

..Univerzitet u Beogradu
...Matematički fakultet

C++ kroz primere

Saša Malkov

...Beograd, 2005.

C++

Odlomci iz knjige u pripremi...

Saša Malkov

5.3 Lista

5.3.1 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 metode:
 - 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;
- operatora za dinamičko pravljenje objekata `new` i operatora za dinamičko uklanjanje objekata `delete`;
- koncepta skrivanja podataka.

5.3.2 Rešavanje zadatka

Nizom koraka ćemo formirati i unapređivati klasu `Lista`:

Korak 1 - Definisanje interfejsa klase <code>Lista</code>	102
Korak 2 - Struktura liste	104
Podrazumevani konstruktor	106

Korak 3 - Oslobađanje nepotrebnih resursa	107
Život objekata u programskom jeziku C++	108
Destruktor	110
Korak 4 - Kopiranje objekata	113
Plitko i duboko kopiranje	114
Konstruktor kopije	115
Operator dodeljivanja	116
Diskusija	120
Korak 5 - Unapređivanje pristupanja elemenatima liste	120
Korak 6 - Efikasno dodavanje elemenata na kraj liste	122
Korak 7 - Uklanjanje elemenata liste	123
Korak 8 - Prijateljske klase i umetnute klase	123
Korak 9 - Optimizacija	126

Korak 1 - Definisane interfejsa klase Lista

Lista je dinamička struktura 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 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 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 u način upotrebe klase što više razdvoje od implementacije, a podaci predstavljaju upravo strukturni deo implementacije, preporučuje se da se podaci ne uključuju u interfejs klase, 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 deo parametara može zameniti upotrebom podataka objekta ili izračunavanjem metoda koji opisuju stanje objekta koji primenjuje metod);
- 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 pocetak: " << 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.

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

Kada je definisano *šta* je to što klasa `Lista` radi, potrebno je da opišemo *kako* to radi. Već je napomenuto da je struktura 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. Poćemo od metoda `DodajNaPocetak`. Prvi posao koji pri tome moramo obaviti je pravljenje novog elementa liste. Najpre ga napravimo primenom operatora `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. Poćemo 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 nikako ga ne bi valjalo menjati:

```
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 ispravno izvedena.*

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? Pa 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 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 lista sa po milion elemenata, da bismo ih na kraju imali čitav milion (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.

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

- *Statički objekti* se prave u postupku učitavanja programa u memoriju, pre pozivanja funkcije `main`. Redosled njihovog 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 klase. Statički podaci klase su detaljnije opisani u **...referenca na statičke podatke klase...**.
- Svi lokalno definisani objekti predstavljaju tzv. *automatske objekte*. Pored njih, automatske objekte predstavljaju i *neimenovani privremeni objekti*¹. Automatski objekti se prave kada

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

se pri izvršavanju koda dođe do definicije promenljive čiji sadržaj predstavlja novi objekat. Zapravo, nije pogrešno definiciju lokalne promenljive smatrati za 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.

- *Dinamički objekti* su oni koji se prave primenom operatora `new`. Za razliku od statičkih i automatskih objekata čije je trajanje precizno definisano lokalnim segmentom koda, trajanje dinamičkih objekata je teže opisati². Kao što se eksplicitno prave primenom operatora `new`, dinamički objekti se i uklanjaju eksplicitno – primenom operatora `delete`.

Kada govorimo o pravljenju i uništavanju objekata, pri tome uvek mislimo na prve dve, odnosno poslednje dve faze života objekta. Ranije je već naglašeno da slično konceptu metoda konstruktora, koji obezbeđuju inicijalizaciju objekata pri njihovom pravljenju, postoji i koncept metoda *destruktor*a, koji obezbeđuju deinicijalizaciju objekata pri njihovom uklanjanju. Zapravo, svaki put kada se pravi objekat izvode se, redom, dve važne operacije – alokacija potrebne memorije i konstrukcija (inicijalizacija) objekta. Slično tome, svaki put kada se objekat uništava izvode se njima inverzne (bar po smislu) operacije u obrnutom redosledu – najpre destrukcija (deinicijalizacija) objekta i zatim oslobađanje memorije.

Primetimo da svaka definicija objekta predstavlja i alokaciju i konstrukciju statičkog ili automatskog objekta. Slično, izračunavanje konstruktora predstavlja i alokaciju i konstrukciju privremenog neimenovanog objekta, a primena operatora `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 izvodi 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;
}
```

napravljen. Razlika je samo u činjenici da takav objekat nije imenovan, pa mu se zbog toga ne može više puta neposredno pristupiti.

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 čitave klase.

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, 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 program, pa i segment programa, mora za sobom počistiti sve što nakon njegovog izvršavanja više nije potrebno. Isto pravilo važi i za objekte. Metod koji se stara da objekti počiste za sobom sve nepotrebne tragove naziva se destruktor.

*Destruktor je metod koji 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 ispred koga stoji simbol '~'. 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:

- 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 izvode oslobađanje resursa na koje referišu. Prvi i najčešći tip podataka na koji se ovo odnosi jesu pokazivači, pa možemo zaključiti: *ako struktura objekata 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 uvek valja biti oprezan sa pokazivačima, jer je suvišno oslobađanje deljenih resursa bar jednako loše kao i neoslobađanje privatnih resursa.

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

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

Ovakvo rešenje ima neke mane. Jedna je uglavnom kozmetičke prirode – standardom programskog jezika C++ predviđeno je da se pri pozivanju operatora `delete` automatski izvede provera 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 operator `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 destruktora klase `ElementListe`:

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

Sada je problem, bar u domenu teorije, rešen: kada destruktora objekta klase `Lista` eksplicitno pokrene uklanjanje prvog elementa liste, njegov destruktora 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 preposlednji element liste i tako redom sve do prvog elementa liste. Realnost je nešto neprijatnija, pa ovako napisani destruktora 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 dopustivu 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. 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 konkretan primer (gde imamo milion rekurzivnih poziva) dovešće do problema u skoro svakom okruženju.

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 život jednostavnijim.

Nerekurzivno rešenje podrazumeva da se u klasi `ElementListe` upotrebljava podrazumevani destruktora, tj. da se ne obezbeđuje nikakav eksplicitno definisan destruktora:

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

Naš program 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 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 u opštem slučaju to još uvek ne važi.

Korak 4 - Kopiranje objekata

Kao što se može zaključiti iz podnaslova, 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;

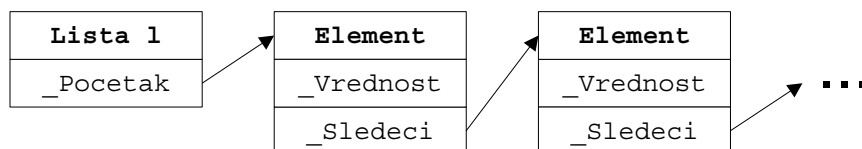
    return 0;
}
```

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 upravo opisan tok života objekata, primetićemo da se pri završavanju bloka uništavaju automatski objekti definisani u bloku. Upravo pri tome dolazi do greške.

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

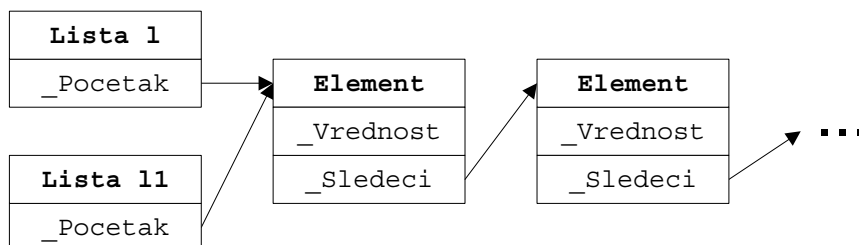
Tačno je da je destruktor ispravan, ali ovde nije u pitanju neispravnost u definiciji destruktora nego neispravnost konteksta u kome se on upotrebljava. Nakon što napravimo i popunimo listu l, naše objekte možemo predstaviti ovako:



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 predstavljeni na sledeći način:



Primetimo da će ovako napravljeni objekti klase `Lista` deliti 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 operatora `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 mogu dovesti do fatalnih grešaka u radu programa ili bar do problema koji mogu ugroziti ispravnost rezultata ili bezbednosni podsistem. Zbog toga operativni sistem onemogućava rad programa ukoliko se iz načina pozivanja funkcija operativnog sistema može zaključiti da program pristupa zonama memorije koje su prethodno oslobođene ili čak nikada nisu ni bile alocirane. Upravo zato će operativni sistem prijaviti problematičan pristup memoriji čim naš program pokuša da pristupa sadržaju uništenog prvog elementa liste, a do čega dolazi odmah po započinjanju uništavanja drugog objekta klase `Lista`.

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`.

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 destruktorka oslobađa*³. Značaj ovog pravila je sasvim lako uočiti ako se analizira šta se dešava ukoliko se ono ne poštuje:

- ako se resursi oslobađaju u destruktorki, a kopiranje se izvodi plitko, a ne duboko, tada će pri uništavanju iskopiranih objekata dolaziti do višestrukog oslobađanja resursa – ovo je baš onaj problem koji smo mi imali 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.

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:

³ 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 takvih 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 tako posmatrano 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.

```

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`).

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 uništavanje prethodnog sadržaja potrebno izvoditi na potpuno isti način kao što se izvodi deinicijalizacija 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 preduprediti. Zbog toga je dobro da se problem reši u samoj definiciji 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 u takvim situacijama je 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, razvijene su neke tehnike koje zabranjuju primenu konstruktora kopije i operatora dodeljivanja. Rešenje kojim se sprečava da se konstruktor kopije i operator dodeljivanja koriste 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 definicije 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 njoj 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) << ' ';
...
```

Pošto je metod `Element` prilično neefikasan, jer je složenost pristupanja elementu 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 odslikava 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.

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;
}
```

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 raspolažemo pokazivačem na prvi element liste, ali ne i pokazivačem na poslednji. Efikasnost dodavanja elemenata na kraj liste možemo značajno povećati jednostavnim 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 ObrisPrvi()
{
    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 ObrisPoslednji()
{
    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 ovde 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 deklaracije 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;
```

time ćemo naglasiti 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 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 klase unutar druge klase. Na taj način se korisniku jasno stavlja do znanja da tzv. *umetnuta 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` jer je jasno da se radi o listi, već je dovoljno da se zove `Element`.

```

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

void ObrisPoslednji()
{
    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 = &_amp;Pocetak;
        _Kraj = 0;
        while( stari ){
            _Kraj = *novi = new Element( stari->_Vrednost );
            stari = stari->_Sledeci;
            novi = &_amp;Kraj->_Sledeci;
        }
        *novi = 0;
        _BrojElemenata = l._BrojElemenata;
    }

...
```

```

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

```

Sada je moguće i brisanje elementa sa kraja liste učiniti malo jednostavnijim:

```

void ObrisiPoslednji()
{
    if( _BrojElemenata>=2 ){
        Element* pretposlednji = _Pocetak;
        for( int i=2; i<_BrojElemenata; i++ )
            pretposlednji = pretposlednji->_Sledeci;
        delete pretposlednji->_Sledeci;
        pretposlednji->_Sledeci = 0;
        _Kraj = pretposlednji;
        _BrojElemenata--;
    }
    else if( _BrojElemenata==1 ){
        delete _Pocetak;
        _Pocetak = _Kraj = 0;
        _BrojElemenata--;
    }
}

```

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

```

    }

void ObrisiPoslednji()
{
    if( _BrojElemenata>=2 ){
        Element* pretposlednji = _Pocetak;
        for( int i=2; i<_BrojElemenata; i++ )
            pretposlednji = pretposlednji->_Sledeci;
        delete pretposlednji->_Sledeci;
        pretposlednji->_Sledeci = 0;
        _Kraj = pretposlednji;
        _BrojElemenata--;
    }
    else if( _BrojElemenata==1 ){
        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;
    }
```

5.3.4 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 i praktično potvrđuje da je jedino ispravno pri definisanju klase 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. Uslediće 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 da *ako je za ispravno funkcionisanje klase neophodan neki od tri opisane metoda (destruktor, konstruktor kopije, operator dodeljivanja), tada su neophodna sva tri metoda*.

Pri implementiranju 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 rekursivno, 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, na primer:

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

U okviru standardne biblioteke za pristupanje elementima kolekcija upotrebljava se koncept iteratora. Taj koncept će biti detaljnije razmotren i primenjen u nekim od narednih primera. Naprednijim čitaocima predlažemo da pokušaju da primene koncept iteratora na klasi `Lista`.