    istream& Citaj( istream& str )
        {
        char c;
        str >> _Brojilac >> c >> _Imenilac;
        if( c != '/' )
            str.setstate( ios::failbit );
        else
            PopraviPodatke();
        return str;
        }
...
};
```

Jedini problem koji nam je preostao je onemogućavanje pravljenja razlomaka sa imeniocem 0. Za rešavanje ovog problema u duhu programskog jezika C++ neophodno je poznavanje rada sa izuzecima. Rad sa izuzecima će biti obrađen tek u primeru *3 - Dinamički nizovi*, na strani 90. Ovaj problem ćemo, za sada, svesno ignorisati.

Korak 10 - Enkapsulacija (nastavak)

Naša klasa `Razlomak` i dalje ne omogućava neposrednu promenu vrednosti imenioca i brojioca. Jedini način da se izmeni vrednost razlomka je da se objektu dodeli nova vrednost primenom operatora dodeljivanja. Na primer:

```
a = Razlomak(3,4);
```

Ovakva promena vrednosti nije efikasna jer obuhvata pravljenje novog objekta, kopiranje sadržaja objekata i uklanjanje nepotrebnog objekta. To znači da nam je potrebno bolje rešenje.

Razmotrimo najpre mogućnost da članove podatke `_Imenilac` i `_Brojilac` proglasimo za javne. Ako nam je već potrebno da ih menjamo, a ne samo da ih čitamo, zašto da ih ne stavimo na raspolaganje svim korisnicima klase? U polaznoj klasi `Razlomak` to ne bi predstavljalo nikakav poseban problem i mogli bismo to učiniti bez negativnih posledica. Međutim, većina iole složenijih klasa, pa i poslednje verzije naše klase `Razlomak`, uvode neke semantičke pretpostavke o vrednostima podataka koji ih čine. Te semantičke pretpostavke mogu se odnositi na dopušten opseg pojedinačnih podataka ili na neke međusobne odnose u kojima članovi podaci moraju biti. Narušavanje tih pretpostavki dovelo bi do neispravnih objekata i njihovog neispravnog ponašanja.

U slučaju razlomka imamo tri semantičke pretpostavke čije se ispunjenje ostvaruje primenom metoda `PopraviPodatke`, a čije bi narušavanje dovelo do neispravnog ponašanja operatora poređenja:

- imenilac mora biti pozitivan;
- brojilac i imenilac moraju biti uzajamno prosti;
- ako je brojilac 0, imenilac mora biti 1 (ako ovo ne bi važilo, operacije ++ i -- bi dovodile razlomak u neispravno stanje, jer brojilac i imenilac ne bi bili uzajamno

prosti – ipak, ovom pravilu nije posebno posvećena pažnja, jer se njegovo poštovanje obezbeđuje implicitno kroz postupak skraćivanja, jer važi `nzd(0,n)=n`).

Zbog toga što su podaci klase često obuhvaćeni nekim takvim pretpostavkama, ili bar postoji značajna mogućnost da se njima obuhvate tokom životnog veka klase (tj. zbog naknadnih izmena klase), *obično se toplo preporučuje da svi članovi podaci budu privatni, ili bar zaštićeni*. U skladu sa time, preporučuje se da se pristupanje podacima ostvaruje isključivo posredstvom tzv. *pristupnih metoda* (metoda za čitanje i menjanje):

- Za čitanje podatka se piše konstantan metod koji izračunava njegovu vrednost. U slučaju prostog tipa obično se vraća kopija podatka, dok se u slučaju klasnog tipa obično vraća referenca na konstantan podatak. Ovaj metod najčešće ima oblik:

```
const <tip>& VratiX() const
    { return X; }
```

- Za menjanje podatka se piše nekonstantan metod koji za argument ima novu vrednost podatka ili parametre koji omogućavaju promenu vrednosti. Osim što menja vrednost podatka ovaj metod se stara da ta promena vrednosti bude usklađena sa svim semantičkim pretpostavkama koje postoje u klasi, a koje se na tu vrednost odnose. Ovaj metod najčešće ne vraća nikakav rezultat. Njegov oblik je obično:

```
void PostaviX( ... )
    {
    ...
    X = ...;
    ...
    }
```

Imenovanje ovih metoda se razlikuje od autora do autora, ali se obično svodi na varijaciju jednog od narednih pravila:

- za čitanje se koristi ime `VratiX`, a za pisanje `PostaviX`;
- za podatak se koristi izmenjeno ime, recimo `_X`, za čitanje ime `X`, a za pisanje `PostaviX`;
- za podatak se koristi izmenjeno ime, recimo `_X`, a za čitanje i pisanje isto ime `X` (primetimo da je ovo sasvim u redu jer programski jezik C++ dopušta definisanje više metoda sa istim imenom sve dok se razlikuju po broju i/ili tipu argumenata);
- neki drugi dosledno primenjen način imenovanja.

Klasa `Razlomak` već sadrži metode za čitanje podataka. Ako bismo dopisali metode za menjanje podataka, to bi moglo da izgleda ovako:

```
class Razlomak
{
...
```

```
        //-------------------------------------------------------------
        // Metodi za pristupanje elementima razlomka.
        //-------------------------------------------------------------
        int Imenilac() const
            { return _Imenilac; }
        int Brojilac() const
            { return _Brojilac; }
        void PostaviImenilac( int n )
            {
            _Imenilac = n;
            PopraviPodatke();
            }
        void PostaviBrojilac( int n )
            {
            _Brojilac = n;
            PopraviPodatke();
            }
    ...
    };
```

Preostaje nam da vidimo kako se novi metodi ponašaju. Dopišimo sledeći segment koda u funkciju `main` i proverimo da li sve radi kako bismo očekivali:

```
Razlomak a(14,15);
a.PostaviBrojilac(5);
a.PostaviImenilac(7);
cout << a << endl;
```

Koja je vrednost razlomka `a` nakon izvršavanja navedenog segmenta koda? Ako ne bismo bili upoznati sa načinom funkcionisanja klase `Razlomak`, mogli bismo pomisliti da bi nova vrednost bila 5/7, ali to je daleko od istine. Jednostavno izvršavanje programa pokazuje da je rezultat 1/7!? Zašto? Ako pratimo vrednost razlomka tokom izvršavanja ovog segmenta koda možemo primetiti:

- nakon konstrukcije vrednost razlomka `a` je 14/15;
- nakon promene brojioca vrednost je 5/15, ali odmah dolazi do skraćivanja i dobija se 1/3;
- nakon promene imenioca dobija se 1/7, što je i konačna vrednost.

Znači da do problema dolazi zbog skraćivanja razlomka. Prave razmere problema sagledaćemo tek ako promene pokušamo da izvedemo u obrnutom redosledu:

```
Razlomak a(14,15);
a.PostaviImenilac(7);
a.PostaviBrojilac(5);
cout << a << endl;
```

Sada vrednost razlomka `a` nije ni 5/7, ni 1/7 nego 5!

Konzistentnost

Za klasu koja dopušta da se javnim sredstvima (promenom javnih podataka ili izvođenjem javnih metoda) ispravan objekat prevede u neispravno stanje kažemo da je *nekonzistentna*.

Napisali smo nove pristupne metode kako ne bismo proglašavanjem podataka za javne načinili klasu nekonzistentnom, a sada ispada da je i novi metodi čine takvom. Očigledno, novi metodi nisu konzistentni. Da bi metod bio konzistentan, pored zahteva da *izvođenje metoda ne sme narušiti semantičke pretpostavke o odnosima u objektu i među objektima*, neophodno je ispoštovati da *stanje objekta nakon izvođenja metoda mora biti jasno uslovljeno prethodnim stanjem i semantikom metoda*. Ukoliko ta uslovljenost nije lako uočljiva korisniku klase, moramo računati s tim da će pri upotrebi dolaziti do problema. U metode `PostaviImenilac` i `PostaviBrojilac` smo ugradili pozivanje metoda `PopraviPodatke`, kako bi po njihovim izvršavanju bile zadovoljene semantičke pretpostavke o odnosu imenioca i brojioca. Međutim, iako smo time obezbedili poštovanje semantičkih pretpostavki, stanje razlomka nakon primene ovih metoda je u suviše velikoj zavisnosti od prethodnog stanja, tako da korisnik klase ne može jednostavno sagledati kakvo će biti konačno stanje objekta.

Problem nekonzistentnih metoda se obično razrešava zamenjivanjem problematičnih metoda nekim drugim metodima čije je ponašanje primerenije postojećem skupu semantičkih pretpostavki. Ako pri izvođenju nekih manjih aktivnosti dolazi do problema, oni se obično mogu izbeći spajanjem tih manjih aktivnosti u veće. U našem slučaju, jasno je da probleme pravi pojedinačno menjanje imenioca i brojioca, pa je zato potrebno da se ove operacije ne definišu pojedinačno, već objedinjene u jednom metodu:

```
class Razlomak
{
...
    //-------------------------------------------------------------
    // Metodi za pristupanje elementima razlomka.
    //-------------------------------------------------------------
    int Imenilac() const
        { return _Imenilac; }
    int Brojilac() const
        { return _Brojilac; }
    void PostaviRazlomak( int i, int b )
        {
        _Imenilac = i;
        _Brojilac = b;
        PopraviPodatke();
        }
...
};
```

Problem konzistentnosti klase bi trebalo razmatrati svaki put pri uvođenju novih semantičkih pretpostavki. U praksi bi problem konzistentnosti i problem robusnosti uvek trebalo rešavati paralelno sa razvojem programa ili klase. To se najčešće rešava primenom enkapsulacije (čime se sprečava neposredno menjanje podataka koje može narušiti semantičke pretpostavke) i pažljivim izborom skupa metoda.

Prostori imena

Prostori imena se definišu radi prevazilaženja problema „zagađivanja“ globalnog prostora imena. Zagađivanje globalnog prostora imena je pojava do koje dolazi pri upotrebi velikog broja različitih biblioteka i pisanju velikih programa. Naziv je dobila zbog činjenice da se na svakom konkretnom mestu u programu obično upotrebljava samo mali broj imena neke biblioteke, a da se u skup globalnih imena uvode sva imena biblioteke, od kojih većina predstavlja suvišno opterećenje.

Zagađivanje globalnog prostora imena ima više neugodnih posledica. Ispostavlja se da je u takvim situacijama prilično velika verovatnoća da se neka imena upotrebljavaju u više biblioteka, što značajno otežava njihovu upotrebu. Prevazilaženje takvih problema zahteva prilično naporne zahvate. Drugi problem je da se u prisustvu velikog broja imena povećava i verovatnoća da se greškom umesto jednog imena navede neko drugo ime koje postoji u globalnom prostoru imena, tako da provera sintakse neće prijaviti nepoznato ime.

Da njihove biblioteke ne bi bile neupotrebljive usled opisanih problema, autori biblioteka na programskom jeziku C su bili prinuđeni da upotrebljavaju dugačka imena, koja bi obično započinjala nekim identifikatorom biblioteke (koji je takođe morao biti dugačak). To je otežavalo pisanje i čitanje programa.

Zbog toga je u programski jezik C++ uveden koncept prostora imena. Prostori imena omogućavaju da se imena organizuju u manje celine. Osnovna ideja je da se imena koja čine neku biblioteku ili neku veću celinu programa definišu u posebnom prostoru imena. Kako se ime prostora imena obično ne navodi često, izbor dugačkih imena prostora imena ne predstavlja značajan problem, pa se preporučuje.

Prostor imena čine sva imena definisana u nekom bloku tog prostora imena. Jedan prostor imena se može definisati u više blokova pa i modula. Svaki od blokova prostora imena ima oblik:

```
namespace <ime prostora imena> {
    ...
}
```

Imena definisana u jednom prostoru imena se mogu upotrebljavati u tom istom prostoru imena na uobičajen način. Da bi se ime definisano u jednom prostoru imena moglo upotrebljavati na nekom mestu u programu, a van tog prostora imena (tj. u nekom drugom prostoru imena ili u globalnom prostoru imena), potrebno je učiniti jednu od tri stvari:

1. Pri upotrebi imena svaki put eksplicitno navoditi i ime odgovarajućeg prostora imena:

   ```
   ... <ime prostora imena>::<ime> ...
   ```

2. Deklarisati da je to ime vidljivo u tekućem prostoru imena:

   ```
   using <ime prostora imena>::<ime>;
   ```

3. Deklarisati da su sva imena iz odgovarajućeg prostora imena vidljiva u tekućem prostoru imena:

```
using namespace <ime prostora imena>;
```

Zbog toga što su sva imena standardne biblioteke definisana u prostoru imena `std`, na početku svakog modula u kome upotrebljavamo elemente standardne bibilioteke ćemo deklarisati da su vidljiva sva imena standardne bibiloteke:

```
using namespace std;
```

Zbog toga što je veličina svakog od primera relativno mala, u daljem tekstu ćemo sva imena definisati u globalnom prostoru imena. Više informacijama o prostorima imena je na raspolaganju u preporučenoj literaturi za programski jezik C++.

1.3 Rešenje

```
#include <iostream>

using namespace std;

//---------------------------------------------------------------
//  Funkcija nzd računa najveći zajednički delilac
//---------------------------------------------------------------
int nzd( int a, int b )
{
    int t;
    //  postaramo se da važi a>=b
    if( a < b ){
        t = b;
        b = a;
        a = t;
        }
    //  stalno održavajući a>=b približavamo se rešenju
    while( b > 0 ){
        t = b;
        b = a % b;
        a = t;
        }
    //  ako je b postalo 0, znači da smo delili sa a bez ostatka
    return a;
}

//---------------------------------------------------------------
//  Klasa Razlomak
//---------------------------------------------------------------
class Razlomak
{
public:
    //-----------------------------------------------------------
    // Konstruktor. Destruktor nije potreban.
    //-----------------------------------------------------------
    explicit Razlomak( int b = 0, int i = 1 )
        : _Brojilac(b), _Imenilac(i)
        { PopraviPodatke(); }
```

```
//-------------------------------------------------------------
// Metodi za pristupanje elementima razlomka.
//-------------------------------------------------------------
int Imenilac() const
    { return _Imenilac; }
int Brojilac() const
    { return _Brojilac; }
void PostaviRazlomak( int i, int b )
    {
    _Imenilac = i;
    _Brojilac = b;
    PopraviPodatke();
    }

//-------------------------------------------------------------
// Pisanje i čitanje.
//-------------------------------------------------------------
ostream& Pisi( ostream& str ) const
    { return str << _Brojilac << '/' << _Imenilac; }

istream& Citaj( istream& str )
    {
    char c;
    str >> _Brojilac >> c >> _Imenilac;
    if( c != '/' )
        str.setstate( ios::failbit );
    else
        PopraviPodatke();
    return str;
    }

//-------------------------------------------------------------
// Binarne aritmetičke operacije.
//-------------------------------------------------------------
Razlomak operator + ( const Razlomak& r )
    {
    return Razlomak (
        Brojilac() * r.Imenilac() + Imenilac() * r.Brojilac(),
        Imenilac() * r.Imenilac()
        );
    }

Razlomak operator - ( const Razlomak& r ) const
    {
    return Razlomak(
        Brojilac() * r.Imenilac() - Imenilac() * r.Brojilac(),
        Imenilac() * r.Imenilac()
        );
    }

Razlomak operator * ( const Razlomak& r ) const
    {
    return Razlomak(
        Brojilac() * r.Brojilac(),
        Imenilac() * r.Imenilac()
        );
    }
```

```
Razlomak operator / ( const Razlomak& r ) const
    {
    return Razlomak(
        Brojilac() * r.Imenilac(),
        Imenilac() * r.Brojilac()
        );
    }

//-------------------------------------------------------------
// Unarne aritmetičke operacije.
//-------------------------------------------------------------
Razlomak operator - () const
    { return Razlomak( -Brojilac(), Imenilac() ); }

Razlomak operator ~ () const
    { return Razlomak( Imenilac(), Brojilac() ); }

//-------------------------------------------------------------
// Operacije poređenja.
//-------------------------------------------------------------
bool operator == ( const Razlomak& r ) const
    {
    return Brojilac() == r.Brojilac()
        && Imenilac() == r.Imenilac();
    }

bool operator != ( const Razlomak& r ) const
    { return !( *this == r ); }

bool operator > ( const Razlomak& r ) const
    {
    return Brojilac() * r.Imenilac()
         > Imenilac() * r.Brojilac();
    }

bool operator >= ( const Razlomak& r ) const
    {
    return Brojilac() * r.Imenilac()
        >= Imenilac() * r.Brojilac();
    }

bool operator < ( const Razlomak& r ) const
    {
    return Brojilac() * r.Imenilac()
         < Imenilac() * r.Brojilac();
    }

bool operator <= ( const Razlomak& r ) const
    {
    return Brojilac() * r.Imenilac()
        <= Imenilac() * r.Brojilac();
    }

//-------------------------------------------------------------
// Operatori inkrementiranja i dekrementiranja
//-------------------------------------------------------------
// prefiksno ++
Razlomak& operator ++ ()
    {
    _Brojilac += _Imenilac;
```

```
            return *this;
            }

        // postfiksno ++
        const Razlomak operator ++ (int)
            {
            Razlomak r = *this;
            _Brojilac += _Imenilac;
            return r;
            }

        // prefiksno --
        Razlomak& operator -- ()
            {
            _Brojilac -= _Imenilac;
            return *this;
            }

        // postfiksno --
        Razlomak operator -- (int)
            {
            Razlomak r = *this;
            _Brojilac -= _Imenilac;
            return r;
            }

        //------------------------------------------------------------
        //  Konverzija u tip double
        //------------------------------------------------------------
        operator double () const
            { return _Brojilac / (double)_Imenilac; }

    private:
        //------------------------------------------------------------
        // Korigovanje podataka
        //------------------------------------------------------------
        void PopraviPodatke()
            {
            // sprečavanje negativnog imenioca
            if( _Imenilac < 0 ){
                _Brojilac = - _Brojilac;
                _Imenilac = - _Imenilac;
                }
            // skraćivanje imenioca i brojioca
            int n = nzd( abs(_Brojilac), _Imenilac );
            if( n>1 ){
                _Imenilac /= n;
                _Brojilac /= n;
                }
            }

        //------------------------------------------------------------
        // Članovi podaci
        //------------------------------------------------------------
        int _Imenilac;
        int _Brojilac;
    };
```

```
//-------------------------------------------------------------
//  Operatori za pisanje i čitanje razlomaka.
//-------------------------------------------------------------
ostream& operator<< ( ostream& str, const Razlomak& r )
    { return r.Pisi( str ); }
istream& operator>> ( istream& str, Razlomak& r )
    { return r.Citaj( str ); }

//-------------------------------------------------------------
// Glavna funkcija programa demonstrira upotrebu klase Razlomak.
//-------------------------------------------------------------
main()
{
    Razlomak a(1,2), b(2,3);
    cout << a << " + " << b << " = " << (a+b) << endl;
    cout << a << " - " << b << " = " << (a-b) << endl;
    cout << a << " * " << b << " = " << (a*b) << endl;
    cout << a << " / " << b << " = " << (a/b) << endl;
    cout << "- " << b << " = " << (-b) << endl;
    cout << "~ " << b << " = " << (~b) << endl;
    cout << a << " == " << a << " = " << (a==a) << endl;
    cout << a << " == " << b << " = " << (a==b) << endl;
    cout << a << " < " << b << " = " << (a<b) << endl;
    cout << a << " > " << b << " = " << (a>b) << endl;
    cout << (a++) << endl;
    cout << (++a) << endl;
    cout << double(a) << endl;
    cout << (double)a << endl;
    cout << static_cast<double>(a) << endl;
    cout << ( Razlomak(1,5) + 2 ) << endl;

    return 0;
}
```

1.4 Rezime

Upotrebljavajte klasu i pokušajte da uočite koji bi još metodi mogli biti od koristi i pokušajte da ih implementirate. Između ostalog:

- nakon upoznavanja rada sa izuzecima, dopunite klasu `Razlomak` onemogućavanjem pravljenja razlomaka koji imaju vrednost imenioca 0;
- napišite operatore `+=`, `-=`, `*=` i `/=`.

2 - Lista

2.1 Zadatak

Napisati klasu `Lista`, koja predstavlja listu celih brojeva, i program koji demonstrira njenu upotrebu. U klasi `Lista` obezbediti:

- metode za dodavanje elemenata na početak i kraj liste;
- metode za pristupanje elementima liste;
- ostale metode neophodne za ispravno funkcionisanje liste.

Cilj zadatka

Kroz ovaj primer ćemo upoznati:

- neke specifične probleme koji se pojavljuju pri radu sa dinamičkim strukturama podataka, uključujući pisanje metoda:
  - destruktor;
  - konstruktor kopije;
  - operator dodeljivanja;
- neka nova sredstva za ostvarivanje enkapsulacije:
  - prijateljske klase;
  - umetnute klase.

Pretpostavljena znanja

Za uspešno praćenje rešavanja ovog zadatka pretpostavlja se poznavanje:

- osnovnih principa upotrebe pokazivača i referenci;
- pisanja klasa;

- izraza `new` za dinamičko pravljenje objekata i izraza `delete` za dinamičko uklanjanje objekata;
- koncepta skrivanja podataka.

2.2 Rešavanje zadatka

Nizom koraka ćemo napraviti i unapređivati klasu `Lista`:

Korak 1 - Definisanje interfejsa klase Lista

Lista je dinamička strukutra podataka koja omogućava da se više elemenata veže u sekvencu čija dužina nije unapred poznata. Maksimalan broj elemenata liste je ograničen samo raspoloživom memorijom, a zauzeće memorije je proporcionalno broju elemenata koje lista sadrži (tj. zauzeće nije proporcionalno maksimalnoj ili planiranoj nego stvarnoj dužini liste). Ako problem posmatramo iz ugla strukture, tada listu možemo definisati rekurzivno:

- Svaka lista je:
  - ili prazna lista;
  - ili se sastoji od jedne vrednosti elementa (tj. prvog elementa liste) i liste koja predstavlja njen nastavak.

Međutim, kao što smo ranije već naglasili, klase se ne definišu na osnovu strukture, već na osnovu ponašanja. Na osnovu zahtevanog ponašanja klase najpre se definiše *interfejs klase*. Interfejs klase predstavlja *sredstvo za upotrebu objekata klase*. Čine ga deklaracije javnih metoda i podataka klase. Zbog težnje da se funkcionalnost i način upotrebe klase što više razdvoje od

implementacije, a podaci predstavljaju upravo strukturni deo implementacije, preporučuje se da se podaci ne uključuju u interfejs klasa, tj. da ne budu javni.

Oblikovanje interfejsa počiva na analizi zahteva:

- za svaku zahtevanu operaciju se dodaje po odgovarajući metod (eventualno jedan metod može odgovarati većem broju srodnih operacija);
- nazivi metoda se određuju odgovaranjem na pitanje „*Šta metod radi?*“;
- argumenti i/ili rezultat metoda se određuju na osnovu semantike metoda (eventualno se umesto nekih parametara mogu upotrebiti podaci objekta ili rezultati izračunavanja metoda koji opisuju stanje objekta);
- da bi klasa mogla ispravno da funkcioniše može biti neophodno da se interfejs proširi još nekim operacijama koje nisu eksplicitno zahtevane.

Kao što je u postavci zadatka naglašeno, naša `Lista` mora obezbediti metode za dodavanje elemenata na početak i kraj, kao i za pristupanje elementima liste. Na osnovu toga možemo napisati minimalan skup metoda, kao i njihovu trivijalnu implementaciju i program koji sve to testira:

```
//---------------------------------------------------------------
//  Klasa Lista
//---------------------------------------------------------------
class Lista
{
public:
    void DodajNaPocetak( int n )
        { cout << "Dodavanje elementa na početak: " << n << endl;}

    void DodajNaKraj( int n )
        { cout << "Dodavanje elementa na kraj: " << n << endl; }

    int Element( int i ) const
        {
        cout << "Citanje elementa: " << i << endl;
        return 0;
        }
};

//---------------------------------------------------------------
// Glavna funkcija programa demonstrira upotrebu klase Lista.
//---------------------------------------------------------------
main()
{
    Lista l;
    for( int i=0; i<10; i++ )
        l.DodajNaPocetak(i);
    for( int i=0; i<10; i++ )
        cout << l.Element(i) << ' ';
    cout << endl;

    return 0;
}
```

Klasa `Lista` za sada ne radi ništa korisno, ali smo ustanovljavanjem interfejsa klase na samom početku njenog razvoja postigli da dalje unapređivanje klase, pa i eventualno proširivanje interfejsa, neće zahtevati izmene na mestima gde se naša klasa koristi, sve do trenutka kada deo interfejsa ne izmenimo ili čak ne odstranimo. Primetimo da pristupanje elementima liste na osnovu njihovih rednih brojeva nije uobičajeno, ali omogućava jednostavniji interfejs, što je za nas u ovom trenutku najvažnije.

Nakon što smo definisali interfejs klase, možemo preći na njenu *implementaciju*. Pod implementacijom klase podrazumevamo definisanje tela metoda koji čine interfejs, kao i definisanje neophodnih privatnih metoda i članova. Implementacija može dovesti do manjih ili većih izmena u interfejsu, kako bi on postao kompaktniji ili bolje zaokružen. Ako se na samom početku posveti dovoljno pažnje oblikovanju interfejsa, takve izmene su relativno retke. Pravi smisao i ulogu interfejsa u objektno orijentisanom programiranju videćemo tek na složenijim primerima sa hijerarhijama klasa, gde se čitave klase definišu samo da bi služile kao interfejsi, dok se implementacija obezbeđuje kroz klase naslednice. U ovom trenutku je važno da se prepozna da interfejs definiše *šta objekti klase mogu da rade* a da implementacija definiše *kako to rade*.

Pre nego što pređemo na implementaciju klase `Lista`, primetimo da jednom interfejsu može odgovarati više različitih implementacija. Kao što se telo jednog metoda može napisati na više različitih načina, a da pri tome njegova deklaracija ne bude menjana, tako se i jedna klasa sa definisanim interfejsom može implementirati na više načina.

Korak 2 - Struktura liste

Nakon što je definisano *šta* je to što klasa `Lista` radi, potrebno je da opišemo i *kako* to radi. Već je napomenuto da je stuktura liste opisana rekurzivno. U nekim programskim jezicima je moguće tu strukturu opisati nalik na:

```
class ElementListe
{
public:
    int             Vrednost;
    ElementListe    Sledeci;
};
```

U programskom jeziku C++ to nije moguće. Razlog je u suprotstavljanju dva principa na kojima počiva C++ (a i mnogi drugi programski jezici):

- podaci koji čine objekat se fizički nalaze u objektu;
- svi objekti istog fizičkog tipa imaju istu veličinu.

Po prvom principu, jedan element liste bi se sastojao od vrednosti i narednog elementa liste, pa je jasno je da bi svaki element liste morao da bude veći od narednog bar za veličinu celobrojnog podatka. Po drugom principu to nije moguće. Zbog toga se za ostvarivanje rekurzivnih struktura podataka moraju upotrebljavati pokazivači:

```
//---------------------------------------------------------------
//  Klasa ElementListe
//---------------------------------------------------------------
class ElementListe
{
public:
    int             Vrednost;
    ElementListe*   Sledeci;
};
```

Pri tome se prazan pokazivač tumači kao pokazivač na praznu listu elemenata. Više objekata klase `ElementListe` može se povezati u *listu elemenata*. Veoma je važno uočiti razliku između takve liste elemenata i klase `Lista`, jer dok je lista elemenata fizička struktura podataka, klasa `Lista` je logička struktura koja tu fizičku strukturu apstrahuje i omogućava njenu upotrebu uz minimalne pretpostavke o strukturi. Čak i kada se klasa ne može implementirati drugačije nego pomoću neke konkretne fizičke strukture podataka, njena suština ne bi trebalo da se oslikava toliko kroz samu strukturu koliko kroz njenu upotrebljivost, tj. interfejs.

Nakon što definišemo `ElementListe`, možemo klasu `Lista` implementirati tako da sadrži pokazivač na listu elemenata:

```
class Lista
{
private:
    ElementListe*   _Pocetak;
...
};
```

Ranije uvedene metode sada možemo napisati u skladu sa ovakvom strukturom liste. Poći ćemo od metoda `DodajNaPocetak`. Prvi posao koji pri tome moramo obaviti je pravljenje novog elementa liste. Najpre ga napravimo primenom izraza `new`, a zatim inicijalizujemo njegov sadržaj. Vrednost novog elementa je određena vrednošću argumenta `n`. Kako novi element dodajemo na početak liste, njemu sledeći element će biti upravo onaj koji je do sada bio na početku liste. Isto pravilo se primenjuje i ako je lista do sada bila prazna – vrednost pokazivača `_Pocetak` je u tom slučaju 0, što će biti i ispravna vrednost pokazivača na sledeći element. Konačno, potrebno je novi element proglasiti za početni:

```
void DodajNaPocetak( int n )
    {
    // Napravimo novi element...
    ElementListe* novi = new ElementListe;
    // ...i inicijalizujemo njegov sadržaj.
    novi->Vrednost = n;
    novi->Sledeci = _Pocetak;
    // Postavimo novi element za početni.
    _Pocetak = novi;
    }
```

Dodavanje elementa na kraj liste je malo složenije. U svakom slučaju, najpre pravimo novi element. Novi element će biti na kraju liste, što znači da iza njega nema drugih elemenata, pa

se pokazivač `Sledeci` inicijalizuje vrednošću 0. Ukoliko je lista do sada bila prazna, novi element će biti ne samo poslednji nego i početni, pa zato menjamo vrednost pokazivača `_Pocetni`. Ako lista nije bila prazna, potrebno je pronaći poslednji element i njegov pokazivač na sledeći element (koji je do sada bio 0) izmeniti tako da pokazuje na novi element:

```
void DodajNaKraj( int n )
    {
    // Napravimo novi element...
    ElementListe* novi = new ElementListe;
    novi->Vrednost = n;
    novi->Sledeci = 0;
    // ...i inicijalizujemo njegov sadržaj.
    // Ako je lista do sada bila prazna...
    if( !_Pocetak )
        // ...novi element će biti istovremeno i početni.
        _Pocetak = novi;
    // Inače,...
    else{
        // ...najpre potražimo poslednji element liste...
        ElementListe* p;
        for( p = _Pocetak; p->Sledeci; p=p->Sledeci )
            ;
        // ...pa označimo da za njim sledi novi element.
        p->Sledeci = novi;
        }
    }
```

Preostaje nam da implementiramo i metod `Element`. Radi jednostavnosti, za sada ćemo pretpostaviti da je argument `i` ispravan, tj. da lista sadrži bar `i+1` elemenata, tako da se može izdvojiti i traženi `i`-ti element. Poći ćemo od početka liste, preskočiti `i` elemenata i vratiti vrednost `i+1`-og elementa. Primetimo da prvi element označavamo indeksom 0. To je uobičajeni način brojanja elemenata u programskom jeziku C++ i nipošto ne bismo smeli brojati drugačije:

```
int Element( int i ) const
    {
    // Preskočimo i elemenata.
    ElementListe* p = _Pocetak;
    for( int j=0; j<i; j++ )
        p=p->Sledeci;
    // Vratimo vrednost i+1-og elementa.
    return p->Vrednost;
    }
```

Podrazumevani konstruktor

Ako bismo sada pokušali da prevedemo i izvršimo naš probni program, mogli bismo se neprijatno iznenaditi. Naime, program bi se uspešno preveo, bez ijedne greške ili upozorenja, ali njegovo izvršavanje bi u nekim slučajevima (zavisno od operativnog sistema i trenutnog stanja memorije) imalo fatalne efekte. Zašto? Zato što ni na jednom mestu nismo naglasili da je nova lista na početku prazna. Prilikom pravljenja novog objekta klase `Lista` podatak `_Pocetak` ostaje neinicijalizovan. Ako bi njegova vrednost bila 0, sve bi prošlo u savršenom

redu, ali ako bi imao bilo koju drugu vrednost smatralo bi se da pokazuje na prvi element liste, što nije tačno. Da se ovakve neprijatnosti ne bi dešavale, *svaki put po dodavanju ili menjanju članova podataka klase neophodno je razmotriti da li ih je i kako potrebno inicijalizovati i zatim prilagoditi konstruktore tako da inicijalizacija bude potpuna i ispravna.*

Kako je uopšte moguće praviti objekte neke klase za koju nije napisan nijedan konstruktor? U programskom jeziku C++ u (skoro) svakoj klasi se podrazumeva postojanje tzv. *podrazumevanog konstruktora*. Podrazumevani konstruktor predstavlja implicitno definisan konstruktor bez argumenata, koji se primenjuje pri pravljenju novih objekata bez navođenja ikakvih argumenata. *Podrazumevani konstruktor obezbeđuje da se svi podaci objekta, uključujući i nasleđene delove i specifične podatke, inicijalizuju primenom konstruktora bez argumenata za odgovarajuće tipove.* Zbog toga podrazumevani konstruktor ne može da postoji u klasama koje imaju neku baznu klasu ili neki podatak za čiji tip ne postoji konstruktor bez argumenata. Štaviše, ukoliko je definicijom klase eksplicitno obezbeđen bar jedan konstruktor, tada neće postojati podrazumevani konstruktor, pa se i konstruktor bez argumenata (ukoliko je potreban) mora eksplicitno definisati.

Kako u klasama `ElementListe` i `Lista` nismo definisali nikakve konstruktore, u obe klase postoje podrazumevani konstruktori, pa je moguće praviti objekte ovih klasa:

```
Lista l;
Lista* lp = new Lista;
ElementListe el;
ElementListe* elp = new ElementListe;
```

Kako podrazumevani konstruktor klase `Lista` ne zadovoljava naše potrebe, jer podrazumevani konstruktori prostih tipova, pa i pokazivača, ne izvode nikakvu inicijalizaciju, potrebno je napisati odgovarajući konstruktor bez argumenata:

```
Lista()
    : _Pocetak(0)
    {}
```

Pregledanjem napisanog koda možemo ustanoviti da se vrednosti podataka objekata klase `ElementListe` uvek inicijalizuju nakon pravljenja. Zbog toga je dobro u toj klasi obezbediti konstruktor i koristiti ga pri pravljenju novih elemenata liste. Primetimo da nakon pisanja ovog konstruktora više neće biti na raspolaganju podrazumevani konstruktor bez argumenata.

```
class ElementListe
{
...
    ElementListe( int v, ElementListe* s )
        : Vrednost(v), Sledeci(s)
        {}
...
};

class Lista
{
public:
```

```
...
    void DodajNaPocetak( int n )
        { _Pocetak = new ElementListe( n, _Pocetak ); }
    void DodajNaKraj( int n )
        {
        ElementListe* novi = new ElementListe( n, 0 );
        ...
        }
...
};
```

Sada program možemo izvršavati a da ne primetimo nikakve probleme. No, to nikako ne znači da sve radi kako bi trebalo i da problemi ne postoje.

Korak 3 - Oslobađanje nepotrebnih resursa

Pokušajmo da na početak funkcije `main` dodamo sledeći segment koda:

```
main()
{
    for( int j=0; j<1000000; j++ ){
        Lista l;
        for( int i=0; i<1000000; i++ )
            l.DodajNaPocetak(i);
        }
    ...
}
```

Šta će se dogoditi kada pokušamo da izvršimo ovakav program?

Prvi tačan zaključak koji možemo izvesti o novom segmentu koda je da on ne radi ništa korisno. Milion puta pravi listu sa po milion elemenata. Dakle, program bi određeno vreme trošio procesorsko vreme na beskorisno računanje, da bi zatim nastavio sa radom kao i ranije. Da li postoje određeni mogući problemi sa resursima? Pa, recimo da je očigledno da program može praviti probleme sa memorijom ako nema na raspolaganju dovoljno memorije da napravi listu sa milion elemenata. Da li je to sve?

Ne. Naš program bi probleme sa memorijom pravio čak i kada na raspolaganju ima dovoljno memorije za listu od milion elemenata. Problem je u tome što je za njegovo izvršavanje potrebno dovoljno memorije da može napraviti *milion lista sa po milion elemenata*!

Kako sad to? Jedna od osnovnih osobina programskog jezika C++ je da se memorija zauzeta za zapisivanje promenljivih (tj. objekata) definisanih u okviru jednog bloka oslobađa po izlasku iz tog bloka. Znači, svaki put pre nego što započnemo pravljenje nove liste ulazeći u novi ciklus, prvo izlazimo iz bloka u kome je definisana promenljiva `l`, čime se oslobađa memorija koju zauzima odgovarajući objekat klase `Lista`. Ispada da u memoriji uvek postoji najviše jedan objekat klase `Lista`, pa je prethodna primedba netačna?

Nije. Zaista, uvek će postojati najviše jedan objekat klase `Lista`, ali nije u tome problem – iako će uvek postojati samo po jedan objekat klase Lista, *u memoriji ćemo ipak imati sve više i više elemenata liste uvezanih u sekvence od po milion elemenata*, da bismo na kraju imali čitav milion sekvenci (ako operativni sistem pre toga ne prekine rad našeg programa zbog preteranog zauzeća memorije, jer će nam na 32-bitnom računaru biti potrebno više od 8000GB, a na

64-bitnom čak više od 16000GB)! Da bismo mogli razumeti pravu prirodu problema moramo se upoznati sa životom objekata u programskom jeziku C++.

Život objekata u programskom jeziku C++

Svaki objekat u programskom jeziku C++ prolazi, redom, kroz sledeće faze života:

1. alokacija memorije;
2. inicijalizacija;
3. upotreba;
4. deinicijalizacija;
5. oslobađanje memorije.

Kada govorimo o pravljenju i uništavanju objekata, pri tome uvek mislimo na prve dve, odnosno poslednje dve faze života objekta. Ranije je već pomenuto da slično konceptu metoda konstruktora, koji obezbeđuju inicijalizaciju objekata pri njihovom pravljenju, postoji i koncept metoda *destruktora*, koji obezbeđuju deinicijalizaciju objekata pri njihovom uklanjanju. Zapravo, svaki put kada se pravi novi objekat, izvršavaju se, redom, dve važne operacije – alokacija potrebne memorije i konstrukcija (inicijalizacija) objekta. Slično tome, svaki put kada se objekat uništava izvršavaju se njima inverzne operacije u obrnutom redosledu – najpre destrukcija (deinicijalizacija) objekta i zatim oslobađanje memorije. Na taj način se onemogućava da se u upotrebi pojave neinicijalizovani objekti.

U ovoj knjizi upotrebljavamo termin *izrazi* `new` i `delete`. Veoma često se u literaturi upotrebljava termin *operatori* `new` i `delete`, pri čemu se misli na izraze a ne na operatore. Problem je u tome što se u programskom jeziku C++ *razlikuju* izrazi i operatori `new` (odnosno `delete`), pri čemu je ta razlika izuzetno značajna. Naime, svaka klasa ima (podrazumevane ili eksplicitno napisane) operatore `new` i `delete` koji *alociraju memoriju* za nove objekte i *oslobađaju memoriju* pri uklanjanju objekata. Za razliku od operatora, izrazi `new` i `delete` obavljaju značajno veći posao: izraz `new` alocira memoriju i inicijalizuje novi objekat, a izraz `delete` deinicijalizacije objekat i oslobađa memoriju. Zapravo, izraz `new` najpre izvršava operator `new`, pa zatim konstruktor, a izraz `delete` najpre izvršava destruktor, pa zatim operator `delete`.

Operatori `new` i `delete` se izuzetno retko eksplicitno upotrebljavaju (mada se nešto češće eksplicitno definišu), što ima za posledicu da terminološka omaška često prolazi neprimećeno. U svakom slučaju, ovde ćemo dosledno insistirati na razlici. I destruktor se relativno retko eksplicitno upotrebljava, pri čemu su takve situacije izuzetno specifične i nećemo im posvetiti posebnu pažnju. U daljem tekstu knjige i najvećem broju praktičnih primena programskog jezika C++ možemo bez ikakvih problema smatrati da se destruktor nikada ne pozivaju eksplicitno. Ranije je navedeno da se konstruktor može eksplicitno upotrebljavati za pravljenje objekata ili konverziju tipova. Da ne bi bilo nesporazuma, naglašavamo da i takva primena konstruktora zapravo predstavlja izvršavanje dveju operacija: alokacije i konstrukcije. Eksplicitna primena konstruktora, bez alokacije, jeste moguća, ali ni njoj nećemo posvetiti posebnu pažnju. Više informacija o eksplicitnom definisanju operatora `new` i `delete` i eksplicitnom korišćenju konstruktora i destruktora može se pročitati u [Lippman 2005].

Vrste objekata

Prema načinu pravljenja i uništavanja objekata, svi objekti u programskom jeziku C++ se dele na *statičke*, *automatske* i *dinamičke*:

- *Statički objekti* se prave tokom učitavanja programa u memoriju, pre pozivanja funkcije `main`. Redosled njhovog pravljenja zavisi kako od redosleda njihovog navođenja u tekstu programa, tako i od redosleda i načina povezivanja modula u izvršnu verziju programa. Često je veoma komplikovano izvoditi zaključke o redosledu pravljenja statičkih objekata, zbog čega se preporučuje izbegavanje pisanja koda koji zavisi od tog redosleda. Statički objekti se uništavaju po izlasku iz funkcije `main`, u redosledu obrnutom od redosleda pravljenja. U statičke objekte spadaju svi globalno definisani objekti i statički podaci klasa. (Statički podaci klasa se upotrebljavaju u primeru *6 - Pretraživanje teksta*, na strani 171.)

  Posebnu podvrstu statičkih objekata čine *statički objekti koji se definišu u okviru blokova koda*. Prostor za takve objekte se obezbeđuje pri učitavanju programa u memoriju, ali se oni inicijalizuju tek pri prvom izvršavanju odgovarajućeg bloka koda. Uništavaju se kao i ostali statički objekti, po izlasku iz funkcije `main`.
- Svi lokalno definisani objekti predstavljaju *automatske objekte*. Automatske objekte predstavljaju i *neimenovani privremeni objekti*[1]. Automatski objekti se prave kada se pri izvršavanju koda dođe do definicije promenljive koja predstavlja novi objekat. Definicija lokalne promenljive u tom smislu predstavlja izvršnu naredbu, jer ona u kodu vrlo precizno određuje i mesto i trenutak pravljenja automatskog objekta. Automatski objekti se uništavaju neposredno pre izlaska iz bloka u kome su definisani, u redosledu obrnutom od redosleda pravljenja.
- *Dinamički objekti* su oni koji se prave primenom izraza `new`. Za razliku od statičkih i automatskih objekata čije je trajanje precizno definisano lokalnim segmentom koda, trajanje dinamičkih objekata je teže opisati[2]. Kao što se eksplicitno prave, primenom izraza `new`, dinamički objekti se i uklanjaju eksplicitno – primenom izraza `delete`.

---

[1] Već smo ranije videli da se svaki konstruktor klase može koristiti kao funkcija koja izračunava novi objekat klase. Tako izračunat objekat predstavlja neimenovan privremeni objekat i njegov životni vek je po svemu sličan lokalno definisanim objektima, tj. uništava se neposredno pre napuštanja bloka u kome je napravljen. Razlika je samo u činjenici da takav objekat nije imenovan, pa mu se zbog toga ne može više puta neposredno pristupati.

[2] Naravno da je i trajanje dinamičkih objekata precizno opisano kodom programa. Međutim, dok je za razumevanje životnog veka automatskih objekata dovoljno analizirati samo blok koda u kojima se definišu, za razumevanje životnog veka dinamičkih objekata je, kao što ćemo uskoro videti, obično neophodno analizirati veći broj metoda, pa i ponašanje čitavih klasa.

Svaka definicija statičkog ili automatskog objekta predstavlja i alokaciju i konstrukciju objekta. Slično, izračunavanje konstruktora predstavlja i alokaciju i konstrukciju privremenog neimenovanog objekta, a primena izraza `new` i alokaciju i konstrukciju dinamičkog objekta. Pri tome, način alokacije memorije zavisi od mesta i načina pravljenja objekta, tj. od toga da li se radi o statičkom, automatskom ili dinamičkom objektu, dok konstrukcija objekata zavisi isključivo od broja i tipa parametara, tj. od upotrebljenog konstruktora. Slično, način oslobađanja memorije zavisi od vrste objekta, dok se destrukcija uvek izvršava na jedini mogući način, primenom destruktora.

Pogledajmo sledeći primer i prodiskutujmo redosled pravljenja i uništavanja objekata:

```
int a(1);

main()
{
    ...
    int b(2);
    int* c = new int(3);
    ...
    delete c;
}
```

Redosled pravljenja i uništavanja bi bio sledeći:

1. pri učitavanju programa se automatski pravi statički objekat `a`;
2. poziva se funkcija `main`;
3. izvršava se kod kojim je definisana funkcija `main`, sve do definicije promenljive `b`;
4. pravi se automatski objekat `b`;
5. eksplicitno se pravi dinamički objekat `c`;
6. izvršava se kod funkcije `main`;
7. eksplicitno se uništava dinamički objekat `c`;
8. neposredno pre napuštanja bloka koda, automatski se uništava automatski objekat `b`;
9. po završetku funkcije `main`, pri izbacivanju programa iz memorije, automatski se uništava objekat `a`.

U sva tri slučaja konstrukcija (inicijalizacija) se obavlja na potpuno isti način – konstruktorom sa jednim celobrojnim argumentom. Takođe, u sva tri slučaja deinicijalizacija se obavlja na potpuno isti način – podrazumevanim destruktorom.

Destruktor

Svaki segment programa mora za sobom počistiti sve što nakon njegovog izvršavanja više nije potrebno. Isto pravilo važi i za objekte: *svaki objekat na kraju svog života mora da počisti sve ono što nakon njegovog uništavanja više nije potrebno*.

Metod koji je odgovoran je da počisti sve nepotrebne tragove za objektom koji se uništava se naziva se *destruktor*. Destruktor obezbeđuje neophodnu deinicijalizaciju objekata pre nego što se oni trajno uklone iz memorije. Da bi klasa mogla da se upotrebljava mora imati najmanje jedan (makar podrazumevani) konstruktor, a po potrebi ih možemo definisati i više, sve dok se razlikuju po broju ili tipu argumenata. Kada je u pitanju deinicijalizacija, *svaka klasa ima tačno jedan destruktor* – ukoliko ga ne definišemo eksplicitno, prevodilac će obezbediti podrazumevani destruktor. *Podrazumevani destruktor se ponaša inverzno podrazumevanom konstruktoru: obezbeđuje da svi podaci objekta, uključujući i nasleđene delove i specifične podatke, budu deinicijalizovani primenom destruktora za odgovarajuće tipove.* Ako podrazumevani destruktor nije dovoljan, može se eksplicitno definisati novi.

Destruktor klase je metod čije ime odgovara nazivu klase sa prefiksom '~'. Destruktor nema rezultat ni argumente. Slično kao i kod konstruktora, tip rezultata se ne definiše. U telu destruktora se piše samo kod koji obavlja neophodne deinicijalizacije koje ne obavlja podrazumevani destruktor. Podrazumeva se da će nakon izvršavanja tela eksplicitno definisanog destruktora biti automatski urađeno i sve ono što bi uradio podrazumevani destruktor.

Podrazumevani destruktor je sasvim dovoljan sve dok u okviru inicijalizacije ili života objekta ne dođe do alociranja nekih resursa koje je neophodno osloboditi prilikom uništavanja objekta. Ti resursi mogu biti vrlo različitog karaktera:

- dinamički napravljeni objekti;
- dodatna memorija;
- otvorene datoteke;
- otvoreni komunikacioni resursi, poput TCP/IP portova, cevi i sl.;
- povezivanje sa sistemima za upravljanje bazama podataka i
- razni drugi resursi.

Zajedničko za sve resurse koji zahtevaju eksplicitnu deinicijalizaciju jeste da destruktori odgovarajućih tipova podataka ne oslobađaju sami resurse na koje referišu. Prvi i najčešći tip podataka na koji se ovo odnosi jesu pokazivači, pa možemo zaključiti: *ako struktura neke klase obuhvata i pokazivače, vrlo je verovatno da je toj klasi potreban destruktor*. Pri tome, ipak, moramo biti oprezni, jer pokazivači u nekim slučajevima mogu ukazivati na neke deljene resurse (često statičke) koji nikako ne smeju da se uništavaju svaki put kada se uništava objekat koji ih koristi. Najopštije pravilo koje bismo mogli primeniti je: *ako objekat klase referiše na resurse koji bi nakon uništavanja objekata mogli ostati istovremeno i rezervisani i nedostupni (pa time i trajno rezervisani), onda je takve resurse neophodno oslobađati u destruktoru klase.*

Da ne bi dolazilo do zabune oko toga da li je neki referisani resurs deljen ili ne, obično se primenjuje neformalna konvencija da se na deljene resurse referiše posredstvom referenci, a na privatne resurse, koje je potrebno oslobađati u destruktorima, posredstvom pokazivača. Na žalost, to nije uvek moguće – pre svega u situacijama kada je tokom života objekta potrebno referisati na različite deljene resurse, jer se vrednost referentnog tipa ne može

menjati. Zato je neophodno uvek biti oprezan pri radu sa pokazivačima, jer je suvišno oslobađanje deljenih resursa bar jednako loše kao i neoslobađanje privatnih resursa.

Priču o destruktorima ćemo završiti pisanjem destruktora klase `Lista`. Pošto se elementi jedne liste koriste samo od strane jednog objekta klase `Lista`, očigledno je da se pokazivač `_Pocetak` odnosi na privatni resurs koji je potrebno oslobađati. Trivijalno rešenje bi moglo biti:

```
class Lista
{
public:
...
    ~Lista()
        {
        if( _Pocetak )
            delete _Pocetak;
        }
...
};
```

Ovakvo rešenje ima više slabosti. Jedna je uglavnom kozmetičke prirode – standardom programskog jezika C++ predviđeno je da se pri pozivanju izraza `delete` automatski proverava da li je dati pokazivač prazan ili ne, tako da nije potrebno da to činimo i mi pre nego što ga upotrebimo. Naredna definicija destruktora je ekvivalentna:

```
    ~Lista()
        { delete _Pocetak; }
```

Daleko veći problem predstavlja činjenica da na ovaj način ne uklanjamo sve elemente liste već *samo prvi*! Podsetimo se šta radi izraz `delete`: poziva destruktor za element liste na koji pokazuje pokazivač `_Pocetak` i zatim oslobađa memoriju koju je taj element liste zauzimao. Kako klasa `ElementListe` koristi podrazumevani destruktor, a on ne deinicijalizuje pokazivač na sledeći element na odgovarajući način, jasno je da će svi elementi liste, osim prvog, ostati gde su bili i u stanju u kom su bili. Jedno rešenje je da, slično kao u klasi `Lista`, obezbedimo i destruktor klase `ElementListe`:

```
class ElementListe
{
public:
...
    ~ElementListe()
        { delete Sledeci; }
...
};
```

Sada izgleda da je problem rešen: kada destruktor objekta klase `Lista` eksplicitno pokrene uklanjanje prvog elementa liste, njegov destruktor će pokrenuti uklanjanje drugog i tako redom sve do kraja liste – nakon što bude oslobođena memorija koju zauzima poslednji element liste, oslobodiće se i memorija koju zauzima pretposlednji element liste i tako redom sve do prvog elementa liste. Realnost je nešto neprijatnija, pa ovako napisani destruktori u praksi neće funkcionisati sa dugačkim listama. Ovaj put uzrok problema leži u primeni

rekurzije i tehnikama njene implementacije u programskom jeziku C++. Uopšteno rečeno, u većini imperativnih programskih jezika pozivanje funkcija i metoda se odvija uz zapisivanje argumenata, adrese povratka i rezultata na računskom steku. Kako taj stek ima ograničenu veličinu, a koja zavisi i od operativnog sistema i od prevodioca, duboka rekurzija će prekoračiti dopuštenu veličinu steka i imati fatalan efekat po izvršavanje programa. Dopuštena dubina rekurzije je u praksi daleko manja nego što bismo mogli pretpostaviti uzimajući u obzir raspoloživu količinu memorije savremenih računara[3]. Zbog toga je rekurziju potrebno izbegavati uvek kada dubina nije unapred poznata ili se pretpostavlja da bi mogla biti dovoljno velika da dovede do problema. Naš prethodni primer (gde imamo milion rekurzivnih poziva) doveo bi do problema u mnogim okruženjima. Kako je pri pisanju klasa potrebno težiti da one budu upotrebljive u različitim situacijama, neophodno je da se vodi računa o upotrebi rekurzije, jer bi se u suprotnom onemogućila njihova primena u okruženjima sa ograničenom veličinom steka.

Nerekurzivno rešenje ima, u ovom slučaju, još jedan značajan kvalitet – omogućava nam da elemente liste posmatramo kao pomoćne podatke, a da ponašanje opisujemo isključivo u okviru klase `Lista`. Pravi značaj takvog pristupa ćemo moći da sagledamo tek kada vidimo da će nam i nakon definisanja destruktora ostati još dosta problema vezanih za dinamičke strukture podataka. Rešavanje tih problema na samo jednom mestu, umesto na dva, učiniće nam posao jednostavnijim.

Nerekurzivno rešenje je da destruktor klase `Lista` eksplicitno obriše svaki pojedinačan element liste. Takvo rešenje pretpostavlja da se u klasi `ElementListe` upotrebljava podrazumevani destruktor, tj. da se ne obezbeđuje nikakav eksplicitno definisan destruktor:

```
~Lista()
    {
    for( ElementListe* p=_Pocetak; p; ){
        ElementListe* p1 = p->Sledeci;
        delete p;
        p = p1;
        }
    }
```

Naš primer će se sada možda neprijatno (i svakako nepotrebno) dugo izvršavati, ali će bar zahtevi za memorijom biti razumni, ako je to uopšte moguće reći za program koji ne radi ništa

[3] Specifikacije savremenih operativnih sistema (uključujući Windows, Linux i druge UNIX-olike sisteme) deklarišu da se statička i dinamička alokacija memorije odvijaju od jednog kraja adresnog prostora, a da računski stek raste od drugog kraja adresnog prostora, te da se može širiti dok se ove dve oblasti ne susretnu. Odatle sledi da je veličina steka ograničena samo veličinom adresnog prostora i virtualne memorije računara. Ipak, stvari ovde nisu tako jednostavne. Postoji više specifičnih situacija u kojima se delovi adresnog prostora rezervišu mimo ova dva osnovna koncepta. Jedna od nasloženijih situacija je u slučaju programa koji rade u više niti. Kako sve niti dele isti adresni prostor, a svaka nit mora imati sopstveni stek, posledica je da se u istom adresnom prostoru mora nalaziti više stekova, pri čemu svaki ima neku unapred ograničenu veličinu.

korisno. Sada će program zahtevati količinu memorije koja je potrebna za jednu listu od milion elemenata, a ne milion puta više. Može se reći da se program ispravno ponaša, jer oslobađa sve rezervisane resurse. Doduše, kao što ćemo uskoro videti, iako bi se to moglo reći za konkretan program, za našu klasu to u opštem slučaju još uvek ne važi.

Korak 4 - Kopiranje objekata

U ovom delu teksta posvetićemo se problemu kopiranja objekata. Najpre ćemo pogledati šta se dešava ako kopiranju pristupimo bez razmatranja mogućih posledica:

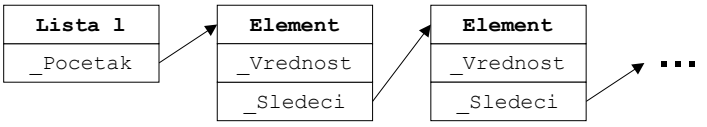
```
main()
{
    // Na isti način kao i do sada, pravimo i koristimo listu l
    Lista l;
    for( int i=0; i<10; i++ )
        l.DodajNaPocetak(i);
    for( int i=0; i<10; i++ )
        cout << l.Element(i) << ' ';
    cout << endl;

    // Pravimo kopiju liste l i proveravamo njen sadržaj
    Lista l1 = l;
    for( int i=0; i<10; i++ )
        cout << l1.Element(i) << ' ';
    cout << endl;

    return 0;
}
```

Prevođenje programa prolazi bez problema, a izvršavanje se završava prilično čudnim porukama, a sve nakon što se potpuno ispravno ispišu i sadržaj liste `l` i sadržaj liste `l1`. Ispada da probleme pravi naredba `return`?

Ne baš. Do problema, ustvari, dolazi pri završavanju bloka u kome je definisan kod funkcije `main`. Ako se osvrnemo na opisan tok života objekata, primetićemo da se pri završavanju bloka uništavaju automatski objekti definisani u bloku. Upravo pri tome dolazi do greške.

Ali u prethodnom primeru smo videli da nam destruktor radi ispravno!? Kako može sada da pravi probleme?

Tačno je da je destruktor ispravan, ali ovde nije u pitanju neispravnost u definiciji destruktora nego neispravnost konteksta u kome se on upotrebljava. Slika 1 predstavlja grafički prikaz memorijske reprezentacije liste `l` i njenih elemenata.

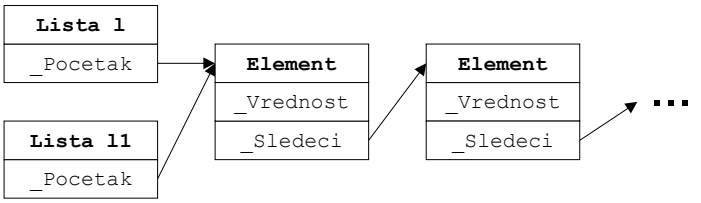


Slika 1: Struktura liste

Kada definišemo novu listu `l1`, ona se inicijalizuje kopiranjem objekta `l`. U svakoj klasi postoji konstruktor čiji argument je objekat istog tipa (ili referenca na objekat istog tipa) – ako ga ne definišemo eksplicitno, prevodilac će ga implicitno definisati. Takav konstruktor se naziva *konstruktor kopije* ili *kopi-konstruktor*. Implicitno definisan (tj. podrazumevani) konstruktor kopije kopira nasleđene delove objekta i sve pojedinačne članove podatke primenjujući konstruktore kopije za odgovarajuće tipove podataka. Štaviše, moramo znati da su naredna dva zapisa međusobno ekvivalentna i da oba predstavljaju pravljenje novog objekta i njegovo inicijalizovanje primenom konstruktora kopije:

```
Lista l1 = l;
Lista l1( l );
```

Ako znamo da konstruktori kopije za proste tipove podataka, pa i za pokazivače, rade tako što prepisuju sadržaj, možemo ispravno zaključiti da su nakon konstruisanja objekta `l1` naši objekti u memoriji organizovani kao što je predstavljeno na slici 2.



Slika 2: Dvostruko referisanje na iste elemente liste

To možemo jednostavno proveriti: dodaćemo novi element listi `l1` i ustanoviti da je on istovremeno i novi element liste `l`:

```
main()
{
...
    Lista l1 = l;
    l1.DodajNaKraj(9999);
    for( int i=0; i<11; i++ )
        cout << l.Element(i) << ' ';
    cout << endl;
...
}
```

Primetimo da će ovako napravljeni objekti klase `Lista` deliti iste elemente liste. To je osnovni uzrok problema. Naime, kada kasnije dođe do uništavanja objekata `l` i `l1`, svaki od njih će tokom destrukcije pokušati da eksplicitno, primenom izraza `delete`, uništi elemente liste. Međutim, kada se prvi od ovih objekata uspešno uništi, prostor u memoriji, koji su zauzimali elementi liste, postaje slobodan. Ukoliko zatim dođe do bilo kakve alokacije memorije, prilično je velika verovatnoća da alociran prostor bude na istom mestu u memoriji na kome su se nalazili neki elementi liste. Problem je u tome što drugi objekat i dalje smatra da se tu nalaze elementi liste, pa će pokušavati da sadržaj memorije i dalje tumači kao zapis elemenata liste. Posledice takvog neispravnog pristupanja uglavnom su nepredvidive i

najčešće dovode do fatalnih grešaka u radu programa ili do problema koji mogu ugroziti ispravnost rezultata ili bezbednost sistema[4]. Primetimo da čak i oslobađanje memorije obuhvata čitanje podataka. Svako oslobađanje memorije započinje čitanjem informacija o bloku koji se oslobađa, a koje se često zapisuju neposredno ispred tog bloka (što, naravno, zavisi od implementacije biblioteke za rad sa memorijom). Zbog toga biblioteka za rad sa memorijom i operativni sistem onemogućavaju rad programa ukoliko se iz načina pristupanja memoriji ili pozivanja funkcija za rad sa memorijom može zaključiti da program pristupa oblastima memorije koje su prethodno oslobođene ili čak nikada nisu ni bile rezervisane od strane tog programa.

U prethodnom odeljku nismo naglasili koji se objekat prvi uništava. Namerno se nismo želeli vezivati za redosled, jer je u ovom slučaju krajnje nebitan – posledice su iste bez obzira na redosled. Ipak, dobro je znati da *uništavanje automatskih objekata teče obrnutim redosledom u odnosu na njihovo pravljenje*. Tako će ovde prvo biti uništena lista `l1`, a zatim će se uništiti (doduše, u našem primeru će to samo biti pokušano, zbog problema na koje smo već ukazali) i lista `l`.

Greške pri pristupanju nedodeljenoj memoriji

Biblioteke za rad sa memorijom obično od operativnog sistema uzimaju samo velike blokove memorije, a zatim programima male blokove izdvajaju iz tako rezervisanih velikih blokova. To ima za posledicu da neka oblast memorije, koja nije dodeljena programu, može biti implicitno rezervisana za taj program od strane biblioteke za rad sa memorijom. Da sve bude još lepše, i operativni sistem i biblioteke za rad sa memorijom obično ne rukuju proizvoljnim veličinama blokova memorije, već ih zaokružuju na neki veći broj (obično je to prvi ne manji broj deljiv nekim stepenom broja dva, pri čemu delioc zavisi od veličine traženog bloka i obično odgovara veličini reči adrese ili veličini stranice operativnog sistema). Zbog toga posledice pristupanja nedodeljenim oblastima memorije, bilo da se radi o oblastima koje su oslobođene ili nikada nisu ni bile rezervisane, nisu uvek iste, ali najčešće su fatalne ili prikriveno fatalne. Razlikujemo tri osnovna slučaja:

1. Svaki pokušaj pristupanja oblasti memorije koja nije dodeljena programu niti je za njega rezervisana od strane biblioteke, dovodi do prekidanja rada programa od strane operativnog sistema. Koliko god to izgledalo neprijatno, to je najbolji slučaj, jer se najjednostavnije ispoljava i prepoznaje.
2. Ukoliko se pristupa memoriji koja nije dodeljena programu, ali je za njega rezervisana od strane biblioteke, problem se neće ispoljiti neposredno, već će se čitati ili menjati oblasti čija funkcija nije definisana. Takav slučaj je *veoma* teško prepoznati jer može imati za posledicu naizgled ispravno ponašanje programa.

[4] Autori zlonamernih programa mogu relativno lako zloupotrebiti greške drugih programa pri radu sa memorijom. Poznat je problem sa serverima i pretraživačima Veba koji su neproveravajući ispravnost prosleđenih podataka upisivali zlonamerni kod u memoriju i zatim ga izvršavali.

Poseban problem je što ta oblast memorije u nekom trenutku može biti vraćena operativnom sistemu, čime se stvaraju uslovi za prvi slučaj, ili dodeljena programu za neke druge podatke, čime se stvaraju uslovi za treći slučaj.

3. Najneugodniji slučaj imamo ukoliko se pristupa memoriji koja je dodeljena programu za neke druge namene. Ni ovaj slučaj ne proizvodi neposredne fatalne posledice, već se čitaju ili menjaju oblasti memorije koje pripadaju nekim drugim, a ne očekivanim podacima. Takve greške se prepoznaju posredno, ali često sa zadrškom, tek pri upotrebi izmenjenih podataka.

Zbog toga što neispravno pristupanje memoriji ima fatalne ili veoma teško prepoznatljive greške, preventiva je *neophodna*.

Plitko i duboko kopiranje

Za opisan način kopiranja objekata klase `List` kažemo da predstavlja *plitko kopiranje*. Plitko kopiranje podrazumeva kopiranje referenci (ili pokazivača) na neke resurse, ali ne i samih resursa. Nasuprot tome, *duboko kopiranje* predstavlja kopiranje čitavih resursa. Plitko kopiranje se primenjuje u slučaju deljenih resursa, dok se duboko kopiranje primenjuje u slučaju privatnih resursa koji se ne smeju deliti (pa se zato moraju kopirati).

U vezi sa tipom kopiranja važi jedno veoma važno pravilo: *resurse je potrebno duboko kopirati ako i samo ako ih destruktor oslobađa*[5]. Značaj ovog pravila je sasvim lako uočiti ako se analizira šta se dešava ukoliko se ono ne poštuje:

- ako se resursi oslobađaju u destruktoru, a kopiranje se izvodi plitko, a ne duboko, tada će pri uništavanju iskopiranih objekata dolaziti do višestrukog oslobađanja resursa – upravo ovaj problem koji mi imamo sa uništavanjem kopiranih objekata klase `Lista`;
- ako se resursi ne oslobađaju, a kopiranje se izvodi duboko, tada će svako kopiranje objekata proizvoditi jednu kopiju resursa više koja nikada neće biti oslobođena – tako dolazi do tzv. *curenja memorije*, tj. do pojave tihog i postepenog gubljenja slobodne memorije, što se kasnije veoma teško otkriva i ispravlja.

[5] Primetimo da se u literaturi može naići i na nešto blaži oblik ovog pravila, u kome umesto apsolutnog uslova „ako i samo ako“ stoji nešto blaža napomena „skoro uvek“. Postoji više posebnih slučajeva, a najčešće je u pitanju neki vid *brojanja referenci*. To su situacije u kojima se deljenje resursa ostvaruje brojanjem koliko se puta resurs upotrebljava. Umesto dubokog kopiranja samo se povećavaju brojači referenci, a pri destrukciji se oni najpre smanjuju, a resursi se zaista oslobađaju tek kada ih više niko ne koristi, tj. kada brojači referenci padnu na 0. Ipak, semantički smisao promene vrednosti brojača referenci je upravo kopiranje (povećavanje brojača) i oslobađanje (smanjivanje brojača) resursa, pa ako tako posmatramo, prethodno navedeno pravilo važi i u svom strogom obliku. Imajući u vidu namenu ovog teksta, u cilju isticanja značaja ovog pravila odabrana je njegova stroga forma.

Konstruktor kopije

Videli smo da je *konstruktor kopije* onaj konstruktor koji za argument ima objekat iste klase. Uobičajeno je da se argument prenosi po referenci (zbog efikasnosti), kao i da bude konstantan (jer ga nije potrebno menjati). Kao i svaki drugi konstruktor, i konstruktor kopije je zadužen za ispravno inicijalizovanje članova podataka. U našem slučaju, konstruktor kopije klase `Lista` mora da obezbedi duboko kopiranje elemenata liste. Kada bi klasa `ElementListe` imala obezbeđen konstruktor kopije, koji izvodi duboko kopiranje, tada bismo konstruktor kopije klase `Lista` mogli definisati sasvim jednostavno:

```
Lista( const Lista& l )
    : _Pocetak( l._Pocetak ? new ElementListe(*l._Pocetak) : 0 )
    {}
```

Ako originalna lista (čiju kopiju pravimo) nije prazna, tada pravimo duboku kopiju njenog prvog elementa i u listi kopiji inicijalizujemo pokazivač na početak liste pokazivačem na napravljenu kopiju prvog elementa. Ukoliko je originalna lista prazna (`l._Pocetak` je 0), tada će i kopija biti prazna lista, pa je dovoljno pokazivač `_Pocetak` inicijalizovati nulom. Važno je primetiti da ovakav konstruktor kopije radi ispravno samo ako konstruktor kopije klase `ElementListe` vrši duboko kopiranje, jer ćemo u suprotnom problem samo prebaciti na drugu klasu ali ga nećemo eliminisati – uspešno će se iskopirati objekat klase `Lista` i prvi element liste, ali će svi ostali elementi i dalje biti deljeni. Čak i pored toga možemo imati problema, jer ako je taj konstruktor kopije implementiran na sličan način (ali za pokazivač `Sledeci` umesto za pokazivač `_Pocetak`), onda će pri kopiranju dolaziti do duboke rekurzije, slično kao u prvoj verziji destruktora.

Zato pišemo rešenje bez rekurzije:

```
Lista( const Lista& l )
    {
    // Ako lista nije prazna...
    if( l._Pocetak ){
        // Iskopiramo prvi element.
        _Pocetak = new ElementListe( *l._Pocetak );
        ElementListe* stari = l._Pocetak;
        ElementListe* novi = _Pocetak;
        // Dok ima još neiskopiranih elemenata...
        while( stari->Sledeci ){
            // ...kopiramo sledeći element...
            novi->Sledeci = new ElementListe( *stari->Sledeci );
            // ...i pomeramo se za jedno mesto.
            novi = novi->Sledeci;
            stari = stari->Sledeci;
            }
        }
    // Ako je lista prazna...
    else
        _Pocetak = 0;
    }
```

Za navedeno nerekurzivno rešenje neophodno je da konstruktor kopije klase `ElementListe` izvodi samo *plitko kopiranje* (tj. dovoljno je da ne postoji eksplicitno definisan konstruktor kopije klase `ElementListe`).

Sada naš primer radi ispravno.

Operator dodeljivanja

Izmenimo sada funkciju `main` dodavanjem još par redova koda:

```
main()
{
    Lista l;
    for( int i=0; i<10; i++ )
        l.DodajNaPocetak(i);
    for( int i=0; i<10; i++ )
        cout << l.Element(i) << ' ';
    cout << endl;

    Lista l1 = l;
    for( int i=0; i<10; i++ )
        cout << l1.Element(i) << ' ';
    cout << endl;

    l1 = l;
    for( int i=0; i<10; i++ )
        cout << l1.Element(i) << ' ';
    cout << endl;

    return 0;
}
```

Novi segment koda skoro da se ne razlikuje od segmenta sa kojim smo prethodno imali probleme. Jedina razlika je u tome što je na početku prethodnog segmenta stajala definicija objekta `l1` i primena konstruktora kopije:

```
Lista l1 = l;
```

a sada imamo red koji je sintaksno veoma sličan:

```
l1 = l;
```

Prevedimo i pokrenimo program. Da li nekoga iznenađuje što sada dobijamo izveštaj koji se sastoji od tri identična reda? Verovatno ne, jer to smo i očekivali. Ali zašto se ponovo pojavljuju slične poruke kao i pre obezbeđivanja konstruktora kopije? Da li smo napravili neku grešku pri pisanju konstruktora kopije, pa on sada ne radi?

Nikakav pokušaj traženja greške u konstruktoru kopije ne bi nas značajno približio rešenju problema, iz jednostavnog razloga što se konstruktor kopije ne upotrebljava u novom segmentu koda. Pošto nismo definisali niti eksplicitno napravili novi objekat klase `Lista`, naravno da se ne primenjuje nikakav konstruktor te klase, pa ni konstruktor kopije. Iako je red na koji smo ukazali po zapisu sličan definiciji nove promenljive `l1`, sada se ne radi o definisanju novog objekta već o dodeljivanju nove vrednosti postojećem objektu. Za dodeljivanje vrednosti objektu klase upotrebljava se operator `=`. Kao i u slučaju konstruktora kopije, ukoliko ovaj operator nije eksplicitno definisan, prevodilac će obezbediti implicitnu

definiciju. Implicitno definisan operator = ponaša se slično implicitno definisanom konstruktoru kopije po tome što izvodi *plitko kopiranje*. Da bi se pri dodeljivanju izvodilo duboko kopiranje, neophodno je da u operatoru dodeljivanja uradimo praktično isti posao kao i u konstruktoru kopije. Zbog toga ćemo logiku kopiranja liste izdvojiti u privatan metod `init`:

```
class Lista
{
public:
...
    Lista( const Lista& l )
        { init(l); }
...
private:
    void init( const Lista& l )
        {
        if( l._Pocetak ){
            _Pocetak = new ElementListe( *l._Pocetak );
            ElementListe* stari = l._Pocetak;
            ElementListe* novi = _Pocetak;
            while( stari->Sledeci ){
                novi->Sledeci = new ElementListe(*stari->Sledeci);
                novi = novi->Sledeci;
                stari = stari->Sledeci;
                }
            }
        else
            _Pocetak = 0;
        }
...
};
```

Pri definisanju operatora dodeljivanja uobičajeno je da se argument prenosi kao referenca na konstantan objekat iste klase. Znači, mogli bismo napisati ovakav operator dodeljivanja u okviru klase `Lista`:

```
    void operator = ( const Lista& l )
        { init(l); }
```

Sada prevođenje i izvršavanje programa prolazi bez problema. Znači li to da je sve u redu? Sve objekte ispravno kopiramo i sve objekte ispravno uništavamo – izgleda da je sve u redu?

Pa, stoji da sada objekte ispravno kopiramo. Stoji i da ih ispravno uništavamo, ali samo one koje uopšte i pokušavamo da uništavamo. Ako malo bolje analiziramo naš kod možemo primetiti da će biti napravljene *tri* liste, a da se uništavaju samo *dve*!

- Prva lista koja se pravi je ona koju eksplicitno definišemo i popunjavamo pod imenom `l`.
- Druga lista je lista `l1` koja nastaje primenom konstruktora kopije.
- Treća lista elemenata nastaje kao rezultat primene operator dodeljivanja.
- Pri uništavanju objekta `l` ispravno će se uništiti prva lista elemenata.

- Pri uništavanju objekta `l1` ispravno će se uništiti treća lista elemenata.

Sviđalo se to nama ili ne, druga lista elemenata ostaje trajno zaboravljena u memoriji. Do problema dolazi pri primeni operatora dodeljivanja, kada se zaboravlja čitava lista elemenata koja predstavlja prethodnu vrednost objekta. Taj problem se može izbeći na samo jedan način – uništavanjem prethodnog sadržaja objekta pre nego što mu se dodeli novi. Ako bismo to obezbedili, mogli bismo računati na ispravno ponašanje. Analizom postojećeg koda lako možemo zaključiti da je prethodni sadržaj potrebno uništavati na potpuno isti način kao što se to čini pri deinicijalizaciji u okviru destruktora. Zato taj deo koda izdvajamo u privatan metod `deinit`:

```
class Lista
{
public:
...
    ~Lista()
        { deinit(); }
...
private:
    void deinit()
        {
        for( ElementListe* p=_Pocetak; p; ){
            ElementListe* p1 = p->Sledeci;
            delete p;
            p = p1;
            }
        }
...
};
```

Pozivanjem metoda `deinit` neznatno usložnjavamo operator dodeljivanja:

```
void operator = ( const Lista& l )
    {
    deinit();
    init(l);
    }
```

Nova definicija operatora dodeljivanja je sasvim prirodna, jer dodeljivanje nove vrednosti sada eksplicitno definišemo kao uništavanje starog sadržaja i pravljenje novog. Jedino što je zajedničko za stari i novi objekat jeste položaj u memoriji, a time i ime koje koristimo da bismo mu pristupali.

Uočimo još jedan problem koji se može pojaviti ako eksplicitno ili implicitno dođe do primene operacije poput:

```
x = x;
```

Stoji da će malo ko napisati nešto slično ovome u svom kodu, ali pri upotrebi pokazivača nisu retke situacije kada se može primeniti ekvivalentna operacija. Na primer, ako pokazivači `p` i `q` pokazuju na isti objekat, tada je naredna operacija ekvivalentna prethodnoj:

```
*p = *q;
```

Ovakve situacije često nije jednostavno prepoznati i predupredititi. Zbog toga je dobro da se problem reši pri definisanju operatora dodeljivanja. Najpre razmotrimo šta će se dogoditi ako ne razmatramo ovaj slučaj:

- sadržaj objekta će biti deinicijalizovan;
- pokušaće se pravljenje kopije objekta, ali to ne može uspeti jer je objekat deinicijalizovan!

Rešavanje ovog problema je sasvim jednostavno, jer je u takvim situacijama dovoljno ne raditi ništa. Ostaje još samo da se takva situacija prepozna. Tu koristimo činjenicu da se *dve reference odnose na jedan isti objekat ako i samo ako se odnose na istu lokaciju u memoriji*. Objekat koji dodeljujemo nalazi se na adresi `&l`, a objekat kome dodeljujemo se nalazi na adresi `this`. Prema tome, ispravna definicija operatora dodeljivanja je:

```
void operator = ( const Lista& l )
    {
    if( this != &l ){
        deinit();
        init(l);
        }
    }
```

Da bi primene operatora dodeljivanja mogle da se nadovezuju, uobičajeno je (mada ne i neophodno) da on vrati kao rezultat referencu na izmenjen objekat. U skladu sa time pišemo:

```
Lista& operator = ( const Lista& l )
    {
    if( this != &l ){
        deinit();
        init(l);
        }
    return *this;
    }
```

Diskusija

Videli smo da je semantika kopiranja objekata tesno vezana za semantiku njihove deinicijalizacije. Štaviše, možemo formulisati jedno od najvažnijih pravila pri pisanju klasa na programskom jeziku C++: *ako je za ispravno funkcionisanje klase neophodan neki od tri opisana metoda (destruktor, konstruktor kopije, operator dodeljivanja), tada su neophodna sva tri metoda.* Ostavljamo čitaocu da analizira i pokaže do kakvih problema može doći ako se implementira neki, ali ne i svi ovi metodi. Pri tome skrećemo pažnju na činjenicu da se svaki od ovih metoda može pozivati kako eksplicitno tako i implicitno (recimo pri pozivanju funkcija čiji se argumenti ili rezultat prenose po vrednosti), pa se nikako ne smeju izostaviti uz samouverenu tvrdnju poput „ja ih nigde ne koristim pa mi nisu potrebni“. Njihova primena će pre ili kasnije zatrebati, a onda će se u njihovom odsustvu primeniti njihove podrazumevane verzije koje ne obavljaju potreban posao, a sve bez ikakvog upozorenja korisnika klase jer prevođenje prolazi bez ikakvih problema!

Zbog značaja problema i činjenice da za neke klase zaista nije potrebno (ili dopušteno) kopiranje i dodeljivanje objekata, predstavljamo jednu tehniku koja zabranjuje primenu konstruktora kopije i operatora dodeljivanja. Rešenje kojim se sprečava upotreba ovih metoda od strane korisnika klase jeste njihovo proglašavanje za privatne metode. Sa druge strane, njihova eksplicitna ili implicitna primena od strane metoda klase, koji smeju koristiti privatne metode, sprečava se izostavljanjem definicija ovih metoda. Pri tome se koristi osobina programskog jezika C++ da se u slučaju da je neki metod deklarisan ali ne i definisan (tj. nedostaje mu telo), greška prijavljuje samo ako se taj metod negde i koristi, bilo eksplicitno ili implicitno. Ako bismo konkretno rešenje primenili na primeru klase `Lista`, to bi izgledalo ovako:

```
class Lista
{
...
private:
    Lista( const Lista& l );
    Lista& operator = ( const Lista& l );
...
};
```

Ako bi pri implementaciji klase `Lista` ili pri njenoj upotrebi došlo do eksplicitne ili implicitne primene operatora dodeljivanja ili konstruktora kopije, prevodilac bi odgovarajućom porukom obavestio programera da je došlo do greške.

Korak 5 - Unapređivanje pristupanja elementima liste

Kada god imamo posla sa nekom kolekcijom podataka, potrebno je omogućiti obrađivanje elemenata kolekcije na što efikasniji način. U slučaju naše klase `Lista` postoji samo jedan način da se obrade elementi liste i taj način je već korišćen u primerima:

```
for( int i=0; i<10; i++ )
    cout << l.Element(i) << ' ';
```

Neposredan pristup elementima možemo učiniti prirodnijim i jednostavnijim ako metod `Element` zamenimo operatorom `[]`. Operator `[]` je još jedan od standardnih operatora programskog jezika C++ za koji možemo definisati kako da se ponaša za objekte naše klase:

```
int operator []( int i ) const
    {
    const Element* p = _Pocetak;
    for( int j=0; j<i; j++ )
        p=p->_Sledeci;
    return p->_Vrednost;
    }
```

Napisaćemo još i metode koji proveravaju da li je lista prazna i izračunavaju broj elemenata liste:

```
bool Prazna() const
    { return !_Pocetak; }
```

```
int Velicina() const
    {
    int n=0;
    for( ElementListe* p=_Pocetak; p; p=p->Sledeci )
        n++;
    return n;
    }
```

Pošto je operator `[]` (tj. metod `Element`) prilično neefikasan, jer je složenost pristupanja elementu linearno proporcionalna njegovom rednom broju, bilo bi dobro da obezbedimo neki efikasniji način da se pristupa elementima liste. Zbog toga obezbeđujemo metod `Pocetak`, koji vraća pokazivač na prvi element liste:

```
const ElementListe* Pocetak() const
    { return _Pocetak; }
```

Sada se ispisivanje svih elemenata liste može izvesti daleko efikasnije, ali je za to neophodno i poznavanje strukture elemenata liste:

```
for( const ElementListe* p=l.Pocetak(); p; p=p->Sledeci )
    cout << p->Vrednost << ' ';
```

Na prvi pogled, mogao bi se steći utisak da se obezbeđivanjem javnih metoda `Pocetak` i `Sledeci` čini javnim deo implementacije naše liste, jer korisnik klase sada postaje svestan da lista ima početak i da iza svakog elementa (osim poslednjeg) postoji sledeći element. Međutim, iako u značajnoj meri odražava implementaciju klase `Lista`, to predstavlja deo *ponašanja* liste. Svaka struktura podataka koju koristimo ima neke značajne osobine po kojima se ona razlikuje od drugih struktura i koje posredno ili neposredno određuju i njen interfejs. U slučaju liste pretpostavlja se da su elementi uvezani u sekvencu koja ima početak i kraj, pa omogućavanje pristupanja tom početku i sledećim elementima, nije neprirodan korak. Jedino što izvesno predstavlja određen problem jeste neposredno pristupanje članu podatku `Sledeci`. U daljem tekstu će klasa `ElementListe` biti dalje unapređena, uključujući implementiranje javnog metoda `Sledeci` i dalje ograničavanje funkcionalnosti ove klase.

Korak 6 - Efikasno dodavanje elemenata na kraj liste

Nije teško uočiti da je dodavanje elemenata na kraj liste daleko složenije nego dodavanje na početak. Tako je samo zato što neposredno raspolažemo pokazivačem na prvi element liste, ali ne i pokazivačem na poslednji. Efikasnost dodavanja elemenata na kraj liste možemo značajno povećati proširivanjem klase `Lista` pokazivačem na poslednji element. Neophodno je izmeniti veći broj metoda, ali te izmene su uglavnom sasvim jednostavne:

```
class Lista
{
public:
    Lista()
        : _Pocetak(0),
          _Kraj(0)
        {}
...
```

```
    void DodajNaPocetak( int n )
        {
        _Pocetak = new ElementListe( n, _Pocetak );
        if( !_Kraj )
            _Kraj = _Pocetak;
        }

    void DodajNaKraj( int n )
        {
        ElementListe* novi = new ElementListe( n, 0 );
        if( !_Kraj )
            _Pocetak = _Kraj = novi;
        else{
            _Kraj->Sledeci = novi;
            _Kraj = novi;
            }
        }
...
private:
    void init( const Lista& l )
        {
        if( l._Pocetak ){
            _Pocetak = new ElementListe( *l._Pocetak );
            ElementListe* stari = l._Pocetak;
            ElementListe* novi = _Pocetak;
            while( stari->Sledeci ){
                novi->Sledeci = new ElementListe(*stari->Sledeci);
                novi = novi->Sledeci;
                stari = stari->Sledeci;
                }
            _Kraj = novi;
            }

        else
            _Kraj = _Pocetak = 0;
        }
...
    ElementListe*   _Pocetak;
    ElementListe*   _Kraj;
};
```

Korak 7 - Uklanjanje elemenata liste

Problem kojim se ovde još nismo bavili jeste uklanjanje elemenata liste. Uklanjanje prvog elementa liste je relativno jednostavno:

```
    void ObrisiPrvi()
        {
        if( _Pocetak ){
            ElementListe* drugi = _Pocetak->Sledeci;
            delete _Pocetak;
            _Pocetak = drugi;
            if( !_Pocetak )
                _Kraj = 0;
            }
        }
```

Brisanje poslednjeg elementa je nešto složenije, jer je neophodno najpre pronaći pretposlednji element liste:

```
void ObrisiPoslednji()
    {
    if( _Pocetak ){
        // Ako lista ima bar dva elementa
        if( _Pocetak->Sledeci ){
            // Tražimo pretposlednji element...
            ElementListe* pretposlednji = _Pocetak;
            while( pretposlednji->Sledeci->Sledeci )
                pretposlednji = pretposlednji->Sledeci;
            // Brišemo poslednji
            delete pretposlednji->Sledeci;
            pretposlednji->Sledeci = 0;
            _Kraj = pretposlednji;
            }
        // Ako lista ima tačno jedan element
        else{
            delete _Pocetak;
            _Pocetak = _Kraj = 0;
            }
        }
    }
```

Korak 8 - Prijateljske klase i umetnute klase

Enkapsulacijom smo se već bavili u nekoliko navrata, ali sada imamo potrebu za specifičnim oblikom enkapsulacije. Svi metodi i podaci klase `Lista` koje ne bi trebalo neposredno upotrebljavati već su proglašeni za privatne. Međutim, prilikom unapređivanja pristupanja elementima liste načinili smo jednu izmenu koja dovodi u pitanje pouzdanost čitavog rešenja – omogućili smo neposredan pristup objektima klase `ElementListe`. Korisnik klase `Lista` sada može doći u iskušenje da preduzme neke rizične korake:

- Promenom tipa pokazivača može da pokuša da neposredno menja elemente liste. Ovakva primena nije posebno zabrinjavajuća, jer ako se neko usudi da na takav način dovodi u pitanje stabilnost i pouzdanost rešenja, tu malo šta možemo učiniti. Jednostavnom analizom koda, makar i samo definicije naše klase, korisnik može doći do informacija koje su mu potrebne za nedokumentovan neposredan pristup podacima. Nije ni moguće ni potrebno štititi se od korisnika koji su spremni na takve korake.
- Primenom konstruktora klase `ElementListe` korisnik može pokušati da sam pravi liste. Ovo već zaslužuje posebnu pažnju, jer korisnik primenom javnog interfejsa klase `ElementListe` može načiniti potpuno neispravne strukture podataka tako što bi, na primer, mogao napraviti dve liste koje imaju zajedničke sve elemente osim prvog. Bilo bi najbolje nekako sprečiti korisnika da pravi objekte ove klase.

Imamo situaciju u kojoj korisniku naših klasa članovi podaci elemenata liste smeju biti dostupni samo za čitanje, dok klasi `Lista` moraju biti dostupni i za menjanje. Konstruktori klase `ElementListe` ne bi smeli da budu dostupni korisniku klase, ali moraju biti dostupni klasi `Lista`. Kada imamo slučaj da jedna klasa služi kao sredstvo implementiranja druge klase, ali i korisnici moraju da pristupaju nekim podacima ili metodima, možemo upotrebiti koncept *prijateljskih klasa*. Ako unutar klase `ElementListe` navedemo deklaraciju:

```
friend class Lista;
```

naglasićemo da metodi klase `Lista` mogu pristupati svim privatnim podacima i metodima klase `ElementListe`. Slično tome, možemo definisati i prijateljske funkcije, kao na primer:

```
friend int f( int, char* );
```

Sada možemo napisati klasu `ElementListe` tako da vidljivost bude zadovoljena. Da bismo izmenili vidljivost konstruktora kopije moramo ga eksplicitno definisati, ali pri tome moramo da zadržimo semantiku plitkog kopiranja. Štaviše, možemo se odlučiti i da zabranimo upotrebu konstruktora kopije i operatore dodeljivanja za ovu klasu. Posle toga će biti neophodno izmeniti način upotrebe ove klase u metodima klase `Lista`, tj. imena podataka `_Sledeci` i `_Vrednost`, kao i upotrebu konstruktora kopije zameniti upotrebom raspoloživih konstruktora. Posebno, da korisnik ne bi mogao da namerno ili slučajno ošteti listu elemenata eksplicitnim brisanjem pojedinačnih elemenata, poželjno je da se i destruktor proglasi za privatran metod:

```
class ElementListe
{
public:
    int Vrednost() const
        { return _Vrednost; }

    const ElementListe* Sledeci() const
        { return _Sledeci; }

private:
    ElementListe( int v, ElementListe* s )
        : _Vrednost(v),
          _Sledeci(s)
        {}

    ~ElementListe()
        {}

    ElementListe( const ElementListe& e );
    ElementListe& operator = ( const ElementListe& e );

    int             _Vrednost;
    ElementListe*   _Sledeci;

    friend class Lista;
};
```

Sada korisnik klase `ElementListe` može samo da dobije vrednost elementa i sledeći element, a nije u mogućnosti da pravi ili uklanja objekte ove klase. Samo metodi klase `Lista` će moći da prave i uklanjaju elemente.

Iako je po pitanju funkcionalnosti ovo sasvim dovoljno, može se ići i dalje. Programski jezik C++ omogućava definisanje jedne klase unutar druge klase. Na taj način se korisniku jasno stavlja do znanja da tzv. *umetnuta klasa* (ili *ugnežđena klasa*) predstavlja sredstvo za implementiranje veće klase. Ako bismo takav koncept primenili na klasu `ElementListe`, bilo bi i iz mesta na kome je klasa definisana i iz njenog imena jasno da služi za implementiranje klase `Lista`.

Uobičajeno je da se za umetnute klase upotrebljavaju jednostavnija imena. U našem slučaju nema potrebe da se umetnuta klasa zove `ElementListe` već je dovoljno da se zove `Element`, jer je zbog mesta na kome je definisana jasno da se radi o elementima liste.

```
//---------------------------------------------------------------
//  Klasa Lista
//---------------------------------------------------------------
class Lista
{
public:
    //-----------------------------------------------------------
    //  Klasa Element
    //-----------------------------------------------------------
    class Element
    {
    public:
        int Vrednost() const
            { return _Vrednost; }

        const Element* Sledeci() const
            { return _Sledeci; }

    private:
        Element( int v, Element* s=0 )
            : _Vrednost(v),
              _Sledeci(s)
            {}

        ~Element()
            {}

        Element( const Element& e );
        Element& operator = ( const Element& e );

        int         _Vrednost;
        Element *   _Sledeci;

        friend class Lista;
    };
...
};
```

U okviru klase `Lista` umetnuta klasa se referiše navođenjem njenog imena, dok se u svim ostalim slučajevima ona referiše u obliku:

```
Lista::Element
```

Korak 9 - Optimizacija

Konačno, kada imamo potpuno funkcionalnu klasu, možemo pokušati da neke metode učinimo jednostavnijim i efikasnijim. Prvi kandidat je najveći metod – `init`. Pri kopiranju liste kod možemo učiniti jednostavnijim ako jednako apstrahujemo kopiranje prvog elementa i kopiranje svih ostalih elemenata. Pošto se pri kopiranju elemenata stalno menjaju pokazivači na naredni element, možemo da umesto pokazivača na poslednji napravljeni element vodimo pokazivač na pokazivač koji će ukazivati na sledeći element koji je potrebno napraviti:

```
void init( const Lista& l )
    {
    Element* stari = l._Pocetak;
    Element** novi = &_Pocetak;
    _Kraj = 0;
    while( stari ){
        _Kraj = *novi = new Element( stari->_Vrednost );
        stari = stari->_Sledeci;
        novi = &_Kraj->_Sledeci;
        }
    *novi = 0;
    }
```

Ukoliko bi analiza predviđene ili stvarne upotrebe klase `Lista` pokazala da se veoma često izračunava dužina liste, imalo bi smisla učiniti odgovarajući metod efikasnijim uvođenjem brojača elemenata:

```
class Lista
{
public:
    Lista()
        : _Pocetak(0),
          _Kraj(0),
          _BrojElemenata(0)
        {}
...
    void DodajNaPocetak( int n )
        {
        _Pocetak = new Element ( n, _Pocetak );
        if( !_Kraj )
            _Kraj = _Pocetak;
        _BrojElemenata++;
        }

    void DodajNaKraj( int n )
        {
        Element* novi = new Element( n, 0 );
        if( !_Kraj )
            _Pocetak = _Kraj = novi;
        else{
            _Kraj->_Sledeci = novi;
            _Kraj = novi;
            }
        _BrojElemenata++;
        }
```

```
    void ObrisiPrvi()
        {
        if( _Pocetak ){
            Element* drugi = _Pocetak->_Sledeci;
            delete _Pocetak;
            _Pocetak = drugi;
            if( !_Pocetak )
                _Kraj = 0;
            _BrojElemenata--;
            }
        }

    void ObrisiPoslednji()
        {
        if( _Pocetak ){
            if( _Pocetak->_Sledeci ){
                Element* pretposlednji = _Pocetak;
                while( pretposlednji->_Sledeci->_Sledeci )
                    pretposlednji = pretposlednji->_Sledeci;
                delete pretposlednji->_Sledeci;
                pretposlednji->_Sledeci = 0;
                _Kraj = pretposlednji;
                }
            else{
                delete _Pocetak;
                _Pocetak = _Kraj = 0;
                }
            _BrojElemenata--;
            }
        }

    int Velicina() const
        { return _BrojElemenata; }
...
private:
    void init( const Lista& l )
        {
        Element* stari = l._Pocetak;
        Element** novi = &_Pocetak;
        _Kraj = 0;
        while( stari ){
            _Kraj = *novi = new Element( stari->_Vrednost );
            stari = stari->_Sledeci;
            novi = &_Kraj->_Sledeci;
            }
        *novi = 0;
        _BrojElemenata = l._BrojElemenata;
        }
...
    Element*    _Pocetak;
    Element*    _Kraj;
    int         _BrojElemenata;
};
```

2.3 Rešenje

```
#include <iostream>
using namespace std;

//---------------------------------------------------------------
//  Klasa Lista
//---------------------------------------------------------------
class Lista
{
public:
    //-----------------------------------------------------------
    //  Klasa Lista::Element
    //-----------------------------------------------------------
    class Element
    {
    public:
        int Vrednost() const
            { return _Vrednost; }

        const Element* Sledeci() const
            { return _Sledeci; }

    private:
        Element( int v, Element* s=0 )
            : _Vrednost(v),
              _Sledeci(s)
            {}
        ~Element()
            {}

        Element( const Element& e );
        Element& operator = ( const Element& e );

        int         _Vrednost;
        Element *   _Sledeci;

        friend class Lista;
    };
    //-----------------------------------------------------------

    Lista()
        : _Pocetak(0),
          _Kraj(0),
          _BrojElemenata(0)
        {}

    Lista( const Lista& l )
        { init(l); }

    Lista& operator = ( const Lista& l )
        {
        if( this != &l ){
            deinit();
            init(l);
            }
        return *this;
        }
```

```
~Lista()
    { deinit(); }

void DodajNaPocetak( int n )
    {
    _Pocetak = new Element( n, _Pocetak );
    if( !_Kraj )
        _Kraj = _Pocetak;
    _BrojElemenata++;
    }

void DodajNaKraj( int n )
    {
    Element* novi = new Element( n, 0 );
    if( !_Kraj )
        _Pocetak = _Kraj = novi;
    else{
        _Kraj->_Sledeci = novi;
        _Kraj = novi;
        }
    _BrojElemenata++;
    }

void ObrisiPrvi()
    {
    if( _Pocetak ){
        Element* drugi = _Pocetak->_Sledeci;
        delete _Pocetak;
        _Pocetak = drugi;
        if( !_Pocetak )
            _Kraj = 0;
        _BrojElemenata--;
        }
    }

void ObrisiPoslednji()
    {
    if( _Pocetak ){
        if( _Pocetak->_Sledeci ){
            Element* pretposlednji = _Pocetak;
            while( pretposlednji->_Sledeci->_Sledeci )
                pretposlednji = pretposlednji->_Sledeci;
            delete pretposlednji->_Sledeci;
            pretposlednji->_Sledeci = 0;
            _Kraj = pretposlednji;
            }
        else{
            delete _Pocetak;
            _Pocetak = _Kraj = 0;
            }
        _BrojElemenata--;
        }
    }

const Element* Pocetak() const
    { return _Pocetak; }
```

```
    int operator []( int i ) const
        {
        const Element* p = _Pocetak;
        for( int j=0; j<i; j++ )
            p=p->_Sledeci;
        return p->_Vrednost;
        }

    bool Prazna() const
        { return !_Pocetak; }

    int Velicina() const
        { return _BrojElemenata; }

private:
    void init( const Lista& l )
        {
        Element* stari = l._Pocetak;
        Element** novi = &_Pocetak;
        _Kraj = 0;
        while( stari ){
            _Kraj = *novi = new Element( stari->_Vrednost );
            stari = stari->_Sledeci;
            novi = &_Kraj->_Sledeci;
            }
        *novi = 0;
        _BrojElemenata = l._BrojElemenata;
        }

    void deinit()
        {
        for( Element* p=_Pocetak; p; ){
            Element* p1 = p->_Sledeci;
            delete p;
            p = p1;
            }
        }

    Element*    _Pocetak;
    Element*        _Kraj;
    int         _BrojElemenata;
};

//-----------------------------------------------------------------
// Glavna funkcija programa demonstrira upotrebu klase Lista.
//-----------------------------------------------------------------
main()
{
    Lista l;
    for( int i=0; i<10; i++ )
        l.DodajNaPocetak(i);
    for( int i=0; i<10; i++ )
        cout << l[i] << ' ';
    cout << endl;

    Lista l1 = l;
    for( int i=0; i<10; i++ )
        cout << l1[i] << ' ';
    cout << endl;
```

```
        l1 = l;
        l1.ObrisiPrvi();
        l1.ObrisiPoslednji();
        for( const Lista::Element* p = l1.Pocetak(); p;
             p = p->Sledeci()
            )
            cout << p->Vrednost() << ' ';
        cout << endl;

        return 0;
    }
```

2.4 Rezime

Razmotrimo, ukratko, šta bismo dobili da smo pri definisanju pošli od strukture, a ne od ponašanja. Verovatno bismo pod imenom `Lista` definisali nešto po strukturi sasvim slično predstavljenoj klasi `Lista::Element`. Tako dobijena klasa bi izvesno bila upotrebljiva, ali bi omogućavala da se neopreznom upotrebom naprave ozbiljne greške. Takođe, ne bi bilo moguće na jednostavan način proširiti klasu pokazivačem na poslednji element ili brojačem elemenata, pa bi dobijena klasa bila manje efikasna. Time se potvrđuje da je pri definisanju klase jedino ispravno poći od ponašanja, a ne od strukture.

Najvažniji problem koji je obrađen na ovom primeru jeste rad sa dinamičkim strukturama podataka, tj. koncept i pisanje destruktora, konstruktora kopije i operatora dodeljivanja. Usledиće još nekoliko primera u kojima će se koristiti dinamičke strukture podataka, tako da će čitaoci biti u prilici da i praktično provere koliko su im dinamičke strukture podataka i odgovarajući metodi bliski. Radi što boljeg razumevanja, predlažemo čitaocima da analiziraju moguće probleme do kojih može doći usled nepoštovanja pravila *„ako je za ispravno funkcionisanje klase neophodan neki od tri opisana metoda (destruktor, konstruktor kopije, operator dodeljivanja), tada su neophodna sva tri metoda“*.

Pri implementitranju klase `Lista` odlučili smo se da fizičku strukturu liste opišemo objektima klase `Lista::Element`. Pri tome je ta klasa posmatrana samo kao struktura podataka, bez ikakvog značajnog ponašanja. Jedine definisane operacije su tu više da bi upotreba bila sintaksno jednostavnije nego zato što obavljaju neki značajan posao. Na taj način je implementiranje svih metoda klase `Lista` izvedeno lokalno, bez oslanjanja na eventualno složeno ponašanje pojedinačnih elemenata. Naravno da nije moralo tako, ali dobijeno rešenje je kompaktnije i lakše za održavanje. Čitaocu predlažemo da radi vežbe pokuša da neke operacije implementira u okviru klase `Element`. Skrećemo pažnju da to posebno ima smisla za metode koji bi se mogli implementirati rekurzivno, ali istovremeno podsećamo da rekurzija nije dobro sredstvo u imperativnim programskim jezicima, jer donosi probleme u slučaju velike dubine rekurzije.

Napisane klase se mogu dalje unapređivati. Potencijalni zadaci za vežbu bi mogli biti, na primer:

- napisati operator za ispisivanje liste;

- obezbediti efikasno uklanjanje elemenata sa kraja liste obezbeđivanjem povezivanja elemenata u oba smera (tzv. *dvostruko povezane liste*);
- napisati metod za čitanje liste.

U okviru standardne biblioteke za pristupanje elementima kolekcija upotrebljava se koncept iteratora (videti *10.2* Iteratori, na strani 348). Taj koncept će biti detaljnije razmotren i primenjen u primeru *6.2 Korak 4 - Problem evidentiranja reči*, na strani 179. Predlažemo čitaocima da pokušaju da primene koncept iteratora na klasi `Lista`.

3 - Dinamički nizovi

3.1 Zadatak

Napisati klasu `Niz`, koja omogućava rad sa nizovima čija veličina nije definisana u trenutku pisanja i prevođenja programa, već se može dinamički menjati tokom izvršavanja programa. Obezbediti:

- metode za menjanje veličine niza;
- metode za dodavanje elemenata na kraj niza i brisanje sa kraja niza;
- metode za efikasno pristupanje elementima niza na osnovu indeksa;
- ostale metode neophodne za ispravno funkcionisanje dinamičkog niza.

Cilj zadatka

Pisanje klase `Niz` će nam omogućiti da:

- utvrdimo već stečena znanja o radu sa dinamičkom memorijom i dopunimo ih specifičnim tehnikama za rad sa nizovima;
- upoznamo koncept šablona u programskom jeziku C++, sa akcentom na njegovu primenu kao sredstva za ostvarivanje generičkog polimorfizma;
- upoznamo principe na kojima počiva klasa `vector` standardne biblioteke programskog jezika C++;
- upoznamo rad sa izuzecima.

Pretpostavljena znanja

Za uspešno praćenje rešavanja ovog zadatka potrebno je poznavanje:

- osnovne sintakse programskog jezika C++;
- pisanja klasa;

- osnovnih principa rada sa dinamičkim strukturama podataka.

3.2 Rešavanje zadatka

Rešavanje ćemo započeti pisanjem primera upotrebe klase `Niz`. Na osnovu tog primera ćemo definisati interfejs klase i zatim pristupiti njenom implementiranju. Na kraju ćemo na osnovu definisane klase napraviti šablon koji omogućava pravljenje nizova elemenata proizvoljnog tipa.

Korak 1 - Oblikovanje interfejsa klase Niz

Oblikovanje interfejsa klase `Niz` započećemo pisanjem jednog primera upotrebe. Primer ćemo napisati tako da obuhvati osnovne zahtevane primene dinamičkog niza, a pre svega pristupanje elementima i dinamičku promenu veličine. Radi jednostavnosti, pretpostavićemo da se radi o nizu celih brojeva.

Primer je moguće napisati na mnogo različitih načina. Pokušaćemo da pretpostavimo jednostavnu i intuitivnu sintaksu.

```
//---------------------------------------------------------------
// Primeri upotrebe klase Niz.
//---------------------------------------------------------------
void primer1()
{
    //  Pravimo jedan prazan niz
    Niz a;
    //  Dodajemo 5 elemenata na kraj niza
    for( int i=0; i<5; i++ )
        a.DodajNaKraj( i );
    //  Smanjujemo niz sa 5 na 3 elementa
    a.PromeniVelicinu(3);
    //  Ipisujemo elemente unazad, prazneci niz
    while( a.Velicina() ){
        cout << a.Poslednji() << endl;
        a.ObrisiPoslednji();
        }
}

void primer2()
{
    //  Pravimo niz koji inicijalno ima 5 elemenata
    Niz b(5);
    //  Punimo niz brojevima 0-9,
    //  automatski menjajuci velicinu niza
    for( int i=0; i<10; i++ )
        b[i] = i;
    //  Ispisujemo elemente niza
    for( int i=0; i<b.Velicina(); i++ )
        cout << b[i] << endl;
    //  Povecavamo niz
    b.PromeniVelicinu(20);
    //  Povecavamo niz
    b[99] = 99;
    cout << "Poslednji element je b["
         << (b.Velicina()-1) << "] = "
         << b.Poslednji() << endl;
}

//---------------------------------------------------------------
// Glavna funkcija programa demonstrira upotrebu klase Niz.
//---------------------------------------------------------------
main()
{
    primer1();
    primer2();

    cin.get();
    return 0;
}
```

U prethodnom primeru upotrebe pojavljuje se nekoliko metoda čija bi namena i način upotrebe trebalo da su očigledni. Na primeru niza `a` pokazujemo kako bi trebalo da se dinamički menja veličina niza dodavanjem elemenata na kraj i uklanjanjem elemenata sa kraja niza. Na primeru niza `b` predstavljena je upotreba operatora za indeksni pristup elementima,

koji automatski povećava niz ako nije dovoljno dugačak da se u njega može upisati željeni element.

Definišimo interfejs klase `Niz` neophodan da bi primer mogao da se prevede:

```
class Niz
{
public:
    //  Konstruktor. Pravi niz date velicine.
    Niz( int velicina = 0 );
    //  Indeksni operator.
    int& operator[] ( int i );

    //  Dodavanje novog elementa na kraj niza.
    void DodajNaKraj( int x );
    //  Izracunavanje poslednjeg elementa niza.
    int Poslednji() const;
    //  Brisanje poslednjeg elementa niza.
    void ObrisiPoslednji();

    //  Povecavanje i smanjivanje niza.
    void PromeniVelicinu( int velicina );
    //  Izracunavanje velicine niza.
    int Velicina() const;
};
```

Svi metodi su deklarisani u potpunosti na osnovu prethodnog primera. Metodi `Velicina` i `Poslednji` su deklarisani kao konstantni jer ne menjaju niz. Operatoru za pristupanje elementima po indeksu će više pažnje biti posvećeno prilikom implementiranja. Ovde ćemo samo napomenuti da on mora da vrati referencu na odgovarajući element niza, da bi se vrednost odgovarajućeg elementa mogla menjati, npr:

```
b[99] = 99;
```

Definisani interfejs će nešto kasnije pretrpeti neophodne izmene, ali funkcije i tipovi navedenih metoda neće biti menjani, sa izuzetkom operatora indeksiranja.

Korak 2 - Uprošćena implementacija

Implementiranje klase `Niz` će obuhvatiti nekoliko složenih postupaka. Da bismo ih međusobno razdvojili, prvo ćemo pretpostaviti da se elementi niza automatski obezbeđuju u nekom unapred definisanom, a za naš primer dovoljnom broju – definisaćemo da svaki niz ima obezbeđen prostor za 200 elemenata. Naravno, takva implementacija nam služi samo da bismo mogli što pre isprobati klasu i neke jednostavnije metode. Da bismo ipak mogli govoriti o različitim veličinama niza, uvešćemo član podatak `_Velicina` čija vrednost predstavlja *logičku* veličinu niza, tj. veličinu za koju zna korisnik niza:

```
class Niz
{...
private:
    //  Clanovi podaci.
    int _Elementi[200];
    int _Velicina;
};
```

Sada možemo definisati konstruktor i metode za izračunavanje i menjanje veličine niza. Pri tome ćemo, za sada, zanemariti eventualnu upotrebu tih metoda sa neispravnim argumentima:

```
class Niz
{
public:
    //  Konstruktor. Pravi niz date velicine.
    Niz( int velicina = 0 )
        : _Velicina(velicina)
        {}

    //  Povecavanje i smanjivanje niza.
    void PromeniVelicinu( int velicina )
        { _Velicina = velicina; }

    //  Izracunavanje velicine niza.
    int Velicina() const
        { return _Velicina; }
...
};
```

Možemo definisati i metode za dodavanje novog elementa na kraj niza, izračunavanje vrednosti poslednjeg elementa i brisanje poslednjeg elementa niza. Ponovo zanemarujemo eventualno neispravne argumente, što uključuje i izračunavanje poslednjeg elementa ili brisanje poslednjeg elementa praznog niza:

```
class Niz
{
public:
...
    //  Dodavanje novog elementa na kraj niza.
    void DodajNaKraj( int x )
        { _Elementi[ _Velicina++ ] = x; }

    //  Izracunavanje poslednjeg elementa niza.
    int Poslednji() const
        { return _Elementi[ _Velicina - 1 ]; }

    //  Brisanje poslednjeg elementa niza.
    void ObrisiPoslednji()
        { _Velicina--; }
...
};
```

Preostaje nam još da implementiramo operator indeksiranja. Pristupanje elementima po indeksu je sasvim jednostavno (naravno, uz pretpostavku da ponovo zanemarujemo neispravne argumente), ali ne smemo zaboraviti da primena ovog metoda može da povećava niz, ukoliko se navede indeks koji je veći od indeksa poslednjeg elementa niza:

```
    //  Indeksni operator.
    int& operator[] ( int i )
        {
        if( i >= _Velicina )
            _Velicina = i+1;
```

```
        return _Elementi[i];
        }
```

Uz sve slabosti koje naša implementacija ima, ona je dovoljna da bi proradio primer koji smo napisali na početku.

Korak 3 - Promena veličine niza

Osnovno ograničenje predstavljene implementacije klase `Niz` je što, iako se iz njene upotrebe u našem primeru to ne vidi, ona zapravo ne predstavlja dinamički niz. Prvi problem je što se ne mogu koristiti nizovi čija dužina je iznad pretpostavljene granice od, u našem slučaju, 200 elemenata. Drugi je što se i u slučaju upotrebe kraćih nizova, uvek obezbeđuje prostor za svih 200 elemenata.

U ovom koraku ćemo izmeniti implementaciju klase `Niz` tako da se veličina obezbeđenog prostora zaista menja u skladu sa potrebnom veličinom. Prvi korak u tom smeru je da se u svim ostalim metodima eksplicitno menjanje podatka `_Velicina` zameni pozivom metoda `PromeniVelicinu`. Iako nije neophodno, dobro je i čitanje podatka `_Velicina` zameniti upotrebom metoda `Velicina`:

```
class Niz
{
public:
    //  Konstruktor. Pravi niz date velicine.
    Niz( int velicina = 0 )
        : _Velicina(0)
        { PromeniVelicinu( velicina ); }

    //  Indeksni operator.
    int& operator[] ( int i )
        {
        if( i >= Velicina() )
            PromeniVelicinu( i+1 );
        return _Elementi[i];
        }

    //  Dodavanje novog elementa na kraj niza.
    void DodajNaKraj( int x )
        {
        PromeniVelicinu( Velicina() + 1 );
        _Elementi[ Velicina() - 1 ] = x;
        }

    //  Izracunavanje poslednjeg elementa niza.
    int Poslednji() const
        { return _Elementi[ Velicina() - 1 ]; }

    //  Brisanje poslednjeg elementa niza.
    void ObrisiPoslednji()
        { PromeniVelicinu( Velicina() - 1 ); }
...
};
```

Eksplicitna promena veličine je poverena samo metodu `PromeniVelicinu`. Da bismo ga mogli implementirati moramo promeniti i način čuvanja elemenata. Umesto automatskog

niza elemenata, uvešćemo dinamički alociran niz na koji pokazuje pokazivač `_Elementi`. Konstruktor mora da inicijalizuje ovaj pokazivač:

```
class Niz
{
public:
    //  Konstruktor. Pravi niz date velicine.
    Niz( int velicina = 0 )
        : _Velicina(0),
          _Elementi(0)
        { PromeniVelicinu( velicina ); }
...
private:
    //  Clanovi podaci.
    int*    _Elementi;
...
};
```

Metod `PromeniVelicinu` je dužan da u slučaju povećavanja niza obezbedi nov prostor dovoljno veliki da primi sve elemente niza, kao i da u njega iskopira postojeće elemente niza:

```
    //  Povecavanje i smanjivanje niza.
    void PromeniVelicinu( int velicina )
        {
        if( velicina > _Velicina ){
            int* noviElementi = new int[velicina];
            if( _Velicina )
                memcpy( noviElementi, _Elementi,
                        _Velicina * sizeof(int) );
            _Elementi = noviElementi;
            }
        _Velicina = velicina;
        }
```

Alokacija niza elemenata obavlja se primenom izraza `new[]`, koji je sličan običnom izrazu `new`. Specifičnost izraza `new[]` je u tome što on obezbeđuje prostor za dati broj objekata datog tipa i sve ih inicijalizuje primenom konstruktora bez argumenata. Očigledna posledica takvog ponašanja je da se na ovaj način ne mogu alocirati nizovi objekata klase koja nema konstruktor bez argumenata.

Korak 4 - Oslobađanje nepotrebnog prostora

Iako možemo da prevedemo program i primetimo da primer ispravno radi, postoji nekoliko veoma važnih problema. Najvažniji problem je vezan za neispravno ponašanje naše klase u odnosu na obezbeđene elemente. Dok metod `PromeniVelicinu` obezbeđuje novi (veći) prostor za elemente, mi nigde nismo napisali kako se izvodi oslobađanje tog obezbeđenog prostora. Nedostaje nam kako oslobađanje niza elemenata u trenutku uklanjanja objekta naše klase `Niz`, tako i oslobađanje starog niza elemenata pri obezbeđivanju novog. Prvi problem rešavamo pisanjem destruktora, a drugi popravljanjem metoda `PromeniVelicinu`:

```
class Niz
{
public:
...
    //  Destruktor.
    ~Niz()
        { delete [] _Elementi; }

    //  Povecavanje i smanjivanje niza.
    void PromeniVelicinu( int velicina )
        {
        if( velicina > _Velicina ){
            int* noviElementi = new int[velicina];
            if( _Velicina )
                memcpy( noviElementi, _Elementi,
                        _Velicina * sizeof(int) );
            delete [] _Elementi;
            _Elementi = noviElementi;
            }
        _Velicina = velicina;
        }
...
};
```

Niz se oslobađa primenom izraza `delete[]`, a ne izraza `delete` kao u ranijim primerima. Najvažnije sličnosti i razlike između ovih izraza su:

- izraz `delete[]` pre oslobađanja prostora poziva destruktore svih objekta koji se nalaze u nizu na koji pokazivač pokazuje;
- izraz `delete` pre oslobađanja prostora poziva destruktor objekta na koji pokazivač pokazuje;
- ako se na pokazivač na niz objekata primeni izraz `delete`, umesto izraza `delete[]`, destruktor će se izvršiti samo za prvi element niza (u zavisnosti od načina implementacije ovog izraza i prevodioca može doći i do nepredviđenih problema u trenutku izvršavanja);
- oba operatora oslobađaju prostor koji niz neposredno zauzima, bez posebnih razlika[6];

Pažljivi čitaoci su verovatno već zaključili da u slučaju niza celih brojeva ova dva operatora imaju potpuno isto ponašanje. Zaista, zbog toga što celi brojevi nemaju destruktor (možemo reći i da imaju prazan destruktor, koji ništa ne radi) u ovoj situaciji nije važno da li ćemo pozivati destruktore ili ne. Ako bismo za oslobađanje nizova celih brojeva upotrebljavali običan izraz `delete`, najverovatnije je da ne bismo imali nikakvih problema. Ipak, u zavisnosti od implementacije može biti izvesnih razlika. Zbog toga, a i zbog sticanja navike upotrebe izraza `delete[]`, uvek kada se radi sa nizovima objekata potrebno je upotrebljavati ovaj izraz.

[6] Primetimo da u nekim od prvih verzija programskog jezika i odgovarajućih prevodilaca niz nije mogao da se ispravno oslobodi operatorom `delete`, kao i da je bilo potrebno eksplicitno navoditi veličinu niza koji se oslobađa.

Veoma pažljivi čitaoci su verovatno primetili i jednu (posrednu) nedoslednost autora u odnosu na ovo pitanje, ali na nju će biti ukazano malo kasnije.

Postojanje destruktora nam ukazuje da bi trebalo napisati i konstruktor kopije i operator dodeljivanja. Ponovićemo napomenu iz prethodnih primera: *destruktor, konstruktor kopije i operator dodeljivanja uvek idu zajedno – ili pišemo sve ove metode, ili ne pišemo nijedan*. Napisaćemo pomoćni metod `init` i neophodne metode poštujući ranije opisan šablon:

```
class Niz
{
public:
...
    //  Konstruktor kopije.
    Niz( const Niz& n )
        { init( n ); }

    //  Operator dodeljivanja.
    const Niz& operator = ( const Niz& n )
        {
        if( this != &n ){
            delete [] _Elementi;
            init( n );
            }
        return *this;
        }
...
private:
    //  Inicijalizacija kopije.
    void init( const Niz& n )
        {
        _Velicina = n._Velicina;
        if( _Velicina ){
            _Elementi = new int[_Velicina];
            memcpy( _Elementi, n._Elementi,
                    _Velicina * sizeof(int) );
            }
        else
            _Elementi = 0;
        }
...
};
```

Korak 5 - Pristupanje elementima niza

Klasa `Niz` sadrži definiciju operatora indeksiranja, pomoću koga se može pristupati elementima niza. Podsetimo se njegove implementacije:

```
class Niz
{
public:
...
    //  Indeksni operator.
    int& operator[] ( int i )
        {
```

```
            if( i >= Velicina() )
                PromeniVelicinu( i+1 );
            return _Elementi[i];
            }
...
};
```

Primetimo jedno važno ograničenje predstavljenog interfejsa – operator indeksiranja nije konstantan, pa se ne može primenjivati u slučaju konstantnih nizova. Šta bi se dogodilo ako bismo ga proglasili konstantnim? Tada bi izvesno mogao da se primenjuje na konstantne nizove, ali to ne smemo olako učiniti, jer bismo omogućili da se primenom ovog operatora menja sadržaj „konstantnog" niza:

```
... const Niz& a ...
a[2] = 5;
```

Time što bismo omogućili da operator bude deklarisan kao konstantan, a da istovremeno vraća referencu na element niza, potencijalno bismo učinili veliki propust, jer bismo omogućili da korisnik operatora menja sadržaj konstantnog niza. Iako se u telu operatora ne menjaju eksplicitno elementi niza, posredno se omogućava njihovo menjanje time što se vraća referenca na elemente interne strukture objekta. Da ne bismo dolazili u slične neugodne situacije potrebno je da se pridržavamo veoma važnog pravila: *Konstantni metodi ne smeju vraćati nekonstantne pokazivače (ili reference) koji se odnose na internu strukturu objekata.*

Rešenje je da tip rezultata konstantnog operatora ne bude `int&` nego `const int&` ili samo `int`:

```
const int& operator [] ( int ) const;
//ili
int operator [] ( int ) const;
```

Na žalost, time problem još nije rešen. Primenom ovakvog operatora čak ni u slučaju nizova koji nisu konstantni neće biti moguće menjanje elemenata! Znači, u prvom slučaju (nekonstantan operator) ne možemo pristupati elementima konstantnih nizova, a u drugom (konstantan operator) uopšte ne možemo menjati elemente nizova. Srećom, nije neophodno da mi pravimo izbor, već ga možemo prepustiti prevodiocu. Znajući da programski jezik C++ dopušta definisanje više metoda (i funkcija i operatora) sa istim imenom, sve dok se međusobno razlikuju po broju ili tipu argumenata, možemo odlučivanje o tome koji će se od ova dva metoda upotrebljavati prepustiti prevodiocu. Veoma je važno da uočimo da ova dva operatora nemaju isti tip argumenata: u prvom slučaju tip objekta na koji se operator primenjuje je `Niz` (tip pokazivača `this` je `Niz*`), a u drugom `const Niz` (tip pokazivača `this` je `const Niz*`). Ako navedemo (i kasnije definišemo) oba ova operatora, prevodilac će prema kontekstu u kome ih upotrebljavamo, a zavisno od toga da li je konkretan niz koji koristimo konstantan ili ne, sam birati koji od ovih operatora će biti upotrebljen.

Pri pisanju konstantnog operatora ne smemo menjati veličinu niza. Zbog toga, kao i zbog činjenice da još nismo upoznali mehanizme za postupanje u neispravnim situacijama, opseg indeksa uopšte nećemo kontrolisati, već ćemo pretpostavljati da je ispravan:

```
class Niz
{
public:
...
    //  Indeksni operator.
    int& operator[] ( int i )
        {
        if( i >= Velicina() )
            PromeniVelicinu( i+1 );
        return _Elementi[i];
        }

    //  Indeksni operator. Konstantna verzija
    const int& operator[] ( int i ) const
        { return _Elementi[i]; }
...
};
```

Primetimo da slično ponašanje nije uvek ograničeno samo na operator indeksiranja, ali i da se pisanje ovog operatora skoro uvek izvodi na sličan način – pisanjem dva metoda.

Korak 6 - Optimizacija promene veličine niza

Napišimo pomoćnu funkciju `primer3` za proveru ispravnosti ponašanja klase `Niz`:

```
void primer3()
{
    Niz c(1);
    for( int i=0; i<100; i++ ){
        for( int j=0; j<1000; j++ )
            c.DodajNaKraj(j);
        cout << '.';
        }
    cout << endl;
}

main()
{
    primer1();
    primer2();
    primer3();

    return 0;
}
```

Dodajemo nizu `c` 1000 elemenata i zatim ispisujemo tačku, pa tako 100 puta. Prevedimo primer i pokrenimo program. Šta primećujemo?

Dok se na početku dodavanje elemenata izvodi relativno brzo, kasnije postaje sve sporije. Što je niz duži, to se dodavanje novih elemenata više usporava. U čemu je problem?

U ciklusu se neposredno poziva samo metod `DodajNaKraj`, ali se posredno upotrebljavaju i `operator[]` i metod `PromeniVelicinu`. I metod `DodajNaKraj` i `operator[]` imaju konstantnu složenost, pa ne bi trebalo da se usporavaju sa povećavanjem niza. Sa druge strane, u metodu `PromeniVelicinu` ima nekoliko koraka čija dužina izvršavanja značajno zavisi od veličine niza:

```
void PromeniVelicinu( int velicina )
    {
    if( velicina > _Velicina ){
        int* noviElementi = new int[velicina];
        if( _Velicina )
            memcpy( noviElementi, _Elementi,
                    _Velicina * sizeof(int) );
        delete [] _Elementi;
        _Elementi = noviElementi;
        }
    _Velicina = velicina;
    }
```

Efikasnost alociranja i oslobađanja memorije svakako zavisi od veličine, ali zavisnost nije jednostavno opisati. Zbog toga se time nećemo detaljno baviti, već ćemo intuitivno proceniti da je složenost alociranja i oslobađanja najviše linearno zavisna od dužine niza.

Sa druge strane, složenost kopiranja sadržaja niza iz starog prostora za elemente u novi prostor za elemente očigledno linearno zavisi od dužine niza, jer je kopiranje niza od `k*n` elemenata tačno `k` puta veći (i `k` puta sporiji) posao nego kopiranje niza od `n` elemenata. Neposredno zaključujemo da je dodavanje elementa na kraj niza koji ima `k*n` elemenata, `k` puta sporije od dodavanja elementa na kraj niza koji ima `n` elemenata. U kontekstu prethodnog primera, ispada da je ukupno vreme punjenja niza proporcionalno kvadratu dužine niza, što je prilično neefikasno.

Da bismo rešili problem moramo otkloniti najsporiji segment postupka. U našem slučaju, ako želimo dodavati elemente na kraj niza, najsporiji deo tog dodavanja je promena veličine niza. Ako veličinu niza moramo promeniti, šta možemo učiniti da je bar ne menjamo u svakom koraku? Možda da menjamo veličinu niza u većim koracima, tako da se smanji učestalost promene veličine? Tako bi opet dolazilo do sporog povećavanja veličine, ali bi se verovatno postigla veća ukupna efikasnost koda.

Predložen koncept možemo veoma lako isprobati bez mnogo programiranja. Dovoljno je da izmenimo funkciju `primer3`, tako da se:

- umesto dodavanja elemenata upotrebljava `operator[]`;
- na početku svakog unutrašnjeg ciklusa niz odmah poveća za dovoljno mesta da primi sve elemente u tom ciklusu.

Da bismo otklonili sumnju u to koji je korak eventualno povećao efikasnost, primenimo najpre prvi korak:

```
void primer3()
{
    Niz c(1);
    for( int i=0; i<100; i++ ){
        for( int j=0; j<1000; j++ )
            c[i*1000+j] = j;
        cout << '.';
        }
    cout << endl;
}
```

Primetimo da se na ovaj način niz povećava i popunjava na potpuno isti način kao ranije. Kao što se i moglo očekivati, efikasnost programa nije ni najmanje promenjena. Sada ćemo uvesti i drugu promenu:

```
void primer3()
{
    Niz c(1);
    for( int i=0; i<100; i++ ){
        c.PromeniVelicinu( c.Velicina() + 1000 );
        for( int j=0; j<1000; j++ )
            c[i*1000+j] = j;
        cout << '.';
        }
    cout << endl;
}
```

Prevođenje i izvršavanje ovako izmenjenog programa pokazuje da program radi daleko brže. Da bismo videli koliko je to zaista brzo, pokušajmo da povećamo veličinu niza:

```
    for( int i=0; i<10000; i++ ){
        ...
        }
```

Kao što smo već pri isprobavanju prethodnog primera videli, ovo zaista radi vrlo brzo. Međutim, možemo primetiti da ipak dolazi do usporavanja kada veličina niza značajno poraste. Doduše, ako je ranije do vidljivog usporavanja dolazilo već pri veličinama od nekoliko hiljada elemenata, sada usporavanje uočavamo tek pri veličinama od blizu milion elemenata. Pokazuje se da smo povećavanjem niza u koracima od po hiljadu elemenata, umesto za po jedan element, postigli značajno povećavanje efikasnosti rada sa nizom, ali efikasnost i dalje značajno pada sa porastom veličine niza.

Pokušajmo da prebrojimo koliko se ukupno elemenata niza premesti od trenutka pravljenja praznog niza do trenutka kada niz u našem primeru poraste do n elemenata. U prvom slučaju, kada se veličina menja pri dodavanju svakog elementa, broj elemenata koji se premeštaju pri dodavanju k-tog elementa je:

$$s_k = k-1$$

a ukupan broj premeštenih elemenata je:

$$\begin{aligned} P_n &= s_1 + s_2 + s_3 + \ldots + s_n \\ &= 0 + 1 + 2 + \ldots + (n-1) \\ &= n*(n-1)/2 \\ &= (n^2-n)/2 \end{aligned}$$

U drugom slučaju imamo:

$$\begin{aligned} s_k &= k-1, \; k = m*1000+1 \\ s_k &= 0 \;\; , \; k \neq m*1000+1 \end{aligned}$$

pa je ukupan broj premeštenih elemenata:

$$\begin{aligned} P_n &= s_1 + s_2 + s_3 + \ldots + s_n \\ &= s_1 + s_{1001} + s_{2001} + \ldots + s_{[(n-1)/1000]*1000+1} \end{aligned}$$

$= 0 + 1000 + 2000 + ... + [(n-1)/1000]*1000$
$= 1000 * (0 + 1 + 2 + ... + [n-1/1000])$
$\leq (n^2/1000-n)/2$

Vidimo da je ukupan broj premeštenih elemenata daleko manji, ali da je i dalje proporcionalan kvadratu dodatih elemenata. Šta možemo učiniti da bi se efikasnost povećala?

Detaljnija analiza bi potvrdila da svako povećavanje niza u konstantnim koracima, koliki god da su koraci, ima slične posledice. Jedini način da se efikasnost značajno poveća jeste da se veličina koraka menja sa veličinom niza. Može se pokazati da se povećavanjem niza u koracima čija je veličina linearno proporcionalna dužini niza, postiže da ukupan broj premeštanja bude linearno proporcionalan veličini niza.

Neka je korak povećavanja niza jednak dužini niza, a najmanje 10. Tada važi:

$s_1 = 0$, niz dobija veličinu 10
$s_{11} = 10$, niz dobija veličinu 20
$s_{21} = 20$, niz dobija veličinu 40
$s_{41} = 40$, niz dobija veličinu 80
...
$s_k = k-1$, $k = 10*2^m+1$, niz dobija veličinu $10*2^{m+1}$
$s_k = 0$, $k \neq 10*2^m+1$

a ukupan broj premeštenih elemenata se izračunava kao:

$P_n = s_1 + s_2 + s_3 + ... + s_n$
$= s_1 + s_{11} + s_{21} + s_{41} + ... + s_a$, $a=1+10*2^{[\log_2((n-1)/10)]}$
$= 0 + 10 + 20 + 40 + ... + 10*2^{[\log_2((n-1)/10)]}$
$= 10 * (0 + 1 + 2 + 4 + 2^{[\log_2((n-1)/10)]})$
$= 10 * (2^{[\log_2((n-1)/10)} + 1] - 1)$
$= 10 * (2 * 2^{[\log_2((n-1)/10)]} - 1)$
$= 20 * 2^{[\log_2((n-1)/10)]} - 10$

gde je $2^{[\log_2((n-1)/10)]}$ najveći pravi stepen broja 2 koji je manji od $n/10$. Tada važi:

$n-10 \leq P_n \leq 2n-12$

Znači, ako bi se veličina niza menjala na opisan način, broj kopiranja elemenata bi bio najviše $2n-12$, što predstavlja linearnu zavisnost broja kopiranja od dužine niza. To znači da se u našem primeru ne bi osetilo usporenje.

Pokušajmo da nešto slično implementiramo u našem primeru. Potrebno je da:

- odredimo da niz ima početnu veličinu od 10 elemenata (može se slobodno izabrati bilo koja konstanta veća od 0);
- svaki put pre unutrašnjeg ciklusa proveravamo da li ima dovoljno elemenata niza, pa ako nema onda udvostručavamo niz sve dok ne postane dovoljno dugačak.

Otprilike ovako:

```
void primer3()
{
    Niz c(10);
```

```
        for( int i=0; i<10000; i++ ){
            int potrebno = (i+1)*1000;
            while( potrebno > c.Velicina() )
                c.PromeniVelicinu( c.Velicina() * 2 );
            for( int j=0; j<1000; j++ )
                c[i*1000+j] = j;
            cout << '.';
            }
        cout << endl;
    }
```

Prevođenje i izvršavanje zaista pokazuje da program sada radi jednako brzo bez obzira na veličinu niza. Naravno, kada veličina niza postane bliska raspoloživoj radnoj memoriji računara, moraće da dođe do usporavanja usled toga što operativni sistem primenjuje odgovarajući algoritam da bi zone memorije koje se trenutno ne koriste prebacio na disk i oslobodio ih za upotrebu, ali to je već nešto na šta ne možemo jednostavno uticati.

Činjenica da uvek alociramo više prostora nego što nam je potrebno, navodi na zaključak da značajna količina memorije tako ostaje neupotrebljena. Nameće se pitanje kolika je prosečna iskorišćenost ovako rezervisanog prostora? Nećemo ulaziti u dublje razmatranje, ali se već površnom analizom dolazi do zaključaka koji su dovoljno precizni.

Neka je za neki niz alocirano dovoljno prostora za `n` elemenata. Ako je `n` dovoljno veliko (u našem primeru veće od 10), to znači da je niz ili inicijalno napravljen u toj veličini, ili u nekom prethodnom trenutku nije bilo dovoljno `n/2` elemenata, koliko je bilo prethodno obezbeđeno. Primetimo da je u prvom slučaju popunjenost niza verovatno upravo `n`, ali takvi slučajevi nas ne zanimaju jer je kod njih ista popunjenost i sa bilo kojim drugim algoritmom povećavanja nizova. Zato nadalje razmatramo samo drugi slučaj, tj. nizove koji rastu tokom vremena. Kako u drugom slučaju `n/2` nije bilo dovoljno, a `n` još uvek jeste, možemo pretpostaviti da je popunjenost niza negde između `n/2` i `n`. Ako pretpostavimo da je svaka veličinu niza jednako verovatna, lako se dolazi do zaključka da je prosečna popunjenost ovakvog niza upravo `3n/4`. U praksi je popunjenost nizova često malo manja, ali je procena popunjenosti od 75% ipak prilično dobra.

Ako, zbog manjka radne memorije, ili iz nekih drugih razloga, ne može da se dopusti luksuz da 25% veličine niza bude neupotrebljeno, niz se može povećavati i u manjim koracima. Na primer, ako se veličina niza uvećava za po 25%, tada je prosečna iskorišćenost 90%, a ako se uvećava za po 11.11%, prosečna iskorišćenost je oko 95%. Pri tome se povećava i broj premeštanja elemenata niza, ali on i dalje ostaje linearno zavisan od dužine niza.

Implementacija efikasnijeg povećavanja niza

Pre nego što pristupimo ugradnji odgovarajućeg algoritma povećavanja niza u klasu `Niz`, potrebno je da razmotrimo kakve implikacije ima takvo ponašanje na funkcionalnost niza. Na početku eksperimentisanja zamenili smo upotrebu metoda `DodajNaKraj` upotrebom indeksnog operatora. Zašto? Kada elemente dodajemo na kraj niza metodom `DodajNaKraj`, mi očekujemo da posle `n` dodavanja niz poraste za tačno `n` elemenata, a to neće biti zadovoljeno ako se veličina niza menja u potencijalno većim koracima. Na osnovu toga možemo zaključiti da povećavanje niza u koracima nije u skladu sa dodavanjem elemenata na

kraj niza i sličnim operacijama. Ali problem nije u tome što koristimo podatak o veličini niza pri dodavanju elemenata na kraj niza, već u činjenici da jedan isti podatak `_Velicina` upotrebljavamo kako za označavanje logičke dužine niza, tako i za označavanje veličine fizički obezbeđenog prostora. Činjenica da niz povećavamo u koracima, ne bi smela ni na koji način uticati na logički, tj. upotrebni koncept pojma veličine niza. Za korisnika, veličina niza nije broj alociranih elemenata niza, već broj elemenata niza koji se upotrebljavaju, pa ta dva broja nikako ne bismo smeli mešati.

Rešenje je sasvim jednostavno – uvešćemo novi član podatak `_Alocirano`, čija vrednost odgovara broju fizički alociranih elemenata. Veličina niza može da raste najviše do tog broja elemenata a da se pri tome ne mora alocirati novi prostor.

Menjamo klasu `Niz`. Navodimo samo izmenjene metode:

```
class Niz
{
public:
    //  Konstruktor. Pravi niz date velicine.
    Niz( int velicina = 0 )
        : _Velicina(0),
          _Alocirano(0),
          _Elementi(0)
        { PromeniVelicinu( velicina ); }
...
    //  Povecavanje i smanjivanje niza.
    void PromeniVelicinu( int velicina )
        {
        if( velicina > _Alocirano ){
            if( _Alocirano < 5 )
                _Alocirano = 5;
            do  { _Alocirano *= 2; }
                while( velicina > _Alocirano );
            int* noviElementi = new int[_Alocirano];
            if( _Velicina )
                memcpy( noviElementi, _Elementi,
                        _Velicina * sizeof(int) );
            delete [] _Elementi;
            _Elementi = noviElementi;
            }
        _Velicina = velicina;
        }
...
private:
    //  Inicijalizacija kopije.
    void init( const Niz& n )
        {
        _Alocirano = _Velicina = n._Velicina;
        if( _Velicina ){
            _Elementi = new int[_Velicina];
            memcpy( _Elementi, n._Elementi,
                    _Velicina * sizeof(int) );
            }
```

```
            else
                _Elementi = 0;
            }

        //  Clanovi podaci.
        int*    _Elementi;
        int     _Velicina;
        int     _Alocirano;
};
```

Sada možemo funkciju `primer3` da vratimo u prvobitno stanje, ali sa većim granicama brojača:

```
void primer3()
{
    Niz c(0);
    for( int i=0; i<10000; i++ ){
        for( int j=0; j<1000; j++ )
            c.DodajNaKraj(j);
        cout << '.';
        }
    cout << endl;
}
```

Prevođenje i izvršavanje pokazuje da se dužina niza efikasno povećava.

Smanjivanje niza

Do sada smo skoro u potpunosti zanemarili smanjivanje niza. Naime, kao što se veličina niza može povećati, kada nam zatreba da u niz upišemo neke nove elemente, tako se može i smanjiti, kada shvatimo da nam neki elementi više nisu potrebni. U slučaju da je nova veličina manja od stare, metod `PromeniVelicinu` samo menja podatak o veličini niza, ne oslobađajući pri tome suvišan memorijski prostor. Nije sasvim jasno ni da li će korisniku niza oslobađanje suvišnog prostora biti od koristi ili će samo nepotrebno umanjivati efikasnost. Problem je u tome što nakon smanjivanja veličine niza vrlo često ponovo dolazi do povećavanja, pa se tada bez potrebe gubi vreme radi smanjivanja i naknadnog povećavanja. Ipak, smanjivanje niza se pokazuje korisno ako:

- nakon smanjivanja ne sledi ponovljeno i vrlo skoro povećavanje niza;
- sistemu nedostaje radne memorije, pa čak i privremeno oslobađanje memorije doprinosi ukupnoj efikasnosti sistema;

Iako je korist od smanjivanja niza prilično diskutabilna, ovde ćemo ga implementirati radi ilustracije. Niz ćemo smanjivati ako mu se veličina smanji na manje od 1/4 alociranog prostora. Kriterijum bi mogao biti i drugačiji, ali bi bilo dobro da bude značajno manji od koraka povećavanja:

```
class Niz
{
public:
...
```

```
        //  Povecavanje i smanjivanje niza.
        void PromeniVelicinu( int velicina )
            {
            if( velicina > _Alocirano ){
                ...
                }
            else if( velicina < _Velicina
                     && _Alocirano > 10
                     && velicina < _Alocirano/4
                    )
                {
                _Alocirano = velicina;
                int* noviElementi = 0;
                if( velicina ){
                    noviElementi = new int[_Alocirano];
                    memcpy( noviElementi, _Elementi,
                            velicina * sizeof(int) );
                    }
                delete [] _Elementi;
                _Elementi = noviElementi;
                }
            _Velicina = velicina;
            }
    ...
    };
```

Korak 7 - Robusnost

Napišimo novi primer upotrebe klase `Niz`:

```
void primer4( const Niz& a )
{
    int v = a.Velicina();
    cout << "Velicina niza: " <<  v << endl;
    cout << "a[" << v << "] = " << a[v] << endl;
    cout << "a[-1] = " << a[-1] << endl;
}

main()
{
...
    Niz a(5);
    primer4(a);
...
}
```

U funkciji `primer4` se pristupa elementima konstantnog niza van opsega. Ako se program prevede i pokrene, najverovatnije će se dobiti neki nasumični brojevi kao rezultati, ali može se dogoditi i da bude prijavljena neka greška. Ponašanje zavisi kako od prevodioca, tako i od okruženja u kome se program pokreće. Ako bismo pokušali da značajnije promašimo opseg, recimo:

```
        cout << "a[" << (v+1000000) << "] = " << a[v+1000000] << endl;
```

onda bi bilo daleko verovatnije da bi to predstavljalo pokušaj pristupanja oblastima memorije koje nisu dodeljene našem programu, što bi za posledicu imalo prekidanje rada programa i prijavljivanje odgovarajuće greške od strane operativnog sistema.

U svim prethodnim koracima pisanja klase `Niz` je grubo zapostavljeno pitanje robusnosti, odnosno provere ispravnosti argumenata funkcija.

Postizanje robusnosti sužavanjem tipova podataka

Pogledajmo redom metode klase niz i pokušajmo da prepoznamo moguće neispravnosti argumenata. Idemo redom, od konstruktora:

```
Niz( int velicina = 0 )
    : _Velicina(0),
      _Alocirano(0),
      _Elementi(0)
    { PromeniVelicinu( velicina ); }
```

Postoje dve osnovne moguće neispravnosti. Prva je navođenje negativne veličine niza, a druga je eventualno neuspešna promena veličine. Reagovanje na negativnu veličinu niza bi moglo da se izvede na različite načine, ali nije dobro zaletati se. Uočimo da se svako programsko reagovanje odnosi isključivno na greške u fazi izvršavanja, a da je neispravnosti mnogo bolje prepoznati još u fazi prevođenja, ako je to ikako moguće. Osnovno sredstvo za blagovremeno predupređivanje grešaka jesu tipovi podataka. Ispostavlja se da problem negativne veličine niza možemo rešiti na elegantan i kvalitetan način – promenom tipa argumenta konstruktora.

Ako znamo da se naši nizovi proširuju samo na jednu stranu, dodavanjem elemenata na kraj niza, jasno je da negativne veličine i negativni indeksi nemaju smisla. Možemo svim podacima članovima i argumentima tipa `int`, koji se odnose na indekse i veličinu niza, izmeniti tip u `unsigned`:

```
class Niz
{
public:
    //  Konstruktor. Pravi niz date velicine.
    Niz( unsigned velicina = 0 )
        : _Velicina(0),
          _Alocirano(0),
          _Elementi(0)
        { PromeniVelicinu( velicina ); }
...
    //  Indeksni operator.
    int& operator[] ( unsigned i )
        {
        if( i >= Velicina() )
            PromeniVelicinu( i+1 );
        return _Elementi[i];
        }

    //  Indeksni operator. Konstantna verzija
    const int& operator[] ( unsigned i ) const
        { return _Elementi[i]; }
```

```
    //  Povecavanje i smanjivanje niza.
    void PromeniVelicinu( unsigned velicina )
        {
        ...
        }

    //  Izracunavanje velicine niza.
    unsigned Velicina() const
        { return _Velicina; }

private:
    ...
    //  Clanovi podaci.
    int*        _Elementi;
    unsigned    _Velicina;
    unsigned    _Alocirano;
};
```

Sada bi i u primerima valjalo izmeniti tipove. Na primer:

```
void primer2()
{
    ...
    //  Ispisujemo elemente niza
    for( unsigned i=0; i<b.Velicina(); i++ )
        cout << b[i] << endl;
    ...
}
```

Upotrebom neoznačenih celih brojeva za veličine i indekse predupredićemo neke od mogućih neispravnosti. Primetimo da će većina prevodilaca neće prekinuti prevođenje programa u slučaju upotrebe negativnih vrednosti umesto neoznačenih, već će samo izdati odgovarajuće upozorenje.

Neke druge greške će moći da se prepoznaju samo u fazi izvršavanja programa.

Izuzeci u programskom jeziku C++

Ako posmatramo konstantan operator indeksiranja, vidimo da vrednost indeksa može biti neispravna u oba smera. Ako znamo veličinu niza, tada ispravni indeksi moraju biti u intervalu [0, `Velicina()`-1]. Samu proveru nije teško izvesti, ali nije sasvim jasno šta valja činiti kada se prepozna da argument nije ispravan? Da li prekinuti rad programa? Ili, možda, vratiti neku posebnu vrednost? Da li pri tome ispisati i neku poruku?

Svako od navedenih rešenja može biti dobro u jednom kontekstu, a loše u nekom drugom, pa je tim teže odabrati pravu reakciju. Zapravo, idealno bi bilo ako bi odluka o načinu reagovanja mogla da se prepusti korisniku operatora, pa da on sam, a u zavisnosti od uslova upotrebe, odabere najprihvatljiviju reakciju. Programski jezik C++, kao i veći broj drugih savremenih programskih jezika, omogućava fleksibilno obrađivanje vanrednih situacija pomoću podsistema za rad sa *izuzecima*. Sledi kraći osvrt na specifičnosti rada sa izuzecima u programskom jeziku C++.

Rad sa izuzecima se zasniva na veoma jednostavnoj podeli posla:

- kada neki kod za proveru ustanovi neispravnost, dužan je da je prijavi i prekine rad odgovarajućeg bloka koda;
- onaj deo koda koji je sposoban da na neku prijavljenu neispravnost odreaguje, potrebno je da proveri da li je kojim slučajem odgovarajuća neispravnost prijavljena i zatim da na nju valjano odreaguje.

U terminologiji programskog jezika C++ (i većine drugih programskih jezika koji podržavaju rad sa izuzecima), prijavljivanje neispravnosti se naziva *izbacivanjem izuzetka* ili *proglašavanjem izuzetka*, a obrada izbačenog izuzetka se naziva *hvatanjem izuzetka*. U programskom jeziku C++ izuzetak može biti objekat bilo kog tipa, uključujući i osnovne tipove podataka (`int`, `char*`, i drugi) i proizvoljno složene klase. Ipak, ne preporučuje se da se za izuzetke koriste veoma složeni ili veliki objekti, iz jednostavnog razloga što činjenica da je uopšte došlo do izuzetaka ukazuje da stanje programa nije redovno, pa je pitanje i da li će takvi složeni objekti uopšte moći da se ispravno konstruišu.

Za izbacivanje izuzetaka upotrebljava se naredba `throw`. Sintaksa ove naredbe je:

```
throw <izuzetak>;
```

Blok koda u kome se očekuju mogući izuzeci koji se mogu obraditi na odgovarajući način, zapisuje se u obliku *bloka pokušaja*:

```
try {
    <blok koda>
    }
<hvatač izuzetaka>
{<hvatač izuzetaka>}
```

Svaki blok pokušaja praćen je bar jednim, a potencijalno i većim brojem hvatača izuzetaka. Izuzetak koji nastane tokom izvršavanja obuhvaćenog bloka koda biva uhvaćen i obrađen od strane prvog od navedenih hvatača izuzetaka koji može uhvatiti izuzetak odgovarajućeg tipa. Hvatači izuzetaka imaju oblik:

```
catch( <argument> ){
    <kod za obradu izuzetka>
    }
```

gde je argument opisan tipom i imenom. Kod hvatača je dužan da obradi potencijalne izuzetke koji predstavljaju objekte čiji tip odgovara tipu argumenta hvatača. Samom objektu izuzetka se pristupa kao argumentu hvatača.

Univerzalan hvatač je poseban hvatač izuzetaka koji je u stanju da uhvati izuzetke svakog tipa. U okviru tela univerzalnog hvatača se ne može pristupati objektu izuzetka (zato što nije poznat njegov tip). U deklaraciji univerzalnog hvatača se umesto tipa i imena argumenta navode tri tačke:

```
catch(...){
    <kod za obradu izuzetka>
    }
```

Ukoliko se tokom obrade izuzetka zaključi da iz nekog razloga on ipak ne može biti u potpunosti obrađen, može se izbaciti isti izuzetak naredbom `throw` bez argumenata. Takav oblik se može naći isključivo u telu hvatača izuzetaka i naziva se *propuštanje izuzetka*. Možda izgleda čudno, ali pokazaćemo da je takva upotreba veoma važna, posebno u situacijama kada je potrebno oslobađati neke dinamički obezbeđene resurse.

Postupak izbacivanja i hvatanja izuzetka obuhvata sledeće korake:

- pravi se objekat koji identifikuje primećenu neispravnost (u daljem tekstu *izuzetak*);
- napravljeni izuzetak se *izbacuje* naredbom `throw`;
- sve dok se izuzetak ne obradi ponavljaju se sledeći koraci:
  - prekida se izvršavanje tekućeg bloka i na uobičajen način se uništavaju svi objekti koji su automatski alocirani u tom bloku, pri čemu se izvode i destrukcija i dealokacija svakog od njih;
  - postupak se nastavlja zavisno od vrste prekinutog bloka:
    - ako je u pitanju glavni (osnovni blok) tela funkcije (ili metoda), postupak se nastavlja neposredno iza naredbe kojom je funkcija (ili metod) pozvana;
    - ako je u pitanju blok koda koji nije ni glavni blok tela funkcije ni blok pokušaja, postupak se nastavlja neposredno iza tog bloka koda;
  - ako je prekinut blok koda upravo blok pokušaja, tada se proverava da li je on praćen nekim hvatačem koji može da uhvati odgovarajući tip izuzetka:
    - ako postoji jedan ili više takvih hvatača, postupak se nastavlja na početku tela prvog od njih:
      - ako se izvršavanje hvatača uspešno okonča, izuzetak je obrađen
      - inače, ako dođe do novog izbacivanja izuzetka, postupak se ponavlja od početka;
    - ako ne postoji odgovarajući hvatač, onda se postupak nastavlja neposredno iza poslednjeg hvatača u grupi;
- nakon što je izuzetak obrađen, izvršavanje programa se nastavlja neposredno iza poslednjeg hvatača koji odgovara istom bloku pokušaja kao i hvatač koji je obradio izuzetak.

Rezultat ovako definisanog postupka je da će se nakon izbacivanja izuzetka izvršavanje programa nastaviti izvršavanjem hvatača odgovarajućeg tipa izuzetaka koji odgovara najbližem obuhvatajućem bloku pokušaja. Ako takav blok pokušaja i hvatač ne postoje, biće prekinuta i sama funkcija `main`, a obaveštenje o problemu će korisniku predstaviti ili poseban

deo programa koji prevodilac dodaje upravo za slučaj nastupanja takvih situacija, ili sam operativni sistem[7].

Nećemo ulaziti u dublje razmatranje mehanizma izuzetaka u programskom jeziku C++. Pregled klasa izuzetaka u standardnoj biblioteci programskog jezika C++ je naveden u odeljku *10.12 Izuzeci*, na strani 400. Više informacija se može naći u [Lippman 2005].

Izbacivanje izuzetaka u metodima klase Niz

U prethodnim odeljcima je naglašeno da pri izvršavanju konstruktora može doći do neuspešnog određivanja veličine. Jedan od kvaliteta podsistema za rad sa izuzecima je u tome da se ne moramo starati o potencijalnim problemima koje ne možemo obraditi. Ako dođe do problema, tj. do izbacivanja izuzetka pri promeni veličine niza, nema razloga da se on obrađuje u konstruktoru. Jednostavno, taj izuzetak će prekinuti kako izvršavanje metoda `PromeniVelicinu`, tako i izvršavanje konstruktora.

Sada ćemo obratiti pažnju prvenstveno na neispravnosti zbog kojih moramo sami izbaciti izuzetke. Eventualno reagovanje na izbačene izuzetke ostavljamo za kasnije.

Prve potencijalne probleme imamo kod konstantnog operatora indeksiranja. Primetimo da nekonstantan operator više ne može imati neispravne argumente, jer sada argumenti mogu biti samo pozitivni brojevi – a ako je indeks veći od niza, povećaćemo niz. Kako u slučaju konstantnog operatora nema automatskog povećavanja niza, dodaćemo proveru i izbacivanje odgovarajućeg izuzetka. Primetimo da ranija verzija može raditi naizgled bez problema i sa neispravnim indeksima (videti odeljak *Greške pri pristupanju nedodeljenoj memoriji*, na strani 53).

```
//  Indeksni operator. Konstantna verzija
const int& operator[] ( unsigned i ) const
    {
    if( i >= Velicina() )
        throw "Neispravan indeks!";
    return _Elementi[i];
    }
```

U najjednostavnije izuzetke spadaju izuzeci tipa `char*`. U slučaju primećene neispravnosti može se, bez mnogo komplikovanja, izbaciti izuzetak koji predstavlja tekst poruke o prepoznatoj neispravnosti. To izvesno nije idealan način, ali je u kontekstu ovog primera sasvim dovoljan.

Do problema može doći i ako pokušavamo da čitamo ili brišemo poslednji element praznog niza. Zbog toga menjamo i odgovarajuće metode:

[7] Operativni sistem, naravno, ne zna ništa o mehanizmu izuzetaka koji je implementiran u nekom konkretnom programu. Ukoliko program proizvede izuzetak koji nigde nije uhvaćen, a u program nije povezan deo koda koji je nadležan za staranje o takvim situacijama, operativni sistem će prepoznati da je program prekinut na neispravan način i dati odgovarajuću poruku. Reakcija operativnog sistema je pri tome ista kao i u slučaju kada se program prekine na neki drugi neispravan način.

```
//  Izracunavanje poslednjeg elementa niza.
int Poslednji() const
    {
    if( !Velicina() )
        throw "Prazan niz nema poslednji element! ";
    return _Elementi[ Velicina() - 1 ];
    }

//  Brisanje poslednjeg elementa niza.
void ObrisiPoslednji()
    {
    if( !Velicina() )
        throw "Prazan niz nema poslednji element! ";
    PromeniVelicinu( Velicina() - 1 );
    }
```

Ponašanje u okruženju sa izuzecima

U slučaju neuspešnog pravljenja novog objekta izrazom `new`, ili čitavog niza objekata izrazom `new[]`, izračunavanje se prekida izbacivanjem izuzetka tipa `bad_alloc`. Zbog toga se u svakom metodu u kome postoji izraz `new` (ili `new[]`) uvek mora razmotriti šta se dešava sa objektima koji se u metodu koriste, a pre svega sa objektom o čijem se metodu radi, u slučaju izuzetka `bad_alloc`. Na primer, ako postoji nekoliko uzastopnih alokacija, često je u slučaju izuzetka potrebno taj izuzetak uhvatiti i osloboditi prostor obezbeđen uspešnim alokacijama.

Jedna moguća neispravnost bi mogla nastupiti i ako bismo imali, na primer, ovakav kod:

```
delete [] _Elementi;
_Elementi = new int[_Alocirano];
```

Gde je problem? U ovom isečku koda postoji jedna od najčešćih grešaka koje se odnose na robusnost programa – *delimična (de)inicijalizacija*. Brisanjem starog niza elemenata objekat niza je postao neispravan. Eventualni izuzetak pri alociranju novog niza bi doveo do prekida izvršavanja navedenog bloka koda, usled čega bi objekat ostao neispravan. Možda bi neko mogao reći da to nije bitno jer će izuzetak prekinuti upotrebu tog objekta, ali to nije tačno – ne samo što je moguće da će objekat i dalje biti upotrebljavan, nego je skoro sasvim sigurno da će u nekom trenutku biti pozvan njegov destruktor. A destrukcija neispravnog objekta može imati fatalne posledice po izvršavanje programa, jer je niz elemenata već oslobođen! Štaviše, ako upotreba objekta bude prekinuta, do destrukcije će doći brže. Da ironija bude veća, do izvršavanja destruktora neće doći samo ako postoji greška još negde u programu. Da zaključimo, *objekat nikada ne sme ostati u stanju delimično izvedene inicijalizacije (ili deinicijalizacije).*

Pre nego što nastavite čitanje, pokušajte da pronađete neispravnosti u metodu `PromeniVelicinu`. Ima li ih? Ako ih niste pronašli, potražite bolje, jer ih izvesno ima.

U okviru metoda `PromeniVelicinu` promena vrednosti pokazivača `_Elementi` se ostavlja za sam kraj postupka, zbog čega tu ne može biti problema. Neće biti problema ni sa neispravnim vrednostima podatka `_Velicina`, jer se i on menja na samom kraju postupka. Ali do problema može doći u vezi podatka `_Alocirano`, jer se on menja pre same alokacije! U slučaju neuspešne alokacije, objekat klase `Niz` bi „mislio“ da ima alociran prostor za niz jedne dužine, a zapravo bi imao alociran prostor za niz druge dužine. Verovatno najpouzdanije

rešenje za ovakve probleme je da se pre osetljivih delova koda, koji mogu napraviti izuzetke, menjaju neke pomoćne promenljive, a da se vrednosti članova podataka promene tek na kraju postupka, kada je jasno da su potencijalno rizične aktvinosti uspešno okončane:

```
//  Povecavanje i smanjivanje niza.
void PromeniVelicinu( unsigned velicina )
    {
    //  Povecavanje...
    if( velicina > _Alocirano ){
        unsigned alocirano = _Alocirano;
        if( alocirano < 5 )
            alocirano = 5;
        do  { alocirano *= 2; }
            while( velicina > alocirano );
        int* noviElementi = new int[alocirano];
        if( _Velicina )
            memcpy( noviElementi, _Elementi,
                    _Velicina * sizeof(int) );
        delete [] _Elementi;
        _Elementi = noviElementi;
        _Alocirano = alocirano;
        }

    //  Smanjivanje...
    else if( velicina < _Velicina
             && _Alocirano > 10
             && velicina < _Alocirano/4
            )
        {
        unsigned alocirano = velicina;
        int* noviElementi = 0;
        if( velicina ){
            noviElementi = new int[alocirano];
            memcpy( noviElementi, _Elementi,
                    velicina * sizeof(int) );
            }
        delete [] _Elementi;
        _Elementi = noviElementi;
        _Alocirano = alocirano;
        }

    _Velicina = velicina;
    }
```

Sličan problem postoji i u metodu `init`. Metod `init` se koristi u okviru konstruktora kopije i operatora dodeljivanja. Ako dođe do izuzetka pri alokaciji u metodu `init` tokom konstrukcije kopije, neće doći do većih problema jer objekti čija konstrukcija nije uspešno dovršena ne podležu destrukciji. Međutim, ako dođe do izuzetka tokom izvršavanja operatora dodeljivanja, doći će i do odgovarajućih problema, jer je pre pozivanja metoda `init` izvedena samo delimična deinicijalizacija objekta, pa pokazivač `_Elementi` i dalje pokazuje na mesto gde su se elementi nekada nalazili. Zbog toga je potrebno menjati metod `init` ili operator dodeljivanja ili oba. Kako se metod `init` svejedno mora menjati, izvešćemo sve što je potrebno samo menjanjem tog metoda.

Kako je na početku niz zapravo prazan, to ćemo i ozvaničiti menjanjem članova podataka. Tek nakon uspešne alokacije ćemo ažurirati novu stvarnu veličinu i veličinu alociranog prostora:

```
//  Inicijalizacija kopije.
void init( const Niz& n )
    {
    _Alocirano = _Velicina = 0;
    _Elementi = 0;
    if( n._Velicina ){
        _Elementi = new int[n._Velicina];
        _Alocirano = _Velicina = n._Velicina;
        memcpy( _Elementi, n._Elementi,
                n._Velicina * sizeof(int) );
        }
    }
```

Opseg tipova

Skup problema u metodu `PromeniVelicinu` još nije iscrpljen. Preostali problem je još teže uočiti. Obratimo pažnju na postupak izračunavanja nove veličine alociranog prostora:

```
...
unsigned alocirano = _Alocirano;
if( alocirano < 5 )
    alocirano = 5;
do  { alocirano *= 2; }
    while( velicina > alocirano );
...
```

i razmotrimo: da li je moguće da dođe do mrtve petlje? Telo petlje udvostručuje vrednost promenljive `alocirano`, a ponavlja se sve dok je `alocirano` manje od `velicina`. Kako je početna vrednost bar 10, čini se da za bilo koju vrednost podatka `velicina`, promenljiva `alocirano` u nekom trenutku mora dobiti veću vrednost. Takvo zaključivanje je ispravno ukoliko se oslonimo na matematička znanja o beskonačnom skupu celih brojeva.

Međutim, celi brojevi koji predstavljaju sadržaje promenljivih nikako nisu u beskonačnom opsegu. Imajući to u vidu, pretpostavimo da tip `unsigned` ima opseg [0, `maxint`]. Tada je moguće da se u nekom trenutku desi da važi:

```
maxint/2 < alocirano < velicina < maxint
```

Prvi pokušaj udvostručavanja podatka `alocirano` će umesto vrednosti 2*`alocirano`, koja je izvesno van opsega tipa `unsigned`, promenljivoj `alocirano` dodeliti vrednost 2*`alocirano`-`maxint`. Da sve bude „zanimljivije“, ako znamo da je obično `maxint`=2^n-1, gde je `n` veličina tipa `unsigned` u bitovima, onda možemo zaključiti da će nakon najviše `n` uzastopnih udvostručavanja vrednosti promenljive `alocirano`, njena vrednost postati 0 (dokaz ostavljamo čitaocima). Zbog toga se ispostavlja da je beskonačno ponavljanje sasvim moguće. Na primer, u sledećim uslovima bi došlo do beskonačnog ponavljanja petlje:

```
maxint = 65535
alocirano = 32768
velicina = 50000
```

Jedan način da sprečimo beskonačno ponavljanje je da uvedemo odgovarajuću proveru:

```
...
const unsigned maxint = -1;
do  {
    if( alocirano < maxint/2 )
        alocirano *= 2;
    else
        alocirano = maxint;
    }
    while( velicina > alocirano );
...
```

Prvom naredbom se definiše maksimalan ceo broj koji se može opisati tipom `unsigned`. Ako bi udvostručavanje podatka `alocirano` moglo da pređe opseg tipa, umesto udvostručavanja dodeljujemo eksplicitno najveću dopuštenu vrednost. Ona sigurno ne može biti manja od `velicina`, pa će ponavljanje biti prekinuto.

Problem koji smo ovde rešili odnosi se prekoračivanje opsega tipa usled neopreznog izračunavanja računskih operacija. I taj tip grešaka predstavlja veoma čest uzrok problema.

Koliko ovo razmatranje ima smisla? Ako pretpostavimo da se na 32-bitnim računarima celi brojevi predstavljaju 32-bitnim brojevima i da je radna memorija ograničena na 2^{32} bajtova, onda se čini da ovo razmatranje nema smisla, jer će alokacija niza postati nemoguća mnogo pre nego što se broj njegovih elemenata približi opsegu celih brojeva. Međutim, takvo razmišljanje je potpuno pogrešno iz bar dva razloga:

- korisnik klase `Niz` može upotrebiti metod `PromeniVelicinu` sa argumentom koji je dovoljno veliki da napravi probleme, bez obzira na to što se niz date veličine možda ne može alocirati, pa u takvom slučaju umesto očekivanog izuzetka dobijamo mrtvu metlju;
- na nekim računarima i prevodiocima se memorijske adrese predstavljaju većim brojem bitova nego celi brojevi tipa `unsigned`.

Hvatanje izuzetaka

Ako pokušamo da izvršimo program koristeći ranije napisane primere i glavnu funkciju, dobićemo poruku čiji tačan tekst zavisi od prevodioca, a u suštini obaveštava korisnika da je došlo do neuobičajenog prekida rada programa. Takvu poruku ćemo dobiti uvek kada se u programu proizvede izuzetak koji sam program ne obrađuje. Kako to nije najbolji način da se programi predstavljaju svojim korisnicima, dobro je telo funkcije `main` u svakom programu napisati u obliku bloka pokušaja sa univerzalnim hvatačem:

```
main()
{
    try {
        primer1();
        primer2();
        primer3();

        Niz a(5);
```

```
            primer4(a);
            }
        catch(...){
            cout << "Doslo je do nepoznatih problema!" << endl;
            }
        cin.get();
        return 0;
}
```

Sada program ispisuje našu poruku, koju možemo izmeniti i prilagoditi kontekstu.

Ako znamo kakvi se izuzeci mogu očekivati i umemo da odreagujemo na odgovarajuće tipove izuzetaka, onda možemo napisati i hvatače koji bolje reaguju na očekivane izuzetke:

```
main()
{
    try {
        primer1();
        primer2();
        primer3();
        Niz a(5);
        primer4(a);
        }
    catch( char* s ){
        cout << "Problem: " << s << endl;
        }
    catch(...){
        cout << "Doslo je do nepoznatih problema!" << endl;
        }
    cin.get();
    return 0;
}
```

Ako pri pisanju funkcije `primer4` s nekim razlogom očekujemo moguće probleme, onda ima smisla napisati kod za hvatanje izuzetaka već u toj funkciji:

```
void primer4( const Niz& a )
{
    try {
        int v = a.Velicina();
        cout << "Velicina niza: " <<  v << endl;
        cout << "a[" << v << "] = " << a[v] << endl;
        cout << "a[" << (v+1000000) << "] = "
             << a[v+1000000] << endl;
        cout << "a[-1] = " << a[-1] << endl;
        }
    catch( char* s ){
        cout << "Problem u primeru 4: " << s << endl;
        }
}
```

Ne postoje opravdani razlozi da u metodima naše klase `Niz` hvatamo izuzetke. Ako bi razlozi za to postojali, izuzeci bi se hvatali na isti način kao što smo ih hvatali u predstavljenim primerima upotrebe klase `Niz`.

Kao što je već navedeno, izuzeci mogu biti objekti bilo kog tipa. Da se pri hvatanju izuzetaka ne bi izvodilo njihovo kopiranje, u slučaju da su izuzeci objekti neke klase obično se pri hvatanju radi pomoću referenci. Nije retkost ni da se izbacuju izuzeci koji predstavljaju dinamički alocirane objekte, ali u tom slučaju je uhvaćene izuzetke potrebno i osloboditi. Uobičajeno je da se izuzeci hvataju primenom referenci (i pokazivača) na konstantne tipove.

Ako bismo, na primer, izbacili izuzetak tipa `Niz` (iako to može izgledati besmisleno), njegovo izbacivanje i hvatanje bi trebalo da bude napisano ovako:

```
try {
    ...
    throw Niz(42);
    ...
    }
catch( const Niz& n ){
    ...
    }
```

ili ovako:

```
try {
    ...
    throw new Niz(42);
    ...
    }
catch( const Niz* n ){
    ...
    delete n;
    ...
    }
```

Propuštanje izuzetka

Neka je funkcija `primer5` definisana ovako:

```
void primer5()
{
    const Niz* a = new Niz(10000);
    for( unsigned i=0; i<100000; i+=2000 )
        cout << "a[" << i << "] = " << (*a)[i] << endl;
    delete a;
}
```

i neka se poziva iz funkcije `main`:

```
main()
{
    try {
        ...
        primer5();
        }
    ...
}
```

Kada brojač `i` u funkciji `primer5` dostigne vrednost 10000, pokušava se čitanje elementa sa indeksom 10000. Kako je niz konstantan, to predstavlja neispravan indeks, pa dolazi do izbacivanja izuzetka u okviru konstantnog operatora indeksiranja. Kako u metodu `primer5` ne postoji odgovarajući hvatač, funkcija će prekinuti izvršavanje i izuzetak će biti uhvaćen tek u okviru funkcije `main`.

Iz ugla staranja o izuzetku može se učiniti da je sve u redu, jer je izuzetak obrađen tamo gde smo znali kako da na njega odreagujemo. Međutim, imamo veliki problem – niz `a`, koji je dinamički napravljen u okviru funkcije `primer5`, nije uništen! Pri prekidanju izvršavanja bloka koda funkcije `primer5` oslobađaju se svi lokalni automatski objekti, u koje spada i pokazivač `a`, ali ne i objekat na koji pokazivač `a` pokazuje. Svi dinamički napravljeni objekti i drugi dinamički obezbeđeni resursi (memorija, otvorene datoteke, mrežni resursi i sl.) moraju se eksplicitno osloboditi.

Izuzetak možemo uhvatiti i obraditi u funkciji `primer5`:

```
void primer5()
{
    const Niz* a = new Niz(10000);
    try {
        for( unsigned i=0; i<100000; i+=2000 )
            cout << "a[" << i << "] = " << (*a)[i] << endl;
        delete a;
        }
    catch( char* s ){
        delete a;
        cout << "Problem u primeru 5: " << s << endl;
        }
}
```

Međutim, ako želimo da izuzetak ne bude obrađen u okviru ove funkcije, već da prođe dalje, bar do funkcije iz koje je ova pozvana, onda ne smemo dovršiti obrađivanje izuzetka. Umesto toga možemo da ponovimo izbacivanje istog izuzetka naredbom `throw` bez argumenata:

```
void primer5()
{
    const Niz* a = new Niz(10000);
    try {
        for( unsigned i=0; i<100000; i+=2000 )
            cout << "a[" << i << "] = " << (*a)[i] << endl;
        delete a;
        }
    catch( char* s ){
        delete a;
        cout << "Problem u primeru 5: " << s << endl;
        throw;
        }
}
```

Sada se naredbom `throw` prekida obrada izuzetka i ponavlja izbacivanje istog izuzetka, nakon čega se po već opisanom postupku pronalazi najbliži odgovarajući hvatač – u ovom

slučaju u funkciji `main`. Ovakav postupak se naziva *propuštanje izuzetka* ili *ponovljeno izbacivanje izuzetka.* Za razliku od prethodnog primera, koji ispisuje poruku na standardnom izlazu, hvatač koji propušta izuzetak obično ne radi ništa osim oslobađanja resursa i propuštanja izuzetka.

Propuštanje izuzetaka ima veliki značaj u situacijama nalik na prethodnu: kada, s jedne strane, moramo uhvatiti izuzetak da bismo oslobodili zauzete resurse, ali, s druge strane, želimo da se obrada izuzetka izvede na nekom drugom mestu. Kako su takve situacije veoma česte, i propuštanje izuzetka je tehnika koja se veoma često mora primenjivati.

Hijerarhija klase izuzetaka standardne biblioteke predstavljena je u odeljku *10.12 Izuzeci*, na strani 400. Namena i prostor ne dopuštaju da detaljno obradimo specifične situacije u radu sa izuzecima, kao što su izbacivanje i hvatanje izuzetaka u konstruktorima, blok pokušaja kao glavni blok funkcije, deklarisanje tipova izuzetaka koje neka funcija ili metod izbacuju i drugo. Više informacija o tome je na raspolaganju u [Lippman 2005].

Korak 8 - Niz celih brojeva

Pre nego što nastavimo unapređivanje klase `Niz`, navešćemo rešenje do koga smo do sada došli. Napisana klasa `Niz` predstavlja sasvim dobar i upotrebljiv dinamički niz celih brojeva.

```
//---------------------------------------------------------------
//  Klasa Niz
//---------------------------------------------------------------
class Niz
{
public:
    //  Konstruktor. Pravi niz date velicine.
    Niz( unsigned velicina = 0 )
        : _Velicina(0),
          _Alocirano(0),
          _Elementi(0)
        { PromeniVelicinu( velicina ); }

    //  Konstruktor kopije.
    Niz( const Niz& n )
        { init( n ); }

    //  Destruktor.
    ~Niz()
        { delete [] _Elementi; }

    //  Operator dodeljivanja.
    const Niz& operator = ( const Niz& n )
        {
        if( this != &n ){
            delete [] _Elementi;
            init( n );
            }
        return *this;
        }

    //  Indeksni operator.
    int& operator[] ( unsigned i )
        {
```

```
    if( i >= Velicina() )
        PromeniVelicinu( i+1 );
    return _Elementi[i];
    }

//  Indeksni operator. Konstantna verzija
const int& operator[] ( unsigned i ) const
    {
    if( i >= Velicina() )
        throw "Neispravan indeks!";
    return _Elementi[i];
    }

//  Dodavanje novog elementa na kraj niza.
void DodajNaKraj( int x )
    {
    PromeniVelicinu( Velicina() + 1 );
    _Elementi[ Velicina() - 1 ] = x;
    }

//  Izracunavanje poslednjeg elementa niza.
int Poslednji() const
    {
    if( !Velicina() )
        throw "Prazan niz nema poslednji element! ";
    return _Elementi[ Velicina() - 1 ];
    }

//  Brisanje poslednjeg elementa niza.
void ObrisiPoslednji()
    {
    if( !Velicina() )
        throw "Prazan niz nema poslednji element! ";
    PromeniVelicinu( Velicina() - 1 );
    }

//  Povecavanje i smanjivanje niza.
void PromeniVelicinu( unsigned velicina )
    {
    //  Povecavanje...
    if( velicina > _Alocirano ){
        unsigned alocirano = _Alocirano;
        if( alocirano < 5 )
            alocirano = 5;
        const unsigned maxint = -1;
        do  {
            if( alocirano < maxint/2 )
                alocirano *= 2;
            else
                alocirano = maxint;
            }
            while( velicina > alocirano );
        int* noviElementi = new int[alocirano];
        if( _Velicina )
            memcpy( noviElementi, _Elementi,
                    _Velicina * sizeof(int) );
        delete [] _Elementi;
        _Elementi = noviElementi;
```

```
                _Alocirano = alocirano;
                }
            //  Smanjivanje...
            else if( velicina < _Velicina
                     && _Alocirano > 10
                     && velicina < _Alocirano/4
                    )
                {
                unsigned alocirano = velicina;
                int* noviElementi = 0;
                if( velicina ){
                    noviElementi = new int[alocirano];
                    memcpy( noviElementi, _Elementi,
                            velicina * sizeof(int) );
                    }
                delete [] _Elementi;
                _Elementi = noviElementi;
                _Alocirano = alocirano;
                }
            _Velicina = velicina;
            }

        //  Izracunavanje velicine niza.
        unsigned Velicina() const
            { return _Velicina; }

    private:
        //  Inicijalizacija kopije.
        void init( const Niz& n )
            {
            _Alocirano = _Velicina = 0;
            _Elementi = 0;
            if( n._Velicina ){
                _Elementi = new int[n._Velicina];
                _Alocirano = _Velicina = n._Velicina;
                memcpy( _Elementi, n._Elementi,
                        n._Velicina * sizeof(int) );
                }
            }

        //  Clanovi podaci.
        int*        _Elementi;
        unsigned    _Velicina;
        unsigned    _Alocirano;
};
```

Korak 9 - Prilagođavanje drugim tipovima elemenata

Osnovna mana napisanog niza je u tome što može da se upotrebljava samo za cele brojeve tipa `int`. Ako bismo želeli da koristimo neki drugi tip elemenata, morali bismo da napišemo skoro potpuno istu klasu još jednom, ali sa izmenjenim tipom elemenata. Srećom, programski jezik C++ raspolaže mehanizmom koji će nam omogućiti da klasu napišemo jednom za sve tipove podataka, zadržavajući strogu proveru tipova i sve ostale pogodnosti na koje smo navikli.

Programski jezik C je nudio programerima jedno prilično primitivno sredstvo za opisivanje algoritama koji su ispravni za više različitih tipova podataka. To su makroi, jedna od pogodnosti kojima raspolaže tekstualni pretprocesor programskog jezika C, koji obrađuje svaki tekst programa pre nego što započne njegovo prevođenje. Makroe je nasledio i C++.

Uočimo, na primer, jednostavnu funkciju `max`:

```
int max( int x, int y )
    { return x>y ? x : y; }
```

Ovako napisana funkcija može se primenjivati samo na cele brojeve, što je jedna od njenih osnovnih mana. Ako bismo primenili makroe, mogli bismo napisati nešto poput:

```
#define max(x,y) ((x)>(y) ? (x) : (y))
```

Svaka upotreba makroa `max` u programu bi imala za posledicu da bi pretprocesor takvu upotrebu zamenio definicijom makroa. Na primer, upotreba poput:

```
... max( a, b*2 ) ...
```

bi pre početka prevođenja programa bila zamenjena tekstom:

```
... ((a)>(b*2) ? (a) : (b*2)) ...
```

Jedan od kvaliteta koje makroi donose je činjenica da ovako napisan makro jednako dobro radi za sve tipove podataka za koje je definisan operator <. Međutim, ima i veoma ozbiljnih slabosti. Navešćemo samo neke od najvažnijih:

- sintaksa je nečitka, zbog mnogih zagrada koje se moraju upotrebljavati da ne bi bilo neispravnih tumačenja argumenata, jer se oni upotrebljavaju i zamenjuju doslovno kao tekstualne niske;
- ne postoji stroga provera tipova pri prevođenju; štaviše, pri definisanju makroa ne postoji praktično nikakva složena provera, a greške do kojih dolazi pri primeni makroa su često prilično nerazumljive (tj. prevodilac detektuje i prijavljuje grešku pri prevođenju teksta programa u kome je makro zamenjen svojom definicijom, a programer vidi samo poziv makroa);
- makro nema semantiku funkcije, pa je semantika poziva poput `max(a++,++b)` sasvim različita u slučaju funkcije `max` i makroa `max`.

Šabloni funkcija

Programski jezik C++ nudi daleko savremeniji i kompletniji mehanizam za opisivanje algoritama koji se mogu primeniti na više tipova. To su šabloni funkcija.

Ideja je da se funkcija napiše na uobičajen način, ali da se pri pisanju naglasi da upotrebljeno ime tipa nije neki konkretan tip nego parametar koji se određuje prema potrebi. Predstavićemo šablone funkcija na istom primeru:

```
template<class T>
T max( T x, T y )
    { return x>y ? x : y; }
```

Definisan šablon funkcije je upotrebljiv za svaki tip `T` za koji se može uspešno prevesti. Pri tome:

- sintaksa definicije šablona je ista kao sintaksa definicije funkcije, uz dodatak deklaracije šablona u prvom redu;
- sintaksa šablona se ne proverava striktno sama za sebe;
- šablon se posebno prevodi za svaki tip T sa kojim se upotrebljava, proizvodeći odgovarajući broj različitih funkcija;
- pri prevođenju šablona se vrši stroga provera tipova i ostalih potencijalnih neispravnosti, upravo kao da se prevodi obična funkcija a ne šablon;
- svaka od tako prevedenih funkcija ima istu semantiku i efikasnost kao da je napisana bez upotrebe šablona, koristeći kod definicije šablona u kome je parametarski tip (ili tipovi) zamenjen odgovarajućim tipom;

Prethodno napisan šablon se može upotrebljavati uz eksplicitno navođenje tipa:

```
... max<int>( a, 2*b ) ...
```

ali ako se na osnovu tipova argumenata može jednoznačno automatski ustanoviti i tip funkcije, onda eksplicitno navođenje tipa nije neophodno:

```
int a,b;
... max( a, 2*b ) ...
```

Šabloni funkcija se intenzivno upotrebljavaju u standardnoj biblioteci (videti *10.3 Funkcionali*, na strani 351). Više o šablonima funkcija može se pročitati u [Lippman 2005]. Nama je ovde od većeg značaja činjenica da se koncept šablona ne primenjuje samo na funkcije već i na klase.

Umetnute funkcije i metodi

Osim što pružaju delimični polimorfizam, makroi predstavljaju i specifično sredstvo za pisanje brzog koda. Uzrok veće brzine makroa, u odnosu na funkcije, je u načinu njihovog prevođenja. Makroi se prevode tako što se pozivi makroa zamenjuju telom makroa prilikom pretprocesiranja. Rezultat je da se makroi ne pozivaju kao ostale funkcije, već se štedi na prosleđivanju argumenata, pozivanju funkcije i preuzimanju rezultata.

Kao što smo već naglasili, semantika upotrebe makroa se razlikuje od semantike upotrebe funkcija, pa korisnik makroa mora biti svestan da koristi makro (a ne funkciju) ili može imati velikih neprijatnosti. Na primer, primena sledećeg makroa:

```
max(a++,++b)
```

bi se prevela kao:

```
((a++)>(++b) ? (a++) : (++b))
```

pa bi rezultat i bočni efekti bili drugačiji nego u slučaju poziva funkcije.

U programskom jeziku C++ postoji posebna vrsta funkcija, koje imaju istu semantiku kao i sve ostale funkcije, a prevode se poput makroa, umetanjem njihove definicije na mestu na kome se upotrebljavaju. Na taj način se postiže povećavanje efikasnosti bez neprijatnosti koje nose makroi. Takve funkcije se nazivaju *umetnute funkcije* (engl. *inline*).

Da bi se naglasilo da je neku funkciju potrebno prevoditi kao umetnutu, potrebno je (i dovoljno) ispred definicije funkcije navesti ključnu reč `inline`. Da bi se šablon funkcije `max` prevodio kao umetnuta funkcija, potrebno ga je napisati ovako:

```
template<class T>
inline T max( T x, T y )
    { return x>y ? x : y; }
```

Nije dobro, ni moguće, prevoditi sve funkcije kao umetnute. Na primer, potpuno je jasno da rekurzivne funkcije ne mogu da budu umetnute. Zatim, ako je telo funkcije veliko, tada se njenim umetanjem dobija relativno malo na efikasnosti (jer je ušteda srazmerno mala u odnosu na trajanje izvršavanja tela funkcije) a značajno se povećava veličina prevedenog programa, čime se posredno gubi i na efikasnosti. Zbog toga svaki prevodilac ima neke kriterijume na osnovu kojih odlučuje da li će funkcija koja je deklarisana kao umetnuta biti zaista i prevedena kao umetnuta. U većini slučajeva se funkcije neće prevoditi kao umetnute ukoliko u njima postoje petlje, uslovne naredbe ili lokalne promenljive, a postoje i drugi kriterijumi. Ukoliko se funkcija deklarisana kao umetnuta ne prevodi kao umetnuta, prevodilac izdaje upozorenje, koje ne predstavlja grešku (tj. ne ometa prevođenje) već samo obaveštava programera da je njegova deklaracija `inline` ignorisana.

I metodi mogu da se prevode kao umetnuti, ali je u tom slučaju priča malo složenija. Podsetimo se da svaki metod klase može da se definiše na dva načina: u telu klase i van tela klase. Sve metode smo do sada definisali u telu klase, navođenjem definicije metoda u okviru bloka definicije klase:

```
class A
{
...
    int metod()
    {...}
...
};
```

Definisanje metoda van tela klase podrazumeva njegovo deklarisanje u telu klase i navođenje njegove definicije van tela klase. Ako se metod definiše van tela klase, to ne mora da bude u istoj datoteci u kojoj je definisana klasa, ali se u tom slučaju definicija klase mora uključiti u vidu zaglavlja. Pri definisanju metoda van klase se ispred imena metoda navode ime klase i dve dvotačke, u obliku `<ime klase>::<ime metoda>`. Na primer:

```
class A
{
...
    int metod();
...
};
```

```
int A::metod()
{...}
```

Da bi se metod prevodio kao umetnut metod, potrebno je definisati ga u telu klase ili ispred njegove definicije van tela klase navesti ključnu reč `inline`.

To znači da će sve metode koji su definisani u okviru klase prevodilac pokušati da prevede kao umetnute. To dalje znači da će u slučaju složenih metoda, koji se ne mogu prevesti kao umetnuti, prevodilac izdavati odgovarajuća upozorenja. U praktičnom radu je dobro razdvajati definicije metoda od definicije klase, osim u slučaju metoda koji se mogu definisati u jednom ili dva reda. Ipak, u većini klasa u ovoj knjizi metodi su definisani u telu klase, jer se pokazalo da je takav način pisanja razumljiviji i jasniji pri učenju i analiziranju napisanog koda. Zbog toga molimo čitaoce za razmevanje kada se pri prevođenju pojavljuju upozorenja kojima nas obaveštava da neki metodi ne mogu biti prevedeni kao umetnuti. Ističemo, ponovo, da to ne ometa prevođenje i izvršavanje programa.

Šabloni klasa

Iz ugla sintakse, pisanje šablona klasa se razlikuje od pisanja običnih klasa na isti način kao što se pisanje šablona funkcija razlikuje od pisanja funkcija. U narednom primeru je definisan šablon klase `Par`:

```
template<class T1, class T2>
class Par
{
public:
    Par()
        {}
    Par( const T1& x, const T2& y )
        : _prvi(x), _drugi(y)
        {}

    const T1& prvi() const
        { return _prvi; }

    const T2& drugi() const
        { return _drugi; }

private:
    T1 prvi;
    T2 drugi;
};
```

Ako bismo hteli da napišemo klasu `ParCelihBrojeva` ona bi imala potpuno istu strukturu kao definicija šablona klase `Par`, s tim da bi se umesto tipova `T1` i `T2` upotrebljavao tip `int`. Zaista, pisanje šablona klasa se ne razlikuje od pisanja običnih klasa, s tim da se neki tip u klasi zameni oznakom tipa koji predstavlja parametar šablona.

Šablon klase se uvek upotrebljava uz eksplicitno navođenje tipova parametara. Na primer:

```
Par<int,float> a(2,3.14);
float x = a.drugi();
```

Šablon klase se može upotrebiti sa bilo kojim tipovima sa kojima se definicija metoda klase može prevesti. Štaviše, da bi se šablon klase mogao upotrebiti sa nekim konkretnim tipovima kao parametrima, dovoljno je da se za te tipove mogu uspešno prevesti svi metodi koji se eksplicitno ili implicitno upotrebljavaju, jer prevodilac ne bi trebalo ni da pokuša da prevede metode koji se ne upotrebljavaju za upravo te tipove. Na primeru šablona klase `Par` možemo istaći da se svi napisani metodi mogu primeniti ukoliko se za tipove elemenata para upotrebe tipovi za koje postoje i konstruktor bez argumenata i konstruktor kopije. Sa druge strane, ako u tim tipovima ne postoji neki od ovih konstruktora, ni odgovarajući konstruktor klase `Par` neće moći da se prevede, ali se ostali metodi mogu upotrebljavati.

Uobičajeno je da se argumenti šablona funkcija i metoda šablona klasa umesto po vrednosti prenose u obliku referenci na konstantne objekte. Osnovni razlog za to je u uopštenoj nameni šablona. Eventualna upotreba složenih tipova podataka u šablonu mogla bi, u slučaju prenošenja po vrednosti, predstavljati značajan faktor usporavanja programa. Zbog toga što navedeni šablon klase `Par` može za elemente parova imati proizvoljno složene objekte (na primer, objekte klase `Niz` ili neki drugi `Par`), svuda gde su prenose argumenti i rezultati po vrednosti (tj, ne kao promenljivi argumenti) upotrebljeno je prenošenje pomoću referenci na konstantne tipove:

```
template<class T1, class T2>
class Par
{
public:
    Par( const T1& x, const T2& y )
        ...
    const T1& prvi() const
        ...
    const T2& drugi() const
        ...
};
```

Evo i jednog primera malo složenije upotrebe šablona. Objekat koji se pravi je par čija oba elementa predstavljaju parove koji se sastoje od broja i niza:

```
Par< Par<int,Niz>, Par<int,Niz> > par;
par.prvi().prvi() = 42;
par.prvi().drugi().PromeniVelicinu( 42 );
par.drugi().prvi() = 19;
par.drugi().drugi().DodajNaKraj(42);
```

Polimorfizam

Šabloni funkcija i klasa omogućavaju *parametarski polimorfizam*.

Polmorfizam je osobina nekog dela programa da se može ponašati na odgovarajući način za više različitih tipova podataka. U slučaju šablona funkcija i klasa ponašanje napisanog programskog koda se prilagođava tipovima koji se navode kao parametri šablona, zbog čega se ova vrsta polimorfizma naziva paremetarskim polimorfizmom.

Parametarski polimorfizam programskog jezika C++ je u velikoj meri blizak polimorfizmu koji je uobičajen za funkcionalne programske jezike. U većem broju funkcionalnih

programskih jezika se tipovi podataka ne navode eksplicitno u programu, već se automatski ustanovljavaju pri prevođenju ili izvršavanju programa. U takvim situacijama se radi o *implicitnom polimorfizmu* jer se svaka napisana funkcija ponaša polimorfno, u meri u kojoj je njena definicija nezavisna od konkretnih tipova argumenata. Na primer, navedena funkcija `max` bi radila ispravno za sve tipove podataka za koje postoji operator poređenja:

```
max(x,y) = if x > y then x else y;
```

U većini objektno orijentisanih programskih jezika se kao jedina vrsta polimorfizma koristi hijerarhijski polimorfizam. Hijerarhijski polimorfizam se ostvaruje primenom nasleđivanja i pravljenjem hijerarhija klasa koje imaju slično ponašanje. Hijerarhijski polimorfizam se predstavlja i upotrebljava u većini narednih poglavlja.

Programski jezik C++ omogućava upotrebu i hijerarhijskog i parametarskog polimorfizma, što u velikoj meri doprinosi njegovoj primenjivosti. Značaj parametarskog polimorfizma se potvrđuje i kroz činjenicu da se u poslednje vreme i neki drugi programski jezici (npr. Java u verziji 1.5) proširuju odgovarajućim mogućnostima.

Šablon klase Niz

Šta je potrebno da uradimo da bismo od klase `Niz`, koja podržava samo rad sa celobrojnim nizovima, dobili šablon klase koji podržava rad sa proizvoljnim tipovima elemenata?

Pravljenje šablona od gotove klase je sasvim jednostavno. Potrebno je:

- prepoznati koji bi tipovi trebalo da se parametrizuju;
- navesti deklaraciju šablona neposredno ispred definicije klase;
- u deklaracijama članova podataka i metoda izmeniti odgovarajuće tipove;
- za svaki metod proveriti da li je možda implementiran uz upotrebu nekih specifičnosti ranije korišćenog konkretnog tipa.

Prvi korak je jednostavan, jer je sasvim jasno da mi želimo da parametrizujemo tip elemenata niza. Drugi korak je neposredna posledica prvog:

```
template<class TipElemenata>
class Niz
{
...
};
```

Prelazimo na treći korak i u svim podacima i metodima menjamo odgovarajuće tipove. Navodimo samo metode u kojima se nešto menja:

```
template<class TipElemenata>
class Niz
{
public:
...
    //  Indeksni operator.
    TipElemenata& operator[] ( unsigned i )
        {
```

```
        if( i >= Velicina() )
            PromeniVelicinu( i+1 );
        return _Elementi[i];
        }

    //  Indeksni operator. Konstantna verzija
    const TipElemenata& operator[] ( unsigned i ) const
        {
        if( i >= Velicina() )
            throw "Neispravan indeks!";
        return _Elementi[i];
        }

    //  Dodavanje novog elementa na kraj niza.
    void DodajNaKraj( const TipElemenata& x )
        {
        PromeniVelicinu( Velicina() + 1 );
        _Elementi[ Velicina() - 1 ] = x;
        }

    //  Izracunavanje poslednjeg elementa niza.
    const TipElemenata& Poslednji() const
        {
        if( !Velicina() )
            throw "Prazan niz nema poslednji element! ";
        return _Elementi[ Velicina() - 1 ];
        }
...
    //  Povecavanje i smanjivanje niza.
    void PromeniVelicinu( unsigned velicina )
        {
        //  Povecavanje...
        if( velicina > _Alocirano ){
            unsigned alocirano = _Alocirano;
            if( alocirano < 5 )
                alocirano = 5;
            const unsigned maxint = -1;
            do  {
                if( alocirano < maxint/2 )
                    alocirano *= 2;
                else
                    alocirano = maxint;
                }
                while( velicina > alocirano );
            TipElemenata* noviElementi
                                = new TipElemenata[alocirano];
            if( _Velicina )
                memcpy( noviElementi, _Elementi,
                        _Velicina * sizeof(TipElemenata) );
            delete [] _Elementi;
            _Elementi = noviElementi;
            _Alocirano = alocirano;
            }
        //  Smanjivanje...
        else if( velicina < _Velicina
                 && _Alocirano > 10
                 && velicina < _Alocirano/4
```

```
                    )
                {
                unsigned alocirano = velicina;
                TipElemenata* noviElementi = 0;
                if( velicina ){
                    noviElementi = new TipElemenata[alocirano];
                    memcpy( noviElementi, _Elementi,
                            velicina * sizeof(TipElemenata) );
                    }
                delete [] _Elementi;
                _Elementi = noviElementi;
                _Alocirano = alocirano;
                }
            _Velicina = velicina;
            }
...
private:
    //  Inicijalizacija kopije.
    void init( const Niz& n )
        {
        _Alocirano = _Velicina = 0;
        _Elementi = 0;
        if( n._Velicina ){
            _Elementi = new TipElemenata[n._Velicina];
            _Alocirano = _Velicina = n._Velicina;
            memcpy( _Elementi, n._Elementi,
                    n._Velicina * sizeof(TipElemenata) );
            }
        }
    //  Clanovi podaci.
    TipElemenata*   _Elementi;
    int             _Velicina;
    int             _Alocirano;
};
```

Time je završen i treći korak. Da bismo mogli isprobati klasni šablon `Niz`, u primerima upotrebe klase `Niz`, u funkciji `main`, svuda tip `Niz` zamenjujemo tipom `Niz<int>`:

```
void primer1()
{
    Niz<int> a;
    ...
}

void primer2()
{
    Niz<int> b(5);
    ...
}

void primer3()
{
    Niz<int> c(0);
    ...
}
```

```
void primer4( const Niz<int>& a )
{...}

void primer5()
{
    const Niz<int>* a = new Niz<int>(10000);
    ...
}

main()
{
    try {
        ...
        Niz<int> a(5);
        ...
        }
    ...
}
```

Ovo bi trebalo da se prevodi i da radi na potpuno isti način kao i ranije.

Nije slučajno što je u prvim primerima upotrebljen upravo tip `Niz<int>`. Znajući da je klasa inicijalno pisana za elemente tipa `int`, trebalo bi očekivati da sa ovakvom primenom ne bude posebnih problema. Pre nego što pokušamo da primenimo šablon `Niz` sa nekim drugim tipom elemenata, trebalo bi da izvedemo i četvrti korak.

Potrebno je detaljno analizirati metode klase i potražiti da li je nešto urađeno uz pretpostavku da su elementi tipa `int`, što više ne mora da važi. Najviše pažnje bi trebalo posvetiti onim metodima koji koriste članove podatke čiji tip odgovara parametarskom tipu. U slučaju šablona `Niz` jedini takav podatak je pokazivač `_Elementi`.

Pokazuje se da u klasi `Niz` postoje neke operacije koje za elemente tipa `int` rade savršeno, ali bi za elemente nekog drugog tipa dovele do problema. Zapravo, već je ranije, pri diskusiji na temu oslobađanja podataka, naglašeno da postoji jedna nedoslednost u metodu za povećavanje niza. Ona se tiče načina prepisivanja sadržaja iz starog niza elemenata u novi. Naime, dok je s jedne strane naglašeno da se za oslobađanje niza elemenata mora upotrebljavati izraz `delete[]`, kako bi se izvršila destrukcija svakog pojedinačnog elementa niza koji se briše, sa druge strane se za kopiranje elemenata niza upotrebljava funkcija `memcpy` koja doslovno kopira nizove bajtova ne uzimajući u obzir prirodu objekata koji se kopiraju. To dovodi do problema u slučaju da su elementi niza nekog klasnog tipa za koji su definisani konstruktor kopije i operator dodeljivanja, jer se primenom funkcije `memcpy` obavlja samo plitko kopiranje, a ne duboko kopiranje koje obavljaju navedeni metodi.

Zato je upotrebu funkcije `memcpy` potrebno zameniti odgovarajućim kopiranjem objekata. Umesto:

```
memcpy( dest, src, n*sizeof(TipElementa) );
```

upotrebljavamo:

```
for( int i=0; i<n; i++ )
    dest[i] = src[i];
```

ili šablon funkcije `copy` standardne biblioteke programskog jezika C++, koji kopira niz objekata na praktično isti način:

```
copy( src, src+n, dest );
```

Šablon funkcije `copy` je definisan u zaglavlju `algorithm`.

```
#include <algorithm>

template<class TipElemenata>
class Niz
{
public:
...
    //  Povecavanje i smanjivanje niza.
    void PromeniVelicinu( unsigned velicina )
        {
        //  Povecavanje...
        if( velicina > _Alocirano ){
            unsigned alocirano = _Alocirano;
            if( alocirano < 5 )
                alocirano = 5;
            const unsigned maxint = -1;
            do  {
                if( alocirano < maxint/2 )
                    alocirano *= 2;
                else
                    alocirano = maxint;
                }
                while( velicina > alocirano );
            TipElemenata* noviElementi
                                = new TipElemenata[alocirano];
            if( _Velicina )
                copy( _Elementi, _Elementi + _Velicina,
                      noviElementi );
            delete [] _Elementi;
            _Elementi = noviElementi;
            _Alocirano = alocirano;
            }

        //  Smanjivanje...
        else if( velicina < _Velicina
                 && _Alocirano > 10
                 && velicina < _Alocirano/4
                )
            {
            unsigned alocirano = velicina;
            TipElemenata* noviElementi = 0;
            if( velicina ){
                noviElementi = new TipElemenata[alocirano];
                copy( _Elementi, _Elementi + velicina,
                      noviElementi );
                }
            delete [] _Elementi;
            _Elementi = noviElementi;
            _Alocirano = alocirano;
            }
```

```
            _Velicina = velicina;
            }
...
private:
    //  Inicijalizacija kopije.
    void init( const Niz& n )
        {
        _Alocirano = _Velicina = 0;
        _Elementi = 0;
        if( n._Velicina ){
            _Elementi = new TipElemenata[n._Velicina];
            _Alocirano = _Velicina = n._Velicina;
            copy( n._Elementi, n._Elementi + n._Velicina,
                  _Elementi );
            }
        }
...
};
```

Pored primene funkcije `memcpy`, nije bilo drugih delova koda klase `Niz` u kojima je pretpostavljano da su elementi niza tipa `int`.

Šabloni u standardnoj biblioteci

Najveći deo standardne biblioteke programskog jezika C++ definisan je u vidu šablona klasa ili šablona funkcija. Jedna od najčešće upotrebljavanih klasa standardne biblioteke je šablon klase `vector`, koji predstavlja implementaciju dinamičkog niza. Druga važna klasa je šablon klase `list`, koji predstavlja implementaciju liste.

Šablon klase `vektor` ima značajnih sličnosti sa našim šablonom klase `Niz`:

- opcioni argument konstruktora je početna veličina niza;
- za pristupanje elementima niza se koristi operator indeksnog pristupa;
- metod `push_back` dodaje novi element na kraj niza;
- metod `back` izračunava poslednji element niza;
- metod `pop_back` briše poslednji element niza;
- metodi `size` i `length` izračunavaju dužinu niza;
- metod `resize` menja veličinu niza.

Primetimo i par razlika:

- kod operatora indeksnog pristupa elementima klase `vector` pretpostavlja se da se uvek pristupa elementima u ispravnom opsegu i ne menja se automatski veličina niza ako je indeks veći od trenutne veličine niza;
- većina implementacija klase `vector` ne vrši automatsko smanjivanje alociranog prostora u slučaju smanjivanja niza.

Naravno, klasa `vector` ima još značajnih metoda. Potpuniji pregled najvažnijih metoda je naveden u dodacima, u odeljku *10.6.2 Vektor*, na strani 360. Klasa `vector` će se upotrebljavati u nekim od narednih primera.

Prevođenje šablona

Kao što je već naglašeno, šabloni se ne prevode kao nezavisna samostalna celina. Umesto toga se, kada se po prvi put upotrebi neka šablonska funkcija ili neki metod šablona klase, izvodi prevođenje funkcije ili metoda odgovarajućeg tipa. Takav način prevođenja ima više važnih posledica:

- i deklaracije i implementacije šablona funkcija i svih metoda šablona klasa moraju biti navedene u istoj datoteci u kojoj se i koriste (bilo neposredno ili putem uključivanja odgovarajućeg zaglavlja):
  - uobičajeno je da se i deklaracija i definicija šablona pišu u istom zaglavlju;
  - čak i kada se implementacije metoda šablona klase pišu izvan definicije klase, one se najčešće pišu u istoj datoteci zaglavlja u kojoj je i definicija klase, jer se i one moraju uključiti pre prevođenja koda koji te metode koristi;
- eventualne greške u implementaciji nekog metoda šablona klase se mogu uočiti tek kada po prvi put pokušamo da prevedemo neku primenu tog metoda, jer se i taj metod tek tada po prvi put prevodi;
- čak i ako šablon klase nije u potpunosti primenjiv za neki konkretan tip (jer se u nekim metodima upotrebljavaju metodi ili operatori koji ne postoje za dati tip), šablon klase se i dalje može upotrebljavati za dati tip sve dok se ne pokušaju upotrebiti i problematični metodi.

Iako značajan broj prevodilaca podržava i drugačiji način prevođenja šablona, ovde toj temi nećemo posvetiti posebnu pažnju. O načinima prevođenja šablona se može pročitati u [Lippman 2005].

U rešenju zadatka, koje sledi, definicija (i deklaracija i implementacija) šablona klase `Niz` je napisana u datoteci `Niz.h`, koja se zatim upotrebljava u primeru programa, koji je napisan u datoteci `NizPrimer.cpp`.

3.3 Rešenje

Datoteka Niz.h

```
//---------------------------------------------------------------
//  Klasa Niz
//---------------------------------------------------------------
template<class TipElemenata>
class Niz
{
public:
    //  Konstruktor. Pravi niz date velicine.
    Niz( unsigned velicina = 0 )
        : _Velicina(0),
          _Alocirano(0),
```

```
      _Elementi(0)
    { PromeniVelicinu( velicina ); }

// Konstruktor kopije.
Niz( const Niz& n )
    { init( n ); }

// Destruktor.
~Niz()
    { delete [] _Elementi; }

// Operator dodeljivanja.
const Niz& operator = ( const Niz& n )
    {
    if( this != &n ){
        delete [] _Elementi;
        init( n );
        }
    return *this;
    }

// Indeksni operator.
TipElemenata& operator[] ( unsigned i )
    {
    if( i >= Velicina() )
        PromeniVelicinu( i+1 );
    return _Elementi[i];
    }

// Indeksni operator. Konstantna verzija
const TipElemenata& operator[] ( unsigned i ) const
    {
    if( i >= Velicina() )
        throw "Neispravan indeks!";
    return _Elementi[i];
    }

// Dodavanje novog elementa na kraj niza.
void DodajNaKraj( const TipElemenata& x )
    {
    PromeniVelicinu( Velicina() + 1 );
    _Elementi[ Velicina() - 1 ] = x;
    }

// Izracunavanje poslednjeg elementa niza.
const TipElemenata& Poslednji() const
    {
    if( !Velicina() )
        throw "Prazan niz nema poslednji element! ";
    return _Elementi[ Velicina() - 1 ];
    }

// Brisanje poslednjeg elementa niza.
void ObrisiPoslednji()
    {
    if( !Velicina() )
        throw "Prazan niz nema poslednji element! ";
    PromeniVelicinu( Velicina() - 1 );
    }
```

```
        //  Povecavanje i smanjivanje niza.
        void PromeniVelicinu( unsigned velicina )
            {
            //  Povecavanje...
            if( velicina > _Alocirano ){
                unsigned alocirano = _Alocirano;
                if( alocirano < 5 )
                    alocirano = 5;
                const unsigned maxint = -1;
                do  {
                    if( alocirano < maxint/2 )
                        alocirano *= 2;
                    else
                        alocirano = maxint;
                    }
                    while( velicina > alocirano );
                TipElemenata* noviElementi
                                    = new TipElemenata[alocirano];
                if( _Velicina )
                    copy( _Elementi, _Elementi + _Velicina,
                          noviElementi );
                delete [] _Elementi;
                _Elementi = noviElementi;
                _Alocirano = alocirano;
                }

            //  Smanjivanje...
            else if( velicina < _Velicina
                     && _Alocirano > 10
                     && velicina < _Alocirano/4
                    )
                {
                unsigned alocirano = velicina;
                TipElemenata* noviElementi = 0;
                if( velicina ){
                    noviElementi = new TipElemenata[alocirano];
                    copy( _Elementi, _Elementi + velicina,
                          noviElementi );
                    }
                delete [] _Elementi;
                _Elementi = noviElementi;
                _Alocirano = alocirano;
                }

            _Velicina = velicina;
            }

        //  Izracunavanje velicine niza.
        unsigned Velicina() const
            { return _Velicina; }

    private:
        //  Inicijalizacija kopije.
        void init( const Niz& n )
            {
            _Alocirano = _Velicina = 0;
            _Elementi = 0;
```

```
            if( n._Velicina ){
                _Elementi = new TipElemenata[n._Velicina];
                _Alocirano = _Velicina = n._Velicina;
                copy( n._Elementi, n._Elementi + n._Velicina,
                      _Elementi );
                }
            }

        //  Clanovi podaci.
        TipElemenata*   _Elementi;
        unsigned        _Velicina;
        unsigned        _Alocirano;
};
```

Datoteka NizPrimer.cpp

```
#include <iostream>
#include <algorithm>

using namespace std;

#include "Niz.h"

//-----------------------------------------------------------------
// Primeri upotrebe klase Niz.
//-----------------------------------------------------------------
void primer1()
{
    //  Pravimo jedan prazan niz
    Niz<int> a;
    //  Dodajemo 5 elemenata na kraj niza
    for( int i=0; i<5; i++ )
        a.DodajNaKraj( i );
    //  Smanjujemo niz sa 5 na 3 elementa
    a.PromeniVelicinu(3);
    //  Ipisujemo elemente unazad, prazneci niz
    while( a.Velicina() ){
        cout << a.Poslednji() << endl;
        a.ObrisiPoslednji();
        }
}

void primer2()
{
    //  Pravimo niz koji inicijalno ima 5 elemenata
    Niz<int> b(5);
    //  Punimo niz brojevima 0-9,
    //  automatski menjajuci velicinu niza
    for( int i=0; i<10; i++ )
        b[i] = i;
    //  Ispisujemo elemente niza
    for( unsigned i=0; i<b.Velicina(); i++ )
        cout << b[i] << endl;
    //  Povecavamo niz
    b.PromeniVelicinu(20);
    //  Povecavamo niz
    b[99] = 99;
    cout << "Poslednji element je b["
         << (b.Velicina()-1) << "] = "
```

```
          << b.Poslednji() << endl;
}

void primer3()
{
    Niz<int> c(0);
    for( int i=0; i<1000; i++ ){
        for( int j=0; j<1000; j++ )
            c.DodajNaKraj(j);
        cout << '.';
        }
    cout << endl;

    for( int i=0; i<1000; i++ ){
        for( int j=0; j<1000; j++ )
            c.ObrisiPoslednji();
        cout << '.';
        }
    cout << endl;
}

void primer4( const Niz<int>& a )
{
    try {
        int v = a.Velicina();
        cout << "Velicina niza: " <<  v << endl;
        cout << "a[" << v << "] = " << a[v] << endl;
        cout << "a[" << (v+1000000) << "] = "
             << a[v+1000000] << endl;
        cout << "a[-1] = " << a[-1] << endl;
        }
    catch( char* s ){
        cout << "Problem u primeru 4: " << s << endl;
        }
}

void primer5()
{
    const Niz<int>* a = new Niz<int>(10000);
    try {
        for( unsigned i=0; i<100000; i+=2000 )
            cout << "a[" << i << "] = " << (*a)[i] << endl;
        delete a;
        }
    catch( char* s ){
        delete a;
        cout << "Problem u primeru 5: " << s << endl;
        throw;
        }
}

//---------------------------------------------------------------
// Glavna funkcija programa demonstrira upotrebu klase Niz.
//---------------------------------------------------------------
main()
{
    try {
        primer1();
```

```
            primer2();
            primer3();
            Niz<int> a(5);
            primer4(a);
            primer5();
            }
        catch( char* s ){
            cout << "Problem: " << s << endl;
            }
        catch(...){
            cout << "Doslo je do nepoznatih problema!" << endl;
            }
        cin.get();
        return 0;
    }
```

3.4 Rezime

Najvažnije teme obuhvaćene ovim primerom su:

- dinamičko alociranje i oslobađanje nizova:
- razlika između izraza `new` i `new[]` i između izraza `delete` i `delete[];`
- koncept dinamičkog menjanja veličine niza;
- način pisanja operatora indeksnog pristupa;
- dvojaka struktura klase `Niz` – klasa `Niz` ima jednu strukturu koju stavlja na raspolaganje korisniku i drugu koju interno koristi: dok korisnik vidi veličinu koja je opisana podatkom `_Velicina`, dotle je zaista alocirano `_Alocirano` elemenata;
- robusnost programa;
- upotreba izuzetaka;
- pisanje šablona klasa.

Radi vežbe se napisana klasa `Niz` može dalje menjati. Moguće je, na primer:

- napisati operatore za čitanje i ispisivanje sadržaja niza;
- primeniti koncept iteratora na klasu `Niz` (iteratori su predstavljeni u dodacima, u odeljku 10.2 *Iteratori*, na strani 348);
- eksperimentisati sa različitim koracima povećavanja i smanjivanja niza;
- na mestima gde se upotrebljavaju (izbacuju i hvataju) izuzeci, upotrebiti izuzetke hijerarhije klasa izuzetaka standardne biblioteke (videti *10.12 Izuzeci*, na strani 400).

4 - Perionica automobila

4.1 Zadatak

Napisati klasu `Perionica`, koja simulira rad perionice automobila. Napisati metode za dodavanje vozila u red, proveravanje da li ima vozila u redu i pranje prvog vozila iz reda. Cenu pranja obračunavati na osnovu vrste vozila, broja vrata, točkova, sedišta i osnovne cene pranja za vrstu vozila. Podržati različite vrste vozila pravljenjem hijerarhije klasa vozila, bez članova podataka čije bi vrednosti predstavljale vrstu vozila, broj vrata, točkova i slično.

Cilj zadatka

Pisanjem hijerarhije klasa vozila i klase `Perionica` ćemo upoznati:

- šta je i kako se koristi hijerarhijski polimorfizam;
- kako se pišu i upotrebljavaju hijerarhije klasa u programskom jeziku C++;
- koje su razlike između statičkog i dinamičkog vezivanja metoda;
- kada se i kako upotrebljava dinamičkog vezivanja metoda;
- apstraktne metode i klase.

Pored toga:

- podsetićemo se kako se pišu šabloni klasa;
- upoznaćemo šablon klase `queue` standardne biblioteke.

Pretpostavljena znanja

Da bi se moglo pratiti rešavanja ovog zadatka, potrebna su osnovna znanja o:

- pisanju klasa;
- konceptu nasleđivanja klasa;
- radu sa pokazivačima i referencama.

4.2 Rešavanje zadatka

Najpre ćemo napisati klasu `Vozilo`, na primeru automobila, i klasu `Perionica` koja je u stanju da radi sa tako modeliranim vozilima. Zatim ćemo, uz postepeno rešavanje uočenih problema, klasu `Vozilo` zameniti odgovarajućom hijerarhijom klasa.

Korak 1 - Prvi koraci

Klasu Vozilo ćemo napisati kao jednostavnu klasu sa svega nekoliko metoda:

```
class Vozilo
{
public:
    string Vrsta() const
        { return "Automobil"; }
    int BrojProzora() const
        { return 6; }
    int BrojTockova() const
        { return 4; }
    int BrojSedista() const
        { return 4; }
};
```

Napišimo prototip klase `Perionica`, koji se ponaša kao da u redu vozila uvek postoji neko vozilo:

```
class Perionica
{
public:
    void DodajVoziloURed( Vozilo v )
        {}

    bool ImaVozilaURedu() const
        { return true; }

    void OperiPrvoVozilo()
        { cout << "Oprano." << endl; }

private:
    Vozilo IzdvojiPrvoVozilo()
        { return Vozilo(); }
};
```

Sada možemo napisati i glavnu funkciju programa:

```
#include <iostream>
...
//---------------------------------------------------------------
// Glavna funkcija programa demonstrira rad perionice.
//---------------------------------------------------------------
main(){
    Perionica kodZike;
    kodZike.DodajVoziloURed( Vozilo() );
    kodZike.OperiPrvoVozilo();
    return 0;
}
```

Postojeći metod `OperiPrvoVozilo` je prilično siromašan. Proširimo izveštaj opisom vozila koje se pere:

```
class Perionica
{
public:
...
    void OperiPrvoVozilo()
        {
        if( ImaVozilaURedu() ){
            Vozilo v = IzdvojiPrvoVozilo();
            cout
                << "Na redu je jedan "
                    << v.Vrsta() << "." << endl
                << " - Prvo peremo prozore, ima ih "
                    << v.BrojProzora() << endl
                << " - Zatim prelazimo na tockove, ima ih "
                    << v.BrojTockova() << endl
                << " - Sada su na redu sedista, njih "
                    << v.BrojSedista() << endl
                << " - Gotovo, za sada." << endl;
            }
        }
...
};
```

Korak 2 - Red za čekanje

Vozila koja pristižu u perionicu bi trebalo da sačekaju svoj red i da budu oprana u redosledu u kome su pristigla. Slični problemi se pojavljuju veoma često u razvoju programa. Radi ilustracije rada sa redovima napisaćemo klasu `Red`. Uobičajene operacije pri radu sa redovima su dodavanje elemenata na kraj reda, proveravanje da li u redu ima elemenata i uzimanje elemenata sa početka reda. Da bismo mogli praviti redove za različite tipove elemenata, možemo napisati šablon klase:

```
template<class T>
class Red
{
public:
    bool Prazan() const;
```

```
    void DodajNaKraj( const T& x );
    const T& Prvi() const;
    void ObrisiPrvi();
};
```

Sekvencijalna struktura podataka koja nam omogućava da jednostavno i efikasno dodajemo elemente na kraj, a uzimamo ih sa početka jeste lista. Zbog toga ćemo za čuvanje elemenata reda upotrebiti šablon klase `list` standardne biblioteke (videti *10.6.1 Lista*, na strani 357):

```
#include <list>

template<class T>
class Red
{
public:
    bool Prazan() const
        { return elementi.empty(); }

    void DodajNaKraj( const T& x )
        { elementi.push_back(x); }

    const T& Prvi() const
        { return elementi.front(); }

    void ObrisiPrvi()
        { elementi.pop_front(); }

private:
    list<T> elementi;
};
```

Nakon pisanja klase `Red` teško je ne zapitati se kako da to neko nije uradio ranije i stavio u standardnu biblioteku? Zapravo, standardna biblioteka sadrži šablon klase `queue`, koji ima veoma slično ponašanje. Umesto naših metoda se koriste, redom, metodi: `empty`, `push`, `front` i `pop`. Šablon klase `queue` je opisan u dodacima, u odeljku *10.7.2 Red*, na strani 367.

Red za čekanje vozila na pranje ćemo implementirati pomoću šablona klase `queue`. Šablon klase `Red` nam više nije neophodan:

```
#include <queue>
...
class Perionica
{
public:
    void DodajVoziloURed( Vozilo v )
        { red.push(v); }

    bool ImaVozilaURedu() const
        { return !red.empty(); }

    void OperiPrvoVozilo()
        {
        if( ImaVozilaURedu() ){
            Vozilo v = IzdvojiPrvoVozilo();
            cout
                << "Na redu je jedan "
                    << v.Vrsta() << "." << endl
```

```
                    << " - Prvo peremo prozore, ima ih "
                        << v.BrojProzora() << endl
                    << " - Zatim prelazimo na tockove, ima ih "
                        << v.BrojTockova() << endl
                    << " - Sada su na redu sedista, njih "
                        << v.BrojSedista() << endl
                    << " - Gotovo, za sada." << endl;
                }
            }

    private:
        Vozilo IzdvojiPrvoVozilo()
            {
            Vozilo v = red.front();
            red.pop();
            return v;
            }

        queue<Vozilo> red;
    };
```

Sada možemo dopuniti glavnu funkciju programa:

```
    main(){
        Perionica kodZike;

        kodZike.DodajVoziloURed( Vozilo() );
        kodZike.DodajVoziloURed( Vozilo() );

        while( kodZike.ImaVozilaURedu() )
            kodZike.OperiPrvoVozilo();

        return 0;
    }
```

Korak 3 - Pravljenje hijerarhije klasa

Pažljivi čitaoci, koji poznaju osnovne koncepte nasleđivanja i izgradnje hijerarhija klasa će sasvim verovatno uočiti da ćemo u narednim koracima učiniti nekoliko grešaka. One će nam poslužiti da skrenemo pažnju na slične probleme koji se veoma često ispoljavaju ne samo tokom učenja već i pri profesionalnom pisanju programa.

Prvi pokušaj

Do sada smo prali samo automobile. Pravi je trenutak da program dopunimo novom vrstom vozila. Na redu su kupei. U postavci zadatka se sugeriše da bi sve vrste vozila trebalo da pripadaju jednoj hijerarhiji klasa. Hijerarhija klasa se uobičajeno predstavlja kao stablo, koje u korenu ima baznu klasu, a u čvorovima i listovima izvedene klase. Hijerarhija klasa se sastoji od jedne *bazne* klase i izvesnog broja *izvedenih* klasa (klasa *naslednica*) koje neposredno ili posredno nasleđuju baznu klasu. Svaka od izvedenih klasa se definiše kao naslednica tačno jedne klase[8].

[8] Programski jezik C++ omogućava i pisanje složenijih hijerarhija uz primenu višestrukog nasleđivanja, u kom slučaju izvedena klasa može naslediti više drugih klasa, a hijerarhije mogu imati više baznih klasa. Neki drugi objektno orijentisani programski jezici (Java, C# i drugi) ne

Površan pristup problemu izgradnje hijerarhije klasa vozila bi mogao da ima za rezultat ovako napisanu klasu `Kupe`:

```
class Kupe : public Vozilo
{
public:
    string Vrsta() const
        { return "Kupe"; }

    int BrojProzora() const
        { return 4; }

    int BrojSedista() const
        { return 2; }
};
```

Predstavljeno dodavanje novog tipa vozila je neispravno iz više razloga. Neispravnosti ćemo uočavati i ispravljati jednu po jednu.

Pri definisanju klase `Kupe`, u samoj deklaraciji imena klase, naveli smo da se ona definiše kao naslednik klase `Vozilo`:

```
class Kupe : public Vozilo
```

Nasleđivanjem klase `Vozilo`, klasa `Kupe` nasleđuje sve njene javne i zaštićene metode. Pri tome se nasleđeni javni metodi klase `Vozilo` u klasi `Kupe` ponašaju kao javno definisani, jer je tako navedeno u deklaraciji nasleđivanja. Da je umesto ključne reči `public` u deklaraciji nasleđivanja navedeno `protected` ili `private`, tada bi se javni metodi klase `Vozilo` u klasi `Kupe` ponašali kao da su, redom, zaštićeni ili privatni.

Javno nasleđivanje se obično naziva i *nasleđivanjem ponašanja*, dok se privatno i zaštićeno nasleđivanje nazivaju *nasleđivanjem strukture*. Ovakvi nazivi su tesno vezani sa uobičajenim sakrivanjem strukture klase od njenog korisnika, kome se predstavlja samo ponašanje klase. Nasleđivanje se skoro uvek definiše kao javno. Suština nasleđivanja je da se omogući upotreba već definisanih metoda klase `Vozilo` u klasi `Kupe`. Ukoliko bi nasleđivanje bilo privatno ili zaštićeno, takva upotreba ne bi bila moguća od strane korisnika klase. Veoma su retke situacije u kojima nasleđivanje strukture ima značajne prednosti u odnosu na agregaciju (tj. ugrađivanje odgovarajućih elemenata u samu klasu), pa se zbog toga veoma retko upotrebljava. *Umesto da neka klasa B privatno nasleđuje neku klasu A, najčešće je daleko bolje da se u klasi B definiše član podatak tipa A.*

U daljem tekstu ćemo pri pominjanju nasleđivanja podrazumevati da se radi o nasleđivanju ponašanja, tj. o javnom nasleđivanju.

Naša klasa `Kupe` nasleđuje klasu `Vozilo` i sve njene metode. Međutim, neki od nasleđenih metoda ne rade na odgovarajući način. Zbog toga ih u klasi `Kupe` *redefinišemo*. Redefinisanje

---

podržavaju višestruko nasleđivanje u istom obliku, već uvode ograničenje da se sme naslediti najviše jedna klasa i proizvoljan broj interfejsa, pri čemu se interfejsom smatra klasa bez podataka, čiji su svi metodi apstraktni. Zbog toga što se u okvirima ove knjige bavimo isključivo jednostrukim nasleđivanjem, možemo se držati navedenog opisa hijerarhije klasa.

metoda (engl. *override*) predstavlja definisanje drugačijeg ponašanja izvedene klase u odnosu na baznu klasu. Pri upotrebi metoda izvedene klase, koji predstavljaju redefiniciju nasleđenih metoda, prevodilac daje prednost novim definicijama. Ako, iz bilo kog razloga, želimo da upotrebimo upravo metod bazne klase, to možemo učiniti navođenjem kvalifikovanog imena metoda u obliku:

```
<ime klase>::<ime metoda>
```

Ponašanje redefinisanih metoda možemo ilustrovati sledećim segmentom koda:

```
Kupe k;
// Koristimo metod Vrsta klase Kupe
cout << k.Vrsta() << endl;
// Koristimo metod Vrsta klase Vozilo
cout << k.Vozilo::Vrsta() << endl;
```

Preostaje nam da izmenimo glavnu funkciju programa tako da se u red dodaje jedno vozilo i jedan kupe:

```
main(){
    Perionica kodZike;

    kodZike.DodajVoziloURed( Vozilo() );
    kodZike.DodajVoziloURed( Kupe() );

    while( kodZike.ImaVozilaURedu() )
        kodZike.OperiPrvoVozilo();

    return 0;
}
```

Iako bismo mogli očekivati (ili se bar nadati) da sada izveštaj o pranju vozila sadrži opis jednog vozila i jednog kupea, rezultat izvršavanja programa je nešto drugačiji:

```
Na redu je jedan Automobil.
 - Prvo peremo prozore, ima ih 6
 - Zatim prelazimo na tockove, ima ih 4
 - Sada su na redu sedista, njih 4
 - Gotovo, za sada.
Na redu je jedan Automobil.
 - Prvo peremo prozore, ima ih 6
 - Zatim prelazimo na tockove, ima ih 4
 - Sada su na redu sedista, njih 4
 - Gotovo, za sada.
```

Gde smo pogrešili?

Upotreba pokazivača

Jedna od osnovnih osobina programskog jezika C++ je da se pri radu sa objektima koji pripadaju hijerarhijama klasa moraju upotrebljavati pokazivači ili reference. Razmotrimo sledeći segment koda:

```
Kupe k;
Vozilo v = k;
```

Da li je `v` objekat klase `Vozilo` ili objekat klase `Kupe`? U opštem slučaju, klasa naslednica sadrži sve što sadrži i bazna klasa i eventualno još ponešto, što je definisano samo u izvedenoj klasi. Odatle je potpuno jasno da se pri kopiranju objekta izvedene klase u objekat bazne klase ne može izvesti potpuno kopiranje, već da se mogu iskopirati samo oni delovi koji postoje u obe klase, tj. pripadaju baznoj klasi. Zbog toga je `v` upravo `Vozilo` a ne `Kupe`.

Jedini način da se rukuje uopštenim objektima klasa neke hijerarhije jeste primena pokazivača i referenci. U narednom segmentu koda i `k` i `v` pokazuju na objekat klase `Kupe`:

```
Kupe* k = new Kupe();
Vozilo* v = k;
```

U klasi `Perionica` je objektima klase `Vozilo` rukovano neposredno, što ima za posledicu da se u redu nalaze samo objekti bazne klase. Znajući za predstavljenu osobinu programskog jezika C++, sada moramo izmeniti klasu `Perionica` tako da se objektima vozila manipuliše posredstvom pokazivača:

- tip elemenata reda menjamo tako da radimo sa pokazivačima;
- menjamo sve metode u kojima se koriste elementi reda tako da im pristupaju posredstvom pokazivača;
- nakon pranja je potrebno obrisati vozilo (ne krpom, nego iz memorije), jer se više nigde ne upotrebljava.

Sa opisanim izmenama, klasa `Perionica` može izgledati ovako:

```
class Perionica
{
public:
    void DodajVoziloURed( Vozilo* v )
        { red.push(v); }
...
    void OperiPrvoVozilo()
        {
        if( ImaVozilaURedu() ){
            Vozilo* v = IzdvojiPrvoVozilo();
            cout
                << "Na redu je jedan "
                    << v->Vrsta() << "." << endl
                << " - Prvo peremo prozore, ima ih "
                    << v->BrojProzora() << endl
                << " - Zatim prelazimo na tockove, ima ih "
                    << v->BrojTockova() << endl
                << " - Sada su na redu sedista, njih "
                    << v->BrojSedista() << endl
                << " - Gotovo, za sada." << endl;
            delete v;
            }
        }
private:
    Vozilo* IzdvojiPrvoVozilo()
        {
        Vozilo* v = red.front();
```

```
            red.pop();
            return v;
            }
        queue<Vozilo*> red;
};
```

Primetimo da je moguće da se objekat klase `Perionica` uništi pre nego što se red vozila isprazni. Zbog toga je neophodno obezbediti destruktor klase `Perionica` koji briše vozila koja su u redu. Zašto takav destruktor ranije nije bio potreban? Pri podrazumevanoj destrukciji objekta klase `Perionica` automatski se pozivaju destruktori za sve članove podatke koji nisu pokazivači ili reference. Tako se automatski poziva i destruktor podatka `red` klase `queue`, koji se pri destrukciji automatski prazni. Problem je u tome što sada elementi kolekcije više nisu objekti (tipa `Vozilo`) nego pokazivači na objekte (tipa `Vozilo*`). Dok je pri pražnjenju reda ranije izvršavano brisanje objekata klase `Vozilo`, sada se pri pražnjenju reda brišu podaci tipa `Vozilo*`, pri čemu se objekti na koje se pokazuje ne brišu. Kada su elementi neke kolekcije dinamički alocirani, neophodno je da ih dinamički i oslobodimo. Pravo mesto za to je destruktor klase `Perionica`:

```
class Perionica
{
public:
    ~Perionica()
        { IsprazniRed(); }
...
private:
    void IsprazniRed()
        {
        while( ImaVozilaURedu() )
            delete IzdvojiPrvoVozilo();
        }
...
};
```

U prethodnim primerima smo više puta naglasili da postojanje destruktora klase zahteva pisanje konstruktora kopije i operatora dodeljivanja. Takođe, opisali smo i kako se implementacija ovih metoda može bezbedno izbeći, ukoliko smo potpuno sigurni da ne postoji potreba da se pravlje kopije objekata:

```
class Perionica
{
...
private:
    Perionica( const Perionica& );
    Perionica& operator=( const Perionica& );
...
};
```

Napominjemo da ovakav način zaobilaženja problema kopiranja objekata klase `Perionica`, iako je potpuno sintaksno i semantički ispravan, nije dobro primenjivati u praksi. Najčešće je lakše i bolje definisati ove metode u potpunosti ispravno pri prvom pisanju klase, nego čekati da oni postanu potrebni u nekom kasnijem trenutku, kada će verovatno biti

potrebno posvetiti neko dodatno vreme podsećanju na principe funkcionisanja klase da bi se oni mogli napisati. U konkretnom slučaju, kopiranje klase `Perionica` nam zaista nije neophodno. Pored toga, implementacija kopiranja bi zahtevala i da naučimo kako da kopiramo objekte hijerarhije klasa. Time ćemo se baviti u primeru *8 - Kodiranje*, na strani 295.

Deklarisanjem konstruktora kopije učinili smo da više ne postoji podrazumevani konstruktor bez argumenata, pa ga moramo eksplicitno definisati:

```
class Perionica
{
public:
    Perionica()
        {}
...
};
```

Prevedimo program i pokrenimo ga:

```
Na redu je jedan Automobil.
 - Prvo peremo prozore, ima ih 6
 - Zatim prelazimo na tockove, ima ih 4
 - Sada su na redu sedista, njih 4
 - Gotovo, za sada.
Na redu je jedan Automobil.
 - Prvo peremo prozore, ima ih 6
 - Zatim prelazimo na tockove, ima ih 4
 - Sada su na redu sedista, njih 4
 - Gotovo, za sada.
```

Rezultat ponovo nije ispravan! Gde smo sada pogrešili?

Dinamičko vezivanje metoda

Zašto se program i dalje ponaša kao da se radi o objektu klase `Vozilo`, a ne o objektu klase `Kupe`? Problem se može predstaviti i na sasvim jednostavnom primeru koda:

```
Vozilo* v = new Kupe();
cout << v->Vrsta() << endl;
```

Šta će biti ispisano? Mi bismo želeli da bude ispisano „`Kupe`“, ali je očigledno da bi sa našim klasama rezultat bio „`Automobil`“. Nama je potrebno da se upotrebi metod klase `Kupe`, a upotrebljava se metod klase `Vozilo`. Problem je u neispravnom ustanovljavanju odgovarajuće implementacije metoda `Vrsta` koji će biti upotrebljen.

Izbor odgovarajućeg metoda se naziva *vezivanje metoda*. Vezivanje metoda može biti statičko i dinamičko. *Statičko vezivanje metoda* se izvodi u trenutku prevođenja programa, dok se *dinamičko vezivanje metoda* izvodi u trenutku pozivanja metoda, pri izvršavanju programa.

U slučaju statičkog vezivanja metoda, prevodilac se pri izboru odgovarajuće implementacije može rukovoditi samo lokalnim znanjem o podacima i metodima. Lokalno znanje obuhvata informaciju o tipu podatka i raspoloživim implementacijama. U ovom slučaju, prevodilac *zna* da je `v` pokazivač na objekat tipa `Vozilo`, i ništa više od toga. Činjenica da u našem primeru `v` pokazuje na objekat tipa `Kupe` se u opštem slučaju ne može jednostavno izvesti i upotrebiti. Radi ilustracije pogledajmo sledeći segment koda:

```
Vozilo* v;
if( rand() & 1 )
    v = new Kupe();
else
    v = new Vozilo();
cout << v->Vrsta() << endl;
```

Kojoj klasi pripada objekat na koji pokazuje `v` u poslednjem redu primera? Očigledno je da to ne može biti prepoznato u vreme prevođenja programa, bez obzira na to koliko informacija prevodilac ima o tekstu programa. Jedino što prevodilac može da upotrebi u zaključivanju je deklaracija tipa pokazivača `v`.

Pri statičkom vezivanju metoda *uvek* se bira implementacija metoda koja odgovara deklarisanom tipu objekta čiji se metod upotrebljava. Posledica je da primena statičkog vezivanja metoda onemogućava pisanje polimorfnih programa.

Odlaganjem odlučivanja o implementaciji metoda sve do trenutka njegovog pozivanja omogućava se izbor metoda koji najbolje odgovara tipu konkretnog objekta. Dinamičko vezivanje metoda podrazumeva da se pri pozivanju metoda najpre proverava tip objekta, zatim ustanovljava implemenatacija tog metoda koja najbolje odgovara konkretnom tipu objekta, pa tek onda poziva izabrana implementacija metoda.

Složenost mehanizma dinamičkog vezivanja metoda ima za posledicu da je dinamičko vezivanje metoda sporije od statičkog. Ta razlika je u praksi mnogo manja nego što bi se na osnovu opisa mehanizma dalo ustanoviti, ali izvesno postoje situacije u kojima se veoma značajno ispoljava. Neefikasnost dinamičkog vezivanja metoda se posebno uočava u slučaju jednostavnih metoda, čija implementacija je definisana u svega par redova koda. U iole složenijim metodima razlika u efikasnosti postaje manje značajna. Na veću efikasnost statički vezanih metoda utiče osobina programskog jezika C++ da se pozivanje statički vezanih metoda može prevoditi i umetanjem koda metoda umesto pozivanjem posebno prevedenih celina, što u slučaju dinamičkog vezivanja nije moguće.

Većina objektno orijentisanih programskih jezika počiva na dinamičkom vezivanju metoda. Štaviše, kod većine objektno orijentisanih programskih jezika uopšte ne postoji drugačiji način vezivanja metoda, već se uvek i isključivo primenjuje dinamičko vezivanje metoda. Pri oblikovanju programskog jezika C++, za razliku od većine drugih objektno orijentisanih programkih jezika, obezbeđivanje što veće efikasnosti programa je bio jedan od primarnih ciljeva. Zbog toga je statičko vezivanje metoda ne samo omogućeno nego i podrazumevano.

Da bi se mogao ostvarivati hijerarhijski polimorfizam neophodno je korišćenje dinamičkog vezivanja metoda. U programskom jeziku C++ dinamičko vezivanje metoda se eksplicitno deklariše pri definisanju metoda. Da bi vezivanje nekog metoda u hijerarhiji klasa bilo dinamičko, potrebno je u baznoj klasi hijerarhije ispred deklaracije metoda navesti ključnu reč `virtual`. Zbog toga se metodi koji se dinamički vezuju nazivaju i *virtualni metodi*.

U našem primeru je jasno da svaki od metoda klase `Vozilo` može biti redefinisan u nekoj od klasa naslednica. Zbog toga ćemo sve metode klase `Vozilo` proglasiti za virtualne:

```
class Vozilo
{
public:
    virtual string Vrsta() const
        { return "Automobil"; }

    virtual int BrojProzora() const
        { return 6; }

    virtual int BrojTockova() const
        { return 4; }

    virtual int BrojSedista() const
        { return 4; }
};
```

Sada program možemo prevesti i dobiti željeni rezultat:

```
Na redu je jedan Automobil.
 - Prvo peremo prozore, ima ih 6
 - Zatim prelazimo na tockove, ima ih 4
 - Sada su na redu sedista, njih 4
 - Gotovo, za sada.
Na redu je jedan Kupe.
 - Prvo peremo prozore, ima ih 4
 - Zatim prelazimo na tockove, ima ih 4
 - Sada su na redu sedista, njih 2
 - Gotovo, za sada.
```

Primetimo da eventualno navođenje ključne reči `virtual` ispred redefinicije metoda nema nikakvu funkciju, ali ne predstavlja sintaksnu grešku.

Dinamičko vezivanje destruktora

Veoma je značajno da primetimo da u našem programu, i praktično u svakom programu koji koristi hijerarhijski polimorfizam, postoji bar još jedan metod koji mora biti dinamički vezivan. Radi se o destruktoru.

Oprana vozila se brišu primenom izraza `delete`. Njegovo izvršavanje podrazumeva pozivanje destruktora i oslobađanje memorije. Zbog uobičajenih internih mehanizama implementacije postupaka alokacije i dealokacije memorije, memorija će u svakom slučaju biti ispravno oslobođena. Međutim, neće se uvek pozvati odgovarajući destruktor. Pozivanje destruktora se ni po čemu ne razlikuje od pozivanja ostalih metoda u našem primeru. Kao što je dolazilo do problema pri statičkom vezivanju ostalih metoda, tako može doći do problema i pri statičkom vezivanju destruktora, jer ono ima za posledicu da se uvek poziva samo destruktor bazne klase.

U baznoj klasi hijerarhije je neophodno uvek pisati virtualan destruktor. Ukoliko za destruktorom ne postoji stvarna potreba, već se piše samo da bismo naglasili da se vezuje dinamički, pišemo ga sa praznim telom. U tom slučaju nije neophodno pisanje ni konstruktora kopije ni operatora dodeljivanja.

Iako u našem primeru statičko vezivanje destruktora ne proizvodi neugodne posledice, jer izvedena klasa nema redefinisan destruktor, dobro je da se destruktor uvek piše na ispravan i bezbedan način. Objektima klase `Vozilo` nije neophodna nikakva deinicijalizacija, pa je telo

destruktora prazno. Zbog toga nam nisu potrebni ni konstruktor kopije ni operator dodeljivanja:

```
class Vozilo
{
public:
    virtual ~Vozilo()
        {}
...
};
```

Hijerarhijski polimorfizam

Hijerarhijski polimorfizam je pisanje funkcija, metoda i klasa primenjivih na sve klase neke hijerarhije klasa. Za razliku od parametarskog polimorfizma, koji podrazumeva da se napisani kod može primenjivati na sve tipove podataka koji podržavaju ponašanje upotrebljeno u kodu, kod hijerarhijskog polimorfizma se primenjivost koda eksplicitno ograničava na objekte koji pripadaju klasama date hijerarhije.

Hijerarhija klasa se može menjati dodavanjem novih klasa i menjanjem ponašanja postojećih klasa, a da se programi koji koriste hijerarhiju ne moraju menjati, sve dok se ne promeni interfejs bazne klase.

Da bi se mogao primenjivati hijerarhijski polimorfizam, bez obzira na to o kom objektno orijentisanom programskom jeziku se radi, moraju biti ispunjeni sledeći uslovi:

- Hijerarhijski polimorfizam se ostvaruje u odnosu na konkretnu izabranu hijerarhiju klasa, koja se izgrađuje primenom javnog nasleđivanja;
- Pri građenju hijerarhije se mora strogo voditi računa o poštovanju principa generalizacije i specijalizacije, tj. izvedena klasa mora predstavljati specijalan slučaj bazne klase, a bazna klasa uopštenje svojih izvedenih klasa;
- Kada je hijerarhija izgrađena primenom javnog nasleđivanja uz poštovanje principa generalizacije i specijalizacije, onda sve što važi za objekte bazne klase, važi i za objekte izvedenih klasa. Zbog toga se hijerarhijski polimorfizam ostvaruje pisanjem koda u kome se upotrebljavaju objekti deklarisani da pripadaju baznoj klasi hijerarhije, a zatim se taj kod primenjuje na proizvoljne objekte klasa te hijerarhije;

Primena hijerarhijskog polimorfizma u programskom jeziku C++ podrazumeva i ispunjavanje nekih dodatnih uslova:

- Hijerarhijski polimorfizam u programskom jeziku C++ se ostvaruje isključivo primenom pokazivača i referenci na tip bazne klase;
- Metodi bazne klase hijerarhije, koji se različito implementiraju u klasama hijerarhije, moraju biti implementirani primenom dinamičkog vezivanja metoda, tj. moraju biti deklarisani kao virtualni;

- Bazna klasa hijerarhije mora imati virtualan destruktor. Ako ne postoji stvarna potreba za deinicijalizacijom, on može imati prazno telo.

Korak 4 - Dodavanje klasa

Dopunimo hijerarhiju vozila još nekim klasama. Napišimo klase `Kamion`, `Kombi` i `Karavan`:

```
class Kamion : public Vozilo
{
public:
    string Vrsta() const
        { return "Kamion"; }

    int BrojProzora() const
        { return 4; }

    int BrojTockova() const
        { return 6; }

    int BrojSedista() const
        { return 2; }
};

//-----------------------------------------------------------------
//  Klasa Kombi
//-----------------------------------------------------------------
class Kombi : public Vozilo
{
public:
    string Vrsta() const
        { return "Kombi"; }

    int BrojProzora() const
        { return 10; }

    int BrojTockova() const
        { return 4; }

    int BrojSedista() const
        { return 10; }
};

//-----------------------------------------------------------------
//  Klasa Karavan
//-----------------------------------------------------------------
class Karavan : public Vozilo
{
public:
    string Vrsta() const
        { return "Karavan"; }

    int BrojProzora() const
        { return 8; }

    int BrojTockova() const
        { return 4; }

    int BrojSedista() const
        { return 4; }
};
```

Izmenimo glavnu funkciju:

```
main(){
    Perionica kodZike;

    kodZike.DodajVoziloURed( new Vozilo() );
    kodZike.DodajVoziloURed( new Kupe() );
    kodZike.DodajVoziloURed( new Kamion() );
    kodZike.DodajVoziloURed( new Kombi() );
    kodZike.DodajVoziloURed( new Karavan() );

    while( kodZike.ImaVozilaURedu() )
        kodZike.OperiPrvoVozilo();

    return 0;
}
```

Ako prevedemo i izvršimo program možemo biti zadovoljni jer sve ispravno funcioniše.

Nasleđivanje je specijalizacija

Bez obzira na to što sve ispravno funkcioniše, postoje određene neispravnosti kojima moramo posvetiti pažnju.

Definisanja hijerarhije klasa bi trebalo da počiva na definisanju odnosa nasleđivanja između klasa hijerarhije. Nasleđivanje mora da odgovara odnosu specijalizacije. To znači da izvedena klasa predstavlja *specijalan slučaj* bazne klase u smislu da sve što važi za baznu klasu važi i za izvedenu klasu, dok obrnuto ne važi. Shodno tome, bazna klasa predstavlja *generalizaciju* (ili *uopštenje*) svojih izvedenih klasa, tj. ponašanje bazne klase odgovara preseku ponašanja (tj. zajedničkom ponašanju) njenih izvedenih klasa.

Jedan od najčešćih i najboljih primera za ilustraciju ispravnog nasleđivanja je odnos između kvadrata i pravougaonika. Razmotrimo, primer, definisanje klasa geometrijskih likova. Neka klasa `Kvadrat` predstavlja pravougli jednakostranični paralelogram, sa stranicama paralelnim osama koordinatnog sistema (koji se jednoznačno opisuje položajem izabranog temena i dužinom jedne stranice), a klasa `Pravougaonik` pravougli paralelogram sa podudarnim naspramnim stranicama (koji se jednoznačno opisuje položajem i dužinama dveju stranica). Da li `Kvadrat` nasleđuje `Pravougaonik` ili `Pravougaonik` nasleđuje `Kvadrat`?

Ako bi se zaključivalo na osnovu strukture, tada bismo mogli da zaključimo da `Pravougaonik` nasleđuje `Kvadrat`, jer je za opisivanje pravougaonika potreban jedan podatak više (imamo dužine dveju stranica), pa se složenija struktura pravougaonika može definisati kao proširenje strukture kvadrata.

Ipak, pri nasleđivanju se kao jedini ispravan kriterijum ne koristi struktura nego *ponašanje*. Ako posmatramo osobine kvadrata i pravougaonika, možemo ustanoviti da je kvadrat specijalan slučaj pravougaonika – kvadrat je upravo pravougaonik čije sve stranice imaju istu dužinu. Zato je jedino ispravno da `Kvadrat` nasledi `Pravougaonik`.

Šta nam može pomoći u analizi odnosa nasleđivanja i proveri da li je hijerarhija klasa ispravno izgrađena? Pre svega dosledno poštovanje odnosa specijalizacije i generalizacije. Prilažemo nekoliko mogućih provera:

- Nasleđivanje se upotrebljava isključivo ako se svaki objekat izvedene klasa ponaša u potpunosti u skladu sa pravilima ponašanja objekata bazne klase, ali ima i neke specifične osobine, tj. ako izvedena klasa predstavlja specijalizaciju bazne klase;
- Ako izvedena klasa redefiniše metod bazne klase tako da ne proširuje njegovo ponašanje već ga u potpunosti menja, onda ona verovatno ne predstavlja specijalan slučaj bazne klase. Pod proširivanjem ponašanja podrazumevamo da se u telu metoda izvedene klase neposredno ili posredno upotrebljava odgovarajući metod bazne klase;
- Ako postoje sličnosti u ponašanju svih klasa izvedenih iz neke bazne klase, a koje nisu obuhvaćene baznom klasom, obično je dobro da se te sličnosti formalizuju prenošenjem odgovarajućeg ponašanja u baznu klasu.
- Ako postoje sličnosti u ponašanju nekih, ali ne svih klasa izvedenih iz neke bazne klase, a koje nisu obuhvaćene baznom klasom, obično je dobro da se napravi nova klasa između bazne klase i ovih klasa, koja bi obuhvatila te sličnosti.
- Ako u baznoj klasi imamo implementirane neke metode, potrebno je razmotriti da li će se oni redefinisati u izvedenim klasama. Ako neće, onda ti metodi možda ne bi trebalo da se dinamički vezuju.

Može se formulisati još mnogo pravila, ali već i ova koja smo naveli predstavljaju samo posledicu poštovanja odnosa specijalizacije i generalizacije između bazne i izvedene klase.

Da li se mogu uočiti neka neslaganja naše hijerarhije sa navedenim pravilima?

Osnovna neispravnost u našoj hijerarhiji je podrazumevanje da su objekti hijerarhije upravo automobili. Naravno da kamioni nisu automobili. Ni kombi nije automobil. Ovu neispravnost ispravljamo pravljenjem klase `Automobil`, kao nove naslednice klase `Vozilo`:

```
class Automobil : public Vozilo
{
public:
    virtual string Vrsta() const
        { return "Automobil"; }

    virtual int BrojProzora() const
        { return 6; }

    virtual int BrojTockova() const
        { return 4; }

    virtual int BrojSedista() const
        { return 4; }
};
```

Apstraktni metodi i klase

Iako smo napisali novu klasu, u klasi `Vozilo` i dalje ostaju implementacije metoda koje odgovaraju automobilima a ne i drugim klasama, tj. ne predstavljaju uopštenje ponašanja za sve vrste vozila. Postavlja se ne samo pitanje kako da implementiramo metode klase `Vozilo`, nego i da li smo uopšte u stanju da ih implementiramo tako da te implementacije

predstavljaju uopštenje za sve klase hijerarhije vozila? Pošto svaka klasa redefiniše neki od metoda, jasno je da ne postoji dovoljno uopštena implementacija.

Rešenje je da metode klase `Vozilo` uopšte ne implementiramo. Umesto implementacije metoda navodimo „`=0`“ čime označavamo da nije moguće definisati ponašanje metoda za klasu `Vozilo`. Metode koji nisu implementirani nazivamo *apstraktnim metodima* ili *potpuno (čisto, sasvim) virtualnim metodima*. Primetimo da destruktor ne sme biti apstraktan, jer se na kraju izvršavanja destruktora izvedene klase uvek automatski poziva i destruktor bazne klase.

Jasno je da bi pozivanje apstraktnih metoda predstavljalo neispravnu situaciju. Problem se rešava tako što u programskom jeziku C++ nije dozvoljeno praviti objekte klase koja ima bar jedan apstraktan metod. Takva klasa se naziva *apstraktna klasa*.

Apstraktne klase se upotrebljavaju za definisanje *interfejsa* hijerarhije klasa, tj. za definisanje imena i tipa metoda koje svaka izvedena klasa (neposredno ili posredno) mora implementirati. Pri definisanju hijerarhija klasa bazna klasa se najčešće piše kao apstraktna klasa. Tako ćemo i klasu `Vozilo` napisati kao apstraktnu klasu:

```
class Vozilo
{
public:
    virtual ~Vozilo()
        {}
    virtual string Vrsta() const = 0;
    virtual int BrojProzora() const = 0;
    virtual int BrojTockova() const = 0;
    virtual int BrojSedista() const = 0;
};
```

U glavnoj funkciji programa više ne možemo praviti objekat klase `Vozilo`. Umesto toga pravimo objekat klase `Automobil`:

```
    ...
    kodZike.DodajVoziloURed( new Automobil() );
    ...
```

Da bi se program mogao prevesti i izvršiti, potrebno je da klasu `Kupe` dopunimo metodom koji izračunava broj točkova. Ranije smo pretpostavljali da se nasleđuje implementacija iz klase `Vozilo`, ali ona više ne postoji. Dok ne implementiramo metod `BrojTockova`, klasa `Kupe` će se smatrati za apstraktnu:

```
class Kupe : public Vozilo
{
public:
...
    int BrojTockova() const
        { return 4; }
...
};
```

Mogli bismo dalje menjati i proširivati hijerarhiju klasa vozila, ali je i ovo sasvim dovoljno za prvi primer sa nasleđivanjem.

4.3 Rešenje

```
#include <iostream>
#include <queue>
using namespace std;

//-------------------------------------------------------------------
//  Klasa Vozilo
//-------------------------------------------------------------------
class Vozilo
{
public:
    virtual ~Vozilo()
        {}

    virtual string Vrsta() const = 0;
    virtual int BrojProzora() const = 0;
    virtual int BrojTockova() const = 0;
    virtual int BrojSedista() const = 0;
};

//-------------------------------------------------------------------
//  Klasa Automobil
//-------------------------------------------------------------------
class Automobil : public Vozilo
{
public:
    virtual string Vrsta() const
        { return "Automobil"; }

    virtual int BrojProzora() const
        { return 6; }

    virtual int BrojTockova() const
        { return 4; }

    virtual int BrojSedista() const
        { return 4; }
};

//-------------------------------------------------------------------
//  Klasa Kupe
//-------------------------------------------------------------------
class Kupe : public Vozilo
{
public:
    string Vrsta() const
        { return "Kupe"; }

    int BrojTockova() const
        { return 4; }

    int BrojProzora() const
        { return 4; }

    int BrojSedista() const
        { return 2; }
};
```

```
//---------------------------------------------------------------
//  Klasa Kamion
//---------------------------------------------------------------
class Kamion : public Vozilo
{
public:
    string Vrsta() const
        { return "Kamion"; }

    int BrojProzora() const
        { return 4; }

    int BrojTockova() const
        { return 6; }

    int BrojSedista() const
        { return 2; }
};

//---------------------------------------------------------------
//  Klasa Kombi
//---------------------------------------------------------------
class Kombi : public Vozilo
{
public:
    string Vrsta() const
        { return "Kombi"; }

    int BrojProzora() const
        { return 10; }

    int BrojTockova() const
        { return 4; }

    int BrojSedista() const
        { return 10; }
};

//---------------------------------------------------------------
//  Klasa Karavan
//---------------------------------------------------------------
class Karavan : public Vozilo
{
public:
    string Vrsta() const
        { return "Karavan"; }

    int BrojProzora() const
        { return 8; }

    int BrojTockova() const
        { return 4; }

    int BrojSedista() const
        { return 4; }
};

//---------------------------------------------------------------
// Klasa Perionica.
//---------------------------------------------------------------
class Perionica
{
```

```
public:
    Perionica()
        {}

    ~Perionica()
        { IsprazniRed(); }

    void DodajVoziloURed( Vozilo* v )
        { red.push(v); }

    bool ImaVozilaURedu() const
        { return !red.empty(); }

    void OperiPrvoVozilo()
        {
        if( ImaVozilaURedu() ){
            Vozilo* v = IzdvojiPrvoVozilo();
            cout
                << "Na redu je jedan "
                    << v->Vrsta() << "." << endl
                << " - Prvo peremo prozore, ima ih "
                    << v->BrojProzora() << endl
                << " - Zatim prelazimo na tockove, ima ih "
                    << v->BrojTockova() << endl
                << " - Sada su na redu sedista, njih "
                    << v->BrojSedista() << endl
                << " - Gotovo, za sada." << endl;
            delete v;
            }
        }

private:
    Perionica( const Perionica& );
    Perionica& operator=( const Perionica& );

    void IsprazniRed()
        {
        while( ImaVozilaURedu() )
            delete IzdvojiPrvoVozilo();
        }

    Vozilo* IzdvojiPrvoVozilo()
        {
        Vozilo* v = red.front();
        red.pop();
        return v;
        }

    queue<Vozilo*> red;
};

//-----------------------------------------------------------------
// Glavna funkcija programa demonstrira rad perionice.
//-----------------------------------------------------------------
main(){
    Perionica kodZike;

    kodZike.DodajVoziloURed( new Automobil() );
    kodZike.DodajVoziloURed( new Kupe() );
    kodZike.DodajVoziloURed( new Kamion() );
    kodZike.DodajVoziloURed( new Kombi() );
```

```
        kodZike.DodajVoziloURed( new Karavan() );
        while( kodZike.ImaVozilaURedu() )
            kodZike.OperiPrvoVozilo();
        return 0;
    }
```

4.4 Rezime

Preporučujemo da se za vežbu izmeni hijerarhija vozila tako da se definišu klase `GradskiAutomobil` (2 vrata) i `PutničkiAutomobil` (4 vrata) i da se sve klase, koje predstavljaju različite vrste automobila, implementiraju kao posredni naslednici apstraktne klase `Automobil`, koja bi i dalje nasleđivala klasu `Vozilo`.

Složeniji primeri hijerarhija klasa slede u narednim primerima.

5 - Igra „Život“

5.1 Zadatak

Matematičar John Conway je većinu svojih radova posvetio ozbiljnim matematičkim pitanjima. Na primer, 1967. godine je otkrio novu grupu koja se po njemu naziva *sazvežđe Conwaya*. Ipak, Conway je široj javnosti poznatiji po svojim aktivnostima u oblasti *zabavne* matematike. Samostalno ili u saradnji sa svojim kolegama, osmislio je veći broj matematičkih igara.

Najpoznatija od njegovih igara je, svakako, igra „Život“ (engl. *Life*) , koju je javnosti predstavio 1970. godine. Osmišljena je kao veoma pojednostavljena simulacija razvoja jedne populacije „živih“ jedinki. Njena jednostavnost čini je pristupačnom za eksperimentisanje, ali i tako pojednostavljen model ima veliki značaj, o čemu najbolje svedoči veliki broj naučnih i stručnih radova u kojima se razmatraju mogućnosti kako formalnog matematičkog zasnivanja i analiziranja toka igre, tako i njene praktične primene u realnim situacijama.

5.1.1 Pravila igre „Život“

Igra „Život“ počiva na relativno malom broju sasvim jednostavnih koncepata:

- životna sredina se predstavlja neograničenom „matricom“ *ćelija*;
- svaka ćelija može biti prazna ili sadržati tačno jednu jedinku;
- svaka ćelija ima tačno osam susednih ćelija;
- jedinke se smatraju susednim ako se nalaze u susednim ćelijama;
- raspored živih jedinki po ćelijama u jednom trenutku se naziva *konfiguracija*.

Conwey je težio da pravila zadovoljavaju nekoliko osnovnih uslova:

- moraju postojati konfiguracije koje su sposobne za vremenski neograničen opstanak i razvoj;

- ne bi trebalo da postoji konfiguracija za koju bi postojao jednostavan dokaz mogućnosti njenog neograničenog širenja;
- moraju postojati jednostavne početne konfiguracije koje tokom vremena rastu, značajno menjajući svoju veličinu, a zatim okončavaju svoju evoluciju na jedan od tri načina:
  - u potpunosti iščezavaju (bilo zbog prenaseljenosti ili iz nekog drugog razloga);
  - prelaze u *konstantnu stabilnu* konfiguraciju i u potpunosti prestaju da se menjaju;
  - prelaze u *dinamičku stabilnu* konfiguraciju koja se ciklično ponavlja, nakon konačnog broja generacija.

Istovremeno je težio da pravila budu što jednostavnija, kako bi igra bila podložna matematičkim analizama. Konačno, pravila igre je definisao na sledeći način:

- pravilo *rađanja*: u praznoj ćeliji se rađa nova jedinka ako ona ima tačno tri susedne jedinke;
- pravilo *preživljavanja*: jedinka preživljava ako ima dve ili tri susedne jedinke;
- pravilo *nepokretnosti*: jedinka čitav životni vek provodi u istoj ćeliji;
- pravilo *umiranja*: jedinka umire ako ima više od tri susedne jedinke (zbog prenaseljenosti) ili manje od dve susedne jedinke (zbog usamljenosti);
- primena pravila na neku ćeliju ili jedinku ne utiče na primenu pravila na susedne ćelije i jedinke, tj. sva pravila se primenjuju istovremeno za sve jedinke i ćelije jedne konfiguracije i kao rezultat daju novu *generaciju*.

5.1.2 Tekst zadatka

Napisati program koji predstavlja implementaciju igre „Život“.

Podržati proizvoljne dimenzije table za igru, čitanje početne konfiguracije iz datoteke, izračunavanje naredne generacije i zapisivanje konfiguracije u datoteci. Konfiguracije predstavljati na standardnom izlazu. Naredbe za izračunavanje naredne generacije, prekidanje igre i zapisivanje konfiguracije u datoteci prihvatati sa standardnog ulaza.

Osnovna matematička definicija igre podrazumeva da životna sredina nije ograničena (što je neophodno da bi svaka ćelija imala tačno osam susednih). Kako je neugodno na računaru implementirati beskonačnu matricu, životna sredina se obično predstavlja pravougaonom (ili kvadratnom) *tablom za igru* sa *torusnom topologijom* – smatra se da su ćelije prvog reda table susedne ćelijama poslednjeg reda table, kao i da su ćelije krajnje leve kolone susedne ćelijama krajnje desne kolone table. U skladu sa time, sve ugaone ćelije su međusobno susedne.

Cilj zadatka

Kroz ovaj primer ćemo:

- predstaviti nekoliko načina implentiranja matrica;
- utvrditi pisanje operatora za pristupanje elementima po indeksu;
- ponoviti kako se implementiraju čitanje i pisanje objekata;
- rešiti manje algoritamske probleme.

Pretpostavljena znanja

Za uspešno praćenje rešavanja ovog zadatka pretpostavlja se poznavanje:

- izraza `new` i `delete` za dinamičko pravljenje i uklanjanje objekata;
- pisanja konstruktora, destruktora, konstruktora kopije i operatora dodeljivanja;
- pisanje metoda i operatora za čitanje i pisanje.

5.2 Rešavanje zadatka

Rešavanje zadatka ćemo započeti pisanjem klase `Matrica`. Zatim ćemo, koristeći klasu `Matrica`, napisati klasu `Igra`:

Korak 1 - Definisanje interfejsa klase Matrica

Da bismo mogli implementirati igru „Život“ potrebno je da nekako predstavimo tablu za igru. Radi jednostavnosti, pretpostavićemo da jedinke nećemo posebno modelirati, jer je sasvim dovoljno da znamo koje ćelije table za igru su naseljene a koje nisu. Zbog toga bismo tablu za igru mogli predstavljati kao matricu logičkih vrednosti koje označavaju da li u ćeliji postoji jedinka. Međutim, mi nemamo na raspolaganju klasu koja ima strukturu matrice. Zbog toga ćemo najpre napisati odgovarajuću klasu. Pokazaćemo da se matrice mogu implementirati na više načina. Na kraju ćemo pokazati da se matrica može sasvim jednostavno implementirati i primenom standardne biblioteke.

Naša klasa bi trebalo da omogući pristupanje elementima na isti način kao da se radi o automatski alociranoj matrici fiksne veličine. Radi podsećanja navodimo primer definisanja matrice logičkih vrednosti sa 5 kolona i 4 vrste, i elementu u 2. koloni i 1. vrsti (brojimo od 0) dodeljujemo vrednost `true`:

```
bool matrica[5][4];
matrica[2][1] = true;
```

Problem nema trivijalno rešenje jer u programskom jeziku C++ nije moguće definisati operator poput:

```
... operator [][] (...)
```

Jedini način da se podrži odgovarajuće ponašanje jeste da se obezbedi da prva primena indeksnog operatora ima rezultat koji će podržati drugu primenu indeksnog operatora. Dakle, matrica bi trebalo da vrati čitavu kolonu, a kolona bi zatim trebalo da vrati konkretan element. Da bismo to postigli možemo napisati klasu `Kolona`, ali dovoljno je da se dosetimo da isti rezultat možemo dobiti i bez nove klase. Umesto toga, možemo da vratimo neki postojeći tip podataka koji *ume* da odreaguje na operator indeksiranja. Iz navedenog primera se vidi da bi to mogao da bude običan niz, ali on zahteva da dimenzije budu poznate u fazi prevođenja programa, što nama ne odgovara. Alternativa je da vratimo pokazivač na elemente odgovarajuće kolone. Ako bi matrica umela da vrati pokazivač na odgovarajuću kolonu, onda bi se primenom operatora indeksiranja na taj pokazivač mogao dobiti upravo traženi element matrice.

Ispostavlja se da je već prethodna minimalna analiza ciljnog interfejsa klase `Matrica` neposredno uslovila neke osobine interne organizacije podataka klase. Time se ponovo pokazuje da je dobro najpre analizirati interfejs klase, pa tek onda njenu implementaciju. Znamo da su nam potrebni konstruktor matrice datih dimenzija i operator za pristupanje elementima. Ako pretpostavimo da ćemo dinamički obezbeđivati potreban prostor za elemente, neophodni su nam i destruktor, konstruktor kopije i operator dodeljivanja:

```
class Matrica
{
public:
    Matrica( int visina, int sirina );
    Matrica( const Matrica& m );
    Matrica& operator = ( const Matrica& m );
    ~Matrica();

    bool* operator []( int i );
};
```

Pogledajmo malo bolje naš `operator[]`. Ako ga primenimo navodeći kao argument redni broj neke kolone, trebalo bi da se kao rezultat dobije pokazivač na elemente odgovarajuće kolone. Zatim, kada na rezultat primenimo postojeći operator indeksiranja za pokazivače, dobijamo konkretan element matrice. To možemo činiti bilo postupno, bilo neposredno:

```
Matrica m(5,4);
// neposredno
m[2][1] = true;
// postupno
bool* kolona = m[2];
kolona[1] = true;
```

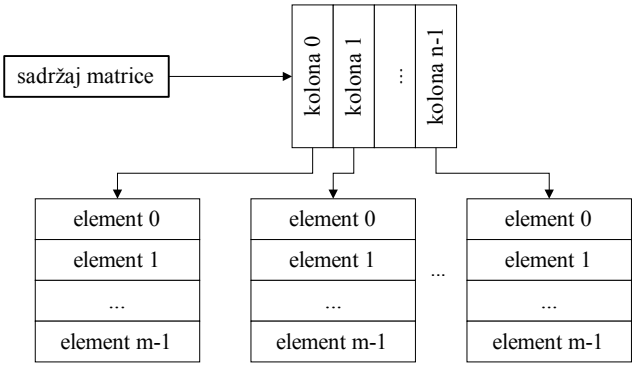
Bilo da elementima matrice pristupamo posredno ili neposredno, da bi bilo moguće odgovarajuće dodeljivanje vrednosti, neophodno je da rezultat primene operatora indeksiranja na pokazivač na kolonu bude *referenca* na element kolone.

Kao i u slučaju ranije upotrebe operatora indeksiranja (primer *3 - Dinamički nizovi*, str. 73), i ovde je potrebno definisati dva operatora: jedan konstantan, koji se koristiti za pristupanje elementima konstantne matrice, i drugi, nekonstantan, koji se koristi kako za čitanje tako i za menjenje elemenata nekonstantnih matrica. Interfejs klase `Matrica` sa oba operatora indeksiranja izgleda ovako:

```
class Matrica
{
public:
    Matrica( int visina, int sirina );
    Matrica( const Matrica& m );
    Matrica& operator = ( const Matrica& m );
    ~Matrica();
    bool* operator []( int i );
    const bool* operator []( int i ) const;
};
```

Korak 2 - Implementacija klase Matrica

Već smo uočili da zbog implementacije operatora indeksiranja moramo biti u stanju da vratimo pokazivač na kolonu matrice. Kao moguće rešenje za definisanje interne strukture matrice, a koje je u skladu sa takvim zahtevom, nameće se čuvanje matrice kao niza kolona. Svaku kolonu možemo predstaviti pokazivačem na njen prvi element, a kompletan sadržaj matrice kao dinamički niz pokazivača na kolone, tj. pokazivača na prve elemente kolona (Slika 3).



Slika 3: Sadržaj matrice predstavljen kao niz kolona

Tip pokazivača na kolonu je `bool*`, a tip pokazivača na niz pokazivača na kolone je `bool**`:

```
bool** _Kolone;
```

Konstruktor je dužan da (1) alocira prostor za niz pokazivača na kolone, a zatim i da (2) alocira jednu po jednu kolonu i (3) ažurira pokazivače na kolone tako da ukazuju na alociran prostor. U opštem slučaju nije neophodno da pri konstrukciji gubimo vreme na inicijalizovanje pojedinačnih elemenata matrice, pa to nećemo ni činiti:

```
class Matrica
{
public:
    Matrica( int visina, int sirina )
        {
        _Kolone = new bool*[sirina];
        for( int i=0; i<sirina; i++ )
            _Kolone[i] = new bool[visina];
        }
...
private:
    bool** _Kolone;
};
```

Sve što konstruktor rezerviše, destruktor je dužan da oslobodi. Destruktor mora osloboditi kako niz pokazivača na kolone tako i svaku pojedinačnu kolonu:

```
class Matrica
{
public:
...
    ~Matrica()
        {
        for( int i=0; i<_Sirina; i++ )
            delete [] _Kolone[i];
        delete [] _Kolone;
        }
...
};
```

Da bi to bilo moguće, matrica mora da zna koliko ima kolona. Zbog toga dodajemo podatak o širini matrice i obezbeđujemo njegovo inicijalizovanje u konstruktoru. Kako imamo destruktor, neophodno je napisati i konstruktor kopije i operator dodeljivanja. Međutim, da bismo uspešno iskopirali matricu, moramo znati ne samo njenu širinu, nego i visinu. Novi članovi podaci i izmenjeni konstruktor izgledaju ovako:

```
class Matrica
{
public:
...
    Matrica( int visina, int sirina )
        : _Sirina(sirina), _Visina(visina)
        {
        _Kolone = new bool*[_Sirina];
        for( int i=0; i<_Sirina; i++ )
            _Kolone[i] = new bool[_Visina];
        }
...
```

```
private:
    bool**  _Kolone;
    int     _Sirina;
    int     _Visina;
};
```

Pri pisanju konstruktora kopije i operatora dodeljivanja moramo iskopirati svaki pojedinačni element, pa je osim alokacije kolona potrebno i kopiranje njihovog sadržaja. Pri pisanju operatora dodeljivanja sledimo šablon predstavljen u prethodnim primerima (videti primer *Lista, Korak 4 - Kopiranje objekata*, na str. 51):

```
class Matrica
{
public:
...
    Matrica( const Matrica& m )
        {
        _Sirina = m._Sirina;
        _Visina = m._Visina;
        _Kolone = new bool*[_Sirina];
        for( int i=0; i<_Sirina; i++ ){
            _Kolone[i] = new bool[_Visina];
            for( int j=0; j<_Visina; j++ )
                _Kolone[i][j] = m._Kolone[i][j];
            }
        }

    Matrica& operator = ( const Matrica& m )
        {
        if( this != &m ){
            for( int i=0; i<_Sirina; i++ )
                delete [] _Kolone[i];
            delete [] _Kolone;
            _Sirina = m._Sirina;
            _Visina = m._Visina;
            _Kolone = new bool*[_Sirina];
            for( int i=0; i<_Sirina; i++ ){
                _Kolone[i] = new bool[_Visina];
                for( int j=0; j<_Visina; j++ )
                    _Kolone[i][j] = m._Kolone[i][j];
                }
            }
        return *this;
        }
...
};
```

Kako imamo značajna ponavljanja koda u destruktoru i operatoru dodeljivanja, odnosno u konstruktoru kopije i operatoru dodeljivanja, izdvojićemo ponovljeni kod u privatne metode `init` i `deinit`:

```
class Matrica
{
...
```

```
private:
    void deinit()
        {
        for( int i=0; i<_Sirina; i++ )
            delete [] _Kolone[i];
        delete [] _Kolone;
        }

    void init( const Matrica& m )
        {
        _Sirina = m._Sirina;
        _Visina = m._Visina;
        _Kolone = new bool*[_Sirina];
        for( int i=0; i<_Sirina; i++ ){
            _Kolone[i] = new bool[_Visina];
            for( int j=0; j<_Visina; j++ )
                _Kolone[i][j] = m._Kolone[i][j];
            }
        }
...
};
```

Kopiranje pojedinačnih elemenata se može malo ubrzati primenom fukcije za kopiranje blokova memorije. Umesto:

```
            for( int j=0; j<_Visina; j++ )
                _Kolone[i][j] = m._Kolone[i][j];
```

koristimo funkciju `copy` standardne biblioteke:

```
        copy( m._Kolone[i], m._Kolone[i] + _Visina, _Kolone[i] );
```

Sada možemo uprostiti destruktor, konstruktor kopije i operator dodeljivanja primenjujući nove metode:

```
class Matrica
{
public:
...
    ~Matrica()
        { deinit(); }

    Matrica( const Matrica& m )
        { init(m); }

    Matrica& operator = ( const Matrica& m )
        {
        if( this != &m ){
            deinit();
            init(m);
            }
        return *this;
        }
...
};
```

Preostaje nam da napišemo operatore indeksiranja. Primetimo da nema razloga da se implementacije ova dva operatora razlikuju. Naravno, ostaje nam razlika u tipu argumenta (tj. objekta) i rezultata, koja je istaknuta već pri definisanju interfejsa:

```
class Matrica
{
public:
...
    bool* operator []( int i )
        { return _Kolone[i]; }
    const bool* operator []( int i ) const
        { return _Kolone[i]; }
...
};
```

Mogla bi se staviti zamerka da smo propustili priliku da proverimo opseg indeksa. Međutim, pitanje je koliko bi ta provera imala smisla, jer se odgovarajuća provera izvesno ne izvodi prilikom primene operatora za pristupanje određenom elementu kolone. Kako nema mnogo smisla proveravati redni broj kolone, a ne proveravati redni broj vrste, ostavićemo našu klasu `Matrica` bez kontrole indeksa pri primeni operatora indeksiranja. Ukoliko bi provera opsega bila neophodna, tada bi se umesto operatora indeksiranja morao napisati i koristiti odgovarajući metod sa dva argumenta.

U praksi nam može biti od značaja podrška za tzv. *prazne* matrice. U skladu sa time menjamo metode koji prave i oslobađaju matricu: ukoliko je matrica prazna (širina ili visina nisu pozitivni), vodićemo računa da pokazivač na kolone bude 0; dopustićemo i upotrebu konstruktora bez argumenata, u kom slučaju ćemo praviti praznu matricu.

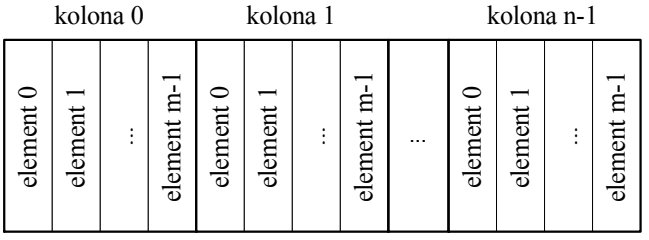
```
class Matrica
{
public:
    Matrica( int visina =0, int sirina =0 )
        : _Sirina(sirina), _Visina(visina)
        {
        if( _Sirina>0 && _Visina>0 ){
            _Kolone = new bool*[_Sirina];
            for( int i=0; i<_Sirina; i++ )
                _Kolone[i] = new bool[_Visina];
            }
        else
            _Kolone = 0;
        }
...
private:
    void deinit()
        {
        if( _Kolone ){
            for( int i=0; i<_Sirina; i++ )
                delete [] _Kolone[i];
            delete [] _Kolone;
            }
        }
```

```
        void init( const Matrica& m )
            {
            _Sirina = m._Sirina;
            _Visina = m._Visina;
            if( _Sirina>0 && _Visina>0 ){
                _Kolone = new bool*[_Sirina];
                for( int i=0; i<_Sirina; i++ ){
                    _Kolone[i] = new bool[_Visina];
                    copy(
                        m._Kolone[i], m._Kolone[i] + _Visina,
                        _Kolone[i]
                        );
                    }
                }
            else
                _Kolone = 0;
            }
    ...
    };
```

Korak 3 - Promena interne strukture klase Matrica

Jedna slabost predstavljene interne strukture klase `Matrica` je u činjenici da inicijalizacija počiva na potencijalno velikom broju alokacija. Sve dok ukupna veličina matrice ne postane približna maksimalnoj raspoloživoj veličini memorije, takav način alociranja je manje efikasan od alociranja jednog velikog bloka memorije, koji bi sadržao sve elemente matrice. Druga slabost predstavljene strukture je u činjenici da se do svakog elementa dolazi nakon čitanja i upotrebe vrednosti dva pokazivača, tj. preko dvostruke indirekcije.

Podsetimo se da nam je važno da svaka kolona ima formu niza elemenata, kako bi se pokazivač na prvi element kolone mogao vratiti kao rezultat operatora indeksiranja. Primetimo da nas ništa ne ograničava da to mora da bude samostalan niz, te da bi bez ikakvih neugodnih posledica to mogao da bude podniz nekog većeg niza. Ako bismo sve nizove koji predstavljaju kolone poređali u jedan veliki niz, tada bismo sve elemente matrice mogli da predstavimo takvim velikim nizom (Slika 4).



Slika 4: Sadržaj matrice predstavljen pomoću jednog niza

Ovakva organizacija podataka obezbeđuje nešto viši nivo efikasnosti, mada obično ne u dovoljnoj meri da bi nam to bio osnovni motiv za njenu primenu. Ovakav način organizovanja podataka predstavljamo pre svega da bi se čitaoci upoznali sa različitim

prilazima rešavanju problema. Izmenimo sada klasu `Matrica` u skladu sa novom predstavljenom strukturom:

```
class Matrica
{
public:
    //-----------------------------
    //  Konstruktor i ostali potrebni metodi
    Matrica( int visina =0, int sirina =0 )
        : _Sirina(sirina), _Visina(visina)
        {
        if( _Sirina>0 && _Visina>0 )
            _Podaci = new bool[ _Sirina * _Visina ];
        else
            _Podaci = 0;
        }

...
    //-----------------------------
    //  pristupanje elementima
    bool* operator []( int i )
        { return &_Podaci[i*_Visina]; }
    const bool* operator []( int i ) const
        { return &_Podaci[i*_Visina]; }

private:
    //-----------------------------
    //  pomocni metodi
    void deinit()
        { delete [] _Podaci; }

    void init( const Matrica& m )
        {
        _Sirina = m._Sirina;
        _Visina = m._Visina;
        if( _Sirina>0 && _Visina>0 ){
            _Podaci = new bool[ _Sirina * _Visina ];
            copy(
                m._Podaci, m._Podaci + _Sirina*_Visina,
                _Podaci
                );
            }
        else
            _Podaci = 0;
        }

    //-----------------------------
    //  podaci clanovi
    bool*   _Podaci;
    int     _Sirina;
    int     _Visina;
};
```

Promene u konstruktoru i metodima `init` i `deinit` su neposredna posledica promene strukture. Sada se izvode alokacija i dealokacija tačno po jednog niza. Kopiranje elemenata matrice se takođe izvodi u jednom koraku.

Obradimo pažnju na operatore indeksiranja. Nama je potrebno da izračunamo pokazivač na `i`-tu kolonu. Pokazivač na `i`-tu kolonu je ujedno i pokazivač na 0-ti element te kolone, pa odgovara adresi elementa 0-te vrste `i`-te kolone. Ako znamo da do `i`-te kolone moramo preskočiti čitavih `i` kolona (redni brojevi idu od 0!), jasno je da se početak `i`-te kolone nalazi neposredno iza `i*_Visina` elemenata. U skladu sa načinom indeksiranja elemenata u programskom jeziku C++ (počev od 0, a ne od 1), element 0-te vrste `i`-te kolone je element niza `_Podaci` sa indeksom `i*_Visina`, a pokazivač na `i`-tu kolonu je upravo njegova adresa:

```
&_Podaci[i*_Visina]
```

Ekvivalentan rezultat može se dobiti i izrazom:

```
_Podaci + i*_Visina
```

Korak 4 - Alternativna definicija klase Matrica

U jednom od prethodnih primera definisali smo šablon klase `Niz` (primer *3 - Dinamički nizovi*, na strani 73.). Ako matricu predstavljamo kao niz kolona, onda za čuvanje elemenata matrice možemo upotrebiti tip `Niz<Niz<bool>>`. Kako je `Niz` definisan sa svim potrebnim metodima (destruktor, konstruktor kopije, operator dodeljivanja) nama preostaje samo trivijalna implementacija konstruktora i operatora indeksiranja. Napominjemo da je ova implementacija manje efikasna nego prethodno predstavljene, mada visok kvalitet implementacija mehanizama za rad sa šablonima u prevodiocima doprinosi da razlika ne bude posebno značajna.

```
class Matrica
{
public:
    Matrica( int visina =0, int sirina =0 )
        : _Kolone(sirina)
        {
        for( int i=0; i<sirina; i++ )
            _Kolone[i].PromeniVelicinu(visina);
        }

    Niz<bool>& operator []( int i )
        { return _Kolone[i]; }
    const Niz<bool>& operator []( int i ) const
        { return _Kolone[i]; }

private:
    Niz< Niz< bool > >  _Kolone;
};
```

U konstruktoru, najpre, pri inicijalizaciji niza kolona navodimo celobrojni argument koji određuje veličinu niza, tj. broj kolona. Zatim primenom metoda `PromeniVelicinu` menjamo veličinu svake pojedinačne kolone.

Kako je sada matrica ponovo predstavljena kao niz kolona (kao i u prvom slučaju), operatori indeksiranja su dužni da vraćaju te kolone. U prvoj implementaciji klase `Matrica` kolone su vraćane posredstvom pokazivača na njihov prvi (tj. 0-ti) element. Kako je sada svaka kolona predstavljena po jednim objektom tipa `Niz<bool>`, oba metoda indeksiranja

vraćaju reference na odgovarajuće objekte. Naravno, u skladu sa predstavljenim pravilom, konstantan operator mora da vrati referencu na konstantnu kolonu.

Pre ovakve definicije klase `Matrica` neophodno je uključiti definiciju šablona klase `Niz`:

```
#include "DinamickiNiz.h"
```

Upotreba šablona klase vector

Ako poznajemo standardnu biblioteku programskog jezika C++, znamo da se za rad sa dinamičkim nizovima podataka upotrebljava klasa (preciznije, šablon klase) `vector`. Umesto našeg šablona klase `Niz` možemo upotrebiti šablon klase `vector`:

```
class Matrica
{
public:
    Matrica( int visina =0, int sirina =0 )
        : _Kolone(sirina)
        {
        for( int i=0; i<sirina; i++ )
            _Kolone[i].resize(visina);
        }

    vector<bool>& operator []( int i )
        { return _Kolone[i]; }
    const vector<bool>& operator []( int i ) const
        { return _Kolone[i]; }

private:
    vector< vector< bool > >    _Kolone;
};
```

Da bi uspelo prevođenje koda u kome se upotrebljavaju vektori, potrebno je uključiti zaglavlje `vector`:

```
#include <vector>
using namespace std;
```

Korak 5 - Klasa Igra

Neka osnovne funkcije igre obavlja klasa `Igra`. Za početak, obezbedićemo čitanje konfiguracije iz toka i zapisivanje konfiguracije u tok. Za čuvanje konfiguracije ćemo koristiti objekat klase `Matrica`:

```
    Matrica _Konfiguracija;
```

Pretpostavimo da se konfiguracija zapisuje tako što se u prvom redu zapišu širina i visina konfiguracije, a zatim se zapisuju vrste, po jedna u redu. Prazne ćelije se predstavljaju tačkom, a pune slovom '`X`'. Pre svake vrste postoji znak za novi red. Iza poslednje vrste se ne zapisuje znak za novi red, kako se ne bi komplikovalo eventualno čitanje podataka koji u toku slede iza konfiguracije.

U skladu sa ranije viđenim principima podržavanja rada sa tokovima, pišemo metode `Citaj` i `Pisi` klase `Igra`, a odgovarajuće operatore pišemo van klase:

```
class Igra
{
public:
    void Pisi( ostream& ostr ) const
        {
        int s = _Konfiguracija.Sirina();
        int v = _Konfiguracija.Visina();
        ostr << s << ' ' << v;
        for( int i=0; i<v; i++ ){
            ostr << endl;
            for( int j=0; j<s; j++ )
                ostr << (_Konfiguracija[j][i] ? 'X' : '.');
            }
        }
    void Citaj( istream& istr )
        {
        int s, v;
        istr >> s >> v;
        Matrica m(v,s);
        for( int i=0; i<v; i++ )
            for( int j=0; j<s; j++ ){
                char c;
                istr >> c;
                m[j][i] = c=='X';
                }
        _Konfiguracija = m;
        }
private:
    Matrica _Konfiguracija;
};
ostream& operator << ( ostream& ostr, const Igra& t )
{
    t.Pisi( ostr );
    return ostr;
}
istream& operator >> ( istream& istr, Igra& t )
{
    t.Citaj( istr );
    return istr;
}
```

Primetimo da ne možemo uspešno zapisivati konfiguraciju ako ne znamo koje su njene dimenzije. Zato ćemo klasi `Matrica` dodati javne metode za izračunavanje širine i visine. U slučaju prve dve verzije klase, oni izgledaju ovako:

```
    int Sirina() const
        { return _Sirina; }
    int Visina() const
        { return _Visina; }
```

U slučaju implementacije pomoću vektora ili našeg šablona klase `Niz`, implementacija ovih metoda je nešto drugačija, jer nisu neophodni odgovarajući podaci, već se koriste metodi vektora (ili niza):

```
    int Sirina() const
        { return _Kolone.size(); }
    int Visina() const
        { return _Kolone.size() ? _Kolone[0].size() : 0; }
```

Pišemo funkciju `main` tako da pročita konfiguraciju iz datoteke i ispiše je na standardnom izlazu:

```
main()
{
    Igra igra;
    ifstream f("konfiguracija.dat");
    f >> igra;
    if( !f )
        return 1;

    cout << igra << endl;
    return 0;
}
```

Za uspešno prevođenje je potrebno uključiti zaglavlja `iostream` i `fstream`.

Da bismo isprobali napisano, pravimo tekstualnu datoteku `konfiguracija.dat` sa sledećim sadržajem:

```
15 10
...............
...............
...XX..........
..X..X.........
..X..X.........
...XX..........
...............
........XX.....
........X.X....
........X......
```

Korak 6 - Tok igre

Igra se odvija tako što se od korisnika očekuje da upisuje naredbe na osnovu kojih se zatim postupa:

- ako korisnik upiše slovo `C` i zatim naziv datoteke, iz datoteke se čita konfiguracija;
- ako upiše slovo `P` i naziv datoteke, konfiguracija se upisuje u datoteku.
- ako upiše slovo `G`, računa se i ispisuje naredna generacija;
- ako upiše slovo `K`, završava se igra.

Potrebno ponašanje ćemo obuhvatiti metodom `Odigravanje`:

```
class Igra
{
public:
    void Odigravanje()
        {
```

```
            cout << "Upisite: " << endl
                 << "   c <naziv datoteke> - "
                 << "za citanje konfiguracije iz datoteke" << endl
                 << "   p <naziv datoteke> - "
                 << "za zapisivanje konfiguracije u datoteci" << endl
                 << "   g                  - "
                 << "za izracunavanje naredne generacije" << endl
                 << "   k                  - za kraj" << endl;

            char c;
            do  {
                Pisi( cout );
                cout << endl;
                cin >> c;
                c = tolower(c);
                switch(c){
                    case 'c':
                        CitanjeKonfiguracije();
                        break;
                    case 'p':
                        PisanjeKonfiguracije();
                        break;
                    case 'g':
                        IzracunavanjeGeneracije();
                        break;
                    }
                } while( c!='k' );
            }
    ...
    };
```

Funkcija `tolower` standardne biblioteke izračunava malo slovo.

Slede metodi `CitanjeKonfiguracije` i `PisanjeKonfiguracije`. Oba metoda obuhvataju čitanje naziva datoteke sa standardnog ulaza:

```
    class Igra
    {
    ...
    private:
        void CitanjeKonfiguracije()
            {
            string s;
            cin >> s;
            ifstream f(s.c_str());
            Citaj(f);
            }

        void PisanjeKonfiguracije()
            {
            string s;
            cin >> s;
            ofstream f(s.c_str());
            Pisi(f);
            }
    ...
    };
```

Primetimo da je za čitanje naziva datoteke upotrebljen objekat klase `string` i odgovarajući operator čitanja iz toka. Prednosti klase `string` u odnosu na obične nizove znakova su višestruke. Osnovni motiv za primenu na ovom mestu jeste dinamičko prilagođavanje veličine niske pročitanom nazivu datoteke, pa nema potrebe da unapred pretpostavljamo maksimalnu dužinu naziva. Ipak, to donosi i neke teškoće, jer se pri konstrukciji i otvaranju datotečnih tokova mogu navoditi isključivo obični pokazivači na nizove znakova. Objekti klase `string` izračunavaju pokazivač na sadržani niz znakova metodom `c_str`. Klasa `string` će biti intenzivnije upotrebljavana i detaljnije prodiskutovana u narednim primerima (videti odeljak *10.11 String*, na strani 394).

Primetimo da će operator `>>`, bez obzira da li radimo sa klasom `string` ili sa nizovima karaktera, pročitati samo prvu reč (tj. niz znakova do prvog praznog znaka). Zbog toga ovako napisan program neće moći da čita i piše datoteke u čijem imenu ima više reči.

Da bismo isprobali napisano, pišemo prazan metod za izračunavanje naredne generacije i odgovarajuću funkciju `main`:

```
class Igra
{
private:
    void IzracunavanjeGeneracije()
        {}
...
};

main()
{
    Igra igra;
    ifstream f("konfiguracija.dat");
    f >> igra;
    igra.Odigravanje();

    return 0;
}
```

Korak 7 - Izračunavanje naredne generacije

Pri izračunavanju naredne generacije moramo imati u vidu da se sva pravila primenjuju *istovremeno*, tj. da primena pravila na neku ćeliju ili jedinku ne sme uticati na primenu pravila na susedne jedinke i ćelije. Da se pri izračunavanju naredne generacije ne bi menjala aktuelna generacija, neophodno je da se rezultat izračunavanja nove generacije upisuje u posebnu matricu.

Najpre pišemo pomoćni metod `BrojSuseda` koji izračunava koliko ćelija u `k`-toj koloni i `v`-toj vrsti ima susednih jedinki. Najjednostavniji način je da „prođemo“ kroz podmatricu ćelija, dimenzija 3x3, u čijem središtu je posmatrana ćelija: svaki put kada naiđemo na neku jedinku, povećavamo brojač. Pri tome ne proveravamo samu ćeliju čije susede brojimo:

```
class Igra
{
private:
    int BrojSuseda( int k, int v ) const
        {
        int n=0;
        for( int i=-1; i<2; i++ )
            for( int j=-1; j<2; j++ )
                if( (i||j) && _Konfiguracija[k+i][v+j] )
                    n++;
        return n;
        }
...
};
```

Alternativno, umesto da u svakom prolazu proveravamo da li je u pitanju upravo centralna ćelija podmatrice, koju ne bi trebalo da obradimo, možemo najpre i nju obraditi, a zatim, nakon što prođemo kroz sve ćelije, proverimo ponovo da li je u njoj jedinka i ako jeste, smanjimo brojač za jedan:

```
class Igra
{
private:
    int BrojSuseda( int k, int v ) const
        {
        int n=0;
        for( int i=-1; i<2; i++ )
            for( int j=-1; j<2; j++ )
                if( _Konfiguracija[k+i][v+j] )
                    n++;
        if( _Konfiguracija[k][v] )
            n--;
        return n;
        }
...
};
```

Ponuđeno rešenje nije kompletno. Kako naša tabla za igru ima topologiju torusa, pri posmatranju susednih tačaka je potrebno uzeti u obzir eventualno prelaženje granica matrice. Ako znamo širinu, tada prelaženje desne granice matrice uzimamo u obzir računanjem ostatka pri celobrojnom deljenju širinom:

```
(k+i)%sirina
```

Međutim, ako se izađe ulevo (tj. ako je `k+i<0`), tada je vrednost prethodnog izraza negativna pa on ne predstavlja korektno rešenje. Zato se pre računanja ostatka dodaje širina:

```
(k+i+sirina)%sirina
```

Tako su na odgovarajući način obuhvaćena oba slučaja. Nakon što na sličan način podržimo i izračunavanje susedne vrste, dobijamo ispravnu implementaciju metoda:

```
class Igra
{
private:
    int BrojSuseda( int k, int v ) const
        {
        int sirina = _Konfiguracija.Sirina();
        int visina = _Konfiguracija.Visina();
        int n=0;
        for( int i=-1; i<2; i++ )
            for( int j=-1; j<2; j++ )
                if( _Konfiguracija[(k+i+sirina)%sirina]
                                  [(v+j+visina)%visina] )
                    n++;
        if( _Konfiguracija[k][v] )
            n--;
        return n;
        }
...
};
```

Primenjujemo pravila na sve ćelije i izračunamo novu generaciju. Na kraju, novu generaciju proglašavamo za aktuelnu:

```
class Igra
{
private:
...
    void IzracunavanjeGeneracije()
        {
        int sirina = _Konfiguracija.Sirina();
        int visina = _Konfiguracija.Visina();
        Matrica nova( visina, sirina );
        for( int i=0; i<sirina; i++ )
            for( int j=0; j<visina; j++ ){
                int bs = BrojSuseda(i,j);
                nova[i][j] =
                    bs == 3
                    || (bs == 2 && _Konfiguracija[i][j]);
                }
        _Konfiguracija = nova;
        }
...
};
```

Analiza rešenja može lako da pokaže da ima prostora za dalja unapređenja, ali njima se ovde nećemo baviti. Dalja nadgradnja ovog rešenja, uz odovarajuću dopunu i samog teksta zadatka, sledi u jednom od narednih primera.

5.3 Rešenje

```
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;
```

```
//---------------------------------------------------------------
//  Klasa Matrica
//---------------------------------------------------------------
class Matrica
{
public:
    //-----------------------------
    //  Konstruktor i ostali potrebni metodi
    Matrica( int visina =0, int sirina =0 )
        : _Kolone(sirina)
        {
        for( int i=0; i<sirina; i++ )
            _Kolone[i].resize(visina);
        }

    //-----------------------------
    //  pristupanje elementima
    vector<bool>& operator []( int i )
        { return _Kolone[i]; }
    const vector<bool>& operator []( int i ) const
        { return _Kolone[i]; }

    int Sirina() const
        { return _Kolone.size(); }
    int Visina() const
        { return _Kolone.size() ? _Kolone[0].size() : 0; }

private:
    //-----------------------------
    //  podaci clanovi
    vector< vector< bool > >    _Kolone;
};

//---------------------------------------------------------------
//  Klasa Igra
//---------------------------------------------------------------
class Igra
{
public:
    //-----------------------------
    //  glavni metod za igranje
    void Odigravanje()
        {
        cout << "Upisite: " << endl
             << "   c <naziv datoteke> - "
             << "za citanje konfiguracije iz datoteke" << endl
             << "   p <naziv datoteke> - "
             << "za zapisivanje konfiguracije u datoteci" << endl
             << "   g                  - "
             << "za izracunavanje naredne generacije" << endl
             << "   k                  - za kraj" << endl;

        char c;
        do  {
            Pisi( cout );
            cout << endl;
            cin >> c;
            c = tolower(c);
```

```
                switch(c){
                    case 'c':
                        CitanjeKonfiguracije();
                        break;
                    case 'p':
                        PisanjeKonfiguracije();
                        break;
                    case 'g':
                        IzracunavanjeGeneracije();
                        break;
                    }
                } while( c!='k' );
            }

        //-----------------------------
        //  pisanje i citanje
        void Pisi( ostream& ostr ) const
            {
            int s = _Konfiguracija.Sirina();
            int v = _Konfiguracija.Visina();
            ostr << s << ' ' << v;
            for( int i=0; i<v; i++ ){
                ostr << endl;
                for( int j=0; j<s; j++ )
                    ostr << (_Konfiguracija[j][i] ? 'X' : '.');
                }
            }

        void Citaj( istream& istr )
            {
            int s, v;
            istr >> s >> v;
            Matrica m(v,s);
            for( int i=0; i<v; i++ )
                for( int j=0; j<s; j++ ){
                    char c;
                    istr >> c;
                    m[j][i] = c=='X';
                    }
            _Konfiguracija = m;
            }

    private:
        //-----------------------------
        //  pomocni metodi
        int BrojSuseda( int k, int v ) const
            {
            int sirina = _Konfiguracija.Sirina();
            int visina = _Konfiguracija.Visina();
            int n=0;
            for( int i=-1; i<2; i++ )
                for( int j=-1; j<2; j++ )
                    if( _Konfiguracija[(k+i+sirina)%sirina]
                                      [(v+j+visina)%visina] )
                        n++;
            if( _Konfiguracija[k][v] )
                n--;
```

```
            return n;
            }
        void CitanjeKonfiguracije()
            {
            string s;
            cin >> s;
            ifstream f(s.c_str());
            Citaj(f);
            }
        void PisanjeKonfiguracije()
            {
            string s;
            cin >> s;
            ofstream f(s.c_str());
            Pisi(f);
            }
        void IzracunavanjeGeneracije()
            {
            int sirina = _Konfiguracija.Sirina();
            int visina = _Konfiguracija.Visina();
            Matrica nova( visina, sirina );
            for( int i=0; i<sirina; i++ )
                for( int j=0; j<visina; j++ ){
                    int bs = BrojSuseda(i,j);
                    nova[i][j] =
                        bs == 3
                        || (bs == 2 && _Konfiguracija[i][j]);
                    }
            _Konfiguracija = nova;
            }
        //-----------------------------
        //  podaci clanovi
        Matrica _Konfiguracija;
};
ostream& operator << ( ostream& ostr, const Igra& t )
{
    t.Pisi( ostr );
    return ostr;
}
istream& operator >> ( istream& istr, Igra& t )
{
    t.Citaj( istr );
    return istr;
}
//-----------------------------------------------------------------
// Glavna funkcija programa
//-----------------------------------------------------------------
main()
{
    Igra igra;
    ifstream f("konfiguracija.dat");
    f >> igra;
    igra.Odigravanje();
```

```
        return 0;
    }
```

5.4 Rezime

Pri izračunavanju nove generacije upotrebljena je pomoćna matrica čije su dimenzije iste kao dimenzije konfiguracije. Predlažemo čitaocima da pokušaju da dokažu da nije potrebno praviti novu matricu istih dimenzija, već je dovoljno da ona ima svega nekoliko vrsta (ili svega nekoliko kolona). Nakon uspešnog dokazivanja, za vežbu bi se mogao implementirati metod za izračunavanje naredne generacije uz primenu novostečenog znanja. Povoljan uticaj na performanse bi mogla imati i jedinstvena alokacija pomoćne matrice, u vidu podatka klase `Igra`, umesto što se ona pri svakom računanju nove generacije iznova pravi i uklanja.

Igra „Život“ predstavlja osnovu za nebrojene izmene pravila. Eksperimenti sa pravilima mogu igru učiniti još zanimljivijom.

Posebno zanimljiva vežba bi bila podrška za neograničenu tablu za igru. Primetimo da je ipak potrebno uvesti neke ograničavajuće pretpostavke kao što su ograničavanje dimenzija opsegom celobrojnog tipa ili ograničavanje veličine početne konfiguracije.

Deo II

Složeni primeri

Drugi deo knjige se bavi tehnikama objektno orijentisanog programiranja na programskom jeziku C++.

Obuhvata četiri složenija primera. Kroz svaki od njih se detaljnije upoznaju neki napredni elementi programskog jezika i njegove standardne biblioteke i neke tehnike objektno orijentisanog programiranja. Ovaj deo knjige namenjen je čitaocima koji su uspešno savladali osnovne teme i u stanju su da samostalno rešavaju elementarne probleme.

Primer 6 - **Pretraživanje teksta**

Primer 7 - **Enciklopedija**

Primer 8 - **Kodiranje**

Primer 9 - **Igra „Život“, II deo**

6 - Pretraživanje teksta

6.1 Zadatak

Napisati program koji pretražuje dati tekst. Program se pokreće komandom:

```
pretraži <ime datoteke>
```

Nakon pokretanja programa, korisniku se omogućava da zada uslov pretraživanja. Uslov pretraživanja se sastoji od reči razdvojenih praznim prostorom. Ispred svake reči može se pojaviti znak '+' ili znak '-'. Formalno, uslov ima oblik:

```
[+|-]<reč> { [+|-]<reč> }
```

Kao rezultat pretraživanja ispisati izveštaj koji se sastoji od rednih brojeva i sadržaja onih rečenica teksta koje:

- sadrže bar jednu navedenu reč (ispred koje ne stoji znak '-');
- sadrže sve reči ispred kojih stoji znak '+';
- ne sadrže nijednu reč ispred koje stoji znak '-'.

Potrebno je omogućiti efikasno pretraživanje teksta tako što će se važne informacije o tekstu, a pre svega podaci o tome u kojim rečenicama se nalaze sadržane koje reči, čuvati u organizovanom obliku kako se ne bi moralo za svaki novi uslov iznova prolaziti kroz tekst.

Rečenicom smatrati tekst koji se završava krajem reda, krajem datoteke ili nekim od znakova '.', '!','?'.

Cilj zadatka

Rešavanjem ovog zadatka pokazaćemo:

- koji su i kako se primenjuju metodi klase `string` za pretraživanje niski;
- podsetićemo se kako se implementira binarno drvo.

- kako se može implementirati skup primenom binarnog drveta;
- kako se može implementirati katalog primenom binarnog drveta;
- kako se mogu implementirati iteratori za binarno drvo;
- kako se primenom šablona može uopštiti napisana klasa;
- kako se upotrebljavaju klase `map` i `set` standardne biblioteke programskog jezika C++.

Zadatak ćemo rešavati implementirajući najpre sve potrebne strukture podataka, a zatim ih zamenjujući klasama `map` i `set`. Na taj način ćemo temeljnije predstaviti ponašanje tih klasa, što će doprineti njihovom dubljem razumevanju i kvalitetnijoj upotrebi. Naravno, nećemo se upuštati u implementiranje svih mehanizama koji postoje u ovim klasama, već samo onih koji su nama potrebni u ovom zadatku, što ujedno predstavlja i najčešće upotrebljavan podskup metoda ovih klasa.

Pretpostavljena znanja

Za uspešno praćenje rešavanja ovog zadatka pretpostavlja se osnovno poznavanje uglavnom svih elemenata programskog jezika C++, a pre svega:

- rada sa pokazivačima i dinamičkim strukturama podataka;
- principa funkcionisanja šablona;
- rada sa izuzecima;
- klasa standardne biblioteke: `string`, `vector`, `set` i `map`;
- standardne biblioteke tokova i datotečnih tokova.

6.2 Rešavanje zadatka

Ovaj zadatak ćemo rešavati postupno, ne ulazeći unapred u detaljnu analizu i projektovanje čitavog programa. Na taj način ćemo primeniti u praksi neke od osnovnih metoda agilnog razvoja softvera:

- program ćemo pisati u malim koracima;
- svaki korak prevodi program iz jednog u drugo funkcionalno stanje;
- nakon svakog koraka program se prevodi i testira se funkcionisanje implementiranih delova;
- ako je korak obiman, podelićemo ga na manje celine tako da nakon svake od njih program ili delovi programa mogu da se testiraju.

Zbog ograničenog obima teksta nećemo se držati svih principa agilnog razvoja softvera. Najznačajnije odstupanje je da nećemo pisati posebne metode za automatsko testiranje, mada ćemo pisati pojedine funkcije ili segmente koda koje će omogućiti neki vid provere. Napominjemo da takav pristup (tj. selektivno pridržavanje principa ovog metoda razvoja softvera) u praksi nije preporučljiv i lako može dovesti do značajnih grešaka.

Pri testiranju programa ćemo pretraživati upravo tekst programa koji pišemo. Kao i u većini prethodnih primera, program nećemo deliti na module već ćemo ga pisati u okviru jedne datoteke – `pretraži.cpp`. Zbog toga što će veći broj metoda biti relativno složen, njihove definicije ćemo navoditi van definicije klase.

Rešenje će biti izloženo u nekoliko koraka:

Na kraju svakog koraka ili celine program je potrebno prevoditi i testirati.

Korak 1 - Glavna funkcija

Najpre ćemo napisati glavnu funkciju programa. Zbog toga što još ne umemo da izvedemo pretraživanje, samo ćemo ispisati poruku.

```
#include <stdexcept>
#include <iostream>

using namespace std;

//-------------------------------------------------------------
// Ovo je funkcija koja pretražuje.
//-------------------------------------------------------------
void pretrazivanjeTeksta( const char* )
{
    // Za sada baš i ne pretražujemo...
    cout << "Pretraživanje...\n";
}

//-------------------------------------------------------------
// Glavna funkcija programa proverava parametre
// i poziva funkciju za pretraživanje teksta.
//-------------------------------------------------------------
int main( int argc, char** argv )
{
```

```
    // Čitav kod stavljamo u jedan try blok.
    try {
        // Proveravamo argumente...
        if( argc < 2 )
            throw invalid_argument( "Nedostaje argument!" );
        // ...pa pretražujemo.
        pretrazivanjeTeksta( argv[1] );
        }

    // Hvatamo i obrađujemo sve greške.
    catch( exception& e ){
        cout <<
            "Greška: \n"
            "   " << e.what() << endl <<
            "Upotreba: \n"
            "   pretraži <datoteka>\n";
        }

    return 0;
}
```

Nakon prevođenja program se može testirati komandom:

```
pretraži pretraži.cpp
```

Korak 2 - Čitanje datoteke i izdvajanje rečenica

Da bi pretraživanje bilo efikasno, kao što je u tekstu zadatka navedeno, potrebno je pročitati tekst i sačuvati kako informacije o sadržaju redova, tako i informacije o tome koja se reč nalazi u kom redu. Za početak, čitamo tekst i pravimo kolekciju redova.

Pročitani tekst ćemo modelirati klasom `Tekst`. Napisaćemo metode za čitanje i pamćenje svih redova date datoteke, kao i metod za ispisivanje pročitanih redova, koji ćemo koristiti za proveru da li je čitanje prošlo sa uspehom. Konstruktor klase `Tekst` će pozivati metod za čitanje redova.

```
//-------------------------------------------------------------
// Klasa Tekst
//-------------------------------------------------------------
// Predstavlja tekstualnu datoteku koju pretražujemo.
//-------------------------------------------------------------
class Tekst
{
public:
    //---------------------------------------------------------
    // Konstruktor.
    //---------------------------------------------------------
    Tekst( const char* nazivDat );

    //---------------------------------------------------------
    // Pomoćni metod koji nam u fazi testiranja služi za proveru
    // da li su redovi teksta dobro pročitani.
    //---------------------------------------------------------
    void IspisiRecenice( ostream& ostr ) const;

private:
```

```
    //-------------------------------------------------------------
    // Čitanje teksta i izdvajanje i pamćenje rečenica.
    //-------------------------------------------------------------
    void CitajRecenice( const char* nazivDat );

    //-------------------------------------------------------------
    // Članovi podaci
    //-------------------------------------------------------------
    vector<string>          _Recenice;
};
```

Konstruktor izvodi prikupljanje informacija o tekstu. Za sada je dovoljno da se pročitaju rečenice:

```
//-------------------------------------------------------------
// Konstruktor.
//-------------------------------------------------------------
Tekst::Tekst( const char* nazivDat )
{
    CitajRecenice( nazivDat );
}
```

Metod `ČitajRečenice` implementiramo tako što čitamo red po red datoteke i iz svakog pročitanog reda izdvajamo rečenice. Pri tome upotrebljavamo metode klase `string` (videti odeljak *10.11 String*, na strani 394).

- Metod `find_first_not_of` pronalazi prvu poziciju na kojoj se nalazi znak različit od svih datih znakova. U slučaju da niska sadrži samo date znakove, rezultat je konstanta `string::npos`. Opcioni drugi argument ovog metoda predstavlja prva pozicija od koje je potrebno tražiti, a koristi nam u slučaju da nije potrebno analizirati početak niske. Ovaj metod koristimo za pronalaženje početka rečenice, tj. za preskakanje praznih znakova, pri čemu kao drugi argument navodimo poznatu poziciju iza poslednjeg znaka prethodne rečenice jer je sadržaj reda do te pozicije već obrađen.
- Metod `find_first_of` pronalazi poziciju prvog pojavljivanja nekog od datih znakova. U slučaju da niska ne sadrži nijedan od datih znakova, rezultat je konstanta `string::npos`. Opcioni drugi argument ovog metoda predstavlja prva pozicija od koje je potrebno tražiti, a koristi nam u slučaju da nije potrebno analizirati početak niske. Ovaj metod koristimo za pronalaženje kraja rečenice, pri čemu kao drugi argument navodimo poznatu poziciju prvog znaka rečenice da ne bismo uključili u razmatranje već obrađene rečenice u istom redu.
- Postoje i slični metodi `find_last_of` i `find_last_not_of`, koji pretražuju od kraja prema početku. Mi koristimo metod `find_last_not_of` za odbacivanje praznih znakova na kraju reda u slučaju poslednje rečenice.

U okviru dela standardne biblioteke za rad sa niskama postoji i veoma korisna funkcija `getline`. Ona čita red iz datoteke. Prvi argument ove funkcije je referenca na ulazni tok, a drugi je referenca na objekat klase `string` u koju se smešta pročitana niska. Prednost ove funkcije u odnosu na istoimeni metod `istream::getline` je u tome što nije neophodno

unapred znati maksimalnu dužinu reda, jer funkcija koristi klasu `string`, a ne pokazivač na znakove. Kao i u slučaju metoda `istream::getline`, i ova funkcija ima opcioni treći argument koji određuje koji se znak prepoznaje kao „kraj reda“. Rezultat funkcije je referenca na tok, pa se može odmah upotrebljavati za proveru da li je čitanje uspelo.

```
//-------------------------------------------------------------
// Čitanje teksta i izdvajanje i pamćenje rečenica.
//-------------------------------------------------------------
void Tekst::CitajRecenice( const char* nazivDat )
{
    // Otvorimo datoteku.
    ifstream f( nazivDat );
    // Ako otvaranje nije uspelo, prijavimo grešku.
    if( !f )
        throw invalid_argument("Ne može se otvoriti datoteka!");

    string red;
    // Čitamo red po red.
    while( getline( f, red )){
        unsigned kraj = 0;
        // Izdvajamo rečenicu po rečenicu iz reda.
        while( kraj < red.size()){
            // Tražimo početak sledeće rečenice u redu.
            unsigned pocetak =
                red.find_first_not_of( " \t\r\n", kraj );
            // Ako ga nema, završili smo.
            if( pocetak == string::npos )
                break;

            // Tražimo kraj rečenice.
            kraj = red.find_first_of( ".?!", pocetak );
            // Ako nema znaka za kraj, smatramo da je kraj
            // poslednji znak koji nije praznina, a takav
            // sigurno postoji, jer inače ne bismo imali
            // ni početak.
            if( kraj == string::npos )
                kraj = red.find_last_not_of(" \t\r\n");
            // Pomerimo 'kraj' na prvi znak iza rečenice.
            kraj++;

            // Zapamtimo rečenicu.
            _Recenice.push_back(
                red.substr( pocetak, kraj-pocetak )
                );
            }
        }
}
```

Ispisivanje rečenica je jednostavno. Ovaj metod nema poseban značaj za samo rešenje zadatka, ali će nam koristiti za proveru da li smo dobro pročitali rečenice.

```
//-------------------------------------------------------------
// Pomoćni metod koji nam u fazi testiranja služi za proveru
// da li su redovi teksta dobro pročitani.
//-------------------------------------------------------------
void Tekst::IspisiRecenice( ostream& ostr ) const
{
```

```
    unsigned n = _Recenice.size();
    for( unsigned i=0; i<n; i++ )
        ostr << (i+1) << ":  " << _Recenice[i] << endl;
}
```

Menjamo funkciju `pretrazivanjeTeksta`. Najpre pravimo objekat klase `Tekst`, koji inicijalizujemo datom datotekom, a zatim, samo radi testiranja ovog koraka, ispisujemo sve rečenice datog teksta.

```
void pretrazivanjeTeksta( const char* nazivDat )
{
    // Analiziramo tekst.
    Tekst t( nazivDat );
    // Pretražujemo. :)
    t.IspisiRecenice( cout );
}
```

Funkciju `main` ne menjamo.

Da bi se mogli upotrebljavati datotečni tokovi i vektori, potrebno je uključiti odgovarajuća zaglavlja:

```
#include <vector>
#include <fstream>
```

Za proveru možemo koristiti tekst programa, ali je neophodno proveriti rad programa i na primeru nekih tekstova koji imaju po više rečenica u redu, kako bi smo proverili da li izdvajanje rečenica iz reda radi kako treba. U tom cilju možemo na kraju teksta programa, u okviru komentara, dodati neki tekst sa više rečenica u jednom redu, poput:

```
// Prva rečenica. Druga rečenica! Treća rečenica? Četvrta rečenica
```

Korak 3 - Izdvajanje reči

Sada idemo korak dalje u analizi teksta. Nakon što smo pročitali sve rečenice, možemo ih detaljnije analizirati kako bismo evidentirali koje se reči nalaze u kojim rečenicama. U ovom koraku se nećemo baviti samim evidentiranjem već samo analizom rečenica. Evidentiranje ostavljamo za naredni korak.

```
//-------------------------------------------------------------
// Izdvajanje reči iz pročitanih rečenica i evidentiranje.
//-------------------------------------------------------------
void Tekst::IzdvojiReci()
{
    // Znaci koji čine reči.
    static const string slova =
        "abcdefghijklmnopqrstuvwxyz"
        "1234567890";

    // Obrađujemo, redom, jednu po jednu rečenicu.
    unsigned n = _Recenice.size();
    for( unsigned i=0; i<n; i++ ){
        // Želimo samo mala slova.
        string recenica = _Recenice[i];
```

```
            for( int i=recenica.size()-1; i>=0; i-- )
                if( recenica[i]>='A' && recenica[i]<='Z' )
                    recenica[i] += 'a' - 'A';
            // Izdvajamo reč po reč.
            unsigned kraj = 0;
            while( kraj < recenica.size()){
                // Tražimo početak reči.
                unsigned pocetak =
                    recenica.find_first_of( slova, kraj );
                if( pocetak == string::npos )
                    break;
                // Tražimo kraj reči.
                kraj = recenica.find_first_not_of( slova, pocetak );
                if( kraj == string::npos )
                    kraj = recenica.size();
                // Evidentiramo reč
                EvidentirajRec(
                    recenica.substr( pocetak, kraj-pocetak ), i
                    );
                }
            }
    }
```

Potrebno je navesti deklaraciju ovog metoda u klasi `Tekst` i definisati metod `EvidentirajRec`. Umesto da reč zaista evidentiramo, za sada samo ispisujemo kontrolnu poruku.

```
class Tekst
{ ...
private:
    //-------------------------------------------------------------
    // Izdvajanje reči iz pročitanih rečenica i evidentiranje.
    //-------------------------------------------------------------
    void IzdvojiReci();
    //-------------------------------------------------------------
    // Evidentiranje reči u rečenici sa datim rednim brojem.
    //-------------------------------------------------------------
    void EvidentirajRec( const string& rec, int recenica )
        {
        // Za sada samo ispisujemo podatke.
        cout << recenica << ": " << rec << endl;
        }
...};
```

Preostaje nam da izmenimo konstruktor tako da nakon čitanja rečenica izdvoji reči:

```
Tekst::Tekst( const char* nazivDat )
{
    CitajRecenice( nazivDat );
    IzdvojiReci();
}
```

Nakon navedenih izmena, pri testiranju se za svaku pročitanu rečenicu ispisuje koje reči sadrži. Iz funkcije `pretrazivanjeTeksta` možemo izostaviti ispisivanje rečenica.

```
void pretrazivanjeTeksta( const char* nazivDat )
{
    // Analiziramo tekst.
    Tekst t( nazivDat );
}
```

Naš kod ne radi ništa posebno, osim što pravi i odmah zatim uklanja objekat `t`. Ipak, to je sasvim dovoljno, jer konstruktor izračunava i ispisuje potrebne podataka o tekstu.

Korak 4 - Problem evidentiranja reči

Do sada nismo imali većih teškoća, ali ovaj korak će nam ih doneti. Oblik u kome bi trebalo čuvati podatke nije sasvim ugodan za rad. Potrebno je za svaku reč evidentirati redne brojeve rečenica u kojima se ta reč nalazi. Jedan način bi bio da napravimo niz elemenata oblika:

```
struct PoložajReči {
    string  reč;
    int     rečenica;
};
```

Za svako pojavljivanje reči u tekstu evidentirali bismo u kojoj je rečenici. Takav pristup bi bio prilično besmislen, jer se količina podataka ne smanjuje značajno u odnosu na originalni tekst. Prvi problem sa kojim se susrećemo je potencijalno višestruko ponavljanje reči. Bilo bi daleko efikasnije ako bi se broj ponavljanja reči smanjio. Sa druge strane, za efikasan rad nam je potrebna struktura koja omogućava da se za datu reč što efikasnije pronađe skup *svih* rečenica u kojima se ona nalazi. Prethodno opisano rešenje nas obavezuje da svaki put prođemo kroz čitav niz da bismo sakupili sva pojavljivanja.

Kako je pronalaženje najefikasnije ako podatke čuvamo upravo u traženom obliku, možemo podatke čuvati na nešto složeniji način, tako da se uz svaku reč vodi skup rednih brojeva rečenica u kojima se ona pojavljuje:

```
struct PoložajReči {
    string              reč;
    SkupCelihBrojeva    rečenice;
};
```

Sada je u nizu dovoljno čuvati po jedan podatak (mada prilično složeniji) za svaku reč koja se pojavljuje u tekstu. Kada njega pronađemo odmah ćemo imati na raspolaganju skup rečenica u kojima se tražena reč pojavljuje. U dobitku smo po bar dva pitanja, jer ne samo što je veličina niza višestruko smanjena (svaka reč je u nizu zastupljena tačno po jedan put) već više nema potrebe da za reči koje postoje u tekstu prolazimo kroz čitav niz, već, u proseku, samo kroz pola niza. Doduše, ako reč ne postoji u tekstu, i dalje je potrebno proveravati do kraja niza.

Dalje povećanje efikasnosti može se ostvariti uređivanjem niza. Na taj način složenost pretraživanja ne bi više rasla linearno proporcionalno složenosti tekstova već logaritamski, jer bismo mogli da koristimo binarno pretraživanje. Nezgoda strana je što nije besplatno napraviti uređen niz, pa se time značajno usporava pravljenje potrebnih struktura podataka.

Rešenje koje se nameće kao bolje (ali i složenije) je upotreba binarnog drveta uređenog po rečima.

Poseban problem je kako napraviti efikasne skupove rednih brojeva rečenica. Mogu se koristiti nizovi, ali onda se susrećemo sa istim problemima kao u slučaju niza reči. Možemo zaključiti da je i ovde binarno drvo verovatno najpogodnije rešenje.

Uređeno binarno drvo

Napisaćemo sve što nam je potrebno za funkcionisanje binarnog drveta. Nešto je lakše raditi sa celim brojevima nego sa složenijim elementima, pa ćemo zato raditi kao da se radi o binarnom drvetu rednih brojeva rečenica, koje ćemo upotrebljavati kao skup. Ponašanje drveta je ugrađeno u klasu `BinarnoDrvo`. Klasa `BinarnoDrvo::Cvor` sadrži samo neophodnu logiku pravljenja, uklanjanja i kopiranja čvorova. Gde god je bilo moguće, primenjena je rekurzija.

```
//-------------------------------------------------------------
// Klasa BinarnoDrvo
//     Predstavlja uređeno binarno drvo celih brojeva.
//-------------------------------------------------------------
class BinarnoDrvo
{
public:
    //---------------------------------------------------------
    // Klasa BinarnoDrvo::Cvor
    //     Predstavlja jedan čvor binarnog drveta. Ima samo
    //     konstruktor, dok je ostala logika u klasi BinarnoDrvo.
    //---------------------------------------------------------
    class Cvor
    {
    public:
        //-----------------------------------------------------
        // Pristupanje podacima
        //-----------------------------------------------------
        int Sadrzaj() const
            { return _Sadrzaj; }

private:
        //-----------------------------------------------------
        // Konstruktor, destruktor,...
        //-----------------------------------------------------
        Cvor( int s, Cvor* l=0, Cvor* d=0 )
            : _Sadrzaj(s),
              _Levi(l), _Desni(d)
            {}

        Cvor( const Cvor& c )
            : _Sadrzaj( c._Sadrzaj ),
              _Levi( c._Levi ? new Cvor( *c._Levi ) : 0 ),
              _Desni( c._Desni ? new Cvor( *c._Desni ) : 0 )
            {}

        ~Cvor()
            {
            delete _Levi;
```

```
            delete _Desni;
            }

        Cvor& operator = ( const Cvor& c )
            {
            if( this != &c ){
                delete _Levi;
                delete _Desni;
                _Levi = c._Levi ? new Cvor( *c._Levi ) : 0;
                _Desni = c._Desni ? new Cvor( *c._Desni ) : 0;
                }
            return *this;
            }

        //-------------------------------------------------------
        // Članovi podaci.
        //-------------------------------------------------------
        int _Sadrzaj;
        Cvor *_Levi, *_Desni;

        friend class BinarnoDrvo;
    };

    //-------------------------------------------------------------
    // Konstruktor, destruktor, ...
    //-------------------------------------------------------------
    BinarnoDrvo()
        : _Koren(0)
        {}

    BinarnoDrvo( const BinarnoDrvo& d )
        : _Koren( d._Koren ? new Cvor(*d._Koren) : 0 )
        {}

    ~BinarnoDrvo()
        { delete _Koren; }

    BinarnoDrvo& operator = ( const BinarnoDrvo& d )
        {
        if( this != &d ){
            delete _Koren;
            _Koren = d._Koren ? new Cvor(*d._Koren) : 0;
            }
        return *this;
        }

    //-------------------------------------------------------------
    // Dodavanje čvora sa novim elementom, ako već ne postoji.
    //-------------------------------------------------------------
    Cvor* Dodaj( int s );

    //-------------------------------------------------------------
    // Pronalaženje elementa u drvetu.
    //-------------------------------------------------------------
    Cvor* Pronadji( int s )
        { return PronadjiPokazivacNaCvor( s ); }
```

```
    //-------------------------------------------------------------
    // Izbacivanje čvora iz drveta je nešto složenije
    // i nećemo ga pisati jer nam ta operacija nije potrebna.
    //-------------------------------------------------------------

private:
    //-------------------------------------------------------------
    // Pronalaženje pokazivača koji pokazuje na traženi čvor,
    // ako postoji, ili pokayivača koji je potrebno ažurirati
    // ako se čvor dodaje.
    //-------------------------------------------------------------
    Cvor*& PronadjiPokazivacNaCvor( int s );

    //-------------------------------------------------------------
    // Članovi podaci.
    //-------------------------------------------------------------
    Cvor* _Koren;
};

//-----------------------------------------------------------------
// Dodavanje novog čvora sa novim elementom, ako takav ne postoji.
//-----------------------------------------------------------------
BinarnoDrvo::Cvor* BinarnoDrvo::Dodaj( int s )
{
    Cvor*& p = PronadjiPokazivacNaCvor( s );
    if( !p )
        p = new Cvor(s);
    return p;
}

//-----------------------------------------------------------------
// Pronalaženje pokazivača koji pokazuje na traženi čvor.
//-----------------------------------------------------------------
BinarnoDrvo::Cvor*& BinarnoDrvo::PronadjiPokazivacNaCvor( int s )
{
    Cvor** p = &_Koren;
    while( *p ){
        Cvor* c = *p;
        if( s < c->_Sadrzaj )
            p = &c->_Levi;
        else if( s > c->_Sadrzaj )
            p = &c->_Desni;
        else
            break;
        }
    return *p;
}
```

Izostavljena je mogućnost selektivnog brisanja čvorova iz drveta, jer predstavlja relativno složen posao, a nije nam neophodna u ovom trenutku.

Skup celih brojeva

Implementacija skupa je prilično jednostavna, s obzirom na to da imamo na raspolaganju prethodno implementirano binarno drvo. Za početak ćemo se baviti samo jednostavnim operacijama dodavanja elemenata skupu i proveravanja da li skup sadrži dati element.

```
//-------------------------------------------------------------
// Klasa Skup
//-------------------------------------------------------------
// Predstavlja skup celih brojeva.
//-------------------------------------------------------------
class Skup
{
public:
    //---------------------------------------------------------
    // Dodavanje elementa skupu.
    //---------------------------------------------------------
    void Dodaj( int s )
        { _Drvo.Dodaj( s ); }

    //---------------------------------------------------------
    // Provera da li skup sadrži dati element.
    //---------------------------------------------------------
    bool Sadrzi( int s )
        { return _Drvo.Pronadji(s); }

    //---------------------------------------------------------
    // Izbacivanje elemenata iz skupa nije podržano.
    //---------------------------------------------------------

private:
    //---------------------------------------------------------
    // Članovi podaci.
    //---------------------------------------------------------
    BinarnoDrvo _Drvo;
};
```

Na ovom mestu program možemo da prevedemo da bismo proverili kako se ponaša. Za proveru funkcionisanja klase `Skup` može poslužiti sledeći segment koda, privremeno umetnut u funkciju `main` ili u funkciju `pretrazivanjeTeksta`:

```
// Testiranje klasa BinarnoDrvo i Skup
Skup q;
// Popunjavanje skupa neparnim brojevima
for( int i=0; i<10; i++ )
    if( i%2 )
        q.Dodaj(i);
// Ispitivanje sadržaja skupa
for( int i=0; i<10; i++ )
    if( q.Sadrzi(i) )
        cout << i << endl;
```

Primetimo da nije implementirano uklanjanje elemenata iz skupa, jer je za to potrebno dograditi binarno drvo.

Obrađivanje svih elemenata binarnog drveta i skupa

Najvažnije što našem skupu nedostaje je mogućnost da efikasno obradimo sve elemente skupa. Sada to možemo samo tako što ćemo za *svaki ceo broj* proveriti da li pripada skupu i zatim, ako pripada, uraditi šta je potrebno. Teško da bi se to moglo nazvati efikasnim rešenjem. Da bismo uopšte bili u prilici da problem rešimo na bolji način, moramo se najpre vratiti na binarno drvo.

Problemu obrađivanja elemenata binarnog drveta (kao i svake druge kolekcije, uostalom) može se prići na više načina. Navešćemo dva najčešće primenjivana načina. Prvi je da se u samom drvetu obezbedi metod koji obilazi sve čvorove redom i tom prilikom obavlja potreban posao. To je relativno jednostavan način ali je ujedno i prilično nefleksibilan:

```
//-------------------------------------------------------------
// Metod izvršava funkciju f za svaki čvor date grane, počev
// od najmanjeg (krajnjeg levog) pa do najvećeg (krajnji desni).
//-------------------------------------------------------------
void BinarnoDrvo::ObidjiGranu( Cvor* p, void(*f)(Cvor*) )
{
    // Obilaženje leve grane.
    if( p->_Levi )
        ObidjiGranu( p->_Levi, f );
    // Primena operacije na sadržaj čvora.
    f(p);
    // Obilaženje desne grane.
    if( p->_Desni )
        ObidjiGranu( p->_Desni,f );
}

//-------------------------------------------------------------
// Metod izvršava funkciju f za svaki čvor drveta.
//-------------------------------------------------------------
void BinarnoDrvo::ObidjiDrvo( void(*f)(Cvor*) )
    { ObidjiGranu( _Koren, f );}
```

Ovakav način, iako je sasvim jednostavan za implementaciju, ne predstavlja najbolje rešenje, jer dovođenje potrebnih funkcija u oblik koji je prihvatljiv za upotrebu navednim metodom može biti prilično nezahvalan posao. Problem se posebno usložnjava ako je potrebno neke elemente zapamtiti radi kasnijeg izvođenja drugih operacija i sl. Drugi često korišćen pristup je znatno složeniji, ali omogućava punu fleksibilnost, pa ćemo ga zbog toga primeniti pri rešavanju ovog zadatka. U pitanju je pristupanje elementima posredstvom *iteratora*, na sličan način kao što se to radi u kolekcijama `vector` i `list` standardne biblioteke (videti *10.2 Iteratori*, na strani 348). Potrebno je u klasi `BinarnoDrvo` obezbediti metode koji vraćaju početne i završne iteratore, kao i metode koji omogućavaju prelazak iteratora na sledeći element. Iterator mora biti objekat koji omogućava jednoznačno pristupanje elementima drveta, pomeranje na sledeći element i poređenje. U našem slučaju iterator se može implementirati uz primenu pokazivača na čvor drveta.

Ako posmatramo proizvoljan čvor, kako da znamo koji je *sledeći* čvor u poretku kojim su čvorovi drveta uređeni? Ako čvor ima desnog potomka, sledeći čvor je krajnji levi čvor u desnoj grani. Ako nema, onda je sledeći čvor upravo najbliži predak u čijoj se levoj grani nalazi tekući čvor. Imamo na raspolaganju bar dva načina da dođemo do pretka: ili da svaki čvor pamti svog pretka ili da iterator pamti istoriju svog kretanja u dubinu drveta. Ovom prilikom se, radi ilustracije iteratora, odlučujemo za drugi način.

```
class BinarnoDrvo
{
public:
    ...
```

```
class iterator;
class Cvor
{
    ...
    friend class BinarnoDrvo;
    friend class iterator;
};

//------------------------------------------------------------
// Klasa BinarnoDrvo::iterator
//------------------------------------------------------------
class iterator
{
public:
    //--------------------------------------------------------
    // Konstruktor.
    //--------------------------------------------------------
    iterator( Cvor* c = 0 )
        { SidjiDoKrajaLevo(c); }

    //--------------------------------------------------------
    // Korak napred, na sledeći čvor drveta. Prefiksno ++.
    //--------------------------------------------------------
    void operator++ ()
        {
        if( _Tekuci ){
            if( _Tekuci->_Desni )
                SidjiDoKrajaLevo( _Tekuci->_Desni );
            else if( _DesniPreci.empty() )
                _Tekuci = 0;
            else{
                _Tekuci = _DesniPreci.back();
                _DesniPreci.pop_back();
                }
            }
        }

    //--------------------------------------------------------
    // Postfiksno ++.
    //--------------------------------------------------------
    void operator++(int)
        { ++(*this); }

    //--------------------------------------------------------
    // Poređenje iteratora.
    //--------------------------------------------------------
    bool operator==( const iterator& i )
        { return _Tekuci == i._Tekuci; }
    bool operator!=( const iterator& i )
        { return _Tekuci != i._Tekuci; }

    //--------------------------------------------------------
    // Dereferenciranje iteratora.
    //--------------------------------------------------------
    int operator *()
        { return _Tekuci ? _Tekuci->_Sadrzaj : 0; }

private:
```

```
        //-------------------------------------------------------
        // Silaženje do krajnjeg levog elementa grane c.
        //-------------------------------------------------------
        void SidjiDoKrajaLevo( Cvor* c );
        //-------------------------------------------------------
        // Članovi podaci
        //-------------------------------------------------------
        Cvor*           _Tekuci;
        vector<Cvor*>   _DesniPreci;
    };

    //-------------------------------------------------------------
    // Metodi klase BinarnoDrvo za pravljenje početnog
    // i završnog iteratora.
    //-------------------------------------------------------------
    iterator begin()
        { return iterator(_Koren); }
    iterator end()
        { return iterator(0); }
...
};

...

//-----------------------------------------------------------------
// Silaženje do krajnjeg levog elementa grane c.
//-----------------------------------------------------------------
void BinarnoDrvo::iterator::SidjiDoKrajaLevo( Cvor* c )
{
    if( c )
        while( c->_Levi ){
            _DesniPreci.push_back(c);
            c = c->_Levi;
            }
    _Tekuci = c;
}
```

Implementacija iteratora za klasu `Skup` je trivijalna:

```
class Skup
{
public:
...
    //-------------------------------------------------------
    // Tip iteratora Skup::iterator.
    //-------------------------------------------------------
    typedef BinarnoDrvo::iterator iterator;

    //-------------------------------------------------------
    // Metodi klase Skup za pravljenje početnog i završnog iter.
    //-------------------------------------------------------
    iterator begin()
        { return _Drvo.begin(); }
    iterator end()
        { return _Drvo.end(); }
...
};
```

Vreme je da se ponovo proveri da li i kako radi napisani deo programa. Za proveru funkcionisanja iteratora može poslužiti sledeći segment koda:

```
// Testiranje iteratora
Skup w;

// Popunjavanje skupa takvim redom da se dobije ravnomerno
// popunjeno drvo dubine 2
w.Dodaj(4);
w.Dodaj(2);
w.Dodaj(6);
w.Dodaj(1);
w.Dodaj(3);
w.Dodaj(5);
w.Dodaj(7);

// Upotreba iteratora za ispisivanje svih elemenata skupa
for( Skup::iterator i=w.begin(); i!=w.end(); i++ )
    cout << (*i) << ' ';
cout << endl;
```

Šabloni

Imamo sve što nam je potrebno da bismo za jednu reč evidentirali redne brojeve rečenica u kojima se nalazi, kao i da bismo tako prikupljene podatke efikasno pretraživali ili ispisivali. Međutim, da bismo mogli pristupiti evidentiranju podataka potrebno je da obezbedimo još i strukturu koja će predstavljati katalog pronađenih reči. Ranije smo već rešili da ćemo za to koristiti binarno drvo u kome ćemo čuvati podatke poput:

```
struct PoložajReči {
    string              reč;
    SkupCelihBrojeva    rečenice;
};
```

Znači tako, sada nam preostaje da naše `BinarnoDrvo` napišemo tako da tip elemenata bude `PoložajReči`. Ipak, pre nego što započnemo modifikovanje, pogledajmo malo bolje napisane klase `BinarnoDrvo` i `Skup`. Iako su napisane tako da rade samo za cele brojeve, možemo primetiti da su značajni delovi koda, a posebno sami algoritmi, nezavisni od tipa elemenata koji se čuvaju u drvetu i skupu. Ako bismo hteli da čuvamo neke složenije podatke, bilo bi dovoljno na mestima gde je tip elemenata označen kao `int` staviti neki drugi tip, uz pretpostavku da su za taj drugi tip definisani neophodni operatori poređenja. Umesto da ponavljamo više puta isti kod, možemo upotrebiti šablone.

Šabloni su predstavljeni u ranijim primerima, pa se ovde ne zadržavamo na detaljima njihove upotrebe. Sasvim je jednostavno napraviti šablon od postojeće klase `BinarnoDrvo`. Jedini parametar šablona je tip elementa drveta. Sledi šablon klase `BinarnoDrvo` u kome su istaknuti redovi u kojima je bilo izmena. Originalni komentari su izostavljeni radi uštede prostora, a navedeni su novi na mestima gde čitaocu može da ne bude očigledan razlog uvođenja izmena:

```
template< class T >
class BinarnoDrvo
{
public:
    class Cvor
    {
    public:
        const T& Sadrzaj() const
            { return _Sadrzaj; }

        Cvor( const T& s, Cvor* l=0, Cvor* d=0 )
            : _Sadrzaj(s),
              _Levi(l), _Desni(d)
            {}
        ...

        T _Sadrzaj;
        Cvor *_Levi, *_Desni;
    };

    class iterator
    {
    public:
        ...
        T& operator *() const
            {
            // Ranije je ovde bio samo jedan red oblika
            //   return Tekuci ? Tekuci->Sadrzaj : 0;
            // ali za nepoznati tip T ne možemo podrazumevati
            // vrednost 0, a T() ne dolazi u obzir jer se
            // rezultat vraća po referenci!
            if( !_Tekuci )
                throw out_of_range("Iterator van opsega!");
            return _Tekuci->_Sadrzaj;
            }
        ...
    };
    ...

    Cvor* Dodaj( const T& s );

    Cvor* Pronadji( const T& s )
        { return PronadjiPokazivacNaCvor( s ); }

private:
    Cvor*& PronadjiPokazivacNaCvor( const T& s );

    Cvor* _Koren;
};

template< class T >
BinarnoDrvo<T>::Cvor* BinarnoDrvo<T>::Dodaj( const T& s )
{...}

template< class T >
BinarnoDrvo<T>::Cvor*&
BinarnoDrvo<T>::PronadjiPokazivacNaCvor( const T& s )
{...}
```

```
template< class T >
void BinarnoDrvo<T>::iterator::SidjiDoKrajaLevo( Cvor* c )
{...}
```

Nije li ovo bilo relativno jednostavno? A kakva je situacija sa klasom `Skup`? Da li se i od nje može napraviti šablon? Naravno. Nama je, doduše, za rešavanje ovog zadatka sasvim dovoljan skup celih brojeva, ali nije loše da radi vežbe i od klase `Skup` napravimo šablon. Tim pre što je to još jednostavnije nego u slučaju klase `BinarnoDrvo`:

```
template< class T >
class Skup
{
public:
    void Dodaj( const T& s )
        { _Drvo.Dodaj( s ); }
    bool Sadrzi( const T& s )
        { return _Drvo.Pronadji(s); }
    typedef BinarnoDrvo<T>::iterator iterator;
    iterator begin()
        { return _Drvo.begin(); }
    iterator end()
        { return _Drvo.end(); }
private:
    BinarnoDrvo<T> _Drvo;
};
```

Da bismo proverili ispravnost napisanih šablona, potrebno je da u delovima koda za testiranje umesto klase `Skup` upotrebimo `Skup<int>`:

```
Skup<int> q;
// Popunjavanje skupa neparnim brojevima
for( int i=0; i<10; i++ )
    if( i%2 )
        q.Dodaj(i);
// Ispitivanje sadržaja skupa
for( int i=0; i<10; i++ )
    if( q.Sadrzi(i) )
        cout << i << endl;
Skup<int> w;
// Popunjavanje skupa takvim redom da se dobije ravnomerno
// popunjeno drvo dubine 2
w.Dodaj(4);
w.Dodaj(2);
w.Dodaj(6);
w.Dodaj(1);
w.Dodaj(3);
w.Dodaj(5);
w.Dodaj(7);
// Upotreba iteratora za ispisivanje svih elemenata skupa
for( Skup<int>::iterator i=w.begin(); i!=w.end(); i++ )
    cout << (*i) << ' ';
cout << endl;
```

Načinili smo relativno male izmene u našim klasama, ali je primenjivost napisanih klasa višestruko uvećana. Sada ih možemo koristiti za različite tipove podataka.

Katalog reči

Razmotrimo najznačajnije sličnosti i razlike između postojeće klase `Skup` i potrebnog kataloga reči:

- ni skup ni katalog ne bi trebalo da sadrže duplikate objekata;
- i skup i katalog bi trebalo da obezbede efikasno pristupanje podacima;
- kada se pristupa elementu skupa, traži se element jednak datom objektu, dok se kod kataloga traži element čiji je samo jedan deo jednak datom objektu – taj deo se naziva *ključ*;
- kod skupa samo proveravamo da li u njemu postoji odgovarajući objekat, a kod kataloga nam je potrebno i da možemo upotrebljavati pronađeni objekat, upravo zbog toga što njegov sadržaj nije samo ključ.

Suština razlike je u osobini kataloga da je jedan njegov element složeniji od elementa skupa jer se sastoji ne samo od ključa već i od nekih drugih podataka. U našem slučaju, katalog reči bi trebalo da kao ključ ima upravo reč, dok bi se osim nje čuvao i skup rednih brojeva rečenica u kojima se ta reč nalazi. Kako već imamo na raspolaganju šablon klase `Skup`, jedno moguće rešenje je da se napravi klasa `PoložajReči` sa operatorima poređenja koji uzimaju u obzir samo reč, pa da se skup takvih objekata koristi kao katalog reči. Jedino mesto gde se zapravo porede elementi skupa predstavlja metod `BinarnoDrvo<T>::PronadjiPokazivacNaCvor`. U tom metodu se koriste operatori '<' i '>'. Jednostavnom prepravkom ovog metoda može se postići da se koristi samo operator '<' čime se pojednostavljuje implementacija klasa koje čuvamo u drvetu i skupu:

```
template< class T >
BinarnoDrvo<T>::Cvor*&
BinarnoDrvo<T>::PronadjiPokazivacNaCvor( const T& s )
{
    Cvor** p = &_Koren;
    while( *p ){
        Cvor* c = *p;
        if( s < c->_Sadrzaj )
            p = &c->_Levi;
        else if( c->_Sadrzaj < s )
            p = &c->_Desni;
        else
            break;
        }
    return *p;
}
```

Po svemu sudeći, sada je potrebno da napišemo jednostavnu klasu `PolozajReci` i zatim možemo napraviti skup objekata te klase kao katalog reči:

```
//---------------------------------------------------------------
// Klasa PolozajReci
//---------------------------------------------------------------
// Predstavlja reč i redne brojeve rečenica u kojima se pojavljuje
//---------------------------------------------------------------
class PolozajReci
{
public:
    string      rec;
    Skup<int>   recenice;

    bool operator <( const PolozajReci& pr ) const
        { return rec < pr.rec; }
};
```

Radi testiranja pišemo još nekoliko redova koda:

```
// Kao katalog reči služi nam skup objekata klase PolozajReci.
Skup<PolozajReci> katalog;

// Dodavanje za sada nije sasvim jednostavno.
PolozajReci p;
p.rec = "a";
katalog.Dodaj(p);
p.rec = "b";
katalog.Dodaj(p);

// I za proveravanje nam je potreban objekat p.
p.rec = "c";
if(katalog.Sadrzi(p) )
    cout << p.rec << endl;
p.rec = "a";
if(katalog.Sadrzi(p) )
    cout << p.rec << endl;

// Obrađivanje svih elemenata se izvodi na uobičajen način.
for( Skup<PolozajReci>::iterator i= katalog.begin();
                                 i!= katalog.end(); i++ )
    cout << (*i).rec << ' ';
cout << endl;
```

Sada imamo katalog reči uz koje idu i skupovi rednih brojeva rečenica u kojima se reči nalaze. Da li? Ne sasvim. Iako možemo da proverimo da li postoji reč, nema načina da vidimo u kojim rečenicama se nalazi niti da taj skup rečenica menjamo. Naime, naša klasa `Skup` nam ne omogućava da pristupamo pronađenom elementu, već samo da u vidu podatka logičkog tipa dobijemo informaciju da li je objekat pronađen ili ne. Jedno moguće rešenje bi bilo da se skupu dodaju odgovarajući metodi, ali time se ponašanje skupa značajno udaljava od ustanovljenog pojma skupa, pa to ne bi bio dobar izbor. Bolje rešenje je da ostavimo po strani klasu `Skup` i da napravimo novu klasu `KatalogReci`. Tako imamo punu slobodu da u nju stavimo sve što nam je potrebno. Još jedna od stvari koje nisu bile dobre kod prethodnog pristupa je što smo pri traženju elemenata morali da navodimo kompletan objekat klase `PolozajReci`. U klasi `KatalogReci` ćemo omogućiti da se navodi samo reč.

Klasu `PolozajReci` ne menjamo. Po uzoru na klasu `Skup` pravimo klasu `KatalogReci`:

```
//-------------------------------------------------------------
// Klasa KatalogReci
//-------------------------------------------------------------
// Predstavlja katalog reči zastupljenih u tekstu.
//-------------------------------------------------------------
class KatalogReci
{
public:
    //---------------------------------------------------------
    // Dodavanje reči i praznog skupa rečenica.
    //---------------------------------------------------------
    PolozajReci* Dodaj( const string& rec )
        {
        PolozajReci p;
        p.rec = rec;
        return &_Drvo.Dodaj( p )->_Sadrzaj;
        }
    //---------------------------------------------------------
    // Pronalaženje podataka o datoj reči u katalogu.
    //---------------------------------------------------------
    PolozajReci* Pronadji( const string& rec )
        {
        PolozajReci p;
        p.rec = rec;
        BinarnoDrvo<PolozajReci>::Cvor* cvor = _Drvo.Pronadji(p);
        return cvor ? &cvor->_Sadrzaj : 0;
        }
    //---------------------------------------------------------
    // Tip iteratora KatalogReci::iterator.
    //---------------------------------------------------------
    typedef BinarnoDrvo<PolozajReci>::iterator iterator;
    //---------------------------------------------------------
    // Metodi za pravljenje početnog i završnog iteratora.
    //---------------------------------------------------------
    iterator begin()
        { return _Drvo.begin(); }
    iterator end()
        { return _Drvo.end(); }
    //---------------------------------------------------------
    // Izbacivanje elemenata iz kataloga nije podržano.
    //---------------------------------------------------------
private:
    //---------------------------------------------------------
    // Članovi podaci.
    //---------------------------------------------------------
    BinarnoDrvo<PolozajReci> _Drvo;
};
```

Zbog toga što je neophodno da katalog menja čvorove drveta, moramo omogućiti pristup privatnim delovima klase `BinarnoDrvo::Cvor`. Možemo te delove učiniti javnim ili i klasu `KatalogReci` proglasiti za prijatelja klase `Cvor`.

Primer za testiranje kataloga modifikujemo tako da nam omogućava da proverimo kako se ponaša sa rečima i rednim brojevima rečenica:

```
// Testiranje klase KatalogReci
KatalogReci katalog;

// Dodajemo par reči i rednih brojeva rečenica.
// Kako katalog.Dodaj() vraća PoložajReči*
// možemo odmah nastaviti sa dodavanjem broja
katalog.Dodaj("a")->recenice.Dodaj(1);
// Dodavanje je u klasi BinarnoDrvo implementirano tako da
// se ponaša kao traženje ako objekat već postoji,
// pa sledećom naredbom ne dodajemo novu reč već samo
// novi redni broj rečenice uz postojeću reč.
katalog.Dodaj("a")->recenice.Dodaj(2);
katalog.Dodaj("a")->recenice.Dodaj(3);
// Ovde dodajemo i novu reč i novi redni broj rečenice.
katalog.Dodaj("b")->recenice.Dodaj(4);
katalog.Dodaj("b")->recenice.Dodaj(5);
katalog.Dodaj("c")->recenice.Dodaj(6);
katalog.Dodaj("c")->recenice.Dodaj(7);
katalog.Dodaj("c")->recenice.Dodaj(8);

// Tražimo i ispisujemo ako smo pronašli
PolozajReci* p;
p = katalog.Pronadji("a");
if( p )
    cout << p->rec << endl;
p = katalog.Pronadji("d");
if( p )
    cout << p->rec << endl;

// Ispisujemo sve reči iz kataloga
for( KatalogReci::iterator i=katalog.begin();
                           i!=katalog.end(); i++ )
    {
    cout << (*i).rec << ": ";
    // Nakon reči ispisujemo i redne brojeve rečenica
    for( Skup<int>::iterator j=(*i).recenice.begin();
                             j!=(*i).recenice.end(); j++ )
        cout << (*j) << ' ';
    cout << endl;
    }
cout << endl;
```

Radi! Sada već imamo ono što nam je bilo potrebno. Možemo pokušati da malo popravimo neke metode kako bi upotreba bila jednostavnija. Pri pisanju klase `KatalogReci` predstavljalo je određen problem to što metodi klase `BinarnoDrvo` vraćaju pokazivače na čvor, koji su korisniku te klase (tj. nama) potpuno nevažni. Jedino što korisnika zanima je sadržaj čvorova, a ne sami čvorovi (bar dok ne postoji mogućnost brisanja čvorova)! Zato je dobro da uvedemo manje izmene u neke metode klase `BinarnoDrvo`. Posle toga klasa `KatalogReci` više ne mora biti prijatelj klase `Cvor`.

```
template< class T >
class BinarnoDrvo
{ ...
    T& Dodaj( const T& s );
    T* Pronadji( const T& s );
```

```
...
};

//---------------------------------------------------------------
// Dodavanje novog čvora sa novim elementom, ako takav ne postoji.
//---------------------------------------------------------------
template< class T >
T& BinarnoDrvo<T>::Dodaj( const T& s )
{
    Cvor*& p = PronadjiPokazivacNaCvor( s );
    if( !p )
        p = new Cvor(s);
    return p->_Sadrzaj;
}

//---------------------------------------------------------------
// Pronalaženje elementa u drvetu.
//---------------------------------------------------------------
template< class T >
T* BinarnoDrvo<T>::Pronadji( const T& s )
{
    Cvor* c = PronadjiPokazivacNaCvor( s );
    return c ? &c->_Sadrzaj : 0;
}
```

Štaviše, klasa `BinarnoDrvo::Cvor` može da se definiše kao privatna u šablonu `BinarnoDrvo`. Odgovarajuću izmenu ostavljamo za vežbu.

Menjamo i klasu `KatalogReci` na mestima na kojima smo upotrebljavali ove metode:

```
class KatalogReci
{
public:
    PolozajReci* Dodaj( const string& rec )
        {
        PolozajReci p;
        p.rec = rec;
        return &_Drvo.Dodaj( p );
        }

    PolozajReci* Pronadji( const string& rec )
        {
        PolozajReci p;
        p.rec = rec;
        return _Drvo.Pronadji(p);
        }
    ...
};
```

Primetimo da bi trebalo zabraniti menjanje elemenata drveta na način koji bi poremetio uređenje, ali da to nije jednostavno. Umesto toga, potrudićemo se da klasa `KatalogReci` bude bezbedna u tom smislu, kao što je već slučaj sa klasom `Skup`.

Sličan problem koji smo ispravili na klasi `BinarnoDrvo` postoji i u samoj klasi `KatalogReci`, jer ukoliko dodajemo neku reč, sigurno nas neće zanimati da preko objekta klase `PolozajReci` pristupamo opet samoj upravo dodatoj reči već isključivo podacima koji stoje uz reč, odnosno skupu rednih brojeva strana. Štaviše, kada smo sigurni da postoji

rezultat dobro je, kao u klasi `BinarnoDrvo`, da vratimo podatak po referenci a ne posredstvom pokazivača. Slično, sada ćemo i kao rezultat traženja vraćati pokazivač na skup rednih brojeva rečenica a ne na `PolozajReci`, jer tako na jednostavan način onemogućavamo da se promeni sama reč.

```
class KatalogReci
{
public:
    Skup<int>& Dodaj( const string& rec )
        {
        PolozajReci p;
        p.rec = rec;
        return _Drvo.Dodaj( p ).recenice;
        }
    Skup<int>* Pronadji( const string& rec )
        {
        PolozajReci p;
        p.rec = rec;
        PolozajReci* q = _Drvo.Pronadji(p);
        return q ? &q->recenice : 0;
        }
    ...
};
```

Sada je sve malo lepše. Nije dobro ostavljati kod u obliku u kome je po prvi put „proradio“. Često se uz malo truda kod može popraviti tako da se kasnije jednostavnije upotrebljava. Ako popravke „ostavimo za kasnije“ verovatno ćemo morati da menjamo sve više i više koda.

Naravno, ponovo bi trebalo prevesti i proveriti program. Neophodno je da najpre prilagodimo primer novom ponašanju klase `KatalogReci`:

```
// Testiranje klase KatalogReci
KatalogReci katalog;

// Dodajemo par reči i rednih brojeva rečenica.
// Kako katalog.Dodaj() vraća PoložajReči*
// možemo odmah nastaviti sa dodavanjem broja
katalog.Dodaj("a").Dodaj(1);
// Dodavanje je u klasi BinarnoDrvo implementirano tako da
// se ponaša kao traženje ako objekat već postoji,
// pa sledećom naredbom ne dodajemo novu reč već samo
// novi redni broj rečenice uz postojeću reč.
katalog.Dodaj("a").Dodaj(2);
katalog.Dodaj("a").Dodaj(3);
// Ovde dodajemo i novu reč i novi redni broj rečenice.
katalog.Dodaj("b").Dodaj(4);
katalog.Dodaj("b").Dodaj(5);
katalog.Dodaj("c").Dodaj(6);
katalog.Dodaj("c").Dodaj(7);
katalog.Dodaj("c").Dodaj(8);

// Tražimo i ispisujemo ako smo pronašli
Skup<int>* p;
p = katalog.Pronadji("a");
```

```
if( p )
    cout << "a" << endl;
p = katalog.Pronadji("d");
if( p )
    cout << "d" << endl;

// Ispisujemo sve reči iz kataloga
...
```

Nije potrebno mnogo mašte da bi se katalog posmatrao kao vrsta niza, kome indeksi nisu redni brojevi (tj. oznake položaja elementa u nizu) nego reči. Kao što za dati niz i dati indeks možemo jednostavno i brzo pristupiti odgovarajućem elementu niza, tako i za dati katalog i datu reč možemo isto tako jednostavno i tek nešto sporije pristupiti odgovarajućem elementu kataloga. Pa zašto onda ne bismo upotrebljavali i istu sintaksu? Pokušajmo da definišemo `operator[]` klase `KatalogReci`, tako da izvodi traženje elementa i vraća referencu na pronađene podatke ako je element pronađen, a u suprotnom dodaje novi element i vraća referencu na taj element. Već smo videli da metod `Dodaj` obavlja obe ove operacije: i traženje i opciono dodavanje. Preostaje nam samo da na odgovarajući način definišemo operator:

```
class KatalogReci
{
public:
    ...
    //-------------------------------------------------------------
    // Pronalaženje elementa ili njegovo dodavanje.
    //-------------------------------------------------------------
    Skup<int>& operator []( const string& rec )
        { return Dodaj( rec ); }
    ...
};
```

Sada možemo i da ga primenimo u segmentu koda koji nam služi za testiranje, jednostavno zamenjujući pozive metoda `Dodaj`:

```
...
katalog["a"].Dodaj(1);
katalog["a"].Dodaj(2);
katalog["a"].Dodaj(3);
katalog["b"].Dodaj(4);
katalog["b"].Dodaj(5);
katalog["c"].Dodaj(6);
katalog["c"].Dodaj(7);
katalog["c"].Dodaj(8);
...
```

Uopštavanje klase KatalogReci

Naš katalog reči sada može da obavlja potrebne operacije. Iako bismo sasvim mirno mogli da pređemo na rešavanje preostalih delova ovog zadatka, radi vežbe ćemo još malo vremena posvetiti katalogu. Kako bismo ga mogli uopštiti da može da se upotrebljava i u drugim

situacijama?[9] Naravno, rešenje je da se od naše klase napravi šablon. Ono što nam predstavlja najveći problem jeste što se ponegde reč i redni brojevi posmatraju kao zasebni objekti, a negde kao celina, pa naizgled nije sasvim jednostavno to uopštiti. Rešenje je da se počne od uopštavanja podatka koji čuvamo u katalogu, tj. naše klase `PolozajReci`. Ona je do sada imala dva člana: reč (tj. ključ po kome se pristupa podacima) i rečenice (tj. podatak koji odgovara konkretnom ključu). Jedan način rešavanja ovog problema je da se napravi šablon čiji su parametri tip ključa (u našem slučaju to je tip reči, tj. `string`) i tip podatka koji mu odgovara (u našem slučaju to je tip rečenica, tj. `Skup<int>`):

```
//-------------------------------------------------------------
// Klasa ElementKataloga
//-------------------------------------------------------------
// Predstavlja par ključa i odgovarajućeg podatka.
//-------------------------------------------------------------
template< class TipKljuca, class TipPodatka >
class ElementKataloga
{
public:
    TipKljuca   Kljuc;
    TipPodatka  Podatak;

    bool operator <( const ElementKataloga& e ) const
        { return Kljuc < e.Kljuc; }
};
```

Sada je jasno da je i sam katalog određen sa ista dva parametra. Štaviše, zbog toga što je tip elementa tesno vezan za tip kataloga, umesto samostalne klase `ElementKataloga` napravićemo identičnu klasu `Katalog::Element`:

```
//-------------------------------------------------------------
// Klasa Katalog
//-------------------------------------------------------------
// Predstavlja uopšteni katalog.
//-------------------------------------------------------------
template< class TipKljuca, class TipPodatka >
class Katalog
{
public:
    //---------------------------------------------------------
    // Klasa Element
    //---------------------------------------------------------
    // Predstavlja par ključa i odgovarajućeg podatka.
    //---------------------------------------------------------
    class Element
    {
    public:
        TipKljuca   Kljuc;
        TipPodatka  Podatak;
```

[9] Stoji da nam to nije neophodno za rešavanje zadatka, ali razmatranje ovog pitanja može značajno doprineti boljem razumevanju klase `map` iz standardne biblioteke.

```
        bool operator <( const Element& e ) const
            { return Kljuc < e.Kljuc; }
    };

    //-------------------------------------------------------------
    // Dodavanje elemenata.
    //-------------------------------------------------------------
    TipPodatka& Dodaj( const TipKljuca& k )
        {
        Element e;
        e.Kljuc = k;
        return _Drvo.Dodaj( e ).Podatak;
        }

    //-------------------------------------------------------------
    // Pristupanje podatku koji odgovara datom ključu.
    //-------------------------------------------------------------
    TipPodatka& operator []( const TipKljuca& rec )
        { return Dodaj( rec ); }

    //-------------------------------------------------------------
    // Provera da li katalog sadrži dati element.
    //-------------------------------------------------------------
    TipPodatka* Pronadji( const TipKljuca& k )
        {
        Element e;
        e.Kljuc = k;
        Element* q = _Drvo.Pronadji(e);
        return q ? &q->Podatak : 0;
        }

    //-------------------------------------------------------------
    // Tip iteratora Katalog::iterator.
    //-------------------------------------------------------------
    typedef BinarnoDrvo<Element>::iterator iterator;

    //-------------------------------------------------------------
    // Metodi za pravljenje početnog i završnog iteratora
    //-------------------------------------------------------------
    iterator begin()
        { return Drvo.begin(); }
    iterator end()
        { return Drvo.end(); }

    //-------------------------------------------------------------
    // Izbacivanje elemenata iz kataloga nije podržano.
    //-------------------------------------------------------------

private:
    //-------------------------------------------------------------
    // Članovi podaci.
    //-------------------------------------------------------------
    BinarnoDrvo<Element> _Drvo;
};
```

Potrebno je još da definišemo tip `KatalogReci`:

```
typedef Katalog< string, Skup<int> > KatalogReci;
```

Nakon što u primerima zamenimo nazive članova podataka `rec` i `recenica` u `Kljuc` i `Podatak`, možemo prevesti i testirati program:

```
...
for( KatalogReci::iterator i=katalog.begin();
                           i!=katalog.end(); i++ )
    {
    cout << (*i).Kljuc << ": ";
    // Nakon reči ispisujemo i redne brojeve rečenica
    for( Skup<int>::iterator j=(*i).Podatak.begin();
                             j!=(*i).Podatak.end(); j++ )
        cout << (*j) << ' ';
    cout << endl;
    }
cout << endl;
```

Primena klase map

Klasa `Katalog` je sasvim pristojna klasa sa mogućnostima primene koje su dužne poštovanja. Možemo je koristiti za pravljenje telefonskih imenika, modela direktorijuma fajlova i razne druge vrste kataloga. Zar ne bi bilo lepo da nešto slično postoji u standardnoj biblioteci programskog jezika C++? Pa, ne samo da bi bilo lepo nego je već tako. Srećom po mnoge programere u standardnu biblioteku programskog jezika C++ ulazi šablon klase `map` koji radi sve ono što radi naš `Katalog` i još dosta toga. Detaljno opisivanje principa funkcionisanja klase `map` bi bilo upravo ponavljanje opisa `Kataloga` navedenih na nekoliko prethodnih strana.

Ukratko, klasa `map` je katalog elemenata koji se sastoje od ključa i odgovarajućeg podatka. Elementima se pristupa posredstvom ključa. Naravno, moguće je koristiti iteratore na uobičajen način. Za dodavanje elemenata najčešće se upotrebljava `operator[]`, na potpuno isti način kao kod klase `Katalog`. Za pretraživanje se najčešće upotrebljava metod `find`, koji (za razliku od našeg metoda `Pronadji`) za rezultat ima iterator koji ukazuje na pronađeni element (ako traženi element postoji) ili završni iterator (ako traženi element ne postoji). Tip elemenata kolekcije map je šablon `pair<const TipKljuča,TipPodatka>` koji je funkcionalno ekvivalentan našoj klasi `Katalog::Element`. Elementi para se zovu `first` i `second` i moguće im je neposredno pristupati, pri čemu se ne može menjati jednom postavljena vrednost ključa. Za upotrebu šablona klase `map` neophodno je uključiti zaglavlje `map` (videti *10.8.3 Katalog*, na strani 374).

Izmenimo prethodni probni primer tako da umesto naših klasa koristi šablon klase `map`:

```
// Naravno, menjamo tip kolekcije.
map<string,Skup<int> > katalog;
// Dodavanje je potpuno isto.
katalog["a"].Dodaj(1);
katalog["a"].Dodaj(2);
katalog["a"].Dodaj(3);
katalog["b"].Dodaj(4);
katalog["b"].Dodaj(5);
katalog["c"].Dodaj(6);
katalog["c"].Dodaj(7);
```

```
katalog["c"].Dodaj(8);

// Traženje se menja jer se menja tip rezultata,
// a sa njim i način provere da li je nešto pronađeno.
map<string,Skup<int> >::iterator p;
p = katalog.find("a");
if( p != katalog.end() )
    cout << (*p).first << endl;
p = katalog.find("d");
if( p != katalog.end() )
    cout << (*p).first << endl;

// Promene pri radu sa iteratorima su minimalne.
// Promenjen je tip kataloga i nazivi članova elemenata.
for( map<string,Skup<int> >::iterator i=katalog.begin();
                            i!=katalog.end(); i++ )
    {
    cout << (*i).first << ": ";
    for( Skup<int>::iterator j=(*i).second.begin();
                              j!=(*i).second.end(); j++ )
        cout << (*j) << ' ';
    cout << endl;
    }
cout << endl;
```

Naravno, na samom početku nam je neophodno i:

```
#include <map>
```

Testiranje pokazuje da klasa `map` radi svoj posao sasvim dobro. Možemo polako obrisati šablon klase `Katalog` (možda uz malo sentimentalnosti prema svom delu, ali bez preterivanja) i predefinisati tip `KatalogReci`:

```
typedef map< string, Skup<int> > KatalogReci;
```

Sada u test primeru možemo vratiti stari dobri naziv tipa i ponovo sve prevesti i proveriti:

```
KatalogReci katalog;
...
KatalogReci::iterator p;
...
for( KatalogReci::iterator i=katalog.begin();
                            i!=katalog.end(); i++ )
    {...}
...
```

Primena klase set

Kao što u standardnoj biblioteci postoji klasa `map` koja predstavlja napredniju implementaciju kataloga nego što je naš `Katalog`, tako postoji i klasa `set` kao implementacija skupa. Osim sličnosti u ponašanju postoji i velika sličnost u načinu implementacije, jer klase `set` i `map` imaju dosta zajedničkih osobina, kao što je to bio slučaj i sa našim klasama `Skup` i `Katalog`.

Osnovni principi funkcionisanja šablona klase `set` odgovaraju već predstavljenim i implementiranim principima funkcionisanja šablona klase `Skup`. Kao i u našem slučaju, `set`

raspolaže osnovnim operacijama uobičajenim za skupove. Elementi se dodaju metodom `insert`, koji odgovara našem metodu `Dodaj`. Umesto našeg metoda `Sadrži`, za proveru da li je nešto element skupa upotrebljava se metod `count`, koji izračunava broj pojavljivanja elementa u skupu. Ne dajte se zbuniti čudnim opisom koji se naizgled ne uklapa u uobičajeni pojam skupa koji ne dopušta ponavljanje elemenata, jer ovaj metod klase `set` uvek vraća 0 ili 1. Za potrebe ovog zadatka potrebno je još i da napomenemo da se iteratori mogu upotrebljavati na uobičajen način. Neophodne definicije i deklaracije u vezi sa klasom set nalaze se u zaglavlju `<set>` (videti *10.8.1 Skup*, na strani 371).

Prionimo na posao: svuda gde smo koristili `Skup`, sada možemo da pređemo na upotrebu klase `set`. Srećom, toga još uvek nema mnogo i svodi se na probni primer:

```
typedef map< string, set<int> > KatalogReci;

...
katalog["a"].insert(1);
katalog["a"].insert(2);
katalog["a"].insert(3);
katalog["b"].insert(4);
katalog["b"].insert(5);
katalog["c"].insert(6);
katalog["c"].insert(7);
katalog["c"].insert(8);
...
for( KatalogReci::iterator i=katalog.begin();
                           i!=katalog.end(); i++ )
    {
    cout << (*i).first << ": ";
    for( set<int>::iterator j=(*i).second.begin();
                            j!=(*i).second.end(); j++ )
        cout << (*j) << ' ';
    cout << endl;
    }
cout << endl;
```

Sada nam više nije potreban šablon klase `Skup`, ali ni šablon klase `BinarnoDrvo`, pa možemo oboje obrisati. Bila je to dobra vežba. Umesto da tugujemo, bolje bi bilo da testiramo napisano.

Evidentiranje reči

Sada, konačno, možemo da evidentiramo sve reči pronađene u tekstu. Potrebno je da obezbedimo odgovarajuće članove podatke u klasi `Tekst` i da izmenimo metod `EvidentirajRec`. Radi provere ćemo napisati i metod `IspisiReci` tako da uz svaku reč ispisuje i spisak rednih brojeva svih rečenica u kojima se reč nalazi, čime dobijamo indeks reči pronađenih u tekstu:

```
class Tekst
{
public:
    ...
```

```
    //-------------------------------------------------------------
    // Pomoćni metodi koji nam u fazi testiranja služe za proveru
    // da li su redovi i reči teksta dobro pročitani.
    //-------------------------------------------------------------
    void IspisiRecenice( ostream& ostr ) const;
    void IspisiReci( ostream& ostr ) const;
    ...
private:
    ...
    //-------------------------------------------------------------
    // Evidentiranje reči u rečenici sa datim rednim brojem.
    //-------------------------------------------------------------
    void EvidentirajRec( const string& rec, int recenica )
        { _Reci[rec].insert(recenica); }
    //-------------------------------------------------------------
    // Članovi podaci
    //-------------------------------------------------------------
    vector<string>          _Recenice;
    map<string,set<int> >   _Reci;
};
//-----------------------------------------------------------------
// Pomoćni metod koji nam u fazi testiranja služi za proveru
// da li su reči teksta dobro pročitane. Ispisuje indeks reči.
//-----------------------------------------------------------------
void Tekst::IspisiReci( ostream& ostr ) const
{
    map<string,set<int> >::const_iterator
        i = _Reci.begin(),
        e = _Reci.end();
    for( ; i!=e; i++ ){
        ostr << i->first << " : ";
        set<int>::const_iterator
            k = i->second.begin(),
            l = i->second.end();
        for( ; k!=l; k++ )
            ostr << (*k) << ' ';
        ostr << endl;
        }
}
```

Posle svega, sada zaista možemo da kažemo da je evidentiranje reči dovršeno. Naravno, potrebno je da proverimo šta smo uradili:

```
//-----------------------------------------------------------------
// Ovo je funkcija koja izvodi pretraživanje.
//-----------------------------------------------------------------
void pretrazivanjeTeksta( const char* nazivDat )
{
    // Analiziramo tekst.
    Tekst t( nazivDat );
    // Pretražujemo. :)
    t.IspisiReci( cout );
}
```

Korak 5 - Korisnički interfejs

Pročitali smo rečenice i evidentirali reči. Došli smo do mesta na kome možemo početi da se bavimo pretraživanjem. Pre nego što zađemo u sam problem pretraživanja, napravimo najpre funkcije i metode koji obezbeđuju komunikaciju sa korisnikom. Najpre menjamo funkciju `pretrazivanjeTeksta` dopunjavajući je kodom za višestruko prihvatanje upita i odgovarajuće pretraživanje. Za sada nećemo izvoditi traženje, ali ćemo ispisati improvizovan rezultat:

```
//---------------------------------------------------------------
// Ovo je funkcija koja izvodi pretraživanje.
//---------------------------------------------------------------
void pretrazivanjeTeksta( const char* nazivDat )
{
    // Analiziramo tekst.
    Tekst t( nazivDat );

    // Ponavljamo komunikaciju sa korisnikom sve dok
    // korisnik ne zahteva prekid.
    string upit;
    while(1){
        // Čitamo upit.
        cout << "\nUpisite upit u obliku: [+|-]rec {...}\n"
             << "  - opcioni znak '+' "
                "znaci da se rec mora pojaviti u recenici\n"
             << "  - opcioni znak '-' "
                "znaci da se rec ne sme pojaviti u recenici\n"
             << "  - reci se razdvajaju razmakom\n"
             << "  - za kraj upisite prazan red\n\n";
        getline( cin, upit );
        if( upit.size() == 0 || !cin )
            break;

        // Tražimo i ispisujemo rezultat.
        set<int> pronadjene;
        t.Trazi( upit, pronadjene );
        t.IspisiRecenice( cout, pronadjene );
        }
}
```

Neophodno je da klasi `Tekst` dodamo metode `Trazi` i `IspisiRecenice`. Metod `Trazi` ima dva argumenta: tekst upita i skup rednih brojeva rečenica. Zbog toga što vraćanje objekata kao rezultata funkcija i metoda dovodi do njihovog kopiranja, a to može biti prilično skupo, bolje je da se skup rednih brojeva pronađenih rečenica prenosi kao promenljivi argument, po referenci. Još uvek nećemo ništa pretraživati. Improvizovaćemo izračunavanje rezultata kako bismo mogli proveriti do sada napisan kod. Metod `IspisiRecenice` ispisuje dati skup rečenica u dati tok. Primetimo da već postoji istoimeni metod koji ispisuje sve rečenice, ali kako ova dva metoda imaju različite argumente (postojeći metod ima samo jedan argument – tok) to ne predstavlja problem:

```
class Tekst
{
public:
```

```
...
    //-------------------------------------------------------------
    // Traženje rečenica.
    //-------------------------------------------------------------
    void Trazi( const string& upit, set<int>& pronadjene ) const;

    //-------------------------------------------------------------
    // Ipisivanje pronadjenih rečenica.
    //-------------------------------------------------------------
    void IspisiRecenice( ostream& ostr,
                         const set<int>& pronadjene ) const;
...
};

//-------------------------------------------------------------------
// Traženje rečenica.
//-------------------------------------------------------------------
void Tekst::Trazi(const string& upit, set<int>& pronadjene ) const
{
    // Simuliramo traženje.
    // OPREZ! Neće raditi ako tekst nema dovoljno rečenica.
    if( upit=="x" ){
        pronadjene.insert(1);
        pronadjene.insert(12);
        pronadjene.insert(7);
        }
}

//-------------------------------------------------------------------
// Ipisivanje pronađenih rečenica.
//-------------------------------------------------------------------
void Tekst::IspisiRecenice( ostream& ostr,
                            const set<int>& pronadjene ) const
{
    if( pronadjene.empty() )
        ostr << "\nNe postoje recenice "
                "koje zadovoljavaju trazeni uslov.\n";
    else{
        set<int>::const_iterator
            ii = pronadjene.begin(),
            ie = pronadjene.end();
        ostr << endl;
        for( ; ii!=ie; ii++ )
            ostr << (*ii) << ": " << _Recenice[*ii] << endl;
        }
}
```

Korak 6 - Analiza upita

Da bi se moglo započeti traženje po datom uslovu, prvo je neophodno analizirati upit i prepoznati tri skupa traženih reči: tražene (bez predznaka), obavezne (sa predznakom '+') i zabranjene (sa predznakom '-'). Ta analiza nije vezana za konkretan tekst koji se pretražuje, pa ćemo odgovarajući metod napisati kao statički metod klase `Tekst`. Da se podsetimo: statički metod se koristi nezavisno od objekata klase (mada sintaksa poziva može imati i uobičajeni oblik) i ne raspolaže implicitnim pokazivačem `this`, pa ne može pristupati nestatičkim metodima i članovima klase bez eksplicitnog navođenja objekta. I ovde ćemo, kao i u

metodima za čitanje teksta i izdvajanje rečenica i reči, upotrebljavati metode klase `string` za traženje:

```
class Tekst
{
...
private:
    //-------------------------------------------------------------
    // Izdvajanje delova upita.
    //-------------------------------------------------------------
    static void AnalizaUpita(
                    string upit,
                    set<string>& obavezne,
                    set<string>& trazene,
                    set<string>& zabranjene
                    );
...
};

//-------------------------------------------------------------------
// Izdvajanje delova upita.
//-------------------------------------------------------------------
void Tekst::AnalizaUpita(
    string upit,
    set<string>& obavezne,
    set<string>& trazene,
    set<string>& zabranjene
    )
{
    // Dopušteni znaci.
    static const string slova =
        "abcdefghijklmnopqrstuvwxyz"
        "1234567890";
    // Prvi znak reči je slovo, cifra ili + ili -.
    static const string slovapm =
        "abcdefghijklmnopqrstuvwxyz"
        "1234567890+-";

    // Zanimaju nas samo mala slova.
    for( int i=upit.size()-1; i>=0; i-- )
        if( upit[i]>='A' && upit[i]<='Z' )
            upit[i] += 'a' - 'A';

    // Izdvajamo reč po reč.
    unsigned kraj = 0;
    while( kraj < upit.size()){
        // Tražimo početak sledeće reči.
        unsigned pocetak = upit.find_first_of( slovapm, kraj );
        if( pocetak == string::npos )
            break;
        // Tražimo kraj reči.
        kraj = upit.find_first_not_of( slova, pocetak+1 );
        if( kraj == string::npos )
            kraj = upit.size();
```

```
            // Izdvajamo reč i svrstavamo je u odgovarajući skup.
            if( upit[pocetak]=='-' )
                zabranjene.insert(
                    upit.substr( pocetak+1, kraj-pocetak-1 )
                    );
            else if( upit[pocetak]=='+' )
                obavezne.insert(
                    upit.substr( pocetak+1, kraj-pocetak-1 )
                    );
            else
                trazene.insert( upit.substr( pocetak, kraj-pocetak ));
            }
    }
```

Da bismo mogli proveriti napisani kod, dopišimo kod za testiranje u metod `Trazi`:

```
//---------------------------------------------------------------
// Traženje rečenica.
//---------------------------------------------------------------
void Tekst::Trazi(const string& upit, set<int>& pronadjene) const
{
    set<string> obavezne, trazene, zabranjene;
    AnalizaUpita( upit, obavezne, trazene, zabranjene );

    // Testiranje analize upita.
    set<string>::iterator i;
    cout << "Obavezne: ";
    for( i=obavezne.begin(); i!=obavezne.end(); i++ )
        cout << (*i) << ' ';
    cout << "\nTrazene: ";
    for( i=trazene.begin(); i!=trazene.end(); i++ )
        cout << (*i) << ' ';
    cout << "\nZabranjene: ";
    for( i=zabranjene.begin(); i!=zabranjene.end(); i++ )
        cout << (*i) << ' ';
    cout << endl;
}
```

Korak 7 - Traženje

Pročitali smo tekst, izdvojili reči, prihvatili upit, analizirali ga i imamo spreman kod za ispisivanje rezultata. Ostaje nam još da tražimo. Na početku (ako se posle svega što smo do sada uradili ovo može nazvati početkom) ćemo razmotriti kako bi traženje trebalo da izgleda:

- ako je skup obaveznih reči prazan, tada je potrebno naći sve rečenice koje sadrže bar jednu od traženih reči i nijednu zabranjenu;
- ako skup obaveznih reči nije prazan, tada skup traženih neobaveznih reči nema značaja i možemo ga zanemariti (zapravo, može se upotrebljavati za rangiranje pronađenih rezultata, ali time se ovde nećemo baviti već ćemo ostaviti za vežbu), a potrebno je naći rečenice koje sadrže sve obavezne reči i nijednu zabranjenu.

Znači, prvi korak je ustanovljavanje da li imamo obaveznih reči ili ne. Ako ih nema, koristimo tražene reči i računamo uniju odgovarajućih skupova rečenica. Ako ih ima, koristićemo obavezne reči i računati presek odgovarajućih skupova rečenica. Napisaćemo

novi metod `Trazi`, koji očekuje skup potrebnih reči, logički podatak koji govori da li se računa presek (`true`) ili unija (`false`) i skup zabranjenih reči:

```
//-------------------------------------------------------------
// Traženje rečenica.
//-------------------------------------------------------------
void Tekst::Trazi( const string& upit, set<int>& pronadjene ) const
{
    set<string> obavezne, trazene, zabranjene;
    AnalizaUpita( upit, obavezne, trazene, zabranjene );
    if( obavezne.empty() )
        Trazi( trazene, false, zabranjene, pronadjene );
    else
        Trazi( obavezne, true, zabranjene, pronadjene);
}
```

Preostaje da napišemo novi metod `Trazi`:

```
class Tekst
{ ...
public:
    //-------------------------------------------------------------
    // Traženje rečenica.
    //-------------------------------------------------------------
    void Trazi( const string& upit, set<int>& pronadjene ) const;
    void Trazi( const set<string>& trazene, bool presek,
                const set<string>& zabranjene,
                set<int>& pronadjene ) const;
...
};
```

Da bismo napisali ovaj metod potrebno je da budemo u stanju da napišemo operacije unije, preseka i razlike skupova. Operacija unije je sasvim jednostavna, jer metod `insert` za dodavanje elementa skupu vodi računa da ne dođe do ponavljanja elemenata. Ako imamo skupove `prvi` i `drugi` i odgovarajući iterator `i`, prvom skupu dodajemo elemente drugog skupa na sledeći način:

```
for( i=drugi.begin(); i!=drugi.end(); i++ )
    prvi.insert(*i);
```

Da bi sve bilo još jednostavnije, klasa `set` ima metod `insert` koji kao argumente ima dva iteratora koji određuju elemente drugog skupa (ili bilo koje druge kolekcije koja podržava iteratore) koje je potrebno dodati prvom:

```
prvi.insert( drugi.begin(), drugi,end() );
```

Sa presekom je situacija nešto nezgodnija jer ne postoji odgovarajući metod klase `set` koji bi omogućio da ga neposredno izvedemo. Možemo ga implementirati koristeći pomoćni skup `presek`:

```
presek.clear();
```

```
for( i=drugi.begin(); i!=drugi.end(); i++ )
    if( prvi.count(*i) )
        presek.insert(*i);
prvi = presek;
```

Razlika se može izvesti slično preseku, uz korišćenje pomoćnog skupa `razlika`:

```
razlika.clear();
for( i=prvi.begin(); i!=prvi.end(); i++ )
    if( !drugi.count(*i) )
        razlika.insert(*i);
prvi = razlika;
```

Međutim, klasa `set` raspolaže metodom `erase` za uklanjanje elemenata. Tako se razlika može izračunati vrlo slično uniji:

```
for( i=drugi.begin(); i!=drugi.end(); i++ )
    prvi.erase(*i);
```

Štaviše, i za brisanje postoji metod `erase` koji prihvata iteratore, slično već opisanom metodu `insert`. Međutim, razlika se ne može napisati na sledeći način:

```
prvi.erase( drugi.begin(), drugi,end() );
```

jer ovaj metod `erase` zahteva da se iteratori odnose na skup iz koga se elementi uklanjaju! Ovde valja biti oprezan jer navedena upotreba nije sintaksno neispravna, pa prevodilac neće prijaviti grešku.

Primetimo da bi se neko mogao dosetiti da implemetira presek koristeći metod `erase`, bez upotrebe pomoćnog objekta `presek`:

```
for( i=prvi.begin(); i!=prvi.end(); i++ )
    if( !drugi.count(*i) )
        prvi.erase (*i);
```

Međutim, takvo rešenje nije ispravno! Iako je sintaksno potpuno u redu i prevodilac ne prijavljuje greške, ovo rešenje ima značajanu semantičku neispravnost. Naime, pri brisanju elementa skupa (kao i pri dodavanju elemenata) dolazi do promene interne strukture kolekcije (menja se izgled drveta), zbog čega iteratori prestaju da budu ispravni. Da sve bude još nezgodnije, neispravnost se ne mora ispoljavati za svaki primer, pa se može desiti da probni primeri prođu bez vidljivih problema. Slično važi i za druge tipove kolekcija iz standardne biblioteke.

Znajući sve ovo, sada možemo implementirati metod `Trazi`:

```
//---------------------------------------------------------------
// Traženje rečenica.
//---------------------------------------------------------------
void Tekst::Trazi(
            const set<string>& trazene,
            bool presek,
            const set<string>& zabranjene,
            set<int>& pronadjene
            ) const
{
```

```
        // Najpre tražimo potrebne rečenice.
        set<string>::const_iterator
            si = trazene.begin(),
            se = trazene.end();
        // Ponavljamo za svaku potrebnu reč.
        for( ; si!=se; si++ ){
            // Trazimo reč.
            map<string,set<int> >::const_iterator f = _Reci.find(*si);
            if( f != _Reci.end() )
                // Čak i kada računamo presek, prvi put ćemo
                // se odlučiti za uniju, jer je skup pronadjene
                // još uvek prazan, pa bi i presek bio prazan.
                if( presek && si!=trazene.begin() ){
                    // Računamo presek.
                    set<int> presek;
                    set<int>::const_iterator
                        ii = f->second.begin(),
                        ie = f->second.end();
                    for( ; ii!=ie; ii++ )
                        if( pronadjene.count(*ii))
                            presek.insert(*ii);
                    pronadjene = presek;
                    }
                // Ako je u pitanju unija...
                else
                    pronadjene.insert(
                                    f->second.begin(),
                                    f->second.end()
                                    );
            }
        // Sada tražimo zabranjene rečenice.
        si = zabranjene.begin();
        se = zabranjene.end();
        // Ponavljamo za svaku zabranjenu reč.
        for( ; si!=se; si++ ){
            // Trazimo reč.
            map<string,set<int> >::const_iterator f = _Reci.find(*si);
            if( f != _Reci.end() ){
                // Brišemo pronađene zabranjene rečenice.
                set<int>::const_iterator
                    ii = f->second.begin(),
                    ie = f->second.end();
                for( ; ii!=ie; ii++)
                    pronadjene.erase(*ii);
                }
            }
    }
```

Prevedimo program i proverimo:

```
pretrazi pretrazi.cpp
```

Upišimo uslov koji određuje uniju redova koji sadrže neku od datih reči:

```
tekst trazi
```

Rezultat je:

```
8: // Klasa Tekst
12: class Tekst
18: Tekst( const char* nazivDat );
22: void Trazi( const string& upit, set<int>& pronadjene ) const;
23: void Trazi( const set<string>& trazene, bool presek,
70: Tekst::Tekst( const char* nazivDat )
78: void Tekst::CitajRecenice( const char* nazivDat )
129: void Tekst::IspisiRecenice( ostream& ostr ) const
139: void Tekst::IzdvojiReci()
185: void Tekst::IspisiReci( ostream& ostr ) const
209: void Tekst::Trazi( const string& upit, set<int>& pronadjene )
const
215: Trazi( trazene, false, zabranjene, pronadjene );
217: Trazi( obavezne, true, zabranjene, pronadjene);
222: void Tekst::Trazi(
305: void Tekst::AnalizaUpita(
362: void Tekst::IspisiRecenice( ostream& ostr,
387: // Analiziramo tekst.
388: Tekst t( nazivDat );
412: Trazi( upit, pronadjene );
```

Pokušajmo sada sa presekom:

```
+tekst +trazi
```

Sada je rezultat:

```
209: void Tekst::Trazi( const string& upit, set<int>& pronadjene )
const
222: void Tekst::Trazi(
```

Da pokušamo i sa zabranjenim rečima:

```
tekst trazi -int -void
```

Rezultat je:

```
8: // Klasa Tekst
12: class Tekst
18: Tekst( const char* nazivDat );
70: Tekst::Tekst( const char* nazivDat )
215: Trazi( trazene, false, zabranjene, pronadjene );
217: Trazi( obavezne, true, zabranjene, pronadjene);
387: // Analiziramo tekst.
388: Tekst t( nazivDat );
412: Trazi( upit, pronadjene );
```

Još jedan primer:

```
+tekst +trazi -int
```

Rezultat je:

```
222: void Tekst::Trazi(
```

Možemo reći da je zadatak uspešno rešen.

Korak 8 - Podrška za naša slova

Jedna manjkavost ponuđenog rešenja je što ne podržava rad sa kodnom stranom 1250 već je ograničeno na ASCII. Takođe, na više mesta u programu se ponavljaju nizovi slova i postupak pretvaranja svih slova u okviru niske u mala slova. Da bismo obezbedili podršku za naša slova i izbegli pomenuto ponavljanje, sakupićemo sve informacije o slovima u klasu `Azbuka`. Ova klasa će imati samo statičke podatke i metode:

```
//-------------------------------------------------------------
// Klasa Azbuka
//-------------------------------------------------------------
// Sadrži informacije i postupke koji se tiču slova.
//-------------------------------------------------------------
class Azbuka
{
public:
    static const string slova;
    static const string praznine;
    static const string krajRecenice;

    // Konverzija niske u mala slova.
    static void uMalaSlova( string& upit );
};

const string Azbuka::slova =
    "abcdefghijklmnopqrstuvwxyz"
    "šđčćžŠĐČĆŽ"
    "1234567890";

const string Azbuka::praznine = " \t\r\n";
const string Azbuka::krajRecenice = ".?!";

// Konverzija niske u mala slova.
void Azbuka::uMalaSlova( string& upit )
{
    for( int i=upit.size()-1; i>=0; i-- )
        if( upit[i]>='A' && upit[i]<='Z' )
            upit[i] += 'a' - 'A';
        else
            switch( upit[i] ){
                case 'Š':
                    upit[i] = 'š';
                    break;
                case 'Đ':
                    upit[i] = 'đ';
                    break;
                case 'Č':
                    upit[i] = 'č';
                    break;
                case 'Ć':
                    upit[i] = 'ć';
                    break;
                case 'Ž':
                    upit[i] = 'ž';
                    break;
                }
}
```

Potrebno je izmeniti metode u kojima smo koristili podatke o slovima:

```
void Tekst::CitajRecenice( const char* nazivDat )
{
    ...
        // Tražimo početak sledeće rečenice u redu.
        unsigned pocetak =
            red.find_first_not_of( Azbuka::praznine, kraj );
    ...
        // Tražimo kraj rečenice.
        kraj = red.find_first_of( Azbuka::krajRecenice, pocetak );
        // Ako nema znaka za kraj, smatramo da je kraj
        // poslednji znak koji nije praznina, a takav
        // sigurno postoji, jer inače ne bismo imali
        // ni početak.
        if( kraj == string::npos )
            kraj = red.find_last_not_of( Azbuka::praznine );
    ...
}

void Tekst::AnalizaUpita(
    string upit, set<string>& obavezne,
    set<string>& trazene, set<string>& zabranjene
    )
{
    // Brišemo:
    // static const string slova = ...

    // Prvi znak reči je slovo, cifra ili + ili -.
    static const string slovapm = Azbuka::slova + "+-";
    Azbuka::uMalaSlova( upit );
    ...
        // Tražimo početak sledeće reči.
        unsigned pocetak = upit.find_first_of( slovapm, kraj );
        if( pocetak == string::npos )
            break;
        // Tražimo kraj reči.
        kraj = upit.find_first_not_of( Azbuka::slova, pocetak+1 );
    ...
}

void Tekst::IzdvojiReci()
{
    // Brišemo red:
    //  static const string slova = ...
...
    // Želimo samo mala slova.
    string recenica = _Recenice[i];
    Azbuka::uMalaSlova( recenica );
...
        // Tražimo početak reči.
        unsigned pocetak =
            recenica.find_first_of( Azbuka::slova, kraj );
...
        // Tražimo kraj reči.
        kraj = recenica.find_first_not_of( Azbuka::slova,
                                           pocetak );
```

```
...
}
```

6.3 Rešenje

```
#include <stdexcept>
#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <set>

using namespace std;

//---------------------------------------------------------------
// Klasa Azbuka
//---------------------------------------------------------------
// Sadrži informacije i postupke koji se tiču slova.
//---------------------------------------------------------------
class Azbuka
{
public:
    static const string slova;
    static const string praznine;
    static const string krajRecenice;

    // Konverzija niske u mala slova.
    static void uMalaSlova( string& upit );
};

const string Azbuka::slova =
    "abcdefghijklmnopqrstuvwxyz"
    "šđčćžŠĐČĆŽ"
    "1234567890";

const string Azbuka::praznine = " \t\r\n";

const string Azbuka::krajRecenice = ".?!";

// Konverzija niske u mala slova.
void Azbuka::uMalaSlova( string& upit )
{
    for( int i=upit.size()-1; i>=0; i-- )
        if( upit[i]>='A' && upit[i]<='Z' )
            upit[i] += 'a' - 'A';
        else
            switch( upit[i] ){
                case 'Š':
                    upit[i] = 'š';
                    break;
                case 'Đ':
                    upit[i] = 'đ';
                    break;
                case 'Č':
                    upit[i] = 'č';
                    break;
                case 'Ć':
                    upit[i] = 'ć';
```

```
                    break;
                case 'Ž':
                    upit[i] = 'ž';
                    break;
                }
}

//-------------------------------------------------------------
// Klasa Tekst
//-------------------------------------------------------------
// Predstavlja tekstualnu datoteku koju pretražujemo.
//-------------------------------------------------------------
class Tekst
{
public:
    //---------------------------------------------------------
    // Konstruktor.
    //---------------------------------------------------------
    Tekst( const char* nazivDat );

    //---------------------------------------------------------
    // Traženje rečenica.
    //---------------------------------------------------------
    void Trazi( const string& upit, set<int>& pronadjene ) const;
    void Trazi( const set<string>& trazene, bool presek,
                const set<string>& zabranjene,
                set<int>& pronadjene ) const;

    //---------------------------------------------------------
    // Ipisivanje pronađenih rečenica.
    //---------------------------------------------------------
    void IspisiRecenice( ostream& ostr,
                         const set<int>& pronadjene ) const;

    //---------------------------------------------------------
    // Pomoćni metodi koji nam u fazi testiranja služe za proveru
    // da li su redovi i reči teksta dobro pročitani.
    //---------------------------------------------------------
    void IspisiRecenice( ostream& ostr ) const;
    void IspisiReci( ostream& ostr ) const;

private:
    //---------------------------------------------------------
    // Čitanje teksta i izdvajanje i pamćenje rečenica.
    //---------------------------------------------------------
    void CitajRecenice( const char* nazivDat );

    //---------------------------------------------------------
    // Izdvajanje i evidentiranje reči iz pročitanih rečenica.
    //---------------------------------------------------------
    void IzdvojiReci();

    //---------------------------------------------------------
    // Evidentiranje reči u rečenici sa datim rednim brojem.
    //---------------------------------------------------------
    void EvidentirajRec( const string& rec, int recenica )
        { _Reci[rec].insert(recenica); }
```

```
    //-------------------------------------------------------------
    // Izdvajanje delova upita.
    //-------------------------------------------------------------
    static void AnalizaUpita(
                    string upit,
                    set<string>& obavezne,
                    set<string>& trazene,
                    set<string>& zabranjene
                    );

    //-------------------------------------------------------------
    // Članovi podaci
    //-------------------------------------------------------------
    vector<string>          _Recenice;
    map<string,set<int> >   _Reci;
};

//-------------------------------------------------------------
// Konstruktor.
//-------------------------------------------------------------
Tekst::Tekst( const char* nazivDat )
{
    CitajRecenice( nazivDat );
    IzdvojiReci();
}

//-------------------------------------------------------------
// Čitanje teksta i izdvajanje i pamćenje rečenica.
//-------------------------------------------------------------
void Tekst::CitajRecenice( const char* nazivDat )
{
    // Otvorimo datoteku.
    ifstream f( nazivDat );
    // Ako otvaranje nije uspelo, prijavimo grešku.
    if( !f )
        throw invalid_argument("Ne može se otvoriti datoteka!");

    string red;
    // Čitamo red po red.
    while( getline( f, red )){
        unsigned kraj = 0;
        // Izdvajamo rečenicu po rečenicu iz reda.
        while( kraj < red.size()){
            // Tražimo početak sledeće rečenice u redu.
            unsigned pocetak =
                red.find_first_not_of( Azbuka::praznine, kraj );
            // Ako ga nema, završili smo.
            if( pocetak == string::npos )
                break;
            // Tražimo kraj rečenice.
            kraj = red.find_first_of( Azbuka::krajRecenice,
                                      pocetak );
            // Ako nema znaka za kraj, smatramo da je kraj
            // poslednji znak koji nije praznina, a takav
            // sigurno postoji, jer inače ne bismo imali
            // ni početak.
            if( kraj == string::npos )
                kraj = red.find_last_not_of( Azbuka::praznine );
```

```
            // Pomerimo 'kraj' na prvi znak iza rečenice.
            kraj++;
            // Zapamtimo rečenicu.
            _Recenice.push_back(
                red.substr( pocetak, kraj-pocetak )
                );
            }
        }
}

//---------------------------------------------------------------
// Izdvajanje i evidentiranje reči iz pročitanih rečenica.
//---------------------------------------------------------------
void Tekst::IzdvojiReci()
{
    // Obrađujemo, redom, jednu po jednu rečenicu.
    unsigned n = _Recenice.size();
    for( unsigned i=0; i<n; i++ ){
        // Želimo samo mala slova.
        string recenica = _Recenice[i];
        Azbuka::uMalaSlova( recenica );
        // Izdvajamo reč po reč.
        unsigned kraj = 0;
        while( kraj < recenica.size()){
            // Tražimo početak reči.
            unsigned pocetak =
                recenica.find_first_of( Azbuka::slova, kraj );
            if( pocetak == string::npos )
                break;
            // Tražimo kraj reči.
            kraj = recenica.find_first_not_of( Azbuka::slova,
                                               pocetak );
            if( kraj == string::npos )
                kraj = recenica.size();
            // Evidentiramo reč
            EvidentirajRec(
                recenica.substr( pocetak, kraj-pocetak ),
                i
                );
            }
        }
}

//---------------------------------------------------------------
// Traženje rečenica.
//---------------------------------------------------------------
void Tekst::Trazi(
            const set<string>& trazene,
            bool presek,
            const set<string>& zabranjene,
            set<int>& pronadjene
            ) const
{
    // Najpre tražimo potrebne rečenice.
    set<string>::const_iterator
        si = trazene.begin(),
        se = trazene.end();
```

```
        // Ponavljamo za svaku potrebnu reč.
        for( ; si!=se; si++ ){
            // Trazimo reč.
            map<string,set<int> >::const_iterator f = _Reci.find(*si);
            if( f != _Reci.end() )
                // Čak i kada računamo presek, prvi put ćemo
                // se odlučiti za uniju, jer je skup pronadjene
                // još uvek prazan, pa bi i presek bio prazan.
                if( presek && si!=trazene.begin() ){
                    // Računamo presek.
                    set<int> presek;
                    set<int>::const_iterator
                        ii = f->second.begin(),
                        ie = f->second.end();
                    for( ; ii!=ie; ii++ )
                        if( pronadjene.count(*ii))
                            presek.insert(*ii);
                    pronadjene = presek;
                    }
                // Ako je u pitanju unija, to je jednostavno.
                else
                    pronadjene.insert(
                                    f->second.begin(),
                                    f->second.end()
                                    );
            }
        // Sada tražimo zabranjene rečenice.
        si = zabranjene.begin();
        se = zabranjene.end();
        // Ponavljamo za svaku zabranjenu reč.
        for( ; si!=se; si++ ){
            // Trazimo reč.
            map<string,set<int> >::const_iterator f = _Reci.find(*si);
            if( f != _Reci.end() ){
                // Brišemo pronađene zabranjene rečenice.
                set<int>::const_iterator
                    ii = f->second.begin(),
                    ie = f->second.end();
                for( ; ii!=ie; ii++)
                    pronadjene.erase(*ii);
                }
            }
    }

    //-------------------------------------------------------------
    // Traženje rečenica.
    //-------------------------------------------------------------
    void Tekst::Trazi( const string& upit, set<int>& pronadjene ) const
    {
        set<string> obavezne, trazene, zabranjene;
        AnalizaUpita( upit, obavezne, trazene, zabranjene );
        if( obavezne.empty() )
            Trazi( trazene, false, zabranjene, pronadjene );
        else
            Trazi( obavezne, true, zabranjene, pronadjene);
    }
```

```
//-------------------------------------------------------------
// Izdvajanje delova upita.
//-------------------------------------------------------------
void Tekst::AnalizaUpita(
    string upit,
    set<string>& obavezne,
    set<string>& trazene,
    set<string>& zabranjene
    )
{
    // Prvi znak reči je slovo, cifra ili + ili -.
    static const string slovapm = Azbuka::slova + "+-";
    Azbuka::uMalaSlova( upit );

    // Izdvajamo reč po reč.
    unsigned kraj = 0;
    while( kraj < upit.size()){
        // Tražimo početak sledeće reči.
        unsigned pocetak = upit.find_first_of( slovapm, kraj );
        if( pocetak == string::npos )
            break;
        // Tražimo kraj reči.
        kraj = upit.find_first_not_of( Azbuka::slova, pocetak+1 );
        if( kraj == string::npos )
            kraj = upit.size();
        // Izdvajamo reč isvrstavamo je u odgovarajući skup.
        if( upit[pocetak]=='-' )
            zabranjene.insert(
                upit.substr( pocetak+1, kraj-pocetak-1 )
                );
        else if( upit[pocetak]=='+' )
            obavezne.insert(
                upit.substr( pocetak+1, kraj-pocetak-1 )
                );
        else
            trazene.insert( upit.substr( pocetak, kraj-pocetak ));
        }
}

//-------------------------------------------------------------
// Pomoćni metod koji nam u fazi testiranja služi za proveru
// da li su redovi teksta dobro pročitani.
//-------------------------------------------------------------
void Tekst::IspisiRecenice( ostream& ostr ) const
{
    unsigned n = _Recenice.size();
    for( unsigned i=0; i<n; i++ )
        ostr << (i+1) << ":  " << _Recenice[i] << endl;
}

//-------------------------------------------------------------
// Pomoćni metod koji nam u fazi testiranja služi za proveru
// da li su reči teksta dobro pročitane. Ispisuje indeks reči.
//-------------------------------------------------------------
void Tekst::IspisiReci( ostream& ostr ) const
{
    map<string,set<int> >::const_iterator
        i = _Reci.begin(),
```

```
            e = _Reci.end();
        for( ; i!=e; i++ ){
            ostr << i->first << " : ";
            set<int>::const_iterator
                k = i->second.begin(),
                l = i->second.end();
            for( ; k!=l; k++ )
                ostr << (*k) << ' ';
            ostr << endl;
            }
    }

    //-----------------------------------------------------------------
    // Ipisivanje pronađenih rečenica.
    //-----------------------------------------------------------------
    void Tekst::IspisiRecenice( ostream& ostr,
                                const set<int>& pronadjene ) const
    {
        if( pronadjene.empty() )
            ostr << "\nNe postoje recenice "
                    "koje zadovoljavaju trazeni uslov.\n";
        else{
            set<int>::const_iterator
                ii = pronadjene.begin(),
                ie = pronadjene.end();
            ostr << endl;
            for( ; ii!=ie; ii++ )
                ostr << (*ii) << ": " << _Recenice[*ii] << endl;
            }
    }

    //-----------------------------------------------------------------
    // Funkcija koja pretražuje.
    //-----------------------------------------------------------------
    void pretrazivanjeTeksta( const char* nazivDat )
    {
        // Analiziramo tekst.
        Tekst t( nazivDat );

        // Ponavljamo komunikaciju sa korisnikom sve dok
        // korisnik ne zahteva prekid.
        string upit;
        while(1){
            // Čitamo upit.
            cout << "\nUpisite upit u obliku: [+|-]rec {...}\n"
                 << "  - opcioni znak '+' "
                    "znaci da se rec mora pojaviti u recenici\n"
                 << "  - opcioni znak '-' "
                    "znaci da se rec ne sme pojaviti u recenici\n"
                 << "  - reci se razdvajaju razmakom\n"
                 << "  - za kraj upisite prazan red\n\n";
            getline( cin, upit );
            if( upit.size() == 0 || !cin )
                break;

            // Tražimo i ispisujemo rezultat.
            set<int> pronadjene;
            t.Trazi( upit, pronadjene );
```

```
            t.IspisiRecenice( cout, pronadjene );
            }
    }

    //---------------------------------------------------------------
    // Glavna funkcija programa proverava parametre
    // i poziva funkciju za pretraživanje teksta.
    //---------------------------------------------------------------
    int main( int argc, char** argv )
    {
        try {
            if( argc < 2 )
                throw invalid_argument( "Nedostaje argument!" );

            pretrazivanjeTeksta( argv[1] );
            }

        catch( exception& e ){
            cout <<
                "Greška: \n"
                "   " << e.what() << endl <<
                "Upotreba: \n"
                "   pretraži <datoteka>\n";
            }

        return 0;
    }

    // Prva rečenica. Druga rečenica! Treća rečenica? Četvrta rečenica
```

6.4 Rezime

Za vežbanje ostavljamo dalje doradivanje rešenja ili neke od implementiranih klasa. Na primer:

- Postoje neke reči koje se veoma često upotrebljavaju u tekstovima i nema mnogo smisla pretraživati po njima (veznici, zamenice i sl.). Izmeniti postupke evidentiranja i pretraživanja podataka da bi se takve reči zanemarivale. Može se početi od nekog konstantnog skupa nepotrebnih reči, a zatim implementirati čitanje tog skupa reči iz datoteke.
- Ako upit sadrži neobavezne uslove pretraživanja, upotrebiti ih za rangiranje rezultata, tako da se viša ocena (i prvenstvo pri prikazivanju rezultata) daje redovima u kojima ima više neobaveznih reči.
- Zameniti implementaciju računanja unije, preseka i razlike primenom skupovnih operacija iz standardne biblioteke (videti odeljak *10.9.2 Operacije nad skupovima*, na strani 381). Uporediti performanse.
- Izmeniti `BinarnoDrvo` (i zatim `Katalog`) tako da kao rezultat traženja vraća iterator na pronađeni element. Obratite pažnju na to da iterator mora imati evidentirane sve pretke u čijoj je levoj grani, da bi mogao da se koristi za kretanje kroz drvo.
- Razmotriti pitanja robusnosti.

Upotrebljavajte program u kritičkom i kreativnom raspoloženju – pribeležite zamerke i ideje za unapređivanje programa i pokušajte da ih implementirate.

U okviru rešenja zadatka predstavljene su najvažnije osobine klasa `map` i `set`. Detaljniji opis ovih klasa, sa pregledom najvažnijih metoda, nalazi se u odeljcima *10.8.1 Skup*, na strani 371, i *10.8.3 Katalog*, na strani 374.

7 - Enciklopedija

7.1 Zadatak

Napisati program koji korisniku prikazuje podatke pohranjene u bazi podataka enciklopedije. Podržani tipovi podataka su *tekst*, *slika*, *zvuk* i *film*. Korisnik pokreće program *Enciklopedija* navodeći redni broj podatka, a program:

- prikazuje na standardnom izlazu podatke sa datim rednim brojem;
- prikazuje na standardnom izlazu redne brojeve, naslove i tipove svih povezanih podataka i
- u slučaju da podatak nije tekst, binarni sadržaj podatka zapisuje u datoteci sa datim imenom.

Argumenti programa se navode u obliku:

```
Enciklopedija <id> [<naziv binarne datoteke>]
```

Na primer:

```
Enciklopedija 42 bin.dat
Enciklopedija 38
```

Ukoliko se ne navede naziv datoteke, a radi se o podatku koji ima binarni sadržaj (slika, zvuk ili film), potrebno je ispisati raspoložive informacije uz napomenu da binarni sadržaj nije upisan u datoteku jer nije naveden njen naziv.

Na raspolaganju je biblioteka za upotrebu baze podataka enciklopedije (zaglavlje `BazaPodataka.h`) u kojoj su definisani tip `Podatak` i klasa `BazaPodataka`:

```
typedef map<string,string> Podatak;
```

```
class BazaPodataka {
public:
    static bool PročitajPodatak(
            int id, Podatak& x, string& tip
            );
    static void PročitajPovezane(
            int id, vector<int>& povezani
            );
};
```

Jedan `Podatak` se sastoji od kolekcije parova oblika (naziv atributa, vrednost atributa). Sve vrednosti atributa su zapisane tekstualno pa se po potrebi moraju prevoditi u cele ili realne brojeve. Ukoliko je u pitanju tekstualni podatak, opisan je atributima sa nazivima `id`, `naslov`, `tekst`, `autor`. Slike su opisane atributima sa nazivima `id`, `naslov`, `slika`, `napomena`, `širina`, `visina`, `autor`. Zvuk je opisan atributima sa nazivima `id`, `naslov`, `zvuk`, `napomena`, `trajanje`, `autor`. Filmovi su opisani atributima sa nazivima `id`, `naslov`, `film`, `napomena`, `trajanje`, `širina`, `visina`, `autor`.

Bazi podataka se pristupa posredstvom metoda klase `BazaPodataka`. Statički metod `PročitajPodatak` čita podatak sa rednim brojem `id` i upisuje ga u objekat `x`. Pri tome se niska sa opisom tipa podatka (koja može biti „`tekst`“, „`slika`“, „`zvuk`“ ili „`film`“) upisuje u parametar `tip`. Metod `PročitajPovezane` čita redne brojeve podataka koji imaju neke veze sa podatkom sa datim rednim brojem i upisuje ih u niz `povezani`.

Cilj zadatka

Rešavanjem ovog zadatka pokazaćemo:

- pravljenje detaljne specifikacije problema;
- projektovanje hijerarhije klasa bez osvrtanja na njihovu internu stukturu;
- neke elemente UML-a primenjene na analizu i projektovanje;
- spuštanje ponašanja niz hijerarhiju;
- podizanje ponašanja uz hijerarhiju;
- umetanje klasa u hijerarhiju;
- značaj dobrog imenovanja klasa i metoda;
- kako se dinamički prave objekti hijerarhije klasa;
- implementiranje hijerarhije klasa na osnovu napravljenog projekta;
- neke aspekte organizovanja teksta programa po datotekama.

Pretpostavljena znanja

Za praćenje ovog primera potrebno je poznavanje uglavnom svih elemenata programskog jezika C++, a pre svega:

- klasa standardne biblioteke: `string`, `vector` i `map`;
- standardne biblioteke tokova i datotečnih tokova;

- rada sa pokazivačima i dinamičkim strukturama podataka;
- izgradnje hijerarhija klasa;
- dinamičkog vezivanja metoda.

7.2 Rešavanje zadatka

Pri rešavanju zadatka aktivnosti ćemo podeliti u tri faze:

- analizu;
- projektovanje i
- implementaciju.

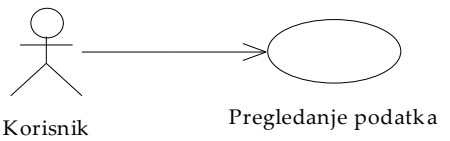
Najpre ćemo analizirati problem i detaljno definisati šta želimo da program radi. Zatim ćemo definisati klase i njihovo ponašanje, uzdržavajući se od bavljenja internom strukturom klasa i implementacijama metoda. Za opisivanje poslova i klasa upotrebljavaćemo dijagrame slučajeva i dijagrame klasa UML-a [Fowler 2003]. UML je vizualni jezik za projektovanje. Prostor nam ne dopušta opisivanje UML-a, ali će njegova primena biti sasvim jednostavna i intuitivna[10].

Rešavanje će biti izloženo u nekoliko koraka:

[10] Savremeni alati omogućavaju automatsko pravljenje koda klasa na osnovu dijagrama, kao i automatsko ažuriranje dijagrama na osnovu promena u kodu. Time se omogućava efikasniji rad i obezbeđuje međusobna saglasnost koda i dijagrama klasa.

Korak 1 - Analiza slučajeva upotrebe

Počinjemo od slučajeva upotrebe. Program se upotrebljava na sasvim jednostavan način i ovaj deo analize je praktično trivijalan. Funkcionalnost programa se može predstaviti sasvim jednostavnim dijagramom:



Slika 5: Osnovni slučaj upotrebe enciklopedije

Naš jedini slučaj upotrebe programa možemo detaljnije opisati:

Naziv:
Pregledanje podatka.

Opis:
Korisnik pregleda podatak sa datim rednim brojem.

Preduslov:
Podatak postoji u bazi podataka.

Pauslov:
Sadržaj baze podataka ostaje neizmenjen.

Tok akcija:

1. Korisnik pokreće program iz komandne linije navodeći redni broj podatka i naziv datoteke u koju se zapisuje binarni sadržaj.
2. Proverava se ispravnost podataka koje je naveo korisnik.
3. Iz baze podataka se čita traženi podatak.
4. Ako čitanje nije uspelo prekida se rad programa.
5. Korisniku se prikazuju podaci.
6. Čitaju se informacije o podacima koji imaju veze sa traženim podatkom.
7. Prikazuju se informacije o povezanim podacima, uređene po abecednom poretku po naslovu.
8. Ako postoji, binarni sadržaj se upisuje u datoteku. Ukoliko nije navedena datoteka, korisniku se prikazuje odgovarajuće obaveštenje.

Prototip izveštaja

Da bi ponašanje programa bilo tačno opisano potrebno je da odredimo u kom obliku se tražene informacije prikazuju korisniku. To možemo učiniti pisanjem prototipova izveštaja.

Prototip izveštaja za tekstualne podatke:

```
> Enciklopedija 42

Lav (tekst 42)
--------------
Lav (Leo ili Panthera leo) je velika snažna mačka iz porodice
Felidae, druga po veličini od velikih mačaka (posle tigra).
...

Pera Perić

Povezani podaci:
 - Lav (slika 47)
 - Lav u lovu (film 48)
 - Tigar (tekst 73)
 - Urlik lava (zvuk 46)
```

Prototip izveštaja za slike (prikazan tekst predstavlja sadržaj napomene):

```
> Enciklopedija 47 slika.dat

Lav (slika 47)
--------------
Odrasli primerak mužjaka u prirodnom okruženju.

Dimenzije: 1024 x 768

Fotko Fotkić

Povezani podaci:
 - Lav (tekst 42)
 - Lav u lovu (film 48)
 - Urlik lava (zvuk 46)
```

Prototip izveštaja za zvučne zapise (prikazan tekst predstavlja sadržaj napomene):

```
> Enciklopedija 46 zvuk.dat

Urlik lava (zvuk 46)
--------------------
Upozoravajući urlik lava spremnog da brani svoju teritoriju.

Trajanje: 21s

Sima Snimić

Povezani podaci:
 - Lav (slika 47)
 - Lav (tekst 42)
 - Lav u lovu (film 48)
```

Prototip izveštaja za slučaj filmova (prikazan tekst predstavlja sadržaj napomene):

```
> Enciklopedija 48

Lav u lovu (film 48)
--------------------
```

```
Lavica lovi antilopu. Obratite pažnju na pomoć koju joj pružaju
ostali lavovi onemogućavajući antilopi bekstvo.

Dimenzije: 1024 x 768
Trajanje: 1m 12s

Žika Žikić

Povezani podaci:
 - Lav (slika 47)
 - Lav (tekst 42)
 - Urlik lava (zvuk 46)
```

Opis aktivnosti koje program izvodi i prototipovi mogućih izveštaja predstavljaju elemente *detaljne specifikacije* programa. Ona bi trebalo da obuhvata i opis mogućih problema i specifikacije načina ponašanja programa u slučaju njihovog pojavljivanja.

Pri upotrebi programa Enciklopedija može doći do sledećih problema:

- pokretanje programa bez argumenata – u tom slučaju ispisujemo poruku

  ```
  Upotreba:
      Enciklopedija <id> [<bindat>]
  ```

- zahtevanje podatka koji ne postoji u bazi podataka – tada je umesto izveštaja potrebno ispisati poruku poput:

  ```
  Ne postoji podatak sa rednim brojem 23!
  ```

- izostavljanje naziva datoteke u slučaju nekog multimedijalnog podatka – tada je na kraju izveštaja potrebno ispisati poruku:

  ```
  Binarni sadržaj nije zapisan
  jer nije naveden naziv datoteke!
  ```

- nije moguće pisanje u navedenu datoteku – tada je na kraju izveštaja potrebno ispisati poruku poput:

  ```
  Nije uspelo pisanje u datoteku "p:\dat"!
  ```

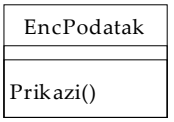
Korak 2 - Analiza i projektovanje prikazivanja podataka

Osnovna ideja – klasa EncPodatak

Složenost predstavljenog problema potiče, pre svega, iz različite prirode podataka koji se čitaju iz enciklopedije. Sam program ima prilično jednostavnu strukturu, koju bismo, u grubim crtama i bez ispitivanja da li je došlo do grešaka, mogli opisati ovako:

```
int main( int argc, char** argv )
{
    int id = atoi( argv[1] );
    string bindat;
    if( argc > 2 )
        bindat = argv[2];
    EncPodatak* p = ProcitajIzBazePodataka( id );
    p->Prikazi( cout, bindat );
    delete p;
}
```

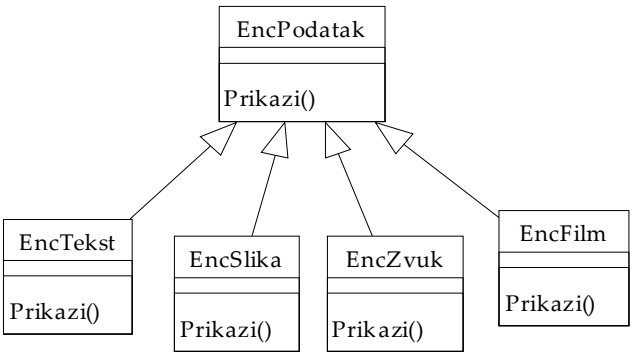
Nakon što smo preciznije definisali kakvo ponašanje želimo od našeg programa, potrebno je da prepoznamo koje su nam klase potrebne i kakvo ponašanje su one dužne da obezbede. Poći ćemo od jednostavne klase `EncPodatak` koja mora biti u stanju da napravi izveštaj u traženom formatu:



Slika 6: Klasa `EncPodatak`

Spuštanje ponašanja niz hijerarhiju

Kako se prikazivanje podatka razlikuje za različite tipove podataka, možemo birati da li ćemo pri ispisivanju proveravati tip podatka i prilagođavati mu način ispisivanja, ili ćemo čitav postupak ispisivanja implementirati različito za različite tipove podataka. Jedna od osnovnih ideja vodilja pri objektno orijentisanom razvoju je da bi trebalo izbegavati eksplicitne analize tipova i izbore ponašanja – bolje je praviti hijerarhije klasa i upotrebljavati ih tako da se provere tipova i izbori ponašanja odvijaju implicitno, primenom dinamičkog vezivanja metoda. U skladu sa time napravićemo više klasa podataka i preneti odgovarajuće odgovornosti na njih. Prenošenje dela ponašanja iz bazne klase na jednu ili više izvedenih klasa naziva se *spuštanje ponašanja niz hijerarhiju*. Slika 7 predstavlja dobijenu hijerarhiju klasa.



Slika 7: Hijerarhija klasa enciklopedijskih podataka

Podizanje ponašanja uz hijerarhiju

Nije dobro sve elemente prikazivanja podatka implementirati iznova za svaku klasu. Jedna mana takvog pristupa je što zahteva više programiranja nego što je neophodno. Daleko značajniji problem, predstavlja činjenica da bi u tako pisanom kodu svaka izmena u ponovljenim delovima koda morala da bude implementirana u svim klasama. Da bismo obezbedili da se prikazivanje podataka razlikuje za različite tipove podataka, oblikovali smo nove klase i odlučili da na svakoj od njih implementiramo prikazivanje na odgovarajući način.

Sada nam je potreban upravo obrnut postupak, jer želimo da zajedničke delove ponašanja vratimo u baznu klasu. To postižemo deljenjem složenih postupaka na manje celine, raspoređujući zajedničke delove u baznoj klasi, a delove koji se razlikuju u izvedene klase. Da bismo mogli prepoznati zajedničke delove postupka prikazivanja podataka razmotrimo malo detaljnije primere opisane pri definisanju načina upotrebe programa. Potrebno je prikazati, redom:

- naslov podatka i redni broj – ispisivanje se odvija na isti način za sve tipove podataka;
- tekst ili napomenu – u slučaju teksta ispisuje se tekst podatka, a u slučaju ostalih tipova podataka ispisuje se napomena;
- specifične podatke poput dimenzija i trajanja – ispisivanje podataka se razlikuje za sve tipove podataka;
- ime i prezime autora – ispisivanje se odvija na isti način za sve tipove podataka;
- naslove i redne brojeve povezanih podataka – isto za sve tipove podataka;
- binarni sadržaj je potrebno zapisati u datoteci – ovo je isto za sve tipove podataka osim za tekst.

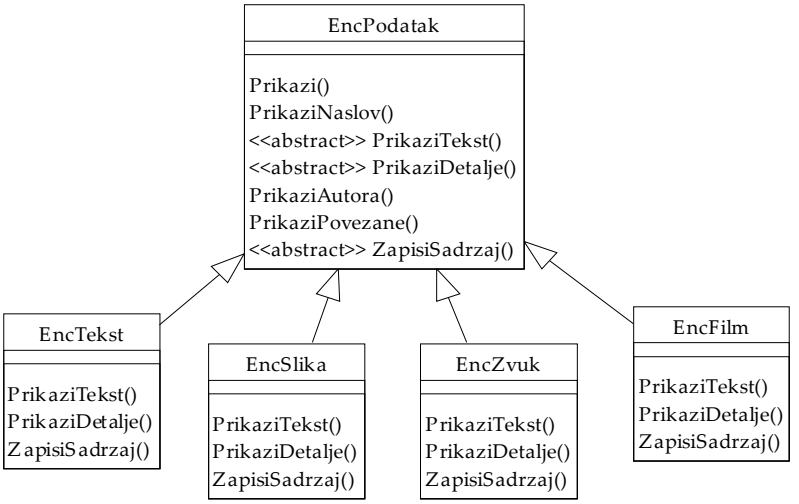
Implementacija metoda `EncPodatak::Prikazi` bi mogla da bude:

```
void EncPodatak::Prikazi( ostream& ostr, string bindat ) const
{
    PrikaziNaslov( ostr );
    PrikaziTekst( ostr );
    PrikaziDetalje( ostr );
    PrikaziAutora( ostr );
    PrikaziPovezane( ostr );
    ZapisiSadrzaj( bindat );
}
```

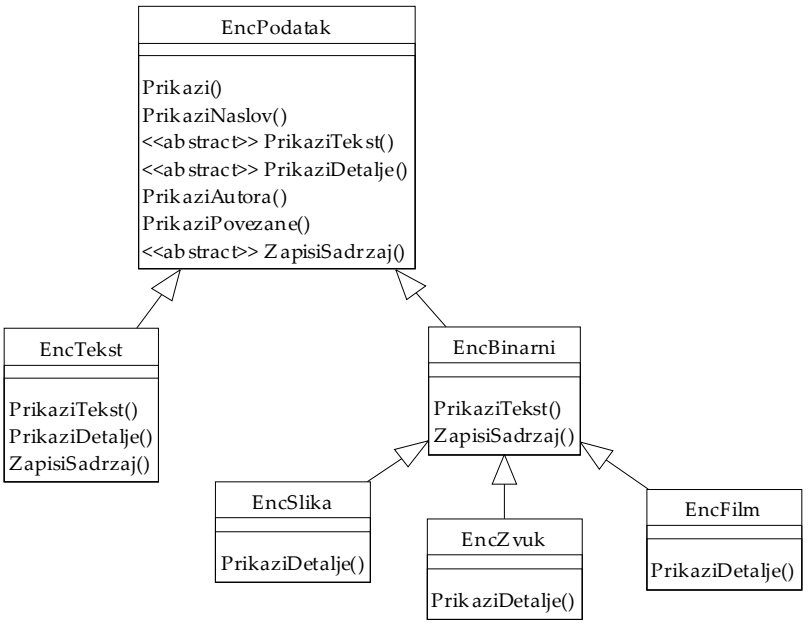
Možemo zaključiti da se metodi `Prikazi`, `PrikaziNaslov`, `PrikaziAutora` i `PrikaziPovezane` mogu implementirati u klasi `EncPodatak`, jer se odgovarajući elementi izveštaja uvek prave na isti način. Metodi `PrikaziTekst`, `PrikaziDetalje` i `ZapisiSadrzaj` se implementiraju različito za različite klase. Premeštanje dela ponašanja iz klasa naslednica u njihove bazne klase naziva se *podizanje zajedničkog ponašanja uz hijerarhiju*. Slika 8 predstavlja dijagram hijerarhije klasa nakon primene podizanja zajedničkog ponašanja uz hijerarhiju.

Dodavanje klasa usred hijerarhije

Na osnovu predstavljenih prototipova izveštaja uočavamo da se prikazivanje teksta za sve tipove podataka, osim za tekst, odvija na isti način. Takođe, i zapisivanje sadržaja u datoteku se odvija na sličan način – jedina razlika je u nazivu binarnog sadržaja u katalogu `Podatak`. Čini se da bi i za ove metode bilo dobro izvesti podizanje zajedničkog ponašanja uz hijerarhiju, ali ne smemo ih podići u klasu `EncPodatak` jer ona ima klasu naslednicu `EncTekst` koja se ponaša drugačije.



Slika 8: Razvijena hijerarhija klasa enciklopedijskih podataka



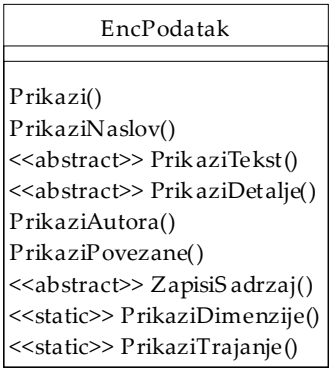
Slika 9: Hijerarhija klasa enciklopedijskih podataka, konačan oblik

U ovakvim slučajevima obično je korisno da uvedemo novu međuklasu koja istovremeno specijalizuje baznu klasu i apstrahuje jedan broj izvedenih klasa. Naša nova klasa bi trebalo da obuhvati najveći mogući presek ponašanja klasa `EncSlika`, `EncZvuk` i `EncFilm`. Njeno ponašanje se razlikuje od ponašanja klase `EncTekst`. Novu klasu ćemo nazvati `EncBinarni`.

Dodavanje nove klase usred hijerarhije se naziva *umetanje klase u hijerarhiju*. Umetanje nove klase je obično praćeno spuštanjem određenog ponašanja iz bazne klase na novu klasu ili podizanjem ponašanja u novu klasu iz klasa kojima se ona uvodi kao zajednička osnova. U našem slučaju primenjujemo podizanje ponašanja, jer metode `PrikaziTekst` i `ZapisiSadrzaj` podižemo iz klasa konkretnih binarnih podataka u novu klasu. Slika 9 predstavlja izmenjenu hijerarhiju klasa.

Pomoćni metodi

Da bi prikazivanje detaljnih informacija o podacima bilo ujednačeno, potrebno je prikazivanje svih vrsta informacija koje se mogu pojaviti za više vrsta objekata implementirati u klasi `EncPodatak`. Kako se nazivi podataka mogu razlikovati, moguće rešenje je definisanje statičkih metoda koji u željenom formatu ispisuju potrebne informacije (Slika 10).



Slika 10: Detalji klase `EncPodatak`

Razdvajanje ponašanja

Primetimo da se ime jednog od metoda razlikuje od ostalih metoda potrebnih za prikazivanje podataka: `ZapisiSadrzaj`. Da li je to slučajno? Ne. Dok se svi ostali metodi bave prikazivanjem podataka na standardnom izlazu, ovaj metod zapisuje binarni sadržaj podatka u datoteku. Zapravo, deluje neprirodno da se ta aktivnost odvija u okviru metoda `Prikazi`. Bilo bi bolje da se prikazivanje podatka na standardnom izlazu odvoji od zapisivanja njegovog binarnog sadržaja u datoteku. Štaviše, za korisnika klase može biti prilično opterećujuće, kako pri upoznavanju tako i pri upotrebi, ako metodi ne rade tačno ono što njihov naziv sugeriše. *Nije dobro da metodi rade ni manje ni više od onoga što bi se na osnovu imena i arumenata moglo očekivati.* Kako zapisivanje binarnog sadržaja u datoteci nema po svojoj prirodi, a ni po nazivu, ništa zajedničko sa prikazivanjem podataka na standardnom izlazu, dobro je ova dva postupka potpuno razdvojiti.

Dovoljno je da metod `Prikazi` obavi sve ostale predviđene aktivnosti:

```
void EncPodatak::Prikazi( ostream& ostr, string bindat ) const
{
    PrikaziNaslov( ostr );
```

```
        PrikaziTekst( ostr );
        PrikaziDetalje( ostr );
        PrikaziAutora( ostr );
        PrikaziPovezane( ostr );
    }
```

Metodu `ZapisiSadrzaj` ćemo proširiti ranije pretpostavljeno ponašanje tako da vrati logičku vrednost `true` ako je sve proteklo u redu, a `false` u slučaju problema. Problemima ćemo smatrati sve obilike grešaka pri zapisivanju binarnog sadržaja u datoteku, kao i slučaj kada nije navedeno ime datoteke za zapisivanje binarnog sadržaja a on postoji.

Veoma je važno voditi računa o preciznom imenovanju metoda. Ime metoda mora ukazivati na operaciju koju metod izvodi. Ako metod radi bilo manje bilo više posla nego što njegovo ime ukazuje, možemo biti potpuno sigurni da će dolaziti do grešaka pri upotrebi metoda. Mnogi autori do te mere insistiraju na preciznom imenovanju metoda da smatraju da pisanje komentara u metodima uopšte nije potrebno:

- ukoliko na osnovu imena nije očigledno šta metod radi, to je zbog toga što metod obavlja više logički nezavisnih celina – tada je potrebno metod podeliti na više metoda koji obavljaju po jednu celinu i sa kojima nema takvih problema;
- ukoliko je implementacija metoda složena do te mere da je neophodno komentarisanje delova koda, to je zato što se suviše složeno ponašanje pokušava implementirati jednim metodom – tada je potrebno svaku celinu iz implementacije metoda, za koju je potreban komentar, izdvojiti u poseban metod čije bi ime dovoljno dobro ilustrovalo obuhvaćeni deo algoritma, pa u polaznom metodu umesto više složenih segmenata koda staviti pozive novih metoda.

Primetimo da ekstremno pridržavanje ovakvog pristupa ima za posledicu veoma veliki broj jednostavnih metoda i klasa, što opet predstavlja problem (mada potpuno drugačiji) pri upoznavanju i upotrebi klase. Pri oblikovanju hijerarhije klasa i svake konkretne klase potrebno je pronaći dobru meru, kako bi skup klasa i metoda bio i pregledan i jasan.

Vidljivost metoda

Što se tiče prikazivanja podataka, oblikovanje naših klasa je privedeno kraju. Primetimo da, od svih navedenih metoda, samo metodi `Prikazi` i `ZapisiSadrzaj` moraju biti javni, dok svi ostali metodi mogu biti zaštićeni.

Korak 3 - Analiza i projektovanje konstruktora

Da bismo mogli prikazivati enciklopedijske podatke, neophodno je da budemo u stanju da na osnovu podataka pročitanih iz baze podataka najpre napravimo odgovarajuće objekte. Kako podatke dobijamo iz baze podataka u formi kataloga `Podatak`, logično je da obezbedimo takve konstruktore koji kao argument imaju upravo `Podatak`:

```
EncPodatak::EncPodatak( const Podatak& p )
EncTekst::EncTekst( const Podatak& p )
EncBinarni::EncBinarni( const Podatak& p )
EncSlika::EncSlika( const Podatak& p )
EncZvuk::EncZvuk( const Podatak& p )
```

```
EncFilm::EncFilm( const Podatak& p )
```

Pri tome će se u svakoj od klasa najpre upotrebiti odgovarajući konstruktor bazne klase, a zatim iz podatka izdvojiti preostale informacije.

Neke informacije su tekstualne (`naslov`, `tekst`, `autor`, `napomene`), neke su celobrojne (`id`, `širina`, `visina`), neke realne (`trajanje`) a neke binarne (`slika`, `zvuk`, `film`). Za tekstualne i binarne podatke se mogu upotrebljavati upravo podaci tipa `string` koji su sadržani u objektu klase `Podatak`. U slučaju celobrojnih i realnih podataka potrebno je budemo u stanju da pročitamo brojeve iz njihove tekstualne reprezentacije. Zbog toga uvodimo nove statičke metode u klasu `EncPodatak`:

```
static int ProcitajCeoBroj( string s );
static double ProcitajRealanBroj( string s );
```

Dinamički konstruktor

Osnovni problem pri pravljenju objekata predstavlja činjenica da pročitani `Podatak` može biti tekst, slika, zvuk ili film, pri čemu mi to ne znamo unapred. Kako onda napraviti objekat odgovarajuće klase? Prirodno rešenje bi bilo da pročitamo podatke iz baze podataka i zatim analiziramo tip dobijenih podataka i pravimo odgovarajuće objekte:

```
...
Podatak podatak;
string tip;
EncPodatak* obj = 0;
if( ProcitajPodatak( id, podatak, tip )){
    if( tip == "tekst" )
        obj = new EncTekst( podatak );
    else if( tip == "slika" )
        obj = new EncSlika( podatak );
    else if( tip == "zvuk" )
        obj = new EncZvuk( podatak );
    else if( tip == "film" )
        obj = new EncFilm( podatak );
    }
...
```

Međutim, pre nekoliko trenutaka smo ustanovili da je u slučaju različitog ponašanja za različite tipove podataka potrebno izbegavati eksplicitnu analizu i izbore, već je bolje da se prave i upotrebljavaju hijerarhije klasa, pa da se provere tipova i izbori ponašanja odvijaju implicitno, primenom dinamičkog vezivanja metoda. Da li možemo na ovom mestu upotrebiti dinamičko vezivanje metoda?

Da bismo mogli primeniti dinamičko vezivanje metoda neophodno je da imamo na raspolaganju objekat koji pripada nekoj od klasa iz hijerarhije. Ali mi ovde tek pravimo takav objekat i još uvek smo daleko od njegove upotrebe. Na žalost programera, nešto poput „dinamičkog konstruktora“ jednostavno ne postoji. Zapravo, uz makar malo poverenja u autore programskog jezika C++, možemo pomisliti da bi tako nešto sigurno postojalo ako bi uopšte moglo da postoji. I bili bismo u pravu. Problem je u tome što ma kako dobro da mi u konkretnoj situaciji znamo na osnovu kog parametra i na koji način želimo da napravimo i

inicijalizujemo objekat tačno odgovarajuće klase, ne postoji neformalan način da se to saopšti prevodiocu već to moramo učiniti sasvim formalno – upravo pisanjem dela programa koji pravi novi objekat.

Deo programa koji smo napisali je sasvim u redu i u konkretnoj situaciji ga ne možemo značajno popraviti. Ipak, ima prostora za manje izmene. Ako posmatramo hijerarhiju enciklopedijskih podataka kao jednu celinu, a glavnu funkciju programa kao drugu, nije teško uočiti da autor funkcije `main` mora znati koji sve tipovi podataka postoje i kako se označavaju. To nije dobro jer predstavlja narušavanje enkapsulacije čitave hijerarhije. Daleko je bolje da se od korisnika hijerarhije sakrije informacija o tome koliko ima klasa i kako se prepoznaju. Sredstvo da to uradimo je prebacivanje dela programa koji se koristi za pravljenje novih objekata u samu hijerarhiju. To se može učiniti pisanjem odgovarajuće funkcije ili pisanjem statičkog metoda klase `EncPodatak`. Sasvim je jasno da se ta funkcija (ili metod) mora pisati, ili bar održavati, nakon svakog proširivanja hijerarhije. U ovom slučaju odlučujemo se za pisanje statičkog metoda:

```
EncPodatak* NapraviPodatak( string tip, const Podatak& p )
{
    EncPodatak* obj = 0;
    if( ProcitajPodatak( id, podatak, tip )){
        if( tip == "tekst" )
            obj = new EncTekst( podatak );
        else if( tip == "slika" )
            obj = new EncSlika( podatak );
        else if( tip == "zvuk" )
            obj = new EncZvuk( podatak );
        else if( tip == "film" )
            obj = new EncFilm( podatak );
        }
    return obj;
}
```

Metod `NapraviPodatak` vraća prazan pokazivač ako se radi o podatku čiji tip nije podržan.

Korak 4 - Ostali metodi

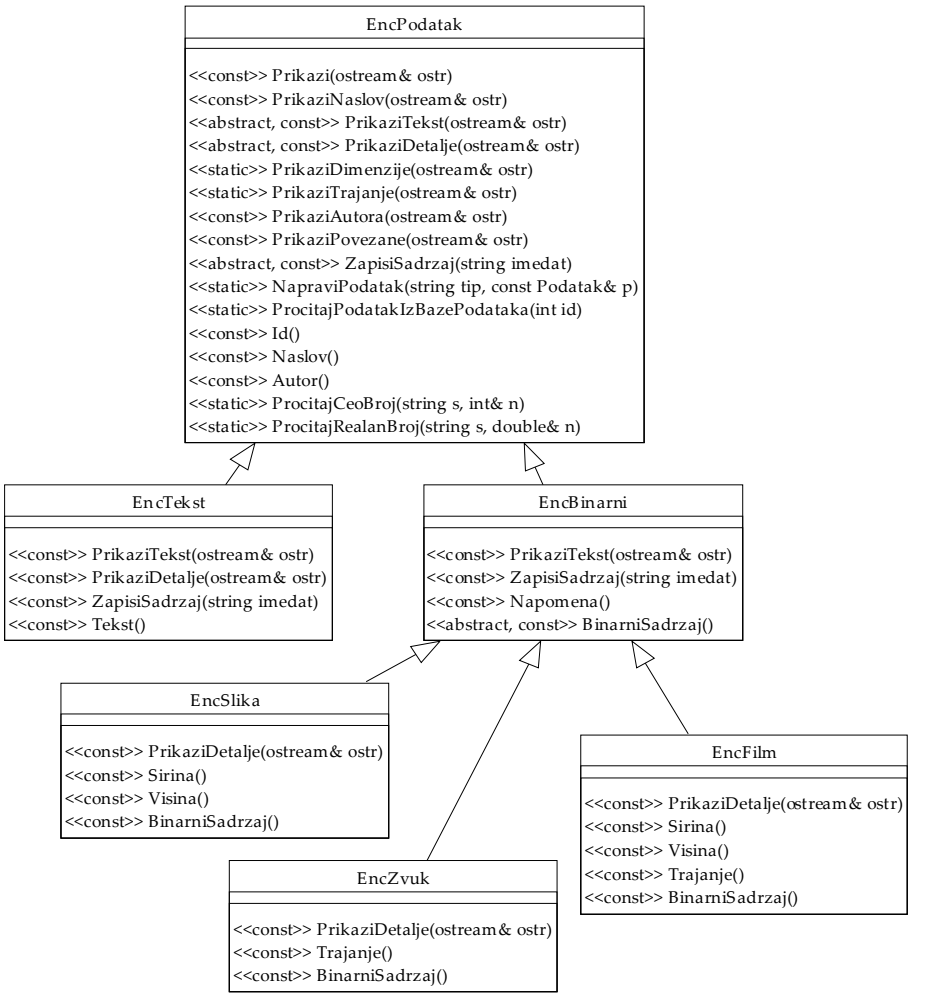
Prethodno opisanim statičkim metodom obezbedili smo pravljenje novog objekta enciklopedijskog podatka na osnovu već pročitanog podatka. Radi jednostavnije upotrebe možemo dodati još jedan statički metod za pravljenje novih objekata, koji za argument ima samo redni broj podatka, dok će podatak sam pročitati iz baze podataka:

```
EncPodatak* ProcitajPodatakIzBazePodataka( int id )
```

Da bi se jedan enciklopedijski podatak mogao predstaviti korisniku na zahtevani način, potrebno je da budu na raspolaganju odgovarajuće informacije. Zbog toga u hijerarhiju klasa enciklopedijskih podataka dodajemo odgovarajuće pristupne metode, i to samo za čitanje:

- u klasu `EncPodatak` dodajemo metode za čitanje rednog broja, naslova i imena autora;

- u klasu `EncTekst` dodajemo metod za čitanje teksta;
- u klase `EncSlika` i `EndFilm` dodajemo metode za čitanje širine i visine;



Slika 11: Detalji hijerarhije enciklopedijskih podataka

- u klase `EncZvuk` i `EndFilm` dodajemo metod za čitanje trajanja;
- u klasu `EncBinarni` dodajemo metod za čitanje napomene.

Pored toga, u klasu `EncBinarni` dodajemo i apstraktni metod `BinarniSadrzaj` koji u konkretnim klasama izračunava sadržaj slike, zvuka ili filma.

Svi ovi podaci mogu da budu zaštićeni, jer u ovom trenutku ne postoji potreba da budu javni. Slika 11 sadrži dijagram hijerarhije klasa, koji obuhvata sve potrebne metode.

Korak 5 - Implementacija pomoćne „baze podataka“

Da bismo mogli isprobavati kod koji pišemo nije dovoljno da imamo deklaraciju biblioteke za upotrebu baze podataka, onako kako je navedena u zaglavlju `BazaPodataka.h`, već je neophodno i da imamo primer implementacije baze podataka. Zato ćemo napisati pomoćnu biblioteku, koja bi trebalo da nam bude dovoljna tokom rešavanja zadatka. Neophodno je da obezbedimo implementaciju deklarisanih metoda, kao i da simuliramo čitanje primera podataka iz baze podataka.

Implementiranje klase `BazaPodataka` nije deo rešenja zadatka, već je pomoćno sredstvo koje nam omogućava da naš program isprobamo. Zbog toga ćemo implementaciju izvesti uz što manje napora, kako bismo pažnju što pre mogli posvetiti pisanju programa *Enciklopedija*.

Pored zahtevanih javnih metoda obezbedićemo i privatni metod

```
static bool ProcitajSve(
    int id, Podatak& x, string& tip, vector<int>& pov
    )
```

koji čita podatak, tip i niz rednih brojeva povezanih podataka. Javne metode ćemo implementirati pomoću metoda `ProcitajSve`. Metod `ProcitajSve` ćemo implementirati tako da pri prvoj upotrebi inicijalizuje statičke podatke koji će simulirati sadržaj baze podataka.

```
#include <map>
#include <vector>
using namespace std;

typedef map<string,string> Podatak;

class BazaPodataka
{
public:
    // Čitanje jednog podatka iz baze podataka
    static bool ProcitajPodatak( int id, Podatak& x, string& tip )
        {
        vector<int> povezani;
        return ProcitajSve( id, x, tip, povezani );
        }
    // Čitanje niza rednih brojeva povezanih podataka
    static void ProcitajPovezane( int id, vector<int>& povezani )
        {
        Podatak x;
        string tip;
        ProcitajSve( id, x, tip, povezani );
        }

private:
    // Čitanje svih informacija o podatku iz baze podataka
    static bool ProcitajSve(
            int id, Podatak& x, string& tip, vector<int>& pov
            )
        {
```

```
// Statički podaci kojima simuliramo sadržaj baze podataka
static map<int,Podatak> podaci;
static map<int,vector<int> > povezani;
static map<int,string> tipovi;
// Podatke inicijalizujemo samo prvi put
if( podaci.empty() ){
  { Podatak p;
    p["id"] = "42";
    p["naslov"] = "Lav";
    p["tekst"] =
        "Lav (Leo ili Panthera leo) je velika snažna "
        "mačka iz porodice Felidae, druga po veličini od "
        "velikih mačaka (posle tigra). ...";
    p["autor"] = "Pera Perić";
    podaci[42] = p;
    povezani[42].push_back( 46 );
    povezani[42].push_back( 47 );
    povezani[42].push_back( 48 );
    povezani[42].push_back( 73 );
    tipovi[42] = "tekst";
  }{
    Podatak p;
    p["id"] = "47";
    p["naslov"] = "Lav";
    p["slika"] = "...binarni zapis slike...";
    p["sirina"] = "1024";
    p["visina"] = "768";
    p["napomena"] =
        "Odrasli primerak mužjaka u prirodnom okruženju.";
    p["autor"] = "Fotko Fotkić";
    podaci[47] = p;
    povezani[47].push_back( 42 );
    povezani[47].push_back( 46 );
    povezani[47].push_back( 48 );
    tipovi[47] = "slika";
  }{
    Podatak p;
    p["id"] = "46";
    p["naslov"] = "Urlik lava";
    p["zvuk"] = "...binarni zapis zvuka...";
    p["trajanje"] = "3764,45";
    p["napomena"] =
        "Upozoravajući urlik lava spremnog da brani "
        "svoju teritoriju.";
    p["autor"] = "Sima Snimić";
    podaci[46] = p;
    povezani[46].push_back( 42 );
    povezani[46].push_back( 47 );
    povezani[46].push_back( 48 );
    tipovi[46] = "zvuk";
  }{
    Podatak p;
    p["id"] = "48";
    p["naslov"] = "Lav u lovu";
    p["film"] = "...binarni zapis filma...";
    p["sirina"] = "640";
```

```
                p["visina"] = "400";
                p["trajanje"] = "124,75";
                p["napomena"] =
                    "Lavica lovi antilopu. Obratite pažnju na pomoć "
                    "koju joj pružaju ostali lavovi onemogućavajući "
                    "antilopi da pobegne.";
                p["autor"] = "Žika Žikić";
                podaci[48] = p;
                povezani[48].push_back( 42 );
                povezani[48].push_back( 46 );
                povezani[48].push_back( 47 );
                tipovi[48] = "film";
              }{
                Podatak p;
                p["id"] = "73";
                p["naslov"] = "Tigar";
                p["tekst"] = "...neki tekst o tigru...";
                p["autor"] = "Žika Žikić";
                podaci[73] = p;
                povezani[73].push_back( 42 );
                tipovi[73] = "tekst";
               }
              }
            // Izračunavamo tražene podatke
            x = podaci[id];
            tip = tipovi[id];
            pov = povezani[id];
            return !tip.empty();
            }
    };
```

Korak 6 - Struktura klasa

Kao rezultat analize problema i projektovanja dobili smo dijagram klasa i skicu implementacije nekih važnih metoda. Većina poslova koje nam je preostalo da uradimo tokom implementacije su prilično jednostavni. Jedina važnija odluka koju još nismo doneli odnosi se na internu strukturu klasa naše hijerarhije. I na ovom primeru vidimo da se ponašanje čitave hijerarhije klasa može u potpunosti opisati bez ikakvog zalaženja u određivanje strukture. Sada, kada znamo šta će i kako naše klase raditi, ipak moramo da vidimo i kako će izgledati.

Obično se faze razvoja koje prethode određivanju interne strukture klasa nazivaju *projektovanjem*, a preostale faze, počev od određivanja interne strukture, se nazivaju *implementiranjem*.

Članovi podaci

Imamo na raspolaganju dva osnovna pristupa definisanju strukture klasa hijerarhije enciklopedijskih podataka. Jedan je da se za svaku konkretnu informaciju o enciklopedijskom podatku definiše po član podatak u klasi u kojoj postoji odgovarajući pristupni metod. Tako bi klasa `EncPodatak` imala naslov i ime autora, klasa `EncTekst` bi imala tekst, klasa `EncBinarni` bi imala napomenu, a ostale klase bi imale njima specifične informacije. U tom slučaju

konstruktori bi bili odgovorni da inicijalizuju te članove podatke, a pristupni metodi bi jednostavno čitali vrednosti članova podataka.

Drugi pristup problemu interne strukture je da se u okviru klase `EncPodatak` definiše član podatak klase `Podatak`. U tom slučaju klase naslednice ne bi imale nikakve nove članove podatke. Konstruktori izvedenih klasa bi samo prenosili odgovornost na konstrukor bazne klase, a on bi iskopirao čitav katalog tipa `Podatak`. Pristupni metodi bi na osnovu sadržaja kataloga klase `Podatak` izračunavali odgovarajuće vrednosti.

Nijedan od dva pristupa nema neku značajnu prednost u odnosu na drugi. Na primer, u prvom slučaju se *svi podaci* tačno *jednom* čitaju iz kataloga i po potrebi prevode u brojeve, dok se u drugom slučaju čitaju *samo oni podaci koji se upotrebljavaju* u programu, ali se to može ponoviti *više puta*. Prvi pristup obezbeđuje preglednu i jasnu strukturu, ali se ona mora dopunjavati sa svakom novom klasom, dok drugi pristup prilično sakriva strukturu (što je upravo nešto dublja enkapsulacija) i oslobađa nas obaveze da dodajemo članove u nove klase. U ovom trenutku izbor pada na prvi pristup oblikovanju strukture klasa. Drugi pristup ostavljamo za vežbu.

Destruktori

Kada koristimo hijerarhije klasa kao u primeru funkcije `main`, postavlja se pitanje kako se ponaša izraz `delete`? Dovoljno je da posmatramo primer poput:

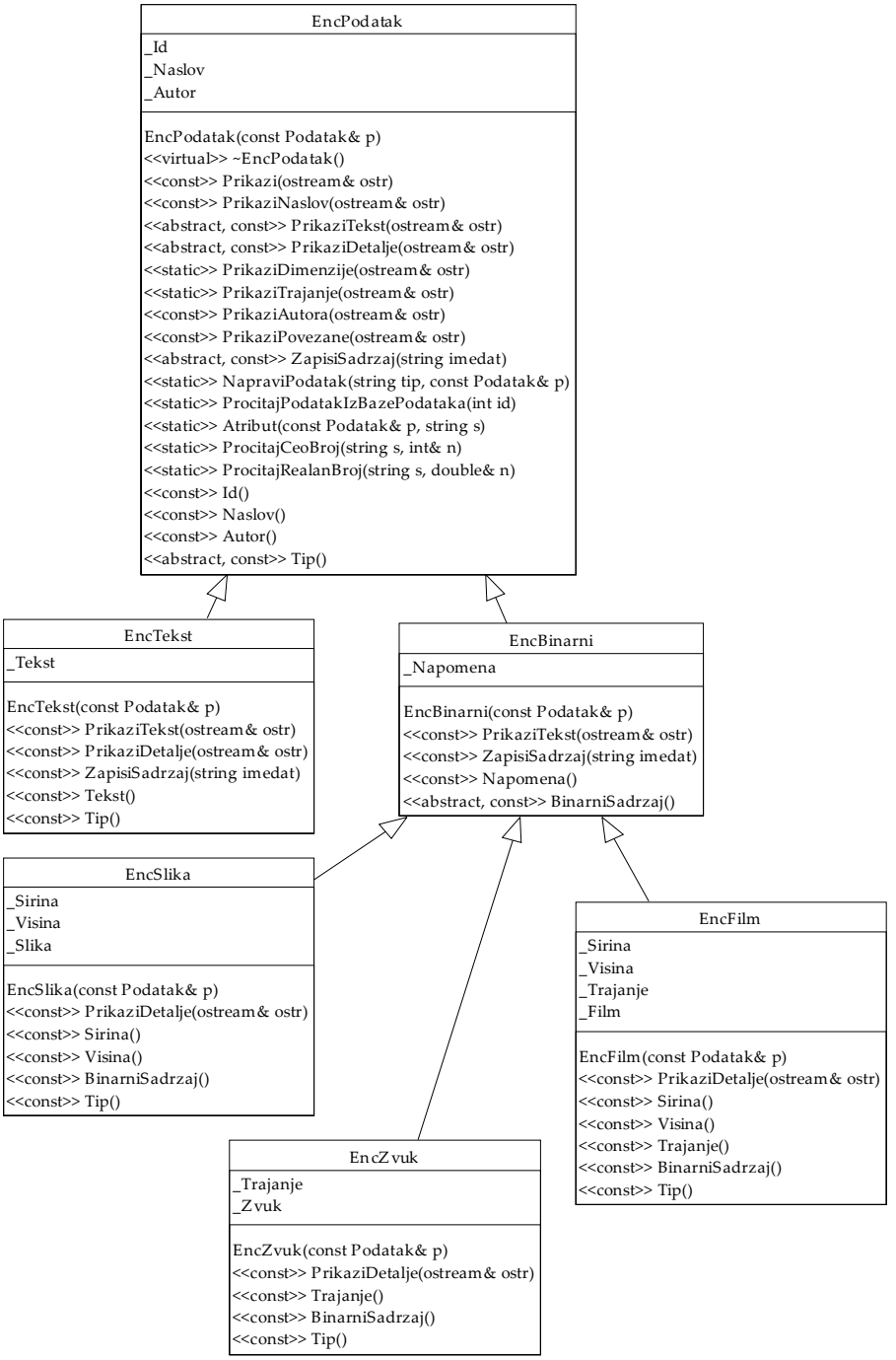
```
delete EncPodatak::ProcitajPodatakIzBazePodataka( id );
```

Metod `EncPodatak::ProcitajPodatakIzBazePodataka` izračunava pokazivač na neki enciklopedijski podatak. Rezultat je tipa `EncPodatak*`. Izraz `delete` se može bezbedno primeniti i ako je rezultat 0. Kao što već znamo, izraz `delete` najpre poziva destruktor, pa zatim oslobađa memoriju koju je objekat zauzimao. To će se desiti i u navedenom primeru. Prevodilac će prepoznati da izraz `delete` ima operand tipa `EncPodatak*`. Na osnovu toga će doći do pozivanja destruktora klase `EncPodatak` i do oslobađanja memorije koju je objekat zauzimao. To bi trebalo da je sasvim ispravno?

Ne! Doći do veoma ozbiljnih problema, jer pozivanjem destruktora bazne klase najčešće ne može da se ispravno deinicijalizuje objekat izvedene klase[11].

Neka je, na primer, pročitan podatak klase `EncSlika`, koji ima član podatak `_Slika` tipa `string` koji zadrži binarni zapis slike. Kako klasa `string` ima ispravno definisan destruktor, podrazumevani destruktor klase `EncSlika` će sasvim ispravno deinicijalizovati podatak `_Slika` i osloboditi odgovarajuću memoriju. Na osnovu toga možemo zaključiti da nema potrebe eksplicitno pisati destruktor klase `EncSlika`. Takav zaključak je potpuno ispravan. Do istog zaključka možemo doći u odnosu na bilo koju klasu hijerarhije enciklopedijskih podataka. Ni u jednoj od klasa nije potrebno pisati destruktor jer podrazumevani destruktori sasvim ispravno rade svoj posao.

[11] U ovom odeljku se ponavlja deo diskusije o pisanju destruktora u hijerarhiji klasa. Tema se namerno ponavlja, zbog izuzetnog značaja i čestog previđanja od strane početnika. Detaljnija diskusija je u odeljku *Dinamičko vezivanje destruktora*, na strani 134.



Slika 12: Implementacija hijerarhije enciklopedijskih podataka

Jedini, ali nimalo bezazlen problem je u činjenici da je taj ispravan podrazumevani destruktor klase `EncSlika` potrebno i pozvati. A u prethodnom primeru se to neće desiti jer se odluka o pozivanju destruktora donosi u fazi prevođenja programa. Umesto destruktora klase `EncSlika` upotrebiće se destruktor klase `EncPodatak` i neće se deinicijalizovati neki podaci.

Jedino rešenje ovog problema koje je u skladu sa principima programiranja na programskom jeziku C++ jeste da se destruktor vezuje dinamički a ne statički. To se postiže stavljanjem ključne reči `virtual` ispred deklaracije destruktora bazne klase hijerarhije:

```
virtual ~EncPodatak();
```

Ostaje još da vidimo kako ćemo implementirati destruktor. Kao što smo već videli, u slučaju klase `EncPodatak` potrebno je da eksplicitno definišemo destruktor isključivo zbog toga da bismo označili da je neophodno da se njegovo vezivanje odvija dinamički. U skladu sa time, implementacija je trivijalna:

```
virtual ~EncPodatak()
    {}
```

Pomoćni metodi

Pri ispisivanju enciklopedijskih podataka potrebno je ispisivati i tip podataka. Zbog toga u baznu klasu uvodimo apstraktan metod `Tip` koji izračunava nisku sa nazivom tipa i koji implementiramo u svim konkretnim klasama.

Da bismo mogli jednostavno da čitamo atribute iz objekta tipa `Podatak`, definisaćemo i pomoćni statički metod `Atribut` u klasi `EncPodatak`.

Kada prethodnom dijagramu klasa dodamo članove podatke, konstruktore, destruktor bazne klase i upravo predstavljene metode, dobijamo potpuniju sliku hijerarhije (Slika 12).

Korak 7 - Implementacija hijerarhije klasa

Poći ćemo od bazne klase `EncPodatak`. Zbog toga što ona ima veći broj metoda, najpre ćemo deklarisati klasu a zatim implementirati metode. Za razliku od nje, ostale klase hijerarhije su sasvim jednostavne i sve metode ćemo implementirati već pri definisanju klasa.

Klasa EncPodatak

Klasa `EncPodatak` predstavlja osnovu hijerarhije. Zbog toga u njoj imamo nekoliko apstraktnih metoda. Pored toga, imamo i nekoliko statičkih metoda koji omogućavaju funkcionisanje čitave hijerarhije. Sada ćemo napisati definiciju klase `EncPodatak` i sve metode koji nisu apstraktni, osim metoda `NapraviPodatak`. Da bismo mogli napisati metod `NapraviPodatak` neophodno je da imamo na raspolaganju bar deklaracije svih klasa podataka čije objekte ovaj metod mora da pravi. Zato implementaciju tog metoda ostavljamo za kraj.

Definiciju klase `EncPodatak` pišemo u datoteci `EncPodatak.h`:

```
class EncPodatak
{
```

```
public:
    //  Konstruktor
    EncPodatak( const Podatak& p );

    //  Destruktor
    virtual ~EncPodatak()
        {}

    //  Prikazivanje i pisanje
    void Prikazi( ostream& ostr ) const;
    virtual bool ZapisiSadrzaj( string imedat ) const = 0;

    //  "Konstruktori"
    static EncPodatak* NapraviPodatak(
                           string tip, const Podatak &p
                           );
    static EncPodatak* ProcitajPodatakIzBazePodataka( int id );

protected:
    //  Pristupni metodi
    int Id() const
        { return _Id; }
    string Naslov() const
        { return _Naslov; }
    string Autor() const
        { return _Autor; }
    virtual string Tip() const = 0;

    //  Prikazivanje dela podatka
    void PrikaziNaslov( ostream& ostr ) const;
    void PrikaziAutora( ostream& ostr ) const;
    void PrikaziPovezane( ostream& ostr ) const;
    virtual void PrikaziTekst( ostream& ostr ) const = 0;
    virtual void PrikaziDetalje( ostream& ostr ) const = 0;

    //  Pomoćni metodi za prikazivanje
    static void PrikaziDimenzije( ostream& ostr, int s, int v );
    static void PrikaziTrajanje( ostream& ostr, double sec );

    //  Pomoćni metodi za čitanje podataka iz strukture Podatak
    static string Atribut( const Podatak& p, string s );
    static int ProcitajCeoBroj( string s )
        { return atoi( s.c_str() ); }
    static double ProcitajRealanBroj( string s )
        { return atof( s.c_str() ); }

private:
    //  Članovi podaci
    int     _Id;
    string  _Naslov;
    string  _Autor;
};
```

Implementacija destruktora i pristupnih metoda je trivijalna. Implementaciji ostalih metoda ćemo posvetiti malo više pažnje.

Čitanje celog i realnog broja iz niske je urađeno na način uobičajen za programski jezik C, jer je tako u konkretnom slučaju najjednostavnije i najefikasnije. Primetimo da standardna biblioteka programskog jezika raspolaže memorijskim tokovima, koji omogućavaju čitanje iz

niski i pisanje u niske kao da se radi o tokovima (videti odeljak *10.10.7 Memorijski tokovi*, na strani 394.). Kada se čitaju ili pišu neki složeniji podaci, to je prikladnije rešenje. U našem slučaju, mogli bismo metode za čitanje brojeva iz niski implementirati ovako:

```
...
    static int ProcitajCeoBroj( string s )
        {
        int n;
        istrstream sstr(s);
        sstr >> n;
        return n;
        }
...
```

Pri implementaciji konstruktora upotrebljavamo pomoćni metod `Atribut` za pristupanje atributima pročitanog podatka i metod `ProcitajCeoBroj` za prevođenje niske u ceo broj. Sve članove podatke klase `EncPodatak` inicijalizujemo navođenjem njihovih vrednosti u listi inicijalizacija:

```
EncPodatak::EncPodatak( const Podatak& p )
    : _Id( ProcitajCeoBroj(Atribut(p,"id")) ),
      _Naslov( Atribut(p,"naslov") ),
      _Autor( Atribut(p,"autor") )
{}
```

Prikazivanje definišemo tako da bude u skladu sa navedenim prototipovima izveštaja:

```
void EncPodatak::Prikazi( ostream& ostr ) const
{
    PrikaziNaslov( ostr );
    PrikaziTekst( ostr );
    PrikaziDetalje( ostr );
    ostr << endl;
    PrikaziAutora( ostr );
    ostr << endl;
    PrikaziPovezane( ostr );
    ostr << endl;
}

void EncPodatak::PrikaziNaslov( ostream& ostr ) const
{
    ostr << Naslov()
         << " (" << Tip() << " " << Id() << ")" << endl;
    ostr << "----------------------------------------" << endl;
}

void EncPodatak::PrikaziAutora( ostream& ostr ) const
{
    ostr << Autor() << endl;
}

void EncPodatak::PrikaziDimenzije( ostream& ostr, int s, int v )
{
    ostr << "Dimenzije: " << s << " x " << v << endl;
}
```

```
void EncPodatak::PrikaziTrajanje( ostream& ostr, double sec )
{
    ostr << "Trajanje: ";
    if( sec >= 60 ){
        int m = sec / 60;
        ostr << m << "m ";
        sec -= m * 60;
        }
    ostr << sec << "s" << endl;
}
```

Prikazivanje povezanih podataka je nešto složenije. Najpre čitamo redne brojeve povezanih podataka. Zatim redom čitamo sve povezane podatke i pravimo njihove opise od naslova i naziva tipa podataka. Tako oblikovane opise i redne brojeve pamtimo u katalogu. Na taj način implicitno uređujemo podatke u abecednom poretku:

```
void EncPodatak::PrikaziPovezane( ostream& ostr ) const
{
    vector<int> povezani;
    BazaPodataka::ProcitajPovezane( Id(), povezani );
    if( povezani.empty() )
        ostr << "Nema povezanih podataka." << endl;
    else{
        ostr << "Povezani podaci:" << endl;
        map<string,int> naslovi;
        for( unsigned i=0; i<povezani.size(); i++ ){
            EncPodatak* p
                = ProcitajPodatakIzBazePodataka( povezani[i] );
            if( p ){
                naslovi[ p->Naslov() + " (" + p->Tip() ]
                    = p->Id();
                delete p;
                }
            }
        map<string,int>::iterator
            i = naslovi.begin(),
            e = naslovi.end();
        for( ; i!=e; i++ )
            ostr << " - " << i->first << " "
                 << i->second << ")" << endl;
        }
}
```

Nove objekte pravimo i čitamo iz baze podataka na ranije opisan način:

```
EncPodatak* EncPodatak::NapraviPodatak(
                            string tip, const Podatak& p
                            )
{
    EncPodatak* obj = 0;
    if( tip == "tekst" )
        obj = new EncTekst( p );
    else if( tip == "slika" )
        obj = new EncSlika( p );
    else if( tip == "zvuk" )
        obj = new EncZvuk( p );
```

```
    else if( tip == "film" )
        obj = new EncFilm( p );
    return obj;
}

EncPodatak* EncPodatak::ProcitajPodatakIzBazePodataka( int id )
{
    Podatak p;
    string tip;
    return BazaPodataka::ProcitajPodatak( id, p, tip )
                ? NapraviPodatak( tip, p )
                : 0;
}
```

Pomoćni metod `Atribut` upotrebljavamo za izdvajanje atributa podataka:

```
string EncPodatak::Atribut( const Podatak& p, string s )
{
    string v;
    Podatak::const_iterator i = p.find(s);
    if( i != p.end() )
        v = i->second;
    return v;
}
```

Klasa EncTekst

Klasa `EncTekst` predstavlja konkretan tekstualni enciklopedijski podatak. Potrebno je definisati konstruktor i nove pristupne metode i implementirati nasleđene apstraktne metode:

```
class EncTekst : public EncPodatak
{
public:
    //  Konstruktor
    EncTekst( const Podatak& p )
        : EncPodatak( p ),
          _Tekst( Atribut(p,"tekst") )
        {}

    //  Prikazivanje i pisanje
    bool ZapisiSadrzaj( string ) const
        { return true; }

protected:
    //  Pristupni metodi
    string Tekst() const
        { return _Tekst; }
    string Tip() const
        { return "tekst"; }

    //  Prikazivanje dela podatka
    void PrikaziTekst( ostream& ostr ) const
        { ostr << Tekst() << endl; }
    void PrikaziDetalje( ostream& ) const
        {}

private:
```

```
    //  Članovi podaci
    string  _Tekst;
};
```

Klasa EncBinarni

Klasa `EncBinarni` apstrahuje sve tipove ekciklopedijskih podataka koji imaju neki binarni sadržaj. Kao u u slučaju klase `EncTekst`, potrebno je definisati konstruktor i nove pristupne metode. Implementiraju se metodi za prikazivanje teksta i zapisivanje binarnog sadržaja, ali i deklariše novi apstraktan metod za izdvajanje binarnog sadržaja:

```
class EncBinarni : public EncPodatak
{
public:
    //  Konstruktor
    EncBinarni( const Podatak& p )
        : EncPodatak( p ),
          _Napomena( Atribut(p,"napomena") )
        {}

    //  Prikazivanje i pisanje
    bool ZapisiSadrzaj( string imedat ) const
        {
        if( imedat.empty() )
            return false;

        ofstream f( imedat.c_str(), ios::binary );
        if( !f )
            return false;
        const string& s = BinarniSadrzaj();
        f.write( s.c_str(), s.length() );
        if( !f )
            return false;

        return true;
        }

protected:
    //  Pristupni metodi
    string Napomena() const
        { return _Napomena; }
    virtual const string& BinarniSadrzaj() const = 0;

    //  Prikazivanje dela podatka
    void PrikaziTekst( ostream& ostr ) const
        { ostr << Napomena() << endl; }

private:
    //  Članovi podaci
    string  _Napomena;
};
```

Pri otvaranju datoteke je eksplicitno navedeno da je potrebno raditi u binarnom režimu. Podrazumevani režim rada je tekstualni. Razlika između ovih režima je u tome što se kod nekih operativnih sistema (u koje spada i Windows) pretpostavlja specifično ponašanje pri čitanju pojedinih bajtova u tekstualnom režimu. Na primer, jedan znak `'\n'` red se zapisuje kao sekvenca `"\r\n"`, a ta sekvenca se čita kao jedan znak.

Ostale klase

Klase `EncSlika`, `EncZvuk` i `EncFilm` su međusobno slične, pa ćemo ovde predstaviti samo definiciju klase `EncSlika`. Definišemo konstruktor i nove pristupne metode i implementiramo sve preostale apstraktne metode:

```
class EncSlika : public EncBinarni
{
public:
    //  Konstruktor
    EncSlika( const Podatak& p )
        : EncBinarni( p ),
          _Sirina( ProcitajCeoBroj(Atribut(p,"sirina")) ),
          _Visina( ProcitajCeoBroj(Atribut(p,"visina")) ),
          _Slika( Atribut(p,"slika") )
        {}

protected:
    //  Pristupni metodi
    int Sirina() const
        { return _Sirina; }
    int Visina() const
        { return _Visina; }
    const string& BinarniSadrzaj() const
        { return _Slika; }
    string Tip() const
        { return "slika"; }

    //  Prikazivanje dela podatka
    void PrikaziDetalje( ostream& ostr ) const
        {
        ostr << endl;
        PrikaziDimenzije( ostr, Sirina(), Visina() );
        }

private:
    //  Članovi podaci
    int     _Sirina;
    int     _Visina;
    string  _Slika;
};
```

Korak 8 - Implementacija glavne funkcije programa

U glavnoj funkciji programa posebnu pažnju posvećujemo mogućim greškama pri pokretanju ili izvršavanju programa.

```
int main(int argc, char* argv[])
{
    // Ako nema dovoljno argumenata, ispišemo uputstvo
    if( argc < 2 ){
        cerr << "Upotreba: " << endl
             << "   " << argv[0] << " <id> [<bindat>]" << endl;
        return 2;
        }

    // Pročitamo redni broj i odgovarajući podatak
    int id = atoi( argv[1] );
```

```
    EncPodatak* p
        = EncPodatak::ProcitajPodatakIzBazePodataka( id );

    // Ako nema traženog podatka obavestimo korisnika i završimo
    if(!p){
        cerr << "Ne postoji podatak sa rednim brojem "
             << id << "!" << endl;
        return 1;
        }

    // Prikažemo podatak,
    p->Prikazi( cout );

    // Ako je na raspolaganju datoteka,
    // zapišemo binarni sadržaj podatka u datoteku
    if( argc > 2 ){
        if( !p->ZapisiSadrzaj( argv[2] ))
            cerr << "Nije uspelo pisanje u datoteku \""
                 << argv[2] << "\"!" << endl;
        }
    // inace proverimo da li je bila potrebna
    else if( !p->ZapisiSadrzaj( "" ))
        cout << "Binarni sadrzaj nije zapisan "
             << "jer nije naveden naziv datoteke!" << endl;

    // Uklonimo viškove
    delete p;

    // Sve je u redu
    return 0;
}
```

Korak 9 - Organizacija teksta programa

Do sada se nismo mnogo bavili organizovanjem teksta programa po datotekama. U nekim jednostavnijim situacijama to pitanje nije posebno važno. Međutim, kada program počne da raste i broj klasa postane veći, možemo se susresti sa nekim novim problemima koji se jednostavno rešavaju dobrim organizovanjem programskog teksta.

Često se u deklaracijama argumenata ili rezultata metoda pojavljuje ime neke druge klase. Ako se argument ili rezultat prenosi po vrednosti, neophodno je da takvoj deklaraciji prethodi definicija odgovarajuće klase. Ako se prenosi po imenu (tj. primenom pokazivača ili reference) tada je dovoljno da toj deklaraciji prethodi deklaracija imena klase. Isto važi i za članove podatke – ako član podatak predstavlja objekat druge klase, neophodno je da deklaraciji prethodi definicija te druge klase, a ako predstavlja referencu ili pokazivač na objekat druge klase, dovoljno je da prethodi deklaracija imena druge klase. Ukoliko implementacija metoda koristi više informacija o argumentu ili rezultatu koji je pokazivač ili referenca na neku drugu klasu, tj. članove podatke ili metode te klase, tada implementaciji metoda mora da prethodi definicija te druge klase.

Deklaracija imena klase ima oblik:

```
class ImeKlase;
```

U našem slučaju, pri implementaciji hijerarhije klasa enciklopedijskih podataka dolazimo u situaciju da implementaciji metoda `NapraviPodatak` klase `EncPodatak` moraju prethoditi definicije svih ostalih klasa hijerarhije. To je sasvim jednostavan problem, ali se njegovo rešavanje ne razlikuje značajno od rešavanja daleko složenijih problema tog tipa.

Osnovna ideja je u razdvajanju definicija klasa i implementacija metoda u zasebne datoteke. Definicije klasa se obično zapisuju u *zaglavljima*. Zaglavlja su datoteke koje obično imaju nastavak imena `.h` ili `.hpp`. Implementaciju metoda zapisujemo u programskim datotekama, koje obično imaju nastavak imena `.cpp` ili `.C`.

Definiciju klase `EncPodatak` ćemo zapisati u datoteci `EncPodatak.h`:

```
#ifndef EncPodatakH
#define EncPodatakH

#include <iostream>
using std::ostream;

#include "BazaPodataka.h"

//---------------------------------------------------------------
// Klasa EncPodatak
//---------------------------------------------------------------
class EncPodatak
{
...
};

#endif  //  #ifndef EncPodatakH
```

Za deklarisanje metoda `NapraviPodatak` nije potrebno da se zna koje su to ostale klase hijerarhije. Jedino zaglavlje koje moramo uključiti jeste `iostream`. Umesto uobičajene deklaracije upotrebe prostora imena:

```
using namespace std;
```

deklarisali smo samo konkretna imena iz prostora imena `std` koja koristimo. Ovde je to učinjeno radi ilustracije, ali u nekim situacijama takvo pojedinačno deklarisanje omogućava fleksibilniju upotrebu različitih biblioteka.

Implementacije metoda klase `EncPodatak` zapisaćemo u datoteci `EncPodatak.cpp`. Da bi implementaciji metoda `NapraviPodatak` prethodile definicije svih ostalih klasa hijerarhije, uključićemo zaglavlje `EncKlase.h` u kome će one biti definisane.

```
#include "EncPodatak.h"
#include "EncKlase.h"

//---------------------------------------------------------------
// Klasa EncPodatak - Implementacija metoda
//---------------------------------------------------------------
EncPodatak::EncPodatak( const Podatak& p )
...
```

Ostale klase su sasvim jednostavne. Zbog relativno malog broja klasa i njihove jednostavnosti možemo se odlučiti da sve klase zapišemo u jednoj datoteci, kao i da

implementacije metoda izvedemo u okviru definicija klasa. Kao što smo već nagovestili, to je datoteka `EncKlase.h`:

```
#ifndef EncKlaseH
#define EncKlaseH

#include "EncPodatak.h"

//---------------------------------------------------------------
// Klasa EncTekst
//---------------------------------------------------------------
class EncTekst : public EncPodatak
{
...
};
...
#endif  //  #ifndef EncKlaseH
```

Glavni program zapisujemo u posebnoj programskoj datoteci `Enciklopedija.cpp`.

```
#include <iostream>
using std::cout;
using std::cerr;
using std::endl;

#include "EncPodatak.h"

//---------------------------------------------------------------
// Program Enciklopedija.
//---------------------------------------------------------------
// Omogućava čitanje i prikazivanje podataka iz enciklopedije
// i zapisivanje binarnih sadržaja u datoteke.
//---------------------------------------------------------------
int main(int argc, char* argv[])
{...}
```

7.3 Rešenje

Datoteka EncPodatak.h

```
#ifndef EncPodatakH
#define EncPodatakH

#include <iostream>
using std::ostream;

#include "BazaPodataka.h"

//---------------------------------------------------------------
// Klasa EncPodatak
//---------------------------------------------------------------
// Predstavlja osnovnu klasu hijerarhije enciklopedijskih
// podtaka. Obuhvata i pomoćne metode za čitanje podataka
// iz baze podataka i njihovo prikazivanje.
//---------------------------------------------------------------
class EncPodatak
{
public:
```

```
    //  Konstruktor
    EncPodatak( const Podatak& p );

    //  Destruktor
    virtual ~EncPodatak()
        {}

    //  Prikazivanje i pisanje
    void Prikazi( ostream& ostr ) const;
    virtual bool ZapisiSadrzaj( string imedat ) const = 0;

    //  "Konstruktori"
    static EncPodatak* NapraviPodatak(
                           string tip, const Podatak &p
                           );
    static EncPodatak* ProcitajPodatakIzBazePodataka( int id );

protected:
    //  Pristupni metodi
    int Id() const
        { return _Id; }
    string Naslov() const
        { return _Naslov; }
    string Autor() const
        { return _Autor; }
    virtual string Tip() const = 0;

    //  Prikazivanje dela podatka
    void PrikaziNaslov( ostream& ostr ) const;
    void PrikaziAutora( ostream& ostr ) const;
    void PrikaziPovezane( ostream& ostr ) const;
    virtual void PrikaziTekst( ostream& ostr ) const = 0;
    virtual void PrikaziDetalje( ostream& ostr ) const = 0;

    //  Pomoćni metodi za prikazivanje
    static void PrikaziDimenzije( ostream& ostr, int s, int v );
    static void PrikaziTrajanje( ostream& ostr, double sec );

    //  Pomoćni metodi za čitanje podataka iz strukture Podatak
    static string Atribut( const Podatak& p, string s );
    static int ProcitajCeoBroj( string s )
        { return atoi( s.c_str() ); }
    static double ProcitajRealanBroj( string s )
        { return atof( s.c_str() ); }

private:
    //  Članovi podaci
    int     _Id;
    string  _Naslov;
    string  _Autor;
};

//---------------------------------------------------------------
#endif  //  #ifndef EncPodatakH
```

Datoteka EncPodatak.cpp

```
#include "EncPodatak.h"
#include "EncKlase.h"
```

```
//-------------------------------------------------------------
// Klasa EncPodatak - Implementacija metoda
//-------------------------------------------------------------
// Predstavlja osnovnu klasu hijerarhije enciklopedijskih
// podtaka. Obuhvata i pomoćne metode za čitanje podataka
// iz baze podataka i njihovo prikazivanje.
//-------------------------------------------------------------
EncPodatak::EncPodatak( const Podatak& p )
    : _Id( ProcitajCeoBroj(Atribut(p,"id")) ),
      _Naslov( Atribut(p,"naslov") ),
      _Autor( Atribut(p,"autor") )
{}

//  Prikazivanje i pisanje
void EncPodatak::Prikazi( ostream& ostr ) const
{
    PrikaziNaslov( ostr );
    PrikaziTekst( ostr );
    PrikaziDetalje( ostr );
    ostr << endl;
    PrikaziAutora( ostr );
    ostr << endl;
    PrikaziPovezane( ostr );
    ostr << endl;
}

//  Prikazivanje dela podatka
void EncPodatak::PrikaziNaslov( ostream& ostr ) const
{
    ostr << Naslov()
         << " (" << Tip() << " " << Id() << ")" << endl;
    ostr << "----------------------------------------" << endl;
}

void EncPodatak::PrikaziAutora( ostream& ostr ) const
{
    ostr << Autor() << endl;
}

//  Pomocni metodi za prikazivanje
void EncPodatak::PrikaziDimenzije( ostream& ostr, int s, int v )
{
    ostr << "Dimenzije: " << s << " x " << v << endl;
}

void EncPodatak::PrikaziTrajanje( ostream& ostr, double sec )
{
    ostr << "Trajanje: ";
    if( sec >= 60 ){
        int m = sec / 60;
        ostr << m << "m ";
        sec -= m * 60;
        }
    ostr << sec << "s" << endl;
}

void EncPodatak::PrikaziPovezane( ostream& ostr ) const
{
    vector<int> povezani;
```

```
    BazaPodataka::ProcitajPovezane( Id(), povezani );
    if( povezani.empty() )
        ostr << "Nema povezanih podataka." << endl;
    else{
        ostr << "Povezani podaci:" << endl;
        map<string,int> naslovi;
        for( unsigned i=0; i<povezani.size(); i++ ){
            EncPodatak* p
                = ProcitajPodatakIzBazePodataka( povezani[i] );
            if( p ){
                naslovi[ p->Naslov() + " (" + p->Tip() ]
                    = p->Id();
                delete p;
                }
            }
        map<string,int>::iterator
            i = naslovi.begin(),
            e = naslovi.end();
        for( ; i!=e; i++ )
            ostr << " - " << i->first << " "
                 << i->second << ")" << endl;
        }
}

//  "Konstruktori"
EncPodatak* EncPodatak::NapraviPodatak(
                            string tip, const Podatak& p )
{
    EncPodatak* obj = 0;
    if( tip == "tekst" )
        obj = new EncTekst( p );
    else if( tip == "slika" )
        obj = new EncSlika( p );
    else if( tip == "zvuk" )
        obj = new EncZvuk( p );
    else if( tip == "film" )
        obj = new EncFilm( p );
    return obj;
}

EncPodatak* EncPodatak::ProcitajPodatakIzBazePodataka( int id )
{
    Podatak p;
    string tip;
    return BazaPodataka::ProcitajPodatak( id, p, tip )
                ? NapraviPodatak( tip, p )
                : 0;
}

//  Pomocni metodi za citanje broja iz niske
string EncPodatak::Atribut( const Podatak& p, string s )
{
    string v;
    Podatak::const_iterator i = p.find(s);
    if( i != p.end() )
        v = i->second;
    return v;
}
```

Datoteka EncKlase.h

```
#ifndef EncKlaseH
#define EncKlaseH

#include <fstream>
using std::endl;
using std::ofstream;
using std::ios;

#include "EncPodatak.h"

//---------------------------------------------------------------
// Klasa EncTekst
//---------------------------------------------------------------
// Tekstualni enciklopedijski podatak.
//---------------------------------------------------------------
class EncTekst : public EncPodatak
{
public:
    //  Konstruktor
    EncTekst( const Podatak& p )
        : EncPodatak( p ),
          _Tekst( Atribut(p,"tekst") )
        {}

    //  Prikazivanje i pisanje
    bool ZapisiSadrzaj( string ) const
        { return true; }

protected:
    //  Pristupni metodi
    string Tekst() const
        { return _Tekst; }
    string Tip() const
        { return "tekst"; }

    //  Prikazivanje dela podatka
    void PrikaziTekst( ostream& ostr ) const
        { ostr << Tekst() << endl; }
    void PrikaziDetalje( ostream& ) const
        {}

private:
    //  Članovi podaci
    string  _Tekst;
};

//---------------------------------------------------------------
// Klasa EncBinarni
//---------------------------------------------------------------
// Osnova za sve binarne (multimedijalne) enciklopedijske podatke.
//---------------------------------------------------------------
class EncBinarni : public EncPodatak
{
public:
```

```
    //  Konstruktor
    EncBinarni( const Podatak& p )
        : EncPodatak( p ),
          _Napomena( Atribut(p,"napomena") )
        {}

    //  Prikazivanje i pisanje
    bool ZapisiSadrzaj( string imedat ) const
        {
        if( imedat.empty() )
            return false;

        ofstream f( imedat.c_str(), ios::binary );
        if( !f )
            return false;
        const string& s = BinarniSadrzaj();
        f.write( s.c_str(), s.length() );
        if( !f )
            return false;

        return true;
        }

protected:
    //  Pristupni metodi
    string Napomena() const
        { return _Napomena; }
    virtual const string& BinarniSadrzaj() const = 0;

    //  Prikazivanje dela podatka
    void PrikaziTekst( ostream& ostr ) const
        { ostr << Napomena() << endl; }

private:
    //  Članovi podaci
    string  _Napomena;
};

//-----------------------------------------------------------------
// Klasa EncSlika
//-----------------------------------------------------------------
// Slika iz enciklopedije.
//-----------------------------------------------------------------
class EncSlika : public EncBinarni
{
public:
    //  Konstruktor

    EncSlika( const Podatak& p )
        : EncBinarni( p ),
          _Sirina( ProcitajCeoBroj(Atribut(p,"sirina")) ),
          _Visina( ProcitajCeoBroj(Atribut(p,"visina")) ),
          _Slika( Atribut(p,"slika") )
        {}

protected:
    //  Pristupni metodi
    int Sirina() const
        { return _Sirina; }
    int Visina() const
        { return _Visina; }
```

```
    const string& BinarniSadrzaj() const
        { return _Slika; }
    string Tip() const
        { return "slika"; }

    //  Prikazivanje dela podatka
    void PrikaziDetalje( ostream& ostr ) const
        {
        ostr << endl;
        PrikaziDimenzije( ostr, Sirina(), Visina() );
        }

private:
    //  Članovi podaci
    int     _Sirina;
    int     _Visina;
    string  _Slika;
};

//---------------------------------------------------------------
// Klasa EncZvuk
//---------------------------------------------------------------
// Zvuk iz enciklopedije.
//---------------------------------------------------------------
class EncZvuk : public EncBinarni
{
public:
    //  Konstruktor
    EncZvuk( const Podatak& p )
        : EncBinarni( p ),
          _Trajanje( ProcitajRealanBroj(Atribut(p,"trajanje")) ),
          _Zvuk( Atribut(p,"zvuk") )
        {}

protected:
    //  Pristupni metodi
    double Trajanje() const
        { return _Trajanje; }
    const string& BinarniSadrzaj() const
        { return _Zvuk; }
    string Tip() const
        { return "zvuk"; }

    //  Prikazivanje dela podatka
    void PrikaziDetalje( ostream& ostr ) const
        {
        ostr << endl;
        PrikaziTrajanje( ostr, Trajanje() );
        }

private:
    //  Članovi podaci
    double  _Trajanje;
    string  _Zvuk;
};
```

```
//-------------------------------------------------------------
// Klasa EncFilm
//-------------------------------------------------------------
// Film iz enciklopedije.
//-------------------------------------------------------------
class EncFilm : public EncBinarni
{
public:
    //  Konstruktor
    EncFilm( const Podatak& p )
        : EncBinarni( p ),
          _Sirina( ProcitajCeoBroj(Atribut(p,"sirina")) ),
          _Visina( ProcitajCeoBroj(Atribut(p,"visina")) ),
          _Trajanje( ProcitajRealanBroj(Atribut(p,"trajanje")) ),
          _Film( Atribut(p,"film") )
        {}

protected:
    //  Pristupni metodi
    int Sirina() const
        { return _Sirina; }
    int Visina() const
        { return _Visina; }
    double Trajanje() const
        { return _Trajanje; }
    const string& BinarniSadrzaj() const
        { return _Film; }
    string Tip() const
        { return "film"; }

    //  Prikazivanje dela podatka
    void PrikaziDetalje( ostream& ostr ) const
        {
        ostr << endl;
        PrikaziDimenzije( ostr, Sirina(), Visina() );
        PrikaziTrajanje( ostr, Trajanje() );
        }

private:
    //  Članovi podaci
    int     _Sirina;
    int     _Visina;
    double  _Trajanje;
    string  _Film;
};

//-------------------------------------------------------------
#endif  //  #ifndef EncKlaseH
```

Datoteka Enciklopedija.cpp

```
#include <iostream>
using std::cout;
using std::cerr;
using std::endl;

#include "EncPodatak.h"
```

```
//-------------------------------------------------------------
// Program Enciklopedija.
//-------------------------------------------------------------
// Omogućava čitanje i prikazivanje podataka iz enciklopedije
// i zapisivanje binarnih sadržaja u datoteke.
//-------------------------------------------------------------
int main(int argc, char* argv[])
{
    // Ako nema dovoljno argumenata, ispišemo uputstvo
    if( argc < 2 ){
        cerr << "Upotreba: " << endl
             << "   " << argv[0] << " <id> [<bindat>]" << endl;
        return 2;
        }

    // Pročitamo redni broj i odgovarajući podatak
    int id = atoi( argv[1] );
    EncPodatak* p
        = EncPodatak::ProcitajPodatakIzBazePodataka( id );

    // Ako nema traženog podatka obavestimo korisnika i završimo
    if(!p){
        cerr << "Ne postoji podatak sa rednim brojem "
             << id << "!" << endl;
        return 1;
        }

    // Prikažemo podatak,
    p->Prikazi( cout );

    // Ako je na raspolaganju datoteka,
    // zapišemo binarni sadržaj podatka u datoteku
    if( argc > 2 ){
        if( !p->ZapisiSadrzaj( argv[2] ))
            cerr << "Nije uspelo pisanje u datoteku \""
                 << argv[2] << "\"!" << endl;
        }
    // inace proverimo da li je bila potrebna
    else if( !p->ZapisiSadrzaj( "" ))
        cout << "Binarni sadrzaj nije zapisan "
             << "jer nije naveden naziv datoteke!" << endl;

    // Uklonimo viškove
    delete p;

    // Sve je u redu
    return 0;
}
```

7.4 Rezime

Predlažemo da se za vežbu klase podataka napišu uz primenu druge ponuđene varijante za definisanje interne strukture: u baznoj klasi hijerarhije obezbeđuje se član podatak tipa `Podatak`; u klasama naslednicama nema potrebe za novim članovima podacima; pristupni metodi neposredno izdvajaju odgovarajuće sadržaje iz tog podatka.

Posebnu vežbu, koja se više tiče upoznavanja i korišćenja radnog okruženja (operativni sistem, razvojni alati i sl.), može predstavljati pisanje programa koji bi umesto (ili nakon) zapisivanja binarnog sadržaja u datoteku izvodio predstavljanje tog binarnog sadržaja korisniku primenom odgovarajućih programa za pregledanje slika, slušanje zvučnih zapisa ili gledanje video zapisa.

8 - Kodiranje

8.1 Zadatak

Napisati program za kodiranje i dekodiranje datoteka sa proizvoljnim sadržajem. Kao parametri programa se navode operacija (kodiranje ili dekodiranje) i oznaka primenjene transformacije. Omogućiti primenu više elementarnih transformacija kodiranja, kao i njihovu kompoziciju.

Program se upotrebljava primenom sledeće sintakse:

```
Kodiranje (k|d) <ulazna dat.> <izlazna dat.> <transformacija>
```

gde se slovom `k` označava operacija kodiranja, a slovom `d` operacija dekodiranja. Transformacija se navodi kao jedan ili više opisa elementarnih transformacija. Ukoliko se navedu opisi više od jedne transformacije, primenjuje se kompozicija navedenih transformacija.

Potrebno je podržati sledeće elementarne operacije kodiranja:

- translacija za datu vrednost – svaki bajt ulazne datoteke se kodira bajtom koji je od njega veći (po modulu 256) za datu vrednost (npr. translacija za 17 transformiše niz bajtova 2, 100, 250, 255, 7 u niz bajtova 19, 117, 11, 16, 24);
  - sintaksa je: **`trans`** `<n>`
- rotacija za dati broj mesta – svaki bajt se kodira bajtom koji se dobija kada se zarotira za dati broj mesta (bitova) ulevo (npr. rotacija za 3 mesta ulevo transformiše niz bajtova 23, 100, 250, 255, 73 u niz 184, 35, 215, 255, 74);
  - sintaksa je: **`rot`** `<n>`
- zamena – svaki par bajtova zamenjuje mesta (npr. niz bajtova 2, 100, 250, 255, 7 se transformiše u niz bajtova 100, 2, 255, 250, 7);
  - sintaksa je: **`zamena`**

- ekskluzivna disjunkcija datom niskom – kao parametar se zadaje niz znakova (tj. bajtova), a kodirani niz bajtova se dobija tako što se na bajtove ulazne datoteke redom primenjuje ekskluzivna disjunkcija sa bajtovima niske koja je data kao parametar (npr. ako se ulazna datoteka sastoji od bajtova ulaz[0], ulaz[1], ulaz[2], ulaz[3],... ulaz[n], a kao vrednost parametra je data niska „abc“, tada se kao rezultat dobija niz bajtova: ulaz[0] ^ 'a', ulaz[1] ^ 'b', ulaz[2] ^ 'c', ulaz[3] ^ 'a',... ulaz[n] ^ parametar[n%3]);
  - sintaksa je: **`xor`** `<niska>`
  - niska može biti navedena sa ili bez navodnika, prema pravilima konkretnog komandnog interfejsa
- ekskluzivna dijsunkcija početnim nizom date dužine – transformacija se izvodi kao u slučaju „obične“ ekskluzivne disjunkcije, s tim da se kao parametar upotrebljava niz koji se sastoji od prvih nekoliko bajtova ulazne datoteke, pri čemu se tih prvih nekoliko bajtova ne kodira (npr. ako je vrednost parametra 3 i ulazna datoteka se sastoji od bajtova ulaz[0], ulaz[1], ulaz[2], ulaz[3],... ulaz[n], tada se kao rezultat dobija niz bajtova: ulaz[0], ulaz[1], ulaz[2], ulaz[3] ^ ulaz[0],... ulaz[n] ^ ulaz[n%3]);
  - sintaksa je: **`xorstart`** `<n>`

Složena transformacija kodiranja se definiše kao kompozicija datih elementarnih transformacija.

Priloženom naredbom se izvodi kodiranje datoteke `ulaz.dat`. Rezultat se zapisuje u datoteku `izlaz.dat`. Primenjuju se, redom, translacija za 17, zamena, rotacija za 2 mesta i ekskluzivna disjunkcija početnim nizom bajtova dužine 15:

```
kodiranje k ulaz.dat izlaz.dat trans 17 zamena rot 2 xorstart 15
```

Cilj zadatka

Rešavanjem ovog zadatka:

- pokazaćemo kako se hijerarhijom klasa predstavljaju operacije koje primenjujemo na neke druge objekte;
- na primeru ćemo predstaviti pisanje kompozitnih klasa;
- predstavićemo koncept refaktorisanja;
- predstavićemo klase izuzetaka standardne biblioteke.

Pretpostavljena znanja

Za praćenje ovog primera potrebno je poznavanje uglavnom svih elemenata programskog jezika C++, a pre svega:

- rada sa pokazivačima i dinamičkim strukturama podataka;
- izgradnje hijerarhija klasa i dinamičkog vezivanja metoda;

- rada sa izuzecima;
- klasa standardne biblioteke: `string` i `vector`;
- standardne biblioteke tokova i datotečnih tokova;
- operacija na bitovima u programskom jeziku C++.

8.2 Rešavanje zadatka

8.2.1 Analiza problema

U prethodnim primerima smo hijerarhijama klasa predstavljali objekte nad kojima smo izvodili neke operacije. Koliko god da su te operacije bile složene, bile su jednoznačno definisane. Raznovrsnost u rešavanju problema je bila u domenu objekata nad kojima su se te operacije izvodile. Zbog određenih razlika u načinu izvođenja operacija na različitim objektima, grupisali smo objekte u klase prema načinu na koji se operacije na njima izvode i od definisanih klasa pravili odgovarajuće hijerarhije klasa.

Ovde je situacija sasvim drugačija. Objekti na kojima je potrebno izvoditi operacije su datoteke. Njih ćemo, radi jednostavnosti, posmatrati kao nizove bajtova. Posmatrano iz ugla rešavanja problema, jedina razlika među datotekama je u njihovom sadržaju, tj. u broju i vrednostima konkretnih bajtova od kojih se datoteke sastoje. Međutim, to ne utiče značajno na način izvođenja operacija kodiranja i dekodiranja. Ovde je složenost problema posledica postojanja više različitih operacija kodiranja koje moramo podržati. Svaka od njih se primenjuje na sve datoteke na isti način, pa se eventualnom podelom datoteka na više klasa ni najmanje ne približavamo rešenju problema. Sa druge strane, činjenica da se operacije razlikuju, kao i da se mogu kombinovati i graditi kompozitne operacije, ukazuje na potrebu da oblikujemo hijerarhiju klasa operacija.

Kao što smo u prethodno napisanim hijerarhijama definisali virtualne metode i tako obezbedili da se izvođenje odgovarajućih operacija *nad objektima* prilagođava konkretnim klasama objekata, tako ćemo sada definisati hijerarhiju klasa transformacija kodiranja i virtualnim metodima obezbediti da izvođenje odgovarajućih operacija *od strane objekata transformacija a nad datim nizovima bajtova* bude prilagođeno odgovarajućim transformacijama kodiranja.

8.2.2 Način rešavanja zadatka

Da bi praćenje i razumevanje postupka rešavanja bilo nešto lakše, pretpostavićemo da se transformacije kodiranja i dekodiranja obavljaju na nizovima bajtova. Na taj način ćemo rad sa datotekama izmestiti u posebne faze koje zaokružuju samo kodiranje.

Rešavanju problema ćemo prići odozgo naniže – najpre ćemo definisati kostur programa i pomoćne klase za rad sa nizovima bajtova. Zatim ćemo implementirati jednu po jednu elementarnu transformaciju kodiranja. Na kraju ćemo definisati kompozitnu transformaciju:

Korak 1 - Kostur programa

Ako pretpostavimo da glavna funkcija programa mora da izvrši analizu parametara programa, pročita sadržaj ulazne datoteke, primeni operaciju kodiranja ili dekodiranja i upiše rezultat u izlaznu datoteku, onda bi ona mogla da izgleda otprilike ovako:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

//-------------------------------------------------------------
//  Program za kodiranje/dekodiranje
//-------------------------------------------------------------
main( int argc, char** argv )
{
    //  Površna provera da li je na raspolaganju
    //  dovoljno parametara
    if( argc < 5 ){
        cerr << "Neispravna upotreba programa!" << endl;
        cerr << "Nisu navedeni svi neophodni parametri!" << endl;
        cerr << "Kodiranje (k|d) <ulazna dat.> <izlazna dat.>"
                " <transformacija>" << endl;
        return 1;
        }
    else{
        //  Smer obavljanja operacije
        bool kodiranje;
        string smer = argv[1];
        if( smer == "k" )
            kodiranje = true;
        else if( smer == "d" )
            kodiranje = false;
        else{
            cerr << "Neispravna upotreba programa!" << endl;
```

```
            cerr << "Tip operacije mora biti 'k' (kodiranje) ili "
                    "'d' (dekodiranje)!" << endl;
            cerr << "Kodiranje (k|d) <ulazna dat.> <izlazna dat.>"
                    " <transformacija>" << endl;
            return 1;
            }

        //  Čitanje transformacije
        //  ...detalje ostavljamo za kasnije...

        //  Otvaranje ulazne datoteke
        ifstream uDat( argv[2], ios::binary );
        if( !uDat ){
            cerr << "Nije uspelo otvaranje ulazne datoteke!";
            return 2;
            }
        //  Čitanje ulazne datoteke
        //  ...ostavljamo za kasnije...

        //  Izvođenje operacije
        if( kodiranje )
            //  Kodiranje
            //  ...ostavljamo za kasnije...
            ;
        else
            //  Dekodiranje
            //  ...ostavljamo za kasnije...
            ;

        //  Otvaranje izlazne datoteke
        ofstream iDat( argv[3], ios::binary );
        if( !iDat ){
            cerr << "Nije uspelo otvaranje izlazne datoteke!";
            return 2;
            }
        //  Pisanje izlazne datoteke
        //  ...ostavljamo za kasnije...

        return 0;
        }
}
```

Ako pogledamo napisanu (u redu, ne baš sasvim napisanu) funkciju `main` teško je oteti se utisku da tu nešto nije u redu. Ali ona nije neispravna: analiziraju se ulazni parametri, otvaraju se i upotrebljavaju datoteke, izvodi se kodiranje ili dekodiranje... Obuhvaćene su sve potrebne operacije. A ipak, što je duže gledamo to nam je takvo gledanje manje prijatno. Zašto?

„Lepo" programiranje

Iskusniji programeri često podučavaju svoje mlađe saradnike kako „programi moraju biti lepi da bi bili dobri". U programerskom jeziku lepota i jednostavnost obično idu zajedno. Razlog uskog vezivanja pojmova lepote i jednostavnosti je pre svega u posledicama koje na određen način napisan kod nosi po autora i korisnike. Tekst programa koji nije jednostavno i dobro oblikovan potencijalno nosi veoma neprijatne trenutke za one koji ga ispravljaju i dograđuju. Mogli bismo se našaliti (mada to nije daleko od istine) da iskustvo kod programera stvara određeni refleks pa je već i običan pogled na „ružan" kod dovoljan da se

osete poprilično loše. Iskustvo ispravljanja i proširivanja loše napisanih programa je nešto što bi malo ko imao obzira da poželi svojim kolegama, ali je istovremeno i nešto kroz šta praktično svaki programer u nekom trenutku svog stručnog usavršavanja prolazi.

Naša funkcija `main` je sve drugo pre nego jednostavna – pored toga što implementira osnovnu funkciju programa, tj. kodiranje/dekodiranje, ona sama obavlja i sve pripremne i završne poslove, što uključuje i analizu parametara komandne linije i detalje rada sa datotekama. Pored toga, u njoj se više puta ponavlja kod za izveštavanje o greškama, a kako ponavljanje uvodi složenost u održavanje i ispravljanje programa, sledi da i ponavljanje narušava lepotu programa, bar po estetskim merilima programera.

Kako pisati dobre (tj. lepe) programe? Na osnovu navedenog mogli bismo zaključiti da je programe potrebno pisati tako da se lako menjaju i proširuju. To, doduše, ima i svoju suprotnost – ako sve elemente programa pišemo tako da omogućimo njihovo lako proširivanje i menjanje, rezultat može biti isuviše zakomplikovan program prepun nepotrebnih apstrakcija. Sa druge strane, činjenica je da *nikada* ne možemo unapred prepoznati sve buduće potrebe za menjanjem i dograđivanjem, pa za neke izmene sigurno nećemo unapred pripremiti program.

Problem je u tome što uvođenjem apstrakcija koje bi trebalo da učine jednostavnijim neke moguće izmene programa, po pravilu otežavamo izvođenje svih ostalih potencijalnih izmena. Naime, ako se ispostavi da je pre očekivanih izmena potrebno izvesti neke druge, neočekivane, tada je pri izvođenju tih izmena neophodno voditi dodatnu brigu o tome da se pretpostavke za one „očekivane“ izmene najpre ne naruše a zatim i prilagode nastalim izmenama. Zato je dobra praksa da se posebna pažnja posveti *samo onim mogućim izmenama i dopunama za koje je skoro sasvim izvesno da će biti implementirane u nekom skorijem periodu života programa*. Sve ostale izmene će biti dovoljno olakšane samim činom *lepog pisanja* programa.

Programerska estetika se obično svodi na poštovanje jednostavnih pravila, čije nepoštovanje (pre ili kasnije) vodi u probleme. Primeri takvih pravila su:

- izbegavati ponavljanje koda – ponavljanje do dva puta se obično može tolerisati, dok ponavljanje istog (ili skoro istog) koda tri ili više puta u istom programu nije dopustivo;
- svaki metod (ili funkcija) bi trebalo da izvodi tačno jednu operaciju – svaki pokušaj definisanja metoda (funkcije) koji obavlja više operacija otežava najpre razumevanje i upotrebu metoda (funkcije), a zatim i menjanje i dopunjavanje;
- davanje preciznih imena metodima (i funkcijama) – ime mora potpuno jasno identifikovati ne samo fizičku definiciju metoda već i njegovu namenu;
- i druga pravila sličnog karaktera.

Pored navedenih pravila postoji i veći broj pravila koja se mogu nešto slobodnije tumačiti. Zajedničko za ova pravila je da su ona u načelu sasvim u redu, ali da njihova neumerena ili neoprezna primena može biti kontraproduktivna. Tipični primeri ovakvih pravila su:

- ne upotrebljavati komentare u telu metoda (i funkcija) – ako nije sasvim očigledno šta radi neki niz naredbi, onda ga je potrebno izdvojiti u poseban metod (funkciju) čije će ime jednoznačno opisati funkciju i namenu metoda;
- svaka hijerarhija klasa se gradi oko tačno jednog aspekta ponašanja objekata – građenje hijerarhije na temelju više različitih aspekata ponašanja objekata će pre ili kasnije izvesno dovesti do značajnih dilema po pitanju odnosa među klasama hijerarhije;
- ako neki podatak predstavlja čest predmet obrade ili apstrahuje neku obradu, potrebno je razmotriti modeliranje tog podatka klasom ili hijerarhijom klasa.

Slepo pridržavanje prvog pravila može voditi definisanju izuzetno velikog broja sasvim jednostavnih metoda. To, dalje, ima za posledicu da se *od drveća ne vidi šuma*, tj. da se zbog mnoštva pojedinačnih metoda značajno otežava uočavanje najvažnijih među njima, kao i sagledavanje ponašanja klase kao celine.

Drugo pravilo je nešto lakše primenjivati u programskim jezicima u kojima je moguće višestruko nasleđivanje. U ostalim programskim jezicima su često ipak neophodni kompromisi po ovom pitanju. Njegova nekritička primena može voditi kako prevelikom broju različitih hijerarhija klasa tako i usložnjavanju odnosa među klasama.

Treće pravilo je sasvim razumno, ali svaka apstrakcija mora imati svoje granice. Ako apstrahujemo svaki element programa i svaki aspekt ponašanja objekata koji se u programu pojavljuju, veoma lako se dolazi u sukob sa ostalim pravilima.

Posle svega rečenog i dalje nije sasvim jasno kako bi to trebalo da izgleda „lep“ tj. „dobar“ program? Martin Fowler je u knjizi *Refaktorisanje* [Fowler 1999] naveo simpatičan citat bake svog saradnika i prijatelja[12]:

„Ako zaudara, a ti ga presvuci. “

Navodeći ovaj citat kao jedinstveno i sveobuhvatno pravilo definisanja *lepih*[13] programa, Martin Fowler je na nedvosmislen i sasvim jednostavan način ukazao na činjenicu da svaki programer ima specifičan prag osetljivosti za različite potencijalne probleme koje složenosti u programima donose. Ta subjektivnost izvesno vodi i subjektivnom doživljavanju pojmova dobrog i lepog u programiranju. Svaki programer može definisati svoja jednostavna pravila koja za rezultat imaju programski kod koji je lep po sopstvenim kriterijumima autora. Važno je da se od pravila ne odstupa i da se ona konzistentno primenjuju u jednoj programskoj celini, kako bi ta celina mogla da *odiše lepotom*.

[12] U izdanju knjige na srpskom jeziku upotrebljen je prevod "ako zaudara, a ti ga izmeni", ali kako na engleskom jeziku reč *change* pored značenja *izmeniti* ima i značenje *presvući*, verujem da će čitaocima koji su osetili draži roditeljstva upotrebljen prevod biti slikovitiji.

[13] Cepidlake bi mogle primetiti da se ovde uvodi vrlo kompromisno shvatanje pojma lepote, po kome je lepo sve ono što ne zaudara. Eh, ti programeri...

Svakako da je za timski rad neophodno približavanje ličnih kriterijuma i pravila članova tima. Da bi tim uspešno funkcionisao, odgovarajuća pravila, pa samim tim i tumačenje pojmova dobrog i lepog programa, moraju se definisati na nivou tima. Jedan sistematičan pregled pravila za pisanje objektno orijentisanih programa dat je u knjizi [Riel 1996].

Klasa Program

Prvi korak ka lepšoj funkciji `main` je izdvajanje određenih celina u zasebne funkcije:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

//-----------------------------------------------------------------
//  Uputstvo
//-----------------------------------------------------------------
void ispisiGreskuUParametrima( char* poruka )
{
    cerr <<"Neispravna upotreba programa!" << endl;
    cerr << poruka << endl;
    cerr << "Kodiranje (k|d) <ulazna dat.> <izlazna dat.> "
            "<transformacija>" << endl;
}

//-----------------------------------------------------------------
//  Prepoznati parametri
//-----------------------------------------------------------------
bool kodiranje;
string imeUlazneDatoteke;
string imeIzlazneDatoteke;

bool citanjeDatoteke( const string& s )
{
    ifstream uDat( s.c_str(), ios::binary );
    if( !uDat ){
        cerr << "Nije uspelo otvaranje ulazne datoteke!";
        return false;
        }
    //  Čitanje ulazne datoteke
    //  ...ostavljamo za kasnije...
    return true;
}

bool pisanjeDatoteke( const string& s )
{
    ofstream iDat( s.c_str(), ios::binary );
    if( !iDat ){
        cerr << "Nije uspelo otvaranje izlazne datoteke!";
        return false;
        }
    //  Pisanje izlazne datoteke
    //  ...ostavljamo za kasnije...
    return true;
}

void izvodjenjeOperacije( bool kodiranje )
{
```

```
    if( kodiranje )
        //  Kodiranje ,,,ostavljamo za kasnije...
        ;
    else
        //  Dekodiranje
        //  ...ostavljamo za kasnije...
        ;
}

bool analizaParametara( int argc, char** argv )
{
    //  Površna provera da li je na raspolaganju
    //  dovoljno parametara
    if( argc < 5 ){
        ispisiGreskuUParametrima(
            "Nisu navedeni svi neophodni parametri!"
            );
        return false;
        }

    //  Smer obavljanja operacije
    string smer = argv[1];
    if( smer == "k" )
        kodiranje = true;
    else if( smer == "d" )
        kodiranje = false;
    else{
        ispisiGreskuUParametrima(
            "Tip operacije mora biti 'k' (kodiranje) ili "
            "'d' (dekodiranje)!"
            );
        return false;
        }

    //  Čitanje transformacije
    //  ...detalje ostavljamo za kasnije...

    //  Izdvajanje naziva ulazne i izlazne datoteke
    imeUlazneDatoteke = argv[2];
    imeIzlazneDatoteke = argv[3];

    //  Sve je u redu
    return true;
}

//---------------------------------------------------------------
//  Program za kodiranje/dekodiranje
//---------------------------------------------------------------
int main( int argc, char** argv )
{
    if( !analizaParametara( argc, argv ) )
        return 1;

    if( !citanjeDatoteke( imeUlazneDatoteke ))
        return 1;
    izvodjenjeOperacije( kodiranje );
    if( !pisanjeDatoteke( imeIzlazneDatoteke ))
        return 1;
```

```
    return 0;
}
```

Ovako izmenjen kod ima bar dve značajne karakteristike koje ga razlikuju od prethodnog:

- Složenost prethodne verzije funkcije `main` je raspoređena na više funkcija, tako da je svaka (osim, možda, `analizaParametara`) sasvim jednostavna i lako razumljiva. Funkcija `analizaParametara` je ostavljena kao celina da bi se pokazalo da se pravila ne moraju slepo pratiti. Ova funkcija, iako je nešto obimnija i složenija, ne predstavlja prepreku daljem menjanju i dogradnji programa.
- Program je sada nešto duži. Umesto jedne funkcije sada imamo nekoliko funkcija i nekoliko globalnih promenljivih.

Da li je cena prihvatljiva? Da li čistija konstrukcija programa vredi više od manje dužine koda? Obično dobra struktura koda zavređuje povećanje njegovog obima. Iako je ukupna dužina koda veća, dužina dela koda koji je potrebno razmatrati pri predstojećim izmenama je, zapravo, smanjena. Tako je i u ovom slučaju.

Šta ćemo sa globalnim promenljivim? Znamo da ih je poželjno izbegavati. Zbog toga ćemo sve i te promenljive i funkcije koje ih koriste ugraditi u klasu `ProgramKoDek`, koja opisuje ponašanje programa za kodiranje i dekodiranje:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

//----------------------------------------------------------------
//  Program
//----------------------------------------------------------------
class ProgramKoDek
{
public:
    bool izvrsi( int argc, char** argv )
        {
        return
            priprema( argc, argv )
            && obrada();
        }

private:
    bool priprema( int argc, char** argv )
        {
        //  Površna provera da li je na raspolaganju
        //  dovoljno parametara
        if( argc < 5 ){
            ispisiGreskuUParametrima(
                "Nisu navedeni svi neophodni parametri!"
                );
            return false;
            }
```

```
    //  Smer obavljanja operacije
    string smer = argv[1];
    if( smer == "k" )
        kodiranje = true;
    else if( smer == "d" )
        kodiranje = false;
    else{
        ispisiGreskuUParametrima(
            "Tip operacije mora biti 'k' (kodiranje) ili "
            "'d' (dekodiranje)!"
            );
        return false;
        }

    //  Čitanje transformacije
    //  ...detalje ostavljamo za kasnije...

    //  Izdvajanje naziva ulazne i izlazne datoteke
    imeUlazneDatoteke = argv[2];
    imeIzlazneDatoteke = argv[3];

    //  Sve je u redu
    return true;
    }

bool obrada()
    {
    if( !citanjeDatoteke())
        return false;
    izvodjenjeOperacije();
    if( !pisanjeDatoteke())
        return false;

    return true;
    }

bool citanjeDatoteke()
    {
    ifstream uDat( imeUlazneDatoteke.c_str(), ios::binary );
    if( !uDat ){
        cerr << "Nije uspelo otvaranje ulazne datoteke!";
        return false;
        }

    //  Čitanje ulazne datoteke
    //  ...ostavljamo za kasnije...
    return true;
    }

bool pisanjeDatoteke()
    {
    ofstream iDat( imeIzlazneDatoteke.c_str(), ios::binary );
    if( !iDat ){
        cerr << "Nije uspelo otvaranje izlazne datoteke!";
        return 2;
        }
    //  Pisanje izlazne datoteke
    //  ...ostavljamo za kasnije...
    return true;
    }
```

```
    void ispisiGreskuUParametrima( char* poruka )
        {
        cerr <<"Neispravna upotreba programa!" << endl;
        cerr << poruka << endl;
        cerr << "Kodiranje (k|d) <ulazna dat.> <izlazna dat.> "
                "<transformacija>" << endl;
        }
    void izvodjenjeOperacije()
        {
        if( kodiranje )
            //  Kodiranje ...ostavljamo za kasnije...
            ;
        else
            //  Dekodiranje ...ostavljamo za kasnije...
            ;
        }
    //-------------------------------------------------------------
    //  Članovi podaci
    //-------------------------------------------------------------
    bool kodiranje;
    string imeUlazneDatoteke;
    string imeIzlazneDatoteke;
};
//-----------------------------------------------------------------
//  Program za kodiranje/dekodiranje
//-----------------------------------------------------------------
int main( int argc, char** argv )
{
    ProgramKoDek program;
    if( !program.izvrsi( argc, argv ))
        return 1;
    return 0;
}
```

Izuzeci

Naredni korak u popravljanju našeg programa jeste odstranjivanje jednog prilično nedopadljivog šablona koji se ponavlja kroz čitav program. Svaki metod vraća naznaku da li je uspešno obavio posao ili ne. To bi moglo da ima smisla ukoliko bi se neki problemi mogli prevazići, ali u ovom slučaju je svaka greška fatalna i onemogućava ispravno privođenje posla kraju. Zato ćemo upotrebiti izuzetke.

Primena izuzetaka nam omogućava da izbegnemo suvišne i zamorne provere da li je pozvani metod dobro obavio svoj posao. Postojeće klase izuzetaka standardne biblioteke (videti odeljak *10.12 Izuzeci*, na strani 400) nam omogućavaju da na različit način obradimo dva moguća tipa grešaka: koristićemo izuzetke tipa `invalid_argument` ukoliko komandna linija nije ispravno zapisana, a ostale tipove izuzetaka (za sada samo `runtime_error`) u drugim neispravnim situacijama:

```
#include <iostream>
#include <fstream>
#include <string>
#include <exception>
```

```
using namespace std;

//---------------------------------------------------------------
//  Program
//---------------------------------------------------------------
class ProgramKoDek
{
public:
    void izvrsi( int argc, char** argv )
        {
        priprema( argc, argv );
        obrada();
        }

private:
    void priprema( int argc, char** argv )
        {
        //  Površna provera da li je na raspolaganju
        //  dovoljno parametara
        if( argc < 5 )
            throw invalid_argument(
                "Nisu navedeni svi neophodni parametri!"
                );

        //  Smer obavljanja operacije
        string smer = argv[1];
        if( smer == "k" )
            kodiranje = true;
        else if( smer == "d" )
            kodiranje = false;
        else
            throw invalid_argument(
                "Tip operacije mora biti 'k' (kodiranje) ili "
                "'d' (dekodiranje)!"
                );

        //  Čitanje transformacije
        citanjeTransformacije( argc, argv );

        //  Izdvajanje naziva ulazne i izlazne datoteke
        imeUlazneDatoteke = argv[2];
        imeIzlazneDatoteke = argv[3];
        }

    void citanjeTransformacije( int argc, char** argv )
        {
        //  ...detalje ostavljamo za kasnije...
        }

    void obrada()
        {
        citanjeDatoteke();
        izvodjenjeOperacije();
        pisanjeDatoteke();
        }

    void citanjeDatoteke()
        {
        ifstream uDat( imeUlazneDatoteke.c_str(), ios::binary );
```

```
        if( !uDat )
            throw runtime_error(
                "Nije uspelo otvaranje ulazne datoteke!"
                );
        //  Čitanje ulazne datoteke
        //  ...ostavljamo za kasnije...
        }

    void pisanjeDatoteke()
        {
        ofstream iDat( imeIzlazneDatoteke.c_str(), ios::binary );
        if( !iDat )
            throw runtime_error(
                "Nije uspelo otvaranje izlazne datoteke!"
                );
        //  Pisanje izlazne datoteke
        //  ...ostavljamo za kasnije...
        }

    void izvodjenjeOperacije()
        {
        if( kodiranje )
            //  Kodiranje
            //  ...ostavljamo za kasnije...
            ;
        else
            //  Dekodiranje
            //  ...ostavljamo za kasnije...
            ;
        }

    //-------------------------------------------------------------
    //  Članovi podaci
    //-------------------------------------------------------------
    bool kodiranje;
    string imeUlazneDatoteke;
    string imeIzlazneDatoteke;
};

//-----------------------------------------------------------------
//  Program za kodiranje/dekodiranje
//-----------------------------------------------------------------
main( int argc, char** argv )
{
    try {
        ProgramKoDek program;
        program.izvrsi( argc, argv );
        }
    catch( invalid_argument& e ){
        cerr <<"Neispravna upotreba programa!" << endl;
        cerr << e.what() << endl;
        cerr << "Kodiranje (k|d) <ulazna dat.> <izlazna dat.> "
                "<transformacija>" << endl;
        return 1;
        }
    catch( exception& e ){
        cerr <<"GRESKA: " << e.what() << endl;
```

```
            return 1;
            }
        return 0;
    }
```

Korak 2 - Niz bajtova

Niz bajtova ćemo implementirati primenom kolekcije `vector`. Napisaćemo klasu `NizBajtova` kako bismo u njoj obezbedili čitanje nizova bajtova iz datoteka i njihovo pisanje u datoteke. Na taj način ćemo odgovornost za rad sa datotekama prebaciti sa programa na nizove bajtova. Bajt ćemo opisati tipom `unsigned char`. Kako još nije sasvim jasno koji su nam sve metodi potrebni za pristupanje elementima niza, to ćemo ostaviti za kasnije. U ovom trenutku je dovoljno da definišemo metode za čitanje i pisanje. Sigurno je da će nam biti potreban i metod za izračunavanje veličine niza:

```
...
#include <vector>

using namespace std;

typedef unsigned char Bajt;

//---------------------------------------------------------------
//  Klasa NizBajtova
//---------------------------------------------------------------
class NizBajtova
{
public:
    unsigned velicina() const
        { return _niz.size(); }

    void pisi( ostream& ostr ) const
        {
        ostr.write( _niz.begin(), velicina() );
        if( !ostr )
            throw runtime_error(
                "Nije uspelo pisanje u datoteku!"
                );
        }

    void citaj( istream& istr )
        {
        istr.seekg( 0, ios::end );
        unsigned v = istr.tellg();
        istr.seekg( 0, ios::beg );
        _niz.resize( v );
        istr.read( _niz.begin(), v );
        if( !istr )
            throw runtime_error(
                "Nije uspelo citanje datoteke!"
                );
        }

private:
    vector<Bajt> _niz;
};
```

U opisu standardne biblioteke programskog jezika C++ stoji da implementacija kolekcije `vector` mora obezbediti da svi elementi vektora zauzimaju kontinualan uzastopni prostor u memoriji, tj. da je u svakom trenutku sadržaj vektora upravo jedan *običan* niz elemenata. Dalje, iteratori nad vektorima se implementiraju kao pokazivači na elemente tog običnog niza. Tako je pokazivač na prvi element niza isto što i iterator koji ukazuje na prvi element niza, a koji se dobija pomoću metoda `begin`.

Znajući to, možemo (sasvim ispravno) zaključiti da se elementi vektora `v` tipa `vector<T>` nalaze u memoriji kao niz koji počinje na adresi `v.begin()` i ima veličinu `v.size() * sizeof(T)` bajtova. Zbog toga čitanje i pisanje možemo izvoditi u jednom koraku, bez potrebe da čitamo ili pišemo bajt po bajt. Kako mi koristimo nizove bajtova, to je veličina niza u bajtovima jednaka veličini niza u broju elemenata.

Na eventualne greške reagujemo izbacivanjem izuzetaka.

Sada možemo da dovršimo metode za čitanje i pisanje klase `ProgramKoDek`:

```
class ProgramKoDek
{
...
private:
    void citanjeDatoteke()
        {
        ifstream uDat( imeUlazneDatoteke.c_str(), ios::binary );
        if( !uDat )
            throw runtime_error(
                "Nije uspelo otvaranje ulazne datoteke!"
                );
        originalniNizBajtova.citaj( uDat );
        }
    void pisanjeDatoteke()
        {
        ofstream iDat( imeIzlazneDatoteke.c_str(), ios::binary );
        if( !iDat )
            throw runtime_error(
                "Nije uspelo otvaranje izlazne datoteke!"
                );
        transformisaniNizBajtova.pisi( iDat );
        }
...
    //-------------------------------------------------------------
    //  Članovi podaci
    //-------------------------------------------------------------
    bool kodiranje;
    string imeUlazneDatoteke;
    string imeIzlazneDatoteke;
    NizBajtova originalniNizBajtova;
    NizBajtova transformisaniNizBajtova;
};
```

Korak 3 - Translacija

Da bismo postepeno izgrađivali nagoveštenu hijerarhiju klasa transformacija kodiranja, za sada ćemo pretpostaviti da je potrebno da naš program primenjuje samo jednu vrstu transformacija. Kao primer biramo translaciju:

```
class Translacija
{
public:
    Translacija( Bajt kod )
        : _kod(kod)
        {}

    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = org[i] + _kod;
        }

    void dekodiranje(const NizBajtova& org, NizBajtova& rez )const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = org[i] - _kod;
        }

private:
    Bajt _kod;
};
```

Kao što je definisano u postavci zadatka, translacija predstavlja transformaciju koja pri kodiranju povećava, a pri dekodiranju smanjuje vrednosti pojedinačnih bajtova za datu vrednost. Koristimo osobinu programskog jezika da prekoračenje pri operacijama sabiranja i oduzimanja ne predstavlja grešku, već se izvodi po modulu 2^N, gde je N broj bitova podatka. U našem slučaju, po modulu 256. Na primer, sabiranje brojeva 250 i 100 daje rezultat 94.

Vidimo da je za predstavljenu implementaciju potrebno da niz bajtova ima metode za promenu veličine i neposredno pristupanje elementima niza. U skladu sa time izmenićemo klasu `NizBajtova`:

```
class NizBajtova
{
public:
    Bajt operator[]( unsigned i ) const
        { return _niz[i]; }

    Bajt& operator[]( unsigned i )
        { return _niz[i]; }

    void promeniVelicinu( unsigned v )
        { _niz.resize(v); }
...
};
```

Implementirane su dve verzije operatora indeksiranja: prva verzija omogućava čitanje elemenata konstantnog niza, dok se druga odnosi na nekonstantne nizove i omogućava kako čitanje tako i menjanje elemenata niza. Ponovo ističemo da bi eventualna implementacija samo jedne verzije operatora:

```
Bajt& operator[]( unsigned i ) const
    { ... }
```

predstavljala ozbiljnu grešku u oblikovanju interfejsa klase, jer se tipom operatora korisniku sugeriše da on *ne menja* niz, a on to ipak posredno čini, jer omogućava *menjanje elemenata niza*:

```
const NizBajtova& niz = ...;
niz[i] = ...;
```

Sada možemo implementirati program tako da uvek primenjuje translaciju za, na primer, 17:

```
class ProgramKoDek
{
...
    void izvodjenjeOperacije()
        {
        Translacija t(17);
        if( kodiranje )
            t.kodiranje(
                originalniNizBajtova,
                transformisaniNizBajtova
                );
        else
            t.dekodiranje(
                originalniNizBajtova,
                transformisaniNizBajtova
                );
        }
...
}
```

Najzad možemo da proverimo kako radi kodiranje. Kodirajmo tekst programa:

```
kodiranje k kodiranje.cpp a.txt a
```

Poslednji parametar je neophodan zbog zahtevanog broja parametara, iako ga za sada ne koristimo. Čitanjem datoteke `a.txt` možemo uvideti da je nešto urađeno, ali ne i proveriti da li je sve u redu. Jedna (ali ne i dovoljna) provera je pokušaj dekodiranja:

```
kodiranje d a.txt b.txt a
```

Sadržaj datoteke `b.txt` bi morao da bude identičan sadržaju datoteke `kodiranje.cpp`.

Korak 4 - Rotacija

Nakon što smo implementirali translaciju, sada je red na rotaciju. Implementiraćemo je po uzoru na translaciju:

```
class Rotacija
{
```

```
public:
    Rotacija( Bajt kod )
        : _kod( kod % 8 )
        {}
    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = (org[i] << _kod) | (org[i] >> (8-_kod));
        }
    void dekodiranje(const NizBajtova& org, NizBajtova& rez )const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = (org[i] >> _kod) | (org[i] << (8-_kod));
        }
private:
    Bajt _kod;
};
```

Jednostavno, zar ne? Pa, ako posmatramo samo klasu `Rotacija`, to bi se još i moglo reći, ali ako posmatramo i klasu `Translacija`, postaje sasvim očigledno da nešto „zaudara“. Sada je pravi trenutak da naš program „presvučemo“.

Funkcionisanje klase `Rotacija` možemo isprobati menjanjem metoda programa za izvođenje operacije:

```
class ProgramKoDek
{
...
    void izvodjenjeOperacije()
        {
        Rotacija t(2);
        ...
        }
...
}
```

Korak 5 - Refaktorisanje

Klase `Translacija` i `Rotacija` su isuviše slične da bi takvo ponavljanje koda smelo da se toleriše. Čak i kada ne bismo od ranije želeli da od ovih klasa napravimo jednu hijerarhiju klasa, sada bismo o tome morali razmišljati. Štaviše, i u samim klasama su implementacije kodiranja i dekodiranja toliko slične da bi trebalo razmišljati kako da se izmeni struktura ovih metoda da bi se to ponavljanje izbeglo.

Za početak, primetimo da naše klase imaju isti interfejs. Izdvajanjem interfejsa u baznu klasu `Transformacija` i definisanjem klasa `Translacija` i `Rotacija` kao specijalizacija bazne klase dobijamo jednostavnu hijerarhiju klasa:

```
//---------------------------------------------------------------
//  Transformacija
//---------------------------------------------------------------
class Transformacija
{
public:
    virtual ~Transformacija()
        {}
    virtual void kodiranje(
                    const NizBajtova& org, NizBajtova& rez
                    ) const = 0;
    virtual void dekodiranje(
                    const NizBajtova& org, NizBajtova& rez
                    )const = 0;
};
//---------------------------------------------------------------
//  Translacija
//---------------------------------------------------------------
class Translacija : public Transformacija
{
...
};
//---------------------------------------------------------------
//  Rotacija
//---------------------------------------------------------------
class Rotacija : public Transformacija
{
...
};
```

Načinjen korak se naziva *izdvajanje interfejsa*. Primenjuje se kada se primeti da neke klase imaju slično ili potpuno isto javno ponašanje (interfejse) i kada se uočena sličnost želi formalizovati građenjem hijerarhije klasa. Predstavlja jedan od najjednostavnijih i najčešće primenjivanih načina da se započne izgradnja hijerarhije klasa. Izvodi se pravljenjem apstraktne klase kao generalizacije posmatranih klasa.

Primetimo da sličnosti klasa `Translacija` i `Rotacija` nisu ograničene na interfejs. Postupak kodiranja i dekodiranja je skoro potpuno isti. Kada u jednoj klasi, ili u više klasa jedne hijerarhije klasa imamo više metoda koji imaju sličnu implementaciju, tada zajedničke delove možemo izdvojiti u nove metode. Taj postupak se naziva *izdvajanje metoda*. Iz metoda `kodiranje` i `dekodiranje` klasa `Translacija` i `Rotacija` možemo, primenom izdvajanja metoda, izmestiti u nove metode izraze koji kodiraju pojedinačne bajtove:

```
//---------------------------------------------------------------
//  Translacija
//---------------------------------------------------------------
class Translacija : public Transformacija
{
public:
    Translacija( Bajt kod )
        : _kod(kod)
        {}
```

```
    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = kodiranjeBajta( org[i] );
        }

    void dekodiranje(const NizBajtova& org, NizBajtova& rez )const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = dekodiranjeBajta( org[i] );
        }

    Bajt kodiranjeBajta( Bajt b ) const
        { return b + _kod; }
    Bajt dekodiranjeBajta( Bajt b ) const
        { return b - _kod; }

private:
    Bajt _kod;
};

//-----------------------------------------------------------------
//  Rotacija
//-----------------------------------------------------------------
class Rotacija : public Transformacija
{
public:
    Rotacija( Bajt kod )
        : _kod( kod % 8 )
        {}

    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = kodiranjeBajta( org[i] );
        }

    void dekodiranje(const NizBajtova& org, NizBajtova& rez )const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = dekodiranjeBajta( org[i] );
        }

    Bajt kodiranjeBajta( Bajt b ) const
        { return (b << _kod) | (b >> (8 - _kod)); }
    Bajt dekodiranjeBajta( Bajt b ) const
        { return (b >> _kod) | (b << (8 - _kod)); }

private:
    Bajt _kod;
};
```

Sada su implementacije metoda za kodiranje i dekodiranje nizova u klasama `Translacija` i `Rotacija` potpuno iste. Logičan potez je *podizanje ponašanja uz hijerarhiju* premeštanjem implementacije ovih metoda u klasu `Transformacija`:

```
class Transformacija
{
public:
    virtual ~Transformacija()
        {}
    virtual void kodiranje(
                    const NizBajtova& org, NizBajtova& rez
                    ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = kodiranjeBajta( org[i] );
        }
    virtual void dekodiranje(
                    const NizBajtova& org, NizBajtova& rez
                    ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = dekodiranjeBajta( org[i] );
        }
protected:
    virtual Bajt kodiranjeBajta( Bajt b ) const = 0;
    virtual Bajt dekodiranjeBajta( Bajt b ) const = 0;
};
```

Primetimo da smo umesto ovakve izmene mogli da izvedemo i nešto drugačije izdvajanje metoda, ograničeno na pojedinačne klase. Naime, u klasi `Translacija` (isto važi i za klasu `Rotacija`) metod `dekodiranje` je veoma sličan metodu `kodiranje`, čak toliko da se može implementirati kao kodiranje sa drugim parametrom. Izdvajanjem zajedničkog dela u statički metod dobilo bi se nešto poput:

```
class Translacija : public Transformacija
{
public:
    Translacija( Bajt kod )
        : _kod(kod)
        {}
    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        { transliranje( org, rez, _kod ); }
    void dekodiranje(const NizBajtova& org, NizBajtova& rez )const
        { transliranje( org, rez, -_kod ); }
```

```
private:
    static void transliranje(
                    const NizBajtova& org, NizBajtova& rez,
                    Bajt kod
                    )const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = org[i] + kod;
        }

    Bajt _kod;
};
```

Kako metod `Transliranje` više ne koristi članove podatke klase, možemo ga proglasiti za statički metod.

Ako imamo na umu da se poslednja opisana promena izvodi lokalno, samo u okviru jedne klase, postavlja se pitanje da li je takvo izdvajanje metoda možda bolje nego primenjenog izdvajanja metoda i podizanje ponašanja uz hijerarhiju? U najvećem broju slučajeva podizanje ponašanja uz hijerarhiju je bolji izbor, jer se na taj način obuhvata ponašanje koje je zajedničko za veći broj klasa, međutim, nekada je horizontalno izdvajanje metoda prihvatljivije. To zavisi, pre svega, od složenosti metoda koji se tako pojednostavljuju. U ovom slučaju ostajemo pri opisanom podizanju ponašanja uz hijerarhiju. Čitaocima ostavljamo da pokušaju da implementiraju drugi način izdvajanja metoda i da uporede složenost i performanse dva rešenja.

Da li možemo izvesti i prethodno izvedeno podizanje ponašanja uz hijerarhiju i ovakvo izdvajanje metoda? U načelu da, ali praksa pokazuje da istovremeno podizanje metoda uz hijerarhiju (tzv. vertikalno pomeranje ponašanja) i izdvajanje metoda u okviru jedne klase (tzv. horizontalno pomeranje ponašanja) mogu da dovedu do povećavanja složenosti ukoliko se primenjuju na iste metode. Na konkretnom primeru, da bismo mogli napraviti statički metod `transliranje`, iskoristili smo činjenicu da se dekodiranje može implementirati kao kodiranje kodom `-_kod`. U slučaju rotacije, dekodiranje se dobija kao kodiranje kodom 8 - `_kod`. Ako na sličan način u klasi `Rotacija` napravimo statički metod `rotiranje`, tada bismo zajedničke delove metoda `transliranje` i `rotiranje` (koji tada više ne bi mogli da budu statički) mogli da izdvojimo i pomerimo uz hijerarhiju, uz uvođenje novih virtualnih metoda nalik na prethodno opisan metod `kodiranjeBajta`. Tako bismo na kraju u svakoj od izvedenih klasa imali po tri implementacije virtualnih metoda, a u baznoj klasi jedan zajednički metod koji izvodi transformaciju.

Sada se čini da bismo mogli da i podatak `_kod` podignemo u klasu `Transformacija`:

```
//-------------------------------------------------------------
//  Transformacija
//-------------------------------------------------------------
class Transformacija
{
```

```
public:
    Transformacija( Bajt kod )
        : _kod(kod)
        {}

    virtual ~Transformacija()
        {}

    virtual void kodiranje(
                    const NizBajtova& org, NizBajtova& rez
                    ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = kodiranjeBajta( org[i] );
        }

    virtual void dekodiranje(
                    const NizBajtova& org, NizBajtova& rez
                    ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = dekodiranjeBajta( org[i] );
        }

protected:
    virtual Bajt kodiranjeBajta( Bajt b ) const = 0;
    virtual Bajt dekodiranjeBajta( Bajt b ) const = 0;

    Bajt _kod;
};

//-------------------------------------------------------------------
//  Translacija
//-------------------------------------------------------------------
class Translacija : public Transformacija
{
public:
    Translacija( Bajt kod )
        : Transformacija(kod)
        {}

protected:
    Bajt kodiranjeBajta( Bajt b ) const
        { return b + _kod; }
    Bajt dekodiranjeBajta( Bajt b ) const
        { return b - _kod; }
};

//-------------------------------------------------------------------
//  Rotacija
//-------------------------------------------------------------------
class Rotacija : public Transformacija
{
```

```
public:
    Rotacija( Bajt kod )
        : Transformacija(kod)
        {}
protected:
    Bajt kodiranjeBajta( Bajt b ) const
        { return (b << _kod) | (b >> (8 - _kod)); }
    Bajt dekodiranjeBajta( Bajt b ) const
        { return (b >> _kod) | (b << (8 - _kod)); }
};
```

Primenjene transformacije koda programa označavaju se terminom *refaktorisanje*. Refaktorisanje predstavlja skup principa i postupaka koji prevode tekst programa iz jednog oblika u drugi, ne menjajući mu funkcionalnost, ali unapređujući njegovu strukturu. Postupci refaktorisanja *popravljaju* program tako da bude *lepši* i spremniji za iskušenja dograđivanja i menjanja. Refaktorisanje se preduzima kako nakon menjanja delova programa ili dodavanja novih programskih elemenata, tako i kao vid pripreme za preduzimanje izmena.

Refaktorisanje se obično izvodi u formi niza jednostavnih izmena, ali iako su po svojoj prirodi jednostavne, te izmene mogu biti prilično zahtevne i prostirati se kroz veliki broj metoda ili klasa. Zbog toga je jedan od najvažnijih principa refaktorisanja upravo sveobuhvatno testiranje funkcionalnosti delova programa kako pre tako i posle svakog preduzetog postupka refaktorisanja.

Principi refaktorisanja i veći broj najvažnijih postupaka opisani su u [Fowler 1999].

Korak 6 - Zamena

Pri implementaciji transformacije `Zamena` suočavamo se sa neprijatnim problemom: pri konstrukciji moramo kosristiti konstruktor bazne klase, ali nemamo na raspolaganju nikakav smisleni parametar koji bismo prosledili kao argument baznog konstruktora. Problem sa kojim se ovde suočavamo je *suviše konkretna* (tj. nedovoljno apstraktna) bazna klasa. Ako se vratimo na prethodni odeljak uočićemo da smo *preterali* sa refaktorisanjem premeštajući u baznu klasu elemente koji jesu zajednički za do tada napisane klase, ali ne i za one koje ćemo tek napisati.

Kao pouku možemo definisati pravilo kojeg bi trebalo da se pridržavamo da ne bismo ponovili istu grešku. *Ne sme se žuriti sa podizanjem ponašanja u baznu klasu – takvo podizanje valja odlagati dok se ne sagledaju okvirne granice hijerarhije koja se gradi, jer se često pokazuje da je bolje rešenje umetanje nove klase*.

Nešto šire tumačenje ovog pravila bi bilo da ne treba žuriti sa podizanjem ponašanja uz hijrarhiju dok se ne sagledaju moguće posledice. Još opštije pravilo je da se *pre započinjanja refaktorisanja moraju sagledati moguće posledice.*

Klasa `Zamena` ne može imati nikakve koristi od svih konkretnih elemenata koji su podignuti u klasu `Transformacija`. Nije potreban član podatak `_kod`, a implementacije kodiranja i dekodiranja su skoro potpuno neupotrebljive. Da bi klasa `Zamena` mogla naslediti klasu `Transformacija` neophodno je izvesti *izdvajanje klase* i *spuštanje ponašanja niz hijerarhiju*. Izdvojićemo sve konkretne elemente iz klase `Transformacija` u njenu novu naslednicu

`TransformacijaBajtova`, koja pretpostavlja da se kodiranje i dekodiranje izvode nezavisno za svaki bajt i da se pri tome upotrebljava neki zadati parametar:

```
//---------------------------------------------------------------
//  Transformacija
//---------------------------------------------------------------
class Transformacija
{
public:
    virtual ~Transformacija()
        {}
    virtual void kodiranje(
                    const NizBajtova& org, NizBajtova& rez
                    ) const = 0;
    virtual void dekodiranje(
                    const NizBajtova& org, NizBajtova& rez
                    ) const = 0;
};
//---------------------------------------------------------------
//  TransformacijaBajtova
//---------------------------------------------------------------
class TransformacijaBajtova : public Transformacija
{
public:
    TransformacijaBajtova( Bajt kod )
        : _kod(kod)
        {}

    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = kodiranjeBajta( org[i] );
        }

    void dekodiranje(const NizBajtova& org, NizBajtova& rez) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = dekodiranjeBajta( org[i] );
        }

protected:
    virtual Bajt kodiranjeBajta( Bajt b ) const = 0;
    virtual Bajt dekodiranjeBajta( Bajt b ) const = 0;

    Bajt _kod;
};
```

Sasvim je očigledno da će `Translacija` i `Rotacija` sada nasleđivati novu klasu:

```
class Translacija : public TransformacijaBajtova
{
```

```
public:
    Translacija( Bajt kod )
        : TransformacijaBajtova(kod)
        {}
...
};

class Rotacija : public TransformacijaBajtova
{
public:
    Rotacija( Bajt kod )
        : TransformacijaBajtova(kod)
        {}
...
};
```

Platili smo danak neiskustvu tako što smo deo programa morali vratiti u prethodno stanje. No, valjda će nam to biti dobar nauk za ubuduće. Sada možemo napisati klasu `Zamena`. Kako se radi o transformaciji koja je sama sebi inverzna, i kodiranje i dekodiranje ćemo implementirati pozivanjem privatnog metoda `transformisanje`:

```
class Zamena : public Transformacija
{
public:
    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        { transformisanje( org, rez ); }
    void dekodiranje(const NizBajtova& org, NizBajtova& rez) const
        { transformisanje( org, rez ); }

private:
    static void transformisanje(
            const NizBajtova& org, NizBajtova& rez
            )
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=1; i<vel; i+=2 ){
            rez[i] = org[i-1];
            rez[i-1] = org[i];
            }
        if( vel % 2 )
            rez[vel-1] = org[vel-1];
        }
};
```

Isprobaćemo novu transformaciju:

```
class ProgramKoDek
{
...
    void izvodjenjeOperacije()
        {
        Zamena t;
        ...
        }
...
}
```

Korak 7 - Ekskluzivna disjunkcija

Za izvođenje ekskluzivne disjunkcije potreban nam je dodatni niz bajtova kojim izvodimo kodiranje. Po svemu ostalom ova transformacija se implementira slično ostalim transformacijama:

```
class EkskluzivnaDisjunkcija : public Transformacija
{
public:
    EkskluzivnaDisjunkcija( const NizBajtova& kod )
        : _kod( kod )
        {}

    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        { transformisanje( org, rez ); }
    void dekodiranje(const NizBajtova& org, NizBajtova& rez) const
        { transformisanje( org, rez ); }

private:
    void transformisanje(
            const NizBajtova& org, NizBajtova& rez
            ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        unsigned n = _kod.velicina();
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = org[i] ^ _kod[i%n];
        }

    NizBajtova _kod;
};
```

Da bismo na jednostavan način napravili niz bajtova koji predstavlja parametar kodiranja, dopunićemo klasu `NizBajtova` konstruktorom na osnovu date niske. Nakon dodavanja tog konstruktora neophodno je eksplicitno definisati i konstruktor bez argumenata:

```
class NizBajtova
{
public:
    NizBajtova()
        {}

    NizBajtova( const string& s )
        {
        unsigned v = s.length();
        _niz.resize( v );
        for( unsigned i=0; i<v; i++ )
            _niz[i] = s[i];
        }
...
};
```

Sada možemo napisati i primer upotrebe nove transformacije i proveriti da li sve radi kako bi trebalo:

```
class ProgramKoDek
{
...
    void izvodjenjeOperacije()
        {
        EkskluzivnaDisjunkcija t( NizBajtova("abc") );
        ...
        }
...
}
```

Primetimo da može biti fatalnih problema u slučaju da je kodirajuća niska prazna. takve probleme ćemo predupredi proverom pri konstrukciji:

```
class EkskluzivnaDisjunkcija : public Transformacija
{
public:
    EkskluzivnaDisjunkcija( const NizBajtova& kod )
        : _kod( kod )
        {
        if( !kod.velicina() )
            throw range_error(
                "Duzina kodirajuceg niza mora biti veca od 0 "
                "(EkskluzivnaDisjunkcija)!"
                );
        }
...
};
```

Korak 8 - Ekskluzivna dijsunkcija početnim nizom date dužine

Po uzoru na predstavljenu transformaciju `EkskluzivnaDisjunkcija` pišemo klasu `EkskluzivnaDisjunkcijaStart`. Kako se u ovom slučaju kodiranje obavlja nekim početnim podnizom ulaznog niza, nije nam neophodan član podatak `_kod`, već samo `_duzinaKoda`. Kao i u prethodnom slučaju, proveravamo da nije slučajno zadat prazan kodirajući podniz:

```
class EkskluzivnaDisjunkcijaStart : public Transformacija
{
public:
    EkskluzivnaDisjunkcijaStart( unsigned duzinaKoda )
        : _duzinaKoda( duzinaKoda )
        {
        if( !duzinaKoda )
            throw range_error(
                "Duzina kodirajuceg podniza mora biti veca od 0 "
                "(EkskluzivnaDisjunkcijaStart)!"
                );
        }

    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        { transformisanje( org, rez ); }

    void dekodiranje(const NizBajtova& org, NizBajtova& rez) const
        { transformisanje( org, rez ); }
```

```
private:
    void transformisanje(
            const NizBajtova& org, NizBajtova& rez
            ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<_duzinaKoda; i++ )
            rez[i] = org[i];
        for( unsigned int i=_duzinaKoda; i<vel; i++ )
            rez[i] = org[i] ^ org[i%_duzinaKoda];
        }

    unsigned _duzinaKoda;
};
```

Poređenje klasa `EkskluzivnaDisjunkcija` i `EkskluzivnaDisjunkcijaStart` nas postavlja pred pitanje da li u ove dve klase ima dovoljno ponavljanja da bismo izvodili neko refaktorisanje ili se postojeća ponavljanja mogu dopustiti? Ako bismo iz ulaznog niza pokušali da izdvojimo početni podniz bajtova, taj podniz bi se odnosio samo na jedno konkretno izvršavanje transformacije a ne na čitav objekat, zbog čega se tako izdvojen podniz ne bi smeo sačuvati kao trajni atribut transformacije. Drugi deo problema predstavlja činjenica da se početni podniz ne kodira već se samo prepisuje. Ispada da su u ovom slučaju postojeća ponavljanja prilično umotana u razlike. Te razlike su prilične, pa bi lako *uprljale* eventualno izdvojene metode u dovoljnoj meri da se efekti *ulepšavanja* izgube. Zbog toga ćemo program ostaviti u postojećem obliku, a čitaocu preporučujemo da za vežbu pokuša da otkloni ponavljanja izdvajanjem metoda i definisanjem klase `EkskluzivnaDisjunkcijaStart` kao naslednice klase `EkskluzivnaDisjunkcija`.

Ostaje nam da proverimo da li nova transformacija ispravno funkcioniše:

```
class ProgramKoDek
{
...
    void izvodjenjeOperacije()
        {
        EkskluzivnaDisjunkcijaStart t( 15 );
        ...
        }
...
}
```

Korak 9 - Kompozicija

Napisali smo pet elementarnih transformacija kodiranja i dekodiranja. Preostaje nam da definišemo složenu transformaciju kao kompoziciju jednostavnijih transformacija. Pri rešavanju problema kompozicije suočićemo se sa nekoliko složenijih problema koje ćemo postepeno rešavati.

Kolekcije objekata hijerarhije klasa

Kompozicija dveju ili više transformacija mora raspolagati kolekcijom transformacija čiju kompoziciju predstavlja. Kako se transformacije primenjuju u definisanom redosledu, izbor

tipa kolekcije se svodi na vektor ili listu. Ovom prilikom ćemo upotrebiti vektor, ali se bez veće razlike može koristiti i lista. Značajna karakteristika kolekcije transformacija je da ona može sadržati različite transformacije. Štaviše, moguće je da će se u okviru kompozicije, pored već napisanih elementarnih transformacija naći i druge kompozicije transformacija, kao i neke transformacije koje u ovom trenutku još nisu definisane. To neposredno ukazuje na potrebu da naša kolekcija funkcioniše uz primenu polimorfizma, tj. dinamičkog i implicitnog raspoznavanja tipova transformacija. Kako programski jezik C++ podržava polimorfizam samo posredstvom pokazivača i referenci, jasno je da ćemo u našoj kolekciji morati da čuvamo pokazivače ili reference na transformacije. Zbog toga što se reference moraju inicijalizovati u trenutku definisanja i kasnije se ne mogu menjati, one su ovde neupotrebljive, pa nam kao jedino rešenje preostaje upotreba pokazivača.

Navedeni pristup nije specifičnost ovog zadatka. Priroda programskog jezika C++ uslovljava da *praktično uvek* kada pravimo kolekcije objekata neke hijerarhije klasa, moramo raditi sa kolekcijama *pokazivača na baznu klasu hijerarhije*. U konkretnom slučaju, odgovarajuća kolekcija će imati tip:

```
vector<const Transformacija*>
```

Najveći problem koji takav tip kolekcije donosi programeru odnosi se na staranje o dinamičkim objektima koji predstavljaju elemente niza. U najvećem broju slučajeva biće neophodno eksplicitno definisanje destruktora, konstruktora kopije i operatora dodeljivanja. To će moći da se izbegne isključivo pod uslovom da elementi niza predstavljaju pokazivače na objekte o kojima se stara (kopira ih, pravi i uklanja) neka druga kolekcija. Najpre ćemo napisati kompoziciju transformacija bez odgovarajućih metoda, da bi smo im kasnije posvetili punu pažnju.

Klasa `SlozenaTransformacija` je specijalizacija klase `Transformacija`, koja sadrži kolekciju transformacija:

```
class SlozenaTransformacija : public Transformacija
{
public:
    void dodaj( const Transformacija* t )
        { _transformacije.push_back( t ); }
    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        { ... }
    void dekodiranje(const NizBajtova& org, NizBajtova& rez) const
        { ... }

private:
    vector<const Transformacija*> _transformacije;
};
```

Implementiran metod `dodaj` dodaje transformaciju na kraj sekvence transformacija. Pretpostavlja se da od tog trenutka staranje o dodatoj transformaciji preuzima složena transformacija.

Kodiranje i dekodiranje

Operacije kodiranja i dekodiranja se u kompoziciji više transformacija izvode redom. Ponudićemo jednostavno rešenje postupka kodiranja koje nije efikasno, ali je lako razumljivo:

```
class SlozenaTransformacija : public Transformacija
{
...
    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        {
        rez = org;
        for( unsigned i=0; i<_transformacije.size(); i++ ){
            NizBajtova p = rez;
            _transformacije[i]->kodiranje( p, rez );
            }
        }
...
};
```

Napisan metod obezbeđuje da:

- na početku svake iteracije niz bajtova `rez` sadrži rezultat primene prvih `i` transformacija (smatramo da je primena praznog niza transformacija identično preslikavanje);
- na kraju svake iteracije niz bajtova `rez` sadrži rezultat primene prvih `i`+1 transformacija.

Ne bi smelo da nam promakne da je redosled primene transformacija pri kodiranju i dekodiranju različit. Zbog toga dekodiranje izvodimo na sledeći način:

```
class SlozenaTransformacija : public Transformacija
{
...
    void dekodiranje(const NizBajtova& org, NizBajtova& rez) const
        {
        rez = org;
        unsigned n = _transformacije.size();
        for( unsigned i=0; i<n; i++ ){
            NizBajtova p = rez;
            _transformacije[n-1-i]->dekodiranje( p, rez );
            }
        }
...
};
```

Kao što smo najavili, dobra strana ovakvog pristupa je jednostavnost i razumljivost. Loša strana je neefikasnost. Problem je u tome što se u svakoj iteraciji pravi kopija trenutnog rezultata da bi se moglo nastaviti sa postupkom. To se može izbeći pažljivom upotrebom pomoćnih nizova. Optimizaciju prepuštamo čitaocima kao dobru vežbu.

Napisani kod ćemo proveriti na sledeći način:

```
class ProgramKoDek
{
```

```
...
    void izvodjenjeOperacije()
        {
        SlozenaTransformacija t;
        t.dodaj( new Translacija(17) );
        t.dodaj( new Zamena );
        ...
        }
...
}
```

Staranje o dinamičkim objektima

Staranje o dinamičkim objektima podrazumeva pisanje odgovarajućih destruktora, konstruktora kopije i operatora dodeljivanja. Upotrebićemo oblik definisanja ovih metoda, koji je opisan u primeru *2 - Lista*, na strani 44:

```
class SlozenaTransformacija : public Transformacija
{
public:
    SlozenaTransformacija()
        {}

    SlozenaTransformacija( const SlozenaTransformacija& s )
        { init(s); }

    ~SlozenaTransformacija()
        { deinit(); }

    SlozenaTransformacija& operator = (
            const SlozenaTransformacija& s
            )
        {
        if( this != &s ){
            deinit();
            init(s);
            }
        return *this;
        }
...
private:
    void deinit()
        { ... }
    void init( const SlozenaTransformacija& s )
        { ... }
...
};
```

Zbog eksplicitnog definisanja konstruktora kopije neophodno je da eksplicitno definišemo i konstruktor bez argumenata.

Metod za deinicijalizaciju je u ovom slučaju sasvim jednostavno napisati. Dovoljno je ukloniti sve elemente kolekcije. Za svaki konkretan element će, pre oslobađanja zauzete memorije, biti pozvan odgovarajući destruktor, jer u klasi `Transformacija` imamo definisan virtualan destruktor:

```
class SlozenaTransformacija : public Transformacija
{
...
    void deinit()
        {
        for( unsigned i=0; i<_transformacije.size(); i++ )
            delete _transformacije[i];
        }
...
};
```

Ako bismo pokušali da metod za inicijalizaciju kopije napišemo „na brzinu“, mogli bismo dobiti sledeći rezultat:

```
class SlozenaTransformacija : public Transformacija
{
...
    void init( const SlozenaTransformacija& s )
        {
        _transformacije.clear();
        for( unsigned i=0; i<s._transformacije.size(); i++ )
            _transformacije.push_back( s._transformacije[i] );
        }
...
};
```

Zašto ovo nije u redu? Najpre primetimo da se ovakva inicijalizacija praktično uopšte ne razlikuje od podrazumevane implementacije konstruktora kopije – naša nova kolekcija će sadržati pokazivače na iste objekte transformacija koji su sadržani u originalnoj koleciji. Posledica, koja obično vodi fatalnim ishodima, je da će obe kolekcije smatrati da raspolažu istim objektima, pa će u nekom trenutku jedna kolekcija ukloniti te objekte, a nakon toga će druga pokušati da ih upotrebljava. Apsolutno je neophodno da u novu kolekciju ne stavljamo pokazivače na objekte koji su u originalnoj kolekciji, već pokazivače na *njihove kopije*.

Obratimo pažnju na jedan veoma čest, a potencijalno izuzetno značajan propust u definisanju ovakvih klasa: pisac programa bi mogao da primeti da se u njegovom programu nigde ne upotrebljava kopiranje složenih transformacija i da, u skladu sa time, odluči da ne piše konstruktor kopije i operator dodeljivanja. Kakve su eventualne posledice?

Razmotrimo prvo mogućnost da je procena bila pogrešna i da se negde u programu ipak implicitno izvodi kopiranje složenih transformacija. Tada bi najčešće neki tok izvršavanja prošao bez problema, dok bi neki drugi doveo do fatalnih grešaka i prekida rada programa. Takve greške, koje se nekada pojave a nekada ne, spadaju u red grešaka koje se najteže prepoznaju i ispravljaju. „Dobra“ strana takvih problema je da obično predstavljaju mnogo uverljivije učitelje nego što su to knjige i dobronamerni saveti.

Ukoliko je procena bila dobra, tada neće biti *kratkoročnih* posledica. Međutim, ako se ima na umu da se programi obično ne pišu za jednokratnu upotrebu, postavlja se pitanje šta će se desiti kada isti programer posle izvesnog vremena (ili neki drugi programer u bilo kom trenutku) pokuša da izmeni program ili prenese neke klase u drugi program? Ako ima poverenje u originalni program, smatraće da su klase ispravno definisane i skoro je izvesno da

će ga u nekom trenutku posledice neimplementiranih metoda sačekati *kao grom iz vedra neba*. A tada je cena obično mnogo veća nego što je to u slučaju kratkoročnih problema, jer u ovom slučaju programer ne zna (zbog toga što je vremenom zaboravio, ili zato što nikada nije ni znao) kako elementi programa funkcionišu, pa je potrebno da značajno radno vreme posveti detaljnom izučavanju kako bi eventualno otkrio i otklonio uzrok svojih problema. U najgorem slučaju, ako originalni kod nije *lepo i čitko* napisan, ili nedostaje neophodna dokumentacija, lako se može desiti da programer odustane od traženja grešaka i odluči se da sve potrebne klase ponovo implementira. Valjda je jasno kakav je tek to gubitak vremena.

Zaključak bi trebalo da bude više nego očigledan – *uvek* se isplati na vreme napisati metode za staranje o dinamičkim aspektima klasa. Svako odlaganje samo uvećava cenu.

Dinamički konstruktor kopije

Kako iskopirati transformaciju?

Odgovor nije trivijalan. Problem je u tome što moramo napraviti novi objekat transformacije koji će biti istog tipa kao objekat koji kopiramo, pri čemu ne znamo unapred koji je to tip. Sve što znamo je da imamo pokazivač na neki objekat neke klase hijerarhije transformacija. U primeru *7 - Enciklopedija*, na strani 223, pokazali smo da ne postoji nešto poput dinamičkog konstruktora. To, naravno, stoji i kada su u pitanju konstruktori kopije. Jedino sredstvo za dinamičko prilagođavanje ponašanja tipovima, koje imamo na raspolaganju, jesu virtualni metodi. Odatle sledi da ispravnu kopiju možemo dobiti samo pozivanjem odgovarajućeg virtualnog metoda datog objekta transformacije, koji će je sam napraviti.

Koncept osposobljavanja objekata naših hijerarhija da prave sopstvene kopije naziva se *kloniranje*. Kloniranje predstavlja tehniku koja je neophodna u skoro svakoj netrivijalnoj hijerarhiji klasa. Implementira se tako što u svakoj konkretnoj klasi obezbeđujemo po implementaciju virtualnog metoda `kopija`, koji pravi sopstvenu kopiju odgovarajućeg tipa. Implementacija ovog metoda je obično trivijalna i svodi se na pozivanje odgovarajućeg konstruktora ili konstruktora kopije.

U baznoj klasi hijerarhije definišemo apstraktan metod:

```
class Transformacija
{
public:
...
    virtual Transformacija* kopija() const = 0;
...
};
```

Klasa `TransformacijaBajtova` je apstraktna (tj. nije konkretna) pa u njoj nije potrebno pisati metod `kopija`. U ostalim klasama pišemo implementacije ovog metoda koristeći neki od raspoloživih konstruktora. Obično je najjednostavnije koristiti konstruktor kopije, ali je zbog performansi nekada bolje pribegavati drugim konstruktorima. U našem slučaju ne postoje značajne razlike u performansama, ali su radi ilustracije primenjeni različiti konstruktori.

U klasi `Translacija` je upotrebljen konstruktor sa jednim argumentom:

```
class Translacija : public TransformacijaBajtova
{
public:
...
    Transformacija* kopija() const
        { return new Translacija(_kod); }
...
};
```

U klasi `Rotacija` je upotrebljen konstruktor kopije:

```
class Rotacija : public TransformacijaBajtova
{
public:
...
    Transformacija* kopija() const
        { return new Rotacija(*this); }
...
};
```

U klasi `Zamena` je upotrebljen podrazumevani konstruktor:

```
class Zamena : public Transformacija
{
public:
...
    Transformacija* kopija() const
        { return new Zamena(); }
...
};
```

U ostalim klasama je primenjen konstruktor kopije:

```
class EkskluzivnaDisjunkcija : public Transformacija
{
public:
...
    Transformacija* kopija() const
        { return new EkskluzivnaDisjunkcija(*this); }
...
};

class EkskluzivnaDisjunkcijaStart : public Transformacija
{
public:
...
    Transformacija* kopija() const
        { return new EkskluzivnaDisjunkcijaStart(*this); }
...
};

class SlozenaTransformacija : public Transformacija
{
public:
```

```
...
    Transformacija* kopija() const
        { return new SlozenaTransformacija(*this); }
...
};
```

Postojanje metoda za kloniranje nam omogućava da na jednostavan način napišemo ispravan metod za inicijalizaciju kopije složene transformacije:

```
class SlozenaTransformacija : public Transformacija
{
...
    void init( const SlozenaTransformacija& s )
        {
        _transformacije.clear();
        for( unsigned i=0; i<s._transformacije.size(); i++ )
            _transformacije.push_back(
                s._transformacije[i]->kopija()
                );
        }
...
};
```

Izvedena implementacija koncepta kloniranja je u potpunosti ispravna, međutim, razmotrimo još nešto.

Pretpostavimo da usred naše hijerarhije klasa postoji *podhijerarhija* u čijoj je osnovi klasa `A`. Pretpostavimo i da u klasi `A` postoji deklarisan virtualan metod `metod` koji je implementiran u svim konkretnim klasama podhijerarhije. Neka objekat `a` pripada nekoj klasi iz te podhijerarhije, i neka nam je potrebna njegova kopija, na kojoj želimo primeniti metod `metod`. Prirodan način da napravimo kopiju jeste upotreba kloniranja, ali tu dolazimo do problema jer kloniranje vraća rezultat tipa `Transformacija*`, bez obzira na to što mi znamo da kopirani objekat, kao i njegova kopija, pripadaju podhijerarhiji. Jedini način da izvedemo ono što nam je potrebno jeste eksplicitna promena tipa:

```
A* a = ...;
A* b = dynamic_cast<A*>( a->kopija() );
b->metod(...);
```

Zbog toga što znamo da će kopija sigurno biti objekat podhijerarhije u čijoj je osnovi klasa `A`, ne moramo proveravati da li je `dynamic_cast` vratio prazan pokazivač. Štaviše, možemo izbeći nepotrebnu proveru tipa u vreme izvršavanja tako što ćemo umesto `dynamic_cast`-a upotrebiti npr. `reinterpret_cast`, ili eksplicitnu promenu tipa u stilu programskog jezika C:

```
A* a = ...;
A* b = (A*) a->kopija();
b->metod(...);
```

Izvesno je samo da nijedno od ponuđenih rešenja nije *lepo*. Zbog toga što se u hijerarhijama klasa često pojavljuju metodi koji za rezultat imaju pokazivač ili referencu na objekat nekog tipa koji je, u svakom konkretnom slučaju, poznat već u fazi prevođenja programa, u programskom jeziku C++ važi jedno pravilo za takve virtualne metode:

- Ako je virtualni metod deklarisan (i/ili definisan) u baznoj klasi tako da mu je tip rezultata pokazivač ili referenca na baznu klasu, tada se u izvedenoj klasi tip rezultata može izmeniti tako da predstavlja pokazivač ili referencu na neku izvedenu klasu.

Novi tip obično, ali ne uvek, odgovara klasi u kojoj se metod definiše.

Kakav je smisao ovog pravila? Ono se upotrebljava upravo da bismo rezultat kloniranja mogli tumačiti slobodnije bez eksplicitne promene tipa. U konkretnom primeru, ako bismo u klasi `A` metod `kopija` deklarisali da vrati pokazivač na klasu `A`:

```
class A : public ...
{ ...
    A* kopija() const
        {...}
... };
```

tada bismo prethodni segment koda mogli napisati bez eksplicitne promene tipa:

```
A* a = ...;
A* b = a->kopija();
b->metod(...);
```

Zbog toga što je uvek moguće doći u priliku da se u nekom delu programa koristi samo izabrana podhijerarhija, preporučljivo je da se metod `kopija` i slični metodi implementiraju tako da vraćaju najspecifičniji mogući tip. Primetimo još nešto: ako bi klasa `A` bila apstraktna klasa, tada je nephodno da se u njoj metod `kopija` deklariše sa izmenjenim tipom:

```
class A
{ ...
    A* kopija() const = 0;
... };
```

U našem slučaju, da bi program bio napisan u skladu sa prethodnom preporukom, potrebno je upotrebljavati odgovarajuće tipove rezultata metoda `kopija`:

```
class TransformacijaBajtova : public Transformacija
{...
    TransformacijaBajtova* kopija() const = 0;
...
};

class Translacija : public TransformacijaBajtova
{...
    Translacija* kopija() const
        { return new Translacija(_kod); }
...
};

class Rotacija : public TransformacijaBajtova
{...
    Rotacija* kopija() const
        { return new Rotacija(*this); }
...
};
```

```
class Zamena : public Transformacija
{...
    Zamena* kopija() const
        { return new Zamena(); }
...
};

class EkskluzivnaDisjunkcija : public Transformacija
{...
    EkskluzivnaDisjunkcija* kopija() const
        { return new EkskluzivnaDisjunkcija(*this); }
...
};

class EkskluzivnaDisjunkcijaStart : public Transformacija
{...
    EkskluzivnaDisjunkcijaStart* kopija() const
        { return new EkskluzivnaDisjunkcijaStart(*this); }
...
};

class SlozenaTransformacija : public Transformacija
{...
    SlozenaTransformacija* kopija() const
        { return new SlozenaTransformacija(*this); }
...
};
```

Bez obzira na određenu specifičnost predstavljenih metoda, moramo imati u vidu da se dinamičko vezivanje metoda uvek, pa i ovde, odvija samo na osnovu imena metoda i tipa objekta čiji se metod poziva, a ne na osnovu tipa rezultata.

Korak 10 - Pravljenje potrebne transformacije

Napisali smo i složenu i elementarne transformacije. Sada je na redu dovršavanje glavnog programa pisanjem metoda za detaljnu analizu komandne linije i pravljenje odgovarajuće transformacije ili kompozicije transformacija. Radi jednostavnosti, pretpostavićemo da se uvek pravi kompozicija transformacija, s tim da ona može biti sačinjena od jedne ili više elementarnih transformacija kodiranja:

```
class ProgramKoDek
{...
    //------------------------------------------------------------
    //  Članovi podaci
    ...
    SlozenaTransformacija transformacija;
};
```

Pri analizi je potrebno prepoznavati nazive transformacija i odgovarajuće parametre.

```
class ProgramKoDek
{
...
    void citanjeTransformacije( int argc, char** argv )
        {
```

```
for( int i=4; i<argc; i++ ){
    string param = argv[i];
    Transformacija* t = 0;

    // translacija
    if( param == "trans" ){
        i++;
        if( i>=argc )
            throw invalid_argument(
                "Nedostaje parametar translacije!"
                );
        int n;
        if( !sscanf( argv[i], "%d", &n ))
            throw invalid_argument(
                "Nije ispravan parametar translacije!"
                );
        t = new Translacija( n );
        }

    // rotacija
    else if( param == "rot" ){
        i++;
        if( i>=argc )
            throw invalid_argument(
                "Nedostaje parametar rotacije!"
                );
        int n;
        if( !sscanf( argv[i], "%d", &n ))
            throw invalid_argument(
                "Nije ispravan parametar rotacije!"
                );
        t = new Rotacija( n );
        }

    // zamena
    else if( param == "zamena" )
        t = new Zamena;

    // ekskluzivna disjunkcija
    else if( param == "xor" ){
        i++;
        if( i>=argc )
            throw invalid_argument(
                "Nedostaje parametar eks.disjunkcije!"
                );
        t = new EkskluzivnaDisjunkcija(
                    NizBajtova(argv[i])
                    );
        }

    // ekskluzivna disjunkcija početnim nizom
    else if( param == "xorstart" ){
        i++;
        if( i>=argc )
            throw invalid_argument(
                "Nedostaje parametar eks.disj.start!"
                );
```

```
                    int n;
                    if( !sscanf( argv[i], "%d", &n ))
                        throw invalid_argument(
                            "Nije ispravan parametar eks.disj.start!"
                            );
                    t = new EkskluzivnaDisjunkcijaStart( n );
                    }

                // greška
                else
                    throw invalid_argument( "Nepoznato kodiranje!" );

                transformacija.dodaj(t);
                }
            }
...
    void izvodjenjeOperacije()
        {
        if( kodiranje )
            transformacija.kodiranje(
                originalniNizBajtova,
                transformisaniNizBajtova
                );
        else
            transformacija.dekodiranje(
                originalniNizBajtova,
                transformisaniNizBajtova
                );
        }
...
};
```

U metodu `citanjeTransformacije` nekoliko puta se ponavlja čitanje celog broja iz komandne linije. Ponovljeni deo koda se može izdvojiti u statički metod `intArg`:

```
class ProgramKoDek
{
...
    void citanjeTransformacije( int argc, char** argv )
        {
        for( int i=4; i<argc; i++ ){
            string param = argv[i];
            Transformacija* t = 0;

            // translacija
            if( param == "trans" )
                t = new Translacija( intArg(
                                ++i, argc, argv, "translacije"
                                ));

            // rotacija
            else if( param == "rot" )
                t = new Rotacija( intArg(
                                ++i, argc, argv, "rotacije"
                                ));
```

```
            // zamena
            else if( param == "zamena" )
                t = new Zamena;

            // ekskluzivna disjunkcija
            else if( param == "xor" ){
                if( ++i >= argc )
                    throw invalid_argument(
                        "Nedostaje parametar eks.disjunkcije!"
                        );
                t = new EkskluzivnaDisjunkcija(
                            NizBajtova(argv[i])
                            );
                }

            // ekskluzivna disjunkcija početnim nizom
            else if( param == "xorstart" )
                t = new EkskluzivnaDisjunkcijaStart( intArg(
                                ++i, argc, argv, "eks.disj.start"
                                ));

            // greška
            else
                throw invalid_argument( "Nepoznato kodiranje!" );

            transformacija.dodaj(t);
            }
        }
    static int intArg( int i, int argc, char** argv, char* op )
        {
        if( i>=argc )
            throw invalid_argument(
                string("Nedostaje parametar ") + op + "!"
                );
        int n;
        if( !sscanf( argv[i], "%d", &n ))
            throw invalid_argument(
                string("Nije ispravan parametar ") + op + "!"
                );
        return n;
        }

...
};
```

Ovde možemo da se zaustavimo.

8.3 Rešenje

```
#include <iostream>
#include <fstream>
#include <string>
#include <exception>
#include <vector>

using namespace std;
```

```
//---------------------------------------------------------------
//  Tip Bajt
//---------------------------------------------------------------
typedef unsigned char Bajt;

//---------------------------------------------------------------
//  Klasa NizBajtova
//---------------------------------------------------------------
class NizBajtova
{
public:
    NizBajtova()
        {}

    NizBajtova( const string& s )
        {
        unsigned v = s.length();
        _niz.resize( v );
        for( unsigned i=0; i<v; i++ )
            _niz[i] = s[i];
        }

    Bajt operator[]( unsigned i ) const
        { return _niz[i]; }

    Bajt& operator[]( unsigned i )
        { return _niz[i]; }

    void promeniVelicinu( unsigned v )
        { _niz.resize(v); }

    unsigned velicina() const
        { return _niz.size(); }

    void pisi( ostream& ostr ) const
        {
        ostr.write( _niz.begin(), velicina() );
        if( !ostr )
            throw runtime_error(
                "Nije uspelo pisanje u datoteku!"
                );
        }

    void citaj( istream& istr )
        {
        istr.seekg( 0, ios::end );
        unsigned v = istr.tellg();
        istr.seekg( 0, ios::beg );
        _niz.resize( v );
        istr.read( _niz.begin(), v );
        if( !istr )
            throw runtime_error(
                "Nije uspelo citanje datoteke!"
                );
        }

private:
    vector<Bajt> _niz;
};
```

```
//-----------------------------------------------------------------
//  Klasa Transformacija
//-----------------------------------------------------------------
class Transformacija
{
public:
    virtual ~Transformacija()
        {}
    virtual void kodiranje(
                    const NizBajtova& org, NizBajtova& rez
                    ) const = 0;
    virtual void dekodiranje(
                    const NizBajtova& org, NizBajtova& rez
                    ) const = 0;
    virtual Transformacija* kopija() const = 0;
};
//-----------------------------------------------------------------
//  Klasa TransformacijaBajtova
//-----------------------------------------------------------------
class TransformacijaBajtova : public Transformacija
{
public:
    TransformacijaBajtova( Bajt kod )
        : _kod(kod)
        {}

    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = kodiranjeBajta( org[i] );
        }

    void dekodiranje(const NizBajtova& org, NizBajtova& rez) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = dekodiranjeBajta( org[i] );
        }

    TransformacijaBajtova* kopija() const = 0;

protected:
    virtual Bajt kodiranjeBajta( Bajt b ) const = 0;
    virtual Bajt dekodiranjeBajta( Bajt b ) const = 0;

    Bajt _kod;
};
//-----------------------------------------------------------------
//  Klasa Translacija
//-----------------------------------------------------------------
class Translacija : public TransformacijaBajtova
{
public:
```

```
    Translacija( Bajt kod )
        : TransformacijaBajtova(kod)
        {}

    Translacija* kopija() const
        { return new Translacija(_kod); }

protected:
    Bajt kodiranjeBajta( Bajt b ) const
        { return b + _kod; }

    Bajt dekodiranjeBajta( Bajt b ) const
        { return b - _kod; }
};

//---------------------------------------------------------------
//  Klasa Rotacija
//---------------------------------------------------------------
class Rotacija : public TransformacijaBajtova
{
public:
    Rotacija( Bajt kod )
        : TransformacijaBajtova(kod)
        {}

    Rotacija* kopija() const
        { return new Rotacija(*this); }

protected:
    Bajt kodiranjeBajta( Bajt b ) const
        { return (b << _kod) | (b >> (8 - _kod)); }

    Bajt dekodiranjeBajta( Bajt b ) const
        { return (b >> _kod) | (b << (8 - _kod)); }
};

//---------------------------------------------------------------
//  Klasa Zamena
//---------------------------------------------------------------
class Zamena : public Transformacija
{
public:
    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        { transformisanje( org, rez ); }
    void dekodiranje(const NizBajtova& org, NizBajtova& rez) const
        { transformisanje( org, rez ); }

    Zamena* kopija() const
        { return new Zamena(); }

private:
    static void transformisanje(
            const NizBajtova& org, NizBajtova& rez
            )
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=1; i<vel; i+=2 ){
            rez[i] = org[i-1];
            rez[i-1] = org[i];
            }
```

```
        if( vel % 2 )
            rez[vel-1] = org[vel-1];
        }
};

//-----------------------------------------------------------------
//  Klasa EkskluzivnaDisjunkcija
//-----------------------------------------------------------------
class EkskluzivnaDisjunkcija : public Transformacija
{
public:
    EkskluzivnaDisjunkcija( const NizBajtova& kod )
        : _kod( kod )
        {
        if( !kod.velicina() )
            throw range_error(
                "Duzina kodirajuceg niza mora biti veca od 0 "
                "(EkskluzivnaDisjunkcija)!"
                );
        }

    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        { transformisanje( org, rez ); }
    void dekodiranje(const NizBajtova& org, NizBajtova& rez) const
        { transformisanje( org, rez ); }

    EkskluzivnaDisjunkcija* kopija() const
        { return new EkskluzivnaDisjunkcija(*this); }

private:
    void transformisanje(
            const NizBajtova& org, NizBajtova& rez
            ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        unsigned n = _kod.velicina();
        for( unsigned int i=0; i<vel; i++ )
            rez[i] = org[i] ^ _kod[i%n];
        }

    NizBajtova _kod;
};

//-----------------------------------------------------------------
//  Klasa EkskluzivnaDisjunkcijaStart
//-----------------------------------------------------------------
class EkskluzivnaDisjunkcijaStart : public Transformacija
{
public:
    EkskluzivnaDisjunkcijaStart( unsigned duzinaKoda )
        : _duzinaKoda( duzinaKoda )
        {
        if( !duzinaKoda )
            throw range_error(
                "Duzina kodirajuceg podniza mora biti veca od 0 "
                "(EkskluzivnaDisjunkcijaStart)!"
                );
        }
```

```
    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        { transformisanje( org, rez ); }
    void dekodiranje(const NizBajtova& org, NizBajtova& rez) const
        { transformisanje( org, rez ); }

    EkskluzivnaDisjunkcijaStart* kopija() const
        { return new EkskluzivnaDisjunkcijaStart(*this); }

private:
    void transformisanje(
            const NizBajtova& org, NizBajtova& rez
            ) const
        {
        unsigned vel = org.velicina();
        rez.promeniVelicinu(vel);
        for( unsigned int i=0; i<_duzinaKoda; i++ )
            rez[i] = org[i];
        for( unsigned int i=_duzinaKoda; i<vel; i++ )
            rez[i] = org[i] ^ org[i%_duzinaKoda];
        }

    unsigned _duzinaKoda;
};

//-----------------------------------------------------------------
//  Klasa SlozenaTransformacija
//-----------------------------------------------------------------
class SlozenaTransformacija : public Transformacija
{
public:
    SlozenaTransformacija()
        {}

    SlozenaTransformacija( const SlozenaTransformacija& s )
        { init(s); }

    ~SlozenaTransformacija()
        { deinit(); }

    SlozenaTransformacija& operator = (
            const SlozenaTransformacija& s
            )
        {
        if( this != &s ){
            deinit();
            init(s);
            }
        return *this;
        }

    void dodaj( const Transformacija* t )
        { _transformacije.push_back( t ); }

    void kodiranje( const NizBajtova& org, NizBajtova& rez ) const
        {
        rez = org;
        for( unsigned i=0; i<_transformacije.size(); i++ ){
            NizBajtova p = rez;
            _transformacije[i]->kodiranje( p, rez );
            }
```

```
        }
    void dekodiranje(const NizBajtova& org, NizBajtova& rez) const
        {
        rez = org;
        unsigned n = _transformacije.size();
        for( unsigned i=0; i<n; i++ ){
            NizBajtova p = rez;
            _transformacije[n-1-i]->dekodiranje( p, rez );
            }
        }
    SlozenaTransformacija* kopija() const
        { return new SlozenaTransformacija(*this); }
private:
    void deinit()
        {
        for( unsigned i=0; i<_transformacije.size(); i++ )
            delete _transformacije[i];
        }
    void init( const SlozenaTransformacija& s )
        {
        _transformacije.clear();
        for( unsigned i=0; i<s._transformacije.size(); i++ )
            _transformacije.push_back(
                s._transformacije[i]->kopija()
                );
        }
    vector<const Transformacija*> _transformacije;
};
//----------------------------------------------------------------
//  Program
//----------------------------------------------------------------
class ProgramKoDek
{
public:
    void izvrsi( int argc, char** argv )
        {
        priprema( argc, argv );
        obrada();
        }
private:
    //------------------------------------------------------------
    //  Analiza parametara
    //------------------------------------------------------------
    void priprema( int argc, char** argv )
        {
        //  Površna provera da li je na raspolaganju
        //  dovoljno parametara
        if( argc < 5 )
            throw invalid_argument(
                "Nisu navedeni svi neophodni parametri!"
                );
```

```
    //  Smer obavljanja operacije
    string smer = argv[1];
    if( smer == "k" )
        kodiranje = true;
    else if( smer == "d" )
        kodiranje = false;
    else
        throw invalid_argument(
            "Tip operacije mora biti 'k' (kodiranje) ili "
            "'d' (dekodiranje)!"
            );

    //  Čitanje transformacije
    citanjeTransformacije(argc, argv );

    //  Izdvajanje naziva ulazne i izlazne datoteke
    imeUlazneDatoteke = argv[2];
    imeIzlazneDatoteke = argv[3];
    }

void citanjeTransformacije( int argc, char** argv )
    {
    for( int i=4; i<argc; i++ ){
        string param = argv[i];
        Transformacija* t = 0;

        // translacija
        if( param == "trans" )
            t = new Translacija( intArg(
                            ++i, argc, argv, "translacije"
                            ));

        // rotacija
        else if( param == "rot" )
            t = new Rotacija( intArg(
                            ++i, argc, argv, "rotacije"
                            ));

        // zamena
        else if( param == "zamena" )
            t = new Zamena;

        // ekskluzivna disjunkcija
        else if( param == "xor" ){
            if( ++i >= argc )
                throw invalid_argument(
                    "Nedostaje parametar eks.disjunkcije!"
                    );
            t = new EkskluzivnaDisjunkcija(
                        NizBajtova(argv[i])
                        );
            }

        // ekskluzivna disjunkcija početnim nizom
        else if( param == "xorstart" )
            t = new EkskluzivnaDisjunkcijaStart( intArg(
                            ++i, argc, argv, "eks.disj.start"
                            ));
```

```
            // greška
            else
                throw invalid_argument( "Nepoznato kodiranje!" );

            transformacija.dodaj(t);
            }
        }

    static int intArg( int i, int argc, char** argv, char* op )
        {
        if( i>=argc )
            throw invalid_argument(
                string("Nedostaje parametar ") + op + "!"
                );
        int n;
        if( !sscanf( argv[i], "%d", &n ))
            throw invalid_argument(
                string("Nije ispravan parametar ") + op + "!"
                );
        return n;
        }

    //-------------------------------------------------------------
    //  Obrada
    //-------------------------------------------------------------
    void obrada()
        {
        citanjeDatoteke();
        izvodjenjeOperacije();
        pisanjeDatoteke();
        }

    void izvodjenjeOperacije()
        {
        if( kodiranje )
            transformacija.kodiranje(
                originalniNizBajtova,
                transformisaniNizBajtova
                );
        else
            transformacija.dekodiranje(
                originalniNizBajtova,
                transformisaniNizBajtova
                );
        }

    //-------------------------------------------------------------
    //  Čitanje i pisanje
    //-------------------------------------------------------------
    void citanjeDatoteke()
        {
        ifstream uDat( imeUlazneDatoteke.c_str(), ios::binary );
        if( !uDat )
            throw runtime_error(
                "Nije uspelo otvaranje ulazne datoteke!"
                );
        originalniNizBajtova.citaj( uDat );
        }
```

```
        void pisanjeDatoteke()
            {
            ofstream iDat( imeIzlazneDatoteke.c_str(), ios::binary );
            if( !iDat )
                throw runtime_error(
                    "Nije uspelo otvaranje izlazne datoteke!"
                    );
            transformisaniNizBajtova.pisi( iDat );
            }

        //-------------------------------------------------------------
        //  Članovi podaci
        //-------------------------------------------------------------
        bool kodiranje;
        string imeUlazneDatoteke;
        string imeIzlazneDatoteke;
        NizBajtova originalniNizBajtova;
        NizBajtova transformisaniNizBajtova;
        SlozenaTransformacija transformacija;
    };

    //-----------------------------------------------------------------
    //  Program za kodiranje/dekodiranje
    //-----------------------------------------------------------------
    main( int argc, char** argv )
    {
        try {
            ProgramKoDek program;
            program.izvrsi( argc, argv );
            }
        catch( invalid_argument& e ){
            cerr <<"Neispravna upotreba programa!" << endl;
            cerr << e.what() << endl;
            cerr << "Kodiranje (k|d) <ulazna dat.> <izlazna dat.> "
                    "<transformacija>" << endl;
            return 1;
            }
        catch( exception& e ){
            cerr <<"GRESKA: " << e.what() << endl;
            return 1;
            }

        return 0;
    }
```

8.4 Rezime

Pravila za pisanje lepih (tj. dobrih) programa najčešće nisu jednosmerna ulica koju nipošto ne smemo napustiti, već smernice čija je osnovna namena da podignu stepen opreznosti programera u određenim situacijama. Ne postoji jedinstven idealan skup pravila, ali se pokazuje da su skupovi pravila koji se pojavljuju u različitim timovima često u velikoj meri ekvivalentni. Problemima lepog pisanja bavi se više odličnih knjiga. Srećom po zainteresovane čitaoce, neke od najboljih knjiga iz oblasti razvoja objektno orijentisanog

softvera su prevedene na srpski jezik. Među njima se ističu [Meyer 1997], [Gamma 1995], i [Fowler 1999].

Među najvažnije delove ovog primera spada pravljenje kompozitne transformacije. Vrlo često je potrebno da se u okviru neke hijerarhije definiše klasa koja predstavlja kompozitni objekat. Takva klasa se mora istovremeno ponašati kao element hijerarhije i kao kompozicija objekata iz te iste hijerarhije. Obično nije veliki problem pomiriti te dve različite prirode, ali je potrebno biti oprezan jer je prilično lako napraviti prilično nezgodne propuste. Skoro uvek u takvim situacijama je u čitavoj hijerarhiji potrebno podržati koncept kloniranja objekta.

Za vežbu preporučujemo:

- eliminisanje ponavljanja iz postupaka kodiranja i dekodiranja klasa `EkskluzivnaDisjunkcija` i `EkskluzivnaDisjunkcijaStart`;
- pisanje efikasnijih metoda za kodiranje i dekodiranje složenih transformacija;
- primeniti horizontalno izdvajanje metoda u klasama `Translacija` i `Rotacija` i uporediti složenost koda i performanse sa ponuđenim rešenjem.

9 - Igra „Život“, II deo

9.1 Zadatak

U jednom od prethodnih primera je implementirana igra „Život“. Sada ćemo menjanjem pravila malo zakomplikovati igru.

Umesto da se sve jedinke ponašaju na isti način, definisaćemo četiri različite vrste jediniki. *Čekalice* se ponašaju kao jedinke u originalnoj igri, tj. ne kreću se i preživljavaju ako imaju tačno dva ili tri suseda. *Puzalice*, *šetalice* i *skakalice* u svakom koraku pokušavaju da pređu u drugu ćeliju i preživljavaju ako u tome uspeju, a umiru ako nemaju gde da se pomere.

Puzalice pokušavaju da pređu u neku od četiri neposredno susedne ćelije. Šetalice pokušavaju da pređu u neku od ukoso susednih ćelija. Skakalice pokušavaju da „uskoče“ u jednu od osam ćelija na udaljenosti 2. Na priloženom crtežu je predstavljen deo matrice na kome se nalazi jedinka X. Ako je X puzalica, ona će pokušati da pređe u ćelije označene slovom P; ako je šetalica, u ćelije označene slovom Š; a ako je skakalica, onda u ćelije označene slovom S. Ćelija u koju jedinka prelazi bira se metodom slučajnog izbora iz skupa slobodnih odgovarajućih ćelija. Ako nijedna od odgovarajućih ćelija nije slobodna, jedinka umire.

S		S		S
	Š	P	Š	
S	P	X	P	S
	Š	P	Š	
S		S		S

Ako više jedinki pokuša da dođe u istu ćeliju (bilo pomeranjem ili samim rađanjem), u tome može uspeti samo najjača, dok ostale ne preživljavaju. Najjača jedinka je skakalica, pa onda šetalica, pa puzalica, a najslabija je čekalica. Ako su jedinke iste snage, samo jedna od njih preživljava.

Jedinke se rađaju na isti način kao i u originalnoj igri, tj. u praznim ćelija koje imaju tačno tri jedinke u susedstvu. Koja vrsta jedinke će biti rođena utvrđuje se metodom slučajnog izbora.

Cilj zadatka

U ovom primeru ćemo:

- upotrebiti hijerarhiju klasa za implementiranje različitog ponašanja za različite vrste modeliranih objekata;
- upoznati tehniku *praznih objekata*;
- upoznati tehniku *muva-lakih objekata*.

Pretpostavljena znanja

Za uspešno praćenje rešavanja ovog zadatka pretpostavlja se:

- da je pažljivo obrađen primer *5 - Igra „Život“*, sa početkom na strani 145;
- poznavanje rada sa hijerarhijama klasa.

9.2 Rešavanje zadatka

Zadatak ćemo rešavati tako što ćemo postepeno proširivati rešenje zadatka *5 - Igra „Život“*, koje je izloženo na strani 163. Najpre ćemo pripremiti program za dodavanje novih vrsta jedinki. Posle dodavanja novih vrsta jedinki razmotrićemo neke mogućnosti za povećavanje jednostavnosti i efikasnosti.

Korak 1 - Priprema za uvođenje više vrsta jedinki

Kada smo implementirali originalnu igru, tablu za igru smo modelirali matricom logičkih vrednosti, gde je logička vrednost `true` označavala da postoji jedinka u ćeliji, a vrednost `false` je označavala da je ćelija prazna. U slučaju igre sa više različitih vrsta jedinki, jasno je da logičke vrednosti više ne zadovoljavaju naše potrebe.

Jedna mogućnost je da upotrebimo cele brojeve od 0 do 4, gde bi 0 bila oznaka za praznu ćeliju, a vrednosti 1 do 4 oznake za različite vrste jedinki. Takav pristup je uobičajen za klasične proceduralne programske jezike, kao što je na primer C. Međutim, u objektno orijentisanim programskim jezicima se takav pristup ne smatra dobrim. Upotreba oznaka za različite vrste jedinki dovodi do upotrebe složenih struktura odlučivanja (tj. naredbe `switch` ili višestrukih naredbi `if`), čije održavanje komplikuje kasnije dodavanje novih vrsta jedinki ili menjanje ponašanja postojećih vrsta jedinki. Umesto toga, preporučuje se izgradnja hijerarhija klasa i implementacija odlučivanja primenom dinamičkog vezivanja metoda.

Priprema za implementaciju sa više vrsta jedinki obuhvata nekoliko koraka. Najpre ćemo definisati klasu `Jedinka`. Članove podatke i metode ćemo dodavati postepeno, kako se za njima bude ukazivala potreba. U ovom trenutku nam je sasvim dovoljna prazna klasa, bez članova i metoda:

```
class Jedinka
{};
```

Umesto matrice logičkih vrednosti, tablu za igru ćemo predstavljati matricom pokazivača na jedinke. Od definicije matrice ćemo napraviti šablon. Proširićemo ponašanje klase dodavanjem novog konstruktora koji inicijalizuje vrednosti svih elemenata matrice datom vrednošću:

```
template< class T >
class Matrica
{
public:
    //-----------------------------
    //  Konstruktor i ostali potrebni metodi
    Matrica( int visina =0, int sirina =0 )
        : _Kolone(sirina)
        {
        for( int i=0; i<sirina; i++ )
            _Kolone[i].resize(visina);
        }

    Matrica( int visina, int sirina, const T& x )
        : _Kolone(sirina)
        {
        for( int i=0; i<sirina; i++ ){
            _Kolone[i].resize(visina);
            for( int j=0; j<visina; j++ )
                _Kolone[i][j] = x;
            }
        }

    //-----------------------------
    //  pristupanje elementima
    vector<T>& operator [](  int i )
        { return _Kolone[i]; }
    const vector<T>& operator [](  int i ) const
        { return _Kolone[i]; }

    int Sirina() const
        { return _Kolone.size(); }
```

```
    int Visina() const
        { return _Kolone.size() ? _Kolone[0].size() : 0; }

private:
    //-----------------------------
    //  podaci clanovi
    vector< vector< T > >   _Kolone;
};
```

Najsloženiji deo posla koji ćemo uraditi u ovom koraku jeste da svuda u programu zamenimo ranije upotrebljavanu klasu `Matrica` instancom šablona `Matrica<Jedinka*>`. Problem je u tome što sada na svakom mestu gde postavljamo novu jedinku, moramo tu novu jedinku i praviti, a na svakom mestu na kome se jedinka uklanja iz ćelije, mi ćemo sada morati da obrišemo objekat koji predstavlja jedinku. Pored toga, na svim mestima na kojima brišemo prethodni sadržaj matrice, bilo zato što uklanjamo objekat ili mu dodeljujemo sadržaj druge matrice, moraćemo da uklanjamo sve jedinke koje postoje u matrici. Slede izmenjeni delovi koda:

```
class Igra
{
public:
    //-----------------------------
    //  destruktor
    ~Igra()
        { ObrisiKonfiguraciju(); }

    ...

    void Citaj( istream& istr )
        {
        int s, v;
        istr >> s >> v;
        Matrica<const Jedinka*> m(v,s);
        for( int i=0; i<v; i++ )
            for( int j=0; j<s; j++ ){
                char c;
                istr >> c;
                m[j][i] =
                    c=='X'
                        ? new Jedinka
                        : 0;
                }
        ObrisiKonfiguraciju();
        _Konfiguracija = m;
        // ne moramo brisati jedinke matrice m,
        // jer nismo pravili njihove kopije vec smo prepisali
        // pokazivace na njih u matricu _Konfiguracija
        }

private:
    //-----------------------------
    //  pomocni metodi
    void ObrisiKonfiguraciju()
        {
```

```
            for( int i=_Konfiguracija.Visina()-1; i>=0; i-- )
                for( int j=_Konfiguracija.Sirina()-1; j>=0; j-- )
                    delete _Konfiguracija[j][i];
            }

        ...

        void IzracunavanjeGeneracije()
            {
            int sirina = _Konfiguracija.Sirina();
            int visina = _Konfiguracija.Visina();
            Matrica<const Jedinka*> nova( visina, sirina, 0 );
            for( int i=0; i<sirina; i++ )
                for( int j=0; j<visina; j++ ){
                    int bs = BrojSuseda(i,j);
                    nova[i][j] =
                        bs == 3
                        || (bs == 2 && _Konfiguracija[i][j])
                            ? new Jedinka
                            : 0;
                    }
            ObrisiKonfiguraciju();
            _Konfiguracija = nova;
            // ne moramo brisati jedinke matrice m,
            // jer nismo pravili njihove kopije vec smo prepisali
            // pokazivace na njih u matricu _Konfiguracija
            }

        //-----------------------------
        //  podaci clanovi
        Matrica<const Jedinka*> _Konfiguracija;
    };
```

Napravljene izmene možemo proveriti prevođenjem i izvršavanjem programa. Trebalo bi da sve radi na potpuno isti način kao ranije.

Korak 2 - Prenošenje odgovornosti na jedinke

Naše jedinke još uvek ne umeju ništa da rade. Razlog za definisanje klase `Jedinka` i planiranja hijerarhije klasa jedinki jeste različito ponašanje različitih vrsta jedinki. Da bi to različito ponašanje moglo da dođe do izražaja, potrebno je da se deo odgovornosti, a koje se tiču tog ponašanja, prenese na jedinke.

Odlučivanje o preživljavanju i promeni položaja

Osnovna razlika u ponašanju jedinki se svodi na prepoznavanje da li jedinka preživljava i izračunavanje njenog novog položaja. To možemo uraditi pomoću dva metoda:

```
bool Prezivljava() const;
void NoviPolozaj( int& x, int& y );
```

Međutim, za nove vrste jedinki rezultati obeju funkcija se izračunavaju na isti način. Ili ćemo izračunavanje ponoviti u svakom od ovih metoda, ili ćemo morati da definišemo tačan redosled pozivanja ovih metoda. Jasno je da oba pristupa imaju svoje mane: prvi se odlikuje suvišnim izračunavanjem, a drugi uvodi potpuno neintuitivno ponašanje metoda. U takvim situacijama obično je najbolje uraditi nešto drugačije od oba opisana načina.

Jedna mogućnost je da se izračunavanje izdvoji iz oba metoda u poseban metod, koji bi morao da se poziva pre upotrebe oba prethodna metoda:

```
void RacunanjeKoraka();
```

Tako bismo imali tri metoda, ali bi njihova semantika bila jasnija.

Alternativa je da definišemo samo jedan metod, koji će uraditi sav posao:

```
void RacunanjeKoraka( bool& prezivljava, int& nx, int& ny );
```

Upotreba više metoda ima za posledicu da svaka jedinka mora da pamti podatke o tome da li preživljava i na kom mestu će se nalaziti, što predstavlja suvišno opterećivanje podacima koji se koriste samo veoma kratko vreme. Zato ćemo u našem rešenju pribeći definisanju samo jednog metoda: `RacunanjeKoraka`.

Međutim, već pri pomisli na njegovu implementaciju postaje nam jasno da su nam neophodni još neki podaci. Da bi jedinka mogla da izračuna šta će se sa njom dešavati, mora znati u kakvom se okruženju nalazi. Jedan način je da se svakoj jedinci dodaju članovi podaci koji predstavljaju koordinate ćelije u kojoj se nalazi, kao i referenca na tablu za igru. Drugi način je da se odgovarajuće informacije prenose kao argumenti metoda `RacunanjeKoraka`. Činjenica da se te informacije upotrebljavaju isključivo u metodu `RacunanjeKoraka` predstavlja dovoljno snažan podsticaj da se odlučimo za drugo rešenje. Umesto da podatak o preživljavanju bude promenljivi argument, vraćaćemo ga kao rezultat metoda:

```
class Jedinka
{
public:
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const
        {
        ...
        }
};
```

Pre nego što implementiramo ovaj metod, prvo ćemo da vidimo u kakvim se tačno uslovima koristi. Potrebno je izmeniti metod `IzracunavanjeGeneracije` klase `Igra`. Pri tome ćemo razdvojiti odlučivanje za pune i prazne ćelije. Umesto:

```
...
int bs = BrojSuseda(i,j);
nova[i][j] =
    bs == 3
    || (bs == 2 && _Konfiguracija[i][j])
        ? new Jedinka
        : 0;
...
```

napisaćemo nešto kao:

```
...
const Jedinka* novaJedinka = 0;
```

```
int nx=i;
int ny=j;

// ako postoji jedinka u celiji...
if( _Konfiguracija[i][j] ){
    if (
        _Konfiguracija[i][j]->RacunanjeKoraka(
            _Konfiguracija, i, j, nx, ny
            )
        )
        novaJedinka = new Jedinka;
    }

// ako ne postoji jedinka u celiji...
else{
    if( BrojSuseda(i,j) == 3 )
        novaJedinka = new Jedinka;
    }

nova[nx][ny] = novaJedinka;
...
```

Dobili smo nešto duži kod, ali sada se odlučivanje o tome da li će jedinka preživeti ili ne prepušta samoj klasi `Jedinka`. Neke probleme još uvek nismo rešili. Tako se još uvek u samom kodu eksplicitno prave nove jedinke, što će biti problem u slučaju više različitih vrsta jedinki. Naime, tip nove jedinke zavisi od tipa postojeće. Zbog toga ćemo i odgovornost za pravljenje nove jedinke prepustiti jedinkama. Klasi `Jedinka` dodajemo metod `NovaJedinka`:

```
const Jedinka* NovaJedinka() const
    { return new Jedinka; }
```

a u metodu `IzracunavanjeGeneracije` klase `Igra`, u slučaju pune ćelije, umesto:

```
novaJedinka = new Jedinka;
```

pišemo:

```
novaJedinka = _Konfiguracija[i][j]->NovaJedinka();
```

Drugi problem nastupa u slučaju kada dve jedinke pokušaju da dođu u istu praznu ćeliju. U tom slučaju razmatra se *snaga* jedinke, kako je navedeno u postavci zadatka. Klasi `Jedinka` dodajemo novi metod:

```
int Snaga() const
    { return 1; }
```

a u metodu `IzracunavanjeGeneracije` klase `Igra`, umesto:

```
nova[nx][ny] = novaJedinka;
```

pišemo:

```
if( novaJedinka )
    if( nova[nx][ny] )
        if( nova[nx][ny]->Snaga()
                < novaJedinka->Snaga()
            ){
```

```
                delete nova[nx][ny];
                nova[nx][ny] = novaJedinka;
                }
            else
                delete novaJedinka;
        else
            nova[nx][ny] = novaJedinka;
```

Pisanje i čitanje

Još jedna karakteristika koja se razlikuje za svaku vrstu jedinki jeste njihov izgled. Izgled jedinki nam je važan prilikom čitanja ili pisanja sadržaja table za igru.

Pisanje oznake jedinke se može prepustiti metodu `Izgled` klase `Jedinka`:

```
char Izgled() const
    { return 'X'; }
```

Metod `Pisi` klase `Igra` menjamo tako da koristi novi metod `Izgled`:

```
void Pisi( ostream& ostr ) const
    {
    int s = _Konfiguracija.Sirina();
    int v = _Konfiguracija.Visina();
    ostr << s << ' ' << v;
    for( int i=0; i<v; i++ ){
        ostr << endl;
        for( int j=0; j<s; j++ )
            ostr << (_Konfiguracija[j][i]
                            ? _Konfiguracija[j][i]->Izgled()
                            : '.'
                    );
        }
    }
```

Čitanje predstavlja malo složeniji problem. Naime, kada čitamo oznaku objekta mi još uvek nemamo napravljen objekat čija je oznaka u pitanju, pa se čin prepoznavanja pročitane oznake mora obaviti drugačije. Jedno rešenje je da se van klasa napiše funkcija koja na oznovu pročitane oznake pravi objekat odgovarajuće klase. Umesto funkcije, bolje je napisati statički metod u baznoj klasi hijerarhije. Takav metod deklarišemo u samoj klasi, a implementiramo ga nakon deklaracija svih klasa hijerarhije. Sličan problem i odgovarajuće rešenje smo već imali u primeru *7 - Enciklopedija*, na strani 233.

U klasi `Jedinka` pišemo:

```
static Jedinka* NapraviJedinku( char c );
```

a neposredno pre definicije klase `Igra` pišemo implementaciju metoda:

```
Jedinka* Jedinka::NapraviJedinku( char c )
{
    if( c=='X' )
        return new Jedinka;
    else
        return 0;
}
```

Pri čitanju koristimo nov metod:

```
void Citaj( istream& istr )
    {
    int s, v;
    istr >> s >> v;
    Matrica<const Jedinka*> m(v,s);
    for( int i=0; i<v; i++ )
        for( int j=0; j<s; j++ ){
            char c;
            istr >> c;
            m[j][i] = Jedinka::NapraviJedinku(c);
            }
    ObrisiKonfiguraciju();
    _Konfiguracija = m;
    }
```

Sličan problem imamo i pri rađanju novih jedinki. Zato pravimo statički metod bez argumenata:

```
static Jedinka* NapraviJedinku();
```

i takođe ga implementiramo neposredno pre definicije klase `Igra`:

```
Jedinka* Jedinka::NapraviJedinku()
{
    return new Jedinka;
}
```

Sada metod `Igra::IzracunavanjeGeneracije` izgleda ovako:

```
void IzracunavanjeGeneracije()
    {
    int sirina = _Konfiguracija.Sirina();
    int visina = _Konfiguracija.Visina();
    Matrica<const Jedinka*> nova( visina, sirina, 0 );
    for( int i=0; i<sirina; i++ )
        for( int j=0; j<visina; j++ ){
            const Jedinka* novaJedinka = 0;
            int nx=i;
            int ny=j;

            // ako postoji jedinka u celiji...
            if( _Konfiguracija[i][j] ){
                if (
                    _Konfiguracija[i][j]->RacunanjeKoraka(
                        _Konfiguracija, i, j, nx, ny
                        )
                    )
                    novaJedinka = _Konfiguracija[i][j]
                                        ->NovaJedinka();
                }
            // ako ne postoji jedinka u celiji...
            else{
                if( BrojSuseda(i,j) == 3 )
                    novaJedinka = Jedinka::NapraviJedinku();
                }
```

```
                // ako jedinka prezivljava ili se radja nova...
                if( novaJedinka )
                    if( nova[nx][ny] )
                        if( nova[nx][ny]->Snaga()
                                < novaJedinka->Snaga()
                            ){
                            delete nova[nx][ny];
                            nova[nx][ny] = novaJedinka;
                            }
                        else
                            delete novaJedinka;
                    else
                        nova[nx][ny] = novaJedinka;
                }
            ObrisiKonfiguraciju();
            _Konfiguracija = nova;
            }
```

Klasa Jedinka

Za implementaciju računanja koraka klase `Jedinka` neophodan nam je metod `BrojSuseda` klase `Igra`. Da ne bismo iz metoda klase `Jedinka` pozivali metode klase `Igra`, kopiramo ovaj metod u klasu `Jedinka` i unosimo minimalne neophodne izmene. Konačno, dobili smo klasu `Jedinka` koja odgovara našim potrebama:

```
//---------------------------------------------------------------
//  Klasa Jedinka
//---------------------------------------------------------------
class Jedinka
{
public:
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const
        {
        int n = BrojSuseda( m, x, y );
        bool prezivljava = n==2 || n==3;
        nx = x;
        ny = y;
        return prezivljava;
        }

    const Jedinka* NovaJedinka() const
        { return new Jedinka; }

    int Snaga() const
        { return 1; }

    static Jedinka* NapraviJedinku();
    static Jedinka* NapraviJedinku( char c );

private:
    int BrojSuseda(
            const Matrica<const Jedinka*>& m,
            int k, int v
        ) const
        {
```

```
            int sirina = m.Sirina();
            int visina = m.Visina();
            int n=0;
            for( int i=-1; i<2; i++ )
                for( int j=-1; j<2; j++ )
                    if( m[(k+i+sirina)%sirina]
                         [(v+j+visina)%visina] )
                        n++;
            n--;
            return n;
            }
};
```

Kako klasa `Jedinka` koristi klasu `Matrica`, ona se u kodu mora nalaziti između definicija šablona `Matrica` i klase `Igra`. Dobijeni kod možemo prevesti i izvršiti.

Korak 3 - Dodavanje novih vrsta jedinki

Napravićemo dve nove klase: `Cekalica` i `Puzalica`. Obe klase će naslediti klasu `Jedinka`. Najpre pravimo samo klasu `Cekalica`.

Čekalice

Implementacije metoda klase `Jedinka` ćemo spustiti niže, u klasu `Cekalica`, a u klasi `Jedinka` ćemo ostaviti samo deklaracije čisto virtualnih metoda, kako bismo obezbedili njihovo dinamičko vezivanje. Metod `BrojSuseda` ostavljamo u klasi `Jedinka`, s tim da bude zaštićen, a ne privatan. Pošto smo dodali virtualne metode, neophodan nam je i virtualan destruktor:

```
class Jedinka
{
public:
    virtual ~Jedinka()
        {}
    virtual bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const = 0;
    virtual const Jedinka* NovaJedinka() const = 0;
    virtual int Snaga() const = 0;
    virtual char Izgled() const = 0;

    static Jedinka* NapraviJedinku();
    static Jedinka* NapraviJedinku( char c );

protected:
    int BrojSuseda(...) const
        { ... }
};

class Cekalica : public Jedinka
{
public:
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
```

```
            ) const
            {
            int n = BrojSuseda( m, x, y );
            bool prezivljava = n==2 || n==3;
            nx = x;
            ny = y;
            return prezivljava;
            }

        const Jedinka* NovaJedinka() const
            { return new Cekalica; }

        int Snaga() const
            { return 1; }

        char Izgled() const
            { return 'X'; }
};
```

Na svim mestima u programu gde su se pravili objekti klase `Jedinka` menjamo kod tako da se prave objekti klase `Čekalica` (u metodu `NovaJedinka` klase `Čekalica` smo to već uradili):

```
Jedinka* Jedinka::NapraviJedinku()
{
    return new Cekalica;
}

Jedinka* Jedinka::NapraviJedinku( char c )
{
    if( c=='X' )
        return new Cekalica;
    else
        return 0;
}
```

Puzalice

Klasu `Puzalica` započinjemo kopiranjem klase `Čekalica`. Menjamo izgled u „`P`“ i snagu u 2. Metod `NovaJedinka` pravi i vraća `Puzalicu`. Jedini ozbiljniji posao predstavlja pisanje odgovarajućeg metoda `RacunanjeKoraka`. Umesto brojanja suseda, ovde je potrebno da se vidi da li je slobodna neka od ćelija u koja puzalica želi da pređe.

U klasi `Puzalica` pišemo odovarajući pomoćni metod koji izračunava niz slobodnih ćelija na osnovu datih ponuđenih ćelija koja je potrebno proveriti. Metod `SlobodneĆelije` dobija referencu na matricu ćelija (`m`), koordinate ćelije čija nas okolina zanima (`x0`, `y0`), niz parova relativnih koordinata ponuđenih ćelija (`pozicije`), dužinu datog niza ponuđenih ćelija (`n`) i adresu niza (`prazna`) u koji će upisati redne brojeve slobodnih ćelija. Rezultat metoda je broj pronađenih praznih ćelija.

```
    int SlobodneCelije(
            const Matrica<const Jedinka*>& m,
            int x0, int y0,
            int pozicije[][2], int n, int prazna[]
            ) const
        {
```

```
        int npraznih = 0;
        int s = m.Sirina();
        int v = m.Visina();
        for( int i=0; i<n; i++ ){
            int x = (x0 + pozicije[i][0] + s) % s;
            int y = (y0 + pozicije[i][1] + v) % v;
            if( m[x][y] == 0 )
                prazna[ npraznih++ ] = i;
            }
        return npraznih;
        }
```

Metod `RacunanjeKoraka` klase `Puzalica` najpre izračunava niz slobodnih ćelija, a zatim bira jednu od njih:

```
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const
        {
        int pozicije[][2] = { {0,-1}, {0,1}, {1,0}, {-1,0} };
        int prazna[4];
        int n = SlobodneCelije( m, x, y, pozicije, 4, prazna );
        if( n>0 ){
            int s = m.Sirina();
            int v = m.Visina();
            int i = random(n);
            nx = (x + pozicije[prazna[i]][0] + s) % s;
            ny = (y + pozicije[prazna[i]][1] + v) % v;
            return true;
            }
        else
            return false;
        }
```

Primetimo da će sličan postupak biti potreban i za druge pokretne klase. Zato razdvajamo implementaciju ponašanja od podataka o željenim ćelijama. Postupak izračunavanja izdvajamo u novi pomoćni metod `Pomeranje`:

```
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const
        {
        int pozicije[][2] = { {0,-1}, {0,1}, {1,0}, {-1,0} };
        return Pomeranje( m, x, y, nx, ny, pozicije, 4 );
        }

protected:
    bool Pomeranje(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny, int pozicije[][2], int np
        ) const
        {
        int prazna[8];
        int n = SlobodneCelije( m, x, y, pozicije, np, prazna );
```

```
        if( n>0 ){
            int s = m.Sirina();
            int v = m.Visina();
            int i = random(n);
            nx = (x + pozicije[prazna[i]][0] + s) % s;
            ny = (y + pozicije[prazna[i]][1] + v) % v;
            return true;
            }
        else
            return false;
        }
```

Menjamo metode `NapraviJedinku` tako da prave i `Puzalice`:

```
Jedinka* Jedinka::NapraviJedinku()
{
    int p = random(100);
    // sa verovatnoćom od 50% pravimo čekalicu
    if( p < 50 )
        return new Cekalica;
    // inače (preostalih 50%) pravimo puzalicu
    else
        return new Puzalica;
}

Jedinka* Jedinka::NapraviJedinku( char c )
{
    if( c=='X' )
        return new Cekalica;
    else if( c=='P' )
        return new Puzalica;
    else
        return 0;
}
```

Šetalice i skakalice

Šetalice i skakalice su veoma slične puzalicama. Zbog toga pravimo zajedničku baznu klasu `PokretnaJedinka` u koju prenosimo pomoćne metode klase `Puzalica`:

```
class PokretnaJedinka : public Jedinka
{
protected:
    bool Pomeranje(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny, int pozicije[][2], int np
        ) const
        { ... }

    int SlobodneCelije(
            const Matrica<const Jedinka*>& m,
            int x0, int y0,
            int pozicije[][2], int n, int prazna[]
            ) const
        { ... }
};
```

```
class Puzalica : public PokretnaJedinka
{
public:
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const
        {
        int pozicije[][2] = { {0,-1}, {0,1}, {1,0}, {-1,0} };
        return Pomeranje( m, x, y, nx, ny, pozicije, 4 );
        }

    const Jedinka* NovaJedinka() const
        { return new Puzalica; }

    int Snaga() const
        { return 2; }

    char Izgled() const
        { return 'P'; }
};
```

Zatim, po uzoru na klasu `Puzalica`, pravimo klasu `Šetalica`:

```
class Setalica : public PokretnaJedinka
{
public:
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const
        {
        int pozicije[][2] = { {-1,-1}, {-1,1}, {1,-1}, {1,1} };
        return Pomeranje( m, x, y, nx, ny, pozicije, 4 );
        }

    const Jedinka* NovaJedinka() const
        { return new Setalica; }

    int Snaga() const
        { return 3; }

    char Izgled() const
        { return 'S'; }
};
```

Na sličan način pravimo i klasu `Skakalica`:

```
class Skakalica : public PokretnaJedinka
{
public:
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const
        {
        int pozicije[][2] = {
            {-2,-2}, {-2,0}, {-2,2}, {0,-2}, {0,2},
            {2,-2}, {2,0}, {2,2}
            };
```

```
            return Pomeranje( m, x, y, nx, ny, pozicije, 8 );
            }

        const Jedinka* NovaJedinka() const
            { return new Skakalica; }

        int Snaga() const
            { return 4; }

        char Izgled() const
            { return 'K'; }
};
```

Menjamo metode `NapraviJedinku` tako da prave i nove klase:

```
Jedinka* Jedinka::NapraviJedinku()
{
    int p = random(100);
    // sa verovatnoćom od 25% pravimo čekalicu
    if( p < 25 )
        return new Cekalica;
    // sa verovatnoćom od 25% pravimo puzalicu
    else if( p < 50 )
        return new Puzalica;
    // sa verovatnoćom od 25% pravimo setalicu
    else if( p < 75 )
        return new Setalica;
    // inače (preostalih 25%) pravimo skakalicu
    else
        return new Skakalica;
}

Jedinka* Jedinka::NapraviJedinku( char c )
{
    if( c=='X' )
        return new Cekalica;
    else if( c=='P' )
        return new Puzalica;
    else if( c=='S' )
        return new Setalica;
    else if( c=='K' )
        return new Skakalica;
    else
        return 0;
}
```

Program možemo prevesti i isprobati.

Uočimo veoma važnu činjenicu – nijedna izmena u ovom koraku se nije odnosila na klasu `Igra`. To je posledica prethodne dobre pripreme, tokom koje smo sve ono što bi moglo da zavisi od specifičnih vrsta jedinki, premestili iz klase `Igra` u klasu `Jedinka`.

Korak 4 - Prazne ćelije

Iako smo nagovestili da ćemo upotrebljavati hijerarhije klasa da bismo izbegli ispitivanja vrste jedinke, još uvek imamo ispitivanja na više mesta u kodu.

Jedna vrsta ispitivanja je prisutna u metodima koji prave objekte hijerarhije klasa jedinki. To je neizbežno. Dok bismo kod pravljenja novih objekata mogli da napravimo određeni pomak (ostavljamo za vežbu), pri čitanju nemamo boljih rešenja.

Druga vrsta ispitivanja je neophodna pri pristupanju sadržaju ćelije. Pre nego što pokušamo da odgovornost za neki posao prenesemo na jedinku, moramo prvo da proverimo da li jedinka uopšte postoji, ili je vrednost ćelije samo prazan pokazivač. Koliko kod da nam se i ta provera može činiti neophodnom, pokazaćemo da možemo bez nje i da rezultat može biti prilično pojednostavljivanje nekih delova programa.

Umesto da prazne ćelije označavamo praznim pokazivačem, možemo hijerarhiji jedinki da dodamo klasu `PraznaĆelija`. Ako bismo se dosledno držali kriterijuma za izgradnju hijerarhije jedinki, našli bismo mnoge argumente protiv ovakvog postupka, jer prazna ćelija nikako nije specijalan slučaj jedinke. Međutim, u našem slučaju ima smisla malo preraditi hijerarhiju: ako bismo umesto hijerarhije jedinki napravili hijerarhiju sadržaja ćelija, onda bi i prazna ćelija tu našla svoje mesto. Suština je u tome da klase hijerarhije mogu imati različit smisao ako samu hijerarhiju posmatramo na različite načine. Mogli bismo da definišemo nove klase: `SadržajĆelije` i `PraznaĆelija`, i da pri tome klasu `Jedinka` izvedemo iz klase `SadržajĆelije`. Ipak, valja imati na umu da nova klasa ima smisao samo u kontekstu igre, a ne i u eventualno šire posmatranoj hijerarhiji jedinki. Zbog toga ćemo se ipak držati implementacije koja ne menja značajno napisanu hijerarhiju klasa jedinki.

Klasu `Jedinka` menjamo dodajući novi virtualni metod:

```
virtual bool Postoji() const
    { return true; }
```

Vrednost ovog metoda bi trebalo da bude `true` za sve stvarne vrste jedinki. Idealno pridržavanje principa OO programiranja bi nalagalo da u baznoj klasi metod ne bude implementiran. Međutim, ovde možemo napraviti mali izizetak: zato što znamo da će u svim klasama koje ikada budemo napravili ovaj metod vraćati `true`, osim u tačno jednoj posebnoj klasi u kojoj vraća `false`, implementiranjem metoda u baznoj klasi sugerišemo čitaocu koda da je naša posebna klasa stavljena u hijerarhiju iz tehničkih razloga i da ona zapravo i nije prava jedinka.

Hijerarhiji dodajemo novu klasu `NepostojećaJedinka`:

```
class NepostojecaJedinka : public Jedinka
{
public:
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const
        { ... }

    const Jedinka* NovaJedinka() const
        { return ...; }

    int Snaga() const
        { return 0; }
```

```
    char Izgled() const
        { return '.'; }

    bool Postoji() const
        { return false; }
};
```

Dok je jednostavno definisati ponašanje poslednja tri metoda, smisao ponašanja prva dva metoda je malo prikriven. Da bismo razumeli kako je potrebno da ih definišemo, moramo prvo razmotriti kada će oni biti pozivani za našu specijalnu *nepostojeću* jedinku.

Računanje koraka se upotrebljava za proveravanje da li jedinka preživljava. Kako to da tumačimo za našu nepostojeću jedinku? Ako pretpostavimo da su sve ćelije matrice inicijalno ispražnjene, nama je bitno da znamo da li na tom mestu postoji ćelija koja preživljava. Ali kako može da preživi jedinka koja ne postoji? Uz malo apstrakcije – može. Preživljavanjem nepostojeće ćelije možemo smatrati rađanje nove ćelije na njenom mestu. U skladu sa time, semantiku metoda `RacunanjeKoraka` možemo definisati tako da vraća rezultat `true` ako i samo ako je potrebno staviti novu jedinku u ćeliju table kojom predstavljammo narednu generaciju. Napišimo ovaj metod slično njegovoj implementaciji u klasi `Čekalica`:

```
class NepostojecaJedinka : public Jedinka
{
public:
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const
        {
        if( BrojSuseda( m, x, y ) == 3 ){
            nx = x;
            ny = y;
            return true;
            }
        else
            return false;
        }
...
};
```

Preostaje da napišemo metod `NovaJedinka`. U svim ostalim klasama on izračunava novi objekat istog tipa. To je zbog toga što se umesto neke jedinke koja postoji na staroj tabli stavlja neka nova jedinka *istog tipa* na novu tablu. Međutim, ako se rađa nova jedinka, tada umesto nepostojeće jedinke na staroj tabli valja vratiti novu jedinku čija se vrsta bira metodom slučajnog izbora. Već imamo na raspolaganju odgovarajući statički metod klase `Jedinka`, koji ćemo ovde primeniti:

```
    const Jedinka* NovaJedinka() const
        { return NapraviJedinku(); }
```

Statički metod za pravljenje nove jedinke na osnovu pročitane oznake sada bi trebalo dopuniti tako da umesto praznog pokazivača za prazne ćelije vraća novi objekat klase `NepostojećaJedinka`:

```
Jedinka* Jedinka::NapraviJedinku( char c )
{
    if( c=='X' )
        return new Cekalica;
    else if( c=='P' )
        return new Puzalica;
    else if( c=='S' )
        return new Setalica;
    else if( c=='K' )
        return new Skakalica;
    else
        return new NepostojecaJedinka;
}
```

Potrebno je da promenimo metode koji proveravaju da li su ćelije slobodne:

```
class Jedinka
{...
    int BrojSuseda(
            const Matrica<const Jedinka*>& m,
            int k, int v
        ) const
        {
        int sirina = m.Sirina();
        int visina = m.Visina();
        int n=0;
        for( int i=-1; i<2; i++ )
            for( int j=-1; j<2; j++ )
                if( m[(k+i+sirina)%sirina][(v+j+visina)%visina]
                        ->Postoji() )
                    n++;
        if( m[k][v]->Postoji() )
            n--;
        return n;
        }
};

class PokretnaJedinka : public Jedinka
{...
    int SlobodneCelije(
            const Matrica<const Jedinka*>& m,
            int x0, int y0,
            int pozicije[][2], int n, int prazna[]
            ) const
        {
        int npraznih = 0;
        int s = m.Sirina();
        int v = m.Visina();
        for( int i=0; i<n; i++ ){
            int x = (x0 + pozicije[i][0] + s) % s;
            int y = (y0 + pozicije[i][1] + v) % v;
            if( !m[x][y]->Postoji() )
                prazna[ npraznih++ ] = i;
            }
        return npraznih;
        }
};
```

Moramo da prođemo kroz sve metode klase `Igra` i da na svim mestima gde pristupamo sadržaju ćelija izmenimo ponašanje. Između ostalog, svuda gde pravimo „praznu“ matricu sada moramo tu matricu da popunimo objektima klase `NepostojećaJedinka`.

Počinjemo od metoda `Piši`. Više nije potrebno posebno razmatrati slučaj prazne ćelije:

```
void Pisi( ostream& ostr ) const
    {
    int s = _Konfiguracija.Sirina();
    int v = _Konfiguracija.Visina();
    ostr << s << ' ' << v;
    for( int i=0; i<v; i++ ){
        ostr << endl;
        for( int j=0; j<s; j++ )
            ostr << _Konfiguracija[j][i]->Izgled();
        }
    }
```

Najviše promena ima u metodu `IzračunavanjeGeneracije`. Najpre novu matricu ćelija moramo eksplicitno napuniti nepostojećim ćelijama. Sada u svakom slučaju pozivamo metod `RačunanjeKoraka`. Ako je rezultat metoda `true`, tada i samo tada pravimo novu jedinku. Snage postojeće i nove jedinke možemo uvek uporediti, jer u ćeliji postoji bar objekat klase `NepostojećaĆelija`.

```
void IzracunavanjeGeneracije()
    {
    int sirina = _Konfiguracija.Sirina();
    int visina = _Konfiguracija.Visina();
    Matrica<const Jedinka*> nova( visina, sirina );
    for( int i=0; i<sirina; i++ )
        for( int j=0; j<visina; j++ )
            nova[i][j] = new NepostojecaJedinka;
    for( int i=0; i<sirina; i++ )
        for( int j=0; j<visina; j++ ){
            int nx=i;
            int ny=j;
            if (
                _Konfiguracija[i][j]->RacunanjeKoraka(
                    _Konfiguracija, i, j, nx, ny
                    )
                ){
                const Jedinka* novaJedinka =
                                    _Konfiguracija[i][j]
                                        ->NovaJedinka();
                if( nova[nx][ny]->Snaga()
                            < novaJedinka->Snaga()
                    ){
                    delete nova[nx][ny];
                    nova[nx][ny] = novaJedinka;
                    }
                else
                    delete novaJedinka;
                }
            }
```

```
        ObrisiKonfiguraciju();
        _Konfiguracija = nova;
        }
```

Metod `BrojSuseda` klase `Igra` više nije potreban, pa ga brišemo. Načinjene izmene doprinose da kod programa bude malo jednostavniji i čitkiji. Program možemo prevesti i izvršiti.

Ovakav način rada, sa uvođenjem posebnih klasa ili posebnih objekata običnih klasa, umesto praznih pokazivača, naziva se tehnikom *praznih objekata*. Primenjuje se relativno često, posebno na mestima gde je zbog pouzdanosti ili iz drugih razloga neprihvatljivo da se pojavi prazan pokazivač.

Korak 5 - Muva-laki objekti

Ako uporedimo novi program sa rešenjem originalne igre Život, primetićemo da novi program radi mnogo više posla, jer umesto da samo dodeljuje vrednosti `true` i `false`, mora stalno da pravi i uklanja objekte. Rad sa objektima i hijerarhijama klasa nas je poštedeo upotrebe naredbi `switch` na više mesta u programu, ali moramo se upitati da li je plaćena cena previsoka?

Već i površan pogled na naše klase je dovoljan da uvidimo da one nemaju nikakav sadržaj. Jedino po čemu se međusobno razlikuju objekti jedinki jesu njihovi tipovi. Tim pre nam se može činiti da je upotreba hijerarhije klasa na ovom mestu bilo „lovljenje muve topom“.

Oba pristupa imaju svoje prednosti i mane. Prednost upotrebe konstanti je u odsustvu potrebe za stalnim pravljenjem i uklanjanjem objekata. Prednost rada sa objektima je u odsustvu složenih uslovnih naredbi, koje se teško održavaju. Da li postoji neka tehnika koja spaja najbolje iz oba pristupa? Zaista, takva tehnika postoji i veoma se često upotrebljava u sličnim situacijama.

Veoma zgodna osobina objekata koji pripadaju klasama bez podataka jeste da su svi objekti iste klase međusobno identični! Koliko god objekata klase `Skakalica` da napravimo, oni se neće međusobno razlikovati: svi imaju isti tip i isti sadržaj, pa se identično i ponašaju. Isti sadržaj? Pa, ako ga već nemaju, on tim pre ne može da bude faktor razlikovanja objekata. Isto važi i za ostale klase hijerarhije jedinki. Posledica te osobine je da umesto da pravimo veliki broj objekata, možemo bez ikakvih razlika više puta upotrebljavati pokazivače na jedan isti objekat, koji nikada nećemo uništavati. Na taj način se zadržavaju sve prednosti rada sa hijerarhijama klasa, a oslobađamo se opterećujućeg ponovljenog pravljenja i uništavanja velikog broja malih objekata.

Upotreba objekata bez sadržaja, ili objekata sa malim sadržajem, zbog male veličine objekata se naziva tehnikom *muva-lakih objekata*. U opštem slučaju se radi o pravljenju više prototipskih objekata, koji se zatim višestruko upotrebljavaju, bez uništavanja, sve do prestanka izvršavanja programa. Osnovni preduslov je da se objekti pri upotrebi *ne menjaju*. Ukoliko se radi o objektima koji imaju neki konstantan sadržaj, najčešće ima smisla praviti

više različitih objekata iste klase. Ako, kao u našem slučaju, objekti nemaju nikakav sadržaj, jasno je da nema potrebe praviti više od jednog objekta svake od klase.

Prvi korak u primeni muva-lakih objekata je da za svaku od klasa naše hijerarhije napravimo po jedan objekat, koji ćemo zatim upotrebljavati u programu:

```
class NepostojecaJedinka : public Jedinka
{...};
const Jedinka* muvaNepostojecaJedinka = new NepostojecaJedinka;
class Cekalica : public Jedinka
{...};
const Jedinka* muvaCekalica = new Cekalica;
class Puzalica : public PokretnaJedinka
{...};
const Jedinka* muvaPuzalica = new Puzalica;
class Setalica : public PokretnaJedinka
{...};
const Jedinka* muvaSetalica = new Setalica;
class Skakalica : public PokretnaJedinka
{...};
const Jedinka* muvaSkakalica = new Skakalica;
```

Svuda gde smo do sada pravili objekte, umesto toga upotrebljavamo napravljene muva-lake objekte. Počnimo od metoda `NovaJedinka` hijerarhije klasa jedinki. Kako u većini klasa metod `NovaJedinka` vraća objekat istog tipa, možemo ga u baznoj klasi `Jedinka` definisati tako da vrati pokazivač `this`:

```
virtual const Jedinka* NovaJedinka() const
    { return this; }
```

Zatim metode `NovaJedinka` možemo obrisati iz svih ostalih klasa jedinki, osim iz klase `NepostojećaJedinka`.

Sledeće mesto na kome smo pravili nove objekte jesu metodi `NapraviJedinku`. Umesto da pravimo nove objekte, sada i u njima koristimo napravljene muva-lake objekte. Označavamo da su rezultati konstantni:

```
const Jedinka* Jedinka::NapraviJedinku()
{
    int p = random(100);
    // sa verovatnoćom od 25% pravimo čekalicu
    if( p < 25 )
        return muvaCekalica;
    // sa verovatnoćom od 25% pravimo puzalicu
    else if( p < 50 )
        return muvaPuzalica;
    // sa verovatnoćom od 25% pravimo setalicu
    else if( p < 75 )
        return muvaSetalica;
```

```
        // inače (preostalih 25%) pravimo skakalicu
        else
            return muvaSkakalica;
    }
    const Jedinka* Jedinka::NapraviJedinku( char c )
    {
        if( c=='X' )
            return muvaCekalica;
        else if( c=='P' )
            return muvaPuzalica;
        else if( c=='S' )
            return muvaSetalica;
        else if( c=='K' )
            return muvaSkakalica;
        else
            return muvaNepostojecaJedinka;
    }
```

U metodu `Igra::IzračunavanjeGeneracije` smo morali da eksplicitno napunimo novu matricu objektima nepostojećih jedinki. Sada možemo iskoristiti konstruktor klase `Matrica` koji prihvata inicijalnu vrednost:

```
        void IzracunavanjeGeneracije()
            {
            int sirina = _Konfiguracija.Sirina();
            int visina = _Konfiguracija.Visina();
            Matrica<const Jedinka*> nova( visina, sirina,
                                          muvaNepostojecaJedinka );
            ...
            }
```

Nakon što smo zamenili sva pravljenja novih objekata, moramo obrisati i sva mesta na kojima se napravljeni objekti uništavaju. Prvi od njih je metod `ObrišiKonfiguraciju` klase `Igra`. On nam više nije potreban. Nije nam potreban ni destruktor klase `Igra`.

Nije nam potrebno ni uništavanje slabijih jedinki pri postavljanju snažnijih u metodu `IzračunavanjeGeneracije` klase `Igra`:

```
        void IzracunavanjeGeneracije()
            {
            ...
                    if (
                        _Konfiguracija[i][j]->RacunanjeKoraka(
                            _Konfiguracija, i, j, nx, ny
                            )
                        ){
                        const Jedinka* novaJedinka =
                                _Konfiguracija[i][j]->NovaJedinka();
                        if( nova[nx][ny]->Snaga()
                                    < novaJedinka->Snaga()
                            )
                            nova[nx][ny] = novaJedinka;
                        }
            ...
```

```
        }
```

Više nigde ne uništavamo jedinke, a pravimo ih samo na mestu na kome pravimo odgovarajuće primerke muva-lakih objekata.

Program možemo prevesti i isprobati.

9.3 Rešenje

```
#include <fstream>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

//---------------------------------------------------------------
//  Klasa Matrica
//---------------------------------------------------------------
template<class T>
class Matrica
{
public:
    //------------------------------
    //  Konstruktori
    Matrica( int visina =0, int sirina =0 )
        : _Kolone(sirina)
        {
        for( int i=0; i<sirina; i++ )
            _Kolone[i].resize(visina);
        }

    Matrica( int visina, int sirina, const T& x )
        : _Kolone(sirina)
        {
        for( int i=0; i<sirina; i++ ){
            _Kolone[i].resize(visina);
            for( int j=0; j<visina; j++ )
                _Kolone[i][j] = x;
            }
        }

    //------------------------------
    //  pristupanje elementima
    vector<T>& operator []( int i )
        { return _Kolone[i]; }
    const vector<T>& operator []( int i ) const
        { return _Kolone[i]; }

    int Sirina() const
        { return _Kolone.size(); }
    int Visina() const
        { return _Kolone.size() ? _Kolone[0].size() : 0; }

private:
```

```
    //-----------------------------
    //  podaci clanovi
    vector< vector< T > >   _Kolone;
};

//-----------------------------------------------------------------
//  Klase Jedinke
//-----------------------------------------------------------------
class Jedinka
{
public:
    virtual ~Jedinka()
        {}
    virtual const Jedinka* NovaJedinka() const
        { return this; }
    virtual bool Postoji() const
        { return true; }

    virtual bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const = 0;
    virtual int Snaga() const = 0;
    virtual char Izgled() const = 0;

    static const Jedinka* NapraviJedinku();
    static const Jedinka* NapraviJedinku( char c );

protected:
    int BrojSuseda(
            const Matrica<const Jedinka*>& m,
            int k, int v
        ) const
        {
        int sirina = m.Sirina();
        int visina = m.Visina();
        int n=0;
        for( int i=-1; i<2; i++ )
            for( int j=-1; j<2; j++ )
                if( m[(k+i+sirina)%sirina][(v+j+visina)%visina]
                        ->Postoji() )
                    n++;
        if( m[k][v]->Postoji() )
            n--;
        return n;
        }
};

class NepostojecaJedinka : public Jedinka
{
public:
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const
        {
        if( BrojSuseda( m, x, y ) == 3 ){
            nx = x;
```

```
                ny = y;
                return true;
                }
            else
                return false;
            }

        const Jedinka* NovaJedinka() const
            { return NapraviJedinku(); }

        int Snaga() const
            { return 0; }
        char Izgled() const
            { return '.'; }

        bool Postoji() const
            { return false; }
    };

    const Jedinka* muvaNepostojecaJedinka = new NepostojecaJedinka;

    class Cekalica : public Jedinka
    {
    public:
        bool RacunanjeKoraka(
                const Matrica<const Jedinka*>& m, int x, int y,
                int& nx, int& ny
            ) const
            {
            int n = BrojSuseda( m, x, y );
            bool prezivljava = n==2 || n==3;
            nx = x;
            ny = y;
            return prezivljava;
            }

        int Snaga() const
            { return 1; }
        char Izgled() const
            { return 'X'; }
    };

    const Jedinka* muvaCekalica = new Cekalica;

    class PokretnaJedinka : public Jedinka
    {
    protected:
        bool Pomeranje(
                const Matrica<const Jedinka*>& m, int x, int y,
                int& nx, int& ny, int pozicije[][2], int np
            ) const
            {
            int prazna[8];
            int n = SlobodneCelije( m, x, y, pozicije, np, prazna );
            if( n>0 ){
                int s = m.Sirina();
                int v = m.Visina();
                int i = random(n);
                nx = (x + pozicije[prazna[i]][0] + s) % s;
                ny = (y + pozicije[prazna[i]][1] + v) % v;
```

```
                return true;
                }
            else
                return false;
            }

        int SlobodneCelije(
                const Matrica<const Jedinka*>& m,
                int x0, int y0,
                int pozicije[][2], int n, int prazna[]
                ) const
            {
            int npraznih = 0;
            int s = m.Sirina();
            int v = m.Visina();
            for( int i=0; i<n; i++ ){
                int x = (x0 + pozicije[i][0] + s) % s;
                int y = (y0 + pozicije[i][1] + v) % v;
                if( !m[x][y]->Postoji() )
                    prazna[ npraznih++ ] = i;
                }
            return npraznih;
            }
    };

    class Puzalica : public PokretnaJedinka
    {
    public:
        bool RacunanjeKoraka(
                const Matrica<const Jedinka*>& m, int x, int y,
                int& nx, int& ny
            ) const
            {
            int pozicije[][2] = { {0,-1}, {0,1}, {1,0}, {-1,0} };
            return Pomeranje( m, x, y, nx, ny, pozicije, 4 );
            }

        int Snaga() const
            { return 2; }
        char Izgled() const
            { return 'P'; }
    };

    const Jedinka* muvaPuzalica = new Puzalica;

    class Setalica : public PokretnaJedinka
    {
    public:
        bool RacunanjeKoraka(
                const Matrica<const Jedinka*>& m, int x, int y,
                int& nx, int& ny
            ) const
            {
            int pozicije[][2] = { {-1,-1}, {-1,1}, {1,-1}, {1,1} };
            return Pomeranje( m, x, y, nx, ny, pozicije, 4 );
            }

        int Snaga() const
            { return 3; }
```

```
    char Izgled() const
        { return 'S'; }
};

const Jedinka* muvaSetalica = new Setalica;

class Skakalica : public PokretnaJedinka
{
public:
    bool RacunanjeKoraka(
            const Matrica<const Jedinka*>& m, int x, int y,
            int& nx, int& ny
        ) const
        {
        int pozicije[][2] = {
            {-2,-2}, {-2,0}, {-2,2}, {0,-2}, {0,2},
            {2,-2}, {2,0}, {2,2}
            };
        return Pomeranje( m, x, y, nx, ny, pozicije, 8 );
        }

    int Snaga() const
        { return 4; }
    char Izgled() const
        { return 'K'; }
};

const Jedinka* muvaSkakalica = new Skakalica;

const Jedinka* Jedinka::NapraviJedinku()
{
    int p = random(100);
    // sa verovatnoćom od 25% pravimo čekalicu
    if( p < 25 )
        return muvaCekalica;
    // sa verovatnoćom od 25% pravimo puzalicu
    else if( p < 50 )
        return muvaPuzalica;
    // sa verovatnoćom od 25% pravimo setalicu
    else if( p < 75 )
        return muvaSetalica;
    // inače (preostalih 25%) pravimo skakalicu
    else
        return muvaSkakalica;
}

const Jedinka* Jedinka::NapraviJedinku( char c )
{
    if( c=='X' )
        return muvaCekalica;
    else if( c=='P' )
        return muvaPuzalica;
    else if( c=='S' )
        return muvaSetalica;
    else if( c=='K' )
        return muvaSkakalica;
    else
        return muvaNepostojecaJedinka;
}
```

```
//---------------------------------------------------------------
//  Klasa Igra
//---------------------------------------------------------------
class Igra
{
public:
    //-----------------------------
    //  glavni metod za igranje
    void Odigravanje()
        {
        cout << "Upisite: " << endl
             << "   c <naziv datoteke> - "
             << "za citanje konfiguracije iz datoteke" << endl
             << "   p <naziv datoteke> - "
             << "za zapisivanje konfiguracije u datoteci" << endl
             << "   g                  - "
             << "za izracunavanje naredne generacije" << endl
             << "   k                  - za kraj" << endl;

        char c;
        do  {
            Pisi( cout );
            cout << endl;
            cin >> c;
            c = tolower(c);
            switch(c){
                case 'c':
                    CitanjeKonfiguracije();
                    break;
                case 'p':
                    PisanjeKonfiguracije();
                    break;
                case 'g':
                    IzracunavanjeGeneracije();
                    break;
                }
            } while( c!='k' );
        }

    //-----------------------------
    //  pisanje i citanje
    void Pisi( ostream& ostr ) const
        {
        int s = _Konfiguracija.Sirina();
        int v = _Konfiguracija.Visina();
        ostr << s << ' ' << v;
        for( int i=0; i<v; i++ ){
            ostr << endl;
            for( int j=0; j<s; j++ )
                ostr << _Konfiguracija[j][i]->Izgled();
            }
        }

    void Citaj( istream& istr )
        {
        int s, v;
        istr >> s >> v;
```

```
            Matrica<const Jedinka*> m(v,s);
            for( int i=0; i<v; i++ )
                for( int j=0; j<s; j++ ){
                    char c;
                    istr >> c;
                    m[j][i] = Jedinka::NapraviJedinku(c);
                    }
            _Konfiguracija = m;
            }

    private:
        void CitanjeKonfiguracije()
            {
            string s;
            cin >> s;
            ifstream f(s.c_str());
            Citaj(f);
            }

        void PisanjeKonfiguracije()
            {
            string s;
            cin >> s;
            ofstream f(s.c_str());
            Pisi(f);
            }

        void IzracunavanjeGeneracije()
            {
            int sirina = _Konfiguracija.Sirina();
            int visina = _Konfiguracija.Visina();
            Matrica<const Jedinka*> nova( visina, sirina,
                                          muvaNepostojecaJedinka );
            for( int i=0; i<sirina; i++ )
                for( int j=0; j<visina; j++ ){
                    int nx=i;
                    int ny=j;
                    if (_Konfiguracija[i][j]->RacunanjeKoraka(
                            _Konfiguracija, i, j, nx, ny )
                        ){
                        const Jedinka* novaJedinka =
                                _Konfiguracija[i][j]->NovaJedinka();
                        if( nova[nx][ny]->Snaga()
                                    < novaJedinka->Snaga()
                            )
                            nova[nx][ny] = novaJedinka;
                        }
                    }
            _Konfiguracija = nova;
            }

        //----------------------------
        //  podaci clanovi
        Matrica<const Jedinka*> _Konfiguracija;
    };

    ostream& operator << ( ostream& ostr, const Igra& t )
    {
        t.Pisi( ostr );
```

```
    return ostr;
}

istream& operator >> ( istream& istr, Igra& t )
{
    t.Citaj( istr );
    return istr;
}

//----------------------------------------------------------------
// Glavna funkcija programa
//----------------------------------------------------------------
main()
{
    Igra igra;
    ifstream f("konfiguracija.dat");
    f >> igra;
    igra.Odigravanje();

    return 0;
}
```

9.4 Rezime

Radi vežbanja, predlažemo da se izvede merenje performansi rešenja sa primenom objekata muva i pre primene te tehnike. Zbog toga što ispisivanje na standardnom izlazu može da stvori pogrešan utisak pri merenju performansi, sugerišemo da se tom prilikom privremeno onemogući ispisivanje izgleda matrice ćelija.

Posebno korisna vežba je implementiranje svih vrsta jedinki (ili bar svih vrsta pokretnih jedinki) jednom klasom, čijim se različitim muva-lakim objektima predstavljaju različite vrste jedinki.

Druga vežba može biti otklanjanje složenih odlučivanja u metodima `NapraviJedinku`. To se može postići pravljenjem pomoćnog niza pokazivača na muva-objekte. Neposredan pristup elementima nizova je i jednostavan i efikasan, što posebno dolazi do izražaja kada se „pravi“ veliki broj različitih objekata.

U ovom primeru smo eksperimentisali sa različitim vrstama jedinki. Drugi mogući smer za eksperimentisanje je da se uvedu, na primer, jedinke iste vrste, ali različite boje.

Deo III

Dodaci

Treći deo knjige predstavljaju dodaci. Izloženi su neki od najvažnijih elemenata standardne biblioteke programskog jezika C++. Najviše pažnje je posvećeno tokovima i kolekcijama podataka.

Navedeni zadaci za vežbu su po složenosti slični rešenim primerima. Nakon upoznavanja programskog jezika i pažljivog praćenja rešavanja izloženih primera, čitaoci bi trebalo da budu u stanju da samostalno reše priložene zadatke.

Indeks pojmova sadrži pregled ključnih reči i naziva upotrebljavanih klasa, metoda i algoritama standardne biblioteke.

10 - Standardna biblioteka

10.1 Osnovni koncepti

Standardna biblioteka programskog jezika C++ je u velikoj meri doprinela širokoj primenjivosti jezika. Ona nije toliko sveobuhvatna kao, na primer, biblioteka klasa programskog jezika Java. Standardna biblioteka programskog jezika C++ je prilično sažeta, ali se odlikuje visokim stepenom apstrakcije, što omogućava da se isti elementi biblioteke upotrebljavaju u potpuno različitim kontekstima. Bez obzira na visok stepen apstrakcije, čitava biblioteka je strogo tipizirana, što otklanja potrebu za proveravanjem tipova podataka u toku izvršavanja programa.

Iako je visok stepen apstrakcije koda obično suprotstavljen aspektu efikasnosti, u slučaju standardne biblioteke programskog jezika C++ to nikako ne stoji – ona se odlikuje izuzetnom efikasnošću. Na tome u velikoj meri možemo da zahvalimo činjenici da je biblioteka implementirana primenom šablona klasa i funkcija, pa se elementi biblioteke posebno prevode, a samim tim i optimizuju, za svaki konkretan tip podataka za koji se upotrebljavaju.

Pored pomenutih kvaliteta, biblioteka ima i neke slabosti, koje otežavaju njenu primenu početnicima. Najčešći uzrok problema je odsustvo provera ispravnosti argumenata metoda i funkcija. To je cena koja je plaćena radi postizanja visoke efikasnosti, što ne bi trebalo da iznenađuje jer je na sličnim kompromisima definisan čitav programski jezik. Odgovarajući problemi su posebno česti pri radi sa iteratorima, jer upotreba nekih metoda kolekcija podataka ima za posledicu da postojeći iteratori mogu postati neažurni, a time i neispravni.

U nekim situacijama može zasmetati i činjenica da sama biblioteka nije objektno orijentisana, ali to ne bi trebalo da pravi probleme, jer je biblioteka ortogonalna u odnosu na objektno orijentisano programiranje – niti ga onemogućava niti ga podstiče.

Biblioteka obuhvata nekoliko osnovnih celina:

- *Koncepti iteratora i funkcionala* određuju način definisanja i upotrebe ostalih elemenata biblioteke;

- *Posebne tehnike* i *pomoćne klase* predstavljaju skup koncepata i odgovarajućih pomoćnih klasa koje značajno podižu nivo apstraktnosti biblioteke. U takve tehnike spada, na primer, apstrahovanje postupka alociranja elemenata kolekcija, koje je u potpunosti prilagodljivo eventualnim specifičnim potrebama programera;
- *Kolekcije podataka* (engl. *containers*) predstavljaju jedan od najvažnijih delova biblioteke;
- *Algoritmi* predstavljaju kolekciju visoko apstraktnih šablona funkcija. Intenzivno upotrebljavaju iteratore, što omogućava da operišu nad različitim vrstama kolekcija podataka;
- *Biblioteka tokova* predstavlja hijerarhiju klasa za podršku čitanja i pisanja primenom apstraktnih uređaja. Biblioteka omogućava istovetan rad sa veoma raznolikim uređajima i podacima i definiše minimalne principe kojih se autori novih klasa moraju pridržavati kako bi se i one upotrebljavale na isti način;
- *Lokali* su skup elemenata koji podržavaju programiranje nezavisno od konkretnih nacionalnih potreba, tako da im se programi jednostavno prilagođavaju.

Sva imena u biblioteci su definisana u prostoru imena `std`. Zbog toga je uobičajeno da programi koji koriste elemente standardne biblioteke započinju deklaracijom:

```
using namespace std;
```

U ovom dodatku su predstavljeni elementi standardne biblioteke. Najpre su predstavljeni koncepti iteratora i funkcionala, a zatim i većina kolekcija. Posle toga sledi kratak pregled algoritama i biblioteke tokova.

Nije predstavljena čitava biblioteka već samo neki od njenih najvažnijih elemenata. Za uspešnu upotrebu standardne biblioteke je najčešće dovoljno imati osnovna znanja o primenjenim konceptima i odgovarajućim mogućnostima primene. Postepeno, kroz upotrebu, važniji delovi biblioteke se sami nameću i lako pamte. Potpunije predstavljanje standardne biblioteke se može pročitati u knjigama [Stroustrup 2000] i [Lippman 2005]. Veoma detaljan pregled standardne biblioteke, sa diskusijom i primerima upotrebe, izložen je u [Josuttis 1999].

10.2 Iteratori

Koncept

Iteratori predstavljaju jedan od osnovnih koncepata na kojima počivaju kolekcije standardne biblioteke programskog jezika C++. Iteratori su apstrakcija pokazivača, koja omogućava da se po istim principima obilaze elementi proizvoljne kolekcije.

Iteratore je najlakše razumeti pomoću nekoliko primera. Pretpostavimo da imamo na raspolaganju niz celih brojeva `niz`, da nam je poznata njegova veličina `veličina` i da je potrebno proslediti u standardni tok sve elemente tog niza. Jedan način bi bio da se koriste indeksi:

```
for( unsigned i=0; i<veličina; i++ )
    cout << niz[i] << endl;
```

Drugi način je da se umesto indeksa koriste pokazivači. Zbog toga što je u programskom jeziku C++ zapis `niz[i]` ekvivalentan zapisu `*(niz+i)`, prethodni kod možemo zapisati i kao:

```
for( unsigned i=0; i<veličina; i++ )
    cout << *(niz+i) << endl;
```

Upotrebu indeksa možemo potpuno zaobići tako što ćemo umesto brojača upotrebiti pokazivač koji će pokazivati, redom, na sve elemente niza:

```
for( int* i=niz; i<niz+veličina; i++ )
    cout << (*i) << endl;
```

Pri tome pretpostavljamo da pokazivač `niz` pokazuje na prvi element niza, da je `niz+veličina-1` pokazivač na poslednji element niza, a da je `niz+veličina` vrednost koju će pokazivač `i` imati nakon što bude obrađen poslednji element niza. Drugim rečima, znamo da iteracija po `i` mora započeti od `niz`, a da je ponavljanje potrebno prekinuti kada dostignemo `niz+veličina`. Kao dodatno uopštenje, u uslovu kojim proveravamo da li je potrebno nastaviti ponavljanje, umesto operatora `<` ćemo upotrebiti operator `!=`:

```
int* početak = niz;
int* kraj = niz + veličina;
for( int* i=početak; i!=kraj; i++ )
    cout << (*i) << endl;
```

Da bismo mogli na sličan način obraditi elemente neke druge kolekcije, a ne samo niza, potrebno je da imamo odgovarajuću zamenu za pokazivače, koja bi omogućila poređenje, povećavanje i pristupanje elementima, kao i odgovarajuće metode u kolekcijama koji bi izračunavali početne i krajnje vrednosti takvih „pokazivača“. Upravo na tom principu se definišu iteratori, kao apstrakcija ovakvog koncepta obrađivanja elemenata kolekcija.

Standardna biblioteka programskog jezika C++ je definisana tako da se sve kolekcije i veliki broj algoritama zasnivaju na iteratorima. Osnovni principi implementacije i upotrebe iteratora u standardnoj biblioteci programskog jezika C++ su:

- U okviru svake apstraktne kolekcije definiše se klasa `iterator`, kao apstrakcija pokazivača na elemente kolekcije, koja ima bar metode:
  - operator uvećavanja `++`, koji menja vrednost iteratora tako da ukazuje na naredni element kolekcije;
  - operator provere jednakosti `==`, koji proverava da li je iterator jednak datom drugom iteratoru istog tipa;
  - operator provere različitosti `!=`, koji proverava da li je iterator različit od datog drugog iteratora istog tipa;
  - operator dereferenciranja `*` (prefiksni unarni), koji izračunava referencu na element kolekcije na koji se odnosi iterator;

  - operator dereferenciranja `->` (postfiksni unarni), koji izračunava pokazivač na element na koji se odnosi iterator.
- U okviru svake apstraktne kolekcije definišu se metodi:
  - `begin()` – izračunava početni iterator, koji se odnosi na prvi element kolekcije;
  - `end()` – izračunava završni iterator, koji se odnosi na „prvi element iza poslednjeg elementa kolekcije“, tj. na vrednost koju bi iterator trebalo da ima nakon što budu obrađeni svi elementi.

Ako pretpostavimo da naša kolekcija celih brojeva `niz` više nije klasičan niz, nasleđen iz programskog jezika C, nego neka kolekcija `K` koja je definisana u skladu sa navedenim principima standardne biblioteke programskog jezika C++, tada bismo sve elemente kolekcije mogli obraditi na sledeći način:

```
K::iterator početak = niz.begin();
K::iterator kraj = niz.end();
for( K::iterator i=početak; i!=kraj; i++ )
    cout << (*i) << endl;
```

Znajući da standardna biblioteka raspolaže većim brojem različitih kolekcija, kao što su vektor, lista, stek, skup, katalog i druge, lako je uvideti značaj definisanja jednobraznog modela za obrađivanje elemenata svih tipova kolekcija. Predstavljeni koncept iteratora predstavlja upravo takav mehanizam. Iteratori se primenjuju na isti način za sve tipove kolekcija. Veoma važna osobina ovako definisanih iteratora je da svi šabloni funkcija standardne biblioteke, koji su napisani kao da su argumenti iteratori, rade jednako dobro sa pokazivačima na elemente običnih nizova.

Vrste iteratora

Pored običnih iteratora, koji su prethodno opisani, i koji omogućavaju kako čitanje tako i menjanje elemenata kolekcije, postoje i konstantni iteratori koji omogućavaju samo čitanje elemenata kolekcije. Konstantni iteratori se definišu u klasama kolekcija pod imenom `const_iterator`. Konstantne kolekcije raspolažu metodima `begin` i `end` koji izračunavaju konstantne početne i završne iteratore.

Da bi bio moguć obilazak elemenata kolekcija i u suprotnom smeru, od poslednjeg prema prvom, kolekcije raspolažu i tzv. obrnutim iteratorima (`reverse_iterator`) i konstantnim obrnutim iteratorima (`const_reverse_iterator`), kao i odgovarajućim metodima za izračunavanje njihove početne (`rbegin`) i završne vrednosti (`rend`). Primetimo da se obrnuti iteratori upotrebljavaju na potpuno isti način kao i obični, tj. uz primenu operatora `++`. Tako bi se, na primer, ispisivanje elemenata kolekcije `K` u obrnutom redosledu moglo implementirati sledećim segmentom programa:

```
K::const_reverse_iterator početak = niz.rbegin();
K::const_reverse_iterator kraj = niz.rend();
for( K::const_reverse_iterator i=početak; i!=kraj; i++ )
    cout << (*i) << endl;
```

U literaturi se može naići na sledeću podelu iteratora:

Vrsta iteratora	Pristup	Smer kretanja
ulazni (*input iterator*)	samo čitanje	unapred
izlazni (*output iterator*)	samo pisanje	unapred
jednosmerni (*forward iterator*)	čitanje i pisanje	unapred
dvosmerni (*bidirectional iterator*)	čitanje i pisanje	u oba smera
indeksni (*random access iterator*)	čitanje i pisanje	proizvoljan pristup
umećući (*insert iterator*)	pisanje	unapred

Tablica 1: Vrste iteratora

Standardna bibiloteka definiše veći broj algoritama (tj. šablona funkcija) koji za argumente imaju jedan ili više tipova iteratora. Pri tome:

- Ulazni iteratori se koriste za izdvajanje većeg broja podataka iz ulaznih tokova. Svi iteratori definisani u kolekcijama se mogu upotrebljavati i kao ulazni iteratori.
- Izlazni se upotrebljavaju za pisanje u tok, ili za umetanje elemenata u kolekciju. Svi iteratori definisani u kolekcijama, osim konstantnih, se mogu upotrebljavati kao izlazni iteratori.
- Svi iteratori definisani u kolekcijama, osim konstantnih, mogu da se upotrebljavaju kao jednosmerni iteratori.
- Dvosmerni iteratori su slični jednosmernim, s tim da podržavaju i primenu operatora umanjivanja iteratora `--`.
- Indeksni iteratori omogućavaju promenu u jednom ili oba smera za proizvoljan broj mesta, tj. podržavaju operatore sabiranja i oduzimanja sa celim brojevima.
- Umećući iteratori izvode automatsko umetanje elemenata u kolekciju pri dodeljivanju vrednosti dereferenciranom elementu.

10.3 Funkcionali

U jednom broju funkcija ili metoda klasa standardne biblioteke potrebno je da se kao argument navede funkcija koja radi neki posao ili izračunava neku vrednost. Posebno se često upotrebljavaju funkcije koje proveravaju da li je ispunjen neki uslov. Takve funkcije se nazivaju *predikati*. Obično se upotrebljavaju unarni ili binarni predikati ili druge funkcije, ali ima i složenijih slučajeva.

Zbog toga što su funkcije nepromenljivi „objekti“ u programskom jeziku C++, one nisu uvek upotrebljive. Na primer, standardna biblioteka sadrži šablon funkcije `count_if`, koja broji za koliko elemenata neke kolekcije je zadovoljen dati unarni predikat `pred` (tj. vraća `true`) i zapisuje dobijen broj u promenljivu `n`:

```
vector<int> v;
```

```
...
int n = 0;
count_if( v.begin(), v.end(), pred, n );
```

Ako znamo unapred koji je uslov potrebno proveriti, sve je u redu, jer možemo napisati odgovarajuću funkciju `pred`. Međutim, ako je potrebno da korisnik upiše vrednost nekog broja za koji želimo da izbrojimo koliko ima elemenata niza `v` koji su, na primer, veći od njega, onda imamo problem.

Programski jezik C++ omogućava nam da pišemo tzv. *funkcionale*. Funkcional je objekat koji *ume* da primeni operator aplikacije, tj. da nešto *izračuna* za date argumente. U narednom primeru, klasa `VeciOd` omogućava da se objekat upotrebljava za proveravanje da li je dati broj veći od broja koji je naveden pri konstrukciji objekta:

```
class VeciOd
{
public:
    VeciOd( int n ) : _N(n) {}
    bool operator() ( int x ) const
        { return x > _N; }
private:
    const int _N;
};
```

Zbog toga što su funkcije koje čine standardnu biblioteku definisane u obliku šablona, moguće je umesto funkcije navesti objekat funkcional:

```
vector<int> v;
...
int k;
cin >> k;
VeciOd veciOdK(k);
int n = 0;
count_if( v.begin(), v.end(), veciOdK, n );
```

Štaviše, odgovarajući neimenovani objekat možemo napraviti na licu mesta:

```
count_if( v.begin(), v.end(), VeciOd(k), n );
```

Klasa `VeciOd` je primer funkcionala. Klase koje predstavljaju funkcionale obično imaju sasvim jednostavnu strukturu, kao u prethodnom primeru: konstruktor objekta služi da inicijalizuje članove podatke, a `operator()` izračunava odgovarajuću funkciju.

Funkcije standardne biblioteke najčešće upotrebljavaju funkcionale sledećih oblika:

- unarna funkcija – izračunava vrednost na osnovu jednog argumenta;
- binarna funkcija – izračunava vrednost na osnovu dva argumenta;
- unarni predikat – proverava da li vrednost zadovoljava neki uslov;
- upoređivač – proverava da li je prvi argument strogo manji od drugog (šta god to značilo u kontekstu konkretnog upoređivača);
- transformator – unarni funkcional koji na određen način menja (transformiše) argument prenesen po referenci.

Posebnu vrstu funkcionala čine oni koji imaju promenljivo stanje. Njihovo stanje se obično menja pri izračunavanju i to stanje može da utiče na naredna izračunavanja.

Standardna biblioteka obuhvata više funkcionala i pomoćnih klasa, kao i funkcija koje omogućavaju građenje funkcionala od funkcija, metoda ili postojećih funkcionala. Više informacija se može pronaći u [Josuttis 1999].

10.4 Pomoćne klase

10.4.1 Uređeni par – struktura pair

Šablon strukture `pair` predstavlja uređeni par koji se sastoji od elemenata proizvoljna dva tipa.

Zaglavlja

Šablon `pair` definisan je u zaglavlju `utility`.

Parametri šablona

Šablon `pair` je definisan sa dva parametra:

```
template <
    class T1,
    class T2
    >
struct pair;
```

Parametar `T1` određuje tip prvog elementa para, a tip `T2` tip drugog elementa para. Ne postoje ograničenja vezana za ove tipove.

Osnovne operacije

Struktura `pair` je sasvim jednostavna i predstavićemo sve njene članove i metode:

- Na raspolaganju su podrazumevani konstruktor i konstruktor na osnovu dva argumenta kojima se inicijalizuju elementi para, kao i konstruktor za inicijalizovanje para parom nekog drugog tipa čiji se odgovarajući elementi mogu konvertovati u potrebne tipove elemenata:

```
pair()
pair(const T1& x, const T2& y)
template <class V, class U>
    pair( const pair <V, U>& p )
```

- Elementima para se pristupa neposredno, koristeći članove podatke:

```
T1 first
T2 second
```

Van same strukture `pair` definisani su operatori poređenja ==, !=, <, <=, >, >= koji porede parove tako što porede najpre prve elemente, a ako su oni jednaki onda i druge. Ukoliko se koriste operacije poređenja, u zavisnosti od operacije poređenja, postavlja se bar jedno ili oba naredna ograničenja:

- Mora biti definisan jedan od operatora
  - `bool T1::operator<(const T1&) const`
  - `bool operator<(const T1&, const T1&)`

  i jedan od operatora
  - `bool T2::operator<(const T2&) const`
  - `bool operator<(const T2&, const T2&)`
- Mora biti definisan jedan od operatora
  - `bool T1::operator==(const T1&) const`
  - `bool operator==(const T1&, const T1&)`

  i jedan od operatora
  - `bool T2::operator==(const T2&) const`
  - `bool operator==(const T2&, const T2&)`

Povezani delovi standardne biblioteke

Struktura pair je veoma važna za funkcionisanje kataloga (videti *10.8.3 Katalog*, na strani 374 i *10.8.4 Katalog sa ponavljanjem ključeva*, na strani 377).

10.5 Kolekcije podataka

Standardna biblioteka programskog jezika C++ obuhvata nekoliko osnovnih kolekcija podataka. Kolekcije standardne biblioteke počivaju na nekoliko osnovnih principa:

- sve kolekcije su implementirane kao šabloni klasa;
- elementi kolekcija se obilaze pomoću iteratora;
- kolekcije su strogo tipizirane – svi elementi jedne kolekcije moraju imati isti tip;
- interfejsi kolekcija su međusobno veoma slični – metodi koji obavljaju isti ili skoro isti posao imaju ista imena i slične ili iste argumente.

Podržane kolekcije se mogu podeliti na sekvencijalne, uređene i apstraktne kolekcije.

Sekvencijalne kolekcije su lista (`list`), vektor (`vector`) i dek (`deque`). Vektor logičkih vrednosti `vector<bool>` se obično razmatra kao posebna kolekcija jer ima neke specifične osobine u odnosu na ostale vektore, a koje se odnose na efikasnije pakovanje sadržaja tako da zauzimaju što manje mesta. U sekvencijalne kolekcije se obično ubraja i klasa `string`, jer predstavlja sekvencu karaktera.

Osnovne uređene kolekcije su skup (`set`) i katalog (`map`). Jednu podvrstu uređenih kolekcija predstavljaju skup sa ponavljanjem (`multiset`) i katalog sa ponovljenim ključevima (`multimap`). Skup bitova (`bitset`) je po mnogo čemu specifična kolekcija, koja ima osobine i uređenih i sekvencijalnih kolekcija.

Apstraktne kolekcije su kolekcije čija apstraktna struktura oslikava njihovu specifičnu primenu, dok interna struktura može biti predmet izbora. Takve kolekcije su stek (`stack`), red (`queue`) i red sa prioritetom (`priority_queue`). One se mogu definisati nad proizvoljnim

sekvencijalnim kolekcijama, a zavisno od pretpostavljenih okolnosti u kojima će se upotrebljavati.

Podržane kolekcije se odlikuju relativno visokim nivoom apstrakcije, što ima za posledicu da je mali broj različitih tipova kolekcija sasvim dovoljan za praktično sve potrebe. Svakako da postoje situacije u kojima će programeri morati da pišu sopstvene kolekcije podataka, ali daleko ređe nego što bi se to na prvi pogled moglo očekivati. Iako nisu podržane neke od kolekcija koje se često upotrebljavaju u programiranju, one najčešće ne moraju da se implementiraju jer postojeće kolekcije u potpunosti zadovoljavaju potrebe. Na primer:

- binarno drvo – Binarno drvo se obično upotrebljava za implementaciju skupova ili kao osnova za brzo binarno pretraživanje. Za veliku većinu problema koji se rešavaju pomoću binarnog drveta, mogu se u potpunosti uspešno primeniti kolekcije `set` i/ili `map`.
- višedimenzioni nizovi – Postojeći nizovi (`vector`) mogu imati za elemente bilo šta, pa i druge nizove. Na taj način se mogu praviti nizovi sa proizvoljnim konačnim brojem dimenzija.
- graf – Grafovi se efikasno i veoma jednostavno implementiraju primenom kataloga (`map`). Na isti način se mogu implementirati i složena drveta.

Veoma važna karakteristika standardne biblioteke je njena izuzetno visoka efikasnost. Bez obzira na visok nivo apstrakcije, činjenica da se implementira primenom šablona klasa obezbeđuje performanse koje se teško mogu dostići čak i pisanjem specifičnih kolekcija prilagođenih konkretnim tipovima podataka.

Pri izboru kolekcije koja bi mogla najbolje poslužiti u konkretnoj situaciji, možemo se poslužiti narednim „upitnikom“:

- *Kako se pristupa elementima kolekcije?*
  Ako je važno da imamo efikasan neposredan pristup putem indeksa, onda je najverovatnije da pravi izbor predstavljaju vektor ili dek.
- *Da li se često proverava da li kolekcija sadrži neku vrednost?*
  Pozitivan odgovor nagoveštava da su uređene kolekcije skup i katalog daleko bolji izbor.
- *Da li se elementi kolekcije mogu poreditи među sobom?*
  Ako ne mogu, onda se ne mogu upotrebljavati uređene kolekcije.
- *Da li se često traži ili upotrebljava najveći/najmanji element kolekcije?*
  U slučaju pozitivnog odgovora bi trebalo razmotriti upotrebu reda sa prioritetom.
- *Gde se dodaju novi elementi?*
  Ako se dodaju isključivo na kraj sekvence, pravi izbor je vektor. Ako se dodaju i na početak, bolje je upotrebiti dek. Ako se dodaju i usred sekvence, a bez pretpostavljenog uređenja, onda je pravi izbor lista. Ukoliko se dodaju tako da se održava uređenje, onda je najbolje upotrebiti neku uređenu kolekciju.

- *Da li je važan poredak elemenata?*
  Ako je važno da sadržaj kolekcije bude stalno uređen, obično ćemo se odlučiti za neku od uređenih kolekcija. Ako, sa druge strane, uređenje nije značajno, ili je značajno samo povremeno, tada su sekvencijalne kolekcije verovatno pogodnije, zbog svoje veće efikasnosti i manjeg utroška prostora. Ako je poredak elemenata u vezi sa redosledom dodavanja ili korišćenja elemenata, onda je potrebno razmotriti upotrebu sekvencijalnih kolekcija `list` i `vector` ili neke od apstraktnih kolekcija.
- *Da li će se veličina kolekcije često i značajno menjati tokom upotrebe?*
  Ako hoće, onda vektor i dek nisu pravi izbor, jer oni zadržavaju maksimalnu dostignutu veličinu sve dok se alcoirani prostor ne smanji uništavanjem kolekcije ili eksplicitnim smanjivanjem od strane programera. Tada lista i skup predstavljaju bolji izbor. Sa druge strane, ako se ne očekuje značajno i često menjanje veličine kolekcije, verovatno je bolje odlučiti se za vektor ili dek, jer u takvim uslovima koriste manje memorije i pružaju bolje performanse. Posebno, ako se veličina može proceniti, bilo unapred ili u toku rada, onda je vektor skoro sigurno najbolji izbor.
- *Da li se često spajaju dve kolekcije?*
  Ako je tako, dobar izbor predstavljaju lista i skup, jer podržavaju veoma efikasno spajanje. Ukoliko se pri tome održava uređenje, bolji je izbor skup, u suprotnom ćemo pre koristiti listu.

Moglo bi se oblikovati još kriterijuma, ali navedeni kriterijumi su obično sasvim dovoljni da se načini ispravan izbor.

U narednim odeljcima su navedene osnovne osobine i najvažniji metodi najčešće upotrebljavanih tipova kolekcija.

10.6 Sekvencijalne kolekcije

Sekvencijalne kolekcije predstavljaju najčešće upotrebljavan vid kolekcija. Kod sekvencijalnih kolekcija se pretpostavlja da su svi elementi poređani u sekvencu i uređeni u nekom poretku. Jedini tip sekvencijalnih kolekcija koji je definisan u okviru specifikacije programskog jezika C++ jeste niz. Programski jezik C++ podržava upotrebu nizova na isti način kao i programski jezik C. Iako su takvi nizovi veoma efikasni, njihova upotreba je prilično nefleksibilna, te često ne zadovoljava potrebe programera. Zbog toga standardna biblioteka programskog jezika C++ definiše sekvencijalne kolekcije `list`, `vector` i `deque`.

Sve sekvencijalne kolekcije imaju veoma slične interfejse, pa je često relativno lako zameniti upotrebljavanu sekvencijalnu kolekciju nekom drugom. Da bi se takva zamena učinila još lakšom, često se u programima definišu pomoćni tipovi podataka, kao recimo:

```
typedef list<int> sekvencaCelihBrojeva;
```

pa se u slučaju potrebe lista zamenjuje nekom drugom kolekcijom, npr. vektorom, samo ispravljanjem odgovarajuće definicije. Naravno, svaka od sekvencijalnih kolekcija ima neke specifičnosti, pa se i mesta na kojima se upotrebljavaju specifični metodi moraju prilagođavati.

Kako niske mogu da se posmatraju kao vid sekvencijalne kolekcije znakova, i interfejs klase `string` standardne biblioteke je oblikovan tako da u značajnoj meri podseća na sekvencijalne kolekcije, a pre svega na interfejs klase `vector`.

Sekvencijalne kolekcije se upotrebljavaju kada je važan redoled u kome se elementi čuvaju i obrađuju. Naredna tablica sadrži pregled osnovnih osobina sekvencijalnih kolekcija, na osnovu kojih se obično odlučuje o tome koja je od ponuđenih kolekcija najpogodnija za upotrebu.

	lista	**vektor**	**dek**
efikasan neposredan pristup	ne	da	da
efikasno dodavanje na kraj i uklanjanje sa kraja sekvence	da	da	da
efikasno dodavanje na početak i uklanjanje sa početka sekvence	da	ne	da
efikasno dodavanje usred kolekcije i brisanje iz sredine kolekcije	da	ne	ne
dobar izbor ako je veličinu moguće unapred relativno tačno proceniti	ne	da	da
pogodna u slučaju čestih i drastičnih promena veličine	da	ne	ne

Tablica 2: Osobine sekvencijalnih kolekcija

10.6.1 Lista

Šablon klase `list` standardne biblioteke programskog jezika C++ predstavlja sekvencijalnu kolekciju u kojoj su elementi poređani tako da svaki element *zna* koji je sledeći element. Liste omogućavaju efikasan redni pristup elementima, ali ne i efikasan pristup proizvoljnom elementu. Podržavaju efikasno umetanje novog elementa ispred datog postojećeg elementa, uključujući i dodavanje na početak i kraj liste, kao i efikasno uklanjanje proizvoljnog datog elementa. Dodavanje elemenata usred liste i brisanje elemenata iz sredine liste ne zahtevaju fizičko premeštanje drugih elemenata.

Zaglavlja

Šablon klase `list` definisan je u zaglavlju `list`.

Parametri šablona

Šablon `list` je definisan sa dva parametra:

```
template<
    class T,
    class Allocator = allocator<T>
    >
class list;
```

Parametar `T` određuje tip elemenata liste. Da bi se mogla inicijalno postavljati ili dinamički povećavati veličina liste, neophodno je da klasa `T` ima konstruktor bez argumenata. Ukoliko

ga nema, lista se može povećavati samo dodavanjem datih pojedinačnih elemenata. Ne postoje druga ograničenja vezana za ovaj tip.

Parametar `Allocator` određuje način alociranja novih objekata klase `T`. Ukoliko se ostavi podrazumevani parametar, prostor će se alocirati na uobičajen način, primenom izraza `new`.

Liste se najčešće upotrebljavaju samo uz navođenje tipa elemenata.

Osnovne operacije

Najvažniji metodi šablona klase `list` su:

- Konstruktor prazne liste:

```
list()
```

- Konstruktor liste date veličine. Elementi se inicijalizuju primenom konstruktora bez argumenata ili kopiranjem datog objekta `x` klase `T`:

```
list( size_type n, const T& x = T() )
```

- Metodi koji vraćaju početne i završne iteratore. Pored navedenih metoda postoje i njihove konstantne verzije, koje za konstantne kolekcije vraćaju konstantne početne i završne iteratore:

```
iterator begin()
iterator end()
reverse_iterator rbegin()
reverse_iterator rend()
```

- Izračunavanje veličine liste i najveće dopuštene veličine liste:

```
size_type size() const
size_type max_size() const
```

- Promena veličine liste. Ako se lista smanjuje, višak elemenata se uklanja. Ako se lista povećava, novi elementi se prave primenom konstruktora bez argumenata ili kopiranjem datog objekta `x` klase `T`:

```
void resize( size_type n )
void resize( size_type n, const T& x )
```

- Metodi za dodavanje elementa na početak ili kraj liste i brisanje elementa sa početka ili kraja liste. Menjaju veličinu liste. Ne sme se brisati element prazne liste.

```
void push_front( const T& x )
void push_back( const T& x )
void pop_front()
void pop_back()
```

- Metodi za pristupanje prvom i poslednjem elementu liste. Postoje u verzijama za konstantne i nekonstantne liste:

```
const T& front() const
T& front()
const T& back() const
T& back()
```

- Provera da li je lista prazna:

```
bool empty() const
```

- Pražnjenje liste:

```
void clear()
```

- Metodi za umetanje elemenata u listu. Prvi metod umeće kopiju elementa `x` pre elementa na koji ukazuje iterator `position`. Rezultat je iterator koji ukazuje na umetnut element. Drugi metod umeće `n` kopija elementa `x` pre elementa na koji ukazuje iterator `position`. Treći metod umeće ispred elementa na koji ukazuje iterator `position` sve elemente neke kolekcije (bilo kog tipa), koji pripadaju opsegu datih iteratora.

```
iterator insert( iterator position, const T& x )
void insert( iterator position, size_type n, const T& x)
template <class InputIterator>
    void insert( iterator position,
                 InputIterator first, InputIterator last )
```

- Metodi za brisanje elemenata liste. Brišu, redom, element na koji ukazuje dati iterator ili sve elemente zahvaćene opsegom datih iteratora. Rezultat je iterator na element neposredno iza poslednjeg obrisanog, ili `end()` ako je obrisan poslednji element liste.

```
iterator erase(iterator position)
iterator erase(iterator first, iterator last)
```

- Metod za efikasnu razmenu kompletnog sadržaja dveju lista istog tipa:

```
void swap( list<T,Allocator>& l )
```

- Metod koji u uređenu listu dodaje elemente date uređene liste `x`, održavajući pri tome uređenje. Prva verzija pretpostavlja da je uređenje definisano operatorom `<`, a druga da se za poređenje upotrebljava dati funkcional `cmp`.

```
void merge( list<T,Allocator>& x )
template<class Compare>
    void merge( list<T,Allocator>& x, Compare cmp )
```

- Metodi koji uređuju listu koristeći za poređenje elemenata operator `<` (prva verzija) ili dati funkcional `cmp` (druga verzija):

```
void sort()
template<class Compare>
    void sort( Compare cmp )
```

- Metod koji uklanja iz liste sve elemente koji su jednaki datom:

```
void remove( const T& x )
```

- Metod koji uklanja iz liste sve elemente koji zadovoljavaju dati predikat:

```
template<class Predicate>
    void remove_if( Predicate pred )
```

- Metod koji izvrće listu, menjajući redosled elemenata:

```
void reverse()
```

Van same klase `list` definisani su operatori poređenja `==`, `!=`, `<`, `<=`, `>`, `>=` koji porede liste leksikografski, tj. porede elemente redom i prvi put kada se dođe do razlike poredak se ustanovljava na osnovu poslednjeg poređenog elementa.

Iteratori

U klasi `list` definisani su tipovi iteratora `iterator`, `const_iterator`, `reverse_iterator` i `const_reverse_iterator`.

Povezani delovi standardne biblioteke

Najčešća alternativa listi jeste klasa `vector`, koja se od liste razlikuje po tome što omogućava efikasan neposredan pristup elementima, ali omogućava efikasno dodavanje i uklanjanje elemenata samo na kraju sekvence. Opisana je u odeljku *10.6.2 Vektor*, na strani 360.

Klasa `deque` je slična klasi `vector`, s tim što omogućava efikasno dodavanje i uklanjanje elemenata i na početku sekvence. Opisana je u odeljku *10.6.3 Dek*, na strani 363.

Klasa `list` se često upotrebljava za pravljenje apstraktnih kolekcija. Apstraktne kolekcije su opisane u odeljku *10.7 Apstraktne kolekcije*, na strani 366.

10.6.2 Vektor

Šablon klase `vector` standardne biblioteke programskog jezika C++ predstavlja sekvencijalnu kolekciju u kojoj su elementi poređani po svom rednom broju – indeksu. Vektor omogućava efikasan pristup elementima na osnovu njihovog indeksa, kao i efikasno dodavanje elemenata na kraj i njihovo uklanjanje sa kraja vektora. Dodavanje elemenata usred vektora i brisanje elemenata iz sredine vektora zahtevaju premeštanje velikog broja drugih elemenata i predstavljaju neefikasne operacije.

Zaglavlja

Šablon klase `vector` definisan je u zaglavlju `vector`.

Parametri šablona

Šablon `vector` je definisan sa dva parametra:

```
template<
    class T,
    class Allocator = allocator<T>
    >
class vector;
```

Parametar `T` određuje tip elemenata vektora. Da bi se mogla inicijalno postavljati ili dinamički povećavati veličina vektora, neophodno je da klasa `T` ima konstruktor bez

argumenata. Ukoliko ga nema, vektor se može povećavati samo dodavanjem pojedinačnih elemenata ili kopiranjem iz drugih kolekcija. Ne postoje druga ograničenja vezana za ovaj tip.

Parametar `Allocator` određuje način alociranja novih objekata klase `T`. Ukoliko se ostavi podrazumevani parametar, prostor će se alocirati na uobičajen način, primenom izraza `new`.

Vektori se najčešće upotrebljavaju samo uz navođenje tipa elemenata.

Osnovne operacije

Najvažniji metodi šablona klase `vector` su:

- Konstruktor praznog vektora:

```
vector()
```

- Konstruktor vektora date veličine. Elementi se inicijalizuju primenom konstruktora bez argumenata ili kopiranjem datog objekta `x` klase `T`:

```
vector( size_type n, const T& x = T() )
```

- Metodi koji vraćaju početne i završne iteratore. Ako je objekat konstantan i rezultat je konstantan iterator:

```
iterator begin()
iterator end()
reverse_iterator rbegin()
reverse_iterator rend()
```

- Izračunavanje veličine vektora i najveće dopuštene veličine vektora:

```
size_type size() const
size_type max_size() const
```

- Promena veličine vektora. Ako se vektor smanjuje, višak elemenata se uklanja. Ako se vektor povećava, novi elementi se prave primenom konstruktora bez argumenata ili kopiranjem datog objekta `x` klase `T`:

```
void resize( size_type n )
void resize( size_type n, const T& x )
```

- Metod za pristupanje elementu sa datim indeksom. Indeks mora biti u opsegu od `0` do `size()-1`. Postoji u obliku metoda `at` i operatora `[]`. Na raspolaganju su verzije za konstantne i nekonstantne vektore:

```
const T& at( size_type n ) const
T& at( size_type n )
const T& operator []( size_type n ) const
T& operator []( size_type n )
```

- Metodi za dodavanje elementa na kraj vektora i brisanje elementa sa kraja vektora. Menjaju veličinu vektora. Ne sme se brisati element praznog vektora:

```
void push_back( const T& x )
void pop_back()
```

- Metodi za pristupanje prvom i poslednjem elementu vektora. Postoje u verzijama za konstantne i nekonstantne vektore:

```
const T& front() const
T& front()
const T& back() const
T& back()
```

- Izračunavanje do kog broja elemenata vektor može da raste bez implicitne alokacije prostora i kopiranja sadržaja:

```
size_type capacity() const
```

- Alokacija rezervnog prostora za čuvanje elemenata, bez njihovog konstruisanja ili kopiranja. Stvarna veličina vektora ostaje neizmenjena, već se menja kapacitet alociranog prostora. Ako je nova vrednost manja od `capacity()`, neće biti promene, u suprotnom se alocira novi prostor i premeštaju se svi elementi, zbog čega svi iteratori postaju neispravni:

```
void reserve( size_type n )
```

- Provera da li je vektor prazan:

```
bool empty() const
```

- Pražnjenje vektora:

```
void clear()
```

- Metodi za umetanje elemenata u vektor. Prvi metod umeće kopiju elementa `x` pre elementa na koji ukazuje iterator `position`. Rezultat je iterator koji ukazuje na umetnut element. Drugi metod umeće `n` kopija elementa `x` pre elementa na koji ukazuje iterator `position`. Treći metod umeće ispred elementa na koji ukazuje iterator `position` sve elemente neke kolekcije (bilo kog tipa), koji pripadaju opsegu datih iteratora:

```
iterator insert( iterator position, const T& x )
void insert( iterator position, size_type n, const T& x)
template <class InputIterator>
    void insert( iterator position,
                 InputIterator first, InputIterator last )
```

- Metodi za brisanje elemenata vektora. Brišu, redom, element na koji ukazuje dati iterator ili sve elemente zahvaćene opsegom datih iteratora. Rezultat je iterator na element neposredno iza poslednjeg obrisanog, ili `end()` ako je obrisan poslednji element vektora:

```
iterator erase(iterator position)
iterator erase(iterator first, iterator last)
```

- Metod za efikasnu razmenu kompletnog sadržaja dva vektora istog tipa:

```
void swap( vector<T,Allocator>& v )
```

Van same klase `vector` definisani su operatori poređenja `==`, `!=`, `<`, `<=`, `>`, `>=` koji porede vektore leksikografski, tj. porede elemente redom i prvi put kada se dođe do razlike poredak se ustanovljava na osnovu poslednjeg poređenog elementa.

Iteratori

U klasi `vector` definisani su tipovi iteratora `iterator`, `const_iterator`, `reverse_iterator` i `const_reverse_iterator`.

Povezani delovi standardne biblioteke

Ukoliko je potrebno praviti nizove logičkih vrednosti, valja imati u vidu da kolekcije tipa `vector<bool>` imaju neke specifičnosti koje omogućavaju kako manji utrošak memorije, tako i povećanu efikasnost.

Klasa `deque` je slična klasi `vector`, ali podržava efikasno dodavanje i uklanjanje elemenata i na početku sekvence. Opisana je u odeljku *10.6.3 Dek*, na strani 363.

Alternativa vektoru je klasa `list`, koja omogućava efikasano dodavanje i uklanjanje elemenata usred sekvence, ali ne omogućava efikasan neposredan pristup elementima. Opisana je u odeljku *10.6.1 Lista*, na strani 357.

Klasa `vector` se često upotrebljava za pravljenje apstraktnih kolekcija. Štaviše, ona predstavlja podrazumevanu vrstu kolekcije za apstraktnu kolekciju `priority_queue`. Apstraktne kolekcije su opisane u odeljku *10.7 Apstraktne kolekcije*, na strani 366.

10.6.3 Dek

Šablon klase `deque` standardne biblioteke programskog jezika C++ je veoma sličan šablonu klase `vector`, s tim da je omogućeno da se kolekcija efikasno povećava i smanjuje u oba smera. Zbog toga je efikasnost pristupanja elementima delimično umanjena.

Kao i u slučaju vektora, elementi su u okviru deka poređani po svom rednom broju – indeksu. Moguće je pristupanje elementima na osnovu indeksa, efikasno dodavanje elemenata na početak i kraj i njihovo uklanjanje sa početka i kraja. Dodavanje elemenata usred kolekcije i brisanje elemenata iz sredine kolekcije zahtevaju premeštanje većeg broja drugih elemenata i predstavljaju neefikasne operacije.

Zaglavlja

Šablon klase `deque` definisan je u zaglavlju `deque`.

Parametri šablona

Šablon `deque` je definisan sa dva parametra:

```
template<
    class T,
    class Allocator = allocator<T>
    >
class deque;
```

Parametar `T` određuje tip elemenata. Da bi se mogla inicijalno postavljati ili dinamički povećavati veličina kolekcije, neophodno je da klasa `T` ima konstruktor bez argumenata.

Ukoliko ga nema, kolekcija se može povećavati samo dodavanjem pojedinačnih elemenata ili kopiranjem iz drugih kolekcija. Ne postoje druga ograničenja vezana za ovaj tip.

Parametar `Allocator` određuje način alociranja novih objekata klase `T`. Ukoliko se ostavi podrazumevani parametar, prostor će se alocirati na uobičajen način, primenom izraza `new`.

Vektori se najčešće upotrebljavaju samo uz navođenje tipa elemenata.

Osnovne operacije

Najvažniji metodi šablona klase `deque` su:

- Konstruktor prazne kolekcije `deque`:

```
deque()
```

- Konstruktor kolekcije date veličine. Elementi se inicijalizuju primenom konstruktora bez argumenata ili kopiranjem datog objekta `x` klase `T`:

```
deque( size_type n, const T& x = T() )
```

- Metodi koji vraćaju početne i završne iteratore. Ako je objekat konstantan i rezultat je konstantan iterator:

```
iterator begin()
iterator end()
reverse_iterator rbegin()
reverse_iterator rend()
```

- Izračunavanje veličine kolekcije i najveće dopuštene veličine kolekcije:

```
size_type size() const
size_type max_size() const
```

- Promena veličine kolekcije. Ako se smanjuje, višak elemenata se uklanja. Ako se povećava, novi elementi se prave primenom konstruktora bez argumenata ili kopiranjem datog objekta `x` klase `T`:

```
void resize( size_type n )
void resize( size_type n, const T& x )
```

- Metod za pristupanje elementu sa datim indeksom. Indeks mora biti u opsegu od `0` do `size()-1`. Postoji u obliku metoda `at` i operatora `[]`. Na raspolaganju su verzije za konstantne i nekonstantne kolekcije:

```
const T& at( size_type n ) const
T& at( size_type n )
const T& operator [](size_type n ) const
T& operator [](size_type n )
```

- Metodi za dodavanje elementa na početak ili kraj kolekcije i brisanje elementa sa početka ili kraja kolekcije. Svi metodi menjaju veličinu kolekcije. Metodi koji se odnose na početak kolekcije menjaju indekse svih elemenata kolekcije. Ne sme se brisati element prazne kolekcije:

```
void push_front( const T& x )
```

```
void push_back( const T& x )
void pop_front()
void pop_back()
```

- Metodi za pristupanje prvom i poslednjem elementu. Postoje u verzijama za konstantne i nekonstantne kolekcije:

```
const T& front() const
T& front()
const T& back() const
T& back()
```

- Provera da li je kolekcija prazna:

```
bool empty() const
```

- Pražnjenje kolekcije:

```
void clear()
```

- Metodi za umetanje elemenata u kolekciju. Prvi metod umeće kopiju elementa `x` pre elementa na koji ukazuje iterator `position`. Rezultat je iterator koji ukazuje na umetnut element. Drugi metod umeće `n` kopija elementa `x` pre elementa na koji ukazuje iterator `position`. Treći metod umeće ispred elementa na koji ukazuje iterator `position` sve elemente neke kolekcije (bilo kog tipa), koji pripadaju opsegu datih iteratora:

```
iterator insert( iterator position, const T& x )
void insert( iterator position, size_type n, const T& x)
template <class InputIterator>
    void insert( iterator position,
                 InputIterator first, InputIterator last )
```

- Metodi za brisanje elemenata kolekcije. Brišu, redom, element na koji ukazuje dati iterator ili sve elemente zahvaćene opsegom datih iteratora. Rezultat je iterator na element neposredno iza poslednjeg obrisanog, ili `end()` ako je obrisan poslednji element kolekcije:

```
iterator erase(iterator position)
iterator erase(iterator first, iterator last)
```

- Metod za efikasnu razmenu kompletnog sadržaja dve kolekcije `deque`, sa elementima istog tipa:

```
void swap( deque<T,Allocator>& v )
```

Van same klase `deque` definisani su operatori poređenja `==`, `!=`, `<`, `<=`, `>`, `>=` koji porede kolekcije leksikografski, tj. porede elemente redom i prvi put kada se dođe do razlike poredak se ustanovljava na osnovu poslednjeg poređenog elementa.

Iteratori

U klasi `deque` definisani su tipovi iteratora `iterator`, `const_iterator`, `reverse_iterator` i `const_reverse_iterator`.

Povezani delovi standardne biblioteke

Klasa `vector` je slična klasi `deque`, ali obezbeđuje nešto efikasniji pristup elementima, po cenu toga što ne pruža efikasno dodavanje i uklanjanje elemenata na početku niza. Opisana je u odeljku *10.6.2 Vektor*, na strani 360.

Alternativa je i klasa `list`, koja omogućava efikasano dodavanje i uklanjanje elemenata na proizvoljno mesto u okviru sekvence, ali ne omogućava efikasan neposredan pristup elementima. Opisana je u odeljku *10.6.1 Lista*, na strani 357.

Klasa `deque` se često upotrebljava za pravljenje apstraktnih kolekcija. Štaviše, ona predstavlja podrazumevanu vrstu kolekcije za apstraktne kolekcije `stack` i `queue`. Apstraktne kolekcije su opisane u odeljku *10.7 Apstraktne kolekcije*, na strani 366.

10.7 Apstraktne kolekcije

Apstraktne kolekcije se definišu kao *adapteri*, koji se primenjuju na neku sekvencijalnu kolekciju, koja će zaista sadržati elemente. Apstraktne kolekcije ne omogućavaju neposrednu primenu iteratora. Standardna biblioteka programskog jezika C++ definiše tri apstraktne kolekcije: `stack`, `queue` i `priority_queue`.

10.7.1 Stek

Šablon klase `stack` predstavlja apstraktnu kolekciju podataka, koja obezbeđuje ponašanje *gomile* (najčešće se upotrebljava termin *stek*) – elementi se dodaju, uzimaju i mogu koristiti samo na vrhu gomile.

Zaglavlja

Šablon klase `stack` definisan je u zaglavlju `stack`.

Parametri šablona

Šablon `stack` je definisan sa dva parametra:

```
template<
    class T,
    class Container = deque<T>
    >
class stack;
```

Parametar `T` određuje tip elemenata steka. Ne postoje druga ograničenja vezana za ovaj tip, osim onih koja implicira primenjena sekvencijalna kolekcija.

Parametar `Container` određuje tip sekvencijalne kolekcije koja se uotrebljava za čuvanje elemenata steka. Ukoliko se kolekcija ne navede eksplicitno, upotrebljava se dek sa podrazumevanim alokatorom.

Stek se najčešće upotrebljava samo uz navođenje tipa elemenata. Ukoliko se očekuje da se popunjenost steka često i intenzivno menja, može biti efikasnije da se upotrebljava lista umesto deka.

Osnovne operacije

Najvažniji metodi šablona klase `stack` su:

- Konstruktor praznog ili inicijalno popunjenog steka:

```
stack( const Container& = Container() )
```

- Metodi za dodavanje elementa na vrh steka i brisanje elementa sa vrha steka. Ne sme se brisati element praznog steka:

```
void push( const T& x )
void pop()
```

- Metodi za pristupanje elementu na vrhu steka:

```
const T& top() const
T& top()
```

- Provera da li je stek prazan:

```
bool empty() const
```

- Izračunavanje broja elemenata na steku:

```
size_type size() const
```

Van same klase `stack` definisani su operatori poređenja ==, !=, <, <=, >, >= koji porede stekove leksikografski, tj. porede elemente redom i prvi put kada se dođe do razlike poredak se ustanovljava na osnovu poslednjeg poređenog elementa.

Iteratori

Klasa `stack` ne podržava iteratore.

Povezani delovi standardne biblioteke

Ukoliko interfejs steka nije dovoljan, već su potrebni i iteratori ili neki drugi metodi, alternativa je da se umesto steka upotrebi neposredno neka od sekvencijalnih kolekcija.

10.7.2 Red

Šablon klase `queue` standardne biblioteke programskog jezika C++ predstavlja apstraktnu kolekciju podataka, koja obezbeđuje ponašanje *reda* – elementi se dodaju na kraj reda, uzimaju sa početka reda, a koriste bilo na početku ili na kraju reda.

Zaglavlja

Šablon klase `queue` definisan je u zaglavlju `queue`.

Parametri šablona

Šablon `queue` je definisan sa dva parametra:

```
template<
    class T,
    class Container = deque<T>
    >
class queue;
```

Parametar `T` određuje tip elemenata reda. Ne postoje druga ograničenja vezana za ovaj tip, osim onih koja implicira primenjena sekvencijalna kolekcija.

Parametar `Container` određuje tip sekvencijalne kolekcije koja se uotrebljava za čuvanje elemenata reda. Ukoliko se kolekcija ne navede eksplicitno, upotrebljava se dek sa podrazumevanim alokatorom.

Red se najčešće upotrebljava samo uz navođenje tipa elemenata. Ukoliko se očekuje da se popunjenost reda često i intenzivno menja, može biti efikasnije da se upotrebljava lista umesto deka.

Osnovne operacije

Najvažniji metodi šablona klase `queue` su:

- Konstruktor praznog ili inicijalno popunjenog reda:

```
queue( const Container& = Container() )
```

- Metod za dodavanje elementa na kraj reda:

```
void push( const T& x )
```

- Metod za brisanje elementa sa početka reda. Ne sme se brisati element praznog reda:

```
void pop()
```

- Metodi za pristupanje elementu na početku reda:

```
const T& front() const
T& front()
```

- Metodi za pristupanje elementu na kraju reda:

```
const T& back() const
T& back()
```

- Provera da li je red prazan:

```
bool empty() const
```

- Izračunavanje broja elemenata u redu:

```
size_type size() const
```

Van same klase `queue` definisani su operatori poređenja `==`, `!=`, `<`, `<=`, `>`, `>=` koji porede redove leksikografski, tj. porede elemente redom i prvi put kada se dođe do razlike poredak se ustanovljava na osnovu poslednjeg poređenog elementa.

Iteratori

Klasa `red` ne podržava iteratore.

Povezani delovi standardne biblioteke

Ukoliko interfejs reda nije dovoljan, već su potrebni i iteratori ili neki drugi metodi, alternativa je da se umesto reda neposredno upotrebi neka od sekvencijalnih kolekcija.

10.7.3 Red sa prioritetom

Šablon klase `priority_queue` standardne biblioteke programskog jezika C++ predstavlja apstraktnu kolekciju podataka, koja obezbeđuje ponašanje *reda sa prioritetom* – elementi se dodaju u red na mesto koje im odgovara na osnovu date funkcije prioriteta, a u svakom trenutku se može koristiti i brisati samo element sa najvećim prioritetom (tj. na početka reda).

Zaglavlja

Šablon klase `priority_queue` definisan je u zaglavlju `queue`.

Parametri šablona

Šablon `priority_queue` je definisan sa tri parametra:

```
template<
    class T,
    class Container = vector<T>,
    class Compare = less<Container::value_type>
    >
class priority_queue;
```

Parametar `T` određuje tip elemenata reda. Ne postoje druga ograničenja vezana za ovaj tip, osim onih koja implicira primenjena sekvencijalna kolekcija.

Parametar `Container` određuje tip sekvencijalne kolekcije koja se uotrebljava za čuvanje elemenata reda. Ukoliko se kolekcija ne navede eksplicitno, upotrebljava se vektor sa podrazumevanim alokatorom.

Parametar `Compare` predstavlja binarni funkcionalni tip koji proverava da li je prvi argument manji od drugog. Pretpostavlja se da su argumenti tipa `T`. Ako važi `Compare()(x,y)`, tada `y` ima veći prioritet nego `x`. Ukoliko se funkcional ne navede eksplicitno, upotrebljava se operator poređenja `<`, onako kako je definsan za objekte tipa `T`.

Red sa prioritetom se upotrebljava samo uz navođenje tipa elemenata, ukoliko je prioritet dobro definisan operatorom `<` za argumente tipa `T`. U suprotnom, neophodno je navesti i tip kolekcije i funkcional koji se upotrebljava za poređenje elemenata reda.

Osnovne operacije

Najvažniji metodi šablona klase `priority_queue` su:

- Konstruktor praznog ili inicijalno popunjenog reda sa prioritetom:

```
priority_queue(
    const Compare& = Compare(),
    const Container& = Container()
    )
```

- Konstruktor reda sa prioritetom koji smešta u red sve elemente obuhvaćene opsegom iteratora `first` i `last`:

```
template <class InputIterator>
priority_queue(
    InputIterator first, InputIterator last,
    const Compare& x = Compare(),
```

```
    const Container& = Container()
    )
```

- Metod za dodavanje elementa u red, na mesto koje odgovara njegovom prioritetu:

```
void push( const T& x )
```

- Metod za brisanje elementa sa najvećim prioritetom (sa početka reda). Ne sme se brisati element praznog reda:

```
void pop()
```

- Metod za čitanje elementa sa najvećim prioritetom (na početku reda):

```
const T& top() const
```

- Provera da li je red prazan:

```
bool empty() const
```

- Izračunavanje broja elemenata u redu:

```
size_type size() const
```

Iteratori

Klasa `priority_queue` ne podržava iteratore.

Povezani delovi standardne biblioteke

Ukoliko interfejs reda sa prioritetom nije dovoljan, već su potrebni i iteratori ili neki drugi metodi, alternativa je da se umesto reda neposredno upotrebi neka od sekvencijalnih kolekcija. Zbog dodavanja na mesto u redu koje zavisi od prioriteta, i lista i vektor imaju svoje kvalitete i mane. Dok lista omogućava efikasnije dodavanje i brisanje elemenata, vektor omogućava efikasno binarno pretraživanje.

10.8 Uređene kolekcije

Jedna od osobina standardne biblioteke programskog jezika C++, koja može izgledati čudno programerima koji su iskustvo sticali koristeći programski jezik C, jeste odsustvo binarnog drveta kao standardnog oblika kolekcije. Ipak, takav dizajn standardne biblioteke je više nego ispravan, što se lako uviđa već posle pisanja nekoliko programa u kojima bi se u slučaju drugačije koncipirane biblioteke verovatno upotrebljavalo binarno drvo, a ovako se koriste šabloni klasa `set` i `map`. Naime, binarno drvo se najčešće upotrebljava za implementiranje skupova ili kataloga, a klase `set` i `map` standardne biblioteke imaju upravo tu ulogu i to uz daleko jednostavniju upotrebu nego što je to slučaj sa binarnim drvetom. Klase `set` i `map` opisujemo u istom odeljku jer imaju slično ponašanje:

- predstavljaju kolekcije čiji su elementi međusobno uređeni;
- omogućavaju brz pristup elementima, odnosno brzo proveravanje da li traženi element pripada kolekciji ili ne;

- upotreba iteratora je u skladu sa uređenjem;
- imaju veoma sličan skup metoda;
- implementiraju se na sličan način;
- imaju slične zahteve u odnosu na tip elemenata kolekcije.

Pored ovih klasa, postoje i varijante koje dopuštaju ponavljanje elemenata: `multiset` i `multimap`.

10.8.1 Skup

Šablon klase `set` standardne biblioteke programskog jezika C++ predstavlja kolekciju koja se ponaša poput skupa. Iako uobičajeni pojam skupa ne podrazumeva uređenje, elementi sadržani u objektu klase `set` su uređeni radi efikasnijeg rada. Šablon klase `set` je projektovan tako da omogući što efikasnije dodavanje, uklanjanje i traženje elemenata.

Zaglavlja

Šablon klase `set` definisan je u zaglavlju `set`.

Parametri šablona

Šablon `set` je definisan sa tri parametra:

```
template<
    class Key,
    class Compare = less<Key>,
    class Allocator = allocator<Key>
    >
class set;
```

Parametar `Key` određuje tip elemenata skupa. Nema ograničenja vezanih za ovaj tip.

Parametar `Compare` određuje način poređenja objekata klase `Key`. Parametar je binarni predikat nad tipom `Key` (videti *10.3 Funkcionali*, na strani 351). Ukoliko se ostavi podrazumevani parametar `less<Key>`, poređenje objekata tipa `Key` će se izvoditi pomoću operatora `<`. Time se uvodi ograničenje:

- Mora biti definisan jedan od operatora:
  - `bool Key::operator<(const Key&) const`
  - `bool operator<(const Key&, const Key&)`

Parametar `Allocator` određuje način alociranja novih objekata klase `Key`. Ako se ostavi podrazumevani parametar, prostor će se alocirati primenom izraza `new`.

Najčešće se upotrebljava uz navođenje samo prvog parametra, uz obezbeđivanje odgovarajućeg operatora poređenja objekata tipa `Key`.

Osnovne operacije

Najvažniji metodi šablona klase `set` su:

- Konstruktor praznog skupa:

```
set()
```

- Metodi koji vraćaju početne i završne iteratore. Ako je objekat konstantan i rezultat je konstantan iterator:

```
iterator begin()
iterator end()
reverse_iterator rbegin()
reverse_iterator rend()
```

- Izračunavanje veličine skupa i maksimalne moguće veličine skupa (redom):

```
size_type size() const
size_type max_size() const
```

- Provera da li je skup prazan:

```
bool empty() const
```

- Pražnjenje skupa:

```
void clear()
```

- Provera da li dati element postoji u skupu. Rezultat je `1` ako postoji ili `0` ako ne postoji:

```
size_type count(const Key& x) const
```

- Metodi za dodavanje elemenata skupu. Prvi metod dodaje dati element. Drugi je sličan ali omogućava da se sugeriše na koje mesto bi trebalo da se izvrši dodavanje. Ako mesto ne odgovara uređenju, biće pronađena odgovarajuća pozicija. Treći metod dodaje sve elemente neke kolekcije (bilo kog tipa), koji pripadaju opsegu datih iteratora. U svakom slučaju se vodi računa da ne dođe do ponavljanja elemenata. Ako u skupu već postoji odgovarajući element ne izvodi se njegovo dodavanje. U slučaju trećeg metoda dodaju se oni objekti koji ne postoje u skupu, a ostali se ne dodaju. Rezultat prvog metoda je par koji čine iterator na dodati element (ili na pronađen, ako je već postojao) i indikator dodavanja (čija je vrednost `true` ako i samo ako je element dodat). Drugi metod vraća kao rezultat samo iterator.

```
pair<iterator, bool> insert(const Key& x)
iterator insert(iterator position, const Key& x)
template <class InputIterator>
    void insert(InputIterator first, InputIterator last)
```

- Metodi za brisanje elemenata skupa. Brišu, redom, dati element, element na koji ukazuje dati iterator ili sve elemente zahvaćene opsegom datih iteratora. Najčešće se koristi prvi, koji je ujedno i najjednostavniji. Njegov rezultat je `1` ako je element sa datim ključem postojao ili `0` ako nije.

```
size_type erase(const Key& x)
void erase(iterator position)
void erase(iterator first, iterator last)
```

- Metodi za traženje elemenata u skupu. Metod `find` traži tačno dati element. Metod `lower_bound` traži prvi element koji nije manji, a metod `upper_bound` prvi element koji jeste veći od datog objekta. Ako traženje uspe, rezultat svakog od metoda je iterator koji ukazuje na pronađeni element, a ako ne uspe, rezultat je završni iterator `end()`.

```
iterator find(const Key& x) const
iterator lower_bound(const Key& x) const
iterator upper_bound(const Key& x) const
```

Van same klase `set` definisani su operatori poređenja `==`, `!=`, `<`, `<=`, `>`, `>=` koji porede skupove leksikografski, tj. porede elemente redom i prvi put kada se dođe do razlike poredak se ustanovljava na osnovu poslednjeg poređenog elementa.

Iteratori

U klasi `set` definisani su tipovi iteratora `iterator`, `const_iterator`, `reverse_iterator` i `const_reverse_iterator`.

Povezani delovi standardne biblioteke

Klasa `multiset`, koja se razlikuje od klase `set` po tome što može sadržati više ponovljenih elemenata, opisana je u odeljku *10.8.2 Skup sa ponavljanjem*, na strani 373.

Algoritmi za izvođenje složenijih operacija na skupovima opisani su u odeljku *10.9.2 Operacije nad skupovima*, na strani 381.

10.8.2 Skup sa ponavljanjem

Šablon klase `multiset` predstavlja kolekciju koja je veoma slična klasi `set`, ali dopušta ponavljanje elemenata. Ovde će biti istaknute samo razlike.

Zaglavlja

Šablon klase `multiset` definisan je u zaglavlju `set`.

Parametri šablona

Broj parametara, njihovo značenje i odgovarajuća ograničenja su potpuno isti kao u slučaju klase `set`:

```
template<
    class Key,
    class Compare = less<Key>,
    class Allocator = allocator<Key>
    >
class multiset;
```

Osnovne operacije

Najvažniji metodi šablona klase `multiset`, koji se razlikuju, ili bar različito ponašaju, u odnosu na metode klase `set` su:

- Konstruktor praznog skupa:

```
multiset()
```

- Metod `count`, koji proverava da li dati element postoji u skupu, kao rezultat vraća broj pojavljivanja elementa u skupu, koji može biti veći od 1.
- Metodi za dodavanje elemenata skupu su slični kao u slučaju klase `set`. Razlikuje se samo navedeni metod, koji, zbog toga što dodavanje elementa uvek uspeva, vraća samo iterator na dodati element:

```
iterator insert(const Key& x)
```

- Metodi za brisanje elemenata skupa su isti. Menja se samo ponašanje navedenog metoda, koji briše sve elemente sa odgovarajućom vrednošću ključa i kao rezultat vraća broj obrisanih elemenata:

```
size_type erase(const Key& x)
```

- Metodi za traženje elemenata u skupu su isti kao u klasi `set`. Specifičnost metoda `find` je da se, u slučaju da ima više elemenata jednakih datom ključu, ne garantuje koji će se od njih pronaći. Ponašanje ostalih metoda je isto kao u klasi `set`.

Operatori poređenja su definisani na isti način kao za klasu `set`.

Iteratori

U klasi `multiset` definisani su tipovi iteratora `iterator`, `const_iterator`, `reverse_iterator` i `const_reverse_iterator`.

Povezani delovi standardne biblioteke

Klasa `set`, koja se razlikuje od klase `multiset` po tome što ne može sadržati više ponovljenih elemenata, opisana je u odeljku *10.8.1 Skup*, na strani 371.

10.8.3 Katalog

Šablon klase `map` je kolekcija sa semantikom kataloga.

Zaglavlja

Šablon klase `map` definisan je u zaglavlju `map`.

Parametri šablona

Šablon `map` je definisan sa četiri parametra:

```
template<
    class Key,
    class T,
    class Compare = less<Key>,
    class Allocator = allocator< pair<const Key, T> >
    >
class map;
```

Parametar `Key` određuje tip ključa po kome se pristupa elementima kataloga. Ne postoje ograničenja vezana za ovaj tip.

Parametar `T` određuje tip podataka koji u katalogu predstavljaju ključu odgovarajuću vrednost. U zavisnosti od načina upotrebe, što je detaljnije opisano u nastavku ovog odeljka, može da postoji jedno ograničenje za ovaj tip:

- Mora postojati javan podrazumevani konstruktor `T::T()`.

Katalog predstavlja kolekciju elemenata tipa `pair<const Key, T>`. Elementima kataloga ne može se menjati jednom određena vrednost ključa, već je dopušteno menjanje samo vrednosti tipa `T`. Tip `pair<const Key,T>` je raspoloživ i pod imenom `map::value_type` (videti *10.4.1 Uređeni par – struktura pair*, na strani 353).

Parametar `Compare` određuje način poređenja objekata klase `Key`. Parametar mora biti binarni predikat nad tipom `Key` (videti *10.3 Funkcionali*, na strani 351). Ukoliko se ostavi podrazumevani parametar `less<Key>`, poređenje objekata tipa `Key` će se izvoditi pomoću operatora <. Time se uvodi ograničenje:

- Mora biti definisan jedan od operatora:
  - `bool Key::operator<(const Key&) const`
  - `bool operator<(const Key&, const Key&)`

Parametar `Allocator` određuje način alociranja novih elemenata, tj. objekata klase `pair<const Key, T>`. Ukoliko se ostavi podrazumevani parametar, prostor će se alocirati na uobičajen način, primenom izraza `new`.

Najčešće se upotrebljava uz navođenje samo prva dva parametra, uz eventualno obezbeđivanje odgovarajućeg operatora poređenja objekata tipa `Key`. Kako je ključ najčešće celobrojnog tipa ili tipa `string`, odgovarajući operatori poređenja već postoje.

Najčešće se upotrebljava uz navođenje samo prva dva parametra, uz obezbeđivanje odgovarajućeg operatora poređenja za tip `Key`.

Osnovne operacije

Najvažniji metodi šablona klase `map` su:

- Konstruktor praznog kataloga:

```
map()
```

- Metodi koji vraćaju početne i završne iteratore. Ako je objekat konstantan i rezultat je konstantan iterator:

```
iterator begin()
iterator end()
reverse_iterator rbegin()
reverse_iterator rend()
```

- Izračunavanje veličine kataloga i maksimalne moguće veličine kataloga:

```
size_type size() const
size_type max_size() const
```

- Provera da li je katalog prazan:

```
bool empty() const
```

- Pražnjenje kataloga:

```
void clear()
```

- Provera da li u katalogu postoji element sa datim ključem. Rezultat je `1` ako postoji ili `0` ako ne postoji:

```
size_type count(const Key& x) const
```

- Metodi za dodavanje elemenata katalogu. Prvi metod dodaje dati element. Drugi je sličan ali omogućava da se sugeriše na koje mesto bi trebalo da se izvrši dodavanje. Ako mesto ne odgovara uređenju, biće pronađena odgovarajuća pozicija. Treći metod dodaje sve elemente neke kolekcije (bilo kog tipa), koji pripadaju opsegu datih iteratora. U svakom slučaju se vodi računa da ne dođe do ponavljanja elemenata sa istim ključem. Ako u katalogu već postoji element sa istim ključem ne izvodi se njegovo dodavanje. Važno je imati u vidu da se u tom slučaju ne izvodi čak ni promena vrednosti podatka odgovarajućeg elementa. U slučaju trećeg metoda dodaju se oni objekti za koje u katalogu ne postoje elementi sa istim ključem, a ostali se ne dodaju. Rezultat prvog metoda je par koji čine iterator na dodati element (ili na pronađen, ako je već postojao element sa istim ključem) i indikator dodavanja (čija je vrednost `true` ako i samo ako je element dodat). Drugi metod vraća kao rezultat samo iterator.

```
pair<iterator, bool> insert(const value_type& x)
iterator insert(iterator position, const value_type& x)
template <class InputIterator>
    void insert(InputIterator first, InputIterator last)
```

- Metodi za brisanje elemenata kataloga. Brišu, redom, element sa datim ključem, element na koji ukazuje dati iterator ili sve elemente zahvaćene opsegom datih iteratora. Najčešće se koristi prvi, koji je ujedno i najjednostavniji. Njegov rezultat je `1` ako je element sa datim ključem postojao ili `0` ako nije.

```
size_type erase(const Key& x)
void erase(iterator position)
void erase(iterator first, iterator last)
```

- Metodi za traženje elemenata u katalogu. Metod `find` traži element sa tačno datim ključem. Metod `lower_bound` traži prvi element čiji ključ nije manji, a metod `upper_bound` prvi element čiji ključ jeste veći od datog ključa. Ako traženje uspe, rezultat svakog od metoda je iterator koji ukazuje na pronađeni element, a ako ne uspe, rezultat je završni iterator `end()`. Postoje konstantne i nekonstantne verzije.

```
iterator find(const key_value& x)
iterator lower_bound(const key_type& x)
iterator upper_bound(const key_type& x)
```

- Operator za neposredno pristupanje podatku koji odgovara datom ključu:

```
T& operator[](const Key& x)
```

Ukoliko u katalogu ne postoji element sa datom vrednošću ključa, on se automatski dodaje i pridružuje mu se podrazumevana vrednost podatka. Upotreba ovog operatora se čini jednostavnom i lako razumljivom, ali neophodno je imati u vidu tri značajne posledice navedenog ponašanja:

  - u slučaju dodavanja elementa, objekat podatka tipa `T` se pravi primenom podrazumevanog konstruktora, zbog čega se operator `[]` ne može koristiti ako u klasi `T` ne postoji podrazumevani konstruktor `T::T()`;
  - `operator []` se ne može koristiti na konstantnom katalogu;
  - primena ovog operatora je uvek uspešna.

Van same klase `map` definisani su operatori poređenja `==`, `!=`, `<`, `<=`, `>`, `>=` koji porede kataloge leksikografski, tj. porede elemente redom i prvi put kada se dođe do razlike poredak se ustanovljava na osnovu poslednjeg poređenog elementa. Pri poređenju elemenata se porede najpre ključevi, pa tek ako su oni jednaki onda i vrednosti.

Iteratori

U klasi `map` definisani su tipovi iteratora `iterator`, `const_iterator`, `reverse_iterator` i `const_reverse_iterator`.

Povezani delovi standardne biblioteke

Klasa `multimap`, koja se razlikuje od klase `map` po tome što može sadržati više ponovljenih elemenata sa istim ključem, opisana je u odeljku *10.8.4 Katalog sa ponavljanjem ključeva*, na strani 377.

Struktura pair, koja predstavlja tip elemenata kataloga, opisana je u odeljku *10.4.1 Uređeni par – struktura pair*, na strani 353.

10.8.4 Katalog sa ponavljanjem ključeva

Šablon klase `multimap` standardne biblioteke programskog jezika C++ predstavlja kolekciju koja je veoma slična klasi `map`, ali dopušta ponavljanje više elemenata sa istom vrednošću ključa. Navodimo samo razlike u odnosu na klasu `map`.

Zaglavlja

Šablon klase `multimap` definisan je u zaglavlju `map`.

Parametri šablona

Broj parametara, njihovo značenje i odgovarajuća ograničenja su potpuno isti kao u slučaju klase `map`:

```
template<
    class Key,
    class T,
    class Compare = less<Key>,
```

```
        class Allocator = allocator< pair<const Key, T> >
        >
    class multimap;
```

Osnovne operacije

Najvažniji metodi šablona klase `multimap`, koji se razlikuju, ili bar različito ponašaju, u odnosu na metode klase `map` su:

- Konstruktor praznog kataloga:

  ```
  multimap()
  ```

- Metod `count`, koji proverava da li u katalogu postoji element sa datim ključem, kao rezultat vraća broj takvih elementa u katalogu, koji može biti veći od 1.
- Metodi za dodavanje elemenata katalogu su slični kao u slučaju klase `map`. Razlikuje se samo navedeni metod, koji, zbog toga što dodavanje elementa uvek uspeva, vraća samo iterator na dodati element:

  ```
  iterator insert(const value_type& x)
  ```

- Metodi za brisanje elemenata kataloga su isti. Menja se samo ponašanje navedenog metoda, koji briše sve elemente sa odgovarajućom vrednošću ključa i kao rezultat vraća broj obrisanih elemenata:

  ```
  size_type erase(const Key& x)
  ```

- Metodi za traženje elemenata u katalogu su isti kao u klasi `map`. Specifičnost metoda `find` je da se, u slučaju da ima više elemenata jednakih datom ključu, ne garantuje koji će se od njih pronaći. Ponašanje ostalih metoda je isto kao u klasi `map`.
- Značajna razlika je da ne postoji `operator[]`, jer njegova primena ne bi mogla biti jednoznačna.

Operatori poređenja su definisani na isti način kao za klasu `map`.

Iteratori

U klasi `map` definisani su tipovi iteratora `iterator`, `const_iterator`, `reverse_iterator` i `const_reverse_iterator`.

Povezani delovi standardne biblioteke

Klasa `map`, koja se razlikuje od klase `multimap` po tome što ne može sadržati više ponovljenih elemenata sa istim ključem, opisana je u odeljku *10.8.3 Katalog*, na strani 374.

Klasa `multiset`, koja se razlikuje od klase `set` po tome što može sadržati više ponovljenih elemenata, opisana je u odeljku *10.8.2 Skup sa ponavljanjem*, na strani 373.

Struktura pair, koja predstavlja tip elemenata kataloga, opisana je u odeljku *10.4.1 Uređeni par – struktura pair*, na strani 353.

10.9 Algoritmi

Standardna biblioteka programskog jezika C++ obuhvata biblioteku *algoritama*. Biblioteku algoritama čine šabloni funkcija koji imlementiraju veći broj algoritama za pretraživanje, uređivanje, inicijalizovanje i transformisanje podataka i druge operacije. Biblioteka algoritama počiva na dva mehanizma za obezbeđivanje apstraktnosti: apstrahovanje tipova podataka se postiže primenom šablona, a apstrahovanje tipova kolekcija se postiže primenom iteratora. Jedna ista implementacija algoritma se tako može primenjivati za sve tipove podataka i kolekcija. Štaviše, iteratori omogućavaju da se jedan isti algoritam upotrebljava za obavljanje više poslova (npr. traženje prvog i narednog pojavljivanja nekog elementa). I pored tako visokog nivoa apstrakcije, biblioteka algoritama pruža visok nivo performansi.

Umesto opsežnog predstavljanja, biblioteku algoritama ćemo ilustrovati predstavljanjem nekoliko algoritama za traženje i rad sa skupovima.

10.9.1 Traženje

Svi algoritmi za traženje pretražuju sekvencu određenu datim iteratorima `first` i `last`. Ukoliko pronađu postoji traženi element, rezultat je iterator koji se odnosi na njega. U suprotnom, rezultat je jednak datom završnom iteratoru `last`. Može se pretraživati bilo koja vrsta kolekcije za koju su definisani iteratori. Mogu se pretraživati i obični nizovi elemenata, u kom slučaju se kao iteratori upotrebljavaju pokazivači.

Zaglavlja

Sve navedene operacije definisane su u zaglavlju `algorithm`.

Operacije

Algoritam `find` traži prvo pojavljivanje elementa koji je jednak datoj vrednosti `value`:

```
template< class InputIterator, class T >
InputIterator find ( InputIterator first, InputIterator last,
                     const T &value )
```

Algoritam `search` traži prvo pojavljivanje podsekvence određene iteratorima `sfirst` i `slast` unutar sekvence određene iteratorima `first` i `last`. Rezultat je iterator koji se odnosi na prvi element pronađene podsekvence, ako je pronađena, ili `last`, ako nije pronađena. U prvom slučaju se elementi porede primenom operatora =, a u drugom se smatra da su jednaki ako zadovoljavaju dati binarni predikat `pred`:

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator1 search(
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 sfirst, ForwardIterator2 slast )

template< class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate >
ForwardIterator1 search(
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 sfirst, ForwardIterator2 slast,
    BinaryPredicate pred )
```

Algoritam `find_end` traži poslednje pojavljivanje podsekvence određene iteratorima `sfirst` i `slast` unutar sekvence određene iteratorima `first` i `last`. Rezultat je iterator koji se odnosi na prvi element pronađene podsekvence, ako je pronađena, ili `last`, ako nije pronađena. U prvom slučaju se elementi porede primenom operatora `=`, a u drugom se smatra da su jednaki ako zadovoljavaju dati binarni predikat `pred`:

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator1 find_end(
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 sfirst, ForwardIterator2 slast )

template< class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate >
ForwardIterator1 find_end(
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 sfirst, ForwardIterator2 slast,
    BinaryPredicate pred )
```

Algoritam `find_first_of` traži unutar sekvence određene iteratorima `first` i `last` prvo pojavljivanje elementa koji postoji i u okviru sekvence određene iteratorima `sfirst` i `slast`. U prvom slučaju se elementi porede primenom operatora `=`, a u drugom se smatra da su jednaki ako zadovoljavaju dati binarni predikat `pred`:

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator1 find_first_of(
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 sfirst, ForwardIterator2 slast )

template< class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate >
ForwardIterator1 find_first_of(
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 sfirst, ForwardIterator2 slast,
    BinaryPredicate pred )
```

Algoritam `binary_search` proverava da li unutar sekvence određene iteratorima `first` i `last` postoji element koji ima vrednost `value`. Primenjuje se binarno pretraživanje. U prvom slučaju se pretpostavlja da je sekvenca uređena u rastućem poretku primenom uobičajenih operatora poređenja. U drugom slučaju se smatra da je uređena primenom datog upoređivača `comp`:

```
template< class ForwardIterator, class Type >
bool binary_search( ForwardIterator first, ForwardIterator last,
                    const Type &value )

template< class ForwardIterator, class Type >
bool binary_search( ForwardIterator first, ForwardIterator last,
                     const Type &value, Compare comp )
```

Primer upotrebe

Naredni segment koda traži i ispisuje na standardnom izlazu sve sufikse niske `s` koji počinju slovom `a`:

```
string s = "Da li ova niska ima neko slovo 'a'?";
```

```
string::iterator f = find( s.begin(), s.end(), 'a' );
while( f != s.end() ){
    cout << f << endl;
    f = find( ++f, s.end(), 'a' );
    }
```

Povezani delovi standardne biblioteke

Pored navedenih algoritama, za operacije traženja su namenjeni i algoritmi `adjacent_find`, `search_n`, `min_element`, `max_element`, `lower_bound`, `upper_bound`, `equal_range`, `nth_element` i `mismatch`. Algoritmi `count` i `count_if` broje koliko ima elemenata u datoj sekvenci koji su jednaki datoj vrednosti ili zaovoljavaju dati unarni predikat.

Iteratori su opisani u odeljku *10.2 Iteratori*, na strani 348.

10.9.2 Operacije nad skupovima

Složenije operacije nad skupovima su implementirane u vidu algoritama, tj. šablona funkcija, kako bi se mogle primenjivati na ekvivalentan način za različite kolekcije podataka. Jedino ograničenje vezano za njihovu primenu je očekivanje da su kolekcije uređene. Isti elementi se mogu više puta ponavljati. Ako se radi o klasama `set`, `multiset`, `map` ili `multimap`, njihovi elementi su već uređeni, dok je u slučaju drugih kolekcija (npr. `vector` i `list`) neophodno urediti kolekcije pre primene ovih operacija.

Zaglavlja

Sve navedene operacije definisane su u zaglavlju `algorithm`.

Operacije

Definisane su sledeće operacije:

- Operacije unije, preseka i razlike skupova imaju isti tip i po dva oblika:

```
template <
        class InputIterator1, class InputIterator2,
        class OutputIterator
        >
OutputIterator ...operacija...(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result
        );

template <
        class InputIterator1, class InputIterator2,
        class OutputIterator,
        class Compare
        >
OutputIterator ...operacija...(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result,
        Compare comp
```

```
        );
```

Rezultat se korišćenjem izlaznog iteratora upisuje u odgovarajuću kolekciju, pri čemu se to čini u redosledu uređivanja, da bi se dobio uređen rezultat čak i ako se upisuje u kolekciju koja ne održava sama svoje uređenje. Parametar `Compare comp` je funkcional kojim se izvodi poređenje elemenata i predstavlja opcioni argument – ako se ne navede, za poređenje elemenata se koristi operator `<`. Unija skupova `set_union` upisuje u rezultat redom sve elemente sekvenci. Presek skupova `set_intersection` upisuje u rezultat redom sve elemente sekvenci koji se pojavljuju u obe kolekcije. Razlika skupova `set_difference` upisuje u rezultat one elemente prve kolekcije koji ne postoje u drugoj. Simetrična razlika skupova `set_symmetric_difference` upisuje u rezultat one elemente koji postoje samo u jednoj od kolekcija.

- Funkcija `includes` proverava da li su svi elementi druge kolekcije sadržani u prvoj. I ovde postoje dva oblika, sa i bez funkcionala za poređenje:

```
template <class InputIterator1, class InputIterator2>
bool includes(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2
        );

template <
        class InputIterator1, class InputIterator2,
        class Compare
        >
bool includes(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        Compare comp
        );
```

Primer upotrebe

Navešćemo samo primer izračunavanja unije. Slično se upotrebljavaju i ostale operacije.

```
set<int> a,b,c;
...
// c = unija skupova a i b
c.clear();
set_union(
    a.begin(), a.end(),
    b.begin(), b.end(),
    inserter( c, c.begin() )
    );
```

Povezani delovi standardne biblioteke

Za implementaciju skupova uobičajeno je da se upotrebljavaju klase `set` (*10.8.1 Skup*, na strani 371) i `multiset` (*10.8.2 Skup sa ponavljanjem*, na strani 373).

Iteratori su opisani u odeljku *10.2 Iteratori*, na strani 348.

10.10 Biblioteka tokova

Definicija programskog jezika C++ ne obuhvata mehanizme za ostvarivanje ulaznih i izlaznih operacija. Umesto toga, značajan deo standardne biblioteke programskog jezika C++ je posvećen radu sa ulazom i izlazom. Taj segment biblioteke predstavlja *Biblioteka ulazno-izlaznih tokova*. U literaturi na engleskom jeziku najčešće se pominje kao biblioteka *iostream*.

U osnovi biblioteke ulaznih i izlaznih tokova je *tok*. Tok predstavlja apstraciju datoteka, ulaznih i izlaznih uređaja i komunikacionih linija. Tok modelira protok bajtova od izvora prema korisniku. Izlazne operacije predstavljaju *uticanje* ili *prosleđivanje* podataka u tok. Ulazne operacije predstavljaju *izdvajanje* podataka iz toka. Na niskom nivou tokovi ostvaruju protok bajtova, ali na višem nivou se prepoznaju koncepti prosleđivanja i izdvajanja proizvoljno složenih objekata u i iz toka.

10.10.1 Koncepti

Biblioteka je izgrađena kao hijerarhija klasa koje predstavljaju različite funkcionalne celine. U velikoj meri počiva na primeni šablona klasa. Kao parametar šablona se pojavljuje tip koji predstavlja osnovnu jedinicu prenosa podataka. U zavisnosti ot okruženja, u upotrebi su dve osnovne grupe instanci šablona klasa: pri radu sa 8-bitnim kodnim stranama za osnovnu jedinicu prenosa podataka se uzima tip `char`, a pri radu sa 16-bitnim kodnim stranama se uzima tip `wchar_t`. U opisima koji slede najčešće ćemo pretpostavljati da je jedinica komunikacije jedan bajt.

Biblioteka tokova možda izgleda previše sofisticirano i složeno da bi bila efikasna, ali ona je daleko efikasnija od odgovarajuće standardne ulazno/izlazne biblioteke programskog jezika C. Jedan od glavnih uzroka veće efikasnosti može se pronaći u činjenici da se već u toku prevođenja prepoznaje tip objekata koje je potrebno proslediti (ili čiji je sadržaj potrebno izdvojiti), dok funkcije standardne biblioteke programskog jezika C tek u vreme izvršavanja ustanovljavaju tipove vrednosti koje se ispisuju ili čitaju.

Pri navođenju formalnih deklaracija metoda klasa tokova, na više mesta se upotrebljavaju pomoćni tipovi podataka. Tipovi `streamsize`, `iostate` i `fmtflags` se obično definišu kao `int` ili `unsigned`, dok se tip `streamoff` obično definiše kao `int`. Tip `streampos` se implementira pomoću šablona `fpos<>` i ima automatsku konverziju u `long`.

Jezgro hijerarhije klasa tokova

U osnovi hijerarhije klasa tokova je klasa `ios_base`. Ona definiše one aspekte tokova koji ne zavise od tipa osnovne jedinice podataka. Iz nje se izvodi šablon `basic_ios<>` koji definiše aspekte koji zavise od tipa jedinice podatka. Obe klase se bave nižim nivoima prenosa podataka i predstavljaju osnovu za definisanje interfejsa višeg nivoa. Slika 13 predstavlja dijagram klasa koje čine jezgro hijerarhije tokova.

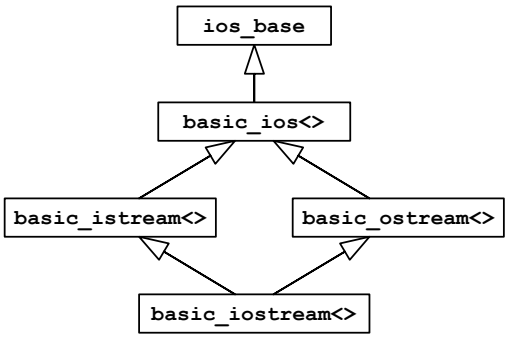
Za korisnika biblioteke najznačaniji su šabloni klase `basic_istream` i `basic_ostream` i interfejsi koje one definišu. Klase `istream`, `ostream` i `iostream` se definišu kao:

```
typedef basic_istream<char> istream;
```

```
typedef basic_ostream<char> ostream;
typedef basic_iostream<char> iostream;
```

Odgovarajuće klase za 16-bitnu komunikaciju se definišu kao:

```
typedef basic_istream<wchar_t> wistream;
typedef basic_ostream<wchar_t> wostream;
typedef basic_iostream<wchar_t> wiostream;
```



Slika 13: Osnova hijerarhije tokova

Klasa `istream` definiše interfejs za izdvajanje podataka iz tokova. Ona predstavlja osnovu za sve ulazne tokove. Klasa `ostream` definiše interfejs za prosleđivanje podataka u tokove. Ona predstavlja osnovu za sve izlazne tokove. Kao naslednica obe prethodne klase, klasa `iostream` obuhvata oba interfejsa i predstavlja osnovu za sve tokove koji se koriste i za čitanje i za pisanje.

Pored ovih klasa, jezgru hijerarhije pripadaju i neke pomoćne klase, kojima se nećemo baviti.

Konzolni tokovi

Biblioteka sadrži nekoliko predefinisanih konzolnih tokova. Oni se upotrebljavaju za ulazno izlazne operacije u konzolnom prozoru:

- `cin` je objekat klase `istream`, koji je vezan za standardni ulaz;
- `cout` je objekat klase `ostream`, vezan za standardni izlaz;
- `cerr` je objekat klase `ostream`, koji predstavlja izlaz za greške i radi bez baferisanja;
- `clog` je objekat klase `ostream`, koji je vezan za standardni kanal za vođenje dnevnika (engl. *log*).

Postoje i odgovarajući 16-bitni objekti. Oni na početku imena imaju još slovo '`w`'.

Svi navedeni objekti deklarisani su u zaglavlju `iostream`.

10.10.2 Izlazni tokovi

Svi izlazni tokovi se izvode, neposredno ili posredno, iz šablona klase `basic_ostream`. Interfejs za pisanje u tok se sastoji iz dve celine: metoda za formatirano pisanje i metoda za neformatirano pisanje.

Formatirano pisanje

Pri formatiranom pisanju se objekat, koji se zapisuje, najpre *prevodi* u niz karaktera, koji se zatim prosleđuje u tok. Formatirano pisanje se odvija primenom operatora prosleđivanja: `<<`. Opšti oblik ovog operatora je:

```
ostream& operator<<( ostream& ostr, const T& x )
```

Sve implementacije ovog operatora moraju zadovoljiti dva osnovna principa:

- objekat koji se zapisuje ne sme se menjati pri zapisivanju;
- operator mora vratiti tok u koji su podaci upisani.

Smisao prvog principa je u poštovanju koncepta da bi jedan metod trebalo da radi tačno jedan posao. U ovom slučaju to je zapisivanje objekta, a ne i njegovo menjanje.

Smisao drugog principa je u omogućavanju serijske primene operatora prosleđivanja u obliku:

```
cout << a << b << c;
```

Operator je levo asocijativan, pa je rezultat prethodne naredbe identičan rezultatu sledećeg niza naredbi:

```
cout << a;
cout << b;
cout << c;
```

Biblioteka tokova obuhvata implementacije ovog operatora za proste tipove podataka. Karakteri, niske, nizovi karaktera i brojevi se ispisuju na očekivan način. Svi pokazivački tipovi za koje nije definisan operator prosleđivanja automatski se konvertuju u tip `void*` i ispisuju se kao brojevi u heksadekadnom zapisu. Ako je potrebno ispisati ih u dekadnom zapisu, neophodno je konvertovati ih u cele brojeve.

Da bi radilo efikasnije, formatirano pisanje se obično izvodi pomoću bafera. Eksplicitno prosleđivanje sadržaja bafera u tok i pražnjenje bafera može se izvršiti primenom metoda `flush` šablona klase `basic_ostream`:

```
ostream& flush()
```

Autori klasa su dužni da obezbede odgovarajuću implementaciju operatora prosleđivanja ako žele da se objekti njihove klase mogu prosleđivati u tokove.

Neformatirano pisanje

Pri neformatiranom pisanju se pretpostavlja da će se u tok proslediti niz bajtova koji predstavlja binarnu memorijsku reprezentaciju objekta. Neformatirano pisanje se odvija primenom metoda `put` i `write` šablona klase `basic_ostream`:

- Metod `put` ispisuje tačno jednan karakter u binarnom režimu:

```
ostream& put( char )
```

- Metod `write` ispisuje u binarnom režimu niz od `n` karaktera, počev od karaktera na koji pokazuje pokazivač `data`:

```
ostream& write( const char* data, int n )
```

Oba navedena metoda predstavljaju operacije binarnog izlaza i prosleđuju sve vrednosti od 0 do 255, uključujući i sve specijalne karaktere i terminalnu nulu. S obzirom na to da na operator prosleđivanja mogu imati uticaja neka podešavanja, a da na metode `put` i `write` ona ne utiču, može se desiti da naredne dve naredbe nemaju isti rezultat:

```
cout << 'a';
cout.put('a');
```

Pri neformatiranom pisanju može biti potrebno pisati „preko reda". To se ostvaruje primenom narednih metoda šablona klase `basic_ostream`:

- Pomeranje pisanja na apsolutnu poziciju u toku:

```
ostream& seekp( streampos pos )
```

- Pomeranje pisanja na relativnu poziciju u toku. Argument `offset` predstavlja relativnu poziciju u toku, u odnosu na referentnu tačku. Argument `dir` određuje referentnu tačku i može biti `ios::beg` (početak toka), `ios::end` (kraj toka) ili `ios::cur` (tekuća pozicija u toku):

```
ostream& seekp( streamoff offset, seek_dir dir )
```

- Čitanje trenutne apsolutne pozicije pisanja:

```
streampos tellp()
```

10.10.3 Ulazni tokovi

Svi ulazni tokovi se izvode, neposredno ili posredno, iz šablona klase `basic_istream`. Interfejs za čitanje iz toka se sastoji iz dve celine: metoda za formatirano čitanje i metoda za neformatirano čitanje.

Formatirano čitanje

Pri formatiranom čitanju se iz toka čita niz karaktera i deo po deo *prevodi* u odgovarajuće tipove podataka, da bi se tako dobijeni podaci upisali u objekat koji se čita. Formatirano čitanje se odvija primenom operatora izdvajanja: >>. Opšti oblik ovog operatora je:

```
istream& operator>>( istream& istr, T& x )
```

Sve implementacije ovog operatora moraju zadovoljiti osnovni princip:

- Operator preskače prazan prostor na početku čitanja.
- Operator mora vratiti tok iz koga su podaci izdvojeni.

Prvi princip je neophodan da na pročitane vrednosti ne bi uticalo formatiranje ulaznih podataka. Ukoliko se implementacija operatora izdvajanja za neki složeni tip podataka zasniva na upotrebi operatora izdvajanja za jednostavnije tipove podataka, potrebno je pretpostaviti da će svaki od upotrebljenih operatora preskakati prazan prostor na početku čitanja. Zbog toga, najčešće, pri čitanju složenih podataka nije neophodno eksplicitno preskakati početni prazan prostor, jer će to učiniti prvi upotrebljen operator izdvajanja za neki jednostavniji tip podataka.

Drugi princip omogućava serijsku primenu operatora izdvajanja u obliku:

```
cin >> a >> b >> c;
```

Operator je levo asocijativan, pa je rezultat prethodne naredbe identičan rezultatu sledećeg niza naredbi:

```
cin >> a;
cin >> b;
cin >> c;
```

Biblioteka tokova obuhvata implementacije ovog operatora za proste tipove podataka. Karakteri, niske, nizovi karaktera i brojevi se čitaju na očekivan način. Prepoznaju se zapisi ovih podataka koji odgovaraju rezultatu primene odgovarajućih operatora prosleđivanja. Nizovi znakova se čitaju najviše do prvog praznog prostora.

Autori klasa su dužni da obezbede odgovarajuću implementaciju operatora izdvajanja ako žele da se objekti njihove klase mogu izdvajati iz tokova.

Neformatirano čitanje

Pri neformatiranom čitanju se pretpostavlja da će se iz toka čitati niz bajtova koji predstavlja binarnu memorijsku reprezentaciju objekta. Neformatirano čitanje se odvija primenom sledećih metoda šablona klase `basic_istream`:

- Čitanje tačno jednog karaktera u binarnom režimu. U prvom obliku može da vrati i vrednost koja je van opsega tipa `char`, na primer `EOF`:

```
int get()
istream& get( char& c )
```

- Čitanje *reda* karaktera. Čita se najviše `n-1` znakova i upisuju se u niz karaktera na koji pokazuje `s`. Ukoliko se pre pročitanih `n-1` znakova naiđe na karakter `delim`, čitanje se prekida i taj karakter se ne čita. U verziji bez argumenta `delim`, pretpostavlja se da se čita do kraja reda (karakter '`\n`'). Na kraju se uvek dopisuje terminalni karakter ('`\0`').

```
istream& get( char* s, streamsize n )
istream& get( char* s, streamsize n, char delim )
```

- Čitanje *reda* karaktera. Čita se najviše `n-1` znakova i upisuju se u niz karaktera na koji pokazuje `s`. Ukoliko se pre pročitanih `n-1` znakova naiđe na karakter `delim`, čitanje se prekida, a taj karakter se izdvaja iz toka, ali ne upisuje u niz pročitanih karaktera. U verziji bez argumenta `delim`, pretpostavlja se da se čita do kraja reda (karakter '`\n`'). Na kraju se uvek dopisuje terminalni karakter ('`\0`').

```
istream& getline( char* s, streamsize n )
istream& getline( char* s, streamsize n, char delim )
```

- Metod `read` čita u binarnom režimu niz od `n` karaktera i upisuje ih u prostor na koji pokazuje pokazivač `data`:

```
istream& read( const char* data, int n )
```

- Metod `readsome` čita u binarnom režimu niz od najviše `n` karaktera i upisuje ih u prostor na koji pokazuje pokazivač `data`. Rezultat je broj pročitanih karaktera:

```
streamsize readsome( const char* data, int n )
```

- Broj karaktera koji su pročitani prethodno primenjenim metodom za neformatirano čitanje može se ustanoviti primenom metoda:

```
streamsize gcount() const
```

- Određen broj karaktera iz ulaznog toka se može namerno preskočiti (tj. izdvojiti i odbaciti) primenom metoda `ignore`. Metod preskače najviše `n` karaktera, ili dok ne naiđe na karakter `delim`. Ako se naiđe na `delim`, on se ne izdvaja iz toka:

```
istream& ignore( streamsize n = 1,
                 int delim = traits::eof() )
```

- Često je potrebno prilagođavati način čitanja podataka konkretnim sadržajima koji se nalaze u toku. Metod `peek` čita naredni karakter u binarnom režimu, ali ga ne izdvaja iz toka. Metod `unget` vraća na početak toka poslednji karakter koji je pročitan u binarnom režimu. Metod `putback` stavlja na početak toka dati karakter. Svi metodi operišu na nivou bafera koji odgovara toku, ne menjajući stvarni izvor podataka:

```
int peek()
istream& unget()
istream& putback( char c )
```

Pri neformatiranom čitanju sve operacije se odvijaju u binarnom režimu, pa se iz toka mogu izdvojiti i kontrolni simboli, uključujući karaktere praznog prostora i terminalni karakter. S obzirom na to da na operator formatiranog izdvajanja mogu imati uticaja neka prethodna podešavanja ili način definisanja toka (npr. da li je datoteka otvorena u tekstualnom ili u binarnom režimu), a da na metode neformatiranog čitanja ta podešavanja ne utiču, naredne dve naredbe često nemaju isti rezultat:

```
char c;
cin >> c;
cin.get(c);
```

Razlika je posebno značajna kada su u pitanju kontrolni simboli i karakteri koji predstavljaju prazan prostor.

Pri neformatiranom čitanju može biti potrebno čitati „preko reda“. To se ostvaruje primenom narednih metoda šablona klase `basic_istream`:

- Pomeranje pozicije narednog čitanja na datu apsolutnu poziciju u toku:

```
istream& seekg( streampos pos )
```

- Pomeranje pozicije narednog čitanja na datu relativnu poziciju u toku. Argument `offset` predstavlja relativnu poziciju u toku, u odnosu na referentnu tačku. Argument `dir` određuje referentnu tačku i može biti `ios::beg` (početak toka), `ios::end` (kraj toka) ili `ios::cur` (tekuća pozicija u toku):

```
istream& seekg( streamoff offset, seek_dir dir )
```

- Čitanje trenutne apsolutne pozicije čitanja:

```
streampos tellg()
```

10.10.4 Stanje toka

Svaki tok ima svoje stanje. Stanje toka opisuje da li se tok može upotrebljavati ili je potrebno preduzeti određene aktivnosti kako bi se stanje toka izmenilo. Stanje toka opisuje se celim brojem. Naredne konstante, definisane u klasi `ios_base`, predstavljaju osnovna stanja. Stanje toka može biti opisano i kao kombinacija više osnovnih stanja:

- `goodbit` – vrednost `0x00` – Tok je u ispravnom stanju i spreman je za upotrebu;
- `eofbit` – vrednost `0x01` – Ulazne operacije su dosegle kraj toka;
- `failbit` – vrednost `0x02` – Nije uspelo izvođenje operacije čitanja ili pisanja;
- `badbit` – vrednost `0x04` – Došlo je fatalne greške, dalja upotreba nije moguća.

Svaki metod klasa tokova (osim konstantnih metoda) može promeniti stanje toka. Očitavanjem stanja toka nakon primene metoda može se ustanoviti da li je metod uspešno uradio svoj posao ili je došlo do problema. Stanju toka pristupaju naredni metodi:

- Čitanje tekućeg stanja toka:

```
iostate rdstate() const
```

- Proveravanje da li stanje toka obuhvata odgovarajuće osnovno stanje:

```
bool good() const
bool eof() const
bool fail() const
bool bad() const
```

- Konverzija toka u tip `void*`. Rezultat je 0 ako je tok u stanju `failbit`, a 1 inače:

```
operator void*() const
```

- Konverzija toka u tip `bool`. Rezultat je `true` ako je tok u stanju `failbit`, a `false` inače:

```
bool operator!() const
```

- Postavljanje novog stanja toka. Ako se ne navede argument, postavlja se ispravno stanje:

```
void clear( iostate state = goodbit )
```

- Dodavanje novog stanja toka. Prethodno detektovana stanja se dopunjavaju navedenim stanjima:

```
void setstate( iostate state )
```

U praktičnom radu se za prepoznavanje stanja toka najčešće upotrebljavaju konverzije u logičku vrednost. Kako se konverzija podatka tipa `void*` u tip `bool` izvodi automatski, provera uspešnosti prethodne operacije čitanja ili pisanja se može sasvim jednostavno ostvariti tumačenjem objekta toka kao logičke vrednosti. Tako naredni izrazi imaju istu logičku vrednost:

```
!(tok.fail())
!(!tok)
tok
```

Proveravanje da li se došlo do kraja toka se ne izvodi unapred, već se nakon ustanovljavanju da prethodna operacija čitanja nije uspela može proveriti da li je razlog za to upravo dostizanje kraja toka.

10.10.5 Formatiranje

Pri formatiranom ulazu i izlazu moguće je prilagođavati potrebama način zapisivanja prostih tipova podataka, za koje su operatori prosleđivanja i izdvajanja implementirani u okviru standardne biblioteke. Formatiranje je moguće preduzimati na dva osnovna načina: eksplicitnim postavljanjem zastavica koje označavaju način formatiranja toka i primenom *manipulatora*.

Eksplicitno određivanje formata

Za ekplicitno rukovanje zastavicama formatiranja služe sledeći metodi:

- Dodavanje zastavica formatiranja. Metodi vraćaju ukupno stanje zastavica. Ukoliko se navede argument `mask`, pre dodavanja formatiranja se poništavaju sve zastavice u grupi koju data maska označava:

```
fmtflags setf( fmtflags f )
fmtflags setf( fmtflags f, fmtflags mask )
```

- Čitanje i postavljanje zastavica formatiranja:

```
fmtflags flags( fmtflags f )
fmtflags flags()
```

- Čitanje prethodne vrednosti i određivanje novog broja cifara iza decimalne tačke. Promena se odnosi samo na prvu sledeću operaciju pisanja:

```
streamsize precision() const
streamsize precision( streamsize n )
```

- Čitanje prethodne vrednosti i određivanje nove širine. Širina određuje broj znakova koji će biti ispisan ili maksimalan broj znakova koji će se pročitati. Promena se odnosi samo na narednu operaciju čitanja ili pisanja:

```
streamsize width() const
streamsize width( streamsize n )
```

Osnovne zastavice formatiranja i odgovarajuće maske (za metod `setf`) su definisane u klasi `ios_base` i predstavljene u tablici 3.

Zastavica	Maska	Značenje
`dec`	`basefield`	Koristi se dekadni sistem.
`hex`	`basefield`	Koristi se heksadekadni sistem.
`oct`	`basefield`	Koristi se oktalni sistem.
`left`	`adjustfield`	Izlaz se poravnava ulevo.
`right`	`adjustfield`	Izlaz se poravnava udesno.
`internal`	`adjustfield`	Umeće se prazan prostor iza znaka broja ili oznake brojčanog sistema. Inače kao right.
`fixed`	`floatfield`	Razlomljeni brojevi se ispisuju u fiksnom decimalnom zapisu.
`scientific`	`floatfield`	Razlomljeni brojevi se ispisuju u eksponencijalnom zapisu.
`boolapha`		Ako je postavljena, logičke vrednosti se ispisuju tekstualno, inače se ispisuju kao brojevi 0 i 1.
`uppercase`		Sva slova se ispisuju kao velika.
`showbase`		Pre broja se ispisuje prefiks koji označava osnovu sistema.
`showpoint`		Uvek se ispisuje decimalna tačka.
`showpos`		Ispred pozitivnih brojeva se ispisuje znak.
`skipws`		Preskače se prazan prostor pri čitanju.
`unitbuf`		Bafer se prosleđuje u tok posle svake izlazne operacije.

Tablica 3: Zastavice formatiranja

Određivanje formata pomoću manipulatora

Manipulatori su posebni objekti čije prosleđivanje u tok ili izdvajanje iz toka ne predstavlja pisanje ili čitanje, već menjanje stanja toka na određeni način. U tablici 4 su predstavljeni osnovni manipulatori koje definiše standardna biblioteka. U koloni *Tip* navedeno je da li se koriste sa ulaznim ili izlaznim tokovima – odsustvo komentara znači da se manipulator može upotrebljavati u oba slučaja.

Manipulator	Tip	Funkcija
`flush`	izlazni	Prosleđuje sadržaj bafera u tok.
`endl`	izlazni	Prosleđuje znak za novi red i sadržaj bafera u tok.
`ends`	izlazni	Prosleđuje u tok terminalni karakter '\0'.
`ws`	ulazni	Preskače sav prazan prostor.

`boolalpha`
`noboolalpha`
`uppercase`
`nouppercase`
`showbase`
`noshowbase`
`showpoint`
`noshowpoint`
`showpos`
`noshowpos`
`unitbuf`
	izlazni	Uključuju i isključuju odgovarajuće zastavice za formatiranje.

`dec`
`oct`
`hex`
		Koristi se odgovarajući brojčani sistem.

`setbase(n)`		Uključuje upotrebu datog brojčanog sistema.
`fixed` `scientific`	izlazni	Razlomljeni brojevi se ispisuju u fiksnom decimalnom ili u eksponencijalnom zapisu.

`left`
`right`
`internal`
	izlazni	Izlaz se poravnava ulevo / udesno / udesno sa umetanjem iza znaka broja.

`setiosflags(f)` `resetiosflags(f)`		Eksplicitno uključivanje i isključivanje zastavica za formatiranje.
`setprecision(n)`		Određuje broj cifara iza decimalne tačke.
`setw(n)`		Određuje širinu za narednu operaciju čitanja ili pisanja.
`setfill(c)`		Određuje karakter kojim će se popunjavati prostor do predviđene širine.

Tablica 4: Manipulatori za formatiranje tokova

10.10.6 Datotečni tokovi

Podrška za rad sa datotekama je u standardnoj biblioteci ostvarena kroz *datotečne tokove*. Datotečni tokovi su objekti klasa `ofstream` (izlazni datotečni dok), `ifstream` (ulazni datotečni dok) i `fstream` (ulazno-izlazni datotečni dok). Sve što je do sada izloženo u vezi sa tokovima, važi i za datotečne tokove.

Specifičnosti datotečnih tokova se odnose, pre svega, na potrebu da se objekti tokova povežu sa datotekama čiji će sadržaj kroz njih *proticati*. Svaki datotečni tok se pre operacija čitanja ili pisanja mora *otvoriti*. Nakon upotrebe tok se mora zatvoriti. Zbog toga datotečni tokovi raspolažu neophodnim specifičnim metodima:

- Otvaranje datotečnog toka. Prvi argument predstavlja naziv datoteke koja odgovara toku. Drugi argument opisuje način otvaranja toka. Veći broj implementacija podržava i opcioni treći argument, koji određuje način zaključavanja datoteke:

```
void open( const char*, ios_base::openmode )
```

- Proveravanje da li je tok otvoren:

```
bool is_open() const
```

- Zatvaranje toka. Nakon zatvaranja tok se ne može koristiti za čitanje i pisanje sve dok se ponovo ne otvori:

```
void close()
```

Način otvaranja toka određuje način njegove upotrebe. Ulazni tokovi se mogu otvarati samo za čitanje, a izlazni samo za pisanje, dok se ulazno-izlazni tokovi mogu otvarati i za čitanje i za pisanje. Dopušteni načini otvaranja datoteke su definisani u klasi `ios_base` i navedeni su u tablici 5. Navedeni načini otvaranja se mogu kombinovati binarnom disjunkcijom.

Svaki od datotečnih tokova se može otvoriti i bez eksplicitnog navođenja načina otvaranja. Podrazumevani način otvaranja tokova je predstavljen u tablici 6.

Način	Smer	Značenje
`in`	čitanje	Tok se otvara za čitanje.
`out`	pisanje	Tok se otvara za pisanje.
`ate`	pisanje	Odmah nakon otvaranja se i čitanje i pisanje pozicioniraju na sam kraj toka.
`app`	pisanje	Pre pisanja se ide do kraja toka.
`trunc`	pisanje	Ako tok nije bio prazan, odbacuje se sav postojeći sadržaj.
`binary`		Tok se upotrebljava u binarnom režimu. Ukoliko se ne navede ovaj način otvaranja, podrazumeva se tekstualan režim rada.

Tablica 5: Načini otvaranja datotečnih tokova

Posebnu lakoću upotrebi datotečnih tokova donose definisani konstruktori i destruktori. Pri konstrukciji toka se mogu navesti naziv datoteke i način otvaranja, pri čemu se tok automatski otvara. Pri destrukciji toka se tok automatski zatvara. Zbog toga se metodi za otvaranje i zatvaranje relativno retko eksplicitno upotrebljavaju.

Tip toka	Podrazumevani način otvaranja
`ifstream`	`ios_base::in`
`ofstream`	`ios_base::out`
`fstream`	`ios_base::in | ios_base::out`

Tablica 6: Podrazumevani načini otvaranja datotečnih tokova

Iako se klasa `fstream` može upotrebljavati i za čitanje i za pisanje, ne preporučuje se njena upotreba u slučajevima kada se tok koristi samo za čitanje ili samo za pisanje. Pored toga što upotreba klasa `ifstream` i `ofstream` pruža strožu proveru tipova, upotreba klase `fstream` može biti i nešto manje efikasna.

10.10.7 Memorijski tokovi

Posebnu vrstu tokova standardne bibilioteke predstavljaju *memorijski tokovi*. Nazivaju se i *tokovi za niske*. Memorijski tokovi predstavljaju tokove koji nemaju drugu fizičku reprezentaciju osim memorijske. Najčešće se upotrebljavaju za postepeno pravljenje formatiranih niski, koje se zatim mogu prosleđivati u neki drugi tok ili upotrebljavati u druge svrhe. Na raspolaganju su klase `istrstream` (tok za čitanje iz niske), `ostrstream` (tok za pisanje u nisku) i `strstream` (tok za pisanje u nisku i čitanje iz niske).

Ulazni memorijski tokovi se konstruišu navođenjem pokazivača na niz karaktera koji predstavlja sadržaj toka. Može se navesti i broj karaktera u toku:

```
istrstream( const char* s )
istrstream( const char* s, streamsize n )
```

Izlazni memorijski tok se pravi prazan ili inicijalizovan nizom karaktera date dužine:

```
ostrstream()
ostrstream( char* s, int n, ios_base::openmode = ios_base::out )
```

Ulazno-izlazni memorijski tokovi se konstruišu navođenjem pokazivača na niz karatera koji predstavlja početni sadržaj toka, veličine tog niza i opcionog načina otvaranja toka. Podrazumevani konstruktor pravi tok koji je inicijalno prazan:

```
strstream()
strstream( char* s, int n,
           ios_base::openmode = ios_base::out | ios_base::in )
```

Svi memorijski tokovi imaju metod koji vraća pokazivač na niz karaktera koji predstavlja sadržaj toka:

```
char* str()
```

Izlazni memorijski tokovi imaju metod koji vraća broj karaktera upisanih u tok:

```
int pcount() const
```

10.11 String

Podršku za niske u standardnoj biblioteci programskog jezika C++ predstavlja šablon klase `basic_string`. On predstavlja osnovu za sve tipove niski. Dve najčešće upotrebljavane instance ovog šablona su klasa `string`, definisana kao `basic_string<char>` i predviđena za rad sa 8-bitnim karakterima, i klasa `wstring`, definisana kao `basic_string<wchar_t>` i predviđena za rad sa 16-bitnim karakterima.

Zaglavlja

Šablon klase `basic_string` i klase `string` i `wstring` definisani su u zaglavlju `string`.

Parametri šablona

Šablon `basic_string` je definisan sa tri parametra:

```
template<
    class charT,
    class traits = char_traits<T>
    class Allocator = allocator<T>
    >
class basic_string;
```

Parametar `charT` određuje tip karaktera koji sačinjavaju nisku.

Parametar `traits` određuje, pre svega, način definisanja pomoćnih tipova podataka, a koji zavise od tipa karaktera.

Parametar `Allocator` određuje način alociranja nizova karaktera.

Šablon klase `basic_string` se relativno retko upotrebljava. Obično se upotrebljavaju klase `string` i `wstring`, koje su definisane na sledeći način:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Osnovne operacije

Operacije ćemo predstaviti na primeru klase `string`, ali sve napisano važi i za druge vrste niski. Najvažniji metodi klase `string` su:

- Konstruktor prazne niske:

  ```
  string()
  ```

- Konstruktor kopije:

  ```
  string( const string& s )
  ```

- Konstruktor niske koji inicijalizuje nisku prepisivanjem najviše `n` karaktera date niske `s`, počevši od pozicije `idx`. Verzija bez argumenta `n` prepisuje sve znakove do kraja niske `s`:

  ```
  string( const string& s, size_type idx, size_type n )
  string( const string& s, size_type idx )
  ```

- Konstruktor niske koji inicijalizuje nisku prepisivanjem karaktera iz niza karaktera na koji pokazuje pokazivač `s`. Verzija bez argumenta `n` prepisuje sve karaktere do kraja niske (tj. do znaka '`\0`'), dok verzija sa `n` ne obraća pažnju na eventualne znakove za kraj niske i kopira tačno `n` znakova. U oba slučaja pokazivač `s` ne sme biti 0:

  ```
  string( const char* s, size_type n )
  string( const char* s )
  ```

- Konstruktor niske koji inicijalizuje nisku sa `n` ponavljanja znaka `c`:

```
string( size_type n, char c )
```

- Konstruktor niske koji inicijalizuje nisku prepisivanjem karaktera iz opsega datih iteratora:

```
template<class InputIterator>
string( InputIterator begin, InputIterator end )
```

- Operator dodeljivanja i metodi koji zamenjuju vrednost niske datom niskom, znakom ili nizom karaktera:

```
string& operator=( const string& s )
string& operator=( const char* s )
string& operator=( char c )
string& assign( const string& s )
string& assign(const string& s,size_type idx,size_type n)
...
```

- Metod za efikasno razmenjivanje vrednosti dveju niski:

```
void swap( string &s )
```

- Operatori i metodi koji omogućavaju pristupanje karakteru sa datim indeksom:

```
const char& operator[](size_type pos) const
char& operator[](size_type pos)
const char& at(size_type pos) const
char& at(size_type pos)
```

- Operatori i metodi koji na nisku dopisuju dati karakter, datu drugu nisku, deo date druge niske ili karaktere iz opsega iteratora. Svi vraćaju referencu na izmenjenu nisku:

```
string& operator+=( char c )
string& operator+=( const char* s )
string& operator+=( const string& s )
string& append(const string& s,size_type pos,size_type n)
string& append( const string& s )
string& append( const char* s, size_type n )
string& append( const char* s )
string& append( size_type n, char c )
template<class InputIterator>
    string& append(InputIterator begin,InputIterator end)
```

- Više sličnih metoda koji u nisku, od pozicije sa indeksom `idx` ili pozicije na koju se odnosi iterator `pos`, umeću datu nisku ili niz karaktera:

```
string& insert( size_type idx, const string& s )
string& insert( size_type idx, const char* s )
string& insert( size_type idx, size_type n, char c )
string& insert( iterator pos, size_type n, char c )
template<class InputIterator>
    string& insert(
                iterator pos,
                InputIterator begin, InputIterator end
```

```
                    )
...
```

- Brisanje svih karaktera niske:

```
void clear()
void erase()
```

- Brisanje datog skupa karaktera iz niske. Skup karaktera može biti opisan indeksom ili iteratorima. Ako argumenti ne opisuju gde se nalazi kraj niske, brišu se svi karakteri do kraja niske:

```
string& erase( size_type idx )
string& erase( size_type idx, size_type n )
string& erase( iterator pos )
string& erase( iterator pos, size_type n )
```

- Zamenjivanje datog skupa karaktera niske datom niskom ili nizom karaktera:

```
string& replace( size_type idx, size_type n,
                 const string& s )
string& replace( size_type idx, size_type n,
                 const string& s,
                 size_type sidx, size_type sn )
string& replace( size_type idx, size_type n,
                 const char* s )
string& replace( size_type idx, size_type n,
                 const char* s,
                 size_type sidx, size_type sn )
string& replace( iterator begin, iterator end,
                 const string& s )
...
```

- Izračunavanje podniske koja počinje od pozicije `idx` i dugačka je `n` karaktera, ili se proteže do kraja niske (ako nije dato `n`):

```
string substr( size_type idx ) const
string substr( size_type idx, size_type n ) const
```

- Konkatenacija niski:

```
string operator+( const string& s1, const string& s2)
string operator+( const string& s1, const char* s2)
string operator+( const char* s1, const string& s2)
string operator+( const string& s, char c)
string operator+( char c, const string& s)
```

- Izračunavanje podniske koja počinje od pozicije `idx` i dugačka je `n` karaktera, ili se proteže do kraja niske (ako nije dato `n`):

```
string substr( size_type idx ) const
string substr( size_type idx, size_type n ) const
```

- Menjanje veličine niske. Ako se niska smanjuje, višak karaktera se briše. Ako se niska povećava, dopunjava se dopisivanjem datog karaktera potreban broj puta. Verzija bez datog karaktera popunjava nisku karakterom '`\0`':

```
void resize( size_type n )
void resize( size_type n, char x )
```

- Izračunavanje dužinu niske. Oba metoda imaju identično ponašanje:

```
size_type size() const
size_type length() const
```

- Metod koji proverava da li je niska prazna:

```
bool empty() const
```

- Metod koji izračunava kolika je maksimalna dopuštena veličina niske:

```
size_type max_size() const
```

- Izračunavanje veličine alociranog prostora za karaktere. Uvek je bar `size()`:

```
size_type capacity() const
```

- Promena veličine alociranog prostora za karaktere. Ako je `n` manje od dužine niske, ili nije dato, alocirani prostor se smanjuje tako da tačno odgovara dužini niske:

```
void reserve( size_type n ) const
void reserve() const
```

- Operatori za leksikografsko poređenje niski:

```
bool operator==( const string& s ) const
bool operator!=( const string& s ) const
bool operator<=( const string& s ) const
bool operator>=( const string& s ) const
bool operator<( const string& s ) const
bool operator>( const string& s ) const
```

- Više sličnih metoda koji porede nisku sa datom niskom, podniskom ili nizom karaktera. Izračunavaju 0 ako su niske jednake, negativnu vrednost ako je niska manja od date, ili pozitivnu vrednost ako je niska veća od date:

```
int compare( const string& s ) const
int compare( const char* s ) const
...
```

- Izračunavanje nizova znakova uobičajenih za programski jezik C. Metod `c_str` vraća pokazivač na nisku koja je garantovano terminisana nulom. Metod `data` vraća pokazivač na nisku znakova koja ne mora biti terminisana nulom.

```
const char* data() const
const char* c_str() const
```

- Prepisivanje najviše `n` karaktera iz niske u dati prostor, i to počevši od početka ili pozicije `idx`, ako je navedena:

```
size_type copy( char* buf,size_type n,size_type idx)const
size_type copy( char* buf,size_type n ) const
```

Traženje

Niske imaju više metoda za traženje. Svi metodi traže od početka prema kraju i vraćaju indeks pozicije na kojoj je pronađeno prvo pojavljivanje traženih karaktera. Ukoliko nije pronađen dati niz karaktera, rezultat je konstanta `string::npos`.

Većina metoda ima više verzija. Niska koja se traži može biti zadata kao niska tipa `string` ili kao pokazivač na niz karaktera. Ukoliko je potrebno da traženje počne od neke pozicije, umesto od početka, tada se zadaje argument `idx`. Neki metodi imaju i verziju koja traži od kraja prema početku, pa pronalazi poslednje pojavljivanje traženih karaktera. Takva verzija ima slovo `r` na početku naziva metoda.

- Traženje karaktera:

```
size_type find( char c ) const
size_type find( char c, size_type idx ) const
size_type rfind( char c ) const
size_type rfind( char c, size_type idx ) const
```

- Traženje podniske. Navodimo svih osam oblika metoda:

```
size_type find( const string& s ) const
size_type find( const string& s, size_type idx ) const
size_type find( const char* s ) const
size_type find( const char* s, size_type idx ) const
size_type rfind( const string& s ) const
size_type rfind( const string& s, size_type idx ) const
size_type rfind( const char* s ) const
size_type rfind( const char* s, size_type idx ) const
```

- Traženje prvog karaktera koji se nalazi i u datoj niski. Metodi koji traže u suprotnom smeru zovu se `find_last_of`:

```
size_type find_first_of( char c ) const
size_type find_first_of( const string& s ) const
```

- Traženje prvog karaktera koji se ne nalazi u datoj niski. Metodi koji traže u suprotnom smeru zovu se `find_last_not_of`:

```
size_type find_first_not_of( char c ) const
size_type find_first_not_of( const string& s ) const
```

Operacije sa tokovima

Za niske je definisano nekoliko operacija za rad sa tokovima. Sve su definisane van klase `string`, kao funkcije:

- Operator prosleđivanja niske u tok:

```
ostream& operator<<( ostream& strm, const string& str )
```

- Operator izdvajanja niske iz toka. Čita do prvog praznog znaka:

```
istream& operator>>( istream& strm, string& str )
```

- Operator izdvajanja reda iz toka. Karakter `delim` se smatra za kraj reda. Karakter `delim` se izdvaja iz toka ali se ne zapisuje u nisku:

```
istream& getline( istream& strm, string& str, char delim )
```

Iteratori

Definisani su osnovni tipovi iteratora: `iterator`, `const_iterator`, `reverse_iterator` i `const_reverse_iterator`. Postoje odgovarajući metodi za izračunavanje graničnih iteratora (postoje i konstantne verzije, ali nisu navedene):

```
iterator begin()
iterator end()
reverse_iterator rbegin()
reverse_iterator rend()
```

Povezani delovi standardne biblioteke

Klasa `string` ima dosta zajedničkih osobina sa klasom `vector`. Za razliku od vektora, niske nemaju metode za dodavanje i uklanjanje karaktera sa kraja i početka niske (`push_back`,...).

10.12 Izuzeci

Standardna biblioteka obuhvata nekoliko klasa izuzetaka koje sve nasleđuju klasu `exception`:

- `bad_cast` – neuspešna promena tipa;
- `bad_alloc` – neuspešna alokacija.
- `logic_error` – logička greška, obično posledica ozbiljnijih previda;
  - `domain_error` – greška u domenu ili tipu;
  - `invalid_argument` – neispravan argument;
  - `out_of_range` – podatak (indeks i sl.) van opsega;
  - `length_error` – neispravna dužina strukture;
- `runtime_error` – greška koja nastaje kao posledica specifičnog toka izvršavanja programa;
  - `overflow_error` – prekoračenje;
  - `underflow_error` – potkoračenje;
  - `range_error` – podatak van opsega (obično se ovaj tip koristi za dinamičke strukture, a `out_of_range` za statičke strukture);

Sve konkretne klase imaju konstruktor sa argumentom tipa `const string&`. Na primer:

```
explicit range_error( const string& )
```

Niska navedena pri konstrukciji se čuva u okviru objekta. Do nje se može doći primenom metoda `what`. Svaka konkretna klasa može kao rezultat ovog metoda vratiti ne samo nisku navedenu pri konstrukciji, već i neke dodatne informacije. Metod je definisan u baznoj klasi kao:

```
virtual const char* what() const
```

Ne preporučuje se dinamičko pravljenje objekata izuzetaka. Specifikacija standardne biblioteke preporučuje automatsko pravljenje objekata hijerarhije izuzetaka, na primer:

```
throw range_error( "...opis problema...");
```

11 - Zadaci za vežbu

11.1 Analitički izrazi

Napisati klase za podršku osnovnih operacija sa analitičkim matematičkim izrazima. U izrazima se mogu pojavljivati:

- konstante
  - vrednosti konstanti predstavljati tipom `double`;
- promenljive
  - svaka promenljiva ima ime;
  - nije ograničen broj promenljivih koje se mogu pojaviti u izrazu;
- uobičajene unarne i binarne operacije i funkcije
  - promena znaka,
  - sabiranje, oduzimanje, množenje deljenje;
  - pri čitanju i pisanju izraza, radi jednostavnosti, umesto operatora upotrebljavati odgovarajuće funkcije; na primer, izraz (x-a2)*(y+7.23) se može zapisati kao

```
(PUTA (MINUS x a2) (PLUS y 7.23))
```

  - funkcije `sin`, `cos`, `tan`, `ln`, `exp`;

Potrebno je obezbediti sledeće operacije na izrazima:

- izračunavanje vrednosti izraza za date vrednosti promenljivih;
- uprošćavanje izraza zamenjivanjem određenog skupa promenljivih datim vrednostima;
- osnovne aritmetičke operacije nad izrazima (nije neophodno predefinisati upotrebu simbola operatora);

- izračunavanje izvoda izraza po datoj promenljivoj.

Napisati program koji iz jedne datoteke čita izraz, iz druge skup vrednosti promenljivih, a u treću upisuje uprošćen izvod pročitanog izraza po promenljivoj `x`.

11.2 Testovi

Napisati program za testiranje uz pomoć računara. Test se sastoji od niza pitanja. Najpre se, redom, za svako pitanje postavlja pitanje i prihvata odgovor. Zatim se, na osnovu upisanih odgovora, ispisuje izveštaj o rezultatu testa.

Potrebno je podržati više različitih tipova pitanja:

- *Abcd pitalice* pored pitanja imaju i nekoliko ponuđenih odgovora. Tačan odgovor nosi 5 bodova, pogrešan –3, a izostavljen odgovor nosi –1 bod.
- *Tekstualno pitanje* podrazumeva postavljanje pitanja i unošenje teksta koji predstavlja odgovor. Izostavljanje odgovora se vrednuje sa –1 bodom, a naveden odgovor sa 0 bodova (očekuje se da bude manuelno pregledan).
- *Abcd redosled* pored pitanja ima i nekoliko ponuđenih pojmova. Kao odgovor se očekuje niz slova koja odgovaraju oznakama ispred ponuđenih pojmova i stoje u odgovarajućem poretku. Ako je bar jedno slovo dobro postavljeno, smatra se da je odgovor tačan i svako dobro postavljeno slovo nosi po 2 boda. Ako nijedno slovo nije na dobrom mestu, odgovor je netačan i nosi –3 boda. Izostavljanje odgovora vredi –1 bod.

Neophodno je podržati čitanje pitanja i testa iz toka, kao i prosleđivanje izveštaja o obavljenom testiranju u dati tok. Zapis testa u datoteci odgovara priloženom primeru.

Program iz datoteke čita test, testira korisnika i izveštaj o testiranju zapisuje u drugu datoteku. Zapis izveštaja odgovara navedenom primeru.

Prilog 1 – Primer datoteke sa pitanjima

Test je zapisan u tekstualnom obliku. U prvom redu je broj pitanja u testu, a zatim sledi odgovarajući broj pitanja. Svako pitanje u prvom redu ima oznaku vrste pitanja, a zatim slede tekst pitanja i ostale odgovarajuće informacije. *Abcd pitalica* je opisana tekstom pitanja, brojem ponuđenih odgovora, tačnim odgovorom i listom ponuđenih odgovora. *Tekstualno pitanje* je opisano samo tekstom pitanja. *Abcd redosled* je opisan tekstom pitanja, brojem ponuđenih pojmova, tačnim odgovorom i listom odgovarajućih pojmova. Tekstovi pitanja i ponudjeni odgovori i pojmovi se navode u po jednom redu.

```
5
AbcdPitalica
Šta je to 'std::map'?
3 b
a - Putokaz do gusarskog blaga.
b - Asocijativni niz u standardnoj biblioteci p.j. C++.
c - Močvarni teren opasan po život.
TekstualnoPitanje
```

```
Kojim programskim jezikom se bavi ova knjiga?
TekstualnoPitanje
Koliko znakova '+' stoji u nazivu p.j. C++?
AbcdPitalica
Koliko tačnih odgovora ima ovo pitanje?
3 c
a - Tri.
b - Dva.
c - Jedan.
AbcdRedosled
Poređati tipove od najmanjeg do najvećeg po broju bajtova.
4 badc
a - short
b - char
c - long double
d - long
```

Prilog 2 – Primer testiranja

Testiranje na osnovu prethodno predstavljenog testa bi trebalo da izgleda ovako:

```
*** Pitanje br. 1 ***
Sta je to 'std::map'?
------------------------------------------------
a - Putokaz do gusarskog blaga.
b - Asocijativni niz u standardnoj biblioteci p.j. C++.
c - Močvarni teren opasan po život.
------------------------------------------------
Upišite slovo koje stoji ispred tačnog odgovora
ili znak @ ako ne znate:
a

*** Pitanje br. 2 ***
Kojim programskim jezikom se bavi ova knjiga?
------------------------------------------------
Upišite odgovor u jednom redu ili @ ako ne znate:
ce dva plusa

*** Pitanje br. 3 ***
Koliko znakova '+' stoji u nazivu p.j. C++?
------------------------------------------------
Upišite odgovor u jednom redu ili @ ako ne znate:
@

*** Pitanje br. 4 ***
Koliko tačnih odgovora ima ovo pitanje?
------------------------------------------------
a - Tri.
b - Dva.
c - Jedan.
------------------------------------------------
Upišite slovo koje stoji ispred tačnog odgovora
ili znak @ ako ne znate:
c

*** Pitanje br. 5 ***
Poređati tipove od najmanjeg do najvećeg po broju bajtova.
------------------------------------------------
```

```
a - short
b - char
c - long double
d - long
-----------------------------------------------
Navedite tačan redosled odgovora upisivanjem slova
koja stoje ispred odgovora (npr: acbd) ili znak @ ako ne znate:
bacd
```

Prilog 3 – Primer izveštaja o testiranju

Za svako pitanje se ispisuje redni broj, osvojeni broj poena i komentar. Za *abcd pitalice* i *abcd redosled* komentar je jedan od tekstova „Tačan odgovor.“, „Netačan odgovor.“ i „Nije odgovoreno“. Za *tekstualno pitanje* komentar je „Nije odgovoreno“ ili tekst odgovora iza napomene „Odgovor je:“. Ako je odgovoreno, umesto broja bodova se ispisuje znak pitanja. Na kraju se ispisuje zbir bodova.

Za prethodno predstavljen primer testiranja izveštaj bi trebalo da izgleda ovako:

```
Rezultat testiranja:
-----------------------------------------------
1.  (-3 ) Netačan odgovor.
2.  ( ? ) Odgovoreno je: ce dva plusa
3.  (-1 ) Nije odgovoreno.
4.  ( 5 ) Tačan odgovor.
5.  ( 4 ) Tačan odgovor.
-----------------------------------------------
Rezultat: 5 poena
```

11.3 Logički izrazi

Napisati hijerarhiju klasa za predstavljanje operacija nad logičkim izrazima. U svim klasama hijerarhije obezbediti metode:

- `bool Tacan(const vector<bool>& x) const`
  proverava da li je logički izraz tačan ako promenljive x_0, x_1, ... uzimaju redom vrednosti iz vektora `x`. Pri tom se vrednost 0 uzima kao netačno, a vrednost 1 kao tačno;
- `int BrojPromenljivih() const`
  izračunava najveći od svih indeksa promenljivih koje se pojavljuju u tom logičkom izrazu, ili `-1` ako izraz ne sadrži promenljive;
- `IzrazOp* Kopija() const`
  izračunava kopiju izraza;

U baznoj klasi hijerarhije, klasi `IzrazOp`, napisati statički metod

```
bool sledeceX(vector<bool>& x)
```

koji nalazi sledeću vrednost niza promeljivih. Ona se dobija ako se vektor `x` posmatra kao binarni broj (tačno predstavlja cifru 1 a netačno cifru 0) koji se uvećava za 1. Metod vraća da li je pri tom uvećanju došlo do prekoračenja.

Napisati bar sledeće klase hijerarhije:

- `IzrazProm` – promenljiva koja učestvuje u izrazu; obezbediti konstruktor čiji je argument indeks promenljive;
- `IzrazT` – konstantna logička vrednost tačno;
- `IzrazNT` – konstantna logička vrednost netačno;
- `IzrazI`, `IzrazILI`, `IzrazNE` – binarne operacije konjunkcije ($\wedge$) i disjunkcije ($\vee$) i unarna operacija negacije ($\neg$). Za binarne operacije napisati konstruktore sa po dva argumenta, koji predstavljaju pokazivače na dva logička izraza – operanda operacije. U klasi `IzrazNe` napisati konstruktor sa jednim argumentom – pokazivačem na izraz koji se negira.

U baznoj klasi hijerarhije napisati statički metod za čitanje izraza iz toka:

```
static IzrazOp* Ucitaj()
```

Sve klase hijerarhije moraju imati metod za ispisivanje u tok:

```
void Ispisi(ostream&) const
```

Izrazi se zapisuju u prefiksnoj notaciji. Binarna operacija se zapisuje u obliku:

```
(OP Izraz1 Izraz2)
```

dok se unarne operacije zapisuju kao

```
(OP Izraz)
```

Same operacije se označavaju sa „`I`“, „`ILI`“ i „`NE`“. Logičke konstante se zapisuju sa „`T`“ i „`NT`“, dok se promenljive zapisuju kao „`X0`“, „`X1`“,...

Napisati klasu `Izraz` koja sakriva složenu strukturu prethodne hijerarhije izraza i pojednostavljuje rad sa izrazima. Ona sadrži pokazivač na `IzrazOp` i implementira odgovarajuće konstruktore, destruktor i sledeće metode i operatore:

- `bool Tacan(const vector<int>& x)`
  izračunava da li je izraz tačan; promenljive uzimaju redom vrednosti iz vektora `x`;
- `operator&`, `operator|`, `operator~`
  odgovaraju, redom, operacijama konjunkcije, disjunkcije i negacije izraza. Rezultati operacija su složeniji simbolički izrazi;
- operator dodeljivanja;
- `bool Tautologija() const`
  proverava da li je logički izraz tautologija. Izraz je tautologija ako ima vrednost tačno bez obzira na vrednsti promenljivih koje u njemu učestvuju;
- `bool Zadovoljiv()`
  proverava da li je logički izraz zadovoljiv, odnosno da li postoji bar jedna vrednost vektora promenljivih, koje u njemu učestvuje, za koju je on tačan;
- pisanje u tok i čitanje iz toka.

Napisati program koji testira napisane klase tako što iz datoteke, čiji se naziv dobija kao argument komandne linije, učitava izraz i ispituje da li je on zadovoljiv.

11.4 Prelom

Napisati program koji prelama dokument po redovima i stranama. Dokument se sastoji od grafičkih elemenata kao što su *reč* (neki niz znakova koji ne sadrži praznine), *blanko* (jedan prazan znak), *slika* (neka slika ili crtež, zadate veličine), *novi pasus* (označava da se sledeći element obavezno prenosi u naredni red).

Pri prelamanju dokumenta radi se po sledećim pravilima:

- U jedan red se stavlja maksimalan broj grafičkih elemenata koji može stati u red a da se ne prekorači predviđena širina reda;
- Visina reda odgovara najvišem elementu sadržanom u redu;
- Na jednu stranu se stavlja što je moguće više redova, a da se ne prekorači predviđena visina strane;
- Podrazumevati da se elementi u redu raspoređuju tako da su poravnati uz levu i donju ivicu reda; eventualne praznine potrebno je ostaviti uz desnu i gornju ivicu reda;
- U okviru strane redovi se ravnaju uz levu i gornju ivicu strane. Eventualne praznine potrebno je ostaviti uz desnu i donju ivicu strane;
- Ako je slika šira ili viša od strane, onda se (privremeno, bez menjanja originalne veličine!) njena veličina proporcionalno smanjuje tako da bude maksimalna moguća a da staje na stranu. Nije potrebno čitati slike iz datoteka;
- Pri ispisivanju reči uzimati u obzir tip i veličinu fonta. Ako je reč šira (ili viša) od strane, njena veličina se smanjivanjem fonta proporcionalno smanjuje tako da bude maksimalna moguća a da staje na stranu.

Nakon pravljenja preloma, na standardnom izlazu ispisati gde bi šta i kako trebalo crtati.

Pretpostavlja se postojanje funkcija (ne pisati ih) koje računaju širinu i visinu datog teksta pri ispisivanju datim fontom date veličine:

```
unsigned lib_SirinaTeksta(
                string tekst, string fontname, double size )
unsigned lib_VisinaTeksta(
                string tekst, string fontname, double size )
```

11.5 Transformacije slika

Na raspolaganju je biblioteka za rad sa slikama, deklarisana u datoteci `slika.h`, u kojoj su definisane klase za rad sa slikama. Slika je bitmapirana i ima oblik pravougaonika. Sastoji se od matrice tačaka. Pored podrazumevanog konstruktora, konstruktora kopije, operatora dodeljivanja i destruktora, klasa `Slika` ima definisane i sledeće metode:

```
Slika(unsigned sirina,unsigned visina);
unsigned Sirina() const;
unsigned Visina() const;
const Tacka& UzmiTacku( unsigned x, unsigned y ) const;
void PostaviTacku( unsigned x, unsigned y, const Tacka& t );
void Citanje( istream& istr );
void Pisanje( ostream& ostr ) const;
```

Klasa `Tacka` ima definisane operatore čitanja i pisanja:

```
istream& operator>>(istream&,Tacka&);
ostream& operator<<(ostream&,const Tacka&);
```

Potrebno je napisati klase za izvođenje operacija sa slikama. Operacije predstavljaju unarne transformacije slika. Mogu se kombinovati u složene operacije. Osnovni metodi operacija su:

- `Slika* operator () ( const Slika* s ) const`
  primenjuje operaciju na datu sliku `s` i vraća novu sliku koja je rezultat operacije;
- `Operacija* operator+ ( const Operacija& op2 ) const`
  pravi kompoziciju operacije i date operacije; ako su `op1` i `op2` dve operacije, i `Operacija* op3=op1+op2`, tada je `op2(op1(slika)) = (*op3)(slika);`
- `void Pisanje ( ostream& ostr ) const`
  zapisuje operaciju u tok u formatu `<naziv op.> <param1> <param2>`.

Potrebno je obezbediti i metode za pravljenje kopije operacije. Pri izračunavanju rezultata operacija koje po prirodi nisu diskretne već kontinualne (rotacija, promena veličine) potrebno je izvoditi interpolaciju tačaka.

Podržati bar sledeće operacije:

- `PromenaVeličine` – menja veličinu date slike u zadatu. Konstruktor ove klase ima dva argumenta: `unsigned ciljnaSirina` i `unsigned ciljnaVisina`, koji određuju veličinu rezultujuće slike;
- `RelativnaPromenaVeličine` – menja veličinu date slike množeći je datim koeficijentom. Klasa ima dva konstruktora: prvi ima jedan argument `float faktor`, koji određuje ciljnu veličinu slike kao (`faktor*sirina`, `faktor*visina`), a drugi ima dva argumenta `float faktorS` i `float faktorV`, koji određuju veličinu slike kao (`faktorS*sirina`, `faktorV*visina`);
- `Rotacija` – izvodi rotiranje slike. Klasa ima dva argumenta: `float ugao` je ugao rotacije u radijanima, a `Tacka& t` određuje boju dodatnih tačaka, tj. pozadine. Naime, rezultat uvek mora sadržati čitavu početnu sliku, pa ako rotacija nije za `n*pi/2`, onda se veličina rezultata određuje tako da on obuhtavi čitavu datu rotiranu sliku, dok se njena okolina popunjava tačkama koje imaju boju date tačke;
- `RotacijaLevo90`, `RotacijaDesno90` i `Rotacija180`, koje imaju konstruktore bez argumenata;

- `Isecanje` – iseca željeni deo slike. Ima konstruktor sa četiri argumenta tipa `unsigned`: `x0`, `y0`, `s` i `v`. Ova operacija izdvaja deo date slike sa koordinatama u datom opsegu: `x0≤x<x0+s`, `y0≤y<y0+v`. Ukoliko je data slika manja nego njen zahtevani deo, izbaciti izuzetak;
- `OgledaloH` i `OgledaloV` – izvode refleksiju slike oko horizontalne i vertikalne ose;
- `Kompozicija` – predstavlja kompoziciju dve operacije.

Obezbediti čitanje i pisanje operacija.

Napisati program `Transformisanje` koji se upotrebljava u formi:

```
Transformisanje <operacija> <slika> <slikarez>
```

i čita operaciju iz datoteke `<operacija>`, čita sliku iz datoteke `<slika>`, primenjuje operaciju na sliku i rezultat upisuje u datoteku `<slikarez>`.

11.6 Šah

Napisati kostur programa za igranje šaha. Pri tome smatrati:

- Šahovska tabla ima dimenzije 8x8 polja označenih po širini slovima A-H i po visini brojevima od 1 do 8. Na primer: `C3`;
- Potezi se opisuju navođenjem para polja: polja sa kojeg i polja na koje se pomera figura. Na primer: `C3E5`;
- Pod proveravanjem ispravnosti poteza podrazumeva se ispitivanje da li je potez u skladu sa osnovnim pravilima pomeranja figura. Potrebno je razmotriti i uklanjanje protivničkih figura.
- Figure se označavaju slovima, i to bele figure velikim a crne figure malim slovima: pešak-`P`, top-`T`, konj-`S`, lovac-`L`, dama-`D` i kralj-`K`;
- Boje figura opisuju se tipom: `enum boja { bela, crna };`
- U svim klasama članovi podaci moraju biti privatni ili zaštićeni.

Ideja rešavanja:

- Napisati klase (ili strukture) `Polje`, koja predstavlja koordinate jednog polja table, i `Potez`, koja obuhvata podatke o polju *sa* koga se pomera figura i polju *na* koje se pomera figura;
- Napisati klasu `Figura` kao baznu klasu hijerarhije klasa kojima se predstavljaju šahovske figure. Obezbediti metode koji računaju boju, oznaku i položaj figure, metod koji proverava da li je dati potez ispravan, kao i metod koji pomera figuru na dato polje;
- Napisati klase `Pešak`, `Top`, `Konj`, `Lovac`, `Dama` i `Kralj`. Ne implementirati metode za proveru dopustivosti poteza;

- Napisati klasu `Tabla`. Pored ostalih neophodnih metoda napisati i metod koji izračunava koja se figura nalazi na datom polju, metod koji odigrava dati potez, metod koji postavlja datu figuru na dato mesto i metod koji postavlja početni raspored figura;
- Pretpostaviti da postoji implementiran metod klase `Tabla` koji izračunava vrednost date pozicije. Uz primenu tog metoda napisati metod koji izračunava najbolji potez za igrača date boje;
- Napisati metod koji od korisnika traži da putem standardnog ulaza upiše ispravan potez za figure date boje.

11.7 Razlaganje grafičkih objekata na duži

Potrebno je obezbediti *prevođenje* dvodimenzionalnih likova u kolekcije duži kojima se ti likovi mogu *dovoljno tačno* predstaviti. Likove predstavljati hijerarhijom klasa likova. Sve klase hijerarhije moraju imati metode:

- `int Prevodjenje( TipKolekcijeDuzi& duzi, double eps )`
  najpre prazni kolekciju `duzi`, a zatim popunjava dužima kojima se aproksimira dati lik;
- `int PrevodjenjePlus( TipKolekcijeDuzi& duzi, double eps )`
  kolekciju `duzi` *dopunjava* dužima kojima se aproksimira dati lik;
- pri tome, parametar `eps` predstavlja najveće dopušteno odstupanje duži od lika koji se njome predstavlja.

Duži se predstavljaju strukturom:

```
struct Duz {
    int x0, y0, x1, y1, boja;
};
```

Za predstavljanje kolekcije duži može se upotrebljavati kolekcija po izboru (u tekstu zadatka se upotrebljava naziv `TipKolekcijeDuzi`).

Potrebno je podržati bar sledeće vrste likova sa odgovarajućim atributima:

- `Kvadrat` – ima boju, koordinate levog gornjeg temena i dužinu stranice;
- `Pravougaonik` – ima boju, koordinate levog gornjeg temena i dužine stranica;
- `Krug` – ima boju, koordinate centra i poluprečnik;
- `Poligon` – ima boju i koordinate svih temena poligona; ne postoji ograničenje broja temena poligona;
- `ParametarskaFunkcija` – ima boju, položaj grafika funkcije (položaj koordinatnog početka grafika u koordinatnom sistemu crteža), razmeru (dužina jedinične duži grafika u koordinatnom sistemu crteža), domen parametra (zadat kao zatvoren interval) i tekstualni zapis parametarske funkcije;

- pretpostavljati da se za izračunavanje vrednosti parametarske funkcije upotrebljava funkcija:

```
void VrednostParametarskeFunkcije(
        char* funkcija,
        double t, double& x, double& y )
```

koja za dati tekstualni zapis funkcije i vrednost parametra `t` izračunava vrednosti `x` i `y`. Radi testiranja programa napisati odgovarajuću funkciju koja izračunava uvek istu funkciju, nezavisno od parametra `funkcija`;

- Pri aproksimiranju lika funkcije dužima može se koristiti metod polovljenja segmenata domena (tj. odgovarajućih duži) do postizanja potrebne tačnosti. Tačnost je dovoljno procenjivati na osnovu položaja središta duži (naravno, to nie dovoljno za rešavanje realnih problema);

- `SlozeniLik` – kolekcija likova. Obavezno mora imati konstruktor kopije.

Indeks

Literatura

[Eckel 2000]
Bruce Eckel, **Thinking in C++**, 2.ed, *Prentice Hall*, 2000. Izdanje na srpskom jeziku: **Misliti na jeziku C++**, *Mikro knjiga*.

[Fowler 1999]
Martin Fowler, **Refactoring: Improving the Design of Existing Code**, *Addison-Wesley*, 1999. Izdanje na srpskom jeziku: **Refaktorisanje: Poboljšanje dizajna postojećeg koda**, *CET*.

[Fowler 2003]
Martin Fowler, **UML Distilled**, 3.ed, *Addison-Wesley*, 2003. Izdanje na srpskom jeziku: **UML ukratko**, *Mikro knjiga*.

[Gamma 1995]
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**, *Addison-Wesley*, 1995. Izdanje na srpskom jeziku: **Gotova rešenja: Elementi objektno orijentisanog softvera**, *CET*.

[Josuttis 1999]
Nicolai M. Josuttis, **The C++ Standard Library: A Tutorial and Reference**, *Addison-Wesley*, 1999.

[Kraus 2005]
Laslo Kraus, **Rešeni zadaci iz programskog jezika C++**, *Akademska misao*, 2005.

[Lippman 1999]
Stanley B. Lippman, **C++ Essentials**, *Addison-Wesley*, 1999. Izdanje na srpskom jeziku: **Osnovi jezika C++**, *CET*.

[Lippman 2005]
Stanley B. Lippman, Josee Lajoie, Barbara E. Moo, **C++ Primer**, 4.ed, *Addison-Wesley*, 2005. Izdanje na srpskom jeziku: **C++ Izvornik**, prevod 3. izdanja, *CET*.

[Martin 2003]
Robert C. Martin, **Agile Software Development: Principles, Patterns, and Practices**, *Prentice Hall*, 2003.

[Meyer 1997]
Bertrand Meyer, **Object-Oriented Software Construction**, 2.ed, *Prentice Hall PTR*, 1997. Izdanje na srpskom jeziku: **Objektno orijentisano konstruisanje softvera**, *CET*.

[Milićev 2001a]
Dragan Milićev, **Objektno orijentisano programiranje na jeziku C++**, *Mikro knjiga*, 2001.

[Milićev 2001b]
Dragan Milićev, Ljubica Lazarević, Jelena Marušić, **Objektno orijentisano programiranje na jeziku C++: Skripta sa praktikumom**, *Mikro knjiga*, 2001.

[Riel 1996]
Arthur J. Riel, **Object-Oriented Design Heuristics**, *Addison-Wesley*, 1996. Izdanje na srpskom jeziku: **Heuristike objektno orijentisanog dizajna**,*CET*.

[Shalloway 2004]
Alan Shalloway, James R. Trott, **Design Patterns Explained**, 2.ed, *Addison-Wesley*, 2004. Izdanje na srpskom jeziku: **Projektni obrasci**, *Mikro knjiga*.

[Stroustrup 2000]
Bjarne Stroustrup, **The C++ Programming Language**, spec.3.ed, *Addidson-Wesley Professional*, 3.ed, 2000.