...Univerzitet u Beogradu
...Matematički fakultet

C++ kroz primere

Saša Malkov

...Beograd, 2005.

C++
Odlomci iz knjige u pripremi...

Saša Malkov

5.2 Razlomak

5.2.1 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;
- podrazumevane vrednosti argumenata;
- listu inicijalizacija u konstruktorima;
- 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.

5.2.2 Rešavanje zadatka

Nizom koraka ćemo formirati 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.

```
//------------------------------------------------------------
//  Klasa Razlomak
//------------------------------------------------------------
class Razlomak
{
public:
    //------------------------------------------------------------
    // Članovi podaci
    //------------------------------------------------------------
    int Imenilac;
    int Brojilac;
};
```

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

```
//------------------------------------------------------------
// Glavna funkcija programa demonstrira upotrebu klase Razlomak.
//------------------------------------------------------------
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 formiranju i definisanju klasa predstavlja njihovo ponašanje, a ne njihov sadržaj ili interna struktura***. Primetimo da je prvi korak u opisivanju načinjen upravo u smeru definisanja interne strukture razlomka. 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 od njih 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 naslednicima će više reči biti nešto kasnije ...referenca na nasleđivanje...).

Deklarisana raspoloživost se odnosi na sve članove klase koji su definisani nakon te deklaracije, a pre naredne deklaracije raspoloživosti. Podrazumevana raspoloživost za klase u programskom jeziku C++ je privatna, a za 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. Ovde 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 izvodi 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 izvoditi samo konstantne metode.

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

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 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. Zato što se takvi metodi izvršavaju neposredno nakon alokacije memorije, a u fazi pravljenja novih objekata, nazivaju se *konstruktori*. Dakle, za inicijalizaciju novih objekata zaduženi su konstruktori.

U programskom jeziku C++ konstruktori se raspoznaju kao metodi čije je ime identično imenu klase. Pri deklarisanju konstruktora ne navodi se 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:
    //-------------------------------------------------------------
    // Konstruktori.
    //-------------------------------------------------------------
    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;
}
```

Podrazumevane vrednosti argumenata

U programskom jeziku C++ pisanje više sličnih metoda sa različitim brojem argumenata možemo pojednostaviti primenom *podrazumevanih vrednosti argumenata*. Ako u deklaraciji metoda iza imena argumenta 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 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, u našem primeru, 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 pojednostavili definisanje konstruktora, dok njihova upotreba ostaje potpuno ista kao i ranije.

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 nije navedena podrazumevana vrednost u deklaraciji.

Lista inicijalizacija

U telu našeg 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 pojednostavljujemo konstruktor klase `Razlomak`:

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

Primetimo da 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 jednom od narednih primera ...referenca na primer... .

Jednostavnom funkcijom `main` isprobavamo da li napisani kod ispravno funkcioniše.

```
#include <iostream>

using namespace std;

//-------------------------------------------------------------
//  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;
};
```

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 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:

- 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...;
- 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 poželjno 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 veoma često javlja. 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;
}
```

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 izvesti 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 izvodi sabiranje razlomaka 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 aspekt 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++ radije ćemo upotrebiti reference. ...referenca na 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 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`. Primetimo da rezultat sabiranja mora da predstavlja novi objekat pa se stoga nikako ne sme vraćati po referenci. Ako bismo pokušali da rezultat vratimo po referenci to bi imalo fatalne posledice po izvršavanje programa iako mnogi prevodioci ne bi prijavili ni grešku ni upozorenje. Naime, bio bi po referenci 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*. 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` izvodimo 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:

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

Operatori

Da bismo obezbedili prihvatljiviju sintaksu upotrebe 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 da *operatore treba definisati i njihova imena birati samo u skladu sa uobičajenim načinom razmišljanja o objektima koje opisujemo*. Recimo, nikako ne bi bilo dobro za operaciju sabiranja upotrebiti npr. 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 sabiranje razlomaka može izvoditi 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. Rezultat svih ovih operacija 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 izvode.

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 izvodi promena vrednosti brojioca, da bi se na kraju sačuvana originalna vrednost vratila kao rezultat. Zbog toga što je rezultat zapisan u privremenom objektu `r`, ovde se vraćanje rezultata ne sme izvoditi po referenci. Ovakvom 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; }
...
};
```

Upotreba ovog operatora se ni po čemu ne razlikuje od uobilčajenih 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 mi 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 može izvesti 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 izvodi konverzije u nekim spornim situacijama. Sa operatorima konverzije je jednostavno izaći na kraj – ako ne želimo da se konverzije izvode 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 (ako već nije moguće sve) 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 brzo se 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 tačno jednog operanda 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.

Prvi problem se može 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 prilično je značajna, pa je svakako valja 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 opsegom pri primeni operatora poređenja `==` i `!=` se takođe mogu eliminisati uvođenjem dodatnog zahteva, da razlomak uvek bude uprošćen (skraćen) koliko je to moguće (tj. da imenilac i brojilac budu uzajamno prosti). Međutim, ovaj zahtev je već prilično skup, jer je svaki put pri konstrukciji razlomka potrebno proveravati da li su imenilac i brojilac uzajamno prosti i menjati 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 `Koriguj` koji izvodi sve korekcije imenioca i brojioca, uključujući i prethodno opisanu korekciju u slučaju negativnog imenioca. Metod `Koriguj` 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)
        { Koriguj(); }
...
    istream& Citaj( istream& str )
        {
        char c;
        str >> _Brojilac >> c >> _Imenilac;
        Koriguj();
        return str;
        }
...
private:
    //-------------------------------------------------------------
    // Korigovanje podataka
    //-------------------------------------------------------------
    void Koriguj()
        {
        // 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 se poređenje operatorima == i != može izvoditi nezavisno od opsega:

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

```
};
```

Provera ispravnosti čitanja i pisanja obično se izvodi 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 ekslicitno označimo tok kao neispravan ...referenca na tokove... :

```
class Razlomak
{
...
    istream& Citaj( istream& str )
        {
        char c;
        str >> _Brojilac >> c >> _Imenilac;
        if( c != '/' )
            str.setstate( ios::failbit );
        else
            Koriguj();
        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 detaljnije obrađen tek u jednom od narednih primera ...referenca na izuzetke... . Zbog toga ćemo ovaj problem 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 dve semantičke pretpostavke čije se ispunjenje ostvaruje primenom metoda `Koriguj`, a čije bi narušavanje dovelo do neispravnog ponašanja operatora poređenja:

- imenilac mora biti pozitivan;
- brojilac i imenilac moraju biti uzajamno prosti.

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;
        Koriguj();
        }
    void PostaviBrojilac( int n )
        {
        _Brojilac = n;
        Koriguj();
        }
...
};
```

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.

Dakle, 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 `Koriguj`, 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 objedine 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;
        Koriguj();
        }
...
};
```

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 izvodi primenom enkapsulacije (čime se sprečava neposredno menjanje podataka koje može narušiti semantičke pretpostavke) i pažljivim izborom skupa metoda.

5.2.3 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, to je zato sto 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)
        { Koriguj(); }

    //---------------------------------------------------------
    // 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;
        Koriguj();
        }
```

```
//-------------------------------------------------------------
// 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
        Koriguj();
    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 Koriguj()
        {
        // 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;
}
```

5.2.4 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 `/=`.