# Sadržaj

**1. Ponešto o alokaciji memorije.......... 2**

1.1. Statička alokacija memorije.......... 2
1.2. Dinamička alokacija memorije.......... 2
1.3. Curenje memorije.......... 3

**2. Generičke funkcije.......... 5**

**3. Strukture.......... 7**

3.1. Strukture.......... 7

3.2. Čvorovi.......... 8
- 3.2.1. Sortirani unos čvorova.......... 9
- 3.2.2. Ispisivanje čvorova.......... 10

**4. Klase.......... 11**

4.1. Sintaksa klase.......... 11
4.2. Deklaracija i implementacija metoda jedne klase.......... 12

4.3. Konstruktori.......... 14
- 4.3.1. Inicijalizacione liste.......... 16
- 4.3.2. Ključna riječ `this`.......... 17

4.4. Destruktori.......... 18
4.5. Ključna riječ `friend` - prijateljske funkcije.......... 19
4.6. Rezime dosadašnjih lekcija i pojmova.......... 21

**4.7. Konstruktor kopije.......... 23**

4.7.1. Plitke i duboke kopije objekata.......... 23
4.7.2. Kad nastaju plitke kopije objekata?.......... 24
4.7.3. Implementacija konstruktora kopije.......... 26
4.7.4. Implementacija preklopljenog operatora dodjele.......... 27
4.7.5. Zabrana kopiranja objekata neke klase.......... 28
4.7.6. Konačna verzija klase `Niz`.......... 28

**4.8. Preklapanje operatora.......... 29**

4.8.1. Preklapanje unarnih operatora.......... 30
4.8.2. Preklapanje binarnih operatora.......... 31
4.8.3. Preklapanje operatora kod kojih prvi operand nije konstanta.......... 33
4.8.4. Preklapanje unarnih operatora "++" i "--".......... 34
4.8.5. Preklapanje operatora ulaza i izlaza.......... 36

# 1. Ponešto o alokaciji memorije

Čuj, alokacija memorije...kome to treba?! Vjerovatno sebi postavljaš to pitanje čitajući naslov ovog dijela skripte. Hmm...zapitajmo se prvo šta znači, zapravo, alocirati memoriju, prevedeno sa stručnog jezika na jezik običnog smrtnika.

Sama riječ *alokacija* prevedena sa latinskog jezika znači sastavljanje riječi *od* i *mjesto, lokacija*, ali iz toga se i baš ne može ništa skontati...

Probajmo skontat logički: šta radi, zapravo, računar prilikom *deklaracije* neke promjenljive? "Saznaje" veličinu, potrebnu za stvaranje takve promjenljive i nalazi prvo, za tu radnju, pogodno mjesto u svojoj memoriji. Ukoliko nema mjesta u memoriji, on će programera obavijestiti o tome. No, o tom po tom.

U programskom jeziku C++ postoje najmanje dva načina za alociranje memorije: *statička* i *dinamička* alokacija memorije.

## 1.1. Statička alokacija memorije

Ovin načinom si se sigurno koristio velik broj puta, a da to nisi ni primijetio. Naime, pri deklaraciji obične promjenljive tipa **int** ili bilo kog drugog tipa, računar alocira memoriju za tu promjenljivu i ona tada ima vrijednost sadržaja tog dijela memorije. Zato i promjenljive, tek kada se deklarijaju, imaju bruku nekih besmislenih brojeva ili znakova.

Logično je, ako ćeš izračunati preko petlje sumu nečega, da ćeš tu sumu smjestiti u neku promjenljivu. Ako zaboraviš deklarirati tu promjenljivu sa početnom vrijednosti 0, budi siguran da nećeš nikad dobiti tačan rezultat.

Primjeri statičke alokacije:

```
int neki_broj = 5;           // inicijalizacija pomoću operatora dodjele
int neki_broj2(5);           // inicijalizacija konstruktorom za tip int
int *pok = &neki_broj;       // pokazivač na promjenljivu tipa int koji
                             // 'u startu' pokazuje na 'neki_broj'
int *pok2 = 0;               // nul-pokazivač na int (ne pokazuje
                             // nigdje)
double niz[2];               // niz od 2 realna broja
int *niz_pok[10];            // niz od 10 pokazivača na int-ove
```

Pri ovim primjerima tačno je alocirano onoliko memorije, koliko je navedeno pri deklaraciji svake od promjenljivih, ali se ta memorija ne može osloboditi do kraja bloka, u kom je dotične promjenljive deklarirane.

## 1.2. Dinamička alokacija memorije

Sama riječ kaže, da se misli na *dinamiku*, a ne *statiku*. Razlika između ta dva pojma je ogromna.

Zamisli dva čovjeka, koji se zovu "Statički alocirana promjenljiva" i "Dinamički alocirana promjenljiva" i obojica stoje na ulici. Prvom čovjeku su cipele zalijepljene za asfalt, a onom drugom nisu. Ti kažeš obojici *'Maknite se sa ulice, nema više mjesta za druge ljude.'* Drugi čovjek će se odmah maknuti, dok će prvi čovjek morati čekati da lijepilo opusti.

Eh, isto tako statički alocirane promjenljive zauzimaju memoriju koja se ne može osloboditi *do kraja programa* (ako je u deklaraciji stavljena ključna riječ **static**) ili *se*

*brišu na kraju bloka* u kom su deklarirane. Međutim, dinamički alocirane promjenljive se *mogu brisati kad god je to potrebno* (tj. brišu se po naredbi programera).

Način deklaracije *dinamičkih promjenljivih* je vrlo jednostavan: deklarira se pokazivač u koji se operatorom dodjele '**=**' smješta adresa alocirane memorije, koja se opet dobiva preko operatora dinamičke alokacije, tj. operatora **new**.

Sintaksa je slijedeća:

```
tip *ime_promjenljive = new tip;
```

Primjeri dinamičke alokacije:

```
int      *neki_broj = new int;        // dinamička promjenljiva tipa int
double   *niz = new double[2];        // dinamički niz od 2 realna broja
int      **niz_pok = new int*[10];    // din. niz od 10 pokazivača na int-
                                      // ove, od kojih svaki može pokazi-
                                      // vati, recimo, na niz int-ova
```

Brisanje dinamičkih promjenljivih se vrši pomoću operatora **delete** ili **delete[]**, ovisno o tome, da li se briše *obična dinamička promjenljiva* ili *dinamički niz*. Ako je operand ovog operatora nul-pokazivač, neće se ništa desiti. Međutim, *ako se pokuša ponovo obrisati prethodno obrisana dinamička promjenljiva*, dolazi do "kraha" programa.

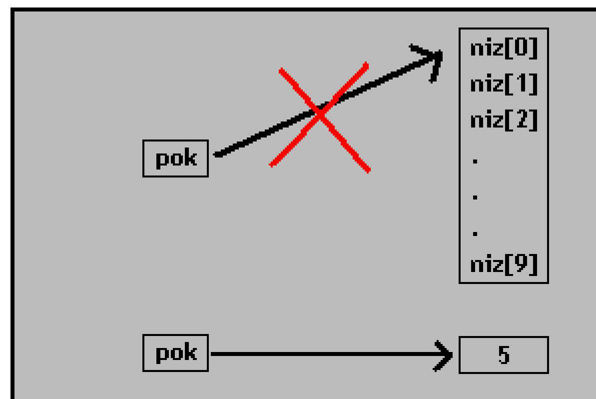
## 1.3. Curenje memorije

Curenje memorije se dešava kad god se izgubi kontrola nad dinamički alociranom memorijskom prostoru. Kako se može ta kontrola izgubiti? Fino, dokle god mi adresu tog memorijskog prostora čuvamo u nekom pokazivaču, mi mu možemo pristupati. Ukoliko promijenimo taj pokazivač, tj. proslijedimo ga na neku drugu adresu, gubimo pristup memorijskom prostoru na koji je prethodno pokazivao taj pokazivač. Taj prostor ostaje alociran, ali ga je nemoguće izbrisati (osloboditi), dolazi do curenja memorije (engl. *memory leak*), a time i do "kraha" programa.

Zamisli slijedeći kôd:

```
int *pok = new int[10];
    *pok = new int(5);
```

Izvođenjem tog kôda, memorija će izgledati ovako:




Šta se desilo? Napravili smo dinamički niz od 10 int-ova i njegovu adresu (adresu prvog elementa) smjestili u pokazivač ***pok**. Nakon toga smo napravili dinamičku promjenljivu tipa **int** sa početnom vrijednosti **5** i njenu adresu smjestili u tom istom pokazivaču ***pok**.

Pitaš se sad, u čemu je problem? Jednostavno, prethodno deklarirani dinamički niz je ostao u memoriji i zauzima tamo prostor. Međutim, niko ne može pristupiti tom nizu, jer je "cijev" (pokazivač) kroz koju su tekli podaci iz memorije do programa pukla. Ajd se sad zapitaj, zašto se zove *curenje memorije*

Zaključak: Uvijek vodi računa o tome da "ne lomiš te cijevi", tj. da ne izgubiš adresu neke dinamički alocirane memorije, a pogotovo da ne pokriješ tu adresu nekom drugom adresom.

Prethodni problem se mogao zaobići npr. ovako:

```
int *pok = new int[10];
delete[] *pok;
*pok = new int(5);
```

Ovdje nema curenja memorije

Sad se postavlja pitanje, kada koristiti statičku, a kada dinamičku alokaciju memorije? Naime, odgovor na ovo pitanje zavisi od toga koliko planiraš memorije zauzeti izvođenjem tvog programa. Ako ćeš koristiti više-dimenzionalne nizove, kao npr. matrice, ili ako planiraš napraviti vezanu listu elemenata, onda ti je najbolje rješenje za to dinamička alokacija. Pogotovo, ako ne znaš unaprijed koliko će čega biti.

# 2. Generičke funkcije

Recimo, želimo da napišemo funkciju koja će izvšiti iste algebarske operacije nad brojevima. Kao što znamo, brojevi nisu samo varijable tipa **int**, već i **float**, **long**, **double** i još neki izvedeni. Ukoliko bi postupak vršenja algebarskih operacija za sve te tipove bio isti, ali bi funkcija primala samo parametre određenog tipa i njena povratna vrijednost bi zavisila od određenog tipa – mi bi morali napisati za svaki tip posebnu funkciju.

Autori programskog jezika C++ su nam omogućili da izbjegnemo to "izmišljanje vruće vode" tako što se koristimo generičkim funkcijama.

Princip generičke funkcije jeste u tome, da se napravi "šablonska funkcija", gdje se koristit neki po volji nazvani (nepostojeći) tip, koji se koristi umjesto stvarnog tipa. Tako kompajler može pod određenim uvjetima saznati o kojem se tipu radi ili mu mi eksplicitno navedemo o kojem je tipu riječ, i onda on formira funkciju identičnu toj i zamjeni naš proizvoljni (nepostojeći) tip, sa tim tipom.

Najbrži mogući način pisanja generičkih funkcija je taj, ako si iskusan programer i znaš je napisati smjesta. Za programere koji tek uče pisanje generičkih funkcija, najefikasniji način je – da se prvo napiše standardna funkcija, pa se samo zamjeni standardni tip, sa proizvoljnim tipom.

Sintaksa pisanja generičkih funkcija je:

```
template<typename Ime_proizvoljnog_tipa>
- glava i tijelo funkcije, ali se standardni tip
  mijenja sa tipom Ime_proizvoljnog_tipa
```

Slijedi jednostavan primjer. Funkcija prima tri parametra:

1. prva dva parametra predstavljaju brojeve
2. treći parametar predstavlja realan broj, s kojim se se podijeliti suma prva dva parametra

```
// prva verzija predstavlja unikatnu funkciju, koja samo može primati
// parametre tipa int
int SaberiPodjeli(int Broj1, int Broj2, double Djelilac) {
      int rezultat = (Broj1 + Broj2) / Djelilac;
      return rezultat;
}
```

Preobrazit ćemo ovu unikatnu funkciju u generičku:

```
template<typename NekiTip>
NekiTip SaberiPodjeli(NekiTip Broj1, NekiTip Broj2, double Djelilac) {
      NekiTip rezultat = (Broj1 + Broj2) / Djelilac;
      return rezultat;
}
```

Aha...nije se puno promijenilo u samoj funkciji. Međutim, za kompajlera se promijenilo mnogo. Sada smo veliki dio našeg posla predali kompajleru. Neki se sigurno sad pitaju – a zašto Dinko, pametnjakoviću, nisi ono "**double**" ispred trećeg parametra promijenio u univezarni tip? Eh, pa...vrlo jednostavno, malo više gore, piše – treći parametar predstavlja realan broj, pa ako predstavlja realan broj, onda će bit realan broj ma bilo koji tip koristio za prva dva parametra .

Kako koristiti sad ovu funkciju? Jednostavno. Evo nekih primjera kako se eksplicitno navodi tip, koji se koristi:

```
int     tri       = SaberiPodjeli<int>    (   4,     5, 3.0);
double  cetiri    = SaberiPodjeli<double> ( 2.5,   7.5, 2.5);
float   sest_i_po = SaberiPodjeli<float>  (4.25, 8.75, 2.0);
```

Jednostavno, zar ne?

Preostaje mi samo još da objasnim onaj slučaj, kada se dinamička alokacija koristi i za generičke funkcije. Princip je isti. Zato ću krenuti odmah sa primjerom.

U slijedećem primjeru radi se o generičkoj funkciji koja dinamički alocira niz od **n** elemenata i popunjava sve njegove elemente na vrijednost, zadanu drugim parametrom, te vraća pokazivač na prvi elemenat kao povratnu vrijednost.

Unikatna (standardna) funkcija bi izgledala ovako:

```
int* DinamickiNiz(int BrojElemenata, int vri) {
      int *niz = new int[BrojElemenata];
      for(int i(0); i < BrojElemenata; i++)
         niz[i] = vri;
      return niz;
}
```

Naravno, slijedi verzija ove funkcije koja je univerzalna (generička):

```
template<typename UniTip>
UniTip* DinamickiNiz(int BrojElemenata, UniTip vri) {
      UniTip *niz = new UniTip[BrojElemenata];
      for(int i(0); i < BrojElemenata; i++)
         niz[i] = vri;
      return niz;
}
```

Primjer kako se koristi ta generička funkcija:

```
int     *niz_intova   = DinamickiNiz<int>       (5, 8);
double  *niz_realnih  = DinamickiNiz<double>    (5, 3.5);
```

Mislim da je ovo potpuno dovoljno o generičkim funkcijama. Ukoliko vas zanimaju sitnice i detalji, pročitajte Jurićevo predavanje br. 5.

# 3. Strukture

*Šta je to OOP* ili *objektno-orijentirano programiranje?* Sâm naziv kaže da se pažnja posvećuje objektima. *Strukture* i *klase* (a i *pobrojani tipovi*) se jednim imenom zovu *korisnički definirani tipovi*.

Kada se deklarira varijabla koja je tipa nekog korisnički definiranog tipa, ona se naziva objekat. Jer, njenim atributima, metodama i ostalim članovima se pristupa operatorima pristupa ("`.`", "`::`" i "`->`").

Princip OOP-a je korištenje objekata, da bi se realizirali neke stvare iz običnog života u programskim jezicima, ali i realizacija programskih riješenja koristeći korisnički definirane tipove.

## 3.1. Strukture

Struktura je tip podataka, koji sastavlja više promjenljivih istih ili različitih tipova u jednu cjelinu – strukturu, ništa više.

Recimo, želiš da smjestiš 10 datuma negdje u memoriji. To bi značilo deklaraciju od 3 niza sa po 10 int-ova (svaki niz redom za dane, mjesece i godine). Deklaracija sama od sebe nije problem, kao ni rukovanje sa tri niza. Šta bi bilo kada bi umjesto 10 datuma želio sačuvati podatke (ime, prezime, redni broj, prosječna ocjena, broj izostanaka) o 10 učenika? 100% bi bilo naporno rukovati sa toliko nizova. Zato fino napraviš strukturu.

Sintaksa «pravljenja» strukture:

```
struct ime_strukture {
      neki_tip ime_polja_1;
      neki_tip ime_polja_2;
      .
      .
      .
      neki_tip  ime_polja_n;
};
```

Tačka-zarez (engl. *semicolon*) na kraju strukture su **obavezni**.

Deklarirana struktura se ponaša kao *novi tip podataka*. S tim, što se svakom polju pristupa *operatorima pristupa*: '**->**' (crtica-veće, strelica) ili '**.**' (tačka). Koji način treba koristiti, zavisi od toga da li promjenljiva tipa te strukture predstavlja pokazivač (**->**) ili *l-vrijednost* (**.**).

Primjer za tip podataka **datum**:

```
using namespace std;

struct datum {
   int dan;
    int mjesec;
   int godina;
};
```

```
int main() {

      datum d1;
      d1.dan      = 6;
      d1.mjesec   = 5;
      d1.godina   = 2006;

      datum *d2   = d1;       // 'd2' pokazuje na 'd1'
      d2->godina  = 1987;     // sada 'd1' sadrži datum 6.5.1987

      return 0;
}
```

Ako znaš tačno kako izgleda neka struktura, onda njena polja možeš popuniti odmah prilikom deklaracije, i to pomoću vitičastih zagrada, tj. istom sintaksom kao pri pravljenju pobrojanih tipova:

```
datum d1 = {6, 5, 1987};
```

U strukturama, polja mogu biti se deklarirati i funkcije članice, pored članica promjenljivih (ili atributa). Međutim, kada se u tijelu neke strukture deklarira funkcija, ta struktura za kompajlera programskog jezika C++ postaje klasa. Zašto onda ne bi koristio odmah klasu umjesto strukture?

### 3.2. Čvorovi

Šta bješe čvorovi? Je'l to ono kada uzmem šnjuru pa joj svežem krajeve? Ili je to sad neka nova vrsta petlje? Pa jedva sam skontao **for**-petlju, a sad me još peglate sa novom petljom...

Hmm...ni jedno, ni drugo. Vjeruj mi, ne znam zašto se zove čvor, kad je to, ustvari, *vezana lista*. Vezana lista je niz povezanih struktura. Članovi vezane liste su povezani tako što sadrže pokazivač na vlastiti tip podataka.

Recimo, trebaš napraviti vezanu listu cijelih brojeva i trebaju biti pri unosu sortirani. Prvo što ti mora pasti na pamet, je da napraviš strukturu, u koju možeš smjestiti cijeli broj i koja se opet može preko pokazivača na svoj tip, povezati sa drugom takvom strukturom.

Sintaksa je slijedeća:

```
struct ime_strukture {
      int ime_polja_za_cijeli_broj;
      ime_strukture *ime_polja_za_vezu;
};
```

Sada si svom programu omogućio da u svaku promjenljivu tipa *ime_strukture*, pored cijelog broja, smjesti adresu neke druge promjenljive tipa *ime_strukture*.

Recimo da je naša struktura slijedeća:

```
struct Cvor {
      int broj;
      Cvor *veza;
};
```

### 3.2.1. Sortirani unos čvorova

Princip pravljenja sortirane vezane liste (cijelih brojeva) bi bio slijedeći:

```
/* trebamo znati gdje nam je početak liste, kao i negdje moramo smjestiti
   vrijednost svakog novog elementa liste */
Cvor  *pocetak = 0;
int    novi_el;

// beskonačna petlja
while(true) {
   // unesi neki broj i smjesti ga u novi_el
   cin >> novi_el;
   // ako je taj broj = 0 prekini petlju
   if(novi_el == 0) break;

   /* napravi dinamičku promjenljivu tipa Cvor, kao novog elementa
      liste, i unesi njene početne vrijednosti */
   Cvor *novi_clan = new Cvor;
   /* pošto je dinamička promjenljiva pokazivač, za pristup njenim
      elementima koristimo operator pristupa strelica (->) */
       novi_clan->broj = novi_el;
       novi_clan->veza = 0;    // u startu je to nul-pokazivač

   // Je li novi_clan prvi član ove liste?
   if(pocetak == 0)
       pocetak = novi_clan;

   /* Ako ima početni član, da li novi_clan treba postaviti, zbog
      uslova sortiranja, ispred početnog člana? */
   else if(novi_clan->broj < pocetak->broj) {
      // DA: novi_clan se stavlja ispred početnog člana
      novi_clan->veza = pocetak; // poslije novi_clan dolazi početni
      pocetak = novi_clan;    // početak liste je sada novi_clan
   }

   // Ako novi_clan nije početni član, niti treba ići ispred početnog...
   else {
   /* for-petljom bez brojača, tražimo prvo pogodno mjesto, za smješta-
      nje novog člana. Ukoliko se takvo mjesto ne nađe, novi član se
      upisuje na kraj vezane liste (član, čija veza je nul-pokazivač) */
      Cvor *tmp;
      for(tmp = pocetak;
           tmp->veza != 0 && tmp->veza->broj < novi_clan->broj;
           tmp = tmp->veza) {}
      /* petlja je završila i u pomoćnom pokazivaču tmp->veza se nalazi
         pogodno mjesto za postavljanje novog člana, pa ćemo to i uradit */
      novi_clan->veza = tmp->veza;
      tmp->veza = novi_clan;
```

> Šta se ovdje, zapravo, desilo?
>
> Naime, iza novog člana je "postavljen" član, koji je bio povezan sa **tmp**. A na njegovo mjesto(tj. iza člana **tmp**) je postavljen **novi_član**, tako da se dobio slijedeći redoslijed članova liste:
>
> **početak**
> -> članovi prije člana **tmp**
> -> **tmp**
> -> **novi_clan**
> -> **tmp**->*veza*
> -> članovi poslije člana **tmp**->*veza*
> -> **0** (kraj liste)

```
    }  // kraj else-bloka
}  // kraj beskonačne while-petlje
```

To je sva mudrost *unošenja elemenata u vezanu listu*.

### 3.2.2. Ispisivanje čvorova

Jedino što treba još znati je, kako ispisati sad sve te elemente. Vrlo jednostavno. Znamo tačno tri stvari koje su nam potrebne: početni član liste, krajnji član ima polje `veza = 0` i svaki član pokazuje na svog slijedbenika (polje ***veza***). Znači, to ispisivanje bi se moglo na slijedeći način odraditi:

```
Cvor *tmp;
for(tmp = pocetak; ; tmp = tmp->veza) {
      if(tmp == 0) break;     // kraj vezane liste je i kraj petlje
      cout << tmp->broj << endl;
}
```

# 4. Klase

Kao što sam naveo u prvom dijelu ove skripte, strukture postaju klase, kada se u njima pored atributa (varijabli članica), deklariraju i metode (funkcije članice). Međutim, stvarna namjena klasa je u tehnikama programiranja, kao što su konstruktorske funkcije, preopterećivanje (engl. *overloading*) operatora, naslijeđivanje drugih klasa i još mnogo toga.

Ne obaziri se sad na sve ove pojmove, većina njih će biti, jedan po jedan, objašnjena u ovoj skripti.

No, upoznajmo se prvo sa nekim bitnim stvarima:

- **deklaracija metode** – to je dio kôda, koji kompajleru govori o povratnom tipu i parametrima metode. Može se još nazvati *prototip metode*. Ovaj dio se uvijek nalazi u tijelu klase;

- **implementacija metode** – to je onaj dio kôda, koji kompajleru govori o tome šta metoda neke klase radi. Može se još nazvati *implementacija metode*. Ovaj dio se najčešće nalazi izvan klase, mada se može izvršiti i odmah u tijelu klase. Postoje programeri, koji svaku metodu neke klase definiraju odmah u tijelu te klase;

- **tijelo klase** – je dio kôda, koji kompajleru govori o izgledu neke klase, tj. mjesto gdje se deklariraju/definiraju atributi, metode, konstruktori i ostali članovi neke klase.

U narednom tekstu skripte će se koristiti gore navedeni izrazi i fraze.

## 4.1. Sintaksa klase

Sintaksa klase je gotovo ista kao i sintaksa strukture. "Glava" i "noge" tijela klase se ne razlikuju puno od onih za strukture:

```
class Neka_klasa {
   deklaracija/definicija_1
   deklaracija/definicija_2
   .
   .
   deklaracija/definicija_N
};
```

Tijelo klase sačinjavaju, među ostalom, njeni atributi i metode. Oni mogu biti vidljivi ili nevidljivi korisniku klase. To zavisi od toga da li se u tijelu klase iznad njihove deklaracije navede ključna riječ **`public:`** ili **`private:`**.

Slijedi primjer klase koja može u sebi čuvati dva cijela broja i dva slova.

```
class Klasa {
  private:
    int neki_broj;
  public:
    char slovo;
    int neki_broj2;
  private:
    char slovo2;
};
```

U ovom primjeru atributi `neki_broj` i `slovo2` su sakriveni, dok ostali atributi nisu. Šta to znači? To znači da će kompajler prijaviti grešku pri pokušaju pristupa sakrivenom atributu.

Kompajler će prijaviti i grešku pri izvođenju slijedećeg kôda, negdje u **`main`** funkciji programa, i to na mjestu crvene linije kôda:

```
Klasa k1;
// popunjavaju se atributi klase
k1.slovo = 'd';
k1.neki_broj = 4;
k1.neki_broj2 = 5;
```

Deklaracijom varijable `k1` (tipa `Klasa`) smo "oživili" klasu, tj. njeno tijelo je dobilo dušu (napravili smo **objekat** tipa te klase).

Međutim, bilo je potrebno da inicijaliziramo naknadno njene atribute, jer su oni na početku puni nekih besmislenih brojeva i slova. To će se kasnije moći izbjeći primjenom konstruktora.

**Napomena**: pošto su *po pravilu* svi članovi tijela neke klase *sakriveni* (dok se eksplicitno ne navede suprotno u njenom tijelu), gornju klasu smo ekvivalentno mogli i ovako zapisati:

```
class Klasa {
   int neki_broj;
   char slovo2;
public:
   int neki_broj2;
   char slovo;
};
```

### 4.2. Deklaracija i implementacija metoda jedne klase

Ova tema je nešto kompleksnija. Počnimo od toga da se deklaracija/definicija metoda mogu vršiti na dva načina: odmah (u tijelu klase) i naknadno (prototip metode u tijelu klase, a definicija metode izvan tijela klase). Međutim, koji god se način koristio, izvršni kôd (tijelo) metode će biti isti.

No, prvo se posjetimo sintakse za deklaraciju funkcije, uopće:

```
povratni_tip ime_funkcije (neki_tip parametar_1,
      neki_tip parametar_2,
      ...
      ...
      neki_tip parametar_n)

{  // tijelo funkcije - početak
      ...
      // izvršni kôd funkcije
      ...
      return povratna_vrijednost;
}  // tijelo funkcije - kraj
```

Pri tome, **povratni_tip** je **void**, ukoliko funkcija ne vraća nikakvu vrijednost, isto tako – spisak parametara može biti prazan ili samo **void** (nema razlike), ukoliko funkcija ne prima parametre.

Pored toga, *za parametre se* već u glavi funkcije *može navesti početna vrijednost* i takvi se parametri onda pri pozivanju date funkcije mogu izostaviti iz sintakse. Početna vrijednost nekog parametra se navodi *operatorom dodjele* (=), u obliku:

```
neki_tip parametar = početna_vrijednost
```

Konačno, funkcija koja bez *tijela*, naziva se *prototip funkcije* i na kraju deklaracije se obavezno stavlja tačka-zarez (;).

Primjer, gdje se odmah definira metoda u tijelu klase:

```
class Prozor {
   bool otvoren;
public:
   void ZatvoriSe() { otvoren = false; }
   bool Promjena(int broj = 1)      // početna vrijednost parametra
                                     // broj je zadana (= 1)
   {
      for(int i(0); i < broj; i++)
         otvoren = !otvoren;  // negacija varijable otvoren
      return otvoren;
   }
};
```

Primjer za istu klasu, samo što su metode *deklarirane* u tijelu klase, a *definirane* izvan tijela klase, primjenom **operatora rezolucije pripadnosti** (::), koji ne radi ništa drugo nego govori kompajleru koja metoda pripada kojoj klasi:

```
class Prozor {
   bool otvoren;
public:
   void ZatvoriSe();
   bool Promjena(int broj = 1);
};

void Prozor::ZatvoriSe() {
   otvoren = false;
}

bool Prozor::Promjena(int broj = 1)
{
   for(int i(0); i < broj; i++)
      otvoren = !otvoren; // negacija
   return otvoren;
}
```

Kao što se da primjetiti, sintaksa za naknadnu definiciju metode neke klase je:

```
povratni_tip_metode ime_klase::prototip_metode
{
   // tijelo metode
};
```

**Savjet:** Kada budeš pisao svoje klase, koristi oba načina definicije metoda, istovremeno. Metode, čije tijelo čini jedna ili dvije linije kôda, definiraj odmah u tijelu klase, a one druge definiraj izvan klase

### 4.3. Konstruktori

Jesi li se ikad zapitao zašto se u C++ programskom jeziku varijable nekih tipova, npr. **int**, mogu incijalizirati i sintaksom zagrada?

Npr. umjesto:

```
int neki_broj = 20;
```

se može koristiti i tzv. *konstruktorska sintaksa inicijalizacije* neke varijable:

```
int neki_broj(20);
```

Razlog toga: mogućnost korištenja klasa u C++ programskom jeziku, naveli su programere tog jezika na ideju da osnovne tipove "preobraze" u klase i tako korisniku omoguće prednosti kompleksnih tipova, u odnosu na obične tipove.

Time su napisali konstruktore za te klase, i tako je od običnog tipa **int** postala klasa **int**, u kojoj je definiran konstruktor sa jednim parametrom (vrijednost varijable pri deklaraciji, tj. inicijalna vrijednost).

Konstruktori su, kratko i jasno, metode koje se pozivaju pri deklaraciji jedne inačice neke klase (varijabla tipa te klase).

Korisnik klase navodi koji će se konstruktor pozvati (naravno, ukoliko ih ima više). C++ nam omogućava definiciju više konstruktora za jednu te istu klasu, međutim, nijedan od njih ne smije biti identičan drugom.

*Konstruktori* se deklariraju/definiraju u tijelu odgovarajuće klase, i smatraju se konstruktorima, ako i samo ako se deklariraju kao *metode koje imaju isto ime kao i klasa* i obavezno se deklariraju *bez povratnog tipa*.

Slijedi klasa za kompleksni broj, sa tri konstruktora, koji će svi biti definirani u tijelu klase:

```
class Kompleksni_broj {
   double realni_dio;
   double imaginarni_dio;
public:
   // 1. konstruktor bez parametara
   Kompleksni_broj() { imaginarni_dio = realni_dio = 0; }
```

```
    // 2. konstruktor sa jednim parametrom
    Kompleksni_broj(double realni)
       { realni_dio = realni; imaginarni_dio = 0; }
    // 3. konstruktor sa dva parametra
    Kompleksni_broj(double realni, double imaginarni)
       { realni_dio = realni; imaginarni_dio = imaginarni; }
};
```

**Napomena:** Obojio sam dijelove kôda u plavu boju da bi naznačio bitnost rečenice "Da bi neka metoda bila konstruktor, ona se mora zvati isto kao i klasa u kojoj je deklarirana."

Sada se postavlja pitanje, kako koristiti te konstruktore? Na slijedeći način:

```
Kompleksni_broj z1();         // pozvati će 1. konstruktor
Kompleksni_broj z2(3.2, 2.0); // pozvati će 3. konstruktor
Kompleksni_broj z3(2.5);      // pozvati će 2. konstruktor
```

Moguće je definirati i konstruktore *sa istim brojem parametara*, ali pri tome svaki konstruktor mora imati parametar drugog tipa:

```
class Slovo_ili_broj {
    char slovo; int broj;
public:
    Slovo_ili_broj(char sl) { slovo = sl; }
    Slovo_ili_broj(int br) { broj = br; }
};
```

Kao što vidiš, imaš dva konstruktora sa jednim parametrom, ali različitog tipa. Kompajler će sâm prepoznati koji konstruktor treba pozvati, prilikom deklaracije varijable tipa `Slovo_ili_broj`:

```
Slovo_ili_broj sib1(5);      // pozvati će 2. konstruktor
Slovo_ili_broj sib2('d');    // pozvati će 1. konstruktor
```

**Savjet:** Malo prije, kada smo definirali klasu `Kompleksni_broj` , umjesto tri konstruktora mogli smo koristiti samo jedan, koji bi ujedinio sva tri slučaja. Koristili bi se pri tome onom tehnikom, da navedemo početnu vrijednost parametara neke funkcije:

```
class Kompleksni_broj {
    double realni_dio;
    double imaginarni_dio;
public:
    Kompleksni_broj(double realni = 0, double imaginarni = 0)
       { realni_dio = realni; imaginarni_dio = imaginarni; }
};
```

### 4.3.1. Inicijalizacione liste

Ove dvije riječi predstavljaju ništa drugo nego posebnu sintaksu za definiciju konstruktora neke klase (ili objekta). Činjenica je da je tu tehniku lakše objasniti nego je napisati ili pročitati

"Fazon" je u tome da se može skratiti programerovo pisanje konstruktora, tako što mu se omogućuje da navede *inicijalizacionu listu*, između glave i tijela konstruktora neke klase.

Inicijalizacione liste se pišu posebnom sintaksom, a služe za inicijalizaciju nekih atributa klase. Njihova sintaksa je jednostavnija od konstruktorske sintakse.

Recimo, pišeš klasu koja čuva u sebi četiri cijela broja (datum rođenja i broj godina). Također, zamisli da želiš korisniku klase omogućiti deklaraciju varijabli tipa te klase preko dva konstruktora. Prvi konstruktor ne prima parametre i treba da inicijalizira atribute u formatu "`1.1.2006 0`". Drugi konstruktor prima četiri parametra i treba da inicijalizira sve brojeve na zadane vrijednosti, respektivno.

Intuitivno bi se to moglo napisati ovako:

```
class Brojevi {
   int dan, mjesec, godina, godine;
public:
   Brojevi()
      { dan = 1; mjesec = 1; godina = 2006; godine = 0; }
   Brojevi(int d, int m, int g, int br_god)
      { dan = d; mjesec = m; godina = g; godine = br_god; }
};
```

Kao što sam naveo, inicijalizacione liste trebaju da olakšaju inicijalizaciju atributa klase i da skrate pisani kôd za konstruktor.

Primjer, u odnosu na prethodni:

```
class Brojevi {
   int dan, mjesec, godina, godine;
public:
   Brojevi(): dan(1), mjesec(1), godina(2006), godine(0) {}
   Brojevi(int d, int m, int g, int br_god):
      dan(d), mjesec(m), godina(g), godine(br_god) {}
};
```

Kao što vidiš, poslije prototipa konstruktora umjesto uobičajenog "tačka-zareza" (`;`) staviš dvotačku (`:`) i navodiš inicijalizacionu listu.

Međutim, niko tebi ne brani da u definiciji konstruktora miješaš te dvije varijante: *eksplicitno* (preko inicijalizacione liste) i *implicitno* (u samom tijelu konstruktora).

**Napomena:** U slučaje da se konstruktor *ne definira* odmah u tijelu klase, ali u deklaraciji ima navedenu inicijalizacionu listu, *ne mora* se ta lista ponovo navoditi pri definiciji tog konstruktora kasnije u kôdu. Inicijalizacione liste se praktično navode samo u tijelu klase.

### 4.3.2. Ključna riječ `this`

Gruba definicija: ova ključna riječ predstavlja pokazivač klase na samu sebe i koristi se za eksplicitni pristup članovima neke klase.

Moja definicija: prilikom poziva metode neke klase, prije početka izvođenja te metode se formira varijabla koja predstavlja pokazivač na objekat, nad kojim je pozvana ta metoda, tako da se može pristupiti tom objektu i unutar njegovih metoda.

Ali, najbolje se nešto može slikovito, tj. preko kôda objasniti:

```
class Klasa {
   int broj1;
   int broj2;
   int broj3;
public:
   Klasa(int broj2);    // prototip konstruktora sa parametrom
                        // koji se zove isto kao jedan od
                        // atributa klase
};

// definicija konstruktora
Klasa::Klasa(int broj2) {
   // pošto se javlja konfuzija između atributa klase broj2 i
   // parametra konstruktora broj2, koristimo ključnu riječ
   // this za eksplicitni pristup atributu klase
   this->broj2 = broj2;
}
```

Kako to da se primjenjuje operator pristupa strelica (->) na ključnu riječ **`this`**?

Pa rekli smo, da ta ključna riječ predstavlja pokazivač klase na samu sebe, a u prvom dijelu skripte smo rekli, da se za pristup članova pokazivača na neku strukturu, kao i klasu koristi operator pristupa strelica.

Ključna riječ **`this`** se automatski formira prilikom deklaracije neke klase. Za našu, gore navedenu klasu, bi to izgledalo ovako:

```
Klasa k1(5);
```

Kompajler je automatski formirao pokazivač **`this`**, koji je sopstven samo objektu **`k1`**, i koji se može koristiti samo u definicijama metoda te klase **`Klasa`**. Pretpostavljamo da to kompajler na slijedeći način radi:

```
Klasa *this = &k1;
```

**Zaključak:** ključnu riječ **`this`** koristi samo, ako želiš eksplicitno pristupiti članu neke klase ili ako želiš onome ko čita tvoj kôd naznačiti da se taj član upravo pripada toj klasi.

### 4.4. Destruktori

Destruktori su, isto kao i konstruktori, automatske metode. Šta to znači?

Kompajler poziva te metode u za to predviđene trenutke. Na nama je, samo, da ih deklariramo i definiramo.

Destruktore kompajler poziva prije brisanja varijable tipa neke klase koja koristi *dinamičku alokaciju* u sklopu svojih atributa. Tvoj zadatak je, ako si ijedan od atributa svoje klase dinamički alocirao, da definiraš destruktorsku metodu, u kojoj oslobađaš svu (klasom) alociranu dinamičku memoriju.

Klasa može imati više konstruktora, ali samo je jedan destruktor sopstven svakoj klasi i on se mora deklarirati *bez parametara*.

Destruktor se deklarira tako kao konstruktor bez parametara, s tim što se na početak deklaracije stavi znak *tilda* (~).

Slijedi primjer klase za čuvanje dva dinamička niza, koja koristi dinamičku alokaciju memorije za svoje atribute, i koja sadrži odgovarajući destruktor:

```
class DvaNiza {
   int *niz1;           // ovdje će se smjestiti adresa 1. niza
   int *niz2;           // ovdje će se smjestiti adresa 2. niza
   int br_elemenata;    // broj elemenata dinamičkih nizova
public:
   DvaNiza(int br_elemenata);
   ~DvaNiza();          // deklaracija destruktora
};

// definicija konstruktora
DvaNiza::DvaNiza(int br_elemenata) {
   // prave se dva dinamička niza sa istim brojem elemenata
   niz1 = new int[br_elemenata];
   niz2 = new int[br_elemenata];
   this->br_elemenata = br_elemenata;
}

// definicija destuktora
DvaNiza::~DvaNiza() {
   // brišu se dinamički nizovi
   delete[] niz1;
   delete[] niz2;
}
```

Zašto uopće destruktore definirati, kad se sve dinamički alocirane varijable brišu (najkasnije) automatski na kraju programa? Jeste, to stoji. Ali program ne može obrisati dinamičku varijablu, ako si ti kao programer negdje u kôdu izgubio adresu te dinamičke varijable (sjeti se *curenja memorije*).

Slijedi primjer za curenje memorije, nastalo nedostatkom destruktora u klasi koja koristi dinamičku alokaciju memorije za čuvanje svojih atributa.

Zamisli prethodnu klasu *bez destruktora* i onda zamisli da izvedeš ovaj kôd:

```
{ // otvaramo blok kôda
      /*
         deklariramo statičkom alokacijom varijablu tipa
         DvaNiza preko prvog konstruktora (tj. stvorit će
         se dva dinamička niza negdje u memoriji)
      */
      DvaNiza dn1(100);
} // zatvaramo dati blok, kompajler briše varijablu dn1,
  // jer je ona bila statički alocirana u tom bloku
```

Sad se pitaš u čemu je problem?

U slijedećem:

Sve statički alocirane varijable se brišu na kraju bloka u kome su deklarirane. Znači, da će se varijabla dn1 izbrisati na kraju bloka. Međutim, kada smo je deklarirali, koristili smo jedan od njenih konstruktora, koji je dinamički alocirao memoriju za dva dinamička niza od po 100 **`int`**-ova.

Adrese tih nizova se nalaze u atributima klase. Kompajler je obrisao datu varijablu, a *pošto smo rekli da ćemo smatrati datu klasu bez destruktora*, nigdje se neće izbrisati ta dva dinamička niza. Izgubili smo njihove adrese. Dolazi do curenja memorije i time do kraha programa.

Destruktori se isto kao konstruktori, ne mogu pozivati *eksplicitno* (operatorom pristupa), kao da su obična metoda klase. To nije dozvoljeno u C++ programskom jeziku.

Ukoliko imaš potrebu, da dinamički alociran prostor nekog objekta oslobađaš eksplicitno, onda pisanje metoda koja će to izvršiti ne možeš izbjeći. Ali, onda imaš utjehu da ne moraš dva put pisati isti kôd i za tu metodu i za destruktor, već u destruktoru samo pozoveš tu metodu.

### 4.5. Ključna riječ `friend` - prijateljske funkcije

Ova ključna riječ je uvedena iz jednog prostog razloga. Nekad se javlja programeru potreba da pristupi skrivenim članovima neke (članovi koji su deklarirani u sekciji **`private:`** u tijelu neke funkcije smatraju se skrivenim članovima).

Kao što znamo, kompajler programskog jezika C++ nam ne dâ da pristupamo takvim članovima (atributi i metode). Zato su uvedeni pojmovi prijateljskih funkcija i klasa.

U narednom primjeru slijedi jedna obična klasa sa jednim skrivenim atributom, koja simulira otpornik. Ispod nje slijede dvije funkcije za računanje paralelne i serijske veze dva otpornika, i vraćaju pokazivač na dinamičku varijablu tipa te klase:

```
class Otpornik {
   double Otpor;
public:
   Otpornik(double Otpor) {this->Otpor = Otpor;}    // konstruktor
   double VratiOtpor() {return Otpor;}    // vraća Otpor otpornika
};
```

```
Otpornik *ParalelnaVeza(const Otpornik &Otp1,
                        const Otpornik &Otp2) {
   // deklaracija dinamičke varijable tipa Otpornik
   Otpornik *rezultat = new Otpornik(0);
   // računa se ekvivalentni otpor
   rezultat->Otpor = Otp1.Otpor * Otp2.Otpor /
                    (Otp1.Otpor + Otp2.Otpor);
   return rezultat;
}

Otpornik *SerijskaVeza(const Otpornik &Otp1,
                       const Otpornik &Otp2) {
    // deklaracija dinamičke varijable tipa Otpornik
    Otpornik *rezultat = new Otpornik(0);
    // računa se ekvivalentni otpor
    rezultat->Otpor = Otp1.Otpor + Otp2.Otpor;

    return rezultat;
}
```

Prvo da vas riješim dvaju dilema:

- obje funkcije vraćaju dinamički alociran objekat tipa **Otpornik** i zato je obavezna zvijezdica (*) ispred povratnog tipa funkcija, jer se radi o pokazivačima;

- obje funkcije kao parametre primaju objekte nekog korisnički definiranog tipa i zato je poželjno korištenje referenci (&), jer time se kompajleru označava da ne treba kopirati stvarne parametre u formalne parametre, već koristiti originale. Ključna riječ **const** ispred deklaracije parametra označava da funkcija neće (ne smije) mijenjati vrijednosti atributa njenih parametara, već ih samo čitati;

Kao što smo do sad naučili, kompajler će prijaviti grešku na linijama sa crvenim dijelovima kôda, jer je zabranjen pristup skrivenim atributima.

Da bi riješili taj problem, proglasit ćemo ove dvije funkcije *prijateljima* klase **Otpornik**. To ćemo opet uraditi tako što ćemo prototipe ove dvije funkcije deklarirati također i u klasi **Otpornik**, s tim da ćemo ispred deklaracije staviti ključnu riječ **friend**.

Sada bi naša klasa trebala izgledati ovako:

```
class Otpornik {
   double Otpor;
public:
   double VratiOtpor() { return Otpor; }
   // prijateljske funkcije ove klase
   friend Otpornik *ParalelnaVeza( const Otpornik &Otp1,
                                   const Otpornik &Otp2);

   friend Otpornik *SerijskaVeza(  const Otpornik &Otp1,
                                   const Otpornik &Otp2);
};
```

Obrati pažnju na tačku-zarez na kraju deklaracija prototipova. Tačka-zarez se svakako stavlja, kada se deklarira funkcija/metoda bez svog tijela (prototip).

Sada kompajler neće prijaviti grešku, kada ove dvije funkcije budu pristupale skrivenim članovima svojih parametara.

### 4.6. Rezime dosadašnjih lekcija i pojmova

Eh, da bi mogli nastaviti sa "škakljivijim" temama OOP-a, bilo bi poželjno napraviti rezime do sada naučenih pojmova OOP-a.

Jer slijede objašnjenja programerskih tehnika i pojmova kao što su: plitke kopije, konstruktori kopije, preklapanje operatora (dodjele, unarnih i binarnih).

Ovaj rezime obuhvata dinamičku alokaciju u klasama, konstruktore i destruktore, u obliku kompletnog i komentiranog C++ programa:

```
#include <iostream>
#include <cstdlib>

using namespace std;

class matrica {
   int **elementi;
   int redovi, kolone;
public:
   matrica(int redovi, int kolone);
   ~matrica();

   int &Elemenat(int red, int kolona);

   void IspuniNizom(int *niz);

   void IspisiMe();
};
```

- `int **elementi;` – Dualni pokazivač na int (npr. adresa za niz nizova tipa int), za smiještanje elemenata matrice.
- `matrica(int redovi, int kolone);` – Konstruktor sa dva parametra.
- `~matrica();` – Destruktor (jer će se kasnije koristiti dinamička alokacija memorije).
- `int &Elemenat(int red, int kolona);` – Metoda koja vraća referencu na traženi element.
- `void IspuniNizom(int *niz);` – Metoda koja ispunjava atribut **elementi**, ako joj se kao parametar zada jednodimenzionalni niz int-ova.
- `void IspisiMe();` – Ispisuje matricu na ekranu.

```
matrica::matrica(int redovi, int kolone)
{
   elementi = new int*[redovi];
   for(int i(0); i < redovi; i++)
      elementi[i] = new int[kolone];

   this->redovi = redovi;
   this->kolone = kolone;
}
```

- `elementi = new int*[redovi];` – Prvo se napravi dinamički niz pokazivača na int...
- `for` petlja – pa se u njega smjeste adrese dinamičkih nizova, od kojih svaki predstavlja jedan red elemenata date matrice.
- `this->redovi = redovi;` / `this->kolone = kolone;` – Pošto su imena parametara ista kao i određeni atributi klase, koristi se ključna riječ **this** za pristup atributima klase.

```
matrica::~matrica()
{
   for(int i(0); i < redovi; i++)
      delete[] elementi[i];
   delete[] elementi;

   elementi = 0;
   redovi = kolone = 0;

}
```

- `delete[] elementi[i];` – Prvo se brišu svi nizovi za redove elemenata...
- `delete[] elementi;` – pa se izbriše niz koji je čuvao adrese tih redova.
- `elementi = 0;` – Atribut **elementi** sada predstavlja nul-pokazivač.
- `redovi = kolone = 0;` – Nema više redova i kolona.

```
int& matrica::Elemenat(int red, int kolona)
{
    if(red < 1 || red > redovi)
      throw "\nGreska u broju redova!";
    if(kolona < 1 || kolona > kolone)
      throw "\nGreska u broju kolona!";

    return elementi[red-1][kolona-1];
}
```

Povratna vrijednost metode je referenca na varijablu tipa int.

Prvo se provjerava validnost parametara...

ako su parametri validni, vraća se referenca na traženi element matrice.

```
void matrica::IspuniNizom(int *niz)
{
    int indeks(0);

    for(int red(0); red < redovi; red++)
       for(int kol(0); kol < kolone; kol++)
          elementi[red][kol] = niz[indeks++];
}
```

Metoda kao parametar prima pokazivač na **int**, tako da se može navesti niz intova kao parametar.

Redom se popunjavaju redovi matrice sa elementima navedenog niza. Brojač **indeks** se povećava svakim ciklusom petlje.

```
void matrica::IspisiMe()
{
    for(int r(0); r < redovi; r++) {
      cout << endl;
       for(int k(0); k < kolone; k++) {
          cout.width(5);
          cout << elementi[r][k];
       }
    }
    cout << endl;
}
```

Metoda ispisuje redom elemente matrice. Širina svakog ispisa iznosi **5** znakova.

```
int main() {
    // deklaracija dinamički alocirane
    // varijable tipa matrica
    matrica *m = new matrica(3, 3);

    // popunjavanje elemenata
    int niz[] = {  3, 2, 12,
                   1, 15, 6,
                   2, 8, 4   };
    m->IspuniNizom(niz);

    m->IspisiMe();

    // kraj programa
    system("PAUSE");
    return 0;
}
```

Ovo je tzv. sintaksa pobrojanih vrijednosti. Kao kod deklaracije pobrojanog tipa.

Prikazuje na ekranu poruku "Press any key to continue" i čeka da korisnik pritisne tipku da se završi program.

# 4.7. Konstruktor kopije

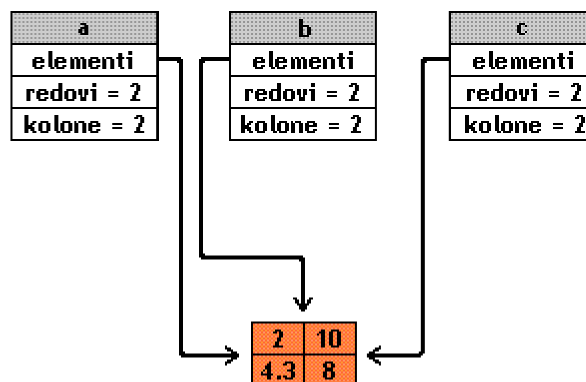
Konstruktor kopije je specijalan slučaj konstruktora klase sa jednim parametrom.

Da bi znali suštinu konstruktora kopije, potrebno je poznavati razliku između plitke i duboke kopije.

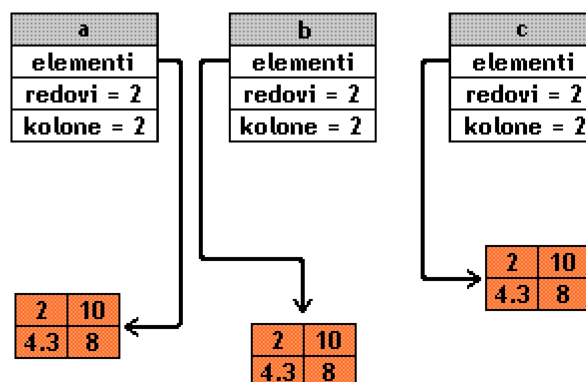
## 4.7.1. Plitke i duboke kopije objekata

I plitke i duboke kopije su kopije jednog objekta. Međutim, ako dati objekat sadrži pokazivače na dinamički alociran niz, onda:

- kod **plitkih kopija** svaka kopija pokazuje na isti memorijski prostor kao i izvorni objekat.




- Kod **dubokih kopija** se za svaku kopiju pravi sopstveni, ali isti takav niz elemenata, i onda sve kopije sadrže pokazivače na različite memorijske prostore, u kojima je smješten jedan te isti podatak.




Po pravilu, kompajler pravi plitke kopije, jer kada naiđe na pokazivač, kompajler ne može znati o čemu se radi (niz, niz nizova itd.) i time koliko čega treba kopirati. Zato, jednostavno samo kopira vrijednost pokazivača.

Zato su nam autori programskog jezika C++ omogućili, da "uhvatimo" pokušaj pravljenja plitkih kopija i – ili ga zabranimo korisniku klase – ili napravimo, umjesto plitke, duboku kopiju na tom mjestu.

Sad, kada znamo razliku između plitke i duboke kopije, valja skontati zašto su one štetne.

### 4.7.2. Kad nastaju plitke kopije objekata?

C++ pod različitim uslovima automatski vrši plitko kopiranje objekata, koje se može izbjeći korištenjem konstruktora kopije i preklopljenog operatora dodjele.

Plitke kopije su štetne za program, jer ukoliko se uništi bar jedna plitka kopija (recimo na kraju bloka gdje je napravljena), pozvati će se njen destruktor (ukoliko ga posjeduje) i osloboditi sav dinamički alociran memorijski prostor – koji se alocirao u tom objektu, što će stvoriti *viseći pokazivač* za sve ostale kopije, a time i za originalni objekat.

Zamislimo da imamo neku klasu koja čuva neki dinamički niz elemenata i posjeduje određeni destruktor za brisanje tog niza, koji se poziva prije uništavanja tog objekta. Zamislimo da se ta klasa zove **`Niz`**.

Recimo da ta klasa izgleda poput sljedeće:

```
class Niz {
      int *p_niz;
      int br_el;
public:
      // konstruktor
      Niz(int BrojElemenata) {
            br_el = BrojElemenata;
            p_niz = new int[br_el];
      }
      // destruktor
      ~Niz() {delete[] p_niz;}
      // vraća referencu na određeni element
      int& element(int indeks) {return p_niz[indeks];}
      // vraća broj elemenata, potreban za druge radnje
      int broj_elemenata() const {return br_el;}
};
```

U tom slučaju, plitko kopiranje se može izvršiti svijesno, kao i nesvijesno, na slijedeća *četiri načina*:

1. Kada se iskoristi postojeći objekat za inicijalizaciju novog:

```
Niz a(5);
Niz b = a;
```

Pošto nije implementiran konstruktor kopije za klasu **`Niz`**, kompajler će jednostavno napraviti plitku kopiju objekta **`a`** u objekat **`b`** i time će oba objekta pokazivati na istu dinamički alociranu memoriju, tj. njihovi pokazivači na tu memoriju će sadržavati identične adrese.

Ukoliko se uništi objekat `b` iz ovog slučaja, osloboditi će se dinamički alocirana memorija, na koju pokazuje i originalni objekat `a`.

Originalni objekat će sadržavati viseći pokazivač, ali to niko neće primjetiti dok ne pokuša pristupiti nekom elementu objekta `a`.

U ovom slučaju je neophodna implementacija konstruktora kopije za klasu `Niz`, ukoliko se želi izbjeći formiranje plitkih kopija, već ostvariti pravljenje dubokih.

2. Kada se objekat neke klase prenese kao *parametar po vrijednosti* (znači, bez reference) u neku funkciju:

```
void SaberiNizove(Niz niz1, Niz niz2) {
   .
   .
   .
}

Niz a(5), b(5);
SaberiNizove(a, b);
```

formalni parametri

stvarni parametri

Kao što znamo, parametri koji se prenose po vrijednosti, kopiraju se u formalne parametre funkcije i izmjene nad njima nemaju nikakav uticaj na stvarne parametre.

Međutim, formalni parametri se na kraju izvođenja funkcije uništavaju, a ako su plitke kopije nekog objekta koji sadrži pokazivač na dinamički alociranu memoriju, ta memorija će se osloboditi i za stvarne parametre, pa će stvarni parametri poslije izvođenja te funkcije sadržavati viseće pokazivače.

U ovom slučaju, plitko kopiranje se može jednostavno izbjeći, tako što se formalni parametar napiše kao "konstantna referenca" na svoj tip:

```
void IspisiNizove(const Niz &niz1, const Niz &niz2) {
      ...
}
```

Konstantna referenca govori kompajleru, da ne vrši plitku kopiju stvarnih parametara u formalne, već da formalne zamjeni stvarnim, ali se u kodu funkcije ne smije promijeniti vrijednost nijednog atributa objekta, jer se radi o konstantnoj referenci.

Praktično neće doći uopće do kopiranja, pa ni do plitke kopije.

3. Kada se objekat neke klase vrati kao povratna vrijednost neke funkcije:

```
Niz SaberiNizove(const Niz &niz1, const Niz &niz2) {
   Niz zbir(niz1.broj_elemenata());
   ...
   return zbir;
}

Niz a(5), b(5);
Niz c = SaberiNizove(a, b);
```

Jasno je da se sve što se deklarira unutar neke funkcije, briše na kraju izvođenja te funkcije. Kada neka funkcija vraća neku povratnu vrijednost, ta povratna vrijednost se plitko kopira u varijablu sa lijeve strane poziva funkcije (u našem slučaju varijabla **c**). Poslije toga se izvorni objekat uništava (u našem slučaju varijabla **zbir**). Time funkcija vraća viseći pokazivač. To ne samo dešava kod funkcija koje vraćaju korisnički definirane tipove, koji "se bave" dinamičkom alokacijom unutar svojih atributa.

Plitko kopiranje se u ovom slučaju može izbjeći korištenjem dinamički alociranih objekata. Zbog toga se neznatno mijenja kôd funkcije. Za gore navedeni primjer bi to izgledalo ovako:

```
Niz *SaberiNizove(const Niz &niz1, const Niz &niz2) {
      Niz *zbir = new Niz(niz1.broj_elemenata());
      ...
      return zbir;
}
```

Varijabla `zbir` će se i dalje uništiti nakon završetka izvođenja date funkcije. Međutim, neće se osloboditi dinamički alocirana memorija, jer kompajlera ne zanima šta je u varijabli, on će je prosto izbrisati, nakon što funkcija vrati svoju vrijednost.

Opet, praktično neće doći uopće do kopiranja, pa ni do plitke kopije.

4. Kada se objekat svjesno kopira operatorom dodjele (=):

```
Niz a(5);
Niz b(6);
b = a;
```

U ovom slučaju je neophodno preklopiti operator dodjele.

### 4.7.3. Implementacija konstruktora kopije

Sintaksa konstruktora kopije je:

```
ime_klase(const ime_klase &original);
```

Kao što vidiš, konstruktor kopije neke klase nije ništa drugo nego konstruktor sa jednim parametrom, koji je konstantna referenca na objekat te klase.

Ime parametra (`original`) je proizvoljno. Parametar predstavlja izvorni objekat, tj. objekat koji se pokušava/treba kopirati.

Primjer za našu klasu `Niz`:

1. Prvo treba prototip konstruktora kopije deklarirati u **`public`** sekciji klase:

```
Niz(const Niz &orig_niz);
```

2. Zatim treba kopirati sve što se nalazi u originalnom objektu:

```
Niz::Niz(const Niz &orig_niz) {
      // atribut po atribut
      br_el = orig_niz.br_el;
      // element po element
      p_niz = new int[br_el];
      for(int i(0); i < br_el; i++)
      p_niz[i] = orig_niz.p_niz[i];
}
```

Konstruktor kopije se poziva umjesto bilo kog drugog konstruktora klase datog objekta, tako da su svi atributi "prazni" (tačnije *neinicijalizirani*) i treba ih kopirati iz originalnog objekta.

Vidićeš, da se kôd preklopljenog operatora dodjele ne razlikuje previše od koda konstruktora kopije neke klase.

### 4.7.4. Implementacija preklopljenog operatora dodjele

Sintaksa preklopljenog operatora dodjele je:

```
ime_klase &operator =(const ime_klase &operand);
```

Ime parametra (`operand`) je proizvoljno.

Preklopljen operator dodjele neke klase je operatorska funkcija sa jednim parametrom, koji je konstantna referenca na objekat te klase, a njena povratna vrijednost je referenca na kopiju objekta, navedenog kao parametar.

Analogno, kao i kod konstruktora kopije, slijedi primjer za našu klasu `Niz`:

1. Prvo treba prototip preklopljenog operatora dodjele deklarirati u **`public`** sekciji klase:

```
Niz &operator =(const Niz &orig_niz);
```

2. Zatim treba kopirati sve što se nalazi u originalnom objektu:

```
Niz& Niz::operator =(const Niz &orig_niz) {
        // atribut po atribut
        br_el = orig_niz.br_el;
        // u slučaju da je već nešto bilo alocirano
        delete[] p_niz;
        // element po element
        p_niz = new int[br_el];
        for(int i(0); i < br_el; i++)
              p_niz[i] = orig_niz.p_niz[i];
        // na kraju se vraća referenca objekta na samog sebe
        return *this;
}
```

Kao što vidiš, kôd preklopljenog operatora dodjele se razlikuje u dvije "stvarke" od koda konstruktora kopije:

- prije kopiranja alociranog dijela, mora se obavezno osloboditi prethodno alocirana memorija, jer inače dolazi do gubljenja adrese prethodno alocirane memorije i time do formiranja viseće reference

- na kraju metode se obavezno vraća referenca objekta na samom sebe. To se čini pomoću pokazivača **`this`**, tako što se on dereferencira (iz predmeta "Osnovi računarstva" smo učili da se stavljanjem zvjezdice "*" ispred nekog pokazivača vraća ne adresa, već ono na šta on pokazuje)

### 4.7.5. Zabrana kopiranja objekata neke klase

Još mi je samo ostalo da objasnim, kako ćeš korisniku tvoje klase jednostavno zabraniti kopiranje objekata te klase (pri inicijalizaciji, prenos parametra po vrijednosti, vraćanje kao rezultat funkcije i pomoću operatora dodjele).

To bi te oslobodilo implementacije konstruktora kopije i preklopljenog operatora dodjele.

Tehnika je vrlo jednostavna. Samo se deklariraju prototipovi te dvije metode u **`private`** sekciji klase. Jer time se kompajleru onemogućava pozivanje tih dviju metoda, pa on javlja grešku pri upotrebe objekata te klase na jedan od četiri načina, navedena u oblasti *4.7.2.* ove skripte. Njihova implementacija nije tada više potrebna.

### 4.7.6. Konačna verzija klase `Niz`

Nakon što smo se upoznali sa konstruktorom kopije i preklopljenim operatorom dodjele, kao i njihovom implementacijom, naša klasa `Niz` konačno izgleda ovako:

```
class Niz {
      int *p_niz;
      int br_el;
public:
      Niz(int BrojElemenata)
      {
            br_el = BrojElemenata;
            p_niz = new int[br_el];
      }
      ~Niz() {delete[] p_niz;}
      Niz(const Niz &orig_niz);
      Niz &operator =(const Niz &orig_niz);
      int& element(int indeks) {return p_niz[indeks];}
      int broj_elemenata() const {return br_el;}
};

Niz::Niz(const Niz &orig_niz)
{
      br_el = orig_niz.br_el;
      p_niz = new int[br_el];
      for(int i(0); i < br_el; i++)
            p_niz[i] = orig_niz.p_niz[i];
}

Niz& Niz::operator =(const Niz &orig_niz)
{
      br_el = orig_niz.br_el;
      delete[] p_niz;
      p_niz = new int[br_el];
      for(int i(0); i < br_el; i++)
            p_niz[i] = orig_niz.p_niz[i];
      return *this;
}
```

## 4.8. Preklapanje operatora

Preklapanje operatora (preopterećevanje ili *overloading*) nije ništa drugo nego preklapanje funkcija.

Programski jezik C++ ima neke ugrađene radnje, koje se preklapanjem mogu prebaciti na odgovornost programera. Kada kompajler naiđe na određene naredbe i tehnike, ukoliko ta radnja nije preklopljena od strane programera, kompajler to pokušava na svoj način riješiti.

Zamisli da imaš robota i šalješ ga u trgovinu da kupi hljeb, a nisi mu objasnio put do trgovine. Preklapanje nije ništa drugo nego "upućivanje" kompajlera na pravi put.

U programskom jeziku C++ postoji nešto što se zove *operatorska funkcija* i ona je uvijek oblika "`operator @()`", gdje je `@` operator koji ta operatorska funkcija obrađuje.

Operatorske funkcije se mogu preklapati na dva načina: kao prijateljske funkcije klase ili kao metode klase.

Pored toga povratna vrijednost i broj parametara operatorske funkcije zavisi od tipa operatora: *unarni* i *binarni*.

Unarni operatori se nalaze neposredno sa lijeve strane svog operanda: "*@operand*". Oni koji se mogu preklopiti u programskom jeziku C++ su:

```
+    -    !    ~    &    *    ++    --
```

Binarni operatori se nalaze uvijek između dva operanda: "*operand1 @ operand2*". Oni koji se mogu preklopiti u programskom jeziku C++ su:

```
+    -    *    /    %    <    >    =    |    &    ^    ,
+=   -=   *=   /=   %=   <=   >=   ==   |=   &=   ^=   ||
&&   <<   <<=  >>   >>=  ->*  !=
```

Zatim se mogu preklopiti i specijalni operatori:

```
[]    ()    ->    new    new[]    delete    delete[]
```

Postupak za preklapanje svake od te tri kategorije operatora je različit. Međutim, operatori "++" i "--" mogu biti prefiksi (s lijeve strane operanda) i sufiksi (s desne strane operanda), pa se oni preklapaju na svoj određeni način.

U slijedećim podnaslovima će se preklapati operatori za banalnu klasu, koja simulira realan broj:

```
class Realan {
      double broj;
public:
      Realan(double Broj) {broj = Broj;} // konstruktor
      double Vrati() const {return broj;}
      void Promijeni(double NoviBroj) {broj = NoviBroj;}
};
```

### 4.8.1. Preklapanje unarnih operatora

Recimo, želimo da preklopimo unarni operator "–" za ovu klasu, tako da možemo definirati suprotan element, grupe realnih brojeva.

```
Realan operator -(const Realan &operand) {
        return Realan(-operand.broj);
}
```

Budući da ova funkcija pristupa skrivenom atributu "broj" objekta operand, moramo je proglasiti prijateljskom funkcijom klase Realan:

```
class Realan {
        double broj;
public:
        Realan(double Broj) {broj = Broj;} // konstruktor
        double Vrati() const {return broj;}
        void Promijeni(double NoviBroj) {broj = NoviBroj;}
        friend Realan operator -(const Realan &operand);
};
```

Kompajler će sada izraz "b = -a" interpretirati (prevesti) kao "b = **operator** -(a)".

Ukoliko bismo htjeli isti operator preklopiti, ne kao prijateljsku funkciju, već kao metodu klase, kôd bi se znatno izmijenio:

```
class Realan {
        double broj;
public:
        Realan(double Broj) {broj = Broj;} // konstruktor
        double Vrati() const {return broj;}
        void Promijeni(double NoviBroj) {broj = NoviBroj;}
        Realan operator -() {
                return Realan(-(this->broj));
        }
};
```

*Napomena*: moglo je i bez klučne riječi **this** ispred:

```
Realan operator -() {
        return Realan(-broj);
}
```

Time sa samo htio naznačit da se radi o trenutnom objektu nad kojim se poziva metoda. Kao što vidiš, nema više onog parametra koji predstavlja operand, već je operand trenutni objekat nad kojim se poziva metoda.

Jer se sada izraz "b = -a" interpretira kao "b = a.**operator** -()".

### 4.8.2. Preklapanje binarnih operatora

Preklapanje binarnih operatora se isto može ostvariti na dva načina: preko prijateljskih funkcija i preko metoda.

Također, mogu se koristiti već preklopljeni operatori za preklapanje novih, jer neki imaju sličnosti u računu, ali programer ima i slobodu, da svaki operator neovisno od drugih preklapa.

Recimo, za našu klasu Realan hoćemo da preklopimo birarno sabiranje, oduzimanje, množenje i dijeljenje:

```
class Realan {
      double broj;
public:
      Realan(double Broj) {broj = Broj;}
      double Vrati() const {return broj;}
      void Promijeni(double NoviBroj) {broj = NoviBroj;}
      Realan operator -() {return Realan(-(this->broj));}

      friend Realan operator +(     const Realan &operand1,
                                    const Realan &operand2);

      friend Realan operator -(     const Realan &operand1,
                                    const Realan &operand2);

      friend Realan operator *(     const Realan &operand1,
                                    const Realan &operand2);

      friend Realan operator /(     const Realan &operand1,
                                    const Realan &operand2);
};

Realan operator +(const Realan &operand1, const Realan &operand2) {
      return Realan(operand1.broj + operand2.broj);
}

Realan operator -(const Realan &operand1, const Realan &operand2) {
      return Realan(operand1.broj - operand2.broj);
}

Realan operator *(const Realan &operand1, const Realan &operand2) {
      return Realan(operand1.broj * operand2.broj);
}

Realan operator /(const Realan &operand1, const Realan &operand2) {
      return Realan(operand1.broj / operand2.broj);
}
```

Iz ovoga se dâ zaključit, da su operatorske funkcije binarnih operatora funkcije koje primaju dva parametra.

Zato se izraz "`c = a @ b`" interpretira (prevodi) kao "`c = operator @(a, b)`".

Ukoliko bi htjeli napisati gore navedene funkcije kao metode preklopljenih operatora u klasi, to bi izgledalo ovako:

```
class Realan {
        double broj;
public:
        Realan(double Broj) {broj = Broj;}
        double Vrati() const {return broj;}
        void Promijeni(double NoviBroj) {broj = NoviBroj;}
        Realan operator -() {return Realan(-broj);}

        Realan operator +(const Realan &operand2);
        Realan operator -(const Realan &operand2);
        Realan operator *(const Realan &operand2);
        Realan operator /(const Realan &operand2);
};

Realan Realan::operator +(const Realan &operand2) {
        return Realan(broj + operand2.broj);
}

Realan Realan::operator -(const Realan &operand2) {
        return Realan(broj - operand2.broj);
}

Realan Realan::operator *(const Realan &operand2) {
        return Realan(broj * operand2.broj);
}

Realan Realan::operator /(const Realan &operand2) {
        return Realan(broj / operand2.broj);
}
```

Budući da se pri, na ovaj način, preklopljenim operatorskim funkcijama izraz "`c = a @ b`" sad interpretira kao "`c = a.operator @(b)`", parametar za prvi (lijevi) operand nam nije više potreban, jer je on ustvari objekat nad kojim se poziva metoda preklopljenog operatora.

Za definiciju množenja objekta tipa `Realan` i cijelog broja, također se preklapa operator množenja, međutim sa dvije strane:

```
class Realan {
        double broj;
public:
        ...
        ...
        // b = 3 * a
        friend Realan operator *(int operand1, const Realan &operand2);
        // b = a * 3
        friend Realan operator *(const Realan &operand1, int operand2);
};
```

```
Realan operator *(int br, const Realan &operand) {
      return Realan(br * operand.broj);
}

Realan operator *(const Realan &operand, int br) {
      return Realan(br * operand.broj);
}
```

Budući da smo u prvoj funkciji već definirali množenje cijelog broja sa objektom tipa `Realan`, sa lijeve strane i budući da je isti postupak računa za množenje sa desne strane, možemo se koristiti tehnikom *umetanja* funkcije u funkciju, pomoću ključne riječi **inline**:

```
Realan operator *(int br, const Realan &operand) {
      return Realan(br * operand.broj);
}

inline Realan operator *(const Realan &operand, int br) {
      return br * operand;
}
```

Ova ključna riječ će naznačiti kompajleru da ne izvodi funkciju u funkciji, već da zamijeni poziv funkcije sa kôdom funkcije. Tako se kôd brže izvodi.

***Napomena***: za ove dvije funkcije ne postoje njene sestre blizankinje u vidu metoda preklopljenih operatora. Budući da bi metoda primala samo jedan parametar umjesto dva, kompajleru ne bi bilo moguće naznačiti da li se radi o "množenju s lijeve" ili "množenju s desne" strane.

### 4.8.3. Preklapanje operatora kod kojih prvi operand nije konstanta

Iz naslova se ne može baš ništa zaključiti. Međutim, radi se o operatorima koji mijenjaju atribute svog prvog operanda. To znači da se operand koji se mijenja ne može više deklarirati kao konstantna referenca (**const** ***ime_tipa*** **&**operand), već kao nekonstantna ili obična referenca.

Takvi su operatori npr. `+=, -=, *=, /=, &=, |=` itd.

***Napomena***: Logički operatori (`==, !=, &&, ||`) se ne ubrajaju u ovu vrstu operatora. Naprotiv, njihovi operandi moraju biti konstantne reference, a njihova povratna vrijednost je obično tipa **bool**.

Kao prijateljske funkcije:

```
// deklaracija u klasi Realan
friend Realan &operator +=(Realan &operand1, const Realan &operand2);
// --------------------------

// definicija van klase
Realan &operator +=(Realan &operand1, const Realan &operand2) {
      operand1.broj += operand2.broj;
      return operand1;
}
```

Kao što vidiš, povratni tip operatorske funkcije je sada referenca i prvi parametar je također referenca.

Budući da smo za našu klasu već prethodno definirali binarno sabiranje, možemo operator preklopiti i na sljedeći način:

```
Realan &operator +=(Realan &operand1, const Realan &operand2) {
      return operand1 = operand1 + operand2;
}
```

Izraz "`b += a`" interpretira kao "`b = operator +=(a)`".

Ovakvi operatori se mogu također kao *metode prekljopljenih operatora* napisati:

```
// prototip u klasi
...
Realan &operator +=(const Realan &operand2);
...

// implementacija van klase
Realan& Realan::operator +=(const Realan &operand2) {
      broj += operand2.broj;
      return *this;
}
```

Kao što se da primjetiti, operatorska metoda opet nema prvog parametra, a kao povratnu vrijednost vraća dereferenciranih pokazivač **this**.

To je zbog toga što se sada izraz "`b += a`" interpretira kao "`b = b.operator +=(a)`".

Naravno, i ovdje smo se mogli poslužiti prethodno definiranim binarnim sabiranjem:

```
Realan& Realan::operator +=(const Realan &operand2) {
      return *this = *this + operand2;
}
```

### 4.8.4. Preklapanje unarnih operatora "++" i "--"

Budući da za ova dva operatora postoji prefiskna (*prefix*) i sufiksna (*postfix*) verzija, koje se razlikuju po tome što

- prefiskna verzija prvo promijeni svoj operand, pa onda kompajleru predstavi njegovu trenutnu vrijednost
- sufkisna verzija prvo preda kompajleru trenutnu vrijednost svog operanda, pa ga onda promijeni

Primjer, da se malo podsjetiš:

```
int a(2), b(2);
cout << ++a << " " << a;      // ispis:   3 3
cout << b++ << " " << b;      // ispis:   2 3
```

Što se tiče preklopljenih verzija ovih operatora:

- izraz "++a" se interpretira kao "**operator ++**(a)";
- izraz "a++" se interpretira kao "**operator ++**(a, 0)";

- izraz "--a" se interpretira kao "**operator --**(a)";
- izraz "a--" se interpretira kao "**operator --**(a, 0)".

Pošto postoji očita analogija između određenih verzija operatora "++" i operatora "--", ja ću pokazati kako se preklapaju obje verzije operatora "++".

Prvo, preko prijateljske funkcije:

```
...
friend Realan &operator ++(Realan &operand);       // prefiksna verzija
friend Realan operator ++(Realan &operand, int);  // sufiksna verzija
...

Realan &operator ++(Realan &operand) {
      operand.broj++;
      return operand;
}

Realan operator ++(Realan &operand, int) {
      Realan kopija = operand;
      operand.broj++;
      return operand;
}
```

A može i kao metoda preklopljenog operatora:

```
...
Realan &operator ++();         // prefiksna verzija
Realan operator ++(int);       // sufiksna verzija
...

Realan& Realan::operator ++() {
      broj++;
      return *this;
}

Realan Realan::operator ++(int) {
      Realan kopija = *this;
      broj++;
      return kopija;
}
```

Obrati pažnju na to da parametru tipa **int** nismo dali ime, budući da je to u programskom jeziku C++ nepotrebno, ukoliko se parametar ne koristi nigdje u funkciji. Ali taj parametar je opet neophodan, jer bez njega ne bi mogli definirati sufiksnu verziju ova dva operatora.

### 4.8.5. Preklapanje operatora ulaza i izlaza

Definitivno, jedna od najkorisnijih tehnika ovog programskog jezika. Jer, ukoliko se preklope operatori ulaza i izlaza (">>" i "<<"), objekti tvoje klase se mogu pisati u konstrukcijama poput "`cout << b << ...`".

Pri tome se:

- izraz "`cout << a`" interpretira kao "**operator <<**`(cout, a)`"
- izraz "`cin >> a`" interpretira kao "**operator >>**`(cin, a)`"

Pretpostavimo, kako će izgledati funkcija preklapanje jednog od ova dva operatora:

- po interpretaciji ona dva izraza gore, funkcija će sigurno kao prvi parametar primati identifikator "`cout`", koji nije ništa drugo nego objekat klase "`ostream`"
- drugi parametar će biti konstantna referenca na objekat koji se ispisuje
- povratna vrijednost će također biti objekat klase "`ostream`", da bi se omogućilo ulančavanje ispisnih konstrukcija (`cout << a << b << c << ...`)

Sad samo treba skontati, da li će prvi parametar i povratna vrijednosti biti konstantne reference na objekte klase "`ostream`" ili obične reference.

Budući da se izraz "`cout << a << b`" interpretira kao:

```
operator <<(operator <<(cout , a), b)
```

To znači, da i parametar i povratna vrijednost moraju kompajleru omogućiti da mijenja njihovo stanje, tj. moraju biti deklarirani kao obične reference.

***Napomena***: većinu dosadašnjih operatora smo mogli preklapati na dva načina – kao prijateljsku funkciju klase ili kao metodu preklopljenog operatora. Međutim, to nije slučaj sa operatorima ulaza i izlaza. Oni se ***obavezno implementiraju kao prijateljske funkcije***.

Poslije svih ovih zamarajućih rečenica, konačno prelazimo na kôd:

```
...
friend ostream &operator <<(ostream &cout, const Realan &operand);
...

// implementacija
ostream& operator <<(ostream &cout, const Realan &operand) {
	// operacije potrebne za ispis: cout << bla bla bla
	cout << "broj = " << operand.broj;
	// povratna vrijednost metode je prvi parametar
	return cout;
}
```

Isti fazon se dešava i kod preklapanja operatora ulaza, samo što se koristi objekat klase "istream" i što drugi parametar nije više konstantna referenca, već obična referenca:

```
...
friend istream &operator >>(istream &cin, Realan &operand);
...

// implementacija
istream& operator >>(istream &cin, Realan &operand) {
      // operacije potrebne za unos: cin >> bla bla bla
      cin >> operand.broj;
      // povratna vrijednost metode je prvi parametar
      return cin;
}
```

To bi bilo to. Ukoliko si pronašao negdje slučajno neku grešku, sintaksičku, logičku ili gramatičku, nemoj se stidit da mi pošalješ mail.

Ukoliko imaš prijedlog šta bi se moglo izbaciti, dodati ili modificirati, nemoj se stidit...

Do tad, ti želim puno uspjeha na ispitima iz ovog predmeta.

Dinko Hasanbašić